CTESK 2.8:
Документация пользователя

Содержание

1 Назначение CTESK

Набор инструментов CTESK предназначен для автоматизированной разработки тестов для систем, предоставляющих программный интерфейс на языке C. Тестирование ПО с помощью инструмента CTESK основано на технологии UniTESK.

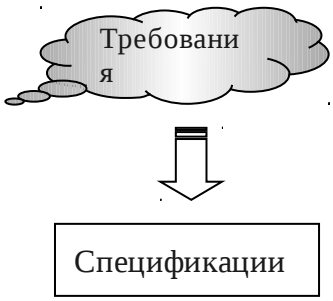
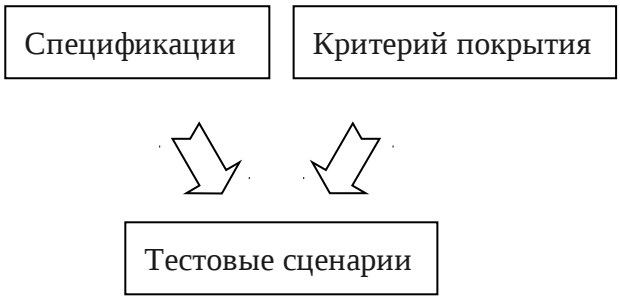
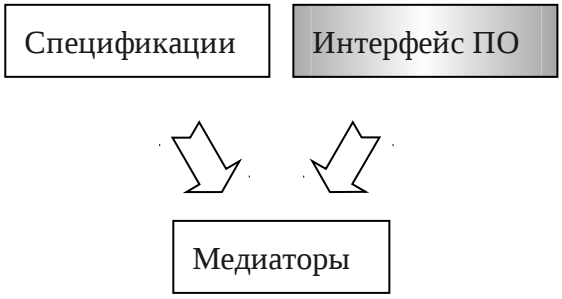
1.1 Технология UniTESK

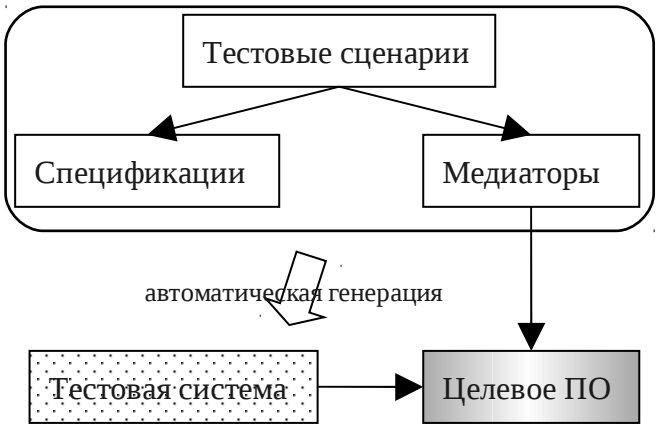
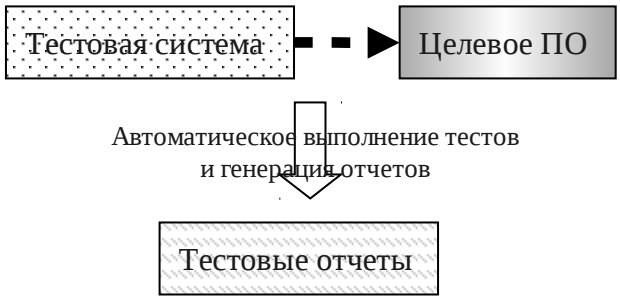
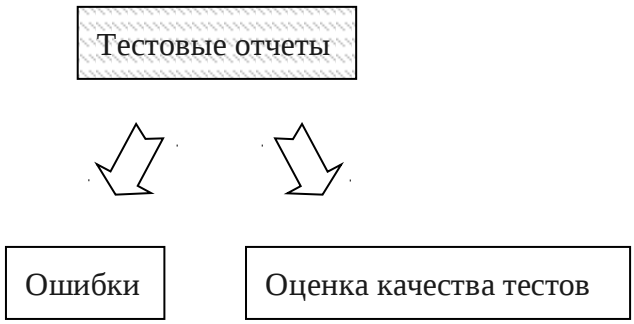
Контроль качества является важной проблемой, стоящей перед разработчиками программного обеспечения. Тестирование является наиболее известным и широко используемым способом оценки и повышения качества. Функциональность и сложность современного программного обеспечения растёт очень быстро, а его жизненный цикл увеличивается. В этих условиях трудоёмкость применения традиционных подходов к тестированию растёт, а эффективность падает. Использование технологии UniTESK помогает преодолеть эти проблемы.

Основные характеристики технологии UniTESK таковы:

- Для чёткого определения функциональности ПО создаются *формальные спецификации*. Это может быть сделано как для вновь разрабатываемого ПО, даже до завершения реализации, так и для уже существующего. Таким образом, технология может применяться в задачах прямой и обратной инженерии ПО.
- Тесты разрабатываются на основе формальных спецификаций, а не реализации. Это позволяет проверять соответствие поведения ПО предъявляемым требованиям. Такое тестирование называют методом “чёрного ящика”. Оно позволяет создать набор тестов, не учитывающий особенности конкретной реализации.
- *Критерии тестового покрытия* также строятся на основе формальных спецификаций. Эти критерии позволяют оценить, насколько полно проверено соответствие поведения ПО заданным требованиям.
- Для выбранного критерия тестового покрытия строится *сценарий тестирования*, который нацелен на достижение максимального покрытия по выбранному критерию. Аналогом сценариев тестирования являются широко используемые тестовые скрипты. Однако сценарии тестирования UniTESK позволяют значительно улучшить качество тестирования при тех же усилиях на разработку.
- Формальные спецификации и сценарии тестирования могут быть использованы в неизменном виде для тестирования различных реализаций, даже если их интерфейсы отличаются. Сопряжение реализации и тестов осуществляется с помощью специальных компонентов тестовой системы — *медиаторов*. Такой подход позволяет увеличить степень переиспользования компонентов тестовой системы, облегчая разработку и сопровождение набора тестов.
- Для записи всех компонентов тестовой системы (формальных спецификаций, медиаторов и тестовых сценариев) используется спецификационное расширение языка программирования, который используется для разработки тестируемого ПО. Это значительно облегчает освоение технологии и понимание связи теста с тестируемой системой.

Действия, которые необходимо выполнить для разработки теста по технологии UniTESK, описаны в следующей таблице:

1. Проанализировав функциональные требования к целевому ПО на основе имеющихся документов или знаний участников проекта, надо записать их в виде формальных спецификаций.	
2. На основе полученных спецификаций сформулировать требования к качеству тестирования — какой уровень покрытия по какому критерию будет считаться достаточным, чтобы прекратить тестирование.	
3. Разработать набор тестовых сценариев, обеспечивающий достижение нужного уровня покрытия. Сценарии разрабатываются на основе спецификаций и не привязаны к конкретной реализации целевого ПО или его конкретной версии.	
4. Для привязки полученных тестов к конкретной реализации целевой системы надо разработать набор медиаторов. Для этого нужно знать точный интерфейс выбранной реализации, хотя сама она еще может быть не готова к тестированию.	

5. Для того, чтобы получить готовые тесты, нужно оттранслировать спецификации, медиаторы и сценарии с расширения языка программирования в целостную тестовую систему на самом этом языке.	
6. После трансляции становится возможным выполнение тестов. При выполнении тестов могут быть выявлены несоответствия между тестовой системой и целевым ПО. Следует выяснить, чем вызвано каждое несоответствие — ошибкой в целевой системе, или в компонентах тестовой системы. После устранения дефектов в спецификациях, сценариях и медиаторах нужно выполнить все тесты “начисто” и получить их результаты в виде отчетов для дальнейшего анализа.	
7. Провести анализ результатов, чтобы определить, какие ошибки обнаружены, достигнут ли нужный уровень покрытия или надо разрабатывать дополнительные сценарии.	

Этапы разработки тестовых сценариев и медиаторов не зависят друг от друга, поэтому шаги 3 и 4 могут выполняться в любом порядке или даже одновременно.

1.2 Реализация UniTESK в CTESK

CTESK реализует UniTESK на платформе языка программирования C.

В CTESK для разработки тестов используется язык *SeC* — специально разработанное спецификационное расширение языка программирования C (Specification Extension of C). SeC расширяет язык C специальными конструкциями, которые позволяют компактным и удобным образом описывать требования к тестируемой системе и другие компоненты тестовой системы. Это делает разработку тестов максимально удобной, а также позволяет сократить затраты на обучение специалистов, уже знакомых с языком программирования C. Также SeC позволяет определять полностью независимые от реализации спецификации и сценарии, что дает возможность их повторного использования.

Набор инструментов CTESK включает транслятор из SeC в C, библиотеку поддержки тестовой системы, библиотеку спецификационных типов и генераторы тестовых отчетов.

Транслятор из SeC в C позволяет генерировать компоненты тестов из спецификаций, медиаторов и тестовых сценариев.

Библиотека поддержки тестовой системы предоставляет *обходчик* — реализацию на языке C алгоритмов построения тестовой последовательности — и обеспечивает трассировку выполнения тестов.

Библиотека спецификационных типов поддерживает типы, интегрированные со стандартными функциями создания, копирования, сравнения и уничтожения значений этих типов, и содержит набор уже определенных спецификационных типов.

Генераторы текстовых и графических тестовых отчетов дают возможность генерации удобных для анализа представлений трассы выполнения теста.

2 Руководство пользователя

В этой главе сделан обзор особенностей использования языка SeC.

- В разделе «*Общие сведения*» приводятся отличия SeC от C, описываются расширяющие C понятия и конструкции.
- В разделе «*Спецификации*» рассматривается способ формального описания требований к тестируемой системе в виде *предусловий*, *постусловий* и *ограничений доступа* в *спецификационных функциях* и *отложенных реакциях*. Описывается способ задания различных *критериев покрытия*. Также вводится концепция *допустимых* и *спецификационных типов*, в деталях рассматривается работа с существующими спецификационными типами и правила создания новых типов данных. Описывается механизм *инвариантов*, накладываемых на типы данных и переменные.
- В разделе «*Медиаторы*» объясняется механизм связывания спецификации с реализацией тестируемой системы в виде *медиаторных функций*, осуществляющих *воздействие* и *синхронизацию состояний*.
- В разделе «*Тестовые сценарии*» описано построение *тестового сценария*, объединяющего набор *сценарных функций* для итерации параметров, механизм построения теста, функцию вычисления *сценарного состояния*, способ инициализации и завершения работы тестовой и тестируемой систем.

3 Общие сведения

SeC полностью поддерживает ANSI C. Дополнительно вводятся *спецификационные типы*, *типы* и *переменные с инвариантами*, *покрытия* и четыре вида функций: *спецификационные*, *реакции*, *медиаторные* и *сценарные*. Эти типы, инварианты и функции определяются в спецификационных файлах с расширением `sec`. Спецификационные заголовочные файлы, содержащие декларации спецификационных типов и функций должны находится в файлах с расширением `seh`.

Спецификационные заголовочные файлы включаются в спецификационные файлы при помощи директивы препроцессора C `#include`. Спецификационные файлы могут содержать и обычные функции C, требуемые для различных вспомогательных целей. При необходимости использования типов данных, констант, переменных или функций включаются обычные заголовочные файлы языка C.

Для удобства записи и чтения логических выражений в язык SeC дополнительно введен оператор импликации `=>`, являющийся бинарным инфиксным оператором, приоритет которого меньше приоритета оператора дизьюнкции `||`, но больше приоритета условного оператора `?:`. Выражение `x => y` эквивалентно выражению `!x || y`, и при его вычислении, также как при вычислении других логических операторов, действуют правила короткой логики. Оператор импликации ассоциативен слева направо, то есть выражение `x => y => z` эквивалентно выражению `(x => y) => z`.

4 Спецификации

Спецификация представляет собой формальное описание требований к тестируемой системе. В тестируемой системе выделяются интерфейсные функции и данные, определяющие состояние этой системы.

В спецификации поведение интерфейсных функций описывается спецификационными функциями, а состояние системы моделируется глобальными переменными состояния. Требования к тестируемой системе формулируются как ограничения на поведение интерфейсных функций (в виде предусловий и постусловий в спецификационных функциях) и на значения данных (в виде инвариантов типов и переменных состояния).

Привязка спецификационных функций и модельных данных к функциям и данным реализации тестируемой системы осуществляется при помощи медиаторов.

Кроме того, спецификации могут содержать критерии покрытия, позволяющие оценить полноту тестирования.

В случае тестирования систем с отложенными реакциями, их поведение при ответе на внешние воздействия состоит из непосредственных и отложенных реакций. Первые описываются обычными спецификационными функциями, а для описания последних в спецификацию добавляются отложенные реакции. Привязка отложенных реакций к тестируемой системе осуществляется с помощью медиаторов и сборщиков реакций. В отличие от спецификационных функций, для отложенных реакций не могут быть заданы критерии покрытия.

4.1 Спецификационные функции

> Спецификационные функции служат для описания поведения интерфейсных функций тестируемой системы. В общем случае спецификационная функция определяет поведение тестируемой системы при определенном воздействии на нее через некоторую часть интерфейса.

Спецификационные функции описывают поведение в форме ограничений доступа к данным, предусловий, локальных критериев покрытия и постусловий.

Декларация спецификационной функции состоит из ключевого слова `specification`, сигнатуры функции (в обычном смысле языка C) и, возможно, ограничений доступа.

```
specification double sqrt_spec(double x);
```

SeC предусматривает три конструкции, предназначенные для включения в тело спецификационной функции: предусловие, постусловие и локальное определение покрытия. Предусловие и критерии покрытия могут отсутствовать, тогда как постусловие должно присутствовать обязательно.

Предусловие проверяет применимость функции к данному набору значений параметров и переменных состояния. Локальные покрытия, которые могут использоваться в спецификационной функции наряду с глобальными, разбивают поведение системы, описанное данной спецификационной, на ветви функциональности. И код предусловия, и код функций вычисления локальных покрытий выполняется в пре-состоянии, т. е. до взаимодействия с тестируемой системой. Значения выражений и переменных в этот момент называются пре-значениями.

Перед вычислением постусловия осуществляется взаимодействие с тестируемой системой с помощью вызова медиатора. Постусловие проверяет соответствие полученного результата ожидаемому. Оно выполняется в пост-состоянии, т. е. после взаимодействия, и имеет дело с пост-значениями выражений.

```
specification double sqrt_spec(double x) {
  pre { ... }
  coverage C { ... }
  post { ... }
}
```

В общем случае между описанными блоками может использоваться дополнительный код. Дополнительный код не должен иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри блока кода, должна освобождаться либо в нем же, либо в парном ему блоке. Этот код обычно используется для определения переменных, общих для предусловия, покрытий и постусловия. Полное описание в семантических требованиях к спецификационным функциям.

Спецификационные функции вызываются, как правило, в сценарных функциях. Вызов спецификационной функции состоит в проверке инвариантов в соответствии с ограничениями доступа, проверке предусловия, вычислении покрытых ветвей в соответствии с критериями покрытия, осуществлении тестового воздействия и синхронизации модельного и реализационного состояний в медиаторе, вторичной проверке инвариантов и проверке постусловия. Спецификационная функция возвращает значение, вычисленное в медиаторе (если она не объявлена как `void`).

4.2 Отложенные реакции

Отложенные реакции служат для описания поведения тестируемой системы при отложенном реагировании на внешние воздействия. Отложенные реакции описывают поведение в форме ограничений доступа к данным, предусловий и постусловий. В отличие от спецификационных функций, в отложенных реакциях не используются критерии покрытия.

Декларация отложенной реакции состоит из ключевого слова `reaction`, сигнатуры функции (в обычном смысле языка C) и, возможно, ограничений доступа.

```
reaction String* recv_spec(void);
```

Отложенная реакция:

- никогда не имеет параметров,
- должна возвращать спецификационную ссылку.

Тело отложенной реакции состоит из двух частей: предусловия (может быть опущено) и постусловия (обязательно одно).

Предусловие проверяет возможность возникновения реакции в данном состоянии. Предусловие выполняется в пре-состоянии и имеет доступ только к пре-значениям выражений.

Постусловие проверяет соответствие результата, полученного при возникновении реакции, ожидаемому. Оно выполняется в пост-состоянии после возникновения реакции и имеет дело с пост-значениями выражений.

```
reaction String* recv_spec (void) {
  pre { ... }
  post { ... }
}
```

В общем случае между описанными блоками может использоваться дополнительный код. Дополнительный код не должен иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри блока кода, должна освобождаться либо в нем же, либо в парном ему блоке. Этот код обычно используется для определения переменных, общих для предусловия и постусловия. Полное описание находится в семантическом требовании к отложенным реакциям.

Отложенная реакция никогда не вызывается явно, поскольку инициируется тестируемой системой.

4.3 Ограничения доступа

Ограничения доступа указывают способ использования в спецификационных функциях и отложенных реакциях глобальных переменных и параметров, а так же выражений с ними. Поддерживаются три вида ограничений доступа: чтение (`reads`), запись (`writes`) и изменение (`updates`).

Ограничения записываются после сигнатуры спецификационной функции или отложенной реакции в виде списка выражений через запятую, помеченного одним из ключевых слов `reads`, `writes` или `updates`:

```
specification void root_spec(
  double a,
  double b,
  double c,
  double *x1,
  double *x2)
  reads a, b, c
  writes *x1, *x2;
```

Ограничение доступа `reads` для некоторого выражения означает, что значение этого выражения не изменяется в результате взаимодействия, т. е. пре-значение этого выражения совпадает с пост-значением. Для таких выражений автоматически проверяются инварианты перед проверкой предусловия, а перед проверкой постусловия проверяется, что значение выражения не изменилось.

Ограничение доступа `writes` для некоторого выражения означает, что пре-значение не используется в спецификационной функции и может быть не определено, а пост-значение вы-рабатывается в результате взаимодействия с тестируемой системой. Выражения с доступом `writes` нельзя использовать в операторе взятия пре-значения `@` (см. Постусловие). Для таких выражений автоматически проверяются инварианты перед проверкой постусловия.

Ограничение доступа `updates` для некоторого выражения означает, что пре-значение этого выражения является входным параметром, от которого может зависеть поведение системы, а пост-значение получается в результате взаимодействия и может не совпадать с пре-значением. Для таких выражений автоматически проверяются инварианты перед проверкой как предусловия, так и постусловия.

Выражениям в ограничениях доступа можно присваивать идентификатор, называемый псевдонимом. В дальнейшем псевдоним можно использовать для доступа к значению выражения, в том числе к псевдониму можно применять оператор `@`:

```
specification void deposit_spec(AccountModel *acct, int sum)
  reads   sum
  updates balance = acct->balance
{
  ...
  post {
    return balance == @balance + sum;
  }
}
```

4.4 Предусловие

При взаимодействии, описываемом спецификационной функцией, поведение тестируемой системы в некоторых ситуациях может быть не определено. Для выделения таких ситуаций и служит предусловие. Во время тестирования предусловие проверяется каждый раз при обращении к спецификационной функции. Нарушение предусловия означает, что тест составлен некорректно.

В случае отложенной реакции, предусловие определяет, возможно ли возникновение такой реакции в данном состоянии. Во время тестирования предусловие проверяется каждый раз при возникновении реакции. При нарушении предусловия фиксируется несоответствие между поведением целевой системы и ее спецификацией.

Предусловие записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом `pre`. Эти инструкции представляют собой тело функции, имеющей те же параметры, что и спецификационная функция или отложенная реакция, и возвращающей результат типа `bool`, показывающий, выполнено ли предусловие.

```
specification double sqrt_spec(double x) {
  pre {
    return x >= 0.0;
  }
  ...
}
```

Если поведение системы определено при любых значениях входных параметров и в любом модельном состоянии (или появление реакции допустимо в любом состоянии), предусловие можно опустить.

Предусловие вычисляется до взаимодействия с тестируемой системой. Значения выражений в этот момент называются *пре-значениями*.

Перед проверкой предусловия автоматически проверяются инварианты параметров спецификационной функции и выражений, описанных в ограничениях доступа как `reads` и `updates`.

Предусловие не должно иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри предусловия, должна освобождаться в нем же.

Предусловие спецификационной функции может быть явно вычислено с помощью конструкции `pre` (используется, как правило, в сценарных функциях для того, чтобы не вызывать спецификационную функцию с некорректными параметрами):

```
if (pre sqrt_spec(-1.0)) ...
```

4.5 Постусловие

Постусловие спецификационной функции служит для описания ограничений, которым должны удовлетворять результаты работы тестируемой системы при взаимодействии, описываемом спецификационной функцией. Во время тестирования постусловие проверяется каждый раз после осуществления взаимодействия. Если постусловие оказывается нарушенным, фиксируется несоответствие поведения целевой системы ее спецификации.

Постусловие отложенной реакции описывает ограничения, которым должны удовлетворять результаты взаимодействия после возникновения реакции. Если постусловие оказывается нарушенным, фиксируется несоответствие поведения целевой системы ее спецификации.

Постусловие записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом `post`. Эти инструкции представляют собой тело функции, имеющей те же параметры, что и спецификационная функция, и возвращающей результат типа `bool`. Значение `true` показывает, что поведение тестируемой системы соответствует ожидаемому (постусловие выполнено), а значение `false` показывает, что поведение отличается от ожидаемого.

В спецификационной функции и отложенной реакции всегда должно быть ровно одно постусловие.

Для доступа к значению, возвращенному медиатором спецификационной функции, используется идентификатор спецификационной функции (если спецификационная функция не объявлена как `void`). Аналогично, для доступа к значению реакции, зафиксированному при регистрации, используется идентификатор отложенной реакции.

```
specification double sqrt_spec(double x) {
  ...
  post {
    if (x == 0.0) return (sqrt_spec == 0.0);
    return    sqrt_spec >= 0.0
           && fabs( (sqrt_spec*sqrt_spec - x) / x ) < EPS;
  }
}
```

Постусловие вычисляется после взаимодействия с тестируемой системой. Значения выражений после воздействия называются пост-значениями.

Перед проверкой постусловия автоматически проверяются инварианты параметров спецификационной функции и выражений, описанных в ограничениях доступа как `writes` и `updates`, а также инвариант возвращаемого значения.

Для доступа из постусловия к пре-значениям используется унарный оператор `@`. Выражение под этим оператором должно иметь допустимый тип и должно быть вычислимо непосредственно перед ключевым словом `post` (поскольку значения таких выражений автоматически сохраняются непосредственно перед осуществлением взаимодействия с тестируемой системой). Запрещено использовать оператор `@` для выражений, имеющих доступ `writes`.

```
/* List — библиотечный спецификационный тип*/
specification void f( List* l ) {
  int j;
  post {
```

```
      int i;
      Object* pre_item;
      for( i = 0, j = 0
         ; i < @size_List(l) /* допустимо */
         ; i++, j++
         )
      {
        /* недопустимо: i не определена вне постусловия */
        pre_item = @get_List(l, i);
        /* недопустимо: j имеет единственное неизвестное значение
                        вне постусловия */
        pre_item = @get_List(l, j);
        pre_item = get_List(@l, j); /* допустимо*/
        if(equals(get_List(l, i), pre_item))
          return  false;
      }
      return  true;
    }
  }
```

Если требуется обеспечить доступ к пре-значению выражения, тип которого не является допустимым, следует вручную сохранить значение этого выражения в локальной переменной до блока `post` (возможно, лучшее решение состоит в использовании подходящего или определении нового спецификационного типа):

```
  {
    char *s = "...", *pre_s;
    ...
    pre_s = strdup(s);
    post {
      return !strcmp(s, pre_s);
    }
    free(pre_s);
  }
```

Постусловие не должно иметь побочных эффектов: в нем не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри постусловия, должна освобождаться там же.

4.6 Типы данных

Язык SeC полностью поддерживает типы данных языка C. Кроме того, в язык SeC введены дополнительные типы и их виды:

1. Булевский тип `bool` для представления логических значений и константы `true` и `false`.
2. Спецификационные типы, объединяющие типы языка C с базовыми операциями работы с данными этих типов: создание, копирование, сравнение, уничтожение.
3. Типы с инвариантами или подтипы — типы данных, множества значений которых являются подмножествами значений других типов данных, которые являются для них надтипами. Подмножество значений подтипа задается при помощи ограничений, описанных в инварианте типа.

Типы, которые допускаются для аргументов и возвращаемых значений спецификационных функций, отложенных реакций и медиаторных функций, для итерационных переменных и переменных сценарного состояния сценарных функций, а так же для глобальных переменных, используемых в вышеперечисленных функциях, называются *допустимыми типами*.

К допустимым типам предъявляются требования, которые не обеспечиваются многими типами языка C. Например, невозможно автоматически вычислить размер массива по указателю на него, соответственно, невозможно скопировать значение типа. Для преодоления этих ограничений и служат *спецификационные типы*. Также спецификационные типы необходимы при использовании библиотеки поддержки тестовой системы CTESK (см. Библиотека поддержки тестовой системы CTESK)

4.7 Допустимые типы

Для аргументов и возвращаемых значений спецификационных функций, отложенных реакций и медиаторных функций, для итерационных переменных и переменных сценарного состояния сценарных функций, а так же для глобальных переменных, используемых в вышеперечисленных функциях, допускаются только следующие типы:

1. Арифметические типы (`int`, `char`, `double`...).
2. Перечислимые типы (`enum`), множество значений которых, в отличие от C, ограничено их константами и не может содержать произвольное целочисленное значение.
3. Типизированный указатель. Указатель на любой допустимый тип. При этом предполагается, что указатель либо нулевой, либо указывает на одно единственное значение соответствующего типа. Структура указателей должна быть древовидной, т. е. не должно быть неориентированных циклов по указателям.
4. Нетипизированный указатель (`void*`). Значения такого типа интерпретируются просто как адрес некоторой ячейки памяти.
5. Функциональный указатель. Значения такого типа интерпретируются просто как адрес некоторой функции.
6. Структурный тип. Структура должна иметь завершенный тип, то есть описание ее тела должно быть видно в месте ее использования. Поля структуры должны иметь допустимые типы.
7. Массив фиксированной длины. Тип элементов массива должен быть допустим.

Те же ограничения накладываются на типы, которые являются базовыми в определениях типов с инвариантами, и на типы переменных с инвариантами. Эти ограничения позволяют тестовой системе автоматически управлять данными таких типов: размещать и освобождать память, копировать и сравнивать значения.

Далее типы, допустимые в вышеперечисленных случаях, будут называться *допустимыми типами SeC*.

4.8 Спецификационные типы

> К допустимым типам предъявляются требования, которые не обеспечиваются многими типами языка C. Например, невозможно автоматически вычислить размер массива по указателю на него, соответственно, невозможно скопировать значение типа. Для преодоления этих ограничений и служат *спецификационные типы*. Также спецификационные типы необходимы при использовании библиотеки поддержки тестовой системы CTESK (см. Библиотека поддержки тестовой системы CTESK).

Спецификационный тип объединяет тип языка C с базовыми операциями работы с данными этого типа: создание, копирование, сравнение, уничтожение.

Значения спецификационных типов всегда размещаются в динамической памяти и доступны только через спецификационные ссылки — указатели соответствующих типов, которые должны быть либо нулевыми, либо указывать на выделенную и инициализированную память. То есть не допускаются объявления переменных и параметров самих спецификационных типов (не ссылок на них), а также непосредственное использование спецификационных типов при определении структур, объединений и массивов.

Если спецификационная ссылка при объявлении не инициализируется явно, то ей автоматически присваивается нулевое значение.

Спецификационные ссылки разыменовываются так же как указатели в C — при помощи операторов разыменования `*` и `->`. Результатом разыменования ссылки является l-value, тип которого совпадает с типом, на основе которого определен спецификационный тип (то есть с базовым типом в определении спецификационного типа), или типом поля спецификационной структуры. (см. Создание новых спецификационных типов).

Не допускается разыменование нулевых ссылок (приводит к ошибке во время исполнения).

Не допускается разыменование спецификационных ссылок, возвращенных вызовом функций, без присваивания возвращенных ссылок в переменные (приводит к утечке памяти или ошибке во время выполнения).

```
/* List – библиотечный спецификационный тип */
List* l = create_List(&type_Integer);
/* Integer – библиотечный спецификационный тип */
Integer* spec_i;
int i;

append_List(l,create_Integer(1));
i = *get_List(l,0); /* не допустимо */
spec_i = get_List(l,0);
i = *spec_i; /* i равно 1 */

specification typedef struct {int a; int b;} IntPair ={};

IntPair* ip = create(&type_IntPair, 1, 1);
int ai = create(&type_IntPair, 1, 1)->a; /* не допустимо */

ai = ip->a; /* ai равно 1 */
```

Не допускается адресная арифметика над ссылками и их индексирование.

Допускается сравнение адресов в ссылках при помощи операторов C `==` и `!=`.

Не допускается сравнение адресов в ссылках, возвращенных вызовом функций, без присваивания возвращенного результата переменной (приводит к утечке памяти).

```
List* l = create_List(&type_Integer);
Integer* spec_i
int i;

append_List(l,create_Integer(1));
spec_i = get_List(l,0);
if (get_List(l,0)!= NULL) /* не допустимо */
if (spec_i != NULL)
  i = *spec_i; /* i равно 1 */
```

В SeC определен встроенный спецификационный тип `Object`, который является неполным спецификационным типом (incomplete specification type). Он является абстрактным базовым типом для всех спецификационных типов. Тип ссылки `Object*` используется по аналогии с `void*`. Ссылка на любой спецификационный тип может быть преобразована к ссылке на `Object` и наоборот. Если при обратном преобразовании тип значения по ссылке не совместим с типом, к которому осуществляется преобразование, то поведение системы не определено.

Для спецификационных ссылок допускается приведение их типов только к указателям на типы `void` и `Object`, а также к спецификационным ссылкам совместимых спецификационных типов. Совместимыми являются спецификационные типы — подтипы одного и того же спецификационного типа (о подтипах см. Инвариант типа).

Управление памятью, на которую ссылаются спецификационные ссылки, автоматизировано при помощи механизма подсчета ссылок с отслеживанием циклических зависимостей. При использовании спецификационных ссылок в операторах присваивания, передаче ссылок в качестве аргументов функций, возвращении ссылки из функции, выходе ссылки из области видимости, счетчики ссылок изменяются автоматически. Память, выделенная под значение спецификационного типа, автоматически освобождается при обнулении счетчика ссылок на это значение.

Использование указателей на спецификационные ссылки и составных типов языка C, содержащих спецификационные ссылки, не рекомендуется, так как в таких случаях не поддерживается автоматическое изменение счетчиков ссылок.

Функции, реализующие базовые операции над значениями спецификационных типов, находятся в библиотеке спецификационных типов CTESK (см. «Библиотека спецификационных типов»).

4.8.1 Создание значения спецификационного типа

```
Object* create(const Type *type, ...)
```

В качестве первого параметра функция получает указатель на дескриптор спецификационного типа. Дескриптор спецификационного типа — это константа, имя которой состоит из имени типа с префиксом `type_`:

```
const Type type_имя_спецификационного_типа;
```

Остальные параметры являются параметрами инициализации типа. Они отличаются для разных типов и передаются в списке типа `va_list*` в функцию инициализации спецификационного типа.

Функция выделяет память для значения спецификационного типа, обнуляет ее, вызывает функцию инициализации типа, передавая ей в списке типа `va_list*` полученные значения

параметров инициализации типа, и возвращает указатель на выделенную и инициализированную память.

```
Integer* i  = create(&type_Integer, 10);
String*  str = create(&type_String, "a string");
```

В приведенном выше коде создаются и инициализируются ссылки библиотечных спецификационных типов `Integer` и `String` — спецификационного представления встроенного типа языка C `int` и строкового спецификационного типа соответственно. При создании ссылки типа `Integer*` функции `create` должно передаваться целочисленное значение, которое будет храниться по ссылке. При создании ссылки типа `String*` должна быть передана обычная строка языка C.

При использовании функции `create` возможны ошибки из-за использования нетипизированного списка аргументов `...`. Для исключения таких ошибок рекомендуется для каждого спецификационного типа создавать функцию `create_имя_спецификационного_типа` и вызывать `create` из нее, например:

```
specification typedef struct {int a; int b;} IntPair ={};

IntPair* create_IntPair(int a, int b)
{
  return create(&type_IntPair, a, b);
}
```

При использовании `create` вне функции, имя которой имеет префикс `create_`, транслятор выдаст предупреждение:

```
warning: call create() out of create_... function
```

4.8.2 Копирование значения спецификационного типа

```
void copy(Object* src, Object* dst)
```

Функция копирует данные, находящиеся по ссылке *`src`*, на место данных, находящихся по ссылке *`dst`*. Ссылки должны быть ненулевыми и одного типа, то есть они должны иметь одинаковые дескрипторы типов. Если эти условия не выполняются, то во время выполнения происходит завершение программы с сообщением об ошибке. Перед вызовом пользовательской функции копирования функция `copy` уничтожает значение по ссылке *`dst`*.

```
SpecificationType* ref1 = create(...);
SpecificationType* ref2 = create(...);
copy(ref1,ref2);
```

В приведенном выше примере ссылки *`ref1`* и *`ref2`* после инициализации ссылаются на разные значения спецификационного типа `SpecificationType`. После вызова функции `copy()` значение по ссылке *`ref2`* становится равным значению по ссылке *`ref1`*.

4.8.3 Клонирование значения спецификационного типа

```
Object* clone(Object* ref)
```

Функция выделяет память для значения типа, на который ссылается *`ref`*, инициализирует выделенную память значением, эквивалентным значению по ссылке *`ref`*, и возвращает указатель на выделенную и инициализированную память.

```
SpecificationType* ref1 = create(...);
SpecificationType* ref2 = clone(ref1);
```

Значения по ссылкам *`ref1`* и *`ref2`* становятся равными после вызова `clone`.

4.8.4 Сравнение значений спецификационных типов

```
int compare(Object* left, Object* right)
```

В случае равенства значений по переданным ссылкам функция возвращает нулевое значение. Если значения не равны, функция возвращает ненулевое значение, которое может интерпретироваться в зависимости от типа сравниваемых значений. Например, для библиотечного типа `String` результат будет таким же, как у функции `strcmp()` для типа языка C `char*`. Если параметры имеют несравнимые типы, то есть типы ссылок неодинаковые, не являются подтипами одного типа, и тип одной ссылки не является подтипом другой (см. Инвариант типа), то функция возвращает ненулевое значение. Если одна из ссылок нулевая, а другая нет, то возвращается ненулевое значение. Если обе ссылки нулевые, то возвращается ноль.

```
/* создание двух ссылок типа SpecificationType* */
SpecificationType* ref1 = create(&type_SpecificationType);
SpecificationType* ref2 = create(&type_SpecificationType);
...
if (!compare(ref1,ref2)) {/* значения эквивалентны */
  ...
}
else {/* значения не эквивалентны */
  ...
}
```

4.8.5 Сравнение значений спецификационных типов на равенство

```
bool equals(Object* self, Object* ref)
```

Функция возвращает значение `true`, если значения по переданным ссылкам эквивалентны, и `false` в противном случае. Если параметры имеют различный тип, то функция возвращает `false`. Если одна из ссылок нулевая, а другая нет, то возвращается `false`. Если обе ссылки нулевые, то возвращается `true`.

```
SpecificationType* ref1 = create(&type_SpecificationType);
SpecificationType* ref2 = create(&type_SpecificationType);
...
if (equals(ref1,ref2)) {/* значения эквивалентны */
  ...
} else {/* значения не эквивалентны */
  ...
}
```

4.8.6 Строковое представление значения спецификационного типа

```
String* toString(Object* ref)
```

Функция возвращает ссылку на значение типа `String` — спецификационное представление строкового типа.

```
SpecificationType* ref = create(&type_SpecificationType);
String* str;
...
/* преобразование *ref в строковое представление */
str = toString(ref);
printf("*ref == %s/n", toCharArray_String(str);
```

В приведенном выше фрагменте кода используется библиотечная функция `toCharArray_String`, возвращающая содержимое строки в виде массива типа `char`, заканчивающегося нулевым значением — `'\0'`. Эта функция возвращает указатель на внутренние данные, доступные через переданную ссылку типа `String*`. Поэтому, во-первых, для возвращаемого указателя нельзя вызывать `free`; во-вторых, этим указателем нельзя пользоваться после уничтожения значения по переданной ссылке.

4.8.7 Построение XML-представления значения спецификационного типа

```
String *to_XML_MyType( Object *ref )
```

Функция возвращает спецификационную строку, содержащую XML-представление параметра *ref*.

4.8.8 Создание новых спецификационных типов

Спецификационные типы вводятся с помощью обычной конструкции языка C `typedef`, помеченной ключевым словом SeC `specification`.

Различаются объявление спецификационного типа

```
specification typedef базовый_тип новый_тип;
```

и его определение, в котором должен присутствовать инициализатор

```
specification typedef базовый_тип новый_тип = {
  .init = указатель_на_функцию_инициализации
, .copy = указатель_на_функцию_копирования
, .compare = указатель_на_функцию_сравнения
, .to_string = указатель_на_функцию_преобразования_в_строку
, .to_XML = указатель_на _функцию_преобразования_в_XML
, .enumerate = указатель_на_функцию_перечисления_ссылок
, .destroy = указатель_на_функцию_освобождающую_ресурсы
};
```

В каждой единице трансляции до использования спецификационного типа он должен быть объявлен или определен.

Определение спецификационного типа должно встречаться ровно один раз только в одной из единиц трансляции, которые собираются в единую систему.

В определении спецификационного типа в качестве базового типа запрещается использовать:

- спецификационные типы, недоопределенные структуры и массивы неопределенной длины;
- структуры, объединения и массивы определенной длины, содержащие элементы типов, указанных выше;
- указатели на все вышеперечисленные типы, кроме спецификационных ссылок.

Допускается использование недоопределенных структур в объявлениях спецификационных типов.

Инициализатор в определении спецификационного типа определяет, какие функции будут использоваться для базовых операций над данными спецификационного типа.

Поле *init* имеет тип `Init`:

```
typedef void (*Init)(void*, va_list*);
```

По этому указателю из библиотечной функции `create` при создании ссылки вызывается функция инициализации данного спецификационного типа. В первом параметре ей передается указатель на выделенную область памяти, которая должна быть инициализирована. Во втором аргументе передается список параметров, на основе которых инициализируются данные спецификационного типа. Список параметров строится из параметров функции `create`, следующих за первым параметром — дескриптором типа. Поэтому параметры функции `create` должны по типам и порядку соответствовать типам и порядку, ожидаемым в функции инициализации спецификационного типа. Дескриптор типа — глобальная константа типа `Type` — неявно объявляется или определяется при объявлении или определении спецификационного типа соответственно, и имеет имя, состоящее из имени типа и префикса `type_: type_имя_типа`.

Поле *`copy`* имеет тип `Copy`:

```
typedef void (*Copy)(void*,void*);
```

По этому указателю из библиотечных функций `copy` и `clone` вызывается функция копирования значений данного спецификационного типа. Функция копирования спецификационного типа вызывается, если ссылки, переданные в функцию `copy` или `clone`, ненулевые. Первый параметр — это ссылка, значение по которой должно быть скопировано в область памяти по ссылке, которая передается во втором параметре.

Поле *`compare`* имеет тип `Compare`:

```
typedef int (*Compare)(void*,void*);
```

По этому указателю из библиотечных функций `compare` и `equals` вызывается функция сравнения значений данного спецификационного типа. Функция сравнения значений спецификационного типа вызывается, если ссылки, переданные в функцию `compare` или `equals`, ненулевые и типы ссылок либо одинаковые, либо являются подтипами одного типа, либо тип одной ссылки является подтипом другой (см. Инвариант типа). В качестве параметров функции передаются ссылки в том же порядке, в котором они были переданы в функцию `compare` или `equals`.

Поле *`to_string`* имеет тип `ToString`:

```
typedef String* (*ToString)(void*);
```

По этому указателю из библиотечной функции `toString` вызывается функция построения строкового представления данного спецификационного типа. Функция построения строкового представления спецификационного типа вызывается, если ссылка, переданная в `toString`, ненулевая.

Поле *`enumerate`* имеет тип `Enumerate`:

```
typedef void (*Enumerate) (void*,void(*callback)(void*,void*),void*);
```

По этому указателю вызывается функция перечисления ссылок спецификационных типов, содержащихся в значении данного спецификационного типа. Данная функция используется для разрешения циклов по спецификационным ссылкам при автоматическом управлении динамической памятью.

Поле *`destroy`* имеет тип `Destroy`:

```
typedef void (*Destroy)(void*);
```

По этому указателю автоматически вызывается функция освобождения ресурсов при обнулении счетчика ссылок на значение данного спецификационного типа.

Если в определении спецификационного типа базовый тип является допустимым типом языка SeC, то инициализация любого поля может быть опущена. При этом для

соответствующей базовой операции над данными спецификационного типа используется функция по умолчанию. Если функции по умолчанию для всех базовых операций реализуют необходимую функциональность, в определении спецификационного типа используется пустой инициализатор.

```
specification typedef struct {int x; int y;} Point = {};
Point *pt2, *pt1 = create(&type_Point, 1, 2);
/* в результате : pt1->x == 1, pt1->y == 2*/
```

В приведенном выше примере спецификационный тип `Point` создается на основе структуры, содержащей два поля типа `int`. В определении типа используется пустой инициализатор. Поэтому для реализации базовых операций над данными этого типа используются функции по умолчанию. При создании ссылки типа `Point*` в функцию `create` надо передавать значения для инициализации полей базовой структуры (см. Функция инициализации по умолчанию). После создания ссылки типа `Point*` с ней можно работать как с указателем на базовую структуру.

Для создания структур данных со сложной топологией, например при определении рекурсивных структур, рекомендуется использовать только спецификационные ссылки.

```
struct link;
specification typedef struct link Link;
struct link
{
  Link* next;
  int item;
};
specification typedef struct link Link = {};
Link*   l1 = create(&type_Link, NULL, 1)
      , l2 = create(&type_Link, NULL, 2);
l1->next = l2;
```

В представленном выше фрагменте кода определяется спецификационный тип `Link`, реализующий односвязный список. При присваивании полю *`next`* ссылки *`l1`* значения ссылки *`l2`* происходит автоматическое увеличение счетчика ссылок на значение по ссылке *`l2`*. Поэтому после уничтожении ссылки *`l2`* значение, на которое она ссылается, не уничтожается.

При использовании в определении типа `Link` рекурсивной структуры, содержащий неспецификационный указатель на саму себя, правильное управление динамической памятью более трудоемко.

```
specification typedef struct link {
  struct link* next;
  int  item
  } Link = {};
...
struct link* s = malloc(sizeof(struct link));
Link* l = create(&type_Link, s, 1);
...
free(s);
```

В приведенном выше фрагменте кода после вызова `free` для указателя *`s`*, поле *`next`* ссылки *`l`* будет указывать на освобожденную память. В этом случае, чтобы избежать таких проблем, необходимо определить специальные функции инициализации и освобождения ресурсов для типа `Link`.

Если функция по умолчанию некоторой базовой операции не реализует функциональность, необходимую для определяемого спецификационного типа, то для этой операции

определяется специальная функция, указателем на которую инициализируется соответствующее поле в инициализаторе определения типа. Чаще всего такая необходимость возникает, когда базовый тип в определении спецификационного типа является указателем на первый элемент динамического массива, объединением, указателем на один из таких типов, структурой или массивом фиксированный длины, содержащих элементы перечисленных типов.

4.8.9 Реализация базовых операций спецификационных типов по умолчанию

Если в определении спецификационного типа базовый тип является допустимым типом языка SeC, то инициализация любого поля может быть опущена. При этом для соответствующей базовой операции над данными спецификационного типа используется функция по умолчанию. Если функции по умолчанию для всех базовых операций реализуют необходимую функциональность, в определении спецификационного типа используется пустой инициализатор.

4.8.10 Функция инициализации по умолчанию

Функция инициализации по умолчанию для всех спецификационных типов, определенных на основе простых типов (не составных), имеет единственный дополнительный параметр базового типа. Она инициализирует значение спецификационного типа посредством глубокого копирования этого параметра, с учетом возможных циклов по указателям и спецификационным ссылкам.

Функция инициализации структурных спецификационных типов имеет дополнительные параметры, типы и порядок которых совпадают с типами и порядком полей базовой структуры. Поля по переданной ссылке инициализируются посредством глубокого копирования переданных параметров.

Функция инициализации спецификационных типов, определенных на базе массива фиксированной длины, имеет один дополнительный параметр, являющийся указателем на набор значений типа элементов массива в количестве, совпадающем с размерностью массива. Массив по переданной ссылке инициализируется посредством глубокого копирования каждого значения по полученному указателю в соответствующий ему элемент массива.

Механизм копирования, используемый в функции инициализации, совпадает с механизмом копирования, используемым в функции копирования по умолчанию.

4.8.11 Функция копирования по умолчанию

Функция копирования по умолчанию обеспечивает глубокое копирование с учетом возможных циклов по указателям и спецификационным ссылкам, содержащихся по копируемой ссылке:

- значения допустимых простых типов языка C, за исключением типизированных указателей, копируются побайтно;
- спецификационные ссылки копируются с помощью функции копирования соответствующего спецификационного типа;
- типизированные указатели рассматриваются как указатели на одиночное значение, не зависящее от месторасположения в памяти, поэтому копируется единственное

значение по ненулевому указателю согласно правилам, перечисленным в данном списке;

- значения составных типов копируются посредством применения перечисленных правил к каждому составляющему элементу.

4.8.12 Функция сравнения по умолчанию

Функция сравнения по умолчанию сравнивает значения базового типа следующим образом:

- арифметические типы, функциональные указатели и нетипизированные указатели сравниваются побайтно;
- типизированные указатели рассматриваются как указатели на одиночное значение, не зависящее от месторасположения в памяти, то есть нулевые указатели всегда считаются равными, нулевой указатель и ненулевой указатель всегда не равны, а ненулевые указатели равны тогда и только тогда, когда равны значения, на которые они указывают, причем значения по указателям сравниваются согласно правилам данного списка;
- спецификационные ссылки сравниваются с помощью библиотечной функции сравнения `compare`;
- составные типы сравниваются посредством применения данных правил сравнения к каждому составляющему элементу.

4.8.13 Функция построения строкового представления по умолчанию

Функция построения строкового представления по умолчанию возвращает строковое представление значения базового типа:

- для арифметических типов — числовое представление;
- для нетипизированных и функциональных указателей — адрес;
- для типизированных указателей — либо `NULL`, либо строковое представление значения, лежащего по ненулевому указателю, помеченное его адресом;
- для спецификационных ссылок — результат вызова библиотечной функции преобразования в строку `toString`;
- для структурных типов — конкатенация строковых представлений полей структуры, разделенных запятыми, обрамленная фигурными скобками и предваренная словом `struct`;
- для массива фиксированной длины — конкатенация строковых представлений элементов массива, разделенных запятыми, обрамленная квадратными скобками.

4.8.14 Функция построения XML-представления по умолчанию

Функция построения XML-представления по умолчанию возвращает XML-представление значения базового типа.

4.8.15 Функция перечисления внутренних спецификационных ссылок по умолчанию

Функция перечисления внутренних спецификационных ссылок по умолчанию ничего не делает, если базовый тип является простым неспецификационным типом. Если базовый тип является спецификационной ссылкой, вызывается функция по переданному указателю *`callback`*, которой в качестве параметров передаются спецификационная ссылка и вспомогательный параметр *`par`*, переданные в функцию перечисления спецификационных ссылок. Если базовый тип является составным, данные правила применяются для каждого составляющего элемента.

4.8.16 Функция освобождения ресурсов по умолчанию

Функция освобождения ресурсов по умолчанию ведет себя следующим образом:

- ничего не делает, если базовый тип является арифметическим, функциональным или нетипизированным указателем;
- если базовый тип является спецификационной ссылкой, счетчик ссылок на значение по ней уменьшается на единицу;
- если базовый тип является типизированным указателем, то значение по ненулевому указателю обрабатывается согласно перечисляемым правилам, после чего вызывается функция `free` для самого указателя;
- если базовый тип является составным, то перечисленные правила применяются для каждого составляющего элемента.

4.8.17 Определение собственных функций для базовых операций спецификационных типов

Если для некоторой базовой операции функция по умолчанию не реализует функциональность, необходимую для определяемого спецификационного типа, то для этой операции определяется специальная функция, указателем на которую инициализируется соответствующее поле в инициализаторе определения типа. Чаще всего такая необходимость возникает, когда базовый тип в определении спецификационного типа является указателем на первый элемент динамического массива, объединением, указателем на один из таких типов, структурой или массивом фиксированный длины, содержащих элементы перечисленных типов.

4.8.18 Функция инициализации спецификационного типа

```
void имя_функции_инициализации(void* p, va_list* arg_list)
```

Функция не имеет возвращаемого значения. В первом аргументе функция получает указатель типа `void*` на обнуленную область памяти, выделенную для хранения данных спецификационного типа и инициализирует эту область значениями, переданными во втором аргументе в списке типа `va_list*`. При необходимости в функции выделяется дополнительная память для хранененения данных.

```
struct integer_seq {
  int length;
  Integer* *items;
};
```

```
void init_IntegerSeq(void* ref, va_list *arg_list) {
  struct integer_seq *is = (struct integer_seq*)ref;

  is->length = va_arg(*arg_list, int);
  is->items  = calloc(is->length, sizeof(Integer*));
}

specification typedef struct integer_seq IntegerSeq = {
  .init = init_IntegerSeq,

};
```

В приведенном выше примере создается спецификационный тип `IntegerSeq`, предназначенный для хранения последовательности заранее не известной длины, содержащей ссылки типа `Integer*` — ссылки на значения библиотечного спецификационного типа `Integer`, который является спецификационным представлением встроенного типа языка C `int`. Тип `IntegerSeq` определяется на основе структуры `integer_seq` с двумя полями: длина последовательности — *`length`* и указатель на массив, содержащий саму последовательность,— *`items`*.

Библиотечная функция `create` выделяет память только для хранения значения самой структуры `integer_seq`. Функция инициализации по умолчанию для такого структурного спецификационного типа имеет два параметра инициализации: первый параметр типа `int` и второй типа `Integer**`. Причем второй параметр в функции инициализации по умолчанию интерпретируется как указатель на единственное значение. Поэтому в функции по умолчанию поле *`items`* инициализируется указателем на ссылку, которая содержит копию значения по ссылке, переданной через указатель. В рассматриваемом случае такая функциональность неприемлема. Поэтому необходимо использовать специальную функцию инициализации `init_IntegerSeq`, реализующую необходимую функциональность.

Функция `init_IntegerSeq` ожидает, что в списке типа `va_list*` передается единственный параметр инициализации типа `IntegerSeq` — длина последовательности. Его значением инициализируется поле *`length`* по инициализируемой ссылке. Второе поле *`items`* инициализируется указателем на динамически выделенную и инициализированную нулями область памяти, достаточную для хранения последовательности ссылок нужной длины.

Указателем на функцию инициализации `init_IntegerSeq` инициализируется поле *`init`* в определении типа `IntegerSeq`.

4.8.19 Функция копирования спецификационного типа

```
void имя_функции_копирования(void* src, void* dst)
```

Функция не имеет возвращаемого значения. Имеет два параметра типа `void*`. Функция должна копировать на необходимую глубину значения данных по переданному в первом параметре указателю *`src`* в обнуленную область памяти по переданному во втором параметре указателю *`dst`*.

```
void copy_IntegerSeq (void *src, void *dst) {
  struct integer_seq   *is_src = (struct integer_seq *)src
                     , *is_dst = (struct integer_seq *)dst;
  int i;

  is_dst->length = is_src->length;
  is_dst->items  = calloc(is_src->length, sizeof(Integer*));
  for (i = 0; i < is_src->length; i++)
```

```
    is_dst->items[i] = clone(is_src->items[i]);
}
specification typedef struct integer_seq IntegerSeq = {
  .init    = init_IntegerSeq,
  .copy    = copy_IntegerSeq,
  .destroy = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция копирования значений типа `IntegerSeq`. Определение этой функции необходимо, так как функция копирования по умолчанию интерпретирует поле *`items`* как указатель на единственное значение типа `Integer*`, то есть копируется только первое значение по ссылке первого элемента последовательности *`src`*. Функция `copy_IntegerSeq` обеспечивает глубокое копирование всей последовательности при помощи библиотечной функции копирования `clone`. Инициализация нулями памяти, выделяемой для `is_dst->items` необходима для того, чтобы при присваивании в цикле `is_dst->items[i] = clone(is_src->items[i])` не происходило попытки уменьшения счетчика ссылок на несуществующее значение по ссылке `is_dst->items[i]`.

Указателем на функцию копирования инициализируется поле *`copy`* в определении типа `IntegerSeq`.

4.8.20 Функция сравнения спецификационного типа

```
int имя_функции_сравнения(void* left, void* right)
```

Функция имеет возвращаемое значение типа `int` и два параметра типа `void*`. Функция сравнивает значения по переданным указателям и возвращает ноль, если значения эквивалентны, и отличное от нуля значение в обратном случае. Ненулевое значение может зависеть от отношения значений по переданным ссылкам.

```
int compare_IntegerSeq(void* left, void* right) {
  struct integer_seq   *isl = (struct integer_seq *)left
                     , *isr = (struct integer_seq *)right;
  if (isl->length != isr->length) return isl->length – isr->length;
  else {
    int i, res;
    for (i = 0; i < isl->length; i++) {
      res = compare(isl->items[i], isr->items[i]);
      if (res) return res;
    }
  }
  return 0;
}
specification typedef struct integer_seq IntegerSeq = {
  .init    = init_IntegerSeq,
  .copy    = copy_IntegerSeq,
  .compare = compare_IntegerSeq,
  .destroy = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция сравнения значений типа `IntegerSeq`. Определение этой функции необходимо, так как функция сравнения по умолчанию интерпретирует поле *`items`* как указатель на единственное значение типа `Integer*`, то есть сравниваются только значения по ссылкам первых элементов последовательностей *`left`* и *`right`*.

Функция сравнения `compare_IntegerSeq` обеспечивает сравнение последовательностей поэлементно. Если последовательности имеют разную длину, то возвращается разность длин последовательностей по переданным ссылкам *`left`* и *`right`*. Если последовательности одинаковой длины, то они сравниваются поэлементно при помощи библиотечной функции `compare`. В этом случае, если все элементы последовательностей совпадают, результатом является ноль, в противном случае возвращается результат сравнения первых несовпадающих элементов.

Указателем на функцию сравнения инициализируется поле *`compare`* в определении типа `IntegerSeq`.

4.8.21 Функция построения строкового представления спецификационного типа

```
String* имя_функции_построения_строкового_представления(void* p)
```

Функция имеет возвращаемое значение типа `String*` и параметр типа `void*`. Функция возвращает ссылку на спецификационный тип `String`, по которой должно содержаться строковое представление значения спецификационного типа, переданного в единственном параметре функции.

```
String* to_string_IntegerSeq(void *ref) {
  struct integer_seq *is = (struct integer_seq *)ref;

  String *start = create_String ("<");
  String *end   = create_String (">");
  String *sep   = create_String (", ");
  String *res = start;

  if (is->length > 0) {
    int i;
    for (i = 0; i < is->length; i++) {
      if (i > 0) res = concat_String(res, sep);
      res = concat_String(res, toString(is->items[i]));
    }
  }
  return concat_String (res, end);
}

specification typedef struct integer_seq IntegerSeq = {
  .init      = init_IntegerSeq,
  .copy      = copy_IntegerSeq,
  .compare   = compare_IntegerSeq,
  .to_string = to_string_IntSeq,
  .destroy   = destroy_IntegerSeq
}
```

В приведенном выше примере определяется функция построения строкового представления `to_string_IntegerSeq` для типа `IntegerSeq`. Определение этой функции необходимо, так как функция построения строкового представления по умолчанию интерпретирует поле *`items`* как указатель на единственное значение типа `Integer*` и создает строку, содержащую в фигурных скобках через запятую значение поля *`length`* и строковое представление значения по ссылке первого элемента последовательности.

Функция `to_string_IntSeq` возвращает ссылку типа `String*`, содержащую строку, в которой в угловых скобках перечисляются через запятую строковые представления значений

по ссылкам всех элементов последовательности с сохранением их порядка. В функции to_string_IntSeq используются функции create_String и create_String.

Указателем на функцию to_string_IntSeq инициализируется поле *to_string* в определении типа IntegerSeq.

4.8.22 Функция построения XML-представления спецификационного типа

```
String* имя_функции_построения_XML_представления(void* p)
```

Функция имеет возвращаемое значение типа String* и параметр типа void*. Функция возвращает ссылку на спецификационный тип String, которая должна содержать XML-представление значения спецификационного типа, переданного в единственном параметре функции.

Если тип не имеет интересующей нас внутренней структуры, то функцию преобразования рекомендуется писать по аналогии с to_XML_Integer():

```
String *to_XML_MyType( MyType *mt )
{
  return to_XML_spec("MyType", to_string_MyType(mt));
}
```

Если внутренняя структура типа важна, функцию преобразования можно написать, например, по аналогии с функцией библиотечного типа Set (множество):

```
String *to_XML_MyType( MyType *mt )
{
  String *res = format_String("<object kind=\"spec\" type=\"MyType\"
text=\"Заголовок\">\n" [, ...]);
  foreach(Object obj: вложенные объекты, наследующие Object)
  {
    res = concat_String( res, toXML( r(obj) ) );
  }
  foreach(data: вложенные объекты с kind = Simple, Struct, Pointer,
Array)
  {
    res = concat_String( res, ts_to_XML_sectype(&typeDesc, &data) );
  }
  res = concat_String(res, create_String("</object>\n"));
  return res;
}
```

или библиотечного типа Map (отображение):

```
String *to_XML_MyType( MyType *mt )
{
  String *res
    = format_String
      ( "<object kind=\"spec\" type=\"MyType\" text=\"Заголовок\">\n
      [, ...]
      );

  foreach( Object obj: вложенные объекты, наследующие Object )
  {
    res = concat_String
          ( res
          , format_String
            ( "<object kind=\"spec\" "
```

```
                "type=\" MyElement\" "
                "text=\"заголовок элемента\">"
              [, ...]
              )
            );
      res = concat_String( res, toXML( r(obj) ) );
      res = concat_String( res, create_String("</object>\n"));
    }
    foreach( data: вложенные объекты с kind = Simple, Struct, Pointer,
  Array )
    {
      res = concat_String( res
                         , format_String
                           ( "<object kind=\"simple\" "
                             "type=\" MyType.Element\" "
                             "text=\"элемент\"/>"
                           [, ...]
                           )
    }
    res = concat_String(res, create_String("</object>\n"));
    return res;
  }
```

Комбинируя шаблоны генерации из вышеприведённых примеров, можно создавать произвольную древовидную структуру объектов.

4.8.23 Функция перечисления внутренних спецификационных ссылок спецификационного типа

```
void имя_функции_перечисления_спецификационных_ссылок(*Enumerate)
    (void* p, void (*callback)(void*,void*), void* par)
```

Функция перечисления ссылок должна для каждой ссылки спецификационного типа, доступной через переданную в первом аргументе ссылку, вызвать функцию, указатель на которую передается ей во втором аргументе. Во всех вызовах этой функции перечисляемые ссылки передаются в первом аргументе, во втором аргументе передаются параметры через указатель *par*, переданный в третьем аргументе функции перечисления.

```
void enumerate_IntegerSeq( void* p
                         , void (*callback)(void*,void*)
                          , void* par
                         ) {
  struct integer_seq *is = (struct integer_seq*)p;
  int i;
  for (i = 0; i < is->length; i++)
    callback(is->items[i], par);
}

specification typedef struct integer_seq IntegerSeq = {
  .init      = init_IntegerSeq,
  .copy      = copy_IntegerSeq,
  .compare   = compare_IntegerSeq,
  .enumerate = enumerate_IntegerSeq,
  .destroy   = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция перечисления ссылок `enumerate_IntegerSeq` для типа `IntegerSeq`. Определение специальной функции

перечисления ссылок необходимо, так как функция перечисления ссылок по умолчанию не обеспечивает перечисление ссылок, доступных через указатель на массив элементов.

В функции `enumerate_IntegerSeq` для всех ссылок последовательности, доступных через поле *`items`* типа `Integer**`, вызывается функция через указатель, переданный в функцию перечисления `enumerate_IntegerSeq` во втором параметре. В первом параметре при этих вызовах передаются перечисляемые спецификационные ссылки, во втором параметре передается указатель, переданный в третьем параметре в функцию перечисления `enumerate_IntegerSeq`.

Указателем на функцию освобождения ресурсов `enumerate_IntegerSeq` инициализируется поле *`enumerate`* в определении типа `IntegerSeq`.

4.8.24 Функция освобождения ресурсов спецификационного типа

```
void имя_функции_освобождения_ресурсов(void* p)
```

Функция не имеет возвращаемого значения. Имеет один параметр типа `void*` — указатель на область памяти, хранящую данные спецификационного типа. Функция должна освободить только дополнительную память, выделенную в функции инициализации данного спецификационного типа.

```
void destroy_IntegerSeq (void *ref) {
  struct integer_seq* is = (struct integer_seq*)ref;
  int i;
  for (i = 0; i < is->length; i++)
    is->items[i] = NULL;
  free (is->items);
}

specification typedef struct integer_seq IntegerSeq = {
  .init    = init_IntegerSeq,
  .destroy = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция освобождения ресурсов `destroy_IntegerSeq` для спецификационного типа `IntegerSeq`. Определение этой функции необходимо, так как функция освобождения ресурсов по умолчанию для составных типов интерпретирует поле *`items`* как указатель на единственное значение типа `Integer*`, поэтому счетчик ссылок уменьшается только для первой ссылки последовательности, после чего вызывается `free` для указателя *`items`*.

Функция `destroy_IntegerSeq` уменьшает счетчик ссылок на единицу для каждого элемента последовательности по переданной ссылке, после чего освобождает память по указателю *`items`*. Счетчики ссылок уменьшаются присваиванием ссылкам `NULL`.

Указателем на функцию освобождения ресурсов `destroy_IntegerSeq` инициализируется поле *`destroy`* в определении типа `IntegerSeq`.

4.9 Инварианты

> В спецификации формулируются требования к тестируемой системе, в том числе требования к данным. Такие требования заключаются в ограничении диапазона допустимых значений и могут накладываться как на некоторый тип данных в целом (с помощью инвариантов типов), так и на значения отдельных глобальных переменных (с помощью инвариантов переменных).

Инварианты могут рассматриваться и как общая часть предусловий и постусловий спецификационных функций, которые используют данные соответствующих типов.

Инварианты автоматически проверяются в спецификационных функциях перед проверкой предусловия:

- для параметров функции;
- для выражений, описанных в ограничения доступа `reads` и `updates`.

и перед проверкой постусловия:

- для параметров функции;
- для выражений, описанных в ограничениях доступа `writes` и `updates`.
- для возвращаемого значения

Также предусмотрен способ явной проверки инварианта, см. примеры в описании инвариантов типа и переменных.

При проверке инвариантов составных типов автоматически происходит проверка инвариантов и для всех его составляющих элементов.

4.9.1 Инвариант типа

Инвариант типа вводит ограничения на диапазон значений некоторого типа. Получающийся новый тип, множество значений которого является подмножеством значений базового типа, называется подтипом. На базовый тип наложено ограничение: он должен являться допустимым типом.

Тип с инвариантом определяется с помощью конструкции `typedef`, помеченной ключевым словом `invariant`:

```
invariant typedef int Nat;
```

В данном случае `int` является базовым типом, а `Nat` — определяемым подтипом. При этом в отличие от обычной конструкции `typedef` языка C, которая лишь вводит новое имя для старого типа, в данном случае определяется новый тип с собственным множеством значений.

Ограничения на множество значений подтипа определяются в конструкции `invariant`, схожей с определением функции с одним параметром определяемого подтипа. Функция возвращает логическое значение: `true`, если переданное значение удовлетворяет ограничениям (инвариант выполнен), или `false`, если не удовлетворяет (инвариант нарушен). Поскольку тип возвращаемого значения фиксирован, он не указывается явно:

```
invariant(Nat n) {
  return n > 0;
}
```

Если в качестве базового типа используется спецификационный тип, то параметр функции-инварианта будет иметь тип соответствующей спецификационной ссылки, поскольку значения спецификационных типов доступны только через указатель:

```
invariant typedef Integer Natural;
invariant(Natural* n) {
  return value_Integer(n) > 0;
}
```

При этом функции работы с подтипом будут взяты из определения базового типа.

Функция-инвариант не должна иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри функции, должна освобождаться в ней же.

Допускаются определения новых спецификационных типов с указанием инварианта:

```
invariant specification typedef int Natural = {};
invariant(Natural* n) {
  return *n > 0;
}
```

В этом случае подтип и базовый тип будут совпадать.

Отметим, что нельзя определять спецификационный тип на основе другого спецификационного типа, однако можно создавать подтипы спецификационных типов. Более того, можно определять подтипы подтипов, создавая таким образом иерархию типов. В таком случае для значения подтипа будут проверяться и инварианты всех «родительских» подтипов на пути к вершине иерархии.

Инвариант переменной соответствующего типа можно явно проверить с помощью функции invariant:

```
invariant typedef int Nat;
Nat n;
if ( invariant(n) ) ...
```

Подтип можно приводить к базовому типу, более того, такое преобразование осуществляется неявно. Базовый тип также можно привести к подтипу. Однако, поскольку значения подтипа являются подмножеством значений базового типа, такое преобразование нужно записывать явно:

```
invariant typedef int Nat;
int i;
Nat n;
...
i = n;
n = (Nat)i;
```

Инвариант типа при приведении может быть нарушен, поэтому после приведения следует проверять его явно.

4.9.2 Инвариант переменной

Инвариант переменной позволяет ввести ограничения на диапазон значений отдельной глобальной переменной, имеющей допустимый тип.

Переменная с инвариантом определяется с помощью обычной декларации или определения, с ключевым словом `invariant`:

```
invariant int Qty;
```

Ограничения на множество значений переменной определяются в конструкции `invariant`, схожей с определением функции без параметров. Функция возвращает логическое значение: `true`, если значение глобальной переменной удовлетворяет ограничениям (инвариант выполнен), или `false`, если не удовлетворяет (инвариант нарушен). Поскольку тип возвращаемого значения фиксирован, он не указывается явно. В скобках указывается имя переменной, для которой определяется инвариант:

```
invariant(Qty) {
  return Qty >= 0;
}
```

Однако переменная не является параметром функции-инварианта. Функция осуществляет доступ непосредственно к значению глобальной переменной. При этом функция не должна иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри функции, должна освобождаться в ней же.

Инвариант переменной можно явно проверить с помощью функции `invariant`:

```
invariant int Qty;
...
if ( invariant(Qty) ) ...
```

Если переменная с инвариантом имеет тип, для которого определен инвариант типа, то сначала будет проверен инвариант типа, а затем инвариант переменной.

5 Покрытия

Критерии покрытия применяются для оценки достигнутой полноты тестирования.

Полное тестирование программы требует проверки правильности ее поведения во всех ситуациях. Общее количество ситуаций обычно слишком велико для проведения полного тестирования. Для уменьшения количества тестовых ситуаций выбирают критерий покрытия, согласно которому все тестовые ситуации разбиваются на классы, называемые *элементами покрытия*.

Выдвигается *гипотеза тестирования* о том, что любая ошибка в программе, которая проявляется в ситуации, отнесенной к определенному классу, проявляется и на любой другой ситуации из этого класса. То есть, если принять гипотезу тестирования, для полного тестирования достаточно проверить правильность поведения программы на одной ситуации из каждого класса. Если критерий покрытия выбран адекватно, то гипотеза тестирования верна для всех ошибок. В этом случае достигнутая степень полноты тестирования соответствует проценту покрытых элементов покрытия для заданного критерия покрытия.

В индустрии разработки ПО применяются критерии покрытия, основанные на структуре исходного кода программы, в частности:

- покрытие операторов;
- покрытие ветвей графа потока управления;
- покрытие путей графа потока управления;
- покрытие условий.

При тестировании методом черного ящика исходный код программы может быть недоступен. В этом случае применяются аналогичные критерии покрытия, основанные на структуре спецификаций.

При тестировании программы на соответствие заданному набору требований критерий покрытия может быть основан на определениях ситуаций, используемых в требованиях. То есть, для требования "если выполнено **условие C**, то должно быть выполнено **требование R**", **условие C** определяет элемент покрытия.

Такое покрытие может оказаться достаточно грубым — **требование R** достаточно будет проверить лишь раз в ситуации, которая удовлетворяет **условию C**. Если этого недостаточно, элементы покрытия могут быть дополнительно разбиты и требование считается полностью протестированным только в случае его проверки при каждом из уточненных условий.

Покрытия в SeC реализуются в виде совокупности элементов, атрибуты которых — имя и опциональное текстовое описание. Каждый элемент покрытия соответствует определённому состоянию тестовой системы.

Покрытия задаются с помощью конструкций SeC, в которых сосредотачивается информация, необходимая для идентификации покрытий и работы с ними и их элементами. SeC позволяет

создавать покрытия различных видов, и требует соблюдения определённых правил для задания покрытий каждого вида.

Базовые операции взаимодействия с элементами покрытий таковы:

- доступ к элементу покрытия (например, вычисление достигнутого элемента);
- сравнение его с другими элементами (например, достигнутого — с ожидаемым);
- занесение информации о достижении элемента покрытия в трассу.

5.1 Различные виды покрытий

Покрытие в языке SeC определяется именем, набором элементов покрытия и необязательной *функцией вычисления* текущего элемента. Элемент покрытия может иметь идентификатор или текстовое представление, или и то, и другое. Функция вычисления по состоянию системы, заданной параметрами и глобальными переменными, вычисляет элемент покрытия, к которому это состояние относится.

Покрытия задаются с помощью конструкций SeC, в которых сосредотачивается информация, необходимая для идентификации покрытий и работы с ними и их элементами. SeC позволяет создавать покрытия различных видов, и требует соблюдения определённых правил для задания покрытий каждого вида.

Виды покрытий различаются следующим образом:

- По принадлежности к спецификационным функциям:
    - глобальное покрытие не привязано к спецификационной функции, определяется и объявляется на уровне модуля компиляции;
    - локальное покрытие привязано к определённой спецификационной функции. Объявления и определения локальных покрытий содержатся, соответственно, в прототипе и определении этой спецификационной функции.
- По способу задания элементов и методу вычисления достигнутого элемента:
    - Если автор теста задаёт совокупность элементов для покрытия непосредственно и отмечает достигнутый элемент самостоятельно, то такое покрытие называется *перечислимым*.
    - Если автор теста задаёт совокупность элементов для покрытия непосредственно и задаёт некоторую функцию вычисления достигнутого элемента, то такое покрытие называется *вычислимым*. Функция вычисления — неотъемлемая часть такого покрытия.
    - Если автор теста задаёт в качестве совокупности элементов покрытия некоторый enum-тип, то функция вычисления генерируется на основе структуры этого типа, а покрытие назвается *enum-покрытием*.
- По положению в иерархии:
    - первичное покрытие, набор элементов которого создавался независимо от других покрытий;
    - производное покрытие, набор элементов которого является комбинацией элементов других покрытий. Покрытия, элементы которых были использованы при создании производного покрытия, называются базовыми по отношению к нему.

Глобальные покрытия могут быть перечислимыми, вычислимыми и enum-покрытиями. Локальные покрытия всегда вычислимые.

5.1.1 Перечислимое покрытие

Перечислимое покрытие — покрытие, не имеющее функции вычисления элементов. Таким образом, достижимость элементов перечислимого покрытия описывается непосредственно автором теста. Задается перечислимое покрытие только набором элементов, обладающих именами и необязательными строковыми описаниями.

```
// объявление перечислимого покрытия
extern coverage ConstCovD =
{
  C1 = "First",
  C2, // строковое описание этого элемента по умолчанию: "C2"
  C3 = "Last"
};

// определение объявленного выше покрытия
coverage ConstCovD;

// определение ранее не объявленного покрытия
coverage ConstCovND =
{
  C1 = "ND1",
  C2 = "ND2",
  C3
};
```

Получить элемент перечислимого покрытия можно только явно указав этот элемент, используя имя покрытия и имя элемента (возможно, составное — у производных покрытий), разделённые точкой.

Явное указание элемента покрытия. Существует возможность в любой момент точно указать нужный элемент покрытия, используя имя покрытия и имя элемента (возможно, составное - у производных покрытий), разделённые точкой.

```
// сброс в трассу указанного элемента перечислимого покрытия
trace ( Cov.C1 );

// сброс в трассу указанного элемента покрытия-произведения
trace ( ProdCov.C1.C2 );
```

5.1.2 Вычислимое покрытие

Вычислимое покрытие используется для реализации определённого алгоритма вычисления достигнутого элемента покрытия в зависимости от некоторых параметров.

Вычислимое покрытие — покрытие, достижимость элементов которого может быть вычислена по алгоритму, определённому автором теста.

Вычислимое покрытие, кроме общих для всех покрытий атрибутов — имени и набора элементов, обладает явно заданной функцией вычисления элементов. Функция вычисления возвращает элемент покрытия, получая в качестве аргументов параметры, которые могут характеризовать состояние тестируемой системы или обозначать ветвь функциональности.

```
// определение не объявленного ранее вычислимого покрытия
coverage AuxCoverage ( int i )
{
    /*
     * имя элемента может отсутствовать,
     * если к нему не требуется доступ из программы -
     * в трассу сбрасываются числовые значения и описания.
     */
    if ( i <= 0 ) return { "Incomprehensible number" };
    if ( i > 3 ) return { MNOGO, "Very large number" };

    /*
     * описание элемента тоже может отсутствовать,
     * в этом случае оно совпадает с именем
     */
```

```
    return { MALO };
}
```

Получить элемент вычислимого покрытия можно либо явно указав нужный элемент (точно так же, как для перечислимых покрытий), либо вычислив достигнутый элемент.

1. *Явное указание элемента покрытия*. Существует возможность в любой момент точно указать нужный элемент покрытия, используя имя покрытия и имя элемента (возможно, составное — у производных покрытий), разделённые точкой.

```
// сброс в трассу указанного элемента перечислимого покрытия
trace ( Cov.C1 );

// сброс в трассу указанного элемента покрытия-произведения
trace ( ProdCov.C1.C2 );
```

2. *Вычисление достигнутого элемента покрытия*. Вычисление элемента покрытия возможно только для вычислимых покрытий.

```
// проверка достигнутого элемента вычислимого покрытия
if ( CalcCov.POZ == CalcCov (i) );
{ ... }
```

5.1.3 Enum-покрытие

Enum-покрытия служат для удобного создания покрытий на основе перечислимых типов.

Покрытие может быть определено на основе перечислимого типа (`enum`). Элементы такого покрытия соответствуют константам перечислимого типа, а функция вычисления определяется автоматически. Эта функция имеет единственный параметр используемого перечислимого типа, для каждого значения перечислимого типа возвращается соответствующий элемент покрытия.

```
typedef enum { RED, GREEN, BLUE } Color;

/*
 * объявление покрытия на основе перечислимого типа,
 * с функцией вычисления по умолчанию
 */
extern coverage enum ColorCoverage( Color color );

/*
 * определение предварительно объявленного покрытия
 * на основе перечислимого типа
 */
coverage ColorCoverage;
```

5.1.4 Производное покрытие

Производные покрытия используются при необходимости повторного использования элементов и функций вычисления покрытий.

Покрытия, элементы и функции вычисления которых задаются непосредственно, называются *первичными*.

Производными называются покрытия, построенные на основе других покрытий, которые в этом контексте являются *базовыми*. Определение производного покрытия должно описывать способ его построения на основе базовых покрытий; базовым покрытием может быть как первичное покрытие, так и производное.

Порядком покрытия называется количество первичных покрытий, комбинации элементов которых являются элементами данного покрытия. Порядок любого первичного покрытия равен 1. Порядок производного покрытия равен сумме порядков всех его базовых покрытий.

Для любого покрытия можно построить список первичных покрытий, на которых производное покрытие в конечном счете основано. Длина этого списка равна порядку покрытия. Этот список формируется из списка, содержащего только данное покрытие, следующим образом: циклически проходя от начала списка к концу, вместо покрытий подставляем их базовые покрытия (произведения становятся последовательностями умножаемых покрытий). В конце концов в списке останутся только первичные покрытия, количество которых будет равно порядку исходного.

Для создания производных покрытий предусмотрено два способа:

- изменение домена покрытия (наследование набора элементов и функции вычисления);
- произведение покрытий (комбинация элементов базовых покрытий и функций вычисления).

Доступ к элементам производного покрытия зависит от его вида.

В результате изменения домена покрытия и произведения вычислимых покрытий получается вычислимое покрытие. Получить элемент, входящий в его состав, можно указав нужный элемент явно, либо вычислив достигнутый элемент, передав параметры функции вычисления. Произведение перечислимого покрытия на перечислимое является перечислимым покрытием. Получить элемент такого покрытия можно только явно указав этот элемент. Таким образом, принципиально доступ к элементам производного покрытия не отличается от доступа к элементам первичного покрытия.

Специфической же для производных покрытий операцией является получение элементов базовых покрытий — операция получения компонента элемента покрытия. Эта операция доступна для покрытий порядка выше первого — то есть для покрытий-произведений или производных от них: тогда элемент данного покрытия является комбинацией элементов других покрытий и можно получить его компонент. Компоненты элементов других покрытий получать запрещено.

5.1.5 Покрытие-произведение

Покрытия-произведения используются при необходимости комбинировать наборы элементов нескольких покрытий.

Элементами произведения покрытий являются сочетания элементов базовых покрытий, каждый из которых называется компонентом элемента произведения покрытий. Элемент произведения покрытий считается достигнутым, если одновременно достигнуты все его компоненты.

```
// определение покрытия-произведения для перечислимых покрытий
coverage ProdCovNF = ConstCovND * ConstCovD;
// определение покрытия-произведения для вычислимых покрытий
coverage ProdCovF (Color color, char ch, int in)
 = AlphaCoverage ( ch ) * ColorCoverage (color) * AuxCoverage ( in );
```

Для покрытий порядка выше первого доступна операция получения компонента элемента покрытия - то есть для покрытий-произведений или производных от них: тогда элемент данного покрытия является комбинацией элементов других покрытий и можно получить его компонент. Компоненты элементов других покрытий получать запрещено.

Получение компонента элемента покрытия

```
// AlphaCoverage и AuxCoverage - вычислимые покрытия

// ColorCoverage - enum-покрытие основанное на типе Color

// определение покрытия-произведения вычислимых покрытий
coverage ProdCovF (Color color, char ch, int in)
 = ColorCoverage (color) * AlphaCoverage ( ch ) * AuxCoverage ( in );

// сброс в трассу компонента элемента покрытия
trace ( ProdCovF( i, (char) i, i )[1] );
```

5.1.6 Изменение домена покрытия

Изменение домена покрытия позволяет использовать функцию вычисления базового покрытия в производном, указав только переменную, которая будет параметром этой функции.

Изменение домена применимо только к вычислимым покрытиям; получаемое производное покрытие также является вычислимым. Такое производное покрытие использует функцию вычисления базового покрытия, задавая выражения, используемые в качестве фактических параметров. В этих выражениях могут использоваться параметры производного покрытия и видимые ему переменные.

```
// базовое вычислимое покрытие
coverage AlphaCoverage( char c )
{
  if ( 'a' <= c && c <= 'z' ) return LOWER;
  if ( 'A' <= c && c <= 'Z' ) return UPPER;
  /*
   * если поток выполнения доходит до этой точки,
   * то покрытие не является полным
   */
}

// переменная, которая будет параметром нового покрытия
extern char charState;

// ( void ) говорит о наличии функции вычисления для этого покрытия
extern coverage StateCoverage ( void ) = AlphaCoverage( charState );
coverage StateCoverage;
```

5.1.7 Локальное покрытие

Покрытие имеет смысл определить как локальное, если его использует единственная спецификационная функция.

Когда состояния тестируемой системы, которые идентифицируют элементы покрытия, достигаются в рамках одной спецификационной функции, удобно сделать это покрытие *локальным*. Локальное покрытие — такое покрытие, которое определено внутри некоторой спецификационной функции; она называется объемлющей по отношению к этому покрытию.

Покрытия, определённые внутри спецификационной функции, называются локальными, в противоположность покрытиям, объявляемым на глобальном уровне. В состав прототипа спецификационной функции включают объявления локальных покрытий, принадлежащих ей, а в ее тело — определения этиих покрытий. Содержательная часть объявления локального покрытия — имя покрытия.

Локальные покрытия должны быть вычислимыми, то есть, могут быть проинициализированы функцией вычисления или произведением покрытий. Набор параметров функции вычисления локального покрытия совпадает с набором параметров спецификационной функции.

Область видимости локальных покрытий — блок, содержащий их определения. Имена локальных покрытий не должны совпадать с именами аргументов спецификационной функции и её именем, а также с именами других локальных покрытий.

```
specification foo (int a, int b, char h)
{
  coverage aCov
  {
    if (a > 0)
      return {a_poz, "a is pozitive"};
    if ((a < 0) && (b > 0))
      return {a_b_oppoz, "a and b are oppozite"};
    if ((a == 0) && (h > a))
      return {a_null_proof, "a is null, approved."};
  }
  post { ... }
}
```

Доступ к элементам локального покрытия можно получить как из постусловия, так и из функций вычисления локальных покрытий, определённых после данного. Для этого достаточно указать имя покрытия: оно будет обозначать элемент, достигнутый в процессе выполнения тестируемой функции. Также для локального покрытия доступна операция получения элемента покрытия, но вычисление элемента покрытия запрещено.

```
coverage CharCmd (char c)
{
  if (c == 'e')
    return {exit, "exit command entered"};
  else if (c == 'h')
    return {help, "help command entered"};
  else
    return {nocmd, "no command or unknown"};
}

specification foo (int a, int b, char h)
{
  coverage hCov = CharCmd (h);
  coverage aCov
  {
    if (hCov == hCov.exit)
      return {abort, "Program terminated"};
    else if (hCov == hCov.help)
      return {pause, "Program standby"};
    else
    {
      if (a > 0)
        return {a_poz, "a is pozitive"};
      if ((a < 0) && (b > 0))
        return {a_b_oppoz, "a and b are oppozite"};
      if ((a == 0) && (h > a))
        return {a_null_proof, "a is null, approved."};
    }
  }
  post
  {
```

```
    if (aCov != aCov.a_null_proof)
    {
      //... }
    }
  }
}
```

5.2 Операции над элементами покрытий.

Операции над элементами покрытий реализуют базовые возможности управления ходом тестирования и отражения его результатов в отчёте.

Базовые операции взаимодействия с элементами покрытий таковы:

1. получение элемента покрытия (например, вычисление достигнутого элемента);
2. сравнение его с другими элементами (например, достигнутого элемента — с ожидаемым);
3. занесение информации о достижении элемента покрытия в отчёт.

Комбинируя эти операции, можно управлять ходом теста в зависимости от достижения тех или иных элементов покрытий и заносить данные о достигнутых элементах в отчёт.

Следует заметить, что от признаков вида покрытия (наличия функции вычисления, принадлежности к спецификационной функции) зависит только первая операция из списка: получение, вычисление элемента покрытия. Операции сравнения и трассировки требуют только правильно полученных элементов покрытия в качестве аргументов и никак не зависят от объявлений и определений покрытий.

Единственная особо предусмотренная SeC комбинация базовых операций — операция итерации по элементам покрытия, которая облегчает реализацию обхода всех нужных состояний тестовой системы — то есть, элементов некоторого покрытия.

5.2.1 Получение элемента покрытия

Доступ к элементам покрытия применяется для занесения в отчёт о тесте информации о достигнутом элементе и для сравнения элементов между собой.

Для различных видов глобальных покрытий предусмотрены различные формы доступа к элементам.

1. *Явное указание элемента покрытия*. Существует возможность в любой момент точно указать нужный элемент покрытия, используя имя покрытия и имя элемента (возможно, составное - у производных покрытий), разделённые точкой.

```
// сброс в трассу указанного элемента перечислимого покрытия
trace ( Cov.C1 );

// сброс в трассу указанного элемента покрытия-произведения
trace ( ProdCov.C1.C2 );
```

2. *Вычисление достигнутого элемента покрытия*. Вычисление элемента покрытия возможно только для вычислимых покрытий.

```
// проверка достигнутого элемента вычислимого покрытия
if ( CalcCov.POZ == CalcCov (i) )
```

Доступ к элементам локального покрытия можно получить как из постусловия, так и из функций вычисления локальных покрытий, определённых после данного. Для этого достаточно указать имя покрытия: оно будет обозначать элемент, достигнутый в процессе выполнения тестируемой функции. Также для локального покрытия доступна операция получения элемента покрытия, но вычисление элемента покрытия запрещено.

```
coverage CharCmd (char c)
{
  if (c == 'e')
    return {exit, "exit command entered"};
  else if (c == 'h')
    return {help, "help command enterd"};
  else
    return {nocmd, "no command or unknown"};
}

specification foo (int a, int b, char h)
{
  coverage hCov = CharCmd (h);
  coverage aCov
  {
    if (hCov == hCov.exit)
      return {abort, "Program terminated"};
    else if (hCov == hCov.help)
      return {pause, "Program standby"};
    else
    {
      if (a > 0)
        return {a_poz, "a is pozitive"};
      if ((a < 0) && (b > 0))
        return {a_b_oppoz, "a and b are oppozite"};
      if ((a == 0) && (h > a))
        return {a_null_proof, "a is null, approved."};
    }
  }
  post
  {
    if (aCov != aCov.a_null_proof)
    {
      //...
    }
  }
}
```

5.2.2 Итерация по элементам покрытия

Итерация по элементам покрытия — конструкция, помогающая обойти все ветви функциональности, описываемые покрытием. Параметрами итерации являются имя вызываемой спецификационной функции и имя покрытия. Элементы покрытия, обход которых по каким-то причинам нежелателен, можно отсеять с помощью фильтра.

Оператор итерации по элементам покрытия `iterate coverage` сходен с оператором итерации `iterate`: значение итерационной переменной глобально по отношению к сценарию, содержащему оператор.

```
iterate coverage
        ( f.PointCoverage : f.PointCoverage[0] != IntCoverage.ZERO )
{
   //...
}
```

В отличие от оператора `iterate` отсутствуют конструкции управления итерационной переменной: в итерации по покрытиям учет обработанных элементов покрытия осуществляется тестовой системой.

5.2.3 Занесение информации об элементе покрытия в отчёт

Операции над элементами покрытий реализуют базовые возможности управления ходом тестирования и отражения его результатов в отчёте.

Параметром предопределенной функции `trace`, которая служит инструментом сбора информации об элементах покрытия, должен быть собственно элемент покрытия, заданный любым способом:

- явно указанный пользователем;
- полученный в качестве результата работы функции вычисления покрытия;
- полученный в качестве результата работы операции получения компонента элемента покрытия.

```
// сброс в трассу указанного элемента перечислимого покрытия
trace ( Cov.C1 );

// сброс в трассу указанного элемента покрытия-произведения
trace ( ProdCov.C1.C2 );
```

5.2.4 Хранение элементов покрытия в переменных типа `CoverageElement`

SeC позволяет хранить элементы покрытий в переменных специального типа `CoverageElement`.

Объявление переменной-элемента покрытия

```
coverage IntCoverage zero = IntCoverage( 0 );
```

После ключевого слова coverage указывается имя покрытия, элементы которого может содержать объявленная таким образом переменная. Объявление переменной-элемента покрытия может содержать инициализатор: выражение, возвращающее элемент указанного покрытия.

В остальном такие переменные подчиняются правилам употребления переменных языка C.

6 Медиаторы

Медиаторы предназначены для связывания спецификации с реализацией тестируемой системы или со спецификацией другого уровня абстрактности.

Медиаторы решают задачу синхронизации состояния спецификационной модели данных с состоянием тестируемой системы.

Задача преобразования модельного представления тестового воздействия в реализационное представление и реализационного — в модельное имеет значение только для медиаторов стимулов.

В SeC медиаторы реализуются как медиаторные функции и сборщики отложенных реакций. В медиаторных функциях специально выделены блоки кода, отвечающие за синхронизацию состояний и преобразование модельных-реализационных воздействий.

6.1 Медиаторная функция

Медиаторные функции предназначены для реализации медиаторов. Каждая медиаторная функция связывает спецификационную функцию или отложенную реакцию с частью реализации тестируемой системы или ее спецификации другого уровня абстрактности.

Медиаторные функции помечаются ключевыми словами SeC `mediator for`, между которыми должен содержаться уникальный идентификатор — имя медиаторной функции. Каждая медиаторная функция соответствует некоторой спецификационной функции или отложенной реакции, сигнатура и ограничения доступа которой должны указываться в предварительных декларациях и определении медиаторной функции.

Медиаторная функция может содержать:

1. Блок воздействия. В блоке воздействия осуществляется тестовое воздействие с преобразованием данных из модельного представления в реализационное и обратно. Блок воздействия обязателен для медиаторов спецификационных функций и отсутствует в медиаторах отложенных реакций (в случае отложенных реакций воздействие инициируется тестируемой системой).
2. Блок синхронизации. Приводит модельное состояние в соответствие с состоянием реализации тестируемой системы. Блок синхронизации может отсутствовать в медиаторах спецификационных функций и обязателен для медиаторов отложенных реакций.
3. Дополнительный код перед первым специальным блоком и после второго. Дополнительный код не должен иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри блока кода, должна освобождаться либо в нем же, либо в парном ему блоке.

Медиаторная функция связывается со спецификационной функцией (или отложенной реакцией) с помощью функции `set_mediator_имя_спецификационной_функции` (или `set_mediator_имя_отложенной_реакции`), принимающей указатель на медиаторную функцию. Обычно связывание осуществляется в функции инициализации сценария:

```
set_mediator_push_spec(push_media);
```

6.1.1 Блок воздействия медиаторной функции

Блоки воздействия медиаторов спецификационных функций предназначены для реализации поведения, описанного в этих спецификационных функциях, посредством оказания воздействий на тестируемую систему.

В блоке воздействия осуществляется тестовое воздействие с преобразованием данных из модельного представления в реализационное и обратно.

Блоки воздействия используются только в медиаторах спецификационных функций. В нем выполняются следующие действия:

- параметры спецификационной функции преобразуются в реализационное представление;
- вызывается интерфейсная функция (или несколько функций) тестируемой системы;
- результат вызова интерфейсной функции и выходные параметры преобразуются из реа-лизационного представления в модельное;
- модельное представление результата возвращается из блока воздействия

Блок воздействия записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом `call`. Эти инструкции представляют собой тело функции, имеющей те же параметры и тип возвращаемого результата, что и соответствующая спецификационная функция.

Медиатор для функции выборки элемента стека

```
stack *stack_impl;
List*  stack_model;

int push(stack*, int);
specification bool push_spec(Integer* i) reads i updates stack_model;

mediator push_media for
  specification bool push_spec(Integer* i) reads i updates
stack_model
{
  call {
    return (bool)push(stack_impl, value_Integer(i));
  }
  //...
}
```

6.1.2 Блок синхронизации медиаторной функции

Блок синхронизации приводит модельное состояние в соответствие с состоянием реализации тестируемой системы.

Блок синхронизации приводит модельное состояние в соответствие с состоянием реализации тестируемой системы после осуществления воздействия или возникновения отложенной реакции. Блок синхронизации записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом `state`. Эти инструкции представляют собой тело функции без возвращаемого результата, имеющей те же параметры, что и соответствующая спецификационная функция или отложенная реакция. Если соответствующая спецификационная функция или отложенная реакция имеет тип возвращаемого значения, отличный от `void`, то доступ к этому значению можно получить через идентификатор этой функции (реакции).

При тестировании систем с открытым состоянием, когда тестовая система имеет доступ к внутренним данным реализации, блок синхронизации должен привести модельное состояние в соответствие с реализационным:

```
stack *stack_impl;
List*  stack_model;

int push(stack*, int);
specification bool push_spec(Integer* i) reads i updates stack_model;

mediator push_media for
specification bool push_spec(Integer* i) reads i updates stack_model
{
  ...
  state {
    int k;
    clear_List(stack_model);
    for( k = stack_impl->size;
         k > 0;
         append_List( stack_model
                    , create_Integer(stack_impl->elems[--k])
```

```
                    )
        );
    }
}
```

Поскольку код построения модельного состояния по внутреннему состоянию реализации будет одинаков для всех спецификационных функций, его удобно вынести в отдельную функцию

При тестировании систем со скрытым состоянием, когда тестирующая система не имеет доступа к внутренним данным реализации, блок синхронизации должен действовать в предположении, что реализация отработала без ошибок в соответствии со спецификацией, и привести модельное состояние в вид, который ожидается получить после воздействия:

```
mediator push_media for
  specification bool push_spec(Integer* i) reads i updates
stack_model
{
  ...
  state {
    add_List(stack_model, 0, create_Integer(push_spec));
  }
}
```

Блок синхронизации спецификационных функций выполняется тестовой системой непосредственно после блока воздействия перед проверкой постусловия в спецификационной функции.

Блок синхронизации отложенных реакций выполняется после возникновения отложенной реакции перед проверкой постусловия.

6.2 Сборщик реакций

Сборщик реакций служит для получения результата отложенных реакций.

Сборщики реакций являются реализационно-зависимыми компонентами. Их задача состоит в том, чтобы собрать все отложенные реакции целевой системы и зарегистрировать их в регистраторе взаимодействий.

7 Тестовые сценарии

Тестовые сценарии определяют исходные данные для построения тестов. Каждый тест представляет собой последовательность тестовых воздействий, выполнение которой обеспечивает решение некоторой задачи тестирования. Обычно в качестве такой задачи выступает тестирование поведения системы при воздействиях на нее через некоторый набор интерфейсных функций реализации до достижения заданного уровня покрытия в соответствии с критериями покрытия спецификации.

Обычная задача тестирования состоит в проверке поведения системы при воздействиях на нее через набор интерфейсных функций до достижения заданного уровня покрытия. Последовательность тестовых воздействий для решения такой задачи называется тестом.

Вызовы одной или нескольких спецификационных функций и способ перебора их параметров задаются в сценарной функции. Во время тестирования тестовая система находится в одном из состояний, называемых сценарными состояниями. Каждый вызов сценарной функции переводит тестовую систему из одного состояния в другое. Автоматически обеспечивается перебор всех параметров в каждом достигнутом сценарном состоянии.

Тестовый сценарий объединяет несколько сценарных функций, указывает механизм построения теста, задает функцию построения сценарного состояния, определяет способ инициализации и завершения работы тестовой и тестируемой систем.

На основании данных, содержащихся в тестовом сценарии, соответствующий тест генерируется автоматически.

7.1 Создание тестового сценария и параметры его вызова

Тестовый сценарий предоставляет всю информацию, необходимую для автоматического построения теста. Он соответствует переменной специального структурного типа `dfsm` или `ndfsm`, помеченной модификатором `scenario`:

```
scenario dfsm testScenario;
```

В качестве имени типа выступает название обходчика, определяющего механизм построения теста:

- `dfsm` — Deterministic Finite State Machine (детерминированный конечный автомат);
- `ndfsm` — Nondeterministic Finite State Machine (недетерминированный конечный автомат).

Во время тестирования `dfsm` применяет тестовые воздействия, которые могут изменять состояние сценария. `dfsm` автоматически отслеживает все изменения состояния и строит конечный автомат, соответствующий процессу тестирования. Состояниями автомата являются все достижимые состояния сценария, а переходы автомата помечаются тестовыми воздействиями, которые их инициировали. Тестирование заканчивается только тогда, когда обходчик подаст все определенные пользователем тестовые воздействия во всех состояниях сценария достижимых из начального.

Чтобы достижение этой цели было возможно, необходимо выполнение следующих условий:

- **Конечность**. Число состояний, достижимых из начального путем применения тестовых воздействий из заданного набора, должно быть конечным.
- **Детерминированность**. Для любого состояния системы применение одного и того же тестового воздействия всегда должно переводить систему в одно и тоже состояние.
- **Сильная связность**. Из любого состояния сценария достижимо любое другое состояние сценария путем применения последовательности тестовых воздействий.

Обходчик ndfsm по сравнению с dfsm позволяет корректно работать на более широком классе автоматов, а именно на конечных атоматах, имеющих детерминированный сильносвязный полный остовный подавтомат.

- **Остовный подавтомат**. Остовный подавтомат содержит все состояния сценария, достижимые в процессе тестирования.
- **Полный подавтомат**. Для каждого состояния сценария и допустимого в нем тестового воздействия подавомат либо содержит все переходы из этого состояния, помеченные этим тестовым воздействием, либо не содержит таких переходов вовсе.

Обходчик `ndfsm` не предназначен для тестирования систем с отложенными реакциями.

Определение тестового сценария должно содержать инициализатор следующего вида:

```
scenario dfsm testScenario = {
  .init     = init,
  .getState = getState,
  .actions  = {
    f_scen,
    g_scen,
    NULL
  },
```

```
    .finish   = finish
};
```

В поле *init* задается функция инициализации. Это поле может быть опущено, однако обычно функция инициализации присутствует как минимум для установки медиаторов.

В поле *getState* задается функция построения сценарного состояния. Если это поле опущено, то тестирование ведется в одном состоянии.

В поле *actions* задается список сценарных функций, включенных в данный тест, завершенный значением NULL. Это поле является обязательным.

В поле *finish* задается функция завершения. Это поле может быть опущено.

Тестовый сценарий вызывается как функция с идентификатором тестового сценария, принимающая два параметра, аналогичных параметрам main, без возвращаемого значения. Обычно вызов производится в функции main:

```
int main(int argc, char **argv) {
  testScenario(argc,argv);
  return 0;
}
```

При вызове тестового сценария происходит разбор переданных ему параметров. Тестовая система обрабатывает следующие стандартные параметры:

-tc

Направить трассировку на консоль.

-tt

Направить трассировку в файл с уникальным именем, составленным из имени сценария и текущего времени.

Запуск теста без указанных параметров командной строки эквивалентен запуску с параметром -tt.

-t имя_файла

Направить трассировку в файл с указанным именем.

-nt

Выключить трассировку.

--trace-accidental

Включить трассировку недостоверных переходов.

-uerr

Выполнять тестирование до возникновения первой ошибки (по умолчанию).

-uerr=число

Выполнять тестирования до возникновения *число* ошибок (только для ndfsm).

-uend

Выполнять тестирование до конца, несмотря на ошибки.

--find-first-series-only, -ffso

Находить только первую успешную серию.

--trace-format

Формат сброса данных в трассу.

--disabled-actions

Список игнорируемых сценарных функций.

Обработанные стандартные параметры удаляются из списка параметров, и измененный список передается функции инициализации сценария.

Допускается несколько вызовов сценариев из одной программы. Следует заметить, что если параметры, переданные исполняемому файлу теста из командной строки, будут просто переданы во все вызываемые сценарии, то при направлении трассировки в файл трасса очередного сценария перезапишется в тот же файл, что и трасса предыдущего. Для того, чтобы в одном файле оказались трассы всех сценариев, следует воспользоваться функциями `addTraceToFile` и `removeTraceToFile`.

```
int main(int argc, char **argv) {
  addTraceToFile("trace.utt");
  testScenario1(argc,argv);
  testScenario2(argc,argv);
  removeTraceToFile("trace.utt");
  return 0;
}
```

В случае тестирования систем с отложенными реакциями, в определении тестового сценария требуется задать большее число полей:

```
scenario dfsm testScenario = {
  .init     = init,
  .getState = getState,
  .isStationaryState = isStationaryState,
  .saveModelState    = saveModelState,
  .restoreModelState = restoreModelState,
  .actions           = {
    f_scen,
    g_scen,
    NULL
  },
  .finish            = finish
};
```

Три дополнительных поля для сценариев с отложенными реакциями являются обязательными.

- В поле *isStationaryState* задается функция определения стационарности состояния.
- В поле *saveModelState* задается функция сохранения модельного состояния.
- В поле *restoreModelState* задается функция восстановления модельного состояния.

7.1.1 Функция инициализации

Функция инициализации предназначена для выполнения подготовительных работ перед проведением тестирования.

Функция принимает два параметра, аналогичных параметрам функции `main`, а возвращает логическое значение, показывающее, успешно ли проведена инициализация:

```
typedef bool (*PtrInit)( int, char** );
```

Обычно функция инициализации выполняет следующие действия:

- инициализация тестируемой системы;
- инициализация спецификационной модели данных;
- инициализация данных сценария;
- установка медиаторов для спецификационных функций и реакций, используемых в данном тестовом сценарии.

При необходимости, функция инициализации может использовать переданные ей параметры запуска.

```
char *impl_data;
String* model_data;

specification void f_spec(int a);
mediator f_media for specification void f_spec(int a);

bool init(int argc, char **argv) {
  impl_data  = (char*)malloc(SIZE);
  model_data = create_String("");
  set_mediator_f_spec(f_media);
  return (impl_date != NULL && model_data != NULL);
}
```

В случае тестирования системы с отложенными реакциями функция инициализации дополнителько используется для:

- установки времени ожидания отложенных реакций
-

```
ChannelID chid;
bool init(int argc, char **argv) {
  chid = getChannelID();
  setStimulusChannel(chid);
  setWTime(1);
  ...
}
```

7.1.2 Функция построения сценарного состояния

Во время работы тестовая система находится в одном из состояний. Функция построения сценарного состояния обычно вычисляет это состояние на основе значений переменных, определяющих модельное состояние. Сценарное состояние часто является обобщением модельного состояния, однако оно может и совпадать с модельным, а может быть вообще с ним не связано.

Функция вычисления состояния сценария не имеет параметров и возвращает объект спецификационного типа.

```
typedef Object* (*PtrGetState)( void );
```

Пример функции, обобщающей модельное состояние, представляющее собой список, как его длину:

```
List* l;

Object* getState(void) {
  return create_Integer(size_List(l));
}
```

7.1.3 Функция завершения сценария

Функция завершения предназначена для выполнения заключительных работ после проведения тестирования. Функция не имеет параметров и возвращаемого значения.

```
typedef void (*PtrFinish)( void );
```

Как правило, функция завершения освобождает ресурсы, выделенные в функции инициализации.

```
char *impl_data;
String* model_data;

void finish( void ) {
  free(impl_data);
  model_data = NULL;
  releaseChannelID(chid);
}
```

7.1.4 Функция определения стационарности состояния

Функция определения стационарности состояния используется только при тестировании систем с отложенными реакциями.

Функция не имеет параметров и возвращает логическое значение, показывающее, является ли текущее модельное состояние стационарным (стационарным называется состояние, в котором не ожидается возникновения отложенных реакций).

```
typedef bool (*PtrIsStationaryState)(void);
```

Если модельное состояние представлено переменной неспецификационного типа или несколькими переменными, потребуется определить новый спецификационный тип, включающий в себя все необходимые данные.

```
List* expectedReactions;
...
bool isStationaryState() {
  return empty_List(expectedReactions);
}
```

7.1.5 Функция сохранения модельного состояния

Функция определения стационарности состояния используется только при тестировании систем с отложенными реакциями.

Функция не имеет параметров и возвращает логическое значение, показывающее, является ли текущее модельное состояние стационарным (*стационарным* называется состояние, в котором не ожидается возникновения отложенных реакций).

```
typedef Object* (*PtrSaveModelState)( void );
```

Пример:

```
List* modelList;
int   modelInt;
specification typedef struct {
  List* a;
  int b;
} ModelState = {};
...
Object* saveModelState() {
```

```
  return create(&type_ModelState, clone(modelList), modelInt);
}
```

7.1.6 Функция восстановления модельного состояния

Функция определения стационарности состояния используется только при тестировании систем с отложенными реакциями.

Функция принимает один параметр — ссылку на значение спецификационного типа:

```
typedef void (*PtrRestoreModelState)( Object* );
```

На вход функции восстановления модельного состояния подается значение, возвращенное функцией сохранения модельного состояния. Задача функции состоит в том, чтобы привести текущее модельное состояние системы в соответствие с этим значением.

```
List* modelList;
int   modelInt;

specification typedef struct {
  List* a;
  int b;
} ModelState = {};

...

void restoreModelState(Object* modelState) {
  ModelState* s = (ModelState*)modelState;
  modelList = clone(s->a);
  modelInt = s->b;
}
```

7.2 Сценарные функции

Сценарные функции предназначены для описания наборов тестовых воздействий. Для этого сценарная функция определяет воздействия (вызовы спецификационных функций) и способ перебора их параметров. Все воздействия будут автоматически выполнены в каждом сценарном состоянии, достигнутом во время тестирования. Также сценарные функции могут производить дополнительную проверку корректности поведения вызываемых функций.

Сценарная функция описывается как функция без параметров, возвращающая значение типа `bool` и помеченная ключевым словом `scenario`:

```
scenario bool f_scen();
```

В простейшем случае каждая сценарная функция соответствует ровно одной спецификационной функции. Если спецификационная функция не имеет аргументов, то тело сценарной функции может содержать единственный вызов этой спецификационной функции. Перебор значений аргументов спецификационной функции удобно осуществлять с помощью оператора итерации.

Сценарная функция должна возвратить `false`, если поведение системы некорректно, и `true` в нормальной ситуации. При этом нужно учесть, что проверка постусловий вызываемых спецификационных функций происходит автоматически и ее результаты учитывать не надо, то есть, проверка в сценарной функции носит дополнительных характер.

```
specification int f_spec(void);
scenario bool f_scen() {
  f_spec();
  return true;
}
```

7.2.1 Оператор итерации

Оператор итерации используется только в сценарной функции и служит для параметризованного (например, аргументами спецификационной функции) перебора тестовых воздействий.

Синтаксически оператор итерации схож с циклом `for`:

```
iterate (декларация; усл_продолжения; инкремент; усл_фильтрации)
тело;
```

Декларация является обязательным выражением и представляет собой декларацию итерационной переменной и ее инициализацию (в отличие от языка C, где переменная декларируется вне цикла). Итерационная переменная должна быть допустимого типа.

Так как вызов сценарной функции производится в каждом новом модельном состоянии, то время жизни итерационной переменной выходит за рамки времени выполнения сценарной функции, в которой она была объявлена: чтобы обеспечить эффективность инкремента каждому модельному состоянию системы должен соответствовать свой экземпляр итерационной переменной. При вызове сценарной функции в некотором состоянии, в качестве значения итерационной переменной используется то значение, которое эта переменная получила в предыдущий вызов в том же состоянии.

Условие продолжения и инкремент действуют так же, как аналогичные выражения цикла `for`.

Условие фильтрации представляет собой логическое выражение. Если оно равно `false`, то происходит переход к следующему значению итерационной переменной. Условие фильтрации можно опустить, в таком случае оно будет эквивалентно выражению `true`, т. е. ни одно значение итерационной переменной не будет отброшено.

Таким образом, следующая конструкция:

```
iterate (int i=0; i<10; i++;i&1==0) { ... }
```

в некотором смысле аналогична циклу `for`:

```
int i;
for (i=0; i<10; i++) {
  if (!(i&1==0)) continue;
  ...
}
```

Такой схемой можно пользоваться, чтобы представить последовательность вызовов, которая будет сгенерирована в пределах одного сценарного состояния. Однако следует понимать разницу между оператором итерации и циклом. Во-первых, значение итерационной переменной зависит от сценарного состояния, а значение переменной цикла — нет. Во-вторых, при одном вызове сценарной функции выполняется только одна итерация; она определяет переход из одного сценарного состояния в другое. Если же внутри сценарной функции будет использован обычный цикл, в рамках одного и того же перехода будут произведены все вызовы, перебираемые циклом.

Допускаются вложенные операторы итерации. Однако последовательные операторы итерации использовать нельзя.

Специально для перебора всех элементов какого-либо покрытия предназначен оператор итерации по покрытиям.

При тестировании без отложенных реакций сценарное состояние изменяется в процессе тестового воздействия, происходящего в момент вызова спецификационной функции. Таким образом, если в результате работы спецификационной функции изменяются глобальные переменные, после ее вызова в сценарной функции будут доступны уже измененные значения. Однако переменные состояния и итерационные переменные сохранят значения, соответствующие прежнему сценарному состоянию, до конца работы сценарной функции.

При тестировании с отложенными реакциями сценарное состояние не изменяется в ходе работы сценарной функции. Изменение происходит в конце работы сценарной функции, после того, как будут собраны все отложенные реакции.

7.2.2 Переменные сценарного состояния

Помимо итерационных переменных, существует и другой вид переменных, значение которых зависит от сценарного состояния — переменные сценарного состояния. Для каждого сценарного состояния имеется свой экземпляр такой переменной со своим значением. При вызове сценарной функции в некотором состоянии, в качестве значения переменной используется то значение, которое эта переменная получила в предыдущий вызов в том же состоянии.

Декларация переменных сценарного состояния начинается с модификатора `stable` и должна содержать инициализацию. Переменные сценарного состояния должны иметь допустимый тип.

```
stable int i = 0;
```

То, в каком месте сценарной функции объявлены такие переменные, влияет лишь на область их видимости, но не на функциональность.

8 Дополнительные возможности

8.1 Строковое и XML представления неспецификационных типов

Инструмент CTESK позволяет определять пользовательские функции формирования строковых и XML- представлений не только для спецификационных типов, но и для любых типов языка C, построенных при помощи конструкции `typedef`, а так же для типов с инвариантами. Формат и правила формирования XML представления значений точно такие же, что и для спецификационных типов.

Пользовательские функции формирования строкового и XML представления типа задаются следующим образом.

Пусть есть декларация именованного типа `T`

```
typedef struct { int x; int y } T;
```

Для переопределения стандартного способа построения строкового представления необходимо определить функцию

```
String* to_String_T( T* );
```

Для переопределения стандартного способа построения XML представления необходимо определить функцию

```
String* to_XML_T( T* );
```

Важно иметь ввиду, что заданные функции будут использоваться для трассировки не только объектов типа `T`, но и:

- объектов типа “указатель на `T`” (любого уровня):

  ```
  specification void f( T *t );
  ```

- массивов, элементом которых является `T` или указатель на `T`:

  ```
  specification void f( T t[10] );
  ```

- объектов типов-инвариантов, построенных на основе типа `T`:

  ```
  invariant typedef T InvT;
  specification void f( InvT *it );
  ```

Кроме того, пользовательские функции представления неспецификационных типов будут вызываться при трассировке значений полей типов, базовых для спецификационных:

```
specification typedef struct { String *str; T t; } S;
specification void f( S *s );
```

Но пользовательские фукнции **не** будут вызываться для не-инвариантных типов, построенных при помощи конструкции `typedef` “над” типами, для которых эти функции определены:

```
typedef T T2;
specification void f( T2 *t2 );
```

При трассировке значения `t2` функции `to_string_T` и `to_XML_T` использоваться **не** будут.

9 Семантика языка SeC

Язык SeC является расширением языка программирования C и предназначен для разработки тестов на основе формальных спецификаций. В нем добавлены специальные конструкции, которые позволяют компактным и удобным образом описывать требования к тестируемой системе и другие компоненты тестовой системы. Это делает разработку тестов максимально удобной, а также позволяет сократить затраты на обучение специалистов, уже знакомых с языком программирования C.

Язык SeC дополнительно вводит спецификационные типы (см. *Спецификационные типы*), инварианты типов данных и глобальных переменных, тестовые сценарии (см. *Тестовые сценарии*) и три вида функций: спецификационные (см. *Спецификационные функции*, *Отложенные реакции*), медиаторные (см. *Медиаторные функции*) и сценарные (см. *Сценарные функции*). Эти типы, инварианты и функции определяются в спецификационных файлах с расширением `sec`. Спецификационные заголовочные файлы, содержащие декларации спецификационных типов и функций, должны находиться в файлах с расширением `seh`. Спецификационные заголовочные файлы включаются в спецификационные файлы при помощи директивы препроцессора C `#include`. Спецификационные файлы могут содержать и обычные функции C, требуемые для различных вспомогательных целей. При необходимости использования специальных данных или функций используется включение соответствующих обычных заголовочных файлов языка C при помощи директивы препроцессора C `#include`.

Для удобства записи и чтения логических выражений в язык SeC дополнительно введен оператор импликации `=>`, являющийся бинарным инфиксным оператором, приоритет которого меньше приоритета оператора дизъюнкции `||`, но больше приоритета условного оператора `?:`. Выражение `x => y` эквивалентно выражению `!x || y`, и при его вычислении, также как при вычислении других логических операторов, действуют правила короткой логики. Оператор импликации ассоциативен слева направо, то есть выражение `x => y => z` эквивалентно выражению `(x => y) => z`.

9.1 Спецификации

Спецификации представляют собой формальное описание требований к тестируемой системе в форме инвариантов данных и спецификации поведения тестируемой системы. Для описания требований спецификации могут использовать как непосредственно данные тестируемой системы, так и собственную спецификационную модель данных.

9.1.1 Спецификационные функции

Спецификационные функции предназначены для спецификации поведения тестируемой системы при внешнем воздействии на нее через некоторую часть интерфейса. Спецификационная функция определяет:

- ограничения доступа к данным;
- предусловие;
- локальные критерии покрытия;
- постусловие.

Спецификационные функции декларируются и определяются в спецификационных файлах и помечаются ключевым словом SeC `specification`. Они могут содержать следующие элементы:

- Описание ограничений доступа данной функции к глобальным переменным и параметрам.
- Описание локальных покрытий (только в прототипах функций).
- Предусловие, описывающее ситуации, в которых определено поведение тестируемой системы.
- Локальные критерии покрытия, описывающие разбиение на ветви функциональности поведения тестируемой системы при взаимодействии с ней через часть интерфейса, описываемого данной спецификационной функцией.
- Постусловие, описывающее ограничения, которым должны удовлетворять результаты работы тестируемой системы, описываемой данной спецификационной функцией.
- Дополнительный код на SeC помимо предусловия, определения локальных критериев покрытия и постусловия.

Спецификационные функции вызываются так же как обычные функции. При тестировании в точке вызова спецификационной функции происходит проверка инвариантов типов выражений с доступом `reads` и `updates`, проверка инвариантов переменных с доступом `reads` и `updates`, проверка значений переданных аргументов на выполнение предусловия, вычисление элементов покрытий для значений переданных аргументов, обращение к медиатору, установленному для вызываемой спецификационной функции, проверка неизменности выражений с доступом `reads`, проверка инвариантов типов выражений с доступом `writes` и `updates`, проверка инвариантов переменных с доступом `writes` и `updates`, проверка выполнения постусловия.

Синтаксис

```
( declaration_specifiers )?
"specification"
( declaration_specifiers )?
declarator
( declaration )*
compound_statement
;
```

Семантические правила

1. Имена спецификационных функций входят в то же пространство имен, что и имена обычных функций языка C.
2. Спецификационная функция должна быть определена ровно один раз среди всех единиц трансляции (`translation_unit`), собираемых в одну систему.
3. В объявлении или определении спецификационной функции запрещается вводить любые сущности с именами, совпадающими с именем этой спецификационной функции.
4. Для всех глобальных переменных и параметров (или их составных частей), которые используются в спецификационной функции, во всех предварительных декларациях и декларатоpе определения функции должны быть указаны ограничения доступа (`se_access_description`).
5. Во всех предварительных декларациях и декларатоpе определения функции ограничения доступа должны быть одинаковы.
6. В составном операторе спецификационной функции должно присутствовать ровно одно постусловие (`se_post_block_statement`) после блоков критериев покрытия и предусловия, если они присутствуют.
7. В составном операторе спецификационной функции может присутствовать не больше одного предусловия (`se_pre_block_statement`) до блоков критериев покрытия, если они присутствуют, и до блока постусловия, после предусловия должен следовать либо составной оператор, либо первый критерий покрытия, либо постусловие.
8. В составном операторе спецификационной функции могут присутствовать несколько локальных критериев покрытия (`se_coverage_statement`, с инициализатором `se_coverage_derivation_initializer` либо `se_coverage_function_initializer`), следующих друг за другом после блока предусловия, если он присутствует, и до блока постусловия. После последнего критерия покрытия должен следовать либо составной оператор, либо постусловие.
9. Между конструкциями `se_coverage_statement` не должно быть других конструкций.
10. Конструкции `se_pre_block_statement`, `se_coverage_statement` и `se_post_block_statement` могут быть вложены только в синтаксические узлы `block_item` (блоки кода, ограниченные фигурными скобками).
11. С учетом возможного дополнительного кода, тело спецификационной функции должно иметь следующую структуру:

```
{
  дополнительный_код_1_1
  pre { ... }
  {
    дополнительный_код_2_1
    coverage имя_1 { ... }
    ...
```

```
      coverage имя_n { ... }
      {
        дополнительный_код_3_1
        post { ... }
        дополнительный_код_3_2
      }
      дополнительный_код_2_2
    }
    дополнительный_код_1_2
  }
```

12. Любой из блоков дополнительного кода может отсутствовать.
13. Если какая-либо из пар `дополнительный_код_2_1` и `дополнительный_код_2_2` или `дополнительный_код_3_1` и `дополнительный_код_3_2` отсутствует, то разрешается не писать фигурные скобки, непосредственно содержащие отсутствующую пару дополнительного кода.
14. Спецификационная функция должна быть без видимых снаружи побочных эффектов:
    14.1. значения глобальных переменных и значения данных, передаваемых по указателям, не должны изменяться;
    14.2. динамическая память, выделяемая в спецификационной функции, должна освобождаться, причем на том же уровне вложенности (в том же составном операторе), на котором она выделялась:
        14.2.1. если память выделяется в предусловии, критерии покрытия или постусловии, она должна освобождаться в том же блоке;
        14.2.2. если память выделяется в начале спецификационной функции, она должна освобождаться в конце после предусловия, критериев покрытия, постусловия и окончания составных операторов, содержащих критерии покрытия и постусловие;
        14.2.3. если память выделяется в составном операторе после предусловия, то она должна освобождаться в конце этого составного оператора после критериев покрытия, постусловия и окончания составного оператора, содержащего постусловие;
        14.2.4. если память выделяется в составном операторе перед постусловием, то она должна освобождаться в его конце после постусловия.

9.1.2 Отложенные реакции

Отложенные реакции объявляются и определяются в спецификационных файлах в виде функций без параметров, помеченных ключевым словом SeC `reaction` и возвращающих указатели на спецификационные типы. Они могут содержать следующие элементы:

- Описание ограничений доступа данной реакции к глобальным переменным.
- Предусловие, описывающее ситуации, в которых возможно возникновение данной реакции.
- Постусловие, описывающее ограничения на значения глобальных переменных, которые должны выполняться в случае возникновения данной реакции.
- Дополнительный код на SeC вне предусловия и постусловия.

Синтаксис

```
( declaration_specifiers )?
"reaction"
( declaration_specifiers )?
declarator
( declaration )*
compound_statement
;
```

Семантические правила

1. Имена отложенных реакций входят в то же пространство имен, что и имена обычных функций языка C.
2. Отложенная реакция должна быть определена ровно один раз среди всех единиц трансляции (`translation_unit`), собираемых в одну систему.
3. У отложенных реакций не может быть параметров, типом возвращаемого значения отложенной реакции может быть только указатель на спецификационный тип.
4. В объявлении или определении отложенной реакции запрещается вводить любые сущности с именами, совпадающими с именем этой отложенной реакции.
5. Для всех глобальных переменных и параметров (или их составных частей), которые используются в реакции, во всех предварительных декларациях и декларторе определения реакции должны быть указаны ограничения доступа (`se_access_description`).
6. Во всех предварительных декларациях и декларторе определения реакции ограничения доступа должны быть одинаковы.
7. В составном операторе отложенной реакции должно присутствовать ровно одно постусловие (`se_post_block_statement`) после предусловия, если оно присутствует.
8. В составном операторе отложенной реакции может присутствовать не больше одного предусловия (`se_pre_block_statement`) до постусловия, после предусловия должен следовать либо составной оператор, либо постусловие.
9. С учетом возможного дополнительного кода, тело отложенной реакции должно иметь следующую структуру:

```
{
  дополнительный_код_1_1
  pre { ... }
  {
    дополнительный_код_2_1
    post { ... }
    дополнительный_код_2_2
  }
  дополнительный_код_1_2
}
```

10. Любой из блоков дополнительного кода может отсутствовать.
11. Если пара `дополнительный_код_2_1` и `дополнительный_код_2_2` отсутствует, то разрешается не писать фигурные скобки, непосредственно её содержащие.
12. Отложенная реакция должна быть без видимых снаружи побочных эффектов:

    12.1.значения глобальных переменных не должны изменяться;

12.2. динамическая память, выделяемая в реакции, должна освобождаться, причем на том же уровне вложенности (в том же составном операторе), на котором она выделялась:

12.2.1. если память выделяется в предусловии или постусловии, она должна освобождаться в том же блоке;

12.2.2. если память выделяется в начале реакции, она должна освобождаться в конце после предусловия, постусловия и окончания составного оператора, содержащего постусловие;

12.2.3. если память выделяется в составном операторе после предусловия, то она должна освобождаться в конце этого составного оператора после постусловия;

9.1.3 Ограничения доступа

Ограничения доступа описываются после сигнатуры функции или реакции. Для указания доступа на чтение используется модификатор `reads`, на запись — `writes`, на изменение — `updates`. Действие модификатора доступа распространяется на все перечисленные вслед за ним через запятую идентификаторы до следующего модификатора или начала тела функции или реакции. В ограничениях доступа можно объявлять псевдонимы выражений, на которые описываются ограничения доступа.

Если для выражения указано ограничение доступа на изменение (модификатор `updates`), то внутри функции или реакции до ключевого слова `post` выражение имеет пре-значение — значение до взаимодействия с тестируемой системой, описываемого данной спецификационной функцией, или значение до возникновения данной реакции. После ключевого слова `post` выражение имеет пост-значение — значение после взаимодействия с тестируемой системой или после возникновения реакции, пре-значение выражения после ключевого слова `post` доступно через оператор SeC `@`.

Синтаксис

```
se_access_description ::= se_access_specifier se_access
                          ( "," se_access )*
                      ;
se_access_specifier ::= "reads" | "writes" | "updates" ;
se_access ::= ( se_access_alias )? assignment_expr ;
```

Семантические правила

1. В пределах одного описания не должно быть двух ограничений доступа, имеющих разные модификаторы доступа и одно и то же выражение, к которому указывается доступ.
2. Если для выражения указано ограничение доступа на запись (модификатор `writes`), то внутри функции или реакции не допускается использование этого выражения до ключевого слова `post`.
3. Параметры функции не могут использоваться в качестве выражений с ограничением доступа на запись (модификатор `writes`).
4. Если для выражения указано ограничение доступа на чтение (модификатор `reads`), то внутри функции или реакции его значение доступно везде и не меняется.

9.1.4 Псевдонимы

Псевдонимы объявляются простым присваиванием в ограничениях доступа. Внутри функции или реакции псевдонимы используются аналогично локальным переменным.

Синтаксис

```
se_access_alias ::= <ID> "=" ;
```

Семантические правила

1. Идентификатор псевдонима не должен совпадать с идентификаторами параметров, с именами локальных покрытий и должен быть уникальным в пределах ограничений доступа спецификационной функции или отложенной реакции.

9.1.5 Предусловие

Предусловие отложенной реакции описывает условия, при которых возможно возникновение данной реакции. Во время тестирования предусловие реакции проверяется всякий раз, когда она возникает. Если предусловие нарушается, это значит, что поведение системы не соответствует спецификации.

Предусловие на языке SeC — это набор инструкций, который синтаксически и семантически эквивалентен телу функции, имеющей те же параметры, что и спецификационная функция (у реакций параметры всегда отсутствуют) и возвращающей результат типа `bool`. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом `pre`.

Синтаксис

```
se_pre_block_statement ::= "pre" compound_statement ;
```

Семантические правила

1. В спецификационной функции или отложенной реакции может быть не больше одного предусловия, которое должно быть определено до локальных критериев покрытия (если они определяются в функции) и постусловия.
2. Предусловие может быть опущено, отсутствие предусловия эквивалентно наличию предусловия, всегда возвращающего `true`.
3. Предусловие не должно иметь побочных эффектов, то есть оно не должно изменять значения глобальных переменных и данных, переданных через указатели.
4. Набор инструкций в предусловии должен быть синтаксически и семантически эквивалентно телу функции, имеющей ту же сигнатуру, что и спецификационная функция или отложенная реакция, частью определения которой оно является, и возвращающей результат типа `bool`.
5. В предусловиях нельзя использовать выражения с ограничением доступа на запись, то есть описанные в ограничениях доступа с модификатором `writes`.

9.1.6 Постусловие

Постусловие спецификационной функции служит для описания ограничений, которым должны удовлетворять результаты работы тестируемой системы при взаимодействиях с ней через часть интерфейса, описываемого данной функцией. Во время тестирования постусловие проверяется всякий раз после работы тестируемой системы при

соответствующих взаимодействиях, и, если постусловие нарушено, значит поведение системы не соответствует спецификации.

Постусловие отложенной реакции описывает ограничения, которым должны удовлетворять значения самой реакции и глобальных переменных после возникновения данной реакции. Если во время тестирования после возникновения реакции и работы ее медиатора постусловие нарушено, значит поведение системы не соответствует спецификации.

Постусловие на языке SeC — это набор инструкций, который синтаксически и семантически эквивалентен телу функции, имеющей те же параметры, что и спецификационная функция (у реакций параметры всегда отсутствуют) и возвращающей результат типа `bool`. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом `post`.

Синтаксис

```
se_post_block_statement ::= "post" compound_statement ;
```

Семантические правила

1. В спецификационной функции или отложенной реакции всегда должно быть ровно одно постусловие.
2. Постусловие определяется после предусловия (если оно не опущено) и после последнего локального критерия покрытия (если они определяются).
3. Набор инструкций в постусловие должен быть синтаксически и семантически эквивалентен телу функции, имеющей те же параметры, что и спецификационная функция или реакция (у реакций параметры всегда отсутствуют), частью определения которой оно является, и возвращающей результат типа `bool`.
4. Постусловие не должно иметь побочных эффектов, то есть оно не должно изменять значения глобальных переменных и данных, переданных через указатели.
5. В постусловии и после него в коде функции или реакции можно использовать следующие дополнительные конструкции:
   5.1. Превыражения с оператором `@`.
   5.2. Идентификатор, совпадающим с именем отложенной реакции или спецификационной функции (только для функций и реакций, имеющих тип результата не `void`), через который можно получить доступ к значению возникшей отложенной реакциии или результата, возвращаемого спецификационной функцией.
   5.3. Обращения к ранее вычисленным элементам локальных покрытий (если локальные покрытия определены в спецификационной функции).
   5.4. Псевдопеременная *`timestamp`*, имеющая тип `TimeInterval`, которая содержит временные метки начала и конца исполнения вызова реализации медиатора данной спецификационной функции или временные метки указанные при регистрации данной отложенной реакции.

9.1.7 Превыражения

Превыражения используются при спецификации совместных ограничений на состояния тестируемой системы до и после тестового воздействия или до и после возникновения отложенной реакции.

Синтаксис

```
"@" cast_expr ;


unary_expr ::=   postfix_expr
               | "++" unary_expr
               | "--" unary_expr
               | unary_operator cast_expr
               | "sizeof" unary_expr
               | "sizeof" "(" type_name ")"
               | gcc_extension_specifier cast_expr
           ;
unary_operator ::= "&" | "*" | "+" | "-" | "~" | "!" | "@" ;
cast_expr ::=   unary_expr
              | "(" type_name ")" cast_expr
              ;
```

Семантические правила

1. Оператор `@` может использоваться только после ключевого слова `post` по потоку управления.
2. Превыражение должно быть вычислимо до постусловия по потоку управления, в котором оно используется.
3. Превыражения не должны быть вложены друг в друга.

9.2 Спецификационные типы

Спецификационные типы вводятся с помощью обычной конструкции языка C `typedef`, помеченной ключевым словом SeC `specification`. В данной конструкции может встречаться несколько деклараций с инициализаторами. Каждая из них вводит новый спецификационный тип со своим именем. Если в декларации инициализатор отсутствует, то это предварительная декларация спецификационного типа, в противном случае — это определение спецификационного типа.

Значения спецификационных типов размещаются в динамической памяти, управление памятью автоматизировано — при изменении количества ссылок на значение, счетчик ссылок на данное значение автоматически изменяется, значение удаляется автоматически после уничтожения последней ссылки.

В спецификационном расширении языка C существует встроенный спецификационный тип `Object`, который является неполным спецификационным типом (incomplete specification type). Он является базовым для всех спецификационных типов, но объектов этого типа быть не может. Тип ссылки на спецификационный тип `Object`, то есть `Object*`, используется по аналогии с `void*`. Любая ссылка на спецификационный тип может быть преобразована к ссылке на `Object` и наоборот. Если при обратном преобразовании тип объекта по ссылке не совместим с типом, к которому осуществляется преобразование, то поведение системы не определено.

Синтаксис

```
declaration ::= (  (( declaration_specifiers )?
                    "specification"
                    ( declaration_specifiers )?
                    "typedef"
                    ( declaration_specifiers )?
                   )
                 | (( declaration_specifiers )?
                    "typedef"
                    ( declaration_specifiers )?
                    "specification"
                    ( declaration_specifiers )?
                   )
                )
                ( init_declarator  ( "," init_declarator )* )?
                ";"
              ;
```

Семантические правила

1. Спецификационные типы не могут быть локальными.
2. Имена спецификационных типов входят в то же пространство имен, что и typedef-имена.
3. Определение спецификационного типа с данным именем (декларатор с инициализатором) может встречаться только один раз во всех единицах трансляции (`translation_unit`), собираемых в единую систему.
4. Инициализатор в определении спецификационного типа должен иметь вид: `= { .<field(1)> = <expr>, .<field(2)> = <expr>, ... }`, где между фигурными скобками находится (возможно пустой) набор конструкций вида `.<field(i)> = <expr>`, разделенных запятыми. `<field(i)>` может принимать одно из следующих

значений: `init`, `copy`, `compare`, `to_string`, `to_XML`, `enumerate` или `destroy`. `<expr>` должно иметь тип, соответствующий указанному полю:

```
/* Тип функции, инициализирующей объект
   (инициализирует объект по указателю 'ref') */
typedef void (*Init)( void* ref, va_list* arg_list );

/* Тип функции, копирующей объект
   (копирует содержимое объекта по указателю 'src' в 'dst') */
typedef void (*Copy)( void* src, void* dst );

/* Тип функции, сравнивающей объекты
   (сравнивает объекты по указателям left и right) */
typedef int (*Compare)( void* left, void* right );

/* Тип функции, преобразующей объект в строку
   (строит строковое представление данного объекта */
typedef String* (*ToString)( void* obj );

/* Тип функции, формирующей XML представление заданного объекта */
typedef String* (*to_XML)( void* ref );

/* Тип функции, перебирающей подобъекты
   (вызывает заданную функцию callback для каждого объекта,
    принадлежащего данному) */
typedef void (*Enumerate)
  ( void* obj, void (*callback)( void* ref, void* par ),
    void* par
  );

/* Тип функции, уничтожающей объект
   (освобождает ресурсы, захваченный данным объектом) */
typedef void (*Destroy)( void* obj );
```

Подробную информацию о функциях, указателями на которые должны быть инициализированы поля при определении спецификационного типа, можно найти в главе «Библиотека спецификационных типов».

5. Если инициализация поля в определении спецификационного типа отсутствует, то используется функция по умолчанию.

В функциях по умолчанию указатели на все типы кроме спецификационных, функциональных и `void` интерпретируются как указатели на единственное значение этого типа. То есть, в том случае, когда базовый тип является указателем на несколько значений (строка, массив), в определении спецификационного типа должны быть инициализированы все поля.

6. В качестве базового типа при определении спецификационного типа запрещается использовать:

   6.1. спецификационные, функциональные и неполные типы;

   6.2. объединения, массивы, и структуры, содержащие вышеперечисленные типы.

Если в определении спецификационного типа отсутствует инициализация хотя бы одного поля, то использовать в качестве базового типа также запрещается:

   6.3. объединения и массивы переменной длины,

   6.4. структуры и массивы, содержащие все вышеперечисленные типы,

   6.5. указатели на любые объединения, массивы и структуры.

7. Если в декларации спецификационного типа присутствует ключевое слово `invariant`, то должен быть определен инвариант типа, ограничивающий множество его значений

8. Все базовые типы, указанные в декларациях и определении спецификационного типа, должны быть совместимы между собой.
9. Если в хотя бы в одной декларации спецификационного типа присутствует ключевое слово `invariant`, то оно должно присутствовать во всех декларациях, а также в определении данного типа.
10. Спецификационные типы, аналогично неполным типам языка C, могут использоваться только через указатели. То есть, не допускаются:
    10.1. объявления переменных спецификационных типов;
    10.2. определения объединений, структур и массивов, содержащих поля и элементы спецификационных типов и т. д.

9.3 Инвариант типа

Для декларации инвариантов типов используется обычная конструкция языка C `typedef`, помеченная ключевым словом SeC `invariant`. Данная конструкция, также как и обычный `typedef`, вводит новое имя типа. Основное ее отличие состоит в том, что множество значений определяемого типа не совпадает со множеством значений базового типа, а является его подмножеством. Таким образом, определяется не синоним для базового типового выражения, а новый тип с собственным множеством значений.

Ограничения на множество значений базового типа описываются в составном операторе (`compound_statement`), который синтаксически и семантически эквивалентен телу функции без побочных эффектов, возвращающей значение булевского типа, с одним параметром, который должен быть:

- определяемого подтипа, если базовый тип является обычным типом языка C;
- указателем на определяемый подтип спецификационного типа, если базовый тип является спецификационным типом.

Составной оператор инварианта помечается модификатором `invariant`, после которого в скобках указывается формальный параметр соответствующего типа. Поскольку тип возвращаемого значения фиксирован, он не указывается.

Инвариант типа может быть вызван для выражения соответствующего типа. Выражение вызова инварианта типа состоит из ключевого слова `invariant`, за которым в скобках следует выражение, для значения которого проверяется выполнение инварианта. Значением данного выражения вызова инварианта типа является `true`, если значение проверяемого выражения в точке вызова удовлетворяет ограничениям инварианта, и `false` — в обратном случае.

Синтаксис

```
declaration ::=   (( declaration_specifiers )?
                     "invariant"
                     ( declaration_specifiers )?
                     "typedef"
                     ( declaration_specifiers )?
                   )
                | (( declaration_specifiers )?
                     "typedef"
                     ( declaration_specifiers )?
                     "invariant"
                     ( declaration_specifiers )?
                   )
                ( init_declarator  ( "," init_declarator )* )?
                ";"
            ;
"invariant" "(" parameter_declaration ")" compound_statement
"invariant" "(" assignment_expr ")"
```

Семантические правила

1. Инвариант типа не может быть определен для локального типа.
2. До точки определения или вызова инварианта тип должен быть объявлен при помощи конструкции `typedef`, причем среди списка спецификаторов его декларации

(`declaration_specifiers`) должен встречаться спецификатор SeC (`se_declaration_specifier`) `invariant`.

3. Если в одной из единиц трансляции (`translation_unit`) при определении типа среди спецификаторов декларации (`declaration_specifiers`) встречается спецификатор SeC (`se_declaration_specifier`) `invariant`, то он должен встречаться в объявлениях этого типа во всех единицах трансляции (`translation_unit`), собираемых в единую систему.
4. Определение инварианта этого типа должно встречаться ровно один раз во всех единицах трансляции (`translation_unit`), собираемых в единую систему.
5. В качестве базового типа при определении типа с инвариантом запрещается использовать функциональные типы и `void`.

Если базовый тип в определении типа с инвариантом является спецификационным типом, то в определении инварианта параметр декларируется как указатель на определяемый тип. При этом пользователь может быть уверен, что в теле инварианта эта ссылка имеет ненулевое значение. При определении инвариантов остальных типов (в том числе указателя на спецификационный тип), параметр декларируется с определяемым типом.

6. Если в качестве базового типа инварианта используется спецификационный тип, то параметр в выражении вызова инварианта должен иметь тип соответствующей спецификационной ссылки. Иначе в выражении вызова инварианта типа в скобках должно содержаться выражение типа, инвариант которого проверяется.
7. В выражении вызова инварианта типа в скобках должно содержаться выражение, являющееся l-value.

9.4 Инвариант переменной

Если глобальная переменная не может принимать все возможные значения своего типа, то необходимо ограничить множество ее допустимых значений с помощью инварианта переменной. Для этого во всех декларациях переменной в ее определении указывается ключевое слово SeC `invariant`. Это слово говорит о том, что все переменные вводимые данной декларацией имеют ограничение на множество допустимых значений.

Ограничения на множество значений переменной описываются в составном операторе (`compound_statement`), который синтаксически и семантически эквивалентен телу функции без побочных эффектов, без параметров, и возвращающей значение булевского типа.

Составной оператор инварианта помечается модификатором `invariant`, после которого в скобках указывается идентификатор переменной. Поскольку тип возвращаемого значения фиксирован, он не указывается.

Инвариант переменной может быть вызван для проверки его выполнения. Если переменная имеет тип, для которого определён инвариант, инвариант типа будет проверен перед проверкой инварианта переменной.

Выражение вызова инварианта переменной состоит из ключевого слова invariant, за которым следует имя переменной в скобках. Значением результата вычисления выражения вызова инварианта является `true`, если значение переменной в точке вызова удовлетворяет ограничениям инварианта, и `false` — в обратном случае.

Синтаксис

```
( declaration_specifiers )? "invariant" ( declaration_specifiers )?
( init_declarator  ( "," init_declarator )* )? ";"
;

"invariant" "(" <ID> ")" compound_statement ;
"invariant" "(" <ID> ")"
```

Семантические правила

1. Инварианты переменных определяются только для глобальных переменных.
2. До точки определения или вызова инварианта переменная уже должна быть декларирована, причем среди списка спецификаторов ее декларации (`declaration_specifiers`) должен встречаться спецификатор SeC (`se_declaration_specifier`) `invariant`.
3. Если в определении или в одной из деклараций переменной среди списка спецификаторов ее декларации (`declaration_specifiers`) встречается спецификатор SeC (`se_declaration_specifier`) `invariant`, то он должен встречаться в декларациях и определении этой переменной, и определение инварианта этой переменной должно встречаться ровно один раз во всех единицах трансляции (`translation_unit`), собираемых в единую систему.
4. В определении инварианта и выражении вызова инварианта переменной в скобках содержится единственный идентификатор — имя переменной, для которой определен или продекларирован инвариант.
5. Тело инварианта должно быть синтаксически и семантически эквивалентено телу функции без побочных эффектов, без параметров, и возвращающей значение булевского типа.

9.5 Тестовые сценарии

В языке SeC определение тестового сценария аналогично определению глобальной переменной, специфицированной ключевым словом scenario. Тип тестового сценария задается идентификатором, определенным с помощью конструкции `typedef`, и определяет механизм построения теста. Каждому механизму построения теста соответствует:

- тип данных, необходимых для построения теста с помощью данного механизма, который является базовым типом в конструкции `typedef`, определяющей идентификатор типа сценария;
- функция запуска теста, построенного с помощью данного механизма.

При определении тестовый сценарий инициализируется значением типа, соответствующего данному механизму построения теста. Для запуска тестового сценария используется конструкция вызова функции, в которой вместо имени функции используется имя тестового сценария, а параметры аналогичны по своей семантике параметрам стандартной функции `main(int argc, char** argv)`. Конструкция вызова тестового сценария возвращает значение типа `bool`, равное `true`, если тест отработал корректно и в ходе тестирования никаких ошибок обнаружено не было, и `false` — в противном случае.

Синтаксис

```
( decl_specifiers )?
"scenario"
( decl_specifiers )?
( init_declarator  ( "," init_declarator )* )? ";"
;
```

Семантические правила

1. Имена тестовых сценариев входят в то же пространство имен, что и имена глобальных переменных языка C.
2. Тестовый сценарий должен быть определен ровно один раз среди всех единиц трансляции (`translation_unit`), собираемых в одну систему.
3. Не может быть определено локальных тестовых сценариев.
4. Тип тестового сценария должен задаваться идентификатором, определенным с помощью конструкции `typedef`.
5. Идентификатор типа тестового сценария является именем механизма построения теста.
6. Тип инициализатора должен быть совместим с типом тестового сценария.
7. Если тип тестового сценария является структурой, то в инициализаторе должен использоваться синтаксис разыменователей стандарта C99, независимо от используемого стандарта языка C.
8. Вызов тестового сценария аналогичен вызову функции с двумя параметрами: первый типа `int` и второй типа `char**`. При этом второй параметр должен указывать на массив строк, оканчивающихся нулем, последний и только последний элемент массива должен быть нулевым указателем, а значение первого параметра должно равняться длине массива без последнего элемента.
9. Вызов тестового сценария возвращает значение типа `bool`.

10. Для полного определения механизма построения теста необходимо, чтобы была определена функция запуска теста, построенного на основе данного механизма, со показанной в примере сигнатурой.

```
bool start_<test_engine>( int argc
                        , char** argv
                        , <test_engine>* td
                        )
```

9.6 Сценарные функции

В языке SeC сценарная функция представляется как функция без параметров, возвращающая булевский результат, помеченная ключевым словом `scenario`. Сценарные функции могут содержать проверки корректности поведения тестируемой системы, основанные на результатах выполнения ее функций. Если проверка показывает, что тестируемая система ведет себя корректно, сценарная функция должна возвратить `true`, иначе — `false`. Так как результаты проверок постусловий вызываемых спецификационных функций или возникающих отложенных реакций автоматически учитываются тестовой системой, в возвращаемом результате сценарной функции не требуется их учитывать.

Синтаксис

```
( decl_specifiers )?
"scenario"
( decl_specifiers )?
declarator
( declaration )*
compound_statement ;
```

Семантические правила

1. Имена сценарных функций входят в то же пространство имен, что и имена обычных функций языка C.
2. Сценарная функция должна быть определена ровно один раз среди всех единиц трансляции (`translation_unit`), собираемых в одну систему.
3. Сценарные функции не должны иметь параметров и должны возвращать результат типа `bool`.
4. Сценарные функции нельзя вызывать.
5. В сценарных функциях допускается использование операторов итерации, операторов итерации по покрытиям и переменных состояния.

9.6.1 Оператор итерации

В сценарных функциях могут использоваться два вида операторов итерации: обычный итератор и итератор по элементам покрытия. Описание последнего довольно сильно связано с терминологией покрытий, поэтому отнесено в соответствующий раздел.

Итератор начинается с ключевого слова `iterate`, после которого в скобках через точку с запятой указаны:

- декларация и инициализация итерационной переменной;
- условие продолжения итерации;
- способ итерации;
- условие фильтрации;

Все указанные части, за исключением первой, являются необязательными и могут быть опущены. При этом следует учитывать, что при наличие фильтрации и отсутствии способа итерации может произойти зацикливание. Декларация заканчивается телом итератора.

Таким образом, следующая конструкция:

```
iterate (int i=0; i<10; i++;i&1==0) { ... }
```

в некотором смысле аналогична циклу for:

```
int i;
for (i=0; i<10; i++) {
  if (!(i&1==0)) continue;
  ...
}
```

Итерационные переменные не являются локальными, они являются частным случаем переменных состояния. Их значения хранятся в специальной структуре данных, связанной с текущим обобщенным состоянием модели. Эти значения используются как только модель снова окажется в соответствующем обобщенном состоянии.

Синтаксис

```
se_iteration_statement ::= "iterate"
                           "("
                           declaration
                           ( expression )?
                           ";"
                           ( expression )?
                           ";"
                           ( expression )?
                           ")"
                           statement
                       ;
```

Семантические правила

1. Итераторы могут использоваться только в сценарных функциях.
2. Выражения, задающие условия продолжения итерации или условия фильтрации, если они присутствуют в итераторе, должны иметь тип `bool`.
3. В стартовой декларации итератора может быть объявлена одна и только одна итерационная переменная.
4. Переменная, объявленная в стартовой декларации итератора, не должна быть неполного или локального типа, и должна быть инициализирована.

9.6.2 Переменные состояния

Переменные состояния предназначены для хранения данных, связанных с обобщенным состоянием модели. Значения таких переменных становятся доступными, как только модель оказывается в этом обобщенном состоянии снова.

Декларация переменных состояния начинается с модификатора `stable`. Код вида:

```
operator_1;
stable int i = 1;
operator_2;
```

эквивалентен следующему:

```
operator_1;
iterate(int i = 1; false ;;)
{
    operator_2;
}
```

Синтаксис

```
( declaration_specifiers )?
"stable"
( declaration_specifiers )?
( init_declarator  ( "," init_declarator )* )? ";" ;
```

Семантические правила

1. Переменные состояния могут использоваться только в сценарных функциях.
2. Переменные состояния не должны быть неполного или локального типа, и должны быть инициализированы при объявлении.

9.7 Покрытия

Полная информация о покрытии, которая должна быть доступна в рамках теста, разбивается на следующие части:

1. Имя покрытия.
2. Элементы покрытия.
3. Для вычислимых покрытий — функция вычисления элемента покрытия.

Задание этих характеристик покрытия может происходить согласно определенным правилам на протяжении всего модуля компиляции: с помощью определений и объявлений различных видов. В пределах модуля компиляции о каждом покрытии должна быть предоставлена полная информация для корректной работы операций обращения к покрытию и его элементам.

Каждое покрытие может принадлежать определенной спецификационной функции или быть независимым. Таким образом, покрытия делятся на локальные и глобальные:

- Глобальные покрытия не привязаны к спецификационным функциям. Имеют областью видимости модуль компиляции, объявляются и определяются на глобальном уровне.
- Каждое локальное покрытие привязано к определённой спецификационной функции; они считаются определёнными на уровне этой функции и имеют областью видимости промежуток от собственного определения до конца тела функции.Объявления и определения локальных покрытий содержатся, соответственно, в прототипе и определении спецификационной функции;

Покрытия также разделяются по способу задания элементов и методу вычисления достигнутого элемента:

- Если автор теста задаёт совокупность элементов для покрытия непосредственно и отмечает достигнутый элемент самостоятельно, то такое покрытие называется *перечислимым*.
- Если автор теста задаёт совокупность элементов для покрытия непосредственно и задаёт некоторую функцию вычисления достигнутого элемента, то такое покрытие называется *вычислимым*. Функция вычисления - неотъемлемая часть такого покрытия.
- Если автор теста задаёт в качестве совокупности элементов покрытия некоторый enum-тип, то функция вычисления генерируется на основе структуры этого типа, а покрытие назвается *enum-покрытием*.

С помощью операций над элементами покрытий реализуются возможности использования информации об элементах покрытий для управления ходом тестирования и занесения данных в отчёт.

9.8 Объявление и определение покрытий

Существует пять видов объявления и определения покрытий:

1. укороченное объявление;
2. полное объявление;
3. объявление первичных вычислимых покрытий;
4. определение первичных вычислимых покрытий;
5. укороченное определение.

Определением покрытий являются следующие виды конструкций:

- полное объявление, не содержащее extern;
- определение первичных вычислимых покрытий;
- укороченное определение.

Все остальные виды конструкций являются объявлениями покрытий. Объявления и определения одного и того же покрытия в рамках одного модуля компиляции должны следовать друг за другом так, чтобы количество информации известной о покрытии с каждым объявлением строго возрастало (определение у каждого покрытия может быть только одно). Ссылки на семантические правила расположения для конкретного вида объявлений и определений покрытий можно найти в конце данного набора правил, общих для всех объявлений и определений.

Синтаксис

```
se_coverage_declaration ::= ( "extern" )? "coverage" ( "enum" )? <ID>
                                ( "(" ( parameter_type_list )? ")" )?
                                (   se_coverage_initializer
                                  | ";"
                                )
                            ;
se_coverage_initializer ::=   se_coverage_elements_initializer
                            | se_coverage_derivation_initializer
                            | se_coverage_function_initializer
                            ;
statement ::=   ...
              | se_coverage_statement
          ;
se_coverage_statement ::= se_coverage_declaration ;
```

Семантические правила

1. Общие требования к объявлениям и определениям покрытий.

   1.1. Имена глобальных покрытий входят в файловую область видимости.

   1.2. Имена локальных покрытий имеют область видимости, ограниченную спецификационной функцией, аналогично тому, как область видимости полей структуры ограничена самой структурой.

   1.3. Для каждого покрытия в пределах одного `translation_unit` может быть не более одного объявления или определения каждого вида (например, не может быть двух полных объявлений одного и того же покрытия).

1.4. Имена элементов покрытия всегда имеют облать видимости, ограниченную самим покрытием, аналогично тому, как область видимости полей структуры ограничена самой структурой.

1.5. В совокупности всех исходных файлов, составляющих одну программу, должно быть ровно одно определение для каждого объявленного покрытия.

2. Требования ко всем `se_coverage_declaration`.

2.1. Первый идентификатор конструкции `se_coverage_declaration` задаёт имя покрытия.

2.2. Конструкция `se_coverage_declaration` может быть непосредственно вложена в:

2.2.1.`translation_unit`;

2.2.2.`se_coverage_statement`.

3. Требования к `se_coverage_declaration` глобальных покрытий:

3.1. Если нет `extern`, то `se_coverage_declaration` всегда является определением покрытия.

3.2. Если в объявлении или определении глобального покрытия присутствуют круглые скобки, то им задаётся вычислимое покрытие или enum-покрытие.

3.3. Если в `se_coverage_declaration`, задающем глобальное покрытие, присутствует `enum`, то задаётся enum-покрытие.

4. Требования к `se_coverage_declaration` локальных покрытий:

4.1. Конструкция `se_coverage_statement` может присутствовать только в теле определения спецификационной фукнции между пред- и постусловием.

4.2. Запрещен спецификатор `extern`.

4.3. Запрещён `enum` (локальных enum-покрытий не может быть).

4.4. Запрещены круглые скобки (для локальных покрытий список параметров предопределён объемлющей спецификационной функцией).

4.5. Рарешены только иницализаторы вида `se_coverage_derivation_initializer` или `se_coverage_function_initializer`.

4.6. `se_coverage_declaration`, вложенное в `se_coverage_statement`, всегда должно быть определением покрытия.

5. Общие требования к `se_local_coverage_description`.

Синтаксис

```
direct_declarator ::=   // ...
                      | direct_declarator
                        "("
                        parameter_type_list
                        ")"
                        ( se_access_description )*
                        ( se_local_coverage_description )*
                      | direct_declarator
                        "("
                        ( <ID> ( "," <ID> )* )?
                        ")"
                        ( se_access_description )*
```

```
                        ( se_local_coverage_description )*
                    ;
se_local_coverage_description ::= "coverage"
                                    se_local_coverage
                                    (
                                      ","
                                      se_local_coverage
                                    )*
                                ;
se_local_coverage ::= <ID>
                      (   se_coverage_elements_initializer
                        | se_coverage_derivation_initializer
                      )?
```

5.1. Идентификатор в `se_local_coverage` задаёт имя локального покрытия.

5.2. Конструкция `se_local_coverage_description` может присутствовать только в `external_declaration`, при этом только если объемлющий `external_declaration` является прототипом спецификационной функции.

5.3. Если в `translation_unit` есть несколько прототипов одной и той же спецификационной функции, то только один из них может содержать описания локальных покрытий.

5.4. `se_local_coverage` всегда является объявлением покрытия.

6. Общие требования к глобальным вычислимым покрытиям (включая enum-покрытия).

6.1. Типы и имена параметров покрытия должны удовлетворять требованиям к типам и именам параметров объявления или определения C-функции в зависимости от того, является ли `se_coverage_declaration` объявлением или определением покрытия.

6.2. Имена параметров не должны совпадать с именем самого объявляемого или определяемого покрытия.

6.3. В рамках `translation_unit` списки параметров всех объявлений и определений одного и того же глобального вычислимого покрытия (включая enum-покрытия) должны быть одинаковыми. То есть, должны совпадать:

6.3.1. количества параметров;

6.3.2. типы параметров;

6.3.3. имена параметров.

7. Общие требования к локальным покрытиям.

7.1. Локальное покрытие является покрытием с функцией вычисления элементов. Набор параметров покрытия совпадает с набором параметров объемлющей спецификационной функции.

7.2. Имена всех локальных покрытий объявленных и определённых для одной спецификационной функции должны быть различны.

7.3. Имена локальных покрытий не должны совпадать с ...

7.3.1. именем объемлющей спецификационной функции;

7.3.2. именами её параметров;

7.3.3. именами псевдонимов из ограничений доступа.

7.4. Уточнение вышеизложенного требования: считается, что в рамках определения спецификационной функции локальные покрытия входят в область видимости, соответствующую блоку, содержащему определения покрытий.

7.5. Для каждого локального покрытия, объявленного в описании локальных покрытий, определение той же спецификационной функции должно содержать определение покрытия с тем же именем.

7.6. В определении спецификационной функции локальные покрытия должны быть определены в том же порядке, в котором они были объявлены в описании локальных покрытий прототипа этой спецификационной функции.

9.8.1 Укороченное объявление

Укороченное покрытие позволяет объявить покрытие, задав только его имя.

Укороченное определение глобальных покрытий используется для того, чтобы использовать в одном файле модуля компиляции определение покрытия, находящееся в другом файле. Укороченные определения локальных покрытий должны входить в состав прототипа объемлющей спецификационной функции. В укороченном объявлении покрытие не может быть определено (недопустим инициализатор).

Для глобальных покрытий, аналогично подобным объявлениям переменных в C, укороченное объявление специфицируется ключевым словом `extern`; также для глобальных покрытий недопустимо ключевое слово `enum`. Для локальных покрытий эти ограничения естественным образом задаются требованиями к `se_local_coverage_description`.

Укороченное объявление глобального покрытия.

```
extern coverage C( int x, int y );
```

Укороченное объявление локального покрытия.

```
specification void f( int x ) coverage C;
```

Семантические правила

1. Укороченному объявлению покрытия не должно предшествовать ни одно объявление или определение того же покрытия.
2. Укороченное объявление покрытия может предшествовать любой другой форме объявления или определения того же покрытия, за исключением укороченного объявления.

9.8.2 Полное объявление

Полное объявление предназначено для задания всей необходимой информации о покрытии — имени, элементов и, возможно, функции вычисления.

Полное объявление — это синтаксическая структура, предоставляющая всю необходимую информацию об описываемом ею покрытии.

1. Если в полном объявлении глобального покрытия опущен спецификатор extern, то полное объявление является определением покрытия.
2. Если полное объявление локального покрытия находится в определении спецификационной функции, оно является определением; если в прототипе — объявлением.

Полные объявления допустимы для следующих видов покрытий:

1. перечислимые покрытия;
2. прозводные покрытия;
3. enum-покрытия.

Семантические правила

1. До полного объявления покрытия текст `translation_unit` может содержать только укороченное объявление того же покрытия.
2. Если в полном объявлении глобального покрытия отсутствует `extern`, то полное объявление является и определением покрытия.

9.8.3 Полное объявление перечислимого покрытия

Полное объявление перечислимого покрытия задаёт его имя, а также имена и текстовые описания его элементов.

Полное объявление покрытия подчиняется синтаксису объявлений и определений. Так как перечислимые покрытия могут быть только глобальными, то полное объявление перечислимого покрытия задаётся только с помощью конструкции `se_coverage_declaration`.

Конструкция `se_coverage_declaration` является полным объявлением перечислимого покрытия, если:

1. нет `enum`;
2. нет круглых скобок;
3. присутствует инициализатор в форме `se_coverage_elements_initializer`.

Полное объявление перечислимого покрытия

```
extern coverage C
  = { C1 = "first element", C2, C3 = "third element" };
```

Синтаксис

```
se_coverage_elements_initializer ::= "="
                                     "{"
                                     se_coverage_element_declaration
                                     ( ","
se_coverage_element_declaration )*
                                     "}"
                                     ";"
                                   ;
se_coverage_element_declaration ::= <ID>
                                    ( "=" <STRING_LITERAL> )?
                                  ;
```

Опциональные строковые литералы задают текстовые представления соответствующих элементов. При отсутствии строкового литерала в качестве текстового представления элемента покрытия используется его идентификатор.

Семантические правила

1. Идентификаторы в `se_coverage_element_declaration` задают имена элементов покрытия.

2. Имена элементов покрытия должны быть различны.

9.8.4 Полное объявление enum-покрытия

Полное объявление enum-покрытия задаёт первичное покрытие, построенное на основе перечислимого типа, с функцией вычисления по умолчанию.

Конструкция `se_coverage_declaration` является полным объявлением enum-покрытия, если:

1. есть `enum`;
2. есть круглые скобки;
3. нет инициализатора.

Полное объявление enum-покрытия

```
extern coverage enum ColorsCoverage( enum Colors col );
```

Семантические правила

1. Покрытие должно иметь ровно один параметр, тип которого должен быть перечислимым типом.
2. Имена элементов покрытия совпадают с именами констант этого перечислимого типа.

9.8.5 Полное объявление производного покрытия

Полные объявления производных покрытий задают производные перечислимые и вычислимые покрытия.

Конструкция `se_coverage_declaration` является полным объявлением глобального производного покрытия, если:

1. нет `enum`;
2. есть инициализатор в виде `se_coverage_derivation_initializer`.

Полное объявление глобального покрытия-произведения

```
extern coverage PointCoverage( int x, int y )
    = IntCoverage( x ) * IntCoverage( y );
```

Конструкции `se_local_coverage` или `se_coverage_statement` является полным объявлением локального производного покрытия, если есть инициализатор в виде `se_coverage_derivation_initializer`.

Полное объявление локального покрытия инструкцией se_coverage_statement

```
// объявление
specification void f( int x, int y )
coverage PointCoverage = IntCoverage( x ) * IntCoverage( y );
```

Полное объявление локального покрытия инструкцией se_local_coverage

```
specification void f( int x, int y )
{
   pre{ ... };
   coverage PointCoverage( int x, int y )
          = IntCoverage( x ) * IntCoverage( y );
```

```
    post{ ... };
}
```

Синтаксис

```
se_coverage_derivation_initializer ::= "="
                        se_base_coverage
                        ( "*" se_base_coverage )*
                        ( se_coverage_filter )?
                    ;
se_base_coverage ::= se_coverage_name
                        ( "("
                          ( assignment_expr ( "," assignment_expr )* )?
                          ")"
                        )?
                    ;
se_coverage_filter ::= ":" expression ;
```

Все покрытия, упомянутые в `se_base_coverage`, называются базовыми покрытиями данного объявляемого (определяемого) покрытия. Инициализатор данного вида определяет, как именно строится элемент объявляемого (определяемого) покрытия на основе элементов базовых покрытий.

Семантические правила

1. Если покрытие является покрытием с функцией вычисления элементов, то покрытия, указанные в списке `se_base_coverage` должны также быть покрытиями с функцией вычисления. (В противном случае объявляемое покрытие будет невычислимым).
2. Иначе ни одно из покрытий указанных в списке `se_base_coverage` не должно быть вычислимым покрытием.
3. `se_base_coverage` не должно указывать на покрытие, о котором известно только его имя (выше по тексту модуля компиляции нет ничего, кроме укороченного определения).
4. Если `se_coverage_derivation_initializer` находится в объявлении или определении локального покрытия, и `se_coverage_name` в некотором из `se_base_coverage` указывает на локальное покрытие, относящееся к той же функции, что и объявляемое (определяемое) покрытие, то `se_base_coverage` не должно содержать круглых скобок. (`se_coverage_name` здесь трактуется как элемент соответствующего локального покрытия, вычисленный на тех же значениях параметров спецификационной функции).
5. Иначе, если `se_coverage_name` в некотором `se_base_coverage` указывает на вычислимое покрытие, то
   5.1. `se_base_coverage` должен иметь аргументы вызова
   5.2. `se_base_coverage` подчиняется правилам операции вычисления элемента покрытия.
6. `se_coverage_name` в `se_base_coverage` не должно указывать на объявляемое (определяемое) покрытие. (Тем самым запрещена ситуация, когда покрытие зависит от самого себя.)
7. Если `se_coverage_derivation_initializer` содержит ровно одно `se_base_coverage`, и в этом `se_base_coverage` используется первичное покрытие, то считается, что производное покрытие имеет те же самые элементы, что и первичное базовое покрытие.

9.8.6 Объявление первичных вычислимых покрытий

Объявление первичного вычислимого покрытия задаёт:

1. имя покрытия;
2. типы и имена параметров функции вычисления;
3. имена и текстовые представления элементов.

Глобальные покрытия: `se_coverage_declaration` является объявлением глобального первичного вычислимого покрытия, если

1. есть `extern`;
2. есть круглые скобки;
3. есть инициализатор в виде `se_coverage_elements_initializer`.

Объявление глобального первичного вычислимого покрытия

```
extern coverage IntCoverage( int x )
       = { NEGATIVE = "negative"
         , ZERO = "zero"
         , POSITIVE = "positive"
         };
```

Локальные покрытия: `se_local_coverage` является объявлением первичного локального покрытия есть инициализатор в виде `se_coverage_elements_initializer`.

Объявление локального первичного вычислимого покрытия

```
specification void f( int x )
      coverage ArgCoverage = { NEGATIVE = "negative"
                             , ZERO = "zero"
                             , POSITIVE = "positive"
                             };
```

Семантические правила

1. До объявления первичного вычислимого покрытия текст translation_unit может содержать только укороченное объявление того же покрытия.
2. Должны выполняться все ранее изложенные требования, относящиеся к `se_coverage_elements_initializer`.

9.8.7 Определение первичных вычислимых покрытий

Определение первичного вычислимого покрытия задаёт

1. имя покрытия;
2. имена и текстовые представления элементов;
3. алгоритм вычисления элементов.

Определение глобального первичного вычислимого покрытия

```
coverage IntCoverage( int x )
{
   if( x < 0 )
      return { NEGATIVE, "negative" };
   else
```

```
      if( x == 0 )
         return { ZERO, "zero" };
      else
         return { POSITIVE, "positive" };
   }
```

Локальные покрытия: `se_coverage_declaration` является определением локального первичного покрытия, если есть инициализатор в виде `se_coverage_function_initializer`.

Определение локального первичного вычислимого покрытия

```
   specification void sqrt( int x )
   {
      pre{ ... }
      coverage ArgCoverage
      {
         if( x < 0 )
            return;
         else
         if( x == 0 )
            return { ZERO, "zero" };
         else
            return { POSITIVE, "positive" };
      }
      post{ ... }
   }
```

Семантические правила

1. Глобальные покрытия: `se_coverage_declaration` является определением глобального первичного вычислимого покрытия, если:
   1.1. нет `extern`;
   1.2. нет `enum`;
   1.3. есть круглые скобки;
   1.4. есть инициализатор в виде `se_coverage_function_initializer`.
2. До определения первичного вычислимого покрытия текст `translation_unit` может содержать:
   2.1. объявление первичного вычислимого покрытия.
   2.2. укороченное объявление этого покрытия.
3. Инициализатор `se_coverage_function_initializer`.

Синтаксис

```
se_coverage_function_initializer ::= compound_statement ;

se_return_expression ::=   expression
                         | se_coverage_element_return_expression
                       ;

se_coverage_element_return_expression
    ::= "{"
        (   <ID> ( "," <STRING_LITERAL> )?
          | <STRING_LITERAL>
        )
```

```
        "}"
    ;
```

3.1. В `compound_statement` допустимы только следующие формы конструкции `return`:

3.1.1. выражение отсутствует. Это означает, что данная ветвь управления функции вычисления элемента не определяет никакого элемента.

3.1.2. `se_return_expression` имеет вид `se_coverage_element_return_expression`.

3.2. Внутри `compound_statement` должна быть хотя бы одна "не пустая" конструкция `return`.

3.3. Если ранее по тексту `translation_unit` было объявление данного покрытия, то:

3.3.1. `se_coverage_element_return_expression` должно содержать только идентификатор элемента (тем самым запрещается переопределять строковые представления элементов);

3.3.2. этот идентификатор задаёт имя элемента покрытия;

3.3.3. для каждого объявления элемента покрытия определение должно содержать `se_coverage_element_return_expression`, определяющий элемент с таким же именем;

3.3.4. для каждого определенного идентификатора из `se_coverage_element_return_expression` объявление должно содержать элемент покрытия с таким же именем.

3.4. Иначе, если первым элементом `se_coverage_element_return_expression` является:

3.4.1. идентификатор, то:

3.4.1.1. он задаёт имя элемента покрытия. Если за идентификатором следует стороковой литерал, то он задаёт текстовое представление этого элемента.

3.4.1.2. среди всех `se_coverage_element_return_expression` данного `compound_statement`, определяющих один и тот же элемент покрытия, только одно `se_coverage_element_return_expression` может содержать текстовое представление этого элемента;

3.4.1.3. если ни в одном из `se_coverage_element_return_expression`, определяющих один и тот же элемент покрытия, нет строкового литерала, то текстовое представление этого элемента покрытия совпадает с его именем.

3.4.2. строковой литерал, то задаётся безымянный элемент покрытия. Все безымянные элементы покрытия считаются различными вне зависимости от совпадения или несовпадения их описаний. К такому элементу невозможно обратиться с помощью операции получения элемента покрытия. Строковой литерал задаёт текстовое представление этого безымянного элемента покрытия.

9.8.8 Укороченное определение

Конструкция `se_coverage_declaration` является укороченным определением покрытия, если:

1. нет `extern`;

2. нет `enum`;
3. нет круглых скобок;
4. нет инициализатора.

Укороченное определение глобального покрытия

```
extern coverage enum C( enum E e );
...
coverage C; // укороченное определение
```

Укороченное определение локального покрытия

```
specification void f( int x, int y )
     coverage PointCoverage = IntCoverage( x ) * IntCoverage( y );
// ...
specification void f( int x, int y )
{
   pre{ ... }
   coverage PointCoverage; // укороченное определение
   post{ ... }
}
```

Семантические правила

До укороченного определения текст `translation_unit` может содержать:

1. полное объявление;
2. укороченное объявление.

9.8.9 Общие правила обращения к покрытиям

Обращение к глобальному покрытию осуществляется после первого его объявления по имени (см. примеры полных объявлений производных покрытий).

Обращение к локальным покрытиям допускает три различных контекста.

1. Вне прототипа или определения функции обращение к локальному покрытию производится посредством последовательности терминалов `<имя функции>` `"."` `<имя покрытия>`. Такое обращение к локальному покрытию называется обращением через квалифицированное имя покрытия.
2. Внутри прототипа спецификационной функции, к локальному покрытию той же функции можно обратиться как просто по имени, так и через квалифицированное имя покрытия.
3. Внутри определения спецификационной функции в определениях локальных покрытий обращения к ранее определённым покрытиям той же функции производятся так же либо по имени, либо по квалифицированному имени, так как имя локального покрытия имеет область видимости, ограниченную телом спецификационной функции.

Обращение к локальному покрытию вне прототипа или определения функции

```
coverage C( int x, int y ) = f.ArgCoverage( x ) * f.ArgCoverage( y );
```

Обращение к локальному покрытию внутри прототипа функции

```
specification void f( int x, int y )
       coverage X_Coverage
       coverage Y_Coverage
       coverage MutualCoverage = X_Coverage * f.Y_Coverage
     ;
```

Обращение к локальному покрытию внутри определения функции

```
specification void f( int x, int y )
{
   coverage X_Coverage ... ;
   coverage Y_Coverage ... ;
   coverage MutualCoverage = f.X_Coverage * f.Y_Coverage
   post{ ... }
}
```

Семантические правила

1. Конструкция `se_coverage_name`, состоящая из одного идентификатора, указывает на покрытие, если процесс разыменования по правилам языка C даёт нам покрытие с тем же именем.
2. Конструкция `se_coverage_name` вида `<ID> "." <ID>` указывает на локальное покрытие, если:
   2.1. первый идентификатор разрешается в имя некоторой спецификационной функции;
   2.2. для этой функции к моменту появления рассматриваемого `se_coverage_name` в коде `translation_unit` объявлено (определено) покрытие, имя которого совпадает со вторым идентификатором.

9.9 Правила проведения операций над элементами покрытий

Многообразие видов покрытий требует предусмотрения различных контекстов проведения операций над элементами покрытий. Поэтому для формулировки правил обращения к элементам покрытий введем несколько определений.

Покрытия, элементы и функции вычисления которых задаются непосредственно, называются *первичными*.

Производными называются покрытия, построенные на основе других покрытий, которые в этом контексте являются *базовыми*. Определение производного покрытия должно описывать способ его построения на основе базовых покрытий; базовым покрытием может быть как первичное покрытие, так и производное.

Порядком покрытия называется количество первичных покрытий, комбинации элементов которых являются элементами данного покрытия. Порядок любого первичного покрытия равен 1. Порядок производного покрытия равен сумме порядков всех его базовых покрытий.

Для любого покрытия можно построить список первичных покрытий, на которых производное покрытие в конечном счете основано. Длина этого списка равна порядку покрытия. Этот список формируется из списка, содержащего только данное покрытие, следующим образом: циклически проходя от начала списка к концу, вместо покрытий подставляем их базовые покрытия (произведения становятся последовательностями умножаемых покрытий). В конце концов в списке останутся только первичные покрытия, количество которых будет равно порядку исходного.

Очевидно, что по этому алгоритму одинаковые списки можно получить для разных покрытий. *Сравнимыми* называют покрытия, для которых совпадает список первичных базовых покрытий.

С синтаксической точки зрения операции над элементами покрытий производятся над объектами специального типа — `CoverageElement`. В отличие от обычных выражений языка С, для которых отслеживается только тип их результата, для операций, дающих элемент покрытия, отслеживается принадлежность покрытию обрабатываемых или возвращаемых элементов.

9.9.1 Получение константного элемента покрытия

Синтаксис

Это выражение синтаксически выглядит так же, как и доступ к элементу объекта структурного типа в языке С.

```
postfix-expression ::=   // ...
                       | postfix-expression
                         "."
                         identifier
                       | // ...
                   ;
```

Элементы первичных покрытий однозначно нумеруются исходя из объявления покрытия. Однозначная нумерация позволяет использовать выражения их получения всюду, где требуется константное выражение, например в `case`-метках.

Для получения элемента покрытия нужно использовать специальный вид такого выражения.

```
identifier ( "." identifier )+
```

Если цепочка идентификаторов начинается с `se_coverage_name`, указывающего на некоторое покрытие, то вся цепочка идентификаторов рассматривается как выражение получения элемента покрытия. Значение этого выражения имеет тип `CoverageElement`.

Получение константного элемента первичного покрытия

```
IntCoverage.POSITIVE
```

Аналогичное выражение для элемента производного покрытия строится с учётом его порядка и списка первичных покрытий.

Порядком покрытия называется количество первичных покрытий, комбинации элементов которых являются элементами данного покрытия. Порядок любого первичного покрытия равен 1. Порядок производного покрытия равен сумме порядков всех его базовых покрытий.

Для любого покрытия можно построить список первичных покрытий, на которых производное покрытие в конечном счете основано. Длина этого списка равна порядку покрытия. Этот список формируется из списка, содержащего только данное покрытие, следующим образом: циклически проходя от начала списка к концу, вместо покрытий подставляем их базовые покрытия (произведения становятся последовательностями умножаемых покрытий). В конце концов в списке останутся только первичные покрытия, количество которых будет равно порядку исходного.

Так как в состав любого элемента производного покрытия входит ровно один элемент каждого покрытия из этого списка, и порядок следования покрытий в списке однозначен, то получение константного элемента производного покрытия — константное выражение.

Получение константного элемента покрытия второго порядка

```
f.PointCoverage.ZERO.ZERO
```

Семантические правила

Пусть `se_coverage_name` указывает на покрытие *Cov* (не важно, имеющее или нет функцию вычисления элементов).

1. Выражение получения элемента покрытия должно рассматриваться как целочисленное константное выражение (Integer constant expression).
2. Если *Cov* является первичным покрытием, то:
   2.1. после `se_coverage_name` точки должен присутствовать ровно один идентификатор;
   2.2. этот идентификатор должен совпадать с именем некоторого элемента покрытия *Cov*.
3. Иначе покрытие *Cov* является производным покрытием порядка N. Должно выполняться следующее:
   3.1. Количество идентификаторов после `se_coverage_name` должно быть равно N.
   3.2. Каждый i-ый (1 <= i <= N) идентификатор, стоящий после `se_coverage_name`, должен совпадать с некоторым элементом i-го первичного покрытия из списка, построенного указанным выше способом.
4. Результат всего выражения целиком:
   4.1. имеет тип `CoverageElement`;
   4.2. является элементом покрытия *Cov*.

9.9.2 Получение компонента элемента производного покрытия

Для элемента производного покрытия порядка больше единицы можно получить сформировавший его элемент любого базового первичного покрытия.

Порядком покрытия называется количество первичных покрытий, комбинации элементов которых являются элементами данного покрытия. Порядок любого первичного покрытия равен 1. Порядок производного покрытия равен сумме порядков всех его базовых покрытий.

Для любого покрытия можно построить список первичных покрытий, на которых производное покрытие в конечном счете основано. Длина этого списка равна порядку покрытия. Этот список формируется из списка, содержащего только данное покрытие, следующим образом: циклически проходя от начала списка к концу, вместо покрытий подставляем их базовые покрытия (произведения становятся последовательностями умножаемых покрытий). В конце концов в списке останутся только первичные покрытия, количество которых будет равно порядку исходного.

Синтаксис

Для получения компонента элемента производного покрытия используется выражение, синтаксически совпадающее с выражением получения элемента массива.

```
postfix-expression "[" expression "]"
```

Если слева от открывающей квадратной скобки стоит выражение типа `CoverageElement`, то всё выражение целиком рассматривается как выражение получения компонента элемента производного покрытия.

Получение компонента элемента покрытия

```
coverage IntCoverage neg = f.PointCoverage.ZERO[0];
```

Семантические правила

Пусть результат левой части выражения является элементом покрытия Cov_k.

1. Покрытие *Cov* должно быть производным покрытием порядка больше единицы.
2. Выражение в квадратных скобках должно быть целочисленной константой.
3. Значение этой константы должно быть больше или равно нулю и меньше порядка покрытия *Cov*.
4. Результат всего выражения целиком:

   4.1. имеет тип `CoverageElement`;

   4.2. является элементом k-го покрытия в списке первичных базовых покрытий.

9.9.3 Вычисление элемента покрытия

Синтаксис

Это выражение синтаксически выглядит так же, как и вызов функции в C.

```
postfix_expression "(" ( argument_expression-list )? ")"
```

Для вычисления элемента покрытия нужно использовать специальный вид такого выражения:

```
( identifier "." )? identifier "(" ( argument_expression_list )? ")"
```

Если левая часть выражения вызова функции является `se_coverage_name`, указывающим на некоторое покрытие, то всё выражение целиком является выражением вычисления элемента покрытия.

Вычисление элемента глобального покрытия

```
IntCoverage( 10 )
```

Вычисление элемента локального покрытия

```
f.PointCoverage( -1, 1 )
```

Семантические правила

1. `se_coverage_name` должно указывать на вычислимое покрытие.
2. Типы выражений-аргументов должны быть согласованы с типами параметров в объявлении покрытия по тем же правилам, что и типы аргументов вызова функции и типы параметров из её объявления.
3. Результат всего выражения целиком:
    1. имеет тип `CoverageElement`;
    2. является элементом покрытия, на которое указывает `se_coverage_name`.

9.9.4 Выражение трассировки элемента покрытия

Выражение трассировки элемента покрытия предназначено для отражения в отчёте того, что в процессе тестирования достигнут некоторый элемент покрытия.

Синтаксис

```
se_primary_expr ::=   ...
                    | se_trace_expression
                  ;
se_trace_expression ::= "trace"
                        "("
                        assignment_expr
                        ")"
                    ;
```

Трассировка элемента покрытия

```
trace( f.PointCoverage( -10, 10 ) );
```

Семантические правила

1. Аргумент должен иметь тип `CoverageElement`.
2. Результат всего выражения целиком имеет тип `void`.

9.9.5 Тип CoverageElement

Тип `CoverageElement` предназначен для хранения информации об отдельных элементах покрытий. С каждым выражением типа `CoverageElement` связывается и покрытие, элементом которого является результат этого выражения.

Семантические правила

1. Объекты типа `CoverageElement` можно использовать в следующих контекстах:
   1.1. В качестве управляющего (controlling) выражения в операторе `switch`.
   1.2. В качестве выражения `case`-метки в операторе `switch`.
   1.3. В качестве параметра внутри круглых скобок `trace`-выражения.
   1.4. В левой и правой частях сравнения на равенство.
2. Тип `CoverageElement` не приводится (is not compatible) ни к одному другому типу.
3. Если в операторе `switch` используется выражение типа `CoverageElement` покрытия *C1*, то для любой `case`-метки, относящейся к данному оператору `switch` должно выполняться следующее.
   3.1. Выражение в `case`-метке должно быть константным выражением типа `CoverageElement`. Единственный вид таких выражений - это выражения получения константного элемента покрытия. Пусть выражение даёт элемент покрытия *C2*.
   3.2. Покрытия *C1* и *C2* должны быть сравнимы между собой.
   3.3. Для любых двух `case`-меток, выражения которых имеют тип `CoverageElement`, эти выражения должны давать разные элементы покрытия. (То есть не должно быть одинаковой последовательности идентификаторов после первой точки в выражениях получения элемента покрытия).
4. (Расширение требований из 6.5.9 стандарта c99) Если один из операндов оператора `==` (или `!=`) имеет тип `CoverageElement` и принадлежит покрытию *C1*, то:
   4.1. Другой операнд также должен быть типа `CoverageElement`.
   4.2. Если другой операнд принадлежит покрытию *C2*, покрытия *C1* и *C2* должны быть сравнимы между собой.

9.9.6 Итерация по элементам покрытия

Итерация по элементам покрытия — конструкция, помогающая обойти все ветви функциональности, заданные покрытием. Является специфическим усовершенствованием простого оператора итерации.

Параметрами итерации являются имя вызываемой спецификационной функции и имя покрытия. Элементы покрытия, обход которых по каким-то причинам нежелателен, можно отсеять с помощью фильтра.

Синтаксис

```
se_iterate_coverage_statement ::= "iterate"
                                  "coverage"
                                  (    "("
                                       se_coverage_name
                                       ( se_coverage_filter )?
                                       ")"
                                     | se_coverage_name
                                  )
                                  statement
                              ;
```

Конструкция `se_coverage_name` в заголовке итерации указывает покрытие, элементы которого будут перебираться в качестве значений итерационной переменной. Опциональный

фильтр позволяет отсеивать неитересные по тем или иным причинам значения элементов покрытия.

Итерация по локальному покрытию с фильтрацией

```
iterate coverage (    f.PointCoverage : f.PointCoverage[0]
                    != IntCoverage.ZERO
                  )
{
   ...
}
```

Семантические правила

1. Конструкция `se_iterate_coverage_statement` может находиться только внутри определения сценарного метода.
2. Не должно быть вложенных `se_iterate_coverage_statement`, итерирующих элементы одного и того же покрытия.
3. Для `se_coverage_filter` должны выполняться все вышеизложенные требования (см. Конструкция `se_coverage_filter`).

Конструкция фильтра элементов покрытий

Выражение-фильтр позволяет отсеивать те комбинации элементов базовых покрытий, которые не входят в результирующее производное покрытие. Фильтр представляет собой предикат от элемента покрытия. Если на некотором конкретном элементе покрытия выражение-фильтр возвращает значение `0`, то этот элемент отсеивается. В случае использования `se_coverage_filter` в инициализаторе производных покрытий отсеиваются элементы результирующего покрытия, а в случае использования фильтра в итерации по элементам покрытия — элементы покрытия, по которому проводится итерация.

```
coverage NoZeroCoverage = IntCoverage( x )
    : NoZeroCoverage != IntCoverage.ZERO
```

Дополнительные правила написания выражений-фильтров изложены в разделе Обращение к ранее вычисленному элементу покрытия.

Семантические правила

1. Если присутствует выражение-фильтр покрытия, то его результат должен быть скалярного типа (scalar type).
2. Внутри выражения-фильтра разрешены толко следующие операции:
   2.1. обращение к ранее вычисленному элементу покрытия;
   2.2. получение константного элемента покрытия;
   2.3. получение компонента элемента покрытия, где в качесве левой части стоят выражения из подпунктов 2.1 и 2.2;
   2.4. сравненение выражений типа `CoverageElement`;
   2.5. логические выражения языка C.

9.9.7 Обращение к ранее вычисленному элементу покрытия

В определённых контекстах определён ранее вычисленный элемент некоторого покрытия. На такой элемент можно сослаться с помощью либо `name_expr`, либо `field_expr` эквивалентных `se_coverage_name` и указывающих на соответствующее покрытие.

Такими контекстами являются:

- выражение-фильтр в инициализаторе производного покрытия;
- выражение-фильтр в итерации по элементам покрытия;
- тело итерации по элементам покрытия;
- пост-условие спецификационной функции.

Внутри `se_iterate_coverage_statement` известен текущий элемент итерируемого покрытия (текущее значение итерационной переменной). Внутри постусловия известны достигнутые на текущих параметрах элементы всех локальных покрытий объемлющей функции.

Использование ранее вычисленного элемента покрытия в теле итерации по покрытиям

```
iterate coverage IntCoverage
{
   if( IntCoverage == IntCoverage.ZERO )
      f( 0 );
   else
   if( IntCoverage == IntCoverage.POSITIVE )
      f( 10 );
   else
      f( -10 );
}
```

Итерация по глобальному покрытию (в комментариях дана трактовка name_expr C).

```
iterate coverage (  C
                  :    C    // элемент
                    != C.C1 // покрытие
                 )
{
   trace( C ); // элемент
}
```

Итерация по локальному покрытию (в комментариях дана трактовка field_expr f.C).

```
iterate coverage (   f.C
                   :    f.C    // элемент
                     != f.C.C2 // покрытие
                 )
{
   if( f.C( 4 ) // покрытие
       == f.C   // элемент
     )
   {...}
}
```

Семантические правила

1. В некоторых контекстах `name_expr` и `field_expr`, эквивалентные `se_coverage_name` и указывающие на определённое покрытие, обозначают элемент этого покрытия:

1.1. Объявляемое (определяемое) покрытие в выражении-фильтре инициализатора производного покрытия.

1.2. Итерируемое покрытие в выражении-фильтре заголовка итерации по элементам покрытия.

1.3. Итерируемое покрытие в теле итерации по элементам покрытия.

1.4. Локальное покрытие в постусловии объемлющей спецификационной функции.

2. При этом из рассмотрения исключаются ситуации, в которых `se_coverage_name` является именем покрытия в:

   2.1. выражении вычисления элемента покрытия;

   2.2. выражении получения константного элемента покрытия.

9.9.8 Объявление переменной-элемента покрытия

Вычисленные в некоторый момент работы теста элемент покрытия можно сохранить в переменной, имеющей тип `CoverageElement`. Такие переменные объявляются с помощью специального спецификатора типа — `se_coverage_type_specifier`.

```
se_coverage_type_specificer ::= "coverage" se_coverage_name ;
```

Объявление переменной-элемента покрытия

```
coverage IntCoverage zero = IntCoverage( 0 );
```

Семантические правила

1. Объявленные таким образом переменные имеют тип `CoverageElement`.
2. Значениями этих переменных являются элементы покрытия, указанного в `se_coverage_type_specifier`.
3. Если объявление (определение) переменной-элемента покрытия содержит инициализатор, то:

   3.1. он должен представлять собой одиночное выражение типа `CoverageElement`;

   3.2. результатом инициализирующего выражения должен быть элемент покрытия, сравнимого с покрытием, указанным в `se_coverage_name`.

4. В остальном переменные-элементы покрытия подчиняются всем правилам семантики языка C.

9.10 Медиаторная функция

Медиаторные функции помечаются ключевыми словами SeC `mediator for`, между которыми должен содержаться уникальный идентификатор — имя медиаторной функции. Каждая медиаторная функция соответствует некоторой спецификационной функции или отложенной реакции, сигнатура и ограничения доступа которой должны указываться в предварительных декларациях и определении медиаторной функции.

Медиаторная функция может содержать:

1. Блок воздействия, помеченный ключевым словом `call`, в котором реализуется поведение, описанное в соответствующей спецификационной функции, посредством оказания воздействий на тестируемую систему.
2. Блок синхронизации, помеченный ключевым словом `state`, в котором реализуется синхронизация состояния спецификационной модели данных с состоянием тестируемой системы после оказанного воздействия или возникновения отложенной реакции.
3. Дополнительный код перед первым специальным блоком и после второго.

```
( declaration_specifiers )?
"mediator" <ID> "for"
( declaration_specifiers )?
declarator
( declaration )*
compound_statement
;
```

Семантические правила

1. Имена медиаторных функций входят в то же пространство имен, что и имена обычных функций языка C.
2. Медиаторная функция должна быть определена ровно один раз среди всех единиц трансляции (`translation_unit`), собираемых в одну систему.
3. Спецификационная функция или отложенная реакция, для которой определяется медиаторная функция, должна быть декларирована до определения или декларации медиаторной функции.
4. Сигнатура и ограничения доступа спецификационной функции или отложенной реакции, для которой описывается медиатор, должны присутствовать среди спецификаторов (`declaration_specifiers`) и в декларатоpе (`declarator`) всех предварительных декларациях и определении медиаторной функции.
5. Тело медиатора спецификационной функции (`compound_statement`) должно иметь следующий вид:
   - дополнительный C-код;
   - блок воздействия;
   - блок синхронизации (опционально);
   - дополнительный C-код.
6. Тело медиатора отложенной реакции (`compound_statement`) должно иметь следующий вид:
   - дополнительный C-код;

- блок синхронизации;
- дополнительный С-код.

9.11 Семантика блока воздействия медиаторной функции

Блок воздействия на языке SeC — это набор инструкций синтаксически и семантически эквивалентный телу функции, имеющей в точности ту же сигнатуру (порядок параметров и их типы, тип возвращаемого результата), что и спецификационная функция, для которой определяется медиатор. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом `call`.

```
se_call_block_statement ::= "call" compound_statement ;
```

Медиатор для функции выборки элемента стека

```
stack *stack_impl;
List*  stack_model;

int push(stack*, int);
specification bool push_spec(Integer* i) reads i updates stack_model;

mediator push_media for
  specification bool push_spec(Integer* i) reads i updates
stack_model
{
  call {
    return (bool)push(stack_impl, value_Integer(i));
  }
  //...
}
```

Семантические правила

1. Блок воздействия является обязательным блоком медиаторов спецификационных функций.
2. В медиаторах отложенных реакций блок воздействия не допускается.
3. Набор инструкций в блоке воздействия должен быть синтаксически и семантически эквивалентен телу функции, имеющей в точности ту же сигнатуру — порядок параметров и их типы и тип возвращаемого результата, — что и спецификационная функция, для которой определяется медиатор.
4. В блоке воздействия запрещается вызывать спецификационные функции и реакции.

9.12 Блок синхронизации медиаторной функции

Блок синхронизации на языке SeC — это набор инструкций синтаксически и семантически эквивалентный телу функции без возвращаемого результата, сигнатура которой в части параметров совпадает с сигнатурой спецификационной функции или отложенной реакции, для которой определяется медиатор. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом `state`.

```
se_state_block_statement ::= "state" compound_statement ;
```

Медиатор для функции добавления элемента в стек

```
stack *stack_impl;
List*  stack_model;

int push(stack*, int);
specification bool push_spec(Integer* i) reads i updates stack_model;

mediator push_media for
  specification bool push_spec(Integer* i)
                reads i updates stack_model
{
  //...
  state {
    int k;
    clear_List(stack_model);
    for( k = stack_impl->size;
         k > 0;
         append_List( stack_model
                    , create_Integer(stack_impl->elems[--k]))
                    );
  }
}
```

Семантические правила

1. Набор инструкций в блоке синхронизации должен быть синтаксически и семантически эквивалентен телу функции без возвращаемого результата, сигнатура которой в части параметров совпадает с сигнатурой спецификационной функции или отложенной реакции, для которой определяется медиатор.
2. В блоке синхронизации медиатора реакции или функции с типом результата, отличным от `void`, можно получать доступ к значению возникшей отложенной реакции или результата, возвращаемого блоком воздействия того же самого медиатора, через идентификатор, совпадающим с именем отложенной реакции или спецификационной функции.
3. В блоке синхронизации допускается использовать псевдопеременную *`timestamp`*, имеющую тип `TimeInterval`, которая содержит временные метки начала и конца исполнения блока воздействия того же самого медиатора или временные метки, указанные при регистрации отложенной реакции.

9.13 Строковое и XML- представления неспецификационных типов

В данном разделе описаны предупреждения, выдаваемые транслятором языка SeC при проверке пользовательских функций формирования строкового и XML- представления неспецификационных типов.

При обработке декларации или определения функции, имя которой начинается с префикса `to_string_` или `to_XML_`, транслятор поступает следующим образом:

1. Вычисляется суффикс `<имя_типа>` — оставшаяся часть имени функции
2. Если `<имя_типа>` совпадает с одной из строк `spec`, `Default`, `Subtype`, то такая функция игнорируется, так как это служебные функции из библиотеки.
3. Так же функция игнорируется, если `<имя_типа>` является именем спецификационного типа.
4. Если тип возвращаемого значения не равен `String*`, то функция игнорируется и выдаётся предупреждение вида:

   ```
   return type of '<полное имя функции>' is not 'String *'
   ```

5. Если количество параметров не равно 1, то такая функция игнорируется с предупреждением вида:

   ```
   size of parameters list in '<полное имя функции>' is not one
   ```

6. Если тип параметра не `<имя_типа>*`, то такая функция игнорируется с предупреждением вида:

   ```
   parameter of '<полное имя функции>' is not '<имя_типа> *'
   ```

Если рассматриваемая функция не была проигрорирована по одной из вышеизложенных причин, то считается, что для типа `<имя_типа>` задана пользовательская функция формирования строкового или XML- представления (в зависимости от префикса).

10 Библиотека поддержки тестовой системы CTESK

Инструмент CTESK включает в себя библиотеку поддержки разрабатываемых тестов. Эта библиотека предоставляет интерфейс для организации взаимодействия с тестовой системой, а также ряд вспомогательных типов данных и функций. Заголовочные файлы библиотеки располагаются в каталоге include дистрибутива CTESK.

Данный раздел описывает часть интерфейса библиотеки, предназначенную для непосредственного использования разработчиками тестов. Другая часть интерфейса, определенная в заголовочных файлах, предназначена только для использования автоматически сгенерированными компонентами тестовой системы CTESK.

10.1 Базовые сервисы тестовой системы

Основные сервисы, предоставляемые тестовой системой, используются с помощью конструкций спецификационного расширения языка C.

Базовые сервисы тестовой системы CTESK, включенные в библиотеку поддержки, состоят из набора типов данных и функций, определяющих модель времени тестовой системы, и небольшого числа системных функций языка SeC.

10.1.1 Системные функции

Системными функциями языка SeC являются следующие функции:

- **`setBadVerdict`**
  Функция `setBadVerdict` устанавливает отрицательный вердикт текущего вызова медиатора.
- **`assertion`**
  Функция `assertion` проверяет выполнение предполагаемого ограничения и, если оно не выполнено, то завершает работу приложения.

`setBadVerdict`

Функция setBadVerdict устанавливает отрицательный вердикт текущего вызова медиатора.

```
void setBadVerdict( const char* msg );
```

Параметры

`msg`

Комментарий, поясняющий причины отрицательного вердикта медиатора. Этот комментарий попадает в трассу и используется только для упрощения анализа результатов тестирования.

Параметр может принимать нулевое значение. В этом случае никакой комментарий в трассу не попадет.

Дополнительная информация

Функция `setBadVerdict` устанавливает отрицательный вердикт текущего вызова медиатора, если медиатор по каким-либо причинам не может выполнить свою задачу. Таким образом медиатор сообщает об этом тестовой системе.

Функция `setBadVerdict` может быть вызвана несколько раз в течении работы одного медиатора.

Если функция `setBadVerdict` была вызвана вне вызова медиатора, то комментарий попадает в трассу в качестве пользовательского сообщения без какого-либо другого побочного эффекта.

Заголовочный файл: ts/ts.h

`assertion`

Функция `assertion` проверяет выполнение предполагаемого ограничения и, если оно не выполнено, то завершает работу приложения.

```
void assertion( int expr, const char* format, ... );
```

Параметры

expr

Выражение, значение которого предполагается отличным от `0`. Если это предположение нарушено, то работа приложения завершается.

format

Строка, определяющая формат сообщения о нарушении предположения. Сообщение формируется на основании строки формата и дополнительных параметров, семантика которых совпадает с семантикой функции printf из стандартной библиотеки языка C.

Дополнительная информация

Функция `assertion` проверяет, что выполняется предполагаемое ограничение. Если это ограничение не выполнено, то работа приложения завершается.

При этом формируется сообщение о нарушении предположения, которое помещается либо в трассу, если проверка происходит во время работы тестового сценария, либо в `stderr`, в противном случае. Затем трасса корректно закрывается и приложение завершает свою работу посредством вызова системной функции `exit` с параметром 1.

Любое аварийное завершение работы тестового сценария должно выполняться посредством вызова функции `assertion`. В противном случае целостность трассы может быть нарушена.

Заголовочный файл: utils/assertion.h

10.1.2 Модель времени

Тестовая система CTESK поддерживает три режима работы со временем:

- без учета времени;
- линейная модель времени;
- распределенная модель времени.

Для характеризации моментов времени в тестовой системе используются *временные метки*. Временная метка является абстрактной величиной, которая по воли разработчика может быть привязана к реальному времени, а может использоваться только для упорядочивания определенных моментов времени.

Каждая временная метка принадлежит некоторой *системе отсчета времени*. Все временные метки принадлежащие одной системе отсчета линейно упорядочены между собой. Однако порядок между временными метками различных систем отсчета не определен.

При работе в режиме линейной модели времени считается, что все временные метки принадлежат одной единственной системе отсчета времени. Поэтому все временные метки являются линейно упорядоченными.

При работе в режиме распределенной модели времени временные метки могут принадлежать различным системам отсчета. Этот режим является наиболее общим, однако алгоритмы работы тестовой системы с распределенными временными метками наименее эффективны.

Управление моделью времени тестовой системы CTESK осуществляется посредством следующих функций:

- `setTSTimeModel`
- `getTSTimeModel`

Временные метки определяются с помощью следующих типов данных, констант и функций.

Типы:

- `LinearTimeMark`
- `TimeFrameOfReferenceID`
- `TimeMark`
- `TimeInterval`

Константы:

- `systemTimeFrameOfReferenceID`
- `minTimeMark`
- `maxTimeMark`

Функции:

- `getTimeFrameOfReferenceID`
- `setSystemTimeFrameOfReferenceName`
- `createTimeMark`
- `createDistributedTimeMark`
- `createTimeInterval`
- `getCurrentTimeMark`
- `setDefaultCurrentTimeMarkFunction`

`setTSTimeModel`

Функция setTSTimeModel изменяет режим работы со временем тестовой системы CTESK.

```
TSTimeModel setTSTimeModel( TSTimeModel time_model );
```

Параметры

`time_model`

Новый режим работы со временем, в котором будет работать тестовая система.

Возвращаемое значение

Предыдущий режим работы со временем тестовой системы.

Дополнительная информация

При использовании механизма тестирования dfsm, тестовая система по умолчанию работает:

- в режиме без учета времени, если хотя бы одно из полей *`saveModelState`*, *`restoreModelState`* или *`isStationaryState`* тестового сценария не определено или инициализировано нулевым указателем;
- в режиме линейного времени, если поля *`saveModelState`*, *`restoreModelState`* и *`isStationaryState`* тестового сценария инициализированы ненулевыми указателями.

Режим работы со временем может быть изменен в функции инициализации сценария.

Заголовочный файл: ts/timemark.h

`getTSTimeModel`

Функция возвращает текущий режим работы со временем тестовой системы CTESK.

```
TSTimeModel getTSTimeModel( void );
```

Возвращаемое значение

Текущий режим работы со временем тестовой системы CTESK.

Заголовочный файл: ts/timemark

`LinearTimeMark`

Тип используется для идентификации временных меток внутри определенной системы отсчета времени.

```
typedef unsigned long LinearTimeMark;
```

Дополнительная информация

Значения этого типа могут отражать любые характеристики моментов времени внутри определенной системы отсчета времени. Например, число секунд (или миллисекунд) прошедших с определенного момента времени.

Если одно значение типа `LinearTimeMark` больше другого значения, то считается, что оно соответствует моменту времени гарантировано более позднему, чем другое. Если два значения равны, то считается, что о взаимном расположении соответствующих им моментов времени ничего не известно. Они могут совпадать, а могут и быть различными.

Заголовочный файл: ts/timemark.h

`TimeFrameOfReferenceID`

Тип определяет идентификаторы систем отсчета времени.

```
typedef int TimeFrameOfReferenceID;
```

Дополнительная информация

Тестовая система работает в выделенной системе отсчета времени, которая имеет предопределенный идентификатор `systemTimeFrameOfReferenceID`. В режиме линейного времени разрешается использовать только этот идентификатор системы отсчета.

Другие идентификаторы строятся в режиме распределенного времени функцией `getTimeFrameOfReferenceID`. Эта функция по имени системы отсчета возвращает ее идентификатор. При двух обращениях к функции с одним и тем же именем будет получен один и тот же идентификатор. Если вместо имени передать нулевой указатель, то функция вернет уникальный идентификатор системы отсчета, который гарантированно не будет использован ею дважды.

Заголовочный файл: ts/timemark.h

`TimeMark`

Структура определяет тип временной метки, используемой в тестовой системе.

```
typedef struct TimeMark TimeMark;

struct TimeMark {
  TimeFrameOfReferenceID frame;
  LinearTimeMark timemark;
};
```

Дополнительная информация

Временная метка характеризуется идентификатором системы отсчета времени и значением идентифицирующим момент времени внутри данной системы отсчета.

Считается, что одна временная метка меньше другой только в том случае, когда обе метки принадлежат одной системе отсчета и значение поля timemark первой метки меньше значения этого поля другой.

Заголовочный файл: ts/timemark.h

`TimeInterval`

Тип определяет интервал времени между двумя временными метками.

```
typedef struct TimeInterval TimeInterval;
struct TimeInterval { TimeMark minMark; TimeMark maxMark; };
```

Дополнительная информация

Тип `TimeInterval` определяет интервал времени между двумя временными метками *`minMark`* и *`maxMark`*. При этом требуется, чтобы эти метки принадлежали одной системе отсчета времени и значение первой не было больше значения второй. Граничные временные метки включаются внутрь интервала.

Для обозначения минимально и максимально возможных временных меток необходимо использовать специальные константы `minTimeMark` и `maxTimeMark` соответственно.

Заголовочный файл: ts/timemark.h

`systemTimeFrameOfReferenceID`

Константа определяет идентификатор системы отсчета времени, в которой работает тестовая система.

```
extern const TimeFrameOfReferenceID systemTimeFrameOfReferenceID;
```

Дополнительная информация

Идентификатору системы отсчета времени, в которой работает тестовая система, *`systemTimeFrameOfReferenceID`* можно присвоить символическое имя с помощью функции `setSystemTimeFrameOfReferenceName`. В этом случае любой вызов функции `getTimeFrameOfReferenceID` с этим символическим именем будет возвращать *`systemTimeFrameOfReferenceID`*.

Заголовочный файл: ts/timemark.h

`minTimeMark`

Константа равняется временной метке гарантировано меньшей любой другой временной метки из любой системы отсчета.

```
extern const TimeMark minTimeMark;
```

Дополнительная информация

Константа *`minTimeMark`* является выделенным значением типа `TimeMark`, которое гарантированно меньше любого другого значения этого типа независимо от его системы отсчета. Константа *`minTimeMark`* равняется только себе самой.

Заголовочный файл: ts/timemark.h

`maxTimeMark`

Константа равняется временной метке гарантировано большей любой другой временной метки из любой системы отсчета.

```
extern const TimeMark maxTimeMark;
```

Дополнительная информация

Константа *`maxTimeMark`* является выделенным значением типа `TimeMark`, которое гарантированно больше любого другого значения этого типа независимо от его системы отсчета. Константа *`maxTimeMark`* равняется только себе самой.

Заголовочный файл: ts/timemark.h

`getTimeFrameOfReferenceID`

Функция возвращает идентификатор системы отсчета, соответствующий указанному имени.

```
TimeFrameOfReferenceID getTimeFrameOfReferenceID( const char* name );
```

Параметры

`name`

Имя системы отсчета.

Параметр может быть нулевым указателем. В этом случае возвращается уникальный идентификатор системы отсчета, который гарантировано не будет возвращен данной функцией дважды.

Возвращаемое значение

Идентификатор системы отсчета с указанным именем. Функция гарантированно возвращает один и тот же идентификатор для любого числа запросов с заданным именем системы отсчета.

Дополнительная информация

Функция `getTimeFrameOfReferenceID` возвращает идентификатор системы отсчета по ее имени. При двух обращениях к функции с одним и тем же именем будет получен один и тот же идентификатор. Если вместо имени передать нулевой указатель, то функция вернет уникальный идентификатор системы отсчета.

Функцию `getTimeFrameOfReferenceID` допускается использовать только в режиме работы с распределенным временем.

Обычно каждому компьютеру соответствует собственная система отсчета времени. В этом случае, системы отсчета можно идентифицировать сетевым именем соответствующего им компьютера.

Для единообразия работы с идентификаторами систем отсчета, идентификатору системы отсчета, в которой работает тестовая система, также можно присвоить символическое имя с помощью функции `setSystemTimeFrameOfReferenceName`.

Если системе отсчета, в которой работает тестовая система, было присвоено имя, то любой вызов функции `getTimeFrameOfReferenceID` с этим именем вернет *`systemTimeFrameOfReferenceID`*.

Заголовочный файл: ts/timemark.h

`setSystemTimeFrameOfReferenceName`

Функция устанавливает имя системы отсчета, в которой работает тестовая система.

```
bool setSystemTimeFrameOfReferenceName( const char* name );
```

Параметры

`name`

Имя системы отсчета, в которой работает тестовая система. Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает `false`, если данное имя уже использовалось для идентификации системы отсчета отличной от системной, и `true` в противном случае.

Дополнительная информация

Функция `setSystemTimeFrameOfReferenceName` устанавливает имя системы отсчета, в которой работает тестовая система. После этого любой вызов функции `getTimeFrameOfReferenceID` с данным именем вернет *`systemTimeFrameOfReferenceID`*.

Система отсчета, в которой работает тестовая система, может иметь несколько имен одновременно.

Заголовочный файл: ts/timemark.h

`createTimeMark`

Функция `createTimeMark` создает временную метку в системе отсчета, в которой работает тестовая система.

```
TimeMark createTimeMark( LinearTimeMark timemark );
```

Параметры

`timemark`

Метка, идентифицирующая момент времени внутри системы отсчета, в которой работает тестовая система.

Возвращаемое значение

Функция возвращает временную метку, принадлежащую системе отсчета времени, в которой работает тестовая система, и соответствующую внутренней метке *`timemark`*.

Дополнительная информация

Функция `createTimeMark` создает временную метку с идентификатором системы отсчета *`systemTimeFrameOfReferenceID`* и внутренней меткой *`timemark`*.

Заголовочный файл: ts/timemark.h

`createDistributedTimeMark`

Функция создает временную метку в указанной системе отсчета.

```
TimeMark createDistributedTimeMark
         ( TimeFrameOfReferenceID frame, LinearTimeMark timemark );
```

Параметры

frame

Идентификатор системы отсчета, которой принадлежит создаваемая временная метка.

timemark

Метка идентифицирующая момент времени внутри системы отсчета *frame*.

Возвращаемое значение

Функция возвращает временную метку с идентификатором системы отсчета *frame* и внутренней меткой *timemark*.

Дополнительная информация

Заголовочный файл: ts/timemark.h

createTimeInterval

Функция создает временной интервал с указанными границами.

```
TimeInterval createTimeInterval( TimeMark minMark
                               , TimeMark maxMark
                               );
```

Параметры

minMark

Временная метка нижней границы интервала.

maxMark

Временная метка верхней границы интервала.

Возвращаемое значение

Функция возвращает временной интервал с указанными границами.

Дополнительная информация

Функция createTimeInterval создает интервал времени между двумя временными метками *minMark* и *maxMark*. При этом требуется, чтобы эти метки принадлежали одной системе отсчета времени и значение первой не было больше значения второй. Граничные временные метки включаются внутрь интервала.

Для обозначения минимально и максимально возможных временных меток необходимо использовать специальные константы *minTimeMark* и *maxTimeMark* соответственно.

Заголовочный файл: ts/timemark.h

getCurrentTimeMark

Функция возвращает временную метку, соответствующую текущему моменту времени внутри данного процесса.

```
TimeMark getCurrentTimeMark( void );
```

Возвращаемое значение

Функция возвращает временную метку, соответствующую текущему моменту времени внутри данного процесса.

Дополнительная информация

Функция возвращает временную метку, соответствующую текущему моменту времени внутри данного процесса. Данная функция используется тестовой системой для автоматического определения временного интервала, в котором выполняется вызов спецификационной функции.

По умолчанию, текущая временная метка принадлежит системе отсчета, в которой работает тестовая система. Метка внутри системы отсчета вычисляется как число секунд прошедших с полуночи (00:00:00) 1 января 1970 года, полученное путем вызова системной функции `time`.

Поведение функции может быть переопределено пользователем с помощью функции `setDefaultCurrentTimeMarkFunction`.

Заголовочный файл: ts/timemark.h

`setDefaultCurrentTimeMarkFunction`

Устанавливает пользовательскую функцию вычисления временной метки, соответствующей текущему моменту времени внутри данного процесса.

```
GetCurrentTimeMarkFuncType
setDefaultCurrentTimeMarkFunction
( GetCurrentTimeMarkFuncType new_func );
```

Параметры

`new_func`

Указатель на функцию, которая будет использоваться для вычисления временной метки, соответствующей текущему моменту времени внутри данного процесса. Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает указатель на предыдущую функцию, вычислявшую текущую временную метку.

Дополнительная информация

Функция `setDefaultCurrentTimeMarkFunction` устанавливает пользовательскую функцию вычисления временной метки, соответствующей текущему моменту времени внутри активного процесса. После этого функция `getCurrentTimeMark` будет возвращать временную метку, полученную путем вызова данной пользовательской функции.

Таким образом, пользовательская функция будет использоваться для автоматического вычисления интервала времени, соответствующего вызову каждой спецификационной функции.

Заголовочный файл: ts/timemark.h

11 Стандартные механизмы построения тестов

В CTESK 2.8 включены два механизма построения тестов (иначе называемых «обходчики») dfsm и ndfsm. Благодаря своей гибкости они позволяют тестировать очень широкий класс программного обеспечения, начиная с простых систем без внутреннего состояния и кончая распределенными системами с асинхронным интерфейсом.

11.1 dfsm

Механизм построения тестов dfsm основывается на понятии обхода детерминированного конечного автомата. Конечный автомат, используемый для построения теста, задается в неявном виде. Пользователь определяет функцию вычисления текущего состояния сценария и набор тестовых воздействий.

Во время тестирования dfsm применяет тестовые воздействия, которые могут изменять состояние сценария. dfsm автоматически отслеживает все изменения состояния и строит конечный автомат, соответствующий процессу тестирования. Состояниями автомата являются все достижимые состояния сценария, а переходы автомата помечаются тестовыми воздействиями, которые их инициировали.

Механизм тестирования dfsm заканчивает тестирование только тогда, когда подаст все определенные пользователем тестовые воздействия во всех состояниях автомата достижимых из начального.

Чтобы достижение этой цели было возможно, необходимо выполнение следующих условий:

- **Конечность**. Число состояний, достижимых из начального путем применения тестовых воздействий из заданного набора, должно быть конечным.
- **Детерминированность**. Для любого состояния системы применение одного и того же тестового воздействия всегда должно переводить систему в одно и тоже состояние.
- **Сильная связность**. Из любого состояния сценария достижимо любое другое состояние сценария путем применения последовательности тестовых воздействий.

Набор тестовых воздействий определяется с помощью сценарных функций.

Тип описания тестового сценария dfsm используется при инициализации тестового сценария, построенного на основе механизма dfsm. Поля этого типа описаны на странице «Поля типов описания тестового сценария».

Дополнительные параметры механизма тестирования могут быть настроены с помощью следующих функций:

- `setFinishMode`
- `setDeferredReactionsMode`
- `setWTime`
- `setFindFirstSeriesOnly`
- `setFindFirstSeriesOnlyBound`

Все функции, предназначенные устанавливать или возвращать параметры механизма тестирования, перечислены на странице «Сервисные функции механизма тестирования»

При запуске тестового сценария ему передается список параметров. Механизм тестирования имеет стандартные параметры, которые настраивают его работу. Все стандартные параметры должны следовать до пользовательских параметров. Механизм тестирования обрабатывает их и передает оставшиеся в функцию инициализации тестового сценария.

11.2 ndfsm

Обходчик ndfsm по сравнению с dfsm позволяет корректно работать на более широком классе автоматов, а именно на конечных атоматах, имеющих детерминированный сильносвязный полный остовный подавтомат.

- **Остовный подавтомат**. Остовный подавтомат содержит все состояния сценария, достижимые в процессе тестирования.
- **Полный подавтомат**. Для каждого состояния сценария и допустимого в нем тестового воздействия подавомат либо содержит все переходы из этим состояния, помеченные этим тестовым воздействием, либо не содержит таких переходов вовсе.

Набор тестовых воздействий определяется с помощью сценарных функций.

Тип описания тестового сценария ndfsm используется при инициализации тестового сценария, построенного на основе механизма ndfsm. Поля этого типа описаны на странице «Поля типов описания тестового сценария».

Дополнительные параметры механизма тестирования могут быть настроены с помощью следующих функций:

- `setFinishMode`
- `setDeferredReactionsMode`
- `setWTime`
- `setFindFirstSeriesOnly`
- `setFindFirstSeriesOnlyBound`

Все функции, предназначенные устанавливать или возвращать параметры механизма тестирования, перечислены на странице «Сервисные функции механизма тестирования»

При запуске тестового сценария ему передается список параметров. Механизм тестирования имеет стандартные параметры, которые настраивают его работу. Все стандартные параметры должны следовать до пользовательских параметров. Механизм тестирования обрабатывает их и передает оставшиеся в функцию инициализации тестового сценария.

11.3 Поля типов описания тестового сценария

Тип описания тестового сценария dfsm используется при инициализации тестового сценария, построенного на основе механизма dfsm. Соответственно, тип описания тестового сценария ndfsm используется при инициализации тестового сценария, построенного на основе механизма ndfsm.

Эти типы являются структурами; множество полей dfsm включает множество полей ndfsm. Эти поля присутствуют в обеих структурах:

- *`init`* (тип `PtrInit`)
- *`finish`* (тип `PtrInit`)
- *`getState`* (тип `PtrGetState`)
- *`actions`*

Обязательным для инициализации является только поле *`actions`*, которое содержит массив сценарных функций, определяющих тестовые воздействия.

Структура dfsm включает в себя дополнительные поля:

- *`saveModelState`* (тип `PtrSaveModelState`)
- *`restoreModelState`* (тип `PtrRestoreModelState`)
- *`isStationaryState`* (тип `PtrIsStationaryState`)
- *`observeState`* (тип `PtrObserveState`)
- ***`init`***
  Поле *`init`* содержит указатель на функцию инициализации тестового сценария.
- ***`finish`***
  Поле *`finish`* содержит указатель на функцию завершения тестового сценария.
- ***`getState`***
  Поле *`getState`* содержит указатель на функцию, вычисляющую текущее состояние тестового сценария.
- ***`actions`***
  Поле *`actions`* содержит массив сценарных функций, завершающийся нулевым указателем.
- ***`saveModelState`***
  Поле *`saveModelState`* содержит указатель на функцию, возвращающую состояние спецификационной модели данных.
- ***`restoreModelState`***
  Поле *`restoreModelState`* содержит указатель на функцию, восстанавливающую состояние спецификационной модели данных.
- ***`isStationaryState`***
  Поле *`isStationaryState`* содержит указатель на функцию проверки стационарности модельного состояния.
- ***`observeState`***
  Поле *`observeState`* содержит указатель на функцию, синхронизирующую модельное состояние с состоянием тестируемой системы по истечении времени стабилизации.

`init`

Поле *`init`* содержит указатель на функцию инициализации тестового сценария.

```
PtrInit init;
```

Дополнительная информация

Функция инициализации получает на вход массив параметров, согласно семантике принятой для параметров *`argc`*, *`argv`* стандартной функции `main`. Она может использовать эти параметры для настройки тестового сценария.

Обычно функция инициализации выполняет следующие действия:

- инициализация тестируемой системы;
- инициализация спецификационной модели данных;
- инициализация данных сценария;
- установка медиаторов для спецификационных функций и реакций, используемых в данном тестовом сценарии.

В случае тестирования систем с отложенными реакциями, функция инициализации дополнительно используется для:

- установки времени ожидания отложенных реакций;
- при необходимости, для распределения каналов обработки непосредственных и отложенных реакций.

Функция инициализации возвращает булевскую величину, которая должна принимать значение `true`, если инициализация завершилась успешно, и `false` – в противном случае. В последнем случае работа тестового сценария на этом завершится, функция завершения сценария вызвана не будет.

Функция инициализация может содержать вызовы спецификационных функций, осуществляющих инициализацию тестируемой системы, спецификационной модели данных или данных сценария. При работе в режиме с отложенными реакциями механизм построения тестов осуществляет сериализацию всех поданных при вызове функции инициализации стимулов и всех полученных реакций.

Поле *`init`* может быть инициализировано нулевым указателем или не инициализировано вовсе. Это будет эквивалентно инициализации поля *`init`* функцией, не выполняющей никаких действий и возвращающей `true`.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

`finish`

Поле *`finish`* содержит указатель на функцию завершения тестового сценария.

```
PtrFinish finish;
```

Дополнительная информация

Функция завершения предназначена для выполнения заключительных работ после проведения тестирования. Функция не имеет параметров и возвращаемого значения.

Обычно функция завершения сценария выполняет следующие действия:

- освобождение ресурсов тестируемой системы, занятых во время работы тестового сценария;

- освобождение ресурсов спецификационной модели данных;
- освобождение ресурсов тестового сценария.

Функция завершения сценария может содержать вызовы спецификационных функций, осуществляющих освобождение ресурсов тестируемой системы, ресурсов спецификационной модели данных или ресурсов сценария. При работе в режиме с отложенными реакциями механизм построения тестов осуществляет сериализацию всех поданных при вызове функции завершения сценария стимулов и всех полученных реакций.

Поле *`finish`* может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае, никаких действий по завершению сценария выполняться не будет.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

`getState`

Поле *`getState`* содержит указатель на функцию, вычисляющую текущее состояние тестового сценария.

```
PtrGetState getState;
```

Дополнительная информация

Функция вычисления состояния сценария не имеет параметров и возвращает объект спецификационного типа.

При определении состояния сценария важно учитывать требование детерминированности механизма построения тестов dfsm или наличие детерминированного сильносвязного полного остовного подавтомата для механизма построения тестов ndfsm.

Поле *`getState`* может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае механизм тестирования dfsm будет считать, что состояние сценария всегда одно и тоже.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

`actions`

Поле *`actions`* содержит массив сценарных функций, завершающийся нулевым указателем.

```
ScenarioFunctionID actions[];
```

Дополнительная информация

Поле *`actions`* содержит массив сценарных функций, завершающийся нулевым указателем.

Поле *`actions`* является обязательным для инициализации. Последнее значение массива должно быть нулевым указателем.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

`saveModelState`

Поле *`saveModelState`* содержит указатель на функцию, возвращающую состояние спецификационной модели данных.

```
PtrSaveModelState saveModelState;
```

Дополнительная информация

Функция сохранения состояния спецификационной модели данных не имеет параметров и возвращает объект спецификационного типа. Этот объект должен содержать все состояние

спецификационной модели данных, чтобы функция восстановления состояния спецификационной модели данных могла полностью его восстановить по данному объекту.

Поле *saveModelState* может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае тестирование систем с отложенными реакциями с помощью данного тестового сценария будет невозможно.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

restoreModelState

Поле *restoreModelState* содержит указатель на функцию, восстанавливающую состояние спецификационной модели данных.

```
PtrRestoreModelState restoreModelState;
```

Дополнительная информация

Функция восстановления состояния спецификационной модели данных получает объект спецификационного типа и восстанавливает по нему состояние спецификационной модели данных. Получаемый объект заведомо был построен ранее с помощью функции сохранения состояния спецификационной модели данных. Функция восстановления состояния не имеет возвращаемого значения.

Поле *restoreModelState* может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае тестирование систем с отложенными реакциями с помощью данного тестового сценария будет невозможно.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

isStationaryState

Поле *isStationaryState* содержит указатель на функцию проверки стационарности модельного состояния.

```
PtrIsStationaryState isStationaryState;
```

Дополнительная информация

Функция проверки стационарности модельного состояния не имеет параметров и возвращает `true`, если текущее модельное состояние является стационарным, и `false` – иначе.

Модельное состояние называется стационарным, если в этом состоянии целевая система, удовлетворяющая данной модели, не может инициировать взаимодействие.

Поле *isStationaryState* может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае тестирование систем с асинхронным интерфейсом с помощью данного тестового сценария будет невозможно.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

observeState

Поле *observeState* содержит указатель на функцию, синхронизирующую модельное состояние с состоянием тестируемой системы по истечении времени стабилизации.

```
PtrObserveState observeState;
```

Дополнительная информация

Функция синхронизации модельного состояния не имеет параметров и возвращаемого значения. Она вызывается после того как тестовая система подаст очередное тестовое

воздействие и подождет в течении времени стабилизации того, чтобы целевая система перешла в стационарное состояние. Во время вызова функция синхронизации может обратиться к одной или нескольким спецификационным функциям, имеющим доступ к состоянию тестируемой системы, но не изменяющие его. Взаимодействия, инициированные из функции синхронизации, будут учитываться во время сериализации наравне с другими взаимодействиями предшествовавшего ему тестового воздействия.

Поле *`observeState`* может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае никакой синхронизации в стационарном состоянии проводиться не будет.

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

11.4 Типы данных, используемые механизмом тестирования

Типы данных в приведенном списке предназначены для хранения значений полей тестового сценария и ссылок на пользовательские функции обслуживания.

- **`FinishMode`**
  Перечислимый тип `FinishMode` определяет возможные режимы завершения работы механизма тестирования `dfsm`.
- **`PtrFinish`**
  Тип `PtrFinish` определяет тип функции завершения тестового сценария.
- **`PtrGetState`**
  Тип `PtrGetState` определяет тип функции, вычисляющей текущее состояние тестового сценария.
- **`PtrInit`**
  Тип `PtrInit` определяет тип функции инициализации тестового сценария.
- **`PtrIsStationaryState`**
  Тип `PtrIsStationaryState` определяет тип функции проверки стационарности модельного состояния.
- **`PtrIsStationaryState`**
  Тип `PtrObserveState` определяет тип функции, синхронизирующей модельное состояние с состоянием тестируемой системы по истечении времени стабилизации.
- **`PtrRestoreModelState`**
  Тип `PtrRestoreModelState` определяет тип функции, восстанавливающей состояние спецификационной модели данных.
- **`PtrSaveModelState`**
  Тип `PtrSaveModelState` определяет тип функции, возвращающей состояние спецификационной модели данных.

`FinishMode`

Перечислимый тип `FinishMode` определяет возможные режимы завершения работы механизма тестирования dfsm.

```
typedef enum { UNTIL_ERROR, UNTIL_END } FinishMode;
```

Дополнительная информация

Перечислимый тип `FinishMode` определяет возможные режимы завершения работы механизма тестирования dfsm.

Первый режим — `UNTIL_ERROR` означает, что тестирование завершается сразу после обнаружения первой ошибки.

Второй режим — `UNTIL_END` означает, что тестирование после обнаружения не критичной ошибки будет продолжено и завершится только после достижения стандартного условия завершения тестирования.

По умолчанию, механизм тестирования `dfsm` работает в режиме до первой ошибки.

Заголовочный файл: ts/engine.h

`PtrFinish`

Тип `PtrFinish` определяет тип функции завершения тестового сценария.

```
typedef void (*PtrFinish)( void );
```

Дополнительная информация

Функция завершения сценария не имеет параметров и возвращаемого значения. Она предназначена для освобождения ресурсов после завершения работы сценария.

Заголовочный файл: ts/engine.h

`PtrGetState`

Тип `PtrGetState` определяет тип функции, вычисляющей текущее состояние тестового сценария.

```
typedef Object* (*PtrGetState)( void );
```

Дополнительная информация

Функция вычисления состояния сценария не имеет параметров и возвращает объект спецификационного типа.

Заголовочный файл: ts/engine.h

`PtrInit`

Тип `PtrInit` определяет тип функции инициализации тестового сценария.

```
typedef bool (*PtrInit)( int, char** );
```

Дополнительная информация

Функция инициализации получает на вход массив параметров, согласно семантике принятой для параметров *`argc`*, *`argv`* стандартной функции main, и возвращает булевскую величину, которая должна принимать значение true, если инициализация завершилась успешно, и `false` – в противном случае.

Заголовочный файл: ts/engine.h

`PtrIsStationaryState`

Тип `PtrIsStationaryState` определяет тип функции проверки стационарности модельного состояния.

```
typedef bool (*PtrIsStationaryState)(void);
```

Дополнительная информация

Функция проверки стационарности модельного состояния не имеет параметров и возвращает значение типа `bool`.

Заголовочный файл: ts/engine.h

`PtrObserveState`

Тип `PtrObserveState` определяет тип функции, синхронизирующей модельное состояние с состоянием тестируемой системы по истечении времени стабилизации.

```
typedef void (*PtrObserveState)( void );
```

Дополнительная информация

Функция синхронизации модельного состояния не имеет параметров и возвращаемого значения.

Заголовочный файл: ts/engine.h

`PtrRestoreModelState`

Тип `PtrRestoreModelState` определяет тип функции, восстанавливающей состояние спецификационной модели данных.

```
typedef void (*PtrSaveModelState)( Object* );
```

Дополнительная информация

Функция получает объект спецификационного типа и восстанавливает по нему состояние спецификационной модели данных. Функция не имеет возвращаемого значения.

Заголовочный файл: ts/engine.h

`PtrSaveModelState`

Тип `PtrSaveModelState` определяет тип функции, возвращающей состояние спецификационной модели данных.

```
typedef Object* (*PtrSaveModelState)( void );
```

Дополнительная информация

Функция сохранения состояния спецификационной модели данных не имеет параметров и возвращает объект спецификационного типа.

Заголовочный файл: ts/engine.h

11.5 Сервисные функции механизма тестирования

Механизмом тестирования можно управлять с помощью функций из приведенного ниже списка.

- **areDeferredReactionsEnabled**
  Функция areDeferredReactionsEnabled возвращает текущий режим поддержки отложенных реакций.
- **getFindFirstSeriesOnlyBound**
  Функция getFindFirstSeriesOnlyBound возвращает текущее значение свойства FindFirstSeriesOnlyBound.
- **getFinishMode**
  Функция getFinishMode возвращает текущий режим завершения работы тестового сценария.
- **getWTime**
  Функция getWTime возвращает время ожидания стабилизации целевой системы.
- **isFindFirstSeriesOnly**
  Функция isFindFirstSeriesOnly возвращает текущее значение свойства FindFirstSeriesOnly.
- **setDeferredReactionsMode**
  Функция setDeferredReactionsMode устанавливает режим поддержки отложенных реакций.
- **setFindFirstSeriesOnly**
  Функция setFindFirstSeriesOnly устанавливает свойство FindFirstSeriesOnly.
- **setFindFirstSeriesOnlyBound**
  Функция setFindFirstSeriesOnlyBound устанавливает свойство FindFirstSeriesOnlyBound.
- **setFinishMode**
  Функция setFinishMode устанавливает значение режима завершения работы тестового сценария.
- **setWTime**
  Функция setWTime устанавливает время ожидания стабилизации целевой системы.

areDeferredReactionsEnabled

Функция возвращает текущий режим поддержки отложенных реакций.

```
bool areDeferredReactionsEnabled( void );
```

finish_mode

Новый режим завершения работы тестового сценария.

Возвращаемое значение

Текущий режим поддержки отложенных реакций.

Дополнительная информация

Для изменения режима поддержки отложенных реакций предназначена функция `setDeferredReactionsMode`.

Заголовочный файл: ts/engine.h

`getFindFirstSeriesOnlyBound`

Функция возвращает текущее значение свойства `FindFirstSeriesOnlyBound`.

```
int getFindFirstSeriesOnlyBound( void );
```

Возвращаемое значение

Текущее значение свойства `FindFirstSeriesOnlyBound`.

Дополнительная информация

Для установки текущего значения свойства `FindFirstSeriesOnlyBound` предназначена функция `setFindFirstSeriesOnlyBound`.

Заголовочный файл: ts/engine.h

`getFinishMode`

Функция возвращает текущий режим завершения работы тестового сценария.

```
FinishMode getFinishMode( void );
```

Возвращаемое значение

Текущий режим завершения работы тестового сценария.

Дополнительная информация

Функция `getFinishMode` возвращает текущий режим завершения работы тестового сценария. Для изменения текущего режима завершения работы предназначена функция `setFinishMode`.

Заголовочный файл: ts/engine.h

`getWTime`

Функция возвращает время ожидания стабилизации целевой системы.

```
time_t getWTime( void );
```

Возвращаемое значение

Время ожидания стабилизации целевой системы.

Дополнительная информация

Функция `getWTime` возвращает время ожидания стабилизации целевой системы. Для изменения времени ожидания стабилизации целевой системы предназначена функция `setWTime`.

Заголовочный файл: ts/engine.h

`isFindFirstSeriesOnly`

Функция возвращает текущее значение свойства `FindFirstSeriesOnly`.

```
bool setFindFirstSeriesOnly( void );
```

Возвращаемое значение

Текущее значение свойства `FindFirstSeriesOnly`.

Дополнительная информация

Для изменения значения свойства `FindFirstSeriesOnly` предназначена функция `setFindFirstSeriesOnly`.

Заголовочный файл: ts/engine.h

setDeferredReactionsMode

Функция устанавливает режим поддержки отложенных реакций.

```
bool setDeferredReactionsMode( bool enable );
```

Параметры

enable

Значение параметра `true` соответствует включенному режиму поддержки отложенных реакций, значение `false` – выключенному.

Возвращаемое значение

Функция возвращает предыдущее значение режима поддержки отложенных реакций.

Дополнительная информация

Функция `setDeferredReactionsMode` устанавливает режим поддержки отложенных реакций. Режим поддержки отложенных реакций не может быть включен, если хотя бы одно из полей *`saveModelState`*, *`restoreModelState`* или *`isStationaryState`* тестового сценария не определено или инициализировано нулевым указателем. Если же все эти поля инициализированы ненулевыми указателями, то режим поддержки отложенных реакций «включен» по умолчанию.

Изменять режим поддержки отложенных реакций разрешается только в функции инициализации тестового сценария.

Для доступа к текущему значению режима поддержки отложенных реакций предназначена функция `areDeferredReactionsEnabled`.

Заголовочный файл: ts/engine.h

setFindFirstSeriesOnly

Функция устанавливает свойство `FindFirstSeriesOnly`.

```
bool setFindFirstSeriesOnly( bool new_value );
```

Параметры

new_value

Устанавливаемое значение свойства `FindFirstSeriesOnly`.

Возвращаемое значение

Функция возвращает предыдущее значение свойства `FindFirstSeriesOnlyBound`.

Дополнительная информация

Функция `setFindFirstSeriesOnly` устанавливает свойство `FindFirstSeriesOnly`. Если свойство принимает значение `false`, то во время сериализации взаимодействий с целевой системой, тестовый механизм строит все допустимые последовательности взаимодействий и проверяет, что все они проводят в одно и тоже состояние спецификационной модели данных. То есть проверяется детерминированность модели. Если из дополнительных знаний о модели известно, что все допустимые цепочки всегда приводят в одно и тоже стационарное состояние, то допускается установить значение свойства `FindFirstSeriesOnly` в `true` и, таким образом, оптимизировать работу тестовой системы. Например, если в модели определено одно-единственное стационарное состояние спецификационной модели данных, то вышеуказанное условие заведомо выполняется и свойству `FindFirstSeriesOnly` допускается присвоить значение `true`.

По умолчанию значение свойства равно `false`.

Изменять свойство `FindFirstSeriesOnly` разрешается только во время работы тестового сценария, в том числе в функции инициализации тестового сценария. Все изменения этого свойства до начала работы тестового сценария на его работу не повлияют.

Для получения текущего значения свойства `FindFirstSeriesOnly` предназначена функция `isFindFirstSeriesOnly`.

Заголовочный файл: ts/engine.h

setFindFirstSeriesOnlyBound

Функция устанавливает свойство `FindFirstSeriesOnlyBound`.

```
int setFindFirstSeriesOnlyBound( int bound );
```

Параметры

bound

Устанавливаемое значение свойства `FindFirstSeriesOnlyBound`.

Возвращаемое значение

Функция возвращает предыдущее значение свойства `FindFirstSeriesOnlyBound`.

Дополнительная информация

Если значение свойства `FindFirstSeriesOnlyBound` равно нулю, то во время сериализации взаимодействий с целевой системой, тестовый механизм будет строить все допустимые последовательности взаимодействий и проверять, что все они проводят в одно и тоже состояние спецификационной модели данных.

Если значение свойства `FindFirstSeriesOnlyBound` положительно, то во время сериализации взаимодействий с целевой системой, тестовый механизм будет только в том случае строить все допустимые последовательности взаимодействий, если число взаимодействий меньше значения свойства `FindFirstSeriesOnlyBound`, в противном случае, тестовый механизм рассмотрим единственную допустимую последовательность. По умолчанию значение свойства равно `0`.

Вызов `setFindFirstSeriesOnlyBound(0)` эквивалентен вызову `setFindFirstSeriesOnly(false)`. Вызов `setFindFirstSeriesOnlyBound(1)` эквивалентен вызову `setFindFirstSeriesOnly(true)`.

Изменять свойство `FindFirstSeriesOnlyBound` разрешается только во время работы тестового сценария, в том числе в функции инициализации тестового сценария. Все изменения этого свойства до начала работы тестового сценария на его работу не повлияют.

Для получения текущего значения свойства `FindFirstSeriesOnlyBound` предназначена функция `getFindFirstSeriesOnlyBound`.

Заголовочный файл: ts/engine.h

`setFinishMode`

Функция устанавливает значение режима завершения работы тестового сценария.

```
FinishMode setFinishMode( FinishMode finish_mode );
```

Параметры

`finish_mode`

Новый режим завершения работы тестового сценария.

Возвращаемое значение

Предыдущий режим завершения работы тестогово сценария.

Дополнительная информация

Функция `setFinishMode` устанавливает значение режима завершения работы механизма тестирования dfsm. По умолчания, механизмы тестирования `dfsm` и `ndfsm` работают в режиме до первой ошибки.

Изменять режим работы разрешается в любой момент работы тестовой системы. При этом изменение режима завершения работы повлияет только на вновь обнаруживаемые ошибки, и никак не скажется на обнаруженных ранее.

Для доступа к значению текущего режима завершения работы предназначена функция `getFinishMode`.

Заголовочный файл: ts/engine.h

`setWTime`

Функция устанавливает время ожидания стабилизации целевой системы.

```
time_t setWTime(time_t secs);
```

Параметры

`secs`

Время ожидания стабилизации целевой системы в секундах. Значение параметра должно быть неотрицательным целым числом.

Возвращаемое значение

Функция возвращает предыдущее значение времени ожидания стабилизации целевой системы.

Дополнительная информация

Функция `setWTime` устанавливает время ожидания стабилизации целевой системы. Это время, которое ждет механизм тестирования после осуществления каждого тестового

воздействия, для того, чтобы вся информация о реакциях была собрана и целевая система перешла в стационарное состояние.

По умолчанию значение времени ожидания равно `0`.

Изменять время ожидания стабилизации целевой системы разрешается только в функции инициализации тестового сценария.

Для получения текущего значения времени ожидания предназначена функция `getWTime`.

Заголовочный файл: ts/engine.h

11.6 Стандартные параметры тестового сценария

При запуске тестового сценария ему передается список параметров. Механизм тестирования имеет стандартные параметры, которые настраивают его работу. Все стандартные параметры должны следовать до пользовательских параметров. Механизм тестирования обрабатывает их и передает оставшиеся в функцию инициализации тестового сценария.

В CTESK 2.8 механизм тестирования поддерживает стандартные параметры, приведенные в списке.

- **`--ffso`**
  `--find-first-series-only`
  Указывает находить только первую успешную серию.
- `-nt`
  `--no-trace`
  Выключает трассировку.
- **`-t`**
  Направляет трассировку в файл с указанным именем.
- **`-tc`**
  Направляет трассировку на консоль.
- **`--trace-accidental`**
  Включает трассировку недостоверных переходов.
- **`-tt`**
  Направляет трассировку в файл с уникальным именем, составленным из имени сценария и текущего времени.
- **`-uend`**
  Указывает выполнять тестирование до конца, несмотря на ошибки.
- **`-uerr`**
  Указывает выполнять тестирование до возникновения первой ошибки (по умолчанию).
- **`--trace-format`**
  Указывает формат сброса данных в трассу.
- **`--disabled-actions`**
  Указывает имя файла со списком сценарных функций, которые не будут вызываться.

`--find-first-series-only`
`-ffso`

Указывает находить только первую успешную серию.

Дополнительная информация

Стандартный параметр `--find-first-series-only` (в сокращенной форме `-ffso`) устанавливает значение свойства `FindFirstSeriesOnly` в `true`. Это означает, что во время сериализации взаимодействий с целевой системой, тестовый механизм не будет строить все допустимые последовательности взаимодействий и проверять, что все они проводят в одно и тоже состояние спецификационной модели данных, а рассмотрит только одну допустимую последовательность взаимодействий.

Значение свойства `FindFirstSeriesOnly` может быть изменено с помощью функции `setFindFirstSeriesOnly` во время инициализации тестового сценария или дальнейшей его работы. По умолчанию, значение свойства равно `false`.

`-nt`
`--no-trace`

Выключает трассировку.

Дополнительная информация

Стандартный параметр `--no-trace` (в сокращенной форме `-nt`) выключает трассировку. Этот параметр нельзя использовать вместе со стандартными параметрами `-t`, `-tt` или `-tc`.

`-t`

Направляет трассировку в файл с указанным именем.

Дополнительная информация

Стандартный параметр `-t имя_файла` добавляет файл, указанный в следующим за ним параметре, в набор устройств для сохранения трассы. Если следующий параметр отсутствует или файл с указанным в нем именем невозможно открыть на запись, то работа тестового сценария аварийно завершается.

`-tc`

Направляет трассировку на консоль.

Дополнительная информация

Стандартный параметр `-tc` добавляет консоль в набор устройств для сохранения трассы. Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

`--trace-accidental`

Включает трассировку недостоверных переходов.

Дополнительная информация

Стандартный параметр `--trace-accidental` включает трассировку информации о несущественных переходах.

Трассировка информации о несущественных переходах может быть включена/отключена с помощью функции `setTraceAccidental` во время инициализации тестового сценария или дальнейшей его работы. По умолчанию трассировка информации о несущественных переходах отключена.

`-tt`

Направляет трассировку в файл с уникальным именем, составленным из имени сценария и текущего времени.

Дополнительная информация

Стандартный параметр `-tt` добавляет файл со сгенерированным именем `<scenario_name>-YYYY-MM-DD--HH-MM-SS.utt` в набор устройств для сохранения трассы. Если файл с таким именем невозможно открыть на запись, то работа тестового сценария аварийно завершается.

Параметр `-tt` является параметром по умолчанию: запуск теста без параметров командной строки эквивалентен запуску с параметром `-tt`.

`-uend`

Указывает выполнять тестирование до конца, несмотря на ошибки.

Дополнительная информация

Стандартный параметр `-uend` устанавливает значение режима завершения работы. Значение режима завершения может быть переопределено с помощью функции `setFinishMode` во время инициализации тестового сценария или дальнейшей его работы.

Если среди параметров сценария присутствует несколько стандартных параметров устанавливающих режим завершения работы, то механизм тестирования использует значение последнего из них.

`-uerr`

Указывает выполнять тестирование до возникновения первой ошибки (по умолчанию).

Дополнительная информация

Стандартный параметр `-uerr` устанавливает значение режима завершения работы. Значение режима завершения может быть переопределено с помощью функции `setFinishMode` во время инициализации тестового сценария или дальнейшей его работы.

Если среди параметров сценария присутствует несколько стандартных параметров устанавливающих режим завершения работы, то механизм тестирования использует значение последнего из них.

Для механизма построения тестов `ndfsm` у стандартного параметра `-uerr` можно указать значение — число ошибок, после возникновения которого тестирование прекращается. Для этого используется формат `-uerr=число_ошибок`.

`--trace-format`

Указывает формат сброса данных в трассу.

Дополнительная информация

Стандартный параметр `--trace-format` устанавливает формат, в котором сбрасываются в трассу данные сложных типов. Этот параметр влияет на то, в каком виде попадают в трассу (а в дальнейшем и в отчёты) значения итерационных переменных сценария, состояния сценария, значения параметров вызова методов и возвращаемых этими методами значений.

Допустимы два значения параметра: `--trace-format=xml` (по умолчанию) и `--trace-format=string`. В первом случае получается трасса, сохраняющая информацию о сложных структурах данных (если таковые использовались при работе сценария), которые позже преобразуются в отчётах в раскрываемое дерево. Во втором случае получается трасса меньшего объёма, требующая меньше ресурсов для генерации отчёта.

`--disabled-actions`

Указывает имя файла со списком сценарных методов, которые не будут вызываться.

Дополнительная информация

Значение стандартного параметра `--disabled-actions=имя_файла` задаёт имя файла со списком имён сценарных функций, которые не будут вызываться обходчиком. Файл должен существовать и содержать список имён сценарных функций, разделённых переводом строки.

12 Сервисы трассировки

Трассировщик тестовой системы CTESK предоставляет возможность сохранять информацию о процессе тестирования для ее последующего анализа. Для этого все компоненты тестовой системы автоматически трассируют данные о своей работе в специальном формате. Впоследствии трасса используется генератором отчетов для анализа результатов тестирования и построения различных отчетов.

Для пользователя трассировщик предоставляет интерфейс управления трассировкой и интерфейс трассировки сообщений.

12.1 Управление трассировкой

Функции управления трассировкой делятся на два класса: функции управления сохранением трассы и функции управления содержанием трассы.

Функции управления сохранением трассы, работают по следующим принципам. Трасса теста может одновременно сохраняется на нескольких устройствах. CTESK 2.8 поддерживается два вида устройств: консоль и файл. Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

Трассировщик сохраняет трассу на устройствах, входящих в набор устройств для сохранения трассы. Чтобы добавить устройство в этот набор используются функции из группы `addTraceTo…` Если добавляемое устройство уже принадлежит набору, то счетчик вхождения этого устройства в набор увеличивается.

Чтобы удалить устройство из набора устройств для сохранения трассы используются функции из группы removeTraceTo… Если удаляемое устройство было добавлено в набор несколько раз, то функция удаления только уменьшает счетчик вхождения этого устройства в набор. Реальное удаление устройства из набора происходит только тогда, когда счетчик вхождения обнуляется.

Изменять состав набора устройств для сохранения трассы разрешается только тогда, когда в тестовой системе не запущен ни один тестовый сценарий. В частности, в функции инициализации тестового сценария изменять этот набор не допускается.

Функции управления сохранением трассы:

- `addTraceToConsole`
- `removeTraceToConsole`
- `addTraceToFile`
- `removeTraceToFile`

Функции управления содержанием трассы определяют набор и формат сообщений трассировщика. В CTESK 2.8 доступны следующие функции управления содержанием трассы:

- `setTraceAccidental`
- `setTraceUserEnv`

Функция установки кодировки трассы:

- `setTraceEncoding`

`addTraceToConsole`

Функция добавляет консоль в набор устройств для сохранения трассы.

```
void addTraceToConsole( void );
```

Дополнительная информация

Функция `addTraceToConsole` добавляет консоль в набор устройств для сохранения трассы. Если консоль уже принадлежит этому набору, то счетчик вхождения консоли в набор увеличивается. Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

Функция `addTraceToConsole` может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Заголовочный файл: ts/c_tracer.h

`removeTraceToConsole`

Функция удаляет консоль из набора устройств для сохранения трассы.

```
void removeTraceToConsole( void );
```

Дополнительная информация

Функция `removeTraceToConsole` удаляет консоль из набора устройств для сохранения трассы. Если счетчик вхождения консоли в этот набор больше единицы, то счетчик уменьшается, а реального удаления консоли из набора не происходит. Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

Функция `removeTraceToConsole` может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Заголовочный файл: ts/c_tracer.h

`addTraceToFile`

Функция добавляет указанный файл в набор устройств для сохранения трассы.

```
bool addTraceToFile( const char* name );
```

Параметры

name

Имя файла, добавляемого в набор устройств для сохранения трассы. Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает `true`, если файл был добавлен успешно, и `false`, если эту операцию провести не удалось. Причиной неудачи, например, может быть невозможность открыть файл с данным именем для записи.

Дополнительная информация

Функция `addTraceToFile` добавляет указанный файл в набор устройств для сохранения трассы. Если этот файл уже принадлежит набору, то счетчик вхождения файла в набор увеличивается.

Функция `addTraceToFile` может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Заголовочный файл: ts/c_tracer.h

`removeTraceToFile`

Функция удаляет указанный файл из набора устройств для сохранения трассы.

```
bool removeTraceToFile( const char* name );
```

Параметры

name

Имя файла, удаляемого из набор устройств для сохранения трассы. Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает `false`, если файл с данным именем не принадлежит набору устройств для сохранения трассы, и `true` в случае успешного удаления файла.

Дополнительная информация

Функция `removeTraceToFile` удаляет указанный файл из набора устройств для сохранения трассы. Если счетчик вхождения этого файла в набор больше единицы, то счетчик уменьшается, а реального удаления файла из набора не происходит.

Функция `removeTraceToFile` может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Заголовочный файл: ts/c_tracer.h

`setTraceAccidental`

Функция включает/отключает трассировку информации о *несущественных переходах*. Под несущественными переходами понимаются переходы, соответствующие вызовам сценарных функций, в течении которых не происходит вызова ни одной спецификационной функции.

```
bool setTraceAccidental( bool enable);
```

Параметры

`enable`

Если значение параметра — `true`, то функция включает трассировку информации о несущественных переходах, иначе — отключает.

Возвращаемое значение

Функция возвращает предыдущее значение свойства сохранения информации о несущественных переходах.

Дополнительная информация

По умолчанию трассировщик не сохраняет информацию о несущественных переходах, поэтому при возникновении такой необходимости нужно использовать функцию `setTraceAccidental`.

Функция `setTraceAccidental` не может быть вызвана во время работы сценарной функции.

Заголовочный файл: ts/c_tracer.h

`setTraceUserEnv`

Функция setTraceUserEnv устанавливает значения пользовательских переменных окружения. Значения этих переменных сбрасываются в начале трассы вместе с информацией о системе.

```
void setTraceUserEnv( const char* name, const char* value);
```

Параметры

`name`

Имя пользовательской переменной окружения.

`value`

Значение переменной.

Дополнительная информация

Устанавливаемые функцией `setTraceUserEnv` значения переменных могут описывать, например параметры удаленных систем, которые не могут быть определены тестом автоматически. Они видны в отчете, а также могут использоваться для идентификации обнаруженных ошибок в БД известных ошибок.

Функция должна быть вызвана до начала работы сценария.

Заголовочный файл: ts/c_tracer.h

`setTraceEncoding`

Функция `setTraceEncoding` устанавливает кодировку трассы.

```
void setTraceAccidental( const char* encoding);
```

Параметры

`encoding`

Идентификатор кодировки трассы.

Дополнительная информация

Функция `setTraceEncoding` устанавливает кодировку трассы. По умолчанию используется кодировка UTF-8.

Установка кодироки трассы необходима, в частности, для того чтобы локализованные строки, используемые в качестве имен ветвей функциональности, названий подсистем или пользовательских сообщений, корректно отображались в отчетах о проведенном тестировании.

Функция должна быть вызвана до начала работы сценария.

Заголовочный файл: ts/c_tracer.h

12.2 Трассировка сообщений

В CTESK 2.8 существует только один вид сообщений трассировщика доступный для пользователя. Эти сообщения так и называются: сообщения пользователя. Они играют вспомогательную роль и используются в основном для ручного анализа трассы. Но в некоторых отчетах об ошибках пользовательские сообщения отображаются для облегчения анализа ошибочных ситуаций.

Для трассировки сообщений пользователя используются функции:

- `traceUserInfo`
- `traceFormattedUserInfo`

`traceUserInfo`

Функция `traceUserInfo` записывает в трассу сообщение пользователя.

```
void addTraceToConsole( const char* info );
```

Параметры

`info`

Указатель на строку, завершающуюся нулевым символом, которая содержит пользовательское сообщение.

Дополнительная информация

Функция traceUserInfo записывает сообщение пользователя в трассу. Сообщения пользователя играют вспомогательную роль и используются в основном для ручного анализа трассы. Но в некоторых отчетах об ошибках пользовательские сообщения отображаются для облегчения анализа ошибочных ситуаций.

Заголовочный файл: ts/c_tracer.h

`traceFormattedUserInfo`

Функция записывает в трассу форматированное сообщение пользователя.

```
void addTraceToConsole( const char* format, ... );
```

Параметры

`format`

Указатель на строку, завершающуюся нулевым символом, которая содержит формат пользовательского сообщения. В строке допускаются все спецификаторы преобразований, допустимые в стандартной функции `printf`, а также спецификатор `$(obj)` для преобразования в строку спецификационного объекта. Спецификаторы `$(obj)` должны располагаться перед спецификаторами функции `printf`.

Дополнительная информация

Функция `traceFormattedUserInfo` форматирует и записывает сообщение пользователя в трассу. Сообщения пользователя играют вспомогательную роль и используются в основном для ручного анализа трассы. Но в некоторых отчетах об ошибках пользовательские сообщения отображаются для облегчения анализа ошибочных ситуаций.

Заголовочный файл: ts/c_tracer.h

13 Сервисы регистрации отложенных реакций

Тестовая система CTESK поддерживает тестирование систем с отложенными реакциями. Системами с отложенными реакциями называются такие системы, которые могут участвовать в нескольких взаимодействиях одновременно или которые могут сами инициировать взаимодействия с окружением.

Одной из важнейших задач при тестировании систем с отложенными реакциями является получение всей необходимой информации о взаимодействиях с целевой системой. Эта информация требуется тестовой системе CTESK для проверки соответствия реального поведения целевой системы требованиям, описанным в спецификациях, поскольку именно эти взаимодействия и отражают поведение целевой системы с отложенными реакциями.

Тестовая система автоматически регистрирует все взаимодействия, которые инициируются посредством вызова спецификационной функции внутри процесса тестовой системы. Все остальные взаимодействия должны быть зарегистрированы разработчиком теста в специальном компоненте тестовой системы — регистраторе взаимодействий.

Каждое взаимодействие с целевой системой характеризуется каналом, в котором оно произошло. Для идентификации каналов в тестовой системе используются идентификаторы каналов взаимодействия.

Дополнительно, для удобства регистрации отложенных взаимодействий, тестовая система предоставляет сервис регистрации функций-сборщиков реакций.

13.1 Каналы взаимодействия

Каждое взаимодействие с целевой системой характеризуется каналом, в котором оно произошло. Все взаимодействия внутри одного канала линейно упорядочены. Поэтому тестовая система считает, что, если два взаимодействия характеризуются одним каналом, то первое зарегистрированное в ней произошло раньше второго.

Идентификаторы каналов взаимодействия, использующиеся для их идентификации внутри тестовой системы, имеют тип `ChannelID`.

Существует две предопределенные константы этого типа:

- *`WrongChannel`*
- *`UniqueChannel`*

Для получения свободного идентификатора канала и последующего его освобождения предназначены функции:

- `getChannelID`
- `releaseChannelID`

`ChannelID`

Тип `ChannelID` используется для идентификации каналов взаимодействия внутри тестовой системы.

```
typedef long ChannelID;
```

Дополнительная информация

Тип ChannelID используется для идентификации каналов взаимодействия внутри тестовой системы. Существует две константы этого типа *`WrongChannel`* и *`UniqueChannel`*.

Функция `getChannelID` возвращает свободный идентификатор канала. После того как идентификатор канала становится ненужным, его можно освободить с помощью функции `releaseChannelID`.

Корректными идентификаторами каналов считаются константа *`UniqueChannel`* и все идентификаторы, полученные с помощью функции `getChannelID` и отличные от *`WrongChannel`*.

Заголовочный файл: ts/register.h

`WrongChannel`

Константа *`WrongChannel`* обозначает некорректный идентификатор канала взаимодействия.

```
extern const ChannelID WrongChannel;
```

Дополнительная информация

Константа *`WrongChannel`* обозначает некорректный идентификатор канала взаимодействия. Эта константа используется для вспомогательных целей. Например, функция `getChannelID` возвращает эту константу, если не может выделить свободный идентификатор канала.

Заголовочный файл: ts/register.h

`UniqueChannel`

Константа *`UniqueChannel`* используется для идентификации уникального канала, в котором произошло одно взаимодействие и других взаимодействий произойти не может в принципе.

```
extern const ChannelID UniqueChannel;
```

Дополнительная информация

Константа *`UniqueChannel`* используется для идентификации уникального канала, в котором произошло одно взаимодействие и других взаимодействий произойти не может в принципе. Эта константа часто используется, когда понятие каналов взаимодействия не применяется при моделировании целевой системы.

Заголовочный файл: ts/register.h

`getChannelID`

Функция `getChannelID` возвращает свободный идентификатор канала взаимодействия.

```
ChannelID getChannelID( void );
```

Возвращаемое значение

Функция возвращает свободный идентификатор канала взаимодействия, если таковой существует, и *`WrongChannel`* — в противном случае.

Дополнительная информация

Если идентификатор канала становится ненужным, его можно освободить с помощью функции `releaseChannelID`.

Заголовочный файл: ts/register.h

`releaseChannelID`

Функция `releaseChannelID` освобождает указанный идентификатор канала взаимодействия.

```
void getChannelID( ChannelID chid );
```

Параметры

`chid`

Освобождаемый идентификатор канала взаимодействия. Параметр должен идентификатором канала, полученный ранее с помощью функции `getChannelID`.

Дополнительная информация

Заголовочный файл: ts/register.h

13.2 Регистратор взаимодействий

Тестовая система автоматически регистрирует все взаимодействия, которые инициируются посредством вызова спецификационной функции внутри процесса тестовой системы. При этом считается, что взаимодействие происходит в канале с идентификатором, заданным свойством `StimulusChannel`. Для управления этим свойством предназначены функции:

- `setStimulusChannel`
- `getStimulusChannel`

По умолчанию, это свойство принимает значение *`UniqueChannel`*.

Все остальные взаимодействия должны быть зарегистрированы разработчиком теста в регистраторе взаимодействий. Для этого предназначены следующие функции:

- `registerReaction`
- `registerReactionWithTimeMark`
- `registerReactionWithTimeInterval`
- `registerWrongReaction`
- `registerStimulusWithTimeInterval`

`setStimulusChannel`

Функция `setStimulusChannel` устанавливает значение свойства `StimulusChannel`.

```
ChannelID setChannelID( ChannelID chid );
```

Параметры

`chid`

Идентификатор канала взаимодействия, который будет использоваться тестовой системой при автоматической регистрации взаимодействий. Параметр должен быть корректным идентификатором канала.

Возвращаемое значение

Функция возвращает предыдущее значение свойства `StimulusChannel`.

Дополнительная информация

Свойство `StimulusChannel` содержит идентификатор канала, используемый тестовой системой для идентификации канала, в котором происходят взаимодействия, инициированные посредством вызова спецификационной функции внутри процесса тестовой системы. По умолчанию, это свойство принимает значение *`UniqueChannel`*.

Для доступа к текущему значению свойства `StimulusChannel` предназначена функция `getStimulusChannel`.

Заголовочный файл: ts/register.h

`getStimulusChannel`

Функция `getStimulusChannel` возвращает значение свойства `StimulusChannel`.

```
ChannelID getChannelID( void );
```

Возвращаемое значение

Функция возвращает предыдущее значение свойства `StimulusChannel`.

Дополнительная информация

Свойство `StimulusChannel` содержит идентификатор канала, используемый тестовой системой для идентификации канала, в котором происходят взаимодействия, инициированные посредством вызова спецификационной функции внутри процесса тестовой системы. По умолчанию, это свойство принимает значение *`UniqueChannel`*.

Для изменения значения свойства `StimulusChannel` предназначена функция `setStimulusChannel`.

Заголовочный файл: ts/register.h

`registerReaction`

Функция `registerReaction` предназначена для регистрации реакций, полученных от целевой системы.

```
void registerReaction( ChannelID chid, const char* name,
SpecificationID reactionID, Object* data );
```

Параметры

`chid`

Идентификатор канала взаимодействия, который будет использоваться тестовой системой при автоматической регистрации взаимодействий. Параметр должен быть корректным идентификатором канала.

`name`

Имя реакции, используемое только для трассировки. Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем реакции *`reactionID`*.

`reactionID`

Идентификатор реакции, которому соответствует регистрируемое взаимодействие.

`data`

Данные, полученные от целевой системы в модельном представлении. Тип параметра должен совпадать с типом возвращаемого значения реакции *`reactionID`*.

Дополнительная информация

Основными свойствами взаимодействия являются имя реакции *`reactionID`* и данные *`data`*, полученные во время взаимодействия. Тип полученных данных должен совпадать с типом возвращаемого значения реакции.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о получении некорректной реакции с помощью функции `registerWrongReaction`.

Заголовочный файл: ts/register.h

`registerReactionWithTimeMark`

Функция `registerReactionWithTimeMark` предназначена для регистрации реакций, полученных от целевой системы, с указанием временной метки момента получения.

```
void registerReactionWithTimeMark( ChannelID chid, const char* name,
SpecificationID reactionID, Object* data, TimeMark mark );
```

Параметры

chid

Идентификатор канала взаимодействия, который будет использоваться тестовой системой при автоматической регистрации взаимодействий. Параметр должен быть корректным идентификатором канала.

name

Имя реакции, используемое только для трассировки. Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем реакции *reactionID*.

reactionID

Идентификатор реакции, которому соответствует регистрируемое взаимодействие.

data

Данные, полученные от целевой системы в модельном представлении. Тип параметра должен совпадать с типом возвращаемого значения реакции *reactionID*.

mark

Временная метка момента взаимодействия.

Дополнительная информация

Основными свойствами взаимодействия являются имя реакции *reactionID* и данные *data*, полученные во время взаимодействия. Тип полученных данных должен совпадать с типом возвращаемого значения реакции.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о получении некорректной реакции с помощью функции `registerWrongReaction`.

Заголовочный файл: ts/register.h

`registerReactionWithTimeInterval`

Функция `registerReactionWithTimeInterval` предназначена для регистрации реакций, полученных от целевой системы, с указанием временного интервала, в котором оно происходило.

```
void registerReactionWithTimeInterval
     ( ChannelID chid
     , const char* name
     , SpecificationID reactionID
     , Object* data
     , TimeInterval interval
     );
```

Параметры

chid

Идентификатор канала взаимодействия, который будет использоваться тестовой системой при автоматической регистрации взаимодействий. Параметр должен быть корректным идентификатором канала.

name

Имя реакции, используемое только для трассировки. Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем реакции *reactionID*.

reactionID

Идентификатор реакции, которому соответствует регистрируемое взаимодействие.

data

Данные, полученные от целевой системы в модельном представлении. Тип параметра должен совпадать с типом возвращаемого значения реакции *reactionID*.

interval

Временной интервал, в котором происходило взаимодействие. При этом имеется ввиду, что взаимодействие происходило где-то внутри интервала, а не занимало весь интервал целиком.

Дополнительная информация

Основными свойствами взаимодействия являются имя реакции *reactionID* и данные *data*, полученные во время взаимодействия. Тип полученных данных должен совпадать с типом возвращаемого значения реакции.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о получении некорректной реакции с помощью функции registerWrongReaction.

Заголовочный файл: ts/register.h

registerWrongReaction

Функция registerWrongReaction предназначена для уведомления тестовой системы о получении некорректной реакции, которую невозможно преобразовать в модельное представление. Реакцией называется взаимодействие, инициированное целевой системой.

```
void registerWrongReaction( const char* info );
```

Параметры

info

Описание некорректной реакции, используемое при анализе результатов тестирования. Параметр может быть нулевым указателем.

Дополнительная информация

Функция registerWrongReaction предназначена для уведомления тестовой системы о получении некорректной реакции, которую невозможно преобразовать в модельное представление. Реакцией называется взаимодействие, инициированное целевой системой.

После регистрации некорректной реакции тестовая система завершит анализ текущего тестового воздействия с отрицательным вердиктом.

Заголовочный файл: ts/register.h

`registerStimulusWithTimeInterval`

Функция `registerStimulusWithTimeInterval` предназначена для регистрации стимула, не зарегистрированного тестовой системой автоматически. Стимулом называется взаимодействие с целевой системой, инициированное тестом.

```
void registerReactionWithTimeInterval
     ( ChannelID chid
     , const char* name
     , SpecificationID stimulusID
     , TimeInterval interval
     , ...
     );
```

Параметры

`chid`

Идентификатор канала взаимодействия, который будет использоваться тестовой системой при автоматической регистрации взаимодействий. Параметр должен быть корректным идентификатором канала.

`name`

Имя реакции, используемое только для трассировки. Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем реакции *`reactionID`*.

`stimulusID`

Идентификатор спецификационной функции, которому соответствует регистрируемое взаимодействие.

`interval`

Временной интервал, в котором происходило взаимодействие. При этом имеется ввиду, что взаимодействие происходило где-то внутри интервала, а не занимало весь интервал целиком.

`arguments`

Дополнительные аргументы, которые должны передаваться строго в следующем порядке:

1. Список значений параметров спецификационной функции до ее вызова.
2. Список значений параметров спецификационной функции после ее вызова.
3. Значение, которое вернула данная спецификационная функция (если тип возвращаемого значения отличен от `void`).

Дополнительная информация

Функция `registerStimulusWithTimeInterval` предназначена для регистрации стимула, не зарегистрированного тестовой системой автоматически. Так как все стимулы, которые инициируются посредством вызова спецификационной функции внутри процесса тестовой системы, регистрируются автоматически, то регистрировать вручную необходимо только те стимулы, которые инициируются либо извне процесса тестовой системы, либо не посредством вызова спецификационной функции.

Основными свойствами взаимодействия являются имя спецификационной функции *`stimulusID`* и данные, передаваемые через дополнительные аргументы.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные во время взаимодействия от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о некорректном поведении целевой системы во время взаимодействия, вызвав функцию `registerWrongReaction` в основном процессе тестовой системы.

Заголовочный файл: ts/register.h

13.3 Сервис регистрации функций-сборщиков реакций

Для удобства регистрации отложенных взаимодействий тестовая система предоставляет сервис регистрации функций-сборщиков реакций. Сервис устроен следующим образом. В тестовой системе регистрируются функции, которые вызываются ей по истечении времени стабилизации после каждого тестового воздействия. За время стабилизации целевая система должна выдать все ожидаемые от нее реакции и перейти в стационарное состояние. Во время вызова функции-сборщики должны собрать все данные о полученных реакциях и зарегистрировать их в регистраторе взаимодействий.

Для регистрации функций-сборщиков предусмотрены следующие функции:

- `registerReactionCatcher`
- `unregisterReactionCatcher`
- `unregisterReactionCatchers`

Параметром каждой из этих функций является функция-сборщик, тип этих функций `ReactionCatcherFuncType`.

`ReactionCatcherFuncType`

Тип `ReactionCatcherFuncType` используется для регистрации функций-сборщиков реакций в тестовой системе.

```
typedef bool (*ReactionCatcherFuncType)(void*);
```

Дополнительная информация

Заголовочный файл: ts/timemark.h

`registerReactionCatcher`

Функция `registerReactionCatcher` регистрирует в тестовой системе функцию-сборщик реакций и ее вспомогательные данные.

```
void registerReactionCatcher(
        ReactionCatcherFuncType catcher,
        void*                   par
                            );
```

Параметры

`catcher`

Указатель на функцию-сборщик реакций.

Параметр не должен быть нулевым указателем.

`par`

Вспомогательные данные регистрируемой функции.

Параметр может быть нулевым указателем.

Дополнительная информация

Функция `registerReactionCatcher` регистрирует в тестовой системе функцию-сборщик реакций и ее вспомогательные данные.

Когда тестовая система вызывает функцию-сборщик, то передает ей в качестве параметра вспомогательные данные.

Можно зарегистрировать в тестовой системе одну функцию-сборщик реакций несколько раз с различными вспомогательными данными. При этом функция будет вызываться соответствующее число раз с различными значениями параметра.

Заголовочный файл: ts/register.h

unregisterReactionCatcher

Функция `unregisterReactionCatcher` удаляет из тестовой системы регистрационную запись о данной функции-сборщике реакций, зарегистрированной с указанными вспомогательными данными.

```
bool unregisterReactionCatcher(
            ReactionCatcherFuncType catcher,
            void*                   par
                                );
```

Параметры

`catcher`

Указатель на функцию-сборщик реакций.

Параметр не должен быть нулевым указателем.

`par`

Вспомогательные данные, переданные при регистрации этой функции.

Возвращаемое значение

Функция возвращает `false`, если данная функция с данными вспомогательными данными не была ранее зарегистрирована, и `true` — иначе.

Дополнительная информация

Функция `unregisterReactionCatcher` удаляет из тестовой системы регистрационную запись о данной функции-сборщике реакций, зарегистрированной с указанными вспомогательными данными.

Для одновременного удаления всех регистрационных записей о данной функции-сборщике реакций используется функция `unregisterReactionCatchers`.

Заголовочный файл: ts/register.h

unregisterReactionCatchers

Функция `unregisterReactionCatchers` удаляет из тестовой системы все регистрационные записи о данной функции-сборщике реакций.

```
bool unregisterReactionCatchers( ReactionCatcherFuncType catcher );
```

Параметры

`catcher`

Указатель на функцию-сборщик реакций.

Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает `false`, если данная функция-сборщик не была ранее зарегистрирована, и `true` — иначе.

Дополнительная информация

Функция `unregisterReactionCatchers` удаляет из тестовой системы все регистрационные записи о данной функции-сборщике реакций.

Для удаления только одной регистрационной записи о данной функции-сборщике с конкретными вспомогательными данными используется функция `unregisterReactionCatcher`.

Заголовочный файл: ts/register.h

14 Библиотека спецификационных типов

Библиотека спецификационных типов содержит стандартные функции для работы со спецификационными типами (создание, копирование, сравнение, построение строкового представления), а также предопределенные спецификационные типы для стандартных типов языка C (`char`, `short`, `int`, `long`, `float`, `double`, `void*`, строки `char*`), для комплексных чисел и типа с единственным значением, и для контейнерных типов (список, множество, отображение).

14.1 Стандартные функции

Стандартные функции могут использоваться с любыми спецификационными ссылками. Результат выполнения функции будет зависеть от типа спецификационных ссылок. Например, если сравниваются две спецификационные ссылки разных типов, то результат всегда будет отрицательный, а если ссылки имеют одинаковый тип, то результат сравнения будет определяться функцией, заданной при описании этого типа.

Тип спецификационных ссылок определяется указателем на дескриптор спецификационного типа. Константа дескриптор всегда имеет имя, состоящее из имени спецификационного типа с префиксом `type_`:

```
const Type type_имя_спецификационного_типа;
```

- **Функция создания ссылок**
  Функция создает объект переданного типа и возвращает ссылку на него.
- **Функция получения типа ссылки**
  Функция определяет тип спецификационной ссылки и возвращает дескриптор этого типа.
- **Функция копирования значений по ссылкам**
  Функция копирует содержимое объекта, находящегося по ссылке *`src`*, в содержимое объекта, находящегося по ссылке *`dst`*.
- **Функция клонирования объекта**
  Функция помогает получить новый объект с тем же содержимым — клонировать объект.
- **Функция сравнения значений по ссылкам**
  Функция сравнивает объекты, переданные ей по спецификационным ссылкам, возвращая целочисленный результат.
- **Функция определения эквивалентности значений по ссылкам.**
  Функция сравнивает объекты, переданные ей по спецификационным ссылкам, возвращая логическое значение.
- **Функция построения строкового представления значения по ссылке**
  Функция возвращает ссылку на значение типа `String` — строковое представление спецификационного типа.
- **Функция построения XML представления значения по ссылке**
  Функция возвращает ссылку на значение типа `String` — XML представление спецификационного типа.

14.1.1 Функция создания ссылок

Функция создает объект переданного типа и возвращает ссылку на него.

```
Object* create(const Type *type, ...)
```

Параметры

`type`

Указатель на дескриптор спецификационного типа.

...

Параметры инициализации типа.

Возвращаемое значение

Функция возвращает спецификационную ссылку на созданный объект.

Дополнительная информация

Параметры инициализации должны соответствовать типу: для всех предопределенных типов параметры описаны в соответствующих разделах, а параметры инициализации пользовательских спецификационных типов задаются при описании этих типов.

```
Integer* ref = create(&type_Integer, 28); // ref → 28
```

В приведенном выше примере создается и инициализируется ссылка библиотечного спецификационного типа `Integer` — спецификационного аналога типа `int`.

При использовании функции `create` возможны ошибки из-за использования нетипизированного списка аргументов `...`. Для исключения таких ошибок рекомендуется для каждого спецификационного типа создавать функцию `create_имя_спецификационного_типа` и вызывать `create` из нее, например:

```
specification typedef struct {int a; int b;} IntPair ={};

IntPair* create_IntPair(int a, int b)
{
  return create(&type_IntPair, a, b);
}
```

При использовании `create` вне функции, имя которой имеет префикс `create_`, транслятор выдаст предупреждение:

```
warning: call create() out of create_... function
```

14.1.2 Функция получения типа ссылки

Функция определяет тип спецификационной ссылки и возвращает дескриптор этого типа.

```
const Type *type( Object* ref )
```

Параметры

ref

Спецификационная ссылка на объект спецификационного типа.

Возвращаемое значение

Функция возвращает указатель на константу-дескриптор спецификационного типа, на значение которого ссылается указатель *ref*.

```
Integer* ref = create(&type_Integer, 28);
if (type(ref) == &type_Integer) // истинно
```

14.1.3 Функция копирования значений по ссылкам

Функция копирует содержимое объекта, находящегося по ссылке *src*, в содержимое объекта, находящегося по ссылке *dst*.

```
void copy( Object* src, Object* dst )
```

Параметры

src

Ссылка на объект, предназначенный для копирования.

dst

Ссылка на объект, в который будет произведено копирование.

Дополнительная информация

Ссылки должны быть ненулевыми и одного типа, то есть они должны иметь одинаковые дескрипторы типов. Если эти условия не выполняются, то во время выполнения происходит завершение программы с сообщением об ошибке.

```
Integer* ref1 = create(&type_Integer, 28); // ref1 → 28
Integer* ref2 = create(&type_Integer, 47); // ref2 → 47
copy(ref1,ref2);                           // ref1 → 28, ref2 → 28
```

В приведенном выше примере ссылки *ref1* и *ref2* после инициализации ссылаются на разные значения спецификационного типа `Integer`. После вызова функции `copy()` значение по ссылке *ref2* становится эквивалентным значению по ссылке *ref1*.

14.1.4 Функция клонирования объекта

Функция помогает получить новый объект с тем же содержимым — клонировать объект.

```
Object* clone( Object* ref )
```

Параметры

ref

Ссылка на объект, предназначенный для клонирования.

Дополнительная информация

Функция выделяет память для значения типа, на который ссылается *ref*, инициализирует выделенную память значением, эквивалентным значению по ссылке *ref*, и возвращает указатель на выделенную и инициализированную память.

```
Integer* ref1 = create(&type_Integer, 28);
String*  ref2 = clone(ref1);
```

ref1 инициализирована значением `28`. После вызова `clone()` значения по ссылкам *ref1* и *ref2* становятся эквивалентными.

14.1.5 Функция сравнения значений по ссылкам

Функция сравнивает объекты, переданные ей по спецификационным ссылкам, возвращая целочисленный результат.

```
int compare(Object* left, Object* right)
```

Параметры

left

Спецификационная ссылка на первый сравниваемый объект.

right

Спецификационная ссылка на второй сравниваемый объект.

Возвращаемое значение

- В случае эквивалентности значений по переданным ссылкам функция возвращает нулевое значение.
- Если значения не эквивалентны, функция возвращает ненулевое значение, которое может интерпретироваться в зависимости от типа сравниваемых значений. Например, для библиотечного типа `String` результат будет таким же, как у функции `strcmp()` для типа языка C `char*`.
- Если параметры имеют несравнимые типы, то есть типы ссылок неодинаковые, не являются подтипами одного типа, и тип одной ссылки не является подтипом другой (см. Инварианты типов), то функция возвращает ненулевое значение.
- Если одна из ссылок нулевая, а другая нет, то возвращается ненулевое значение.
- Если обе ссылки нулевые, то возвращается ноль.

```
if (!compare(ref1,ref2)) {/* значения эквивалентны */
  ...
}
else {/* значения не эквивалентны */
  ...
}
```

14.1.6 Функция определения эквивалентности значений по ссылкам.

Функция сравнивает объекты, переданные ей по спецификационным ссылкам, возвращая логическое значение.

```
bool equals(Object* self, Object* ref)
```

Параметры

self

Спецификационная ссылка на первый сравниваемый объект.

ref

Спецификационная ссылка на второй сравниваемый объект.

Возвращаемое значение

- Функция возвращает значение `true`, если значения по переданным ссылкам эквивалентны, и `false` в противном случае.
- Если параметры имеют различный тип, то функция возвращает `false`.
- Если одна из ссылок нулевая, а другая нет, то возвращается `false`.
- Если обе ссылки нулевые, то возвращается `true`.

```
if (equals(ref1,ref2)) {/* значения эквивалентны */
  ...
}
else {/* значения не эквивалентны */
  ...
}
```

14.1.7 Функция построения строкового представления значения по ссылке

Функция возвращает ссылку на значение типа `String` — строковое представление спецификационного типа.

```
String* toString(Object* ref)
```

Параметры

ref

Указатель на объект спецификационного типа.

Возвращаемое значение

Функция возвращает ссылку на объект типа `String`, в котором содержится строковое представление переданного функции объекта.

```
Integer* ref = create(&type_Integer, 28);       // ref → 28
String*  str = toString(ref);                   // str → "28"
printf("*ref == '%s'\n", toCharArray_String(str));
```

14.1.8 Функция построения XML представления значения по ссылке

Функция возвращает ссылку на значение типа `String` — XML представление спецификационного типа.

```
String* toXML(Object* ref)
```

Параметры

ref

Указатель на объект спецификационного типа.

Возвращаемое значение

Функция возвращает ссылку на объект типа `String`, в котором содержится XML представление переданного функции объекта.

```
Integer* ref = create(&type_Integer, 28);       // ref → 28
String*  str = toXML(ref);
// str → "<object kind=\"spec\" type=\"Integer\" text=\"28\"/>"
printf("*ref == '%s'\n", toCharArray_String(str));
```

14.2 Предопределенные спецификационные типы

Предопределенными типами удобно пользоваться в спецификациях (например, для моделирования реализационного состояния), поскольку они предоставляют уже готовую, универсальную, гарантированно безошибочную функциональность.

14.2.1 Char

Тип `Char` является спецификационным аналогом встроенного типа языка C `char`.

Дополнительная информация
Функции, определенные для типа `Char`:

- создание спецификационной ссылки;
- извлечение значения типа `char`.

Заголовочный файл: atl/char.h

`create_Char`
Функция создает спецификационную ссылку типа `Char`.

```
Char* create_Char( char c )
```

Параметры

c

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `Char` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
Char* ch1 = create(&type_Char, 'a');
Char* ch2 = create_Char('a');
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`value_Char`
Функция получает значение `char`, содержащееся в спецификационном типе.

```
char value_Char( Char* c )
```

Параметры

c

Спецификационная ссылка за значение типа `Char`.

Возвращаемое значение

Символ, являющийся значением спецификационной ссылки *c*.

Дополнительная информация

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

```
Char* ch = create_Char('a');
```

```
char  val = *ch;
```

Функция `value_Char()` обеспечивает контроль типов во время исполнения и ей можно передавать указатель `Object*` без предварительного приведения:

```
List* l;
char val;
...
val = value_Char(get_List(l,0)); // Object* get_List(List*,int)
```

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции `type()`:

```
Object* o = get_List(l,0));
if (type(o) == &type_Char) {
  val = value_Char(o);
}
```

14.2.2 `Integer` и `UInteger`

Типы `Integer` и `UInteger` являются спецификационными аналогами встроенных типов языка C `int` и `unsigned int`.

Дополнительная информация

Функции, определенные для типов `Integer` и `UInteger`:

- создание спецификационной ссылки;
- извлечение значения типа `int` или `unsigned int`.

Заголовочный файл: `atl/Integer.h`

`create_Integer`, `create_UInteger`

Функции создает спецификационные ссылки типов `Integer` и `UInteger`.

```
Integer* create_Integer( int i )
UInteger* create_UInteger( unsigned int i )
```

Параметры

i

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `Integer` (`UInteger`) на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
Integer* i1 = create(&type_Integer, -28);
Integer* i2 = create_Integer(-28);


UInteger* i1 = create(&type_UInteger, 28);
UInteger* i2 = create_UInteger(28);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`value_Integer`, `value_UInteger`

Функция получает значение `int`, содержащееся в спецификационном типе.

```
int value_Integer ( Integer* i )
unsigned int value_UInteger ( UInteger* i )
```

Параметры

i

Спецификационная ссылка за значение типа `Integer` или `UInteger`.

Дополнительная информация

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

```
Integer* i = create_Integer(-28);
int  val = *i;
```

Функция `value_Integer()` обеспечивает контроль типов во время исполнения и ей можно передавать указатель `Object*` без предварительного приведения:

```
List* l;
int val;
...
val = value_Integer(get_List(l,0)); // Object* get_List(List*,int)
```

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции `type()`:

```
Object* o = get_List(l,0));
if (type(o) == &type_Integer) {
  val = value_Integer(o);
}
```

14.2.3 `Short` и `UShort`

Типы `Short` и `UShort` являются спецификационными аналогами встроенных типов языка C `short` и `unsigned short`.

Дополнительная информация

Функции, определенные для типов `Short` и `UShort`:

- создание спецификационной ссылки;
- извлечение значения типа `short` или `unsigned short`.

Заголовочный файл: `atl/short.h`

`create_Short`, `create_UShort`

Функции создают спецификационные ссылки типов `Short` и `UShort`.

```
Short* create_Short ( short i )
UShort* create_UShort ( unsigned short i )
```

Параметры

i

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `Short` (`UShort`) на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
Short* i1 = create(&type_Short, -28);
Short* i2 = create_Short(-28);
UShort* i1 = create(&type_UShort, 28);
UShort* i2 = create_UShort(28);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`value_Short`, `value_UShort`

Функция получает значение `short`, содержащееся в спецификационном типе.

```
short value_Short ( Short* i )
unsigned short value_UShort ( UShort* i )
```

Параметры

i

Спецификационная ссылка за значение типа `Short` или `UShort`.

Дополнительная информация

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

```
Short* i = create_Short(-28);
short  val = *i;
```

Функция `value_Short()` обеспечивает контроль типов во время исполнения и ей можно передавать указатель `Object*` без предварительного приведения:

```
List* l;
short val;
...
val = value_Short(get_List(l,0)); // Object* get_List(List*,int)
```

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции `type()`:

```
Object* o = get_List(l,0));
if (type(o) == &type_Short) {
  val = value_Short(o);
}
```

14.2.4 Long и ULong

Тип Long и ULong являются спецификационными аналогами встроенных типов языка C long int и unsigned long int.

Дополнительная информация

Функции, определенные для типа Long:

- создание спецификационной ссылки;
- извлечение значения типа long int или unsigned long int.

Заголовочный файл: atl/long.h

create_Long, create_ULong

Функции создают спецификационные ссылки типов Long и ULong.

```
Long* create_Long( long i )
ULong* create_ULong ( unsigned long i )
```

Параметры

i

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа Long (ULong) на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций create.

```
Long* i1 = create(&type_Long, -28);
Long* i2 = create_Long(-28);
ULong* i1 = create(&type_ULong, 28);
ULong* i2 = create_ULong(28);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование create не является типобезопасным и вызывает предупреждение транслятора (см. описание функции create).

value_Long, value_ULong

Функция получает значение long, содержащееся в спецификационном типе.

```
long value_Long ( Long* i )
unsigned long value_ULong ( ULong* i )
```

Параметры

i

Спецификационная ссылка за значение типа Long или ULong.

Дополнительная информация

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

```
Long* i = create_Long(-28);
long  val = *i;
```

Функция `value_Long()` обеспечивает контроль типов во время исполнения и ей можно передавать указатель `Object*` без предварительного приведения:

```
List* l;
long val;
...
val = value_Long(get_List(l,0)); // Object* get_List(List*,int)
```

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции `type()`:

```
Object* o = get_List(l,0));
if (type(o) == &type_Long) {
  val = value_Long(o);
}
```

14.2.5 `Float`

Тип `Float` является спецификационным аналогом встроенного типа языка C `float`.

Дополнительная информация
Функции, определенные для типа Float:

- создание спецификационной ссылки;
- извлечение значения типа `float`.

Заголовочный файл: `atl/float.h`

create_Float
Функция создает спецификационную ссылку типа `Float`.

```
Float* create_Float( float f )
```

Параметры

f

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `Float` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
Float* f1 = create(&type_Float, 3.14);
Float* f2 = create_Float(3.14);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`value_Float`
Функция получает значение `float`, содержащееся в спецификационном типе.

```
float value_Float( Float* f )
```

Параметры

f

Спецификационная ссылка за значение типа `Float`.

Дополнительная информация

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

```
Float* f = create_Float(3.14);
float  val = *f;
```

Функция `value_Float()` обеспечивает контроль типов во время исполнения и ей можно передавать указатель `Object*` без предварительного приведения:

```
List* l;
float val;
...
val = value_Float(get_List(l,0)); // Object* get_List(List*,int)
```

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции `type()`:

```
Object* o = get_List(l,0));
if (type(o) == &type_Float) {
  val = value_Float(o);
}
```

14.2.6 `Double`

Тип `Double` является спецификационным аналогом встроенного типа языка C `double`.

Дополнительная информация

Функции, определенные для типа `Double`:

- создание спецификационной ссылки;
- извлечение значения типа `double`.

Заголовочный файл: `atl/double.h`

`create_Double`

Функция создает спецификационную ссылку типа `Double`.

```
Double* create_Double( double d )
```

Параметры

d

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `Double` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
Double* d1 = create(&type_Double, 3.14);
Double* d2 = create_Double(3.14);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`value_Double`

Функция получает значение `double`, содержащееся в спецификационном типе.

```
double value_Double( Double* d )
```

Параметры

d

Спецификационная ссылка за значение типа `Double`.

Дополнительная информация

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

```
Double* d = create_Double(3.14);
double *p = (double*)d;
```

Функция `value_Double()` обеспечивает контроль типов во время исполнения и ей можно передавать указатель `Object*` без предварительного приведения:

```
List* l;
double val;
...
val = value_Double(get_List(l,0)); // Object* get_List(List*,int)
```

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции `type()`:

```
Object* o = get_List(l,0));
if (type(o) == &type_Double) {
  val = value_Double(o);
}
```

14.2.7 `VoidAst`

Тип `VoidAst` является спецификационным аналогом встроенного типа языка C `void`.

Дополнительная информация
Функции, определенные для типа `VoidAst`:

- создание спецификационной ссылки;
- извлечение значения типа `void*`.

Заголовочный файл: `atl/voidast.h`

`create_VoidAst`
Функция создает спецификационную ссылку типа `VoidAst`.

```
VoidAst* create_VoidAst( void* v )
```

Параметры

v

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `VoidAst` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
VoidAst* v1 = create(&type_VoidAst, NULL);
VoidAst* v2 = create_VoidAst(NULL);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`value_VoidAst`
Функция получает значение `void*`, содержащееся в спецификационном типе.

```
void* value_VoidAst( VoidAst* v )
```

Параметры

v

Спецификационная ссылка за значение типа `VoidAst`.

Дополнительная информация

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

```
VoidAst* v = create_VoidAst(NULL);
voidast  val = *v;
```

Функция `value_VoidAst` обеспечивает контроль типов во время исполнения и ей можно передавать указатель `Object*` без предварительного приведения:

```
List* l;
void* val;
...
val = value_VoidAst(get_List(l,0)); // Object* get_List(List*,int)
```

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции `type`:

```
Object* o = get_List(l,0));
if (type(o) == &type_VoidAst) {
  val = value_VoidAst(o);
}
```

14.2.8 `Unit`

Тип `Unit` является типом с единственным значением: две ненулевые спецификационные ссылки типа `Unit` всегда считаются равными друг другу.

Дополнительная информация

Функции, определенные для типа `Unit`:

- создание спецификационной ссылки

Заголовочный файл: atl/unit.h

`create_Unit`

Функция создает спецификационную ссылку типа `Unit`.

```
Unit* create_Unit(void)
```

Возвращаемое значение

Спецификационная ссылка типа `Unit` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
Unit* u1 = create(&type_Unit);
Unit* u2 = create_Unit();
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

14.2.9 `BigInteger`

Тип `BigInteger` служит для представления целых чисел произвольного размера.

Заголовочный файл: `atl/bigint.h`

- **`add_BigInteger`**
  Функция возвращает результат сложения двух чисел типа `BigInteger`.
- **`create_BigInteger`**
  Функции создает спецификационную ссылку типа `BigInteger`, представляющую данное целое число.
- **`divide_BigInteger`**
  Функция возвращает результат деления двух чисел типа `BigInteger`.
- **`intValue_BigInteger`**
  Функция возвращает значение `int`, содержащееся в спецификационном типе (если это значение помещается в `int`).
- **`longValue_BigInteger`**
  Функция возвращает значение `long`, содержащееся в спецификационном типе (если это значение помещается в `long`).
- **`multiply_BigInteger`**
  Функция возвращает результат умножения двух чисел типа `BigInteger`.
- **`negate_BigInteger`**
  Функция возвращает отрицание числа типа `BigInteger`.
- **`power_BigInteger`**
  Функция возвращает результат возведения числа типа `BigInteger` в целую степень.
- **`remainder_BigInteger`**
  Функция возвращает остаток от деления двух чисел типа `BigInteger`.
- **`subtract_BigInteger`**
  Функция возвращает результат вычитания двух чисел типа `BigInteger`.
- **`valueOf_BigInteger`**
  Функции создает спецификационную ссылку типа `BigInteger`, представляющую целое значение в данном строковом виде.

`add_BigInteger`

Функция возвращает результат сложения двух чисел типа `BigInteger`.

```
BigInteger* add_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Параметры

n1

Первое слагаемое.

n2

Второе слагаемое.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на новый объект — сумму.

`create_BigInteger`

Функция создает спецификационную ссылку типа `BigInteger`, представляющую данное целое число.

```
BigInteger* create_BigInteger( int i )
```

Параметры

`i`

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
BigInteger* i1 = create(&type_BigInteger, -28);
BigInteger* i2 = create_BigInteger(-28);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`divide_BigInteger`

Функция возвращает результат деления двух чисел типа `BigInteger`.

```
BigInteger* divide_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Параметры

`n1`

Делимое.

`n2`

Делитель.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на новый объект — частное.

`intValue_BigInteger`

Функция возвращает значение `int`, содержащееся в спецификационном типе (если это значение помещается в `int`).

```
int intValue_BigInteger ( BigInteger* i )
```

Параметры

`i`

Спецификационная ссылка за значение типа `BigInteger`.

Дополнительная информация

Если значение `BigInteger` не помещается в `int`, выполнение программы прекращается с помощью функции `assertion`.

`longValue_BigInteger`

Функция возвращает значение `long`, содержащееся в спецификационном типе (если это значение помещается в `long`).

```
long value_BigInteger ( BigInteger* i )
```

Параметры

`i`

Спецификационная ссылка на значение типа `BigInteger`.

Дополнительная информация

Если значение `BigInteger` не помещается в `int`, выполнение программы прекращается с помощью функции `assertion`.

`multiply_BigInteger`

Функция возвращает результат умножения двух чисел типа `BigInteger`.

```
BigInteger* multiply_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Параметры

`n1`

Первый множитель.

`n2`

Второй множитель.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на новый объект — произведение.

`negate_BigInteger`

Функция возвращает отрицание числа типа `BigInteger`.

```
BigInteger* negate_BigInteger( BigInteger* n )
```

Параметры

`n`

Аргумент отрицания.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на новый объект — отрицание.

`power_BigInteger`

Функция возвращает результат возведения числа типа `BigInteger` в целую степень.

```
BigInteger* divide_BigInteger( BigInteger* n1, int n2 )
```

Параметры

`n1`

Основание степени.

`n2`

Показатель степени.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на новый объект — результат возведения в степень.

`remainder_BigInteger`

Функция возвращает остаток от деления двух чисел типа `BigInteger`.

```
BigInteger* remainder_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Параметры

`n1`

Делимое.

`n2`

Делитель.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на новый объект — остаток.

`subtract_BigInteger`

Функция возвращает результат вычитания двух чисел типа `BigInteger`.

```
BigInteger* subtract_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Параметры

`n1`

Уменьшаемое.

`n2`

Вычитаемое.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на новый объект — разность.

`valueOf_BigInteger`

Функция создает спецификационную ссылку типа `BigInteger`, представляющую целое значение в данном строковом виде.

```
BigInteger* valueOf_BigInteger( String* str )
```

Параметры

`str`

Строковый вид значения числа.

Возвращаемое значение

Спецификационная ссылка типа `BigInteger` на созданный объект.

14.2.10 `Complex`

Тип `Complex` служит для представления комплексных чисел.

Дополнительная информация

Для спецификационных ссылок на тип `Complex` действуют обычные правила сравнения ($(re_1, im_1) = (re_2, im_2) \leftrightarrow re_1 = re_2, im_1 = im_2$). Строковое представление имеет вид ($re + im*i$).

Базовым типом для типа `Complex` служит структура:

```
struct {
  double re;
  double im;
};
```

Для получения действительной и мнимой частей не определено специальных функций; следует пользоваться разыменованием спецификационной ссылки:

```
Complex* c = create_Complex(1.4, -0.6);
double re = c->re;
double im = c->im;
```

Функции, определенные для типа `Complex`:

- создание спецификационной ссылки;

Заголовочный файл: `atl/complex.h`

`create_Complex`

Функция создает спецификационную ссылку типа `Complex`.

```
Complex* create_Complex ( double re, double im )
```

Параметры

`re`

Вещественная часть комплексного числа.

`im`

Мнимая часть комплексного числа.

Возвращаемое значение

Спецификационная ссылка типа `Complex` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
Complex* c1 = create(&type_Complex, 1.4, -0.6);
Complex* c2 = create_Complex(1.4, -0.6);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

14.2.11 `String`

Тип `String` служит для представления строк.

Дополнительная информация

Принятый в языке C способ представления строк в виде массива типа `char` не укладывается в рамки допустимого типа. Поэтому везде, где требуется допустимый тип, удобно представлять строки данным спецификационным типом.

Для спецификационных ссылок на тип `String` действуют обычные правила сравнения строк (как в функции `strcmp`). Позиции символов нумеруются от `0`.

Со спецификационными строками можно работать как с обычными C-строками, принимая во внимание, что значение этой строки не должно изменяться. Для получения доступа к C-строке используется функция `toCharArray_String`, возвращающая указатель на массив внутри спецификационного типа.

В то же время определено большое количество функций и для самих спецификационных строк. Во всех этих функциях спецификационные ссылки на тип `String` не должны быть нулевыми.

Заголовочный файл: `atl/string.h`

- **`create_String`**
  Функция создает спецификационную ссылку типа `String`.
- **`charAt_String`**
  Функция возвращает символ в данной позиции *`index`*.
- **`concat_String`**
  Функция возвращает спецификационную ссылку типа `String` на конкатенацию двух заданных строк.
- **`startsWith_String`**
  Функция проверяет, начинается ли строка *`self`* строкой *`suffix`*.
- **`startsWithOffset_String`**
  Функция проверяет наличие подстроки *`fix`* в строке *`self`* на заданной позиции.
- **`endsWith_String`**
  Функция проверяет, заканчивается ли строка *`self`* строкой *`suffix`*.
- **`indexOfChar_String`**
  Ищет заданный символ в строке и возвращает номер первой найденной позиции.
- **`indexOfCharFrom_String`**
  Ищет заданный символ в строке, начиная с позиции *`fromIndex`*, и возвращает номер первой найденной позиции.
- **`indexOfString_String`**
  Ищет заданную подстроку в строке и возвращает номер первой найденной позиции.
- **`indexOfStringFrom_String`**
  Ищет заданную подстроку в строке, начиная с позиции *`fromIndex`*, и возвращает номер первой найденной позиции.

- **lastIndexOfChar_String**
  Ищет заданный символ в строке справа налево и возвращает номер первой найденной позиции.
- **lastIndexOfCharFrom_String**
  Ищет заданный символ в строке справа налево, начиная с позиции и возвращает номер первой найденной позиции.
- **lastIndexOfString_String**
  Ищет заданную подстроку в строке справа налево и возвращает номер первой найденной позиции.
- **lastIndexOfStringFrom_String**
  Ищет заданную подстроку в строке справа налево, начиная с позиции *fromIndex*, и возвращает номер первой найденной позиции.
- **length_String**
  Функция возвращает длину строки.
- **regionMatches_String, regionMatchesCase_String**
  Проверяет совпадение фрагмента строки *self* с фрагментом строки *other*. Функция `regionMatchesCase_String` учитывает регистр символов, функция `regionMatches_String` имеет дополнительный параметр *ignoreCase*, позволяющий игнорировать регистр букв или учитывать его.
- **replace_String**
  Заменяет в данной строке все символы *oldChar* на символы *newChar*.
- **substringFrom_String**
  Функция возвращает подстроку, включающую символы с позиции *beginIndex* до конца.
- **substring_String**
  Функция возвращает подстроку, включающую символы с позиции *beginIndex* по *endIndex – 1* включительно.
- **toLowerCase_String**
  Функция переводит буквы строки в нижний регистр.
- **toUpperCase_String**
  Функция переводит буквы строки в верхний регистр.
- **toCharArray_String**
  Функция возвращает С-строку, соответствующую данной спецификационной строке.
- **trim_String**
  Функция возвращает строку, полученную из *self* отбрасыванием пробельных символов в начале и конце строки.
- **format_String**
  Функция возвращает спецификационную строку, соответствующую выводу функции `printf` с теми же параметрами.
- **vformat_String**
  Функция возвращает спецификационную строку, соответствующую выводу функции `vprintf` с теми же параметрами.
- **valueOfBool_String**
  Функция возвращает строковое представление значения типа `bool`.

- **valueOfChar_String**
  Функция возвращает строковое представление значения типа `char`.
- **valueOfShort_String**
  Функция возвращает строковое представление значения типа `short`.
- **valueOfUShort_String**
  Функция возвращает строковое представление значения типа `unsigned short`.
- **valueOfInt_String**
  Функция возвращает строковое представление значения типа `int`.
- **valueOfUInt_String**
  Функция возвращает строковое представление значения типа `unsigned int`.
- **valueOfLong_String**
  Функция возвращает строковое представление значения типа `long`.
- **valueOfULong_String**
  Функция возвращает строковое представление значения типа `unsigned long`.
- **valueOfFloat_String**
  Функция возвращает строковое представление значения типа `float`.
- **valueOfDouble_String**
  Функция возвращает строковое представление значения типа `double`.
- **valueOfPtr_String**
  Функция возвращает строковое представление значения типа `void*`.
- **valueOfObject_String**
  Функция возвращает строковое представление значения спецификационного типа.
- **valueOfBytes_String**
  Функция Возвращает строковое шестнадцатеричное представление массива байтов *p* длины *l*.

create_String

Функция создает спецификационную ссылку типа `String`.

```
String* create_String ( const char *cstr )
```

Параметры

cstr

Инициализационное значение.

Возвращаемое значение

Спецификационная ссылка типа `String` на созданный объект.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
String* s1 = create(&type_String, "a string");
String* s2 = create_String("a string");
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`charAt_String`

Функция возвращает символ в данной позиции *`index`*.

```
char charAt_String( String* self, int index )
```

Параметры

`self`

Строка, символ которой требуется определить.

`index`

Номер позиции внутри строки.

Возвращаемое значение

Символ, найденный в строке *`self`* на позиции *`index`*.

Дополнительная информация

Номер позиции должен находиться в пределах строки. Нумерация начинается с `0`; позиция последнего значимого символа вычисляется с помощью функции `length_String` как `length_String(self) - 1`.

```
String* s = create_String("abracadabra");
printf("%c\n", charAt_String(s,5));
```

```
c
```

`concat_String`

Функция возвращает спецификационную ссылку типа `String` на конкатенацию двух заданных строк.

```
String *concat_String( String* str1, String* str2 )
```

Параметры

`str1`

Первая часть конкатенации.

`str2`

Вторая часть конкатенации.

Возвращаемое значение

Спецификационная строка, полученная конкатенацией строк *`str1`* и *`str2`*.

Дополнительная информация

```
String* s1 = create_String("abra");
String* s2 = create_String("cadabra");
String* s  = concat_String(s1,s2);
printf("%s\n", toCharArray_String(s));
```

```
abracadabra
```

`startsWith_String`

Функция проверяет, начинается ли одна строка другой.

```
bool startsWith_String( String *self, String *prefix )
```

Параметры

self

Строка, начало которой требуется проверить.

prefix

Искомое начало строки.

Возвращаемое значение

`true`, если строка *self* начинается подстрокой *prefix*. `false` в любом другом случае.

Дополнительная информация

Если строка *prefix* пуста, возвращает `true`. Если длина строки *prefix* больше, чем длина *self*, возвращает `false`.

```
String* s  = create_String("abracadabra");
String* s1 = create_String("abra");
String* s2 = create_String("cadabra");
printf("1) %d\n2) %d\n ",
  startsWith_String(s,s1),
  startsWith_String(s,s2)
);
```

```
1) 1
2) 0
```

startsWithOffset_String

Функция проверяет наличие подстроки *fix* в строке *self* на заданной позиции.

```
bool startsWithOffset_String
     ( String *self, String *fix, int toffset )
```

Параметры

self

Строка, в которой требуется искать подстроку.

fix

Искомая подстрока.

toffset

Позиция в заданной строке.

Возвращаемое значение

`true`, если в строке *self* на позиции *toffset* начинается подстрока *prefix*. `false` в любом другом случае.

Дополнительная информация

Если позиция отрицательна или превосходит длину строки *self*, возвращает `false`. Если строка *prefix* пуста, возвращает `true`. Если длина строки *prefix* больше, чем длина *self*, возвращает `false`.

```
String* s  = create_String("abracadabra");
```

```
String* s1 = create_String("abra");
printf("1) %d\n2) %d\n",
  startsWithOffset_String(s,s1,7),
  startsWithOffset_String(s,s2,8)
);
```

```
1) 1
2) 0
```

endsWith_String

Функция проверяет, заканчивается ли строка *self* строкой *suffix*.

```
bool endsWith_String( String *self, String *suffix )
```

Параметры

self

Строка, в окончание которой требуется проверить.

suffix

Искомое окончание строки.

Возвращаемое значение

`true`, если строка *self* заканчивается подстрокой *suffix*. `false` в любом другом случае.

Дополнительная информация

Если строка *suffix* пуста, функция возвращает `true`. Если длина строки *suffix* больше, чем длина *self*, функция возвращает `false`.

```
String* s  = create_String("abracadabra");
String* s1 = create_String("abr");
String* s2 = create_String("cadabra");
printf( "1) %d\n2) %d\n "
      , endsWith_String(s,s1)
      , endsWith_String(s,s2)
      );
```

```
1) 0
2) 1
```

indexOfChar_String

Ищет заданный символ в строке и возвращает номер первой найденной позиции.

```
int indexOfChar_String( String* self, int ch )
```

Параметры

self

Строка, в которой требуется искать символ.

ch

Искомый символ.

Возвращаемое значение

Номер первого символа *ch* в строке *self*. Если символ не найден, возвращается -1. Для символа с кодом 0 функция всегда возвращает –1.

Дополнительная информация

```
String* s = create_String("abracadabra");
printf( "1) %d\n2) %d\n"
      , indexOfChar_String(s,'b')
      , indexOfChar_String(s,'z')
      );
```

```
1) 1
2) -1
```

indexOfCharFrom_String

Ищет заданный символ в строке, начиная с позиции *fromIndex*, и возвращает номер первой найденной позиции.

```
int indexOfCharFrom_String( String* self, int ch, int fromIndex )
```

Параметры

self

Строка, в которой требуется искать символ.

ch

Искомый символ.

fromIndex

Позиция начала поиска.

Возвращаемое значение

Номер первого символа *ch* в строке *self*, начиная с позиции *fromIndex*. Если позиция *fromIndex* превосходит длину строки *self*, т. е. *fromIndex* > length_String(*self*), возвращает –1. Если символ не найден, возвращается -1. Для символа с кодом 0 функция всегда возвращает –1.

Дополнительная информация

Если *fromIndex* < 0, то за позицию начала поиска принимается 0.

```
String* s = create_String("abracadabra");
printf( "1) %d\n2) %d\n"
      , indexOfCharFrom_String(s,'b',5)
      , indexOfCharFrom_String(s,'b',9)
      );
```

```
1) 8
2) -1
```

indexOfString_String

Ищет заданную подстроку в строке и возвращает номер первой найденной позиции.

```
int indexOfString_String( String* self, String* str )
```

Параметры

self

Строка, в которой требуется искать подстроку.

str

Искомая подстрока.

Возвращаемое значение

Если подстрока *str* найдена в строке *self*, то возвращается позиция в строке *self*, с которой начинается *str*. Если подстрока не найдена, возвращается -1.

Дополнительная информация

Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции `0`.

```
String* s  = create_String("abracadabra");
String* s1 = create_String("abra");
String* s2 = create_String("cdbr");
printf( "1) %d\n2) %d\n"
      , indexOfString_String(s,s1)
      , indexOfString_String(s,s2)
      );
```

```
1) 0
2) -1
```

`indexOfStringFrom_String`

Ищет заданную подстроку в строке, начиная с позиции *fromIndex*, и возвращает номер первой найденной позиции.

```
int indexOfStringFrom_String
    ( String* self, String* str, int fromIndex )
```

Параметры

self

Заданная строка.

str

Искомая подстрока.

fromIndex

Позиция начала поиска.

Возвращаемое значение

Если подстрока *str* найдена в строке *self*, начиная с позиции *fromIndex*, то возвращается позиция в строке *self*, с которой начинается *str*. Если позиция *fromIndex* превосходит длину строки *self*, т. е. *fromIndex* `>` `length_String(`*self*`)`, функция возвращает `–1`. Если подстрока не найдена, возвращается `-1`.

Дополнительная информация

Если *fromIndex* < 0, то *fromIndex* полагается равным 0. Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции *fromIndex*.

```
String* s  = create_String("abracadabra");
String* s1 = create_String("abra");
printf( "1) %d\n2) %d\n"
      , indexOfString_String(s,s1,5)
      , indexOfString_String(s,s1,8)
      );
```

```
1) 7
2) -1
```

lastIndexOfChar_String

Ищет заданный символ в строке справа налево и возвращает номер первой найденной позиции.

```
int lastIndexOfChar_String( String* self, int ch )
```

Параметры

self

Заданная строка.

ch

Искомый символ.

Возвращаемое значение

Позиция последнего символа *ch* в строке *self*. Если символ не найден, возвращается –1. Для символа с кодом 0 всегда возвращается –1.

Дополнительная информация

```
String* s = create_String("abracadabra");
printf( "1) %d\n2) %d\n"
      , lastIndexOfChar_String(s,'b')
      , lastIndexOfChar_String(s,'z')
      );
```

```
1) 8
2) -1
```

lastIndexOfCharFrom_String

Ищет символ в строке справа налево, начиная с заданной позиции, и возвращает номер первой найденной позиции.

```
int lastIndexOfCharFrom_String( String* self, int ch, int fromIndex )
```

Параметры

self

Заданная строка.

ch

Искомый символ.

fromIndex

Позиция начала поиска.

Возвращаемое значение

Позиция символа *ch* в строке *self*. Если *fromIndex* < 0, возвращает –1. Если символ не найден, функция возвращает –1. Для символа с кодом 0 функция всегда возвращает –1.

Дополнительная информация

Если *fromIndex* превосходит длину строки *self*, т. е. *fromIndex* > length_String(*self*), то за позицию начала поиска принимается позиция последнего символа строки.

```
String* s = create_String("abracadabra");
printf( "1) %d\n2) %d\n"
      , lastIndexOfCharFrom_String(s,'b',5)
      , lastIndexOfCharFrom_String(s,'b',0)
      );
```

```
1) 1
2) -1
```

lastIndexOfString_String

Ищет заданную подстроку в строке справа налево и возвращает номер первой найденной позиции.

```
int lastIndexOfString_String( String* self, String* str )
```

Параметры

self

Заданная строка.

str

Искомая подстрока.

Возвращаемое значение

Если подстрока не найдена, возвращает –1.

Дополнительная информация

Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции length_String(*self*).

```
String* s  = create_String("abracadabra");
String* s1 = create_String("abra");
String* s2 = create_String("cdbr");
printf( "1) %d\n2) %d\n"
      , indexOfString_String(s,s1)
      , indexOfString_String(s,s2)
      );
```

```
1) 7
2) -1
```

`lastIndexOfStringFrom_String`

Ищет заданную подстроку в строке справа налево, начиная с позиции *`fromIndex`*, и возвращает номер первой найденной позиции.

```
int lastIndexOfStringFrom_String( String* self,
                                  String* str,
                                  int fromIndex
                                )
```

Параметры

`self`

Заданная строка.

`str`

Искомая подстрока.

`fromIndex`

Позиция начала поиска.

Возвращаемое значение

Позиция подстроки *`str`* в строке *`self`*. Если *`fromIndex`* `< 0`, возвращает `-1`.

Дополнительная информация

Если *`fromIndex`* превосходит длину строки *`self`*, т. е. *`fromIndex`* `>` `length_String(`*`self`*`)`, то за позицию начала поиска принимается `length_String(self)`. Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции *`fromIndex`*.

```
String* s  = create_String("abracadabra");
String* s1 = create_String("abra");
printf( "1) %d\n2) %d\n"
      , lastIndexOfString_String(s,s1,3)
      , lastIndexOfString_String(s,s1,2)
      );
```

```
1) 0
2) -1
```

`length_String`

Функция возвращает длину строки.

```
int length_String( String* self )
```

Параметры

`self`

Строка, длину которой требуется найти.

Возвращаемое значение

Точное количество символов в строке *`self`*.

Дополнительная информация

Эта функция определена наряду со стандартной функций `create`.

```
String* s = create_String("abracadabra");
printf( "%d\n", length_String(s) );
```

```
11
```

regionMatches_String, regionMatchesCase_String

Проверяет совпадение фрагмента строки *self* с фрагментом строки *other*. Функция regionMatchesCase_String учитывает регистр символов, функция regionMatches_String имеет дополнительный параметр *ignoreCase*, позволяющий игнорировать регистр букв или учитывать его.

```
bool regionMatches_String
     ( String* self, bool ignoreCase, int toffset, String* other, int
ooffset, int len)
```

```
bool regionMatchesCase_String
    ( String* self, int toffset, String* other, int ooffset, int len)
```

Параметры

self

Первая строка.

toffset

Позиция фрагмента в первой строке.

other

Вторая строка.

ooffset

Позиция фрагмента во второй строке.

len

Длина сравниваемых фрагментов.

ignoreCase

Если параметр равен true, то регистр игнорируется, если false, то учитывается.

Возвращаемое значение

true, если указанные сегменты строк совпадают и false в любом другом случае.

Дополнительная информация

Длина фрагментов должна быть неотрицательной (*len* ≥ 0). Если фрагменты, определяемые позицией и длиной, выходят за границы строки, возвращает false.

```
SString* s1 = create_String("aBrAcAdabra");
String* s2 = create_String("cadabra");
printf( "a1) %d\na2) %d\n"
      , regionMatchesCase_String(s1,0,s2,3,4)
      , regionMatchesCase_String(s1,7,s2,3,4)
      );
printf( "b1) %d\nb2) %d\n"
      , regionMatches_String(s1,false,0,s2,3,4)
      , regionMatches_String(s1,true,0,s2,3,4)
      );
```

```
a1) 0
a2) 1
b1) 0
b2) 1
```

replace_String

Заменяет в данной строке все символы *oldChar* на символы *newChar*.

```
String* replace_String( String* self, char oldChar, char newChar )
```

Параметры

self

Заданная строка.

oldChar

Символ, подлежащий замене.

newChar

Символ, которым заменяются вхождения *oldChar*.

Возвращаемое значение

Спецификационная строка, полученная из *self* заменой символов *oldChar* на *newChar*.

Дополнительная информация

Символы не должны иметь нулевой код.

```
String* s = create_String("abracadabra");
String* res = replace_String(s,'a','_');
printf( "%s\n", toCharArray(res) );
```

```
_br_c_d_br_
```

substringFrom_String

Функция возвращает подстроку, включающую символы с позиции *beginIndex* до конца.

```
String* substringFrom_String( String* self, int beginIndex )
```

Параметры

self

Заданная строка.

beginIndex

Позиция начала искомой подстроки.

Возвращаемое значение

Спецификационная строка, являющаяся подстрокой *self*, извлеченной по указанным правилам.

Дополнительная информация

Позиция *beginIndex* не должна выходить на пределы строки:
`0 ≤ beginIndex ≤ length_String(self)`.

```
String* s  = create_String("abracadabra");
String* res = substringFrom_String(s,4);
printf( "%s\n", toCharArray_String(res) );
```

```
cadabra
```

substring_String

Возвращает подстроку, включающую символы с позиции *beginIndex* по *endIndex* – 1 включительно.

```
String* substring_String
        ( String* self, int beginIndex, int endIndex )
```

Параметры

self

Заданная строка.

beginIndex

Позиция начала искомой подстроки.

endIndex

Позиция конца искомой подстроки, увеличенная на единицу.

Возвращаемое значение

Спецификационная строка, являющаяся подстрокой *self*, извлеченной по указанным правилам.

Дополнительная информация

Позиция *beginIndex* не должна быть отрицательной и не должна превосходить *endIndex*, а позиция *endIndex* не должна превосходить длину строки:
$0 \le beginIndex \le endIndex \le$ length_String(*self*).

Если начальная и конечная позиции совпадают, возвращает пустую строку.

```
String* s  = create_String("abracadabra");
String* res = substring_String(s,4,7);
printf( "%s\n", toCharArray_String(res) );
```

```
cad
```

toLowerCase_String

Функция переводит буквы строки в нижний регистр.

```
String* toLowerCase_String( String* self )
```

Параметры

self

Строка, которую требуется перевести в нижний регистр.

Возвращаемое значение

Спецификационная строка, полученная из *self* переведением в нижний регистр.

Дополнительная информация

```
String* s   = create_String("aBrAcAdAbRa");
String* res = toLowerCase_String(s);
printf( "%s\n", toCharArray_String(res) );
```

```
abracadabra
```

`toUpperCase_String`

Функция переводит буквы строки в верхний регистр.

```
String* toUpperCase_String( String* self )
```

Параметры

`self`

Строка, которую требуется перевести в верхний регистр.

Возвращаемое значение

Спецификационная строка, полученная из *`self`* переведением в верхний регистр.

Дополнительная информация

```
String* s   = create_String("aBrAcAdAbRa");
String* res = toLowerCase_String(s);
printf( "%s\n", toCharArray_String(res) );
```

```
ABRACADABRA
```

`toCharArray_String`

Функция возвращает C-строку, соответствующую данной спецификационной строке.

```
const char* toCharArray_String( String* self )
```

Параметры

`self`

Данная спецификационная ссылка типа `String`.

Возвращаемое значение

Массив символов, соответствующий спецификационной строке *`self`*.

Дополнительная информация

```
String* s = create_String("abracadabra");
printf( "%s\n", toCharArray_String(s) );
```

```
abracadabra
```

`trim_String`

Функция возвращает строку, полученную из *`self`* отбрасыванием пробельных символов (whitespace) в начале и конце строки.

```
String* trim_String( String* self )
```

Параметры

self

Строка, из которой нужно исключить начальные и конечные пробельные символы.

Возвращаемое значение

Спецификационная ссылка типа String на строку, полученную из *self* отбрасыванием начальных и конечных пробельных символов.

Дополнительная информация

Пробельными символами считаются пробелы, табуляции и переводы строк.

```
String* s   = create_String(" \tabracadabra  \n");
String* res = trim_String(s);
printf( "'%s'\n", toCharArray_String(res) );
```

```
'abracadabra'
```

format_String

Функция возвращает спецификационную строку, соответствующую выводу функции printf с теми же параметрами.

```
String* format_String(const char *format, ...)
```

Параметры

format

Строка, задающая форматирование по правилам функции printf.

Возвращаемое значение

Спецификационная ссылка типа String, соответствующая выводу функции printf с параметрами *format*.

Дополнительная информация

```
char s[12];
String* str;
sprintf(s,"abra%s","cadabra");
str = create_String(s);
String* str = format_String("abra%s","cadabra");
```

Два показанных способа создания строки эквивалентны.

vformat_String

Функция делает тоже самое, что и format_String, но в отличие от неё принимает в качестве параметра не произвольное число аргументов, а значение типа va_list.

```
String* vformat_String(const char *format, va_list args )
```

Параметры

format

Строка, задающая форматирование по правилам функции printf.

args

Аргументы, передающиеся одной из функций семейства `printf`

Возвращаемое значение

Спецификационная ссылка типа `String`, соответствующая выводу функции `vprintf` с параметрами *`format`*.

Дополнительная информация

```
char s[12];
String* str;
vsprintf(s,"abra%s","cadabra");
str = create_String(s);
String* str = vformat_String("abra%s","cadabra");
```

Два показанных способа создания строки эквивалентны.

`valueOfBool_String`

Функция возвращает строковое представление значения типа `bool`.

```
String* valueOfBool_String( bool b )
```

Параметры

b

Значение типа `bool`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *b*.

Дополнительная информация

```
String* s1 = valueOfBool_String(true);
String* s2 = valueOfBool_String(false);
printf( "1) %s\n2) %s\n"
      , toCharArray_String(s1)
      , toCharArray_String(s2)
      );
```

```
1) true
2) false
```

`valueOfChar_String`

Функция возвращает строковое представление значения типа `char`.

```
String* valueOfChar_String( char c )
```

Параметры

c

Значение типа `char`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *c*.

Дополнительная информация

```
String* s = valueOfChar_String('a');
printf( "%s\n", toCharArray_String(s) );
```

```
a
```

valueOfShort_String

Функция возвращает строковое представление значения типа `short`.

```
String* valueOfShort_String( short i )
```

Параметры

i

Значение типа `short`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *i*.

Дополнительная информация

```
String* s = valueOfShort_String (-28);
printf( "%s\n", toCharArray_String(s) );
```

```
-28
```

valueOfUShort_String

Функция возвращает строковое представление значения типа `unsigned short`.

```
String* valueOfUShort_String( unsigned short i )
```

Параметры

i

Значение типа `unsigned short`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *i*.

Дополнительная информация

```
String* s = valueOfUShort_String(47);
printf( "%s\n", toCharArray_String(s) );
```

```
47
```

valueOfInt_String

Функция возвращает строковое представление значения типа `int`.

```
String* valueOfInt_String( int i )
```

Параметры

i

Значение типа `int`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *`i`*.

Дополнительная информация

```
String* s = valueOfInt_String(-28);
printf( "%s\n", toCharArray_String(s) );
```

```
-28
```

`valueOfUInt_String`

Функция возвращает строковое представление значения типа `unsigned int`.

```
String* valueOfUInt_String( unsigned int i )
```

Параметры

`i`

Значение типа `unsigned int`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *`i`*.

Дополнительная информация

```
String* s = valueOfUInt_String(47);
printf( "%s\n", toCharArray_String(s) );
```

```
47
```

`valueOfLong_String`

Функция возвращает строковое представление значения типа `long`.

```
String* valueOfLong_String( long i )
```

Параметры

`i`

Значение типа `long`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *`i`*.

Дополнительная информация

```
String* s = valueOfLong_String(-28);
printf( "%s\n", toCharArray_String(s) );
```

```
-28
```

`valueOfULong_String`

Функция возвращает строковое представление значения типа `unsigned long`.

```
String* valueOfULong_String( unsigned long i )
```

Параметры

i

Значение типа `unsigned long`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *i*.

Дополнительная информация

```
String* s = valueOfULong_String(47);
printf( "%s\n", toCharArray_String(s) );
```

```
47
```

`valueOfFloat_String`

Функция возвращает строковое представление значения типа `float`.

```
String* valueOfFloat_String( float f )
```

Параметры

f

Значение типа `float`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *f*.

Дополнительная информация

```
String* s = valueOfFloat_String(3.14);
printf( "%s\n", toCharArray_String(s) );
```

```
3.140000
```

`valueOfDouble_String`

Функция возвращает строковое представление значения типа `double`.

```
String* valueOfDouble_String( double d )
```

Параметры

d

Значение типа `double`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения *d*.

Дополнительная информация

```
String* s = valueOfDouble_String(3.14);
printf( "%s\n", toCharArray_String(s) );
```

```
3.140000
```

`valueOfPtr_String`

Функция возвращает строковое представление значения типа `void*`.

```
String* valueOfPtr_String( void *p )
```

Параметры

p

Значение типа `void*`, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление нетипизированного указателя *p*.

Дополнительная информация

```
int i;
String* s = valueOfPtr_String((void*)&i);
printf( "%s\n", toCharArray_String(s) );
```

```
0012FF6C
```

`valueOfObject_String`

Функция возвращает строковое представление значения спецификационного типа.

```
String* valueOfObject_String( Object* ref )
```

Параметры

ref

Значение спецификационного типа, для которого строится строковое представление.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление значения спецификационного типа *p*.

Дополнительная информация

Результат совпадает с возвращаемым значением стандартной функции `toString`:

```
Object* ref;
String* s = valueOfObject_String(ref);
String* s = toString(ref);
```

`valueOfBytes_String`

Функция Возвращает строковое шестнадцатеричное представление массива байтов определенной длины.

```
String* valueOfBytes_String( const char* p, int l )
```

Параметры

p

Массив байтов, для которого строится строковое представление.

1

Длина массива байтов *p*.

Возвращаемое значение

Спецификационная ссылка типа `String` на строковое представление массива байтов *p*.

Дополнительная информация

```
char a[6] = { 0x00, 0x33, 0x66, 0x99, 0xCC, 0xFF };
String* s = valueOfBytes_String(a,6);
printf( "%s\n", toCharArray_String(s) );
```

```
[00 33 66 99 CC FF]
```

14.2.12 `List`

Тип `List` является контейнерным типом, реализующим упорядоченный список элементов.

Элементами списка могут быть любые спецификационные ссылки. При создании списка можно ограничить тип его элементов. Такой список называется типизированным, а все функции, работающие с ним и имеющие параметр типа `Object*`, будут проверять, чтобы тип этого параметра совпадал с типом элементов списка.

Два списка считаются равными, если они имеют одинаковую длину и их элементы попарно равны друг другу. При этом не учитывается типизация списков, в частности, пустые списки всегда равны.

Элементы списка нумеруются с нуля.

Заголовочный файл: `atl/list.h`

- **`add_List`**
  Функция вставляет элемент на заданную позицию в списке.
- **`addAll_List`**
  Добавляет в один список все элементы другого списка, вставляя их в том же порядке с заданной позиции.
- **`append_List`**
  Функция добавляет переданный элемент в конец списка.
- **`appendAll_List`**
  Добавляет конец одного списка все элементы другого списка.
- **`clear_List`**
  Функция удаляет из списка все элементы.
- **`contains_List`**
  Функция проверяет, содержится ли в списке элемент, равный данному.
- **`create_List`**
  Функция создает список и возвращает спецификационную ссылку типа List.
- **`elemType_List`**
  Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного списка.
- **`get_List`**
  Функция возвращает спецификационную ссылку на элемент в заданной позиции.
- **`indexOf_List`**
  Функция возвращает номер позиции первого элемента в списке, имеющего заданное значение.
- **`isEmpty_List`**
  Функция проверяет, пуст ли данный список.
- **`lastIndexOf_List`**
  Функция возвращает номер позиции последнего элемента в списке, имеющего заданное значение.
- **`remove_List`**
  Функция удаляет элемент с заданной позиции в списке.

- **set_List**
  Функция заменяет переданным объектом элемент на заданной позиции.
- **size_List**
  Функция возвращает длину списка.
- **subList_List**
  Возвращает в виде нового списка элементы данного списка, заключенные между указанными позициями.
- **toSet_List**
  Функция возвращает множество (Set), состоящее из уникальных элементов данного списка.

add_List

Функция вставляет элемент на заданную позицию в списке.

```
void add_List( List* self, int index, Object* ref )
```

Параметры

self

Список, в который осуществляется вставка.

index

Позиция в списке, на которую помещается значение *ref*.

ref

Значение спецификационного типа, вставляемое в список.

Дополнительная информация

Если список типизирован, то тип элемента *ref* должен совпадать с типом элементов списка. Номер позиции должен находиться в интервале от 0 до длины списка включительно, т. е. $0 \le index \le$ size_List(*self*). Если номер позиции равен длине списка, то элемент вставляется в конец этого списка.

```
List* l = create_List(&type_Integer);
add_List(l,0,create_Integer(28));
add_List(l,0,create_Integer(47));
add_List(l,1,create_Integer(63));
```

Список *l* меняется по мере выполнения этого кода так:

1. < 28 >
2. < 47, 28 >
3. < 47, 63, 28 >

addAll_List

Добавляет в один список все элементы другого списка, вставляя их в том же порядке с заданной позиции.

```
void addAll_List(List* self, int index, List* other)
```

Параметры

self

Список, в который вставляются новые элементы.

index

Позиция, с которой начинается вставка новых элементов.

other

Список, чьи элементы вставляются в список *self*.

Дополнительная информация

Если список *self* типизирован, то типы всех элементов списка *other* должны совпадать с типом элементов списка *self* (при этом сам список *other* не обязан быть типизированным). Номер позиции должен находиться в интервале от 0 до длины списка *self* включительно, т. е. 0 ≤ *index* ≤ size_List(*self*). Если номер позиции равен длине списка *self*, то элементы вставляется в конец этого списка.

```
List* l1 = create_List(&type_Integer);
List* l2 = create_List(NULL);
append_List(l1,create_Integer(28));
append_List(l1,create_Integer(47));
append_List(l2,create_Integer(63));
append_List(l2,create_Integer(85));
addAll_List(l1,1,l2);
```

Список *l1* до вызова функции addAll_List содержит такие элементы: < 28, 47 >. Список *l2* — такие: < 63, 85 >. После вызова addAll_List список *l1* содержит также и элементы *l2*: < 28, 63, 85, 47 >.

append_List

Функция добавляет переданный элемент в конец списка.

```
void append_List( List* self, Object* ref )
```

Параметры

self

Список, в который осуществляется вставка.

ref

Значение спецификационного типа, вставляемое в список.

Возвращаемое значение

Если список не типизирован (при создании не было наложено ограничений на типы элементов), функция возвращает NULL.

Дополнительная информация

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
append_List(l,create_Integer(63));
```

Список *l* меняется по мере выполнения этого кода так:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 47, 63 >`

appendAll_List

Добавляет конец одного списка все элементы другого списка.

```
void appendAll_List(List* self, List* other)
```

Параметры

self

Список, в который вставляются новые элементы.

other

Список, чьи элементы вставляются в список *self*.

Дополнительная информация

Если список *self* типизирован, то типы всех элементов списка *other* должны совпадать с типом элементов списка *self* (при этом сам список *other* не обязан быть типизированным).

```
List* l1 = create_List(&type_Integer);
List* l2 = create_List(NULL);
append_List(l1,create_Integer(28));
append_List(l1,create_Integer(47));
append_List(l2,create_Integer(63));
append_List(l2,create_Integer(85));
appendAll_List(l1,l2);
```

Список *l1* до вызова функции *addAll_List* содержит такие элементы: `< 28, 47 >`. Список *l2* — такие: `< 63, 85 >`. После вызова `appendAll_List` список *l1* содержит также и элементы *l2*: `< 28, 47, 63, 85 >`.

clear_List

Функция удаляет из списка все элементы.

```
void clear_List( List* self )
```

Параметры

self

Список, который требуется очистить.

Дополнительная информация

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
clear_List(l);
```

Список *l* меняется по мере выполнения этого кода так:

1. `< 28 >`
2. `< 28, 47 >`
3. `<>`

Тот же результат можно получить, пересоздав список:

```
List* l = create_List(&type_Integer);
...
l = create_List(elemType_List(l));
```

Но такой способ менее эффективен, чем вызов функции `clear_List`.

`contains_List`

Функция проверяет, содержится ли в списке элемент, равный данному.

```
bool contains_List( List* self, Object* ref )
```

Параметры

`self`

Проверяемый список.

`ref`

Значение спецификационного типа, поиск которого происходит в списке.

Возвращаемое значение

`true`, если *`ref`* содержится в *`self`*. `false` в любом другом случае.

Дополнительная информация

Если список типизирован, то тип элемента *`ref`* должен совпадать с типом элементов списка.

```
bool contains;
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
contains = contains_List(l,create_Integer(28));
```

Список *l* меняется по мере выполнения этого кода так:

1. `< 28 >`
2. `< 47, 28 >`

Значение переменной contains после выполнения кода будет равно `true`.

`create_List`

Функция создает список и возвращает спецификационную ссылку типа `List`.

```
List* create_List( const Type *elem_type )
```

Параметры

`elem_type`

Указатель на константу-дескриптор типа элементов списка.

Возвращаемое значение

Спецификационная сслыка типа `List` на созданный список.

Дополнительная информация

Если параметр elem_type нулевой, то типы элементов списка не ограничены.

```
List* l1 = create_List(NULL);
List* l2 = create_List(&type_Integer);
```

Первым вызовом `create_List` создан список, в который можно включать любые элементы. Список, созданный вторым вызовом, может содержать только элементы типа `Integer`.

Функция `create_List` определена наряду со стандартной функций `create`.

```
List* l1 = create(&type_List, NULL);
List* l2 = create_List(NULL);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`elemType_List`

Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного списка.

```
Type *elemType_List(List* self)
```

Параметры

`self`

Список, ограничения которого должна вернуть функция.

Возвращаемое значение

Указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного списка.

Если список не типизирован (при создании не было наложено ограничений на типы элементов), функция возвращает `NULL`.

Дополнительная информация

```
List* l = create_List(&type_Integer);
Type *t = elemType_List(l);
```

Переменная *`t`* приобретет значение `&type_Integer`.

`get_List`

Функция возвращает спецификационную ссылку на элемент в заданной позиции.

```
Object* get_List( List* self, int index )
```

Параметры

`self`

Список, содержащий необходимый элемент.

index

Позиция элемента, который должна вернуть функция.

Возвращаемое значение

Ссылка на элемент с индексом `index` или `NULL`, если `index` меньше 0 или больше, либо равен длине списка..

Дополнительная информация

Номер позиции должен находиться в интервале от `0` до длины списка, уменьшенной на `1`, т. е. `0 ≤ index < size_List(self)`.

```
List* l = create_List(&type_Integer);
Object* o;
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
append_List(l,create_Integer(63));
o = get_List(l,1);
```

Список *l* меняется по мере выполнения этого кода так:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 47, 63 >`

Значение объекта *o* после вызова `get_List` будет равно `47`.

`indexOf_List`

Функция возвращает номер позиции первого элемента в списке, имеющего заданное значение.

```
int indexOf_List( List* self, Object* ref )
```

Параметры

self

Список, содержащий необходимый элемент.

ref

Спецификационная ссылка, со значением которой сравниваются элементы списка.

Возвращаемое значение

Позиция, на которой в списке находится значение *ref*. Если в списке такого значения нет, возвращаемое значение равно `-1`.

Дополнительная информация

Если список типизирован, то тип элемента *ref* должен совпадать с типом элементов списка. Если список не содержит указанного элемента, функция возвращает `–1`.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
```

```
append_List(l,create_Integer(28));
int pos = indexOf_List(l,create_Integer(28));
```

Список *l* меняется по мере выполнения этого кода так:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 47, 28 >`

Значение переменной *pos* после вызова `get_List` будет равно `0`.

`isEmpty_List`

Функция проверяет, пуст ли данный список.

```
bool isEmpty_List( List* self )
```

Параметры

self

Список, содержащий необходимый элемент.

Возвращаемое значение

`true`, если в списке нет ни одного элемента. `false` в любом другом случае.

Дополнительная информация

Если список типизирован, то тип элемента *ref* должен совпадать с типом элементов списка. Если список не содержит указанного элемента, функция возвращает `-1`.

```
bool empty;
List* l = create_List(&type_Integer);
empty = isEmpty_List(l);
append_List(l,create_Integer(28));
empty = isEmpty_List(l);
```

Значение переменной *empty* после первого вызова `isEmpty_List` будет равно `true`, а после второго (когда в список будет добавлено число `28`) — `false`.

`lastIndexOf_List`

Функция возвращает номер позиции последнего элемента в списке, имеющего заданное значение.

```
int lastIndexOf_List( List* self, Object* ref )
```

Параметры

self

Список, содержащий необходимый элемент.

ref

Спецификационная ссылка, со значением которой сравниваются элементы списка.

Возвращаемое значение

Позиция, на которой в списке находится значение *ref*. Если в списке такого значения нет, возвращаемое значение равно `-1`.

Дополнительная информация

Если список типизирован, то тип элемента *ref* должен совпадать с типом элементов списка. Если список не содержит указанного элемента, функция возвращает –1.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
append_List(l,create_Integer(28));
int pos = lastIndexOf_List(l,create_Integer(28));
```

Список *l* меняется по мере выполнения этого кода так:

1. < 28 >
2. < 28, 47 >
3. < 28, 47, 28 >

Значение переменной *pos* после вызова get_List будет равно 2.

remove_List

Функция удаляет элемент с заданной позиции в списке.

```
void remove_List( List* self, int index )
```

Параметры

self

Список, в который осуществляется вставка.

index

Позиция в списке, с которой удаляется элемент.

Дополнительная информация

Номер позиции должен находиться в интервале от 0 до длины списка включительно, т. е. 0 ≤ *index* ≤ size_List(*self*).

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
append_List(l,create_Integer(63));
remove_List(l,1);
```

Список *l* меняется по мере выполнения этого кода так:

1. < 28 >
2. < 28, 47 >
3. < 28, 47, 63 >
4. < 28, 63 >

set_List

Функция заменяет переданным объектом элемент на заданной позиции.

```
void set_List( List* self, int index, Object* ref )
```

Параметры

self

Список, в котором осуществляется замена.

index

Позиция в списке, в которую помещается значение *ref*.

ref

Значение спецификационного типа, вставляемое в список.

Дополнительная информация

Если список типизирован, то тип элемента *ref* должен совпадать с типом элементов списка. Номер позиции должен находиться в интервале от `0` до длины списка включительно, т. е. `0 ≤ index ≤ size_List(self)`. Если номер позиции равен длине списка, то элемент вставляется в конец этого списка.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
set_List(l,1,create_Integer(63));
```

Список *l* меняется по мере выполнения этого кода так:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 63 >`

size_List

Функция возвращает длину списка.

```
int size_List( List* self )
```

Параметры

self

Список, длину которого должна вернуть функция.

Возвращаемое значение

Точное количество элементов в списке *self*.

Дополнительная информация

```
int size;
List* l = create_List(&type_Integer);
size = size_List(l);
append_List(l,create_Integer(28));
size = size_List(l);
```

После первого вызова `size_List` переменная *size* становится равной нулю. После второго вызова `size_List` переменная равна `1`: в список было добавлено одно значение.

`subList_List`

Возвращает в виде нового списка элементы данного списка, заключенные между указанными позициями.

```
List* subList_List( List* self, int fromIndex, int toIndex )
```

Параметры

`self`

Исходный список.

`fromIndex`

Позиция первого элемента нового списка.

`toIndex`

Число, на единицу большее позиции последнего элемента нового списка.

Возвращаемое значение

Спецификационная ссылка типа `List`, подсписок списка *`self`*.

Дополнительная информация

Позиция первого элемента нового списка — *`fromIndex`*; позиция последнего элемента — *`toIndex`*`-1`. Позиция *`fromIndex`* не должна быть отрицательной и не должна превосходить *`toIndex`*, а позиция *`toIndex`* не должна превосходить длину списка: `0 ≤` *`fromIndex`* `≤` *`toIndex`* `≤ size_List(`*`self`*`)`. Если начальная и конечная позиции совпадают, возвращает пустой список.

```
List* l = create_List(&type_Integer);
List* l2;
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
append_List(l,create_Integer(63));
l2 = subList_List(l,1,3);
```

Список *`l`* меняется в результате содержит такие элементы: `< 28, 47, 63 >`. Список *`l2`* после вызова функции `subList_List` содержит два элемента: `< 47, 63 >`.

`toSet_List`

Функция возвращает множество (`Set`), состоящее из уникальных элементов данного списка.

```
Set* toSet_List(List* self)
```

Параметры

`self`

Список, множество элементов которого требуется получить.

Возвращаемое значение

Спецификационная ссылка типа `Set`, множество элементов списка *`self`*.

Дополнительная информация

Возвращаемое множество сохраняет типизацию списка: если элементы списка были ограничены каким-либо типом, то и элементы множества будут ограничены тем же типом.

```
List* l = create_List(&type_Integer);
Set* s;
Type *t;
append_List(l,create_Integer(28));
append_List(l,create_Integer(47));
append_List(l,create_Integer(28));
s = toSet_List(l);
t = elemType_Set(s);
```

Список *l* меняется по мере выполнения этого кода так:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 47, 28 >`

Множество *s*, возвращенное функцией `toSet_List`, содержит элементы: `< 28, 47 >`. Вызов `elemType_Set(s)` возвращает `&type_Integer`.

14.2.13 `Set`

Тип `Set` является контейнерным типом, реализующим множество элементов.

Элементами множества могут быть любые спецификационные ссылки. При создании множества можно ограничить тип его элементов. Такое множество называется типизированным, а все функции, работающие с ним и имеющие параметр типа `Object*`, будут проверять, чтобы тип этого параметра совпадал с типом элементов множества.

Два множества считаются равными, если они имеют одинаковые элементы. При этом не учитывается типизация множеств, в частности, пустые множества всегда равны.

Заголовочный файл: `atl/set.h`

- **`add_Set`**
  Функция вставляет заданный элемент в множество.
- **`addAll_Set`**
  Объединяет два множества: добавляет элементы одного множества в другое.
- **`clear_Set`**
  Функция удаляет из множества все элементы.
- **`contains_Set`**
  Функция проверяет, содержится ли в множестве элемент, равный данному.
- **`containsAll_Set`**
  Определяет, является ли одно множество подмножеством другого, то есть, содержит ли первое множество все элементы второго.
- **`create_Set`**
  Функция создает множество и возвращает спецификационную ссылку типа `Set`.
- **`elemType_Set`**
  Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного множества.
- **`get_Set`**
  Функция возвращает спецификационную ссылку на элемент с заданным номером; элементы множества при этом нумеруются произвольно.
- **`isEmpty_Set`**
  Функция проверяет, пусто ли данное множество.
- **`remove_Set`**
  Функция удаляет элемент из множества.
- **`removeAll_Set`**
  Вычитает одно множество из другого: удаляет из первого множества те элементы, которые есть во втором.
- **`retainAll_Set`**
  Вычисляет пересечение множеств: оставляет в первом множестве только те элементы, которые есть во втором.
- **`size_Set`**
  Функция возвращает количество элементов множества.
- **`toList_Set`**
  Функция возвращает список (`List`), состоящий из всех элементов данного множества.

`add_Set`

Функция вставляет заданный элемент в множество.

```
bool add_Set( Set* self, Object* ref )
```

Параметры

`self`

Множество, в которое осуществляется вставка.

`ref`

Значение спецификационного типа, вставляемое в множество.

Возвращаемое значение

`true`, если вставка была успешной. В любом другом случае — `false`.

Дополнительная информация

Если множество типизировано, то тип элемента *`ref`* должен совпадать с типом элементов множества.

```
Set* s = create_Set(&type_Integer);
add_Set(s, create_Integer(28));
add_Set(s, create_Integer(47));
add_Set(s, create_Integer(28));
```

Множество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47 }`

`addAll_Set`

Объединяет два множества: добавляет элементы одного множества в другое.

```
bool addAll_Set( Set* self, Set* set )
```

Параметры

`self`

Множество, в которое добавляются элементы.

`set`

Множество, элементы которого добавляются в *`self`*.

Возвращаемое значение

Если в результате объединения в множество *`self`* был добавлен хотя бы один элемент, функция возвращает `true`. В любом другом случае — `false`.

Дополнительная информация

Если множество *`self`* типизировано, то типы всех элементов множества *`set`* должны совпадать с типом элементов множества *`self`* (при этом само множество *`set`* не обязано быть типизированным).

```
bool changed;
```

```
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));
add_Set(s1,create_Integer(47));
add_Set(s2,create_Integer(28));
add_Set(s2,create_Integer(47));
add_Set(s2,create_Integer(63));
changed = addAll_Set(s1,s2);
```

Множество *s1* меняется таким образом:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

Множество *s2* меняется так:

1. `{ 28 }`
2. `{ 28, 47 }`

После вызова `addAll_Set` в множество *s1* добавляется элемент `63`, а переменная *changed* становится равной `true`.

clear_Set

Функция удаляет из множества все элементы.

```
void clear_Set( Set* self )
```

Параметры

self

Очищаемое множество.

Дополнительная информация

```
Set* s = create_Set(&type_Integer);
add_Set(s,create_Integer(28));
add_Set(s,create_Integer(47));
clear_Set(s);
```

Множество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{}`

Тот же результат можно получить, пересоздав множество:

```
Set* s = create_Set(&type_Integer);
...
s = create_Set(elemType_Set(s));
```

Но такой способ менее эффективен, чем вызов функции `clear_Set`.

`contains_Set`

Функция проверяет, содержится ли в множестве элемент, равный данному.

```
bool contains_Set( Set* self, Object* ref )
```

Параметры

`self`

Проверяемое множество.

`ref`

Значение спецификационного типа, поиск которого происходит в множестве.

Возвращаемое значение

`true`, если все элемент *`ref`* содержится в множестве *`self`*. `false` в обратном случае.

Дополнительная информация

Если множество типизировано, то тип элемента *`ref`* должен совпадать с типом элементов множества.

```
bool contains;
Set* s = create_Set(&type_Integer);
add_Set(s,create_Integer(28));
add_Set(s,create_Integer(47));
contains = contains_Set(s,create_Integer(28));
```

Множество *`s`* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 47, 28 }`

Значение переменной *`contains`* после вызова `contains_Set` будет равно `true`.

`containsAll_Set`

Определяет, является ли одно множество подмножеством другого, то есть, содержит ли первое множество все элементы второго.

```
bool containsAll_Set( Set* self, Set* set )
```

Параметры

`self`

Множество, которое нужно проверить на содержание элементов множества *`set`*.

`set`

Предположительное подмножество *`self`*.

Возвращаемое значение

`true`, если все элементы множества *`set`* содержатся в множестве *`self`*. `false` в любом другом случае.

Дополнительная информация

```
bool contains;
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));
add_Set(s1,create_Integer(47));
add_Set(s2,create_Integer(28));
add_Set(s2,create_String("a"));
add_Set(s2,create_Integer(47));
contains = containsAll_Set(s1,s2));
contains = containsAll_Set(s2,s1));
```

Множество *s1* меняется таким образом:

1. { 28 }
2. { 28, 47 }

Множество *s2* меняется так:

1. { 28 }
2. { 28, "a" }
3. { 28, "a", 47 }

После первого вызова containsAll_Set переменная *contains* становится равной false, а после второго вызова — true.

create_Set

Функция создает множество и возвращает спецификационную ссылку типа Set.

```
Set* create_Set( const Type *elem_type )
```

Параметры

elem_type

Указатель на константу-дескриптор типа элементов множества.

Возвращаемое значение

Указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного множества.
Если множество не типизировано (при создании не было наложено ограничений на типы элементов), функция возвращает NULL.

Дополнительная информация

Если параметр *elem_type* нулевой, то типы элементов множества не ограничены.

```
Set* s1 = create_Set(NULL);
Set* s2 = create_Set(&type_Integer);
```

Первым вызовом create_Set создано множество, в которое можно включать любые элементы. Множество, созданное вторым вызовом, может содержать только элементы типа Integer.

Функция `create_Set` определена наряду со стандартной функций `create`.

```
Set* s1 = create(&type_Set, NULL);
Set* s2 = create_Set(NULL);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

`elemType_Set`

Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного множества.

```
Type *elemType_Set(Set* self)
```

Параметры

`self`

Множество, ограничения которого должна вернуть функция.

Возвращаемое значение

Указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного множества.
Если множество не типизировано (при создании не было наложено ограничений на типы элементов), функция возвращает `NULL`.

Дополнительная информация

```
Set* s = create_Set(&type_Integer);
Type *t = elemType_Set(s);
```

Переменная *`t`* приобретет значение `&type_Integer`.

`get_Set`

Функция возвращает спецификационную ссылку на элемент с заданным номером; элементы множества при этом нумеруются произвольно.

```
Object* get_Set( Set* self, int index )
```

Параметры

`self`

Множество, элемент которого должна вернуть функция.

`index`

Номер элемента, который должна вернуть функция.

Возвращаемое значение

Спецификационная ссылка типа `Object`.

Дополнительная информация

Эта функция служит для перебора всех элементов множества. Поскольку множество неупорядочено, элементы нумеруются некоторым произвольным образом. Номер позиции должен находиться в интервале от `0` до размера множества – 1, т. е. `0 ≤ index < size_Set(self)`.

```
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(&type_Integer);
int i;
bool equ;
add_Set(s1,create_Integer(28));
add_Set(s1,create_Integer(47));
add_Set(s1,create_Integer(63));
for(i=0; i < size_Set(s1); i++)
  add_Set(s2, get_Set(s1,i));
equ = equals(s1,s2);
```

Множество *s1* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

Порядок добавления элементов в множество *s2* неопределен, однако в после выполнения цикла все элементы *s1* будут добавлены в *s2* и значение переменной *equ* после вызова `equals` будет равно `true`.

isEmpty_Set

Функция проверяет, пусто ли данное множество.

```
bool isEmpty_Set( Set* self )
```

Параметры

self

Проверяемое множество.

Возвращаемое значение

`true`, если в множестве нет ни одного элемента и `false` в обратном случае.

Дополнительная информация

```
bool empty;
Set* s = create_Set(&type_Integer);
empty = isEmpty_Set(s);
add_Set(s,create_Integer(28));
empty = isEmpty_Set(s);
```

Значение переменной *empty* после первого вызова `isEmpty_Set` будет равно `true`, а после второго (когда в множество будет добавлено число 28) — `false`.

remove_Set

Функция удаляет элемент из множества.

```
void remove_Set( Set* self, Object* ref )
```

Параметры

self

Множество, в которое осуществляется вставка.

ref

Удаляемый из множества элемент.

Дополнительная информация

Если множество типизировано, то тип элемента *ref* должен совпадать с типом элементов множества.

```
Set* s = create_Set(&type_Integer);
add_Set(s,create_Integer(28));
add_Set(s,create_Integer(47));
remove_Set(s,create_Integer(28));
```

Множество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 47 }`

removeAll_Set

Вычитает одно множество из другого: удаляет из первого множества те элементы, которые есть во втором.

```
bool removeAll_Set( Set* self, Set* set )
```

Параметры

self

Множество, из которого удаляются элементы.

set

Множество, элементы которого удаляются из *self*.

Возвращаемое значение

Если в результате вычитания из множества *self* был удален хотя бы один элемент, функция возвращает `true`. В любом другом случае — `false`.

Дополнительная информация

```
bool changed;
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));
add_Set(s1,create_Integer(47));
add_Set(s1,create_Integer(63));
add_Set(s2,create_Integer(28));
add_Set(s2,create_Integer(47));
```

```
changed = removeAll_Set(s1,s2);
```

Множество *s1* меняется таким образом:

1. { 28 }
2. { 28, 47 }
3. { 28, 47, 63 }

Множество *s2* меняется так:

1. { 28 }
2. { 28, 47 }

После вызова `removeAll_Set` в множестве *s1* остается элемент 63, а переменная *changed* становится равной `true`.

`retainAll_Set`

Вычисляет пересечение множеств: оставляет в первом множестве только те элементы, которые есть во втором.

```
bool retainAll_Set( Set* self, Set* set )
```

Параметры

self

Первый аргумент пересечения множеств. Это множество будет хранить результат пересечения.

set

Второй аргумент пересечения множеств.

Возвращаемое значение

`true`, если в пересечении множеств элементов получилось меньше, чем в множестве *self*. В любом другом случае — `false`.

Дополнительная информация

```
bool changed;
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));
add_Set(s1,create_Integer(47));
add_Set(s1,create_Integer(63));
add_Set(s2,create_Integer(28));
add_Set(s2,create_Integer(47));
changed = retainAll_Set(s1,s2);
```

Множество *s1* меняется таким образом:

1. { 28 }
2. { 28, 47 }
3. { 28, 47, 63 }

Множество *s2* меняется так:

1. `{ 28 }`
2. `{ 28, 47 }`

После вызова `retainAll_Set` в множестве *s1* остаются элементы 28 и 47, а переменная *changed* становится равной `true`.

`size_Set`

Функция возвращает количество элементов множества.

```
int size_Set( Set* self )
```

Параметры

self

Множество, мощность которого должна вернуть функция.

Возвращаемое значение

Точное количество элементов в множестве *self*.

Дополнительная информация

```
int size;
Set* s = create_Set(&type_Integer);
size = size_Set(s);
add_Set(s,create_Integer(28));
size = size_Set(s);
```

После первого вызова `size_Set` переменная *size* становится равной нулю. После второго вызова `size_Set` переменная равна `1`: в множество был добавлен один элемент.

`toList_Set`

Функция возвращает список (`List`), состоящий из всех элементов данного множества.

```
List* toList_Set( Set* self )
```

Параметры

self

Данное множество.

Возвращаемое значение

Спецификационная ссылка типа `List`.

Дополнительная информация

Порядок элементов в списке не определен. Возвращаемый список сохраняет типизацию множества: если элементы множества были ограничены каким-либо типом, то и элементы списка будут ограничены тем же типом.

```
Set* s = create_Set(&type_Integer);
List* l;
Type *t;
```

```
add_Set(s,create_Integer(28));
add_Set(s,create_Integer(47));
l = toList_Set(s);
t = elemType_Set(l);
```

Множество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`

Список *l*, возвращенный функцией `toList_Set`, содержит элементы: `{ 28, 47 }`; они могут располагаться в любом порядке. Вызов `elemType_Set(`*l*`)` возвращает `&type_Integer`.

14.2.14 `Map`

Тип `Map` является контейнерным типом, реализующим отображение элементов из области определения в область значений. Элемент из области определения называется ключом, а соответствующий элемент из области значений — просто значением. При этом одному ключу соответствует ровно одно значение.

Элементами отображения могут быть любые спецификационные ссылки. При создании отображения можно ограничить тип его элементов, причём отдельно для элементов области определения и для элементов области значений. Такое отображение называется типизированным, а все функции, работающие с ним и имеющие параметр типа `Object*`, будут проверять, чтобы тип этого параметра совпадал с типом элементов соответствующей области.

Два отображения считаются равными, если они имеют одинаковые множества ключей, и каждому ключу в обоих отображениях соответствуют одинаковые значения. При этом не учитывается типизация отображений, в частности, пустые отображения всегда равны.

Заголовочный файл: `atl/map.h`

- **`clear_Map`**
  Функция удаляет из отображения все элементы.
- **`containsKey_Map`**
  Функция проверяет, содержит ли отображение ключ, равный данному.
- **`containsValue_Map`**
  Функция проверяет, содержит ли отображение ключ, равный данному.
- **`create_Map`**
  Функция создает отображение и возвращает спецификационную ссылку типа `Map`.
- **`get_Map`**
  Функция возвращает спецификационную ссылку на значение, соответствующее ключу.
- **`getKey_Map`**
  Функция возвращает спецификационную ссылку на какой-нибудь ключ, которому соответствует данное значение.
- **`isEmpty_Map`**
  Функция проверяет, пусто ли данное отображение.
- **`key_Map`**
  Функция возвращает ключ отображения в указанной позиции.
- **`keyType_Map`**
  Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены ключи данного отображения.
- **`put_Map`**
  Функция добавляет в отображение пару «ключ-значение».
- **`putAll_Map`**
  Добавляет в первое отображение все пары «ключ–значение» из второго отображения.
- **`remove_Map`**
  Функция удаляет из отображения пару «ключ-значение» для данного ключа.

- **`size_Map`**
  Функция возвращает размер отображения.
- **`valueType_Map`**
  Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного отображения.

`clear_Map`

Функция удаляет из отображения все элементы.

```
void clear_Map( Set* self )
```

Параметры

`self`

Очищаемое отображение.

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a"));
put_Map(m,create_Integer(47),create_String("b"));
clear_Map(m);
```

Отображение *`m`* меняется по мере выполнения этого кода так:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`
3. `{}`

Тот же результат можно получить, пересоздав множество:

```
Map* m;
...
m = create_Map(keyType_Map(m), valueType_Map(m));
```

Но такой способ менее эффективен, чем вызов функции `clear_Map`.

`containsKey_Map`

Функция проверяет, содержит ли отображение ключ, равный данному.

```
bool containsKey_Map( Map* self, Object* key )
```

Параметры

`self`

Проверяемое отображение.

`key`

Ключ, поиск которого происходит в отображении.

Возвращаемое значение

`true`, если ключ *`key`* содержится в *`self`*. `false` в обратном случае.

Дополнительная информация

Если область определения типизирована, то тип элемента *`key`* должен совпадать с типом ключей отображения.

```
bool contains;
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a"));
put_Map(m,create_Integer(47),create_String("b"));
contains = containsKey_Map(m,create_Integer(28);
```

Отображение *m* меняется по мере выполнения этого кода так:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`

Значение переменной *contains* после вызова `containsKey_Map` будет равно `true`.

containsValue_Map

Функция проверяет, содержит ли отображение ключ, равный данному.

```
bool containsValue_Map( Map* self, Object* value )
```

Параметры

self

Проверяемое отображение.

value

Значение, поиск которого происходит в отображении.

Возвращаемое значение

`true`, если значение *value* содержится в *self*. `false` в обратном случае.

Дополнительная информация

Если область значений типизирована, то тип элемента *value* должен совпадать с типом значений отображения.

```
bool contains;
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a"));
put_Map(m,create_Integer(47),create_String("b"));
contains = containsValue_Map(m,create_String("b"));
```

Отображение *m* меняется по мере выполнения этого кода так:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`

Значение переменной *contains* после вызова `containsValue_Map` будет равно `true`.

create_Map

Функция создает отображение и возвращает спецификационную ссылку типа `Map`.

```
Map* create_Map( const Type *key_type, const Type *val_type )
```

Параметры

key_type

Указатель на константу-дескриптор типа ключей отображения.

val_type

Указатель на константу-дескриптор типа значений отображения.

Возвращаемое значение

Спецификационная ссылка типа `Map`.

Дополнительная информация

Если параметр *key_type* нулевой, то типы элементов области определения не ограничены. В противном случае параметр должен являться указателем на константу-дескриптор типа для ключей отображения. Аналогично, если параметр *val_type* нулевой, то типы элементов области значений не ограничены. Иначе параметр должен являться указателем на константу-дескриптор типа для значений отображения.

Первая строка примера создает отображение без типизации; вторая строка создает отображение с ключами типа `Integer`; третья строка создает отображение из `Integer` в `String`.

```
Map* m1 = create_Map(NULL, NULL);
Map* m2 = create_Map(&type_Integer, NULL);
Map* m3 = create_Map(&type_Integer, &type_String);
```

Функция `create_Map` определена наряду со стандартной функций `create`.

```
Map* m1 = create(&type_Set, NULL, NULL);
Map* m2 = create_Map(NULL, NULL);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование `create` не является типобезопасным и вызывает предупреждение транслятора (см. описание функции `create`).

get_Map

Функция возвращает спецификационную ссылку на значение, соответствующее ключу.

```
Object* get_Map( Map* self, Object* key )
```

Параметры

self

Отображение, элемент которого должна вернуть функция.

key

Ключ, соответствующий нужному значению.

Возвращаемое значение

Возвращается значение, соответствующее переданному ключу. Если отображение не содержит данного ключа, возвращает `NULL`.

Дополнительная информация

Если область определения типизирована, то тип элемента *key* должен совпадать с типом ключей отображения.

```
Map* m = create_Map(&type_Integer, &type_String);
Object* val;
put_Map(m,create_Integer(28),create_String("a"));
put_Map(m,create_Integer(47),create_String("b"));
val = get_Map(m,create_Integer(28));
```

Отображение *m* меняется по мере выполнения этого кода так:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`

Значение переменной *val* после вызова `get_Map` будет равно `"a"`.

getKey_Map

Функция возвращает спецификационную ссылку на какой-нибудь ключ, которому соответствует данное значение.

```
Object* getKey_Map( Map* self, Object* value )
```

Параметры

self

Отображение, элемент которого должна вернуть функция.

value

Значение, соответствующее нужному ключу.

Возвращаемое значение

Возвращается один из ключей, соответствующих данному значению. Если отображение не содержит ни одного ключа, которому соответствует данное значение, возвращает `NULL`.

Дополнительная информация

Если область значений типизирована, то тип элемента *value* должен совпадать с типом значений отображения.

```
bool equ;
Map* m = create_Map(&type_Integer, &type_String);
Object* key;
Object* val;
put_Map(m,create_Integer(28),create_String("a"));
put_Map(m,create_Integer(47),create_String("a"));
val = create_String("a");
key = getKey_Map(m,val);
equ = equals( get_Map(m,key), val);
```

Отображение *m* меняется по мере выполнения этого кода так:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "a" }`

Значение спецификационной ссылки *key* после вызова `getKey_Map` будет равно `28` или `47`. Значение переменной *equ* после вызова `equals` равно `true`.

`isEmpty_Map`

Функция проверяет, пусто ли данное отображение.

```
bool isEmpty_Map( Map* self )
```

Параметры

`self`

Проверяемое отображение.

Возвращаемое значение

Возвращает `true`, если отображение не содержит ни одной пары «ключ–значение»; в противном случае возвращает `false`.

Дополнительная информация

```
bool empty;
Map* m = create_Map(&type_Integer, &type_String);
empty = isEmpty_Map(m);
put_Map(m,create_Integer(28),create_String("a"));
empty = isEmpty_Map(m);
```

Значение переменной *`empty`* после первого вызова `isEmpty_Map` будет равно `true`, а после второго (когда в отображение будет добавлена пара `28 → "a"`) — `false`.

`key_Map`

Функция возвращает ключ отображения в указанной позиции.

```
Object* key_Map( Map* self, int index )
```

Параметры

`self`

Отображение, ключ которого должна вернуть функция.

`index`

Номер ключа, который должна вернуть функция.

Возвращаемое значение

Спецификационная ссылка на ключ отображения в позиции *`index`*.

Дополнительная информация

Эта функция служит для перебора всех ключей отображения. Поскольку множество ключей неупорядочено, элементы нумеруются некоторым произвольным образом. Номер позиции должен находиться в интервале от `0` до размера множества – 1, т. е. `0 ≤ index < size_Map(self)`.

```
Map* m1 = create_Map(&type_Integer, &type_String);
Map* m2 = create_Map(&type_Integer, &type_String);
int i;
bool equ;
put_Map(m1,create_Integer(28),create_String("a"));
```

```
put_Map(m1,create_Integer(47),create_String("b"));
for (i = 0; i < size_Map(m1); i++) {
  Object* key = key_Map(m1,i);
  Object* val = get_Map(m1,key);
  put_Map(m2,key,val);
}
equ = equals(m1,m2);
```

Отображение *m1* меняется по мере выполнения этого кода так:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`

Аналогично изменяется отображение *m2*, но из-за неупорядоченности ключей в отображении порядок добавления пар «ключ-значение» однозначно описать нельзя. Значение переменной *equ* после вызова `equals` будет равно `true`.

keyType_Map

Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены ключи данного отображения.

```
Type *keyType_Map( Map* self )
```

Параметры

self

Спецификационная ссылка на данное отображение.

Возвращаемое значение

Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены ключи данного отображения.

Дополнительная информация

Если область определения отображения не типизирована, возвращает `NULL`.

```
Map* m = create_Map(&type_Integer, &type_String);
Type *t = keyType_Map(m);
```

Значение спецификационной ссылки *t* после вызова функции `keyType_Map` будет равно `&type_Integer`.

put_Map

Функция добавляет в отображение пару «ключ-значение».

```
Object* put_Map( Map* self, Object* key, Object* value )
```

Параметры

self

Отображение, в которое добавляется пара «ключ-значение».

key

Ключ добавляемой в отображение пары.

value

Значение добавляемой в отображение пары.

Возвращаемое значение

Если отображение не содержит ключа *key*, функция возвращает NULL. Если же отображение уже содержит ключ *key*, то функция возвращает соответствующее ему старое значение.

Дополнительная информация

Если область определения типизирована, то тип элемента *key* должен совпадать с типом ключей отображения. Если область значений типизирована, то тип элемента *value* должен совпадать с типом значений отображения.

Если отображение не содержит ключа *key*, то в отображение добавляется ключ *key* и соответствующее ему значение *value*. Если же отображение уже содержит ключ *key*, то функция возвращает соответствующее ему старое значение, а в отображении старое значение заменяется новым (*value*).

```
String *val;
Map* m = create_Map(&type_Integer, &type_String);
val = put_Map(m,create_Integer(28),create_String("a"));
val = put_Map(m,create_Integer(47),create_String("b"));
val = put_Map(m,create_Integer(28),create_String("c"));
```

Отображение *m* меняется по мере выполнения этого кода так:

1. { 28 → "a" }
2. { 28 → "a", 47 → "b" }
3. { 28 → "c", 47 → "b" }

Значение *val* после каждого из первых двух вызовов put_Map равно NULL; после третьего вызова оно становится равным "a".

putAll_Map

Добавляет в первое отображение все пары «ключ–значение» из второго отображения.

```
void putAll_Map( Map* self, Map* t )
```

Параметры

self

Отображение, в которое добавляются элементы.

t

Отображение, элементы которого добавляются в *self*.

Дополнительная информация

Если область определения отображения *self* типизирована, то типы всех ключей отображения *t* должны совпадать с типом ключей отображения *self*. Аналогично, если область значений отображения *self* типизирована, то типы всех значений отображения *t* должны совпадать с типом значений отображения *self*. При этом не требуется, чтобы области определения и значений отображения *t* были типизированы.

Если отображение *self* уже содержит некоторый ключ из отображения *t*, то соответствующее ему значение заменяется на новое.

```
Map* m1 = create_Map(&type_Integer, &type_String);
Map* m2 = create_Map(NULL, NULL);
put_Map(m1,create_Integer(28),create_String("a"));
put_Map(m1,create_Integer(63),create_String("b"));
put_Map(m2,create_Integer(28),create_String("c"));
put_Map(m2,create_Integer(47),create_String("d"));
putAll_Map(m1,m2);
```

Отображение *m1* меняется таким образом:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 63 → "b" }`

Отображение *m2* меняется так:

1. `{ 28 → "c" }`
2. `{ 28 → "c", 47 → "d" }`

После вызова `putAll_Map` в отображение *m1* добавляется пара `47 → "d"` и значение по ключу `28` меняется на `"c"`:

3. `{ 28 → "c", 47 → "d", 63 → "b" }`

`remove_Map`

Функция удаляет из отображения пару «ключ-значение» для данного ключа.

```
Object* remove_Map(Map* self, Object* key);
```

Параметры

self

Отображение, из которого удаляется элемент.

key

Ключ удаляемой пары «ключ-значение».

Возвращаемое значение

Возвращает значение, которое соответствовало ключу *key* в отображении *self*, или `NULL`, если такого значения не было.

Дополнительная информация

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a"));
remove_Map(m,create_Integer(28));
remove_Map(m,create_Integer(28));
```

Первый вызов функции `remove_Map` возвращает строковое значение, соответствующее ключу `28`: `"a"`. Второй вызов возвращает `NULL`, так как пара с ключом `28` уже была удалена.

`size_Map`

Функция возвращает размер отображения.

```
int size_Map( Map* self )
```

Параметры

`self`

Отображение, размер которого должна вернуть функция.

Возвращаемое значение

Функция взвращает точное количество пар в отображении.

Дополнительная информация

Размером считается количество пар «ключ–значение», содержащихся в отображении.

```
int size;
Map* m = create_Map(&type_Integer, &type_String);
size = size_Map(m);
put_Map(m,create_Integer(28),create_String("a"));
size = size_Map(m);
```

После первого вызова `size_Map` переменная *`size`* становится равной нулю. После второго вызова `size_Map` переменная равна `1`, так как в отображение был добавлен один элемент.

`valueType_Map`

Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного отображения.

```
Type *valueType_Map( Map* self )
```

Параметры

`self`

Спецификационная ссылка на данное отображение.

Возвращаемое значение

Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного отображения.

Дополнительная информация

Если область определения отображения не типизирована, возвращает `NULL`.

```
Map* m = create_Map(&type_Integer, &type_String);
Type *t = valueType_Map(m);
```

Значение спецификационной ссылки *`t`* после вызова функции `valueType_Map` будет равно `&type_String`.

14.2.15 `MultiSet`

Тип `MultiSet` является контейнерным типом, реализующим мультимножество элементов (т.е. множество, элементы которого могут повторяться).

Элементами мультимножества могут быть любые спецификационные ссылки. При создании мультимножества можно ограничить тип его элементов. Такое мультимножество называется типизированным, а все функции, работающие с ним и имеющие параметр типа `Object*`, будут проверять, чтобы тип этого параметра совпадал с типом элементов мультимножества.

Два мультимножества считаются равными, если они имеют одинаковые элементы (с учетом кратности). При этом не учитывается типизация мультимножеств, в частности, пустые мультимножества всегда равны.

Заголовочный файл: `atl/multiset.h`

- **`add_MultiSet`**
  Функция вставляет заданный элемент в мультимножество.
- **`addAll_MultiSet`**
  Добавляет элементы одного мультимножества в другое.
- **`clear_MultiSet`**
  Функция удаляет из мультимножества все элементы.
- **`contains_MultiSet`**
  Функция возвращает кратность вхождения в мультимножество элемента, равного данному.
- **`containsAll_MultiSet`**
  Определяет, является ли одно мультимножество подмультимножеством другого, то есть, содержит ли первое мультимножество все элементы второго (с учетом кратности).
- **`create_MultiSet`**
  Функция создает мультимножество и возвращает спецификационную ссылку типа MultiSet.
- **`elemType_MultiSet`**
  Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного мультимножества.
- **`get_MultiSet`**
  Функция возвращает спецификационную ссылку на элемент с заданным номером; элементы мультимножества при этом нумеруются произвольно.
- **`isEmpty_MultiSet`**
  Функция проверяет, пусто ли данное мультимножество.
- **`remove_MultiSet`**
  Функция однократно удаляет элемент из мультимножества (т.е. кратность вхождения этого элемента в мультимножество уменьшается на 1).
- **`removeAll_MultiSet`**
  Вычитает одно мультимножество из другого: удаляет из первого мультимножества те элементы, которые есть во втором (с учетом кратности).

- **`removeCount_MultiSet`**
  Функция уменьшает кратность вхождения данного элемента в мультимножество на указанную величину.
- **`removeFull_MultiSet`**
  Функция полностью удаляет элемент из мультимножества (т.е. кратность вхождения этого элемента в мультимножество становится равна `0`).
- **`retainAll_MultiSet`**
  Вычисляет пересечение мультимножеств: оставляет в первом мультимножестве только те элементы, которые есть во втором (с учетом кратности).
- **`size_MultiSet`**
  Функция возвращает количество элементов мультимножества.
- **`toList_MultiSet`**
  Функция возвращает список (`List`), состоящий из всех элементов данного мультимножества.
- **`toSet_MultiSet`**
  Функция возвращает множество (`Set`), содержащее все элементы данного мультимножества.

`add_MultiSet`

Функция вставляет заданный элемент в мультимножество.

```
bool add_MultiSet( MultiSet* self, Object* ref )
```

Параметры

`self`

мультимножество, в которое осуществляется вставка.

`ref`

Значение спецификационного типа, вставляемое в мультимножество.

Возвращаемое значение

`true`, если вставка была успешной. В любом другом случае — `false`.

Дополнительная информация

Если мультимножество типизировано, то тип элемента *`ref`* должен совпадать с типом элементов мультимножества.

```
MultiSet* s = create_MultiSet(&type_Integer);
add_MultiSet(s, create_Integer(28));
add_MultiSet(s, create_Integer(47));
add_MultiSet(s, create_Integer(28));
```

мультимножество *`s`* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

`addAll_MultiSet`

Добавляет элементы одного мультимножества в другое.

```
bool addAll_MultiSet( MultiSet* self, MultiSet* set )
```

Параметры

self

мультимножество, в которое добавляются элементы.

set

мультимножество, элементы которого добавляются в *self*.

Возвращаемое значение

Если в результате объединения в мультимножество *self* был добавлен хотя бы один элемент, функция возвращает `true`. В любом другом случае — `false`.

Дополнительная информация

Если мультимножество *self* типизировано, то типы всех элементов мультимножества *set* должны совпадать с типом элементов мультимножества *self* (при этом само мультимножество *set* не обязано быть типизированным).

```
bool changed;
MultiSet* s1 = create_MultiSet(&type_Integer);
MultiSet* s2 = create_MultiSet(NULL);
add_MultiSet(s1,create_Integer(28));
add_MultiSet(s1,create_Integer(47));
add_MultiSet(s2,create_Integer(28));
add_MultiSet(s2,create_Integer(47));
add_MultiSet(s2,create_Integer(63));
changed = addAll_MultiSet(s1,s2);
```

мультимножество *s1* меняется таким образом:

1. `{ 28 }`
2. `{ 28, 47 }`

мультимножество *s2* меняется так:

3. `{ 28 }`
4. `{ 28, 47 }`
5. `{ 28, 47, 63 }`

После вызова `addAll_MultiSet` мультимножество *s1* меняется так:

6. `{ 28, 47, 28, 47, 63 }`

а переменная *changed* становится равной `true`.

`clear_MultiSet`

Функция удаляет из мультимножества все элементы.

```
void clear_MultiSet( MultiSet* self )
```

Параметры

self

Очищаемое мультимножество.

Дополнительная информация

```
MultiSet* s = create_MultiSet(&type_Integer);
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(47));
clear_MultiSet(s);
```

мультимножество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{}`

Тот же результат можно получить, пересоздав мультимножество:

```
MultiSet* s = create_MultiSet(&type_Integer);
...
s = create_MultiSet(elemType_MultiSet(s));
```

Но такой способ менее эффективен, чем вызов функции `clear_MultiSet`.

`contains_MultiSet`

Функция возвращает кратность вхождения в мультимножество элемента, равного данному.

```
int contains_MultiSet( MultiSet* self, Object* ref )
```

Параметры

self

Проверяемое мультимножество.

ref

Значение спецификационного типа, поиск которого происходит в мультимножестве.

Возвращаемое значение

Кратность вхождения в мультимножество *self* элемента *ref*.

Дополнительная информация

Если мультимножество типизировано, то тип элемента *ref* должен совпадать с типом элементов мультимножества.

```
int contains;
MultiSet* s = create_MultiSet(&type_Integer);
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(47));
add_MultiSet(s,create_Integer(28));
contains = contains_MultiSet(s,create_Integer(28));
```

мультимножество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

Значение переменной *contains* после вызова `contains_MultiSet` будет равно 2.

containsAll_MultiSet

Определяет, является ли одно мультимножество подмультимножеством другого, то есть, содержит ли первое мультимножество все элементы второго.

```
bool containsAll_MultiSet( MultiSet* self, MultiSet* set )
```

Параметры

self

Мультимножество, которое нужно проверить на содержание элементов мультимножества *set*.

set

Предположительное подмультимножество *self*.

Возвращаемое значение

`true`, если все элементы мультимножества *set* содержатся в мультимножестве *self*. `false` в любом другом случае.

Дополнительная информация

```
bool contains;
MultiSet* s1 = create_MultiSet(&type_Integer);
MultiSet* s2 = create_MultiSet(NULL);
add_MultiSet(s1,create_Integer(28));
add_MultiSet(s1,create_Integer(47));
add_MultiSet(s2,create_Integer(28));
add_MultiSet(s2,create_String("a"));
add_MultiSet(s2,create_Integer(47));
contains = containsAll_MultiSet(s1,s2));
contains = containsAll_MultiSet(s2,s1));
```

мультимножество *s1* меняется таким образом:

1. `{ 28 }`
2. `{ 28, 47 }`

мультимножество *s2* меняется так:

3. `{ 28 }`
4. `{ 28, a }`
5. `{ 28, a, 47 }`

После первого вызова containsAll_MultiSet переменная *contains* становится равной false, а после второго вызова — true.

create_MultiSet

Функция создает мультимножество и возвращает спецификационную ссылку типа MultiSet.

```
MultiSet* create_MultiSet( const Type *elem_type )
```

Параметры

elem_type

Указатель на константу-дескриптор типа элементов мультимножества.

Возвращаемое значение

Спецификационная ссылка типа MultiSet на вновь созданное мультимножество.

Дополнительная информация

Если параметр *elem_type* нулевой, то типы элементов мультимножества не ограничены.

```
MultiSet* s1 = create_MultiSet(NULL);
MultiSet* s2 = create_MultiSet(&type_Integer);
```

Первым вызовом create_MultiSet создано мультимножество, в которое можно включать любые элементы. Мультимножество, созданное вторым вызовом, может содержать только элементы типа Integer.

Функция create_MultiSet определена наряду со стандартной функций create.

```
MultiSet* s1 = create(&type_MultiSet, NULL);
MultiSet* s2 = create_MultiSet(NULL);
```

Два показанных способа создания спецификационной ссылки функционально эквивалентны, однако использование create не является типобезопасным и вызывает предупреждение транслятора (см. описание функции create).

elemType_MultiSet

Функция возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного мультимножества.

```
Type *elemType_MultiSet(MultiSet* self)
```

Параметры

self

мультимножество, ограничения которого должна вернуть функция.

Возвращаемое значение

Указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного мультимножества.

Если мультимножество не типизировано (при создании не было наложено ограничений на типы элементов), функция возвращает NULL.

Дополнительная информация

```
MultiSet* s = create_MultiSet(&type_Integer);
```

```
Type *t = elemType_MultiSet(s);
```

Переменная *t* приобретет значение `&type_Integer`.

get_MultiSet

Функция возвращает спецификационную ссылку на элемент с заданным номером; элементы мультимножества при этом нумеруются произвольно.

```
Object* get_MultiSet( MultiSet* self, int index )
```

Параметры

self

мультимножество, элемент которого должна вернуть функция.

index

Номер элемента, который должна вернуть функция.

Возвращаемое значение

Спецификационная ссылка типа `Object`.

Дополнительная информация

Эта функция служит для перебора всех элементов мультимножества. Поскольку мультимножество неупорядочено, элементы нумеруются некоторым произвольным образом. Номер позиции должен находиться в интервале от `0` до размера мультимножества `–` `1`, т. е. `0 ≤` *index* `<` `size_MultiSet(`*self*`)`.

```
MultiSet* s1 = create_MultiSet(&type_Integer);
MultiSet* s2 = create_MultiSet(&type_Integer);
int i;
bool equ;
add_MultiSet(s1,create_Integer(28));
add_MultiSet(s1,create_Integer(47));
add_MultiSet(s1,create_Integer(63));
for(i=0; i < size_MultiSet(s1); i++)
  add_MultiSet(s2, get_MultiSet(s1,i));
equ = equals(s1,s2);
```

мультимножество *s1* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

Значение переменной *equ* после вызова `equals` будет равно `true`, поскольку в цикле все элементы *s1* будут добавлены в *s2*, то есть мультимножества будут равны.

isEmpty_MultiSet

Функция проверяет, пусто ли данное мультимножество.

```
bool isEmpty_MultiSet( MultiSet* self )
```

Параметры

self

Проверяемое мультимножество.

Возвращаемое значение

`true`, если в мультимножестве нет ни одного элемента и `false` в обратном случае.

Дополнительная информация

```
bool empty;
MultiSet* s = create_MultiSet(&type_Integer);
empty = isEmpty_MultiSet(s);
add_MultiSet(s,create_Integer(28));
empty = isEmpty_MultiSet(s);
```

Значение переменной *empty* после первого вызова `isEmpty_MultiSet` будет равно `true`, а после второго (когда в мультимножество будет добавлено число `28`) — `false`.

remove_MultiSet

Функция однократно удаляет элемент из мультимножества (т.е. кратность вхождения этого элемента в мультимножество уменьшается на `1`).

```
bool remove_MultiSet( MultiSet* self, Object* ref )
```

Параметры

self

мультимножество, из которого удаляется элемент.

ref

Удаляемый из мультимножества элемент.

Возвращаемое значение

Если в результате вычитания из мультимножества *self* был удален хотя бы один элемент, функция возвращает `true`. В любом другом случае — `false`.

Дополнительная информация

Если мультимножество типизировано, то тип элемента *ref* должен совпадать с типом элементов мультимножества.

```
bool changed;
MultiSet* s = create_MultiSet(&type_Integer);
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(47));
add_MultiSet(s,create_Integer(28));
changed = remove_MultiSet(s,create_Integer(28));
changed = remove_MultiSet(s,create_Integer(63));
```

мультимножество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`

2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`
4. `{ 47, 28 }`

Значение переменной *changed* после первого вызова `remove_MultiSet` будет равно `true`, а после второго — `false`.

`removeAll_MultiSet`

Вычитает одно мультимножество из другого: удаляет из первого мультимножества те элементы, которые есть во втором.

```
bool removeAll_MultiSet( MultiSet* self, MultiSet* set )
```

Параметры

self

мультимножество, из которого удаляются элементы.

set

мультимножество, элементы которого удаляются из *self*.

Возвращаемое значение

Если в результате вычитания из мультимножества *self* был удален хотя бы один элемент, функция возвращает `true`. В любом другом случае — `false`.

Дополнительная информация

```
bool changed;
MultiSet* s1 = create_MultiSet(&type_Integer);
MultiSet* s2 = create_MultiSet(NULL);
add_MultiSet(s1,create_Integer(28));
add_MultiSet(s1,create_Integer(47));
add_MultiSet(s1,create_Integer(28));
add_MultiSet(s1,create_Integer(63));
add_MultiSet(s2,create_Integer(28));
add_MultiSet(s2,create_Integer(47));
changed = removeAll_MultiSet(s1,s2);
```

мультимножество *s1* меняется таким образом:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`
4. `{ 28, 47, 28, 63 }`

мультимножество *s2* меняется так:

5. `{ 28 }`
6. `{ 28, 47 }`

После вызова `removeAll_MultiSet` в мультимножестве *s1* остаются элементы `28` (однократно) и `63`, а переменная *changed* становится равной `true`.

`removeCount_MultiSet`

Функция уменьшает кратность вхождения данного элемента в мультимножество на указанную величину.

```
int removeCount_MultiSet( MultiSet* self, Object* ref, int count )
```

Параметры

`self`

мультимножество, из которого удаляется элемент.

`ref`

Удаляемый из мультимножества элемент.

`count`

На сколько требуется уменьшить кратность вхождения элемента в мультимножество.

Возвращаемое значение

Количество реально удаленных вхождений элемента *`ref`* в мультимножество *`self`*.

Дополнительная информация

Если мультимножество типизировано, то тип элемента *`ref`* должен совпадать с типом элементов мультимножества.

```
int changed;
MultiSet* s = create_MultiSet(&type_Integer);
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(47));
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(28));
changed = removeCount_MultiSet(s,create_Integer(28),2);
changed = removeCount_MultiSet(s,create_Integer(28),2);
```

мультимножество *`s`* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28 47 }`
3. `{ 28 47 28 }`
4. `{ 28 47 28 28 }`
5. `{ 47 28 }`
6. `{ 47}`

После первого вызова `removeCount_MultiSet` переменная *`changed`* становится равной `2`, а после второго — `1`.

`removeFull_MultiSet`

Функция полностью удаляет элемент из мультимножества (т.е. кратность вхождения этого элемента в мультимножество становится равна `0`).

```
int removeFull_MultiSet( MultiSet* self, Object* ref )
```

Параметры

self

мультимножество, из которого удаляется элемент.

ref

Удаляемый из мультимножества элемент.

Возвращаемое значение

Количество реально удаленных вхождений элемента *ref* в мультимножество *self*.

Дополнительная информация

Если мультимножество типизировано, то тип элемента *ref* должен совпадать с типом элементов мультимножества.

```
int changed;
MultiSet* s = create_MultiSet(&type_Integer);
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(47));
add_MultiSet(s,create_Integer(28));
changed = removeFull_MultiSet(s,create_Integer(28));
```

мультимножество *s* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`
4. `{ 47 }`

После вызова `removeFull_MultiSet` переменная *changed* становится равной 2.

`retainAll_MultiSet`

Вычисляет пересечение мультимножеств: оставляет в первом мультимножестве только те элементы, которые есть во втором (с учетом кратности).

```
bool retainAll_MultiSet( MultiSet* self, MultiSet* set )
```

Параметры

self

Первый аргумент пересечения мультимножеств. Это мультимножество будет хранить результат пересечения.

set

Второй аргумент пересечения мультимножеств.

Возвращаемое значение

`true`, если в пересечении мультимножеств элементов получилось меньше, чем в мультимножестве *self*. В любом другом случае — `false`.

Дополнительная информация

```
bool changed;
```

```
MultiSet* s1 = create_MultiSet(&type_Integer);
MultiSet* s2 = create_MultiSet(NULL);
add_MultiSet(s1,create_Integer(28));
add_MultiSet(s1,create_Integer(47));
add_MultiSet(s1,create_Integer(28));
add_MultiSet(s1,create_Integer(63));
add_MultiSet(s2,create_Integer(28));
add_MultiSet(s2,create_Integer(47));
changed = retainAll_MultiSet(s1,s2);
```

мультимножество *s1* меняется таким образом:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`
4. `{ 28, 47, 28, 63 }`

мультимножество *s2* меняется так:

5. `{ 28 }`
6. `{ 28, 47 }`

После вызова `retainAll_MultiSet` в мультимножестве *s1* остаются элементы `28` и `47`, а переменная *changed* становится равной `true`.

size_MultiSet

Функция возвращает количество элементов мультимножества.

```
int size_MultiSet( MultiSet* self )
```

Параметры

self

мультимножество, мощность которого должна вернуть функция.

Возвращаемое значение

Точное количество элементов в мультимножестве *self*.

Дополнительная информация

```
int size;
MultiSet* s = create_MultiSet(&type_Integer);
size = size_MultiSet(s);
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(28));
size = size_MultiSet(s);
```

После первого вызова `size_MultiSet` переменная *size* становится равной нулю. После второго вызова `size_MultiSet` переменная равна `2`: в мультимножество было добавлено два (равных) элемента.

`toList_MultiSet`

Функция возвращает список (`List`), состоящий из всех элементов данного мультимножества.

```
List* toList_MultiSet( MultiSet* self )
```

Параметры

`self`

Данное мультимножество.

Возвращаемое значение

Спецификационная ссылка типа `List`.

Дополнительная информация

Порядок элементов в списке не определен. Возвращаемый список сохраняет типизацию мультимножества: если элементы мультимножества были ограничены каким-либо типом, то и элементы списка будут ограничены тем же типом.

```
MultiSet* s = create_MultiSet(&type_Integer);
List* l;
Type *t;
add_MultiSet(s,create_Integer(28));
add_MultiSet(s,create_Integer(47));
add_MultiSet(s,create_Integer(28));
l = toList_MultiSet(s);
t = elemType_MultiSet(l);
```

мультимножество *`s`* меняется по мере выполнения этого кода так:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

Список *`l`*, возвращенный функцией `toList_MultiSet`, содержит элементы: `< 28, 47, 28 >`; они могут располагаться в любом порядке. Вызов `elemType_MultiSet(`*`l`*`)` возвращает `&type_Integer`.

`toSet_MultiSet`

Функция возвращает множество (`Set`), содержащее все элементы данного мультимножества.

```
Set* toSet_MultiSet( MultiSet* self )
```

Параметры

`self`

Данное мультимножество.

Возвращаемое значение

Спецификационная ссылка типа `Set`.

Дополнительная информация

Возвращаемое множество сохраняет типизацию мультимножества: если элементы мультимножества были ограничены каким-либо типом, то и элементы множества будут ограничены тем же типом.

```
MultiSet* ms = create_MultiSet(&type_Integer);
Set* s;
Type *t;
add_MultiSet(ms,create_Integer(28));
add_MultiSet(ms,create_Integer(47));
add_MultiSet(ms,create_Integer(28));
s = toSet_MultiSet(ms);
t = elemType_MultiSet(s);
```

мультимножество *ms* меняется по мере выполнения этого кода так:

1. `{ 28`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

Множество *s*, возвращенное функцией `toSet_MultiSet`, содержит элементы: `{ 28, 47 }`. Вызов `elemType_MultiSet(`*s*`)` возвращает `&type_Integer`.