Справочное руководство по языку Cool *

Содержание

1. Введение

Это руководство описывает язык программирования Cool: учебный объектно-ориентированный язык (*C*lassroom *O*bject-*O*riented *L*anguage). Cool — небольшой язык, который может быть реализован путем приложения вполне разумных усилий в рамках семестрового курса. При этом Cool содержит множество возможностей, присущих современным языкам программирования, включая объекты, статическую типизацию и автоматическое управление памятью.

Программа на Cool представляет собой множество *классов* (*classes*). Класс инкапсулирует в себе переменные и процедуры типа данных. Экземплярами классов являются *объекты* (*objects*). В Cool классы и типы тождественны: каждый класс определяет тип. Классы позволяют определять новые типы данных и ассоциированные с ними процедуры (или *методы*, *methods*), предназначенные для работы с этими типами данных. Наследование позволяет новым типам расширять поведение уже существующих.

Cool является языком *выражений* (*expression* language). Большинство конструкций Cool являются выражениями, и каждое выражение имеет значение и тип. Cool является *типобезопасным* (*type safe*) языком: гарантируется, что процедуры применяются к данным корректного типа. Хотя статическая типизация налагает строгие правила на программирование на языке Cool, она гарантирует, что в процессе выполнения программ, написанных на Cool, не могут возникнуть никакие ошибки типизации времени выполнения.

Это руководство состоит из неформальной и формальной частей. Неформальный обзор языка содержится в первой половине документа (до раздела 9 включительно). Формальное описание начинается с раздела 10.

2. Начало работы

Читателям, желающим для начала получить представление о Cool, следует начать с чтения и запуска программ-примеров, находящихся в директории `/home/compilers_course/examples`. Файлы исходного кода Cool имеют расширение `.cl`, а ассемблерные файлы — расширение `.s`. Компилятор Cool — `/home/compilers_course/bin/coolc`. Чтобы скомпилировать программу, выполните следующую команду:

```
coolc [ -o fileout ] file1.cl file2.cl ... filen.cl
```

Файлы с `file1.cl` по `filen.cl` компилируются, как если бы они были просто соединены в один большой файл. Каждый файл должен определять множество полных классов — определение класса не может быть разделено на несколько файлов. Опция `-o` задает опциональное имя, используемое для выходного файла с ассемблерным кодом. Если `fileout` не задан, выходной ассемблерный файл именуется как `file1.s`.

Компилятор `coolc` генерирует ассемблерный код MIPS. Так как не всем доступны основанные на MIPS компьютеры, программы Cool запускаются на симуляторе MIPS, который называется `spim`. Для запуска программы Cool выполните следующие команды:

```
% spim
(spim) load "file.s"
(spim) run
```

Чтобы запустить другую программу в рамках той же сессии `spim`, перед загрузкой нового ассемблерного файла нужно переинициализировать состояние симулятора:

```
(spim) reinit
```

Альтернативным и более быстрым способом запуска `spim` является запуск с указанием файла:

```
% spim -file file.s
```

Такой способ загружает файл, запускает программу и по завершении ее выполнения выходит из `spim`. Убедитесь, что `spim` запускается путем использования сценария `/home/compilers_course/bin/spim`. В некоторых системах может быть установлена другая версия `spim`, которая не может выполнять программы Cool. Простым способом обеспечения запуска правильной версии является создание псевдонима (alias) `spim` для `/home/compilers_course/bin/spim`. На веб-странице курса доступно справочное руководство по `spim`.

Ниже приведен полный пример компиляции и выполнения `/home/compilers_course/examples/list.cl`. Хотя эта программа довольно проста, она хорошо подходит для иллюстрации многих возможностей Cool.

```
% coolc list.cl
% spim -file list.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
54321
4321
321
21
1
COOL program successfully executed
%
```

3. Классы

Весь код на Cool организован в классы. Каждое определение класса должно полностью содержаться в одном файле, при этом в одном файле можно определять несколько классов. Определение класса имеет следующую форму:

```
class <type> [ inherits <type> ] {
  <feature_list>
};
```

Нотация `[...]` обозначает опциональную конструкцию. Все имена классов имеют глобальную область видимости. Имена классов начинаются с большой буквы. Классы не могут быть переопределены.

3.1. Элементы определения класса

Тело определения класса состоит из списка определений элементов. Элемент (feature) — это либо *атрибут* (*attribute*), либо *метод* (*method*). Атрибут класса `A` задает переменную, являющуюся частью состояния объектов класса `A`. Метод класса `A` — это процедура, которая может манипулировать переменными и объектами класса `A`.

Одним из важных понятий современных языков программирования является *сокрытие информации* (*information hiding*) — идея о том, что определенные аспекты реализации типа данных должны быть абстрагированы и скрыты от пользователя этого типа данных. Cool обеспечивает поддержку сокрытия информации путем реализации простого механизма: все атрибуты имеют локальную классу область видимости, а все методы — глобальную область видимости. Таким образом, единственный способ предоставления доступа к состоянию класса в Cool — использование методов.

Имена элементов класса должны начинаться со строчной буквы. В классе не могут быть определены ни методы с одинаковыми именами, ни атрибуты, но в то же время метод и атрибут одинаковые имена иметь могут.

В качестве простого примера определения атрибутов и методов приведем небольшой фрагмент из `list.cl`:

```
class Cons inherits List {
  xcar : Int;
  xcdr : List;

  isNil() : Bool { false };

  init(hd : Int, tl : List) : Cons {
    {
      xcar <- hd;
      xcdr <- tl;
      self;
    }
  }
  ...
};
```

В этом примере класс `Cons` имеет два атрибута `xcar` и `xcdr`, а также два метода `isNil` и `init`. Обратите внимание, что типы атрибутов, типы формальных параметров и типы возвращаемых значений методов явно объявлены программистом.

Имея объект `c` класса `Cons` и объект `l` класса `List`, можно установить поля `xcar` и `xcdr` путем использования метода `init`:

```
c.init(1, l)
```

Эта нотация является *объектно-ориентированной посылкой сообщений* (*object-oriented dispatch*). В нескольких различных классах могут быть определены несколько методов `init`, и при посылке сообщения происходит обращение к классу объекта `c` для определения того, какой именно метод `init` нужно вызывать. Так как классом объекта `c` является `Cons`, вызывается метод `init` именно этого класса. Внутри этого вызова переменные `xcar` и `xcdr` ссылаются на атрибуты объекта `c`. Специальная переменная `self` ссылается на объект, для которого в результате посылки сообщения был вызван метод, а это в приведенном примере — сам `c`.

Существует специальная форма `new C`, которая создает новый объект класса `C`. Объект может рассматриваться как запись, имеющая слоты для каждого из атрибутов класса, а также указатели на методы класса. Типичная посылка сообщения для метода `init`:

```
(new Cons).init(1, new Nil)
```

В этом примере создается новая cons-ячейка (cons cell) и «car» ячейки инициализируется единицей, а «cdr» — `new Nil`[1]. Cool не предоставляет механизм удаления объектов из памяти; этот язык использует *автоматическое управление памятью* (*automatic memory management*): объекты, которые более не могут использоваться программой, освобождаются сборщиком мусора.

Более подробно атрибуты обсуждаются в разделе 5, а методы — в разделе 6.

3.2. Наследование

Если определение класса имеет форму

```
class C inherits P { ... };
```

то класс `C` наследует элементы класса `P`. В этом случае класс `P` является *родительским* классом (*parent* class) класса `C`, а класс `C`, в свою очередь, является *дочерним* классом (*child* class) класса `P`.

Семантика `C inherits P` состоит в том, что класс `C` помимо своих собственных элементов содержит все элементы, определенные в `P`. В случае, когда и родительский, и дочерний класс оба определяют методы с одним и тем же именем, определение, данное в дочернем классе, имеет приоритет. Переопределение атрибутов не допускается. Более того, в целях типобезопасности имеются ограничения на то, как могут переопределяться методы (смотрите раздел 6).

Существует отдельный класс `Object`. Если определение класса не указывает родительский класс, то по умолчанию определяемый класс наследуется от `Object`. Класс может наследоваться только от одного класса; это уместно называется «одиночным наследованием» («single inheritance»)[2]. Отношение родитель-потомок классов определяет граф, не содержащий циклов. Для примера, если `C` унаследован от `P`, то `P` не может быть унаследован от `C`. Более того, если `C` унаследован от `P`, то `P` должен иметь определение класса где-то в программе. Так как Cool поддерживает только одиночное наследование, граф наследования образует дерево с классом `Object` в качестве корня.

В дополнение к `Object` Cool имеет четыре других *базовых класса* (*basic classes*): `Int`, `String`, `Bool` и `IO`. Базовые классы рассматриваются в разделе 8.

4. Типы

В Cool каждое имя класса является также и типом. Вдобавок существует тип `SELF_TYPE`, который используется в специальных случаях.

Объявление типа (*type declaration*) имеет форму `x:C`, где `x` является переменной, а `C` — типом. Каждая переменная должна иметь объявление типа в том месте, где она вводится, будь то `let`, `case` или формальный параметр метода. Типы всех атрибутов также должны быть объявлены.

Базовое правило типизации Cool состоит в том, что если метод или переменная ожидают значение типа `P`, то любое значение типа `C` может быть использовано вместо него при условии, что `P` является

[1] В этом примере подразумевается, что `Nil` является подтипом `List`

[2] Некоторые объектно-ориентированные языки позволяют классу наследоваться от нескольких классов, что также уместно называется «множественным наследованием» («multiple inheritance»)

предком C в иерархии классов. Другими словами, если C наследуется от P, напрямую или косвенно, то C может использоваться везде, где ожидается P.

Когда объект класса C может использоваться на месте объекта класса P, будем говорить, что C *согласуется* с (*conforms* to) P, или что $\texttt{C} \leqslant \texttt{P}$ (такую запись можно интерпретировать как то, что C находится ниже в дереве наследования). Как упоминалось ранее, согласование определяется в терминах графа наследования.

Определение 4.1 (Согласование)**.** Пусть A, C и P являются типами.

- $\texttt{A} \leqslant \texttt{A}$ для всех типов A
- если C унаследован от P, то $\texttt{C} \leqslant \texttt{P}$
- если $\texttt{A} \leqslant \texttt{C}$ и $\texttt{C} \leqslant \texttt{P}$, то $\texttt{A} \leqslant \texttt{P}$

Так как Object является корнем иерархии классов, то $\texttt{A} \leqslant \texttt{Object}$ для любого типа A.

4.1. SELF_TYPE

Тип SELF_TYPE используется для обращения к типу переменной self. Это полезно в классах, которые будут наследоваться другими классами, так как позволяет избежать задания фиксированного конечного типа при объявлении класса. Для примера рассмотрим программу:

```
class Silly {
  copy() : SELF_TYPE { self };
};

class Sally inherits Silly { };

class Main {
  x : Sally <- (new Sally).copy();

  main() : Sally { x };
};
```

Благодаря использованию SELF_TYPE в определении метода copy результат его выполнения имеет тот же тип, что и параметр self. Отсюда следует, что (new Sally).copy() имеет тип Sally, что согласуется с объявлением атрибута x.

Важно понимать, что значение SELF_TYPE не фиксировано, а зависит от класса, в котором используется. В общем SELF_TYPE может ссылаться на класс C, в котором он используется, или на любой другой класс, согласующийся с C. Когда полезным оказывается явное указание того, на что ссылается SELF_TYPE, имя класса C, в котором SELF_TYPE находится, будет указываться как индекс $\texttt{SELF\_TYPE}_{\texttt{C}}$. Эта нотация не является частью синтаксиса Cool — она используется лишь для того, чтобы явно показать, в каком именно классе возникло конкретное появление SELF_TYPE.

Из определения 4.1 следует, что $\texttt{SELF\_TYPE}_{\texttt{X}} \leqslant \texttt{SELF\_TYPE}_{\texttt{X}}$. Также существует специальное правило согласования для SELF_TYPE:

$$\texttt{SELF\_TYPE}_{\texttt{C}} \leqslant \texttt{P} \text{ если } \texttt{C} \leqslant \texttt{P}$$

И наконец, SELF_TYPE может использоваться в следующих случаях: new SELF_TYPE, как тип возвращаемого значения метода, как объявленный тип переменной let и как объявленный тип атрибута. Никакие другие использования SELF_TYPE не разрешены.

4.2. Проверка типов

Система типов Cool во время компиляции дает гарантии того, что выполнение программы не приведет к ошибкам типизации времени выполнения. Используя объявления типов, данных идентификаторам, система проверки типов выводит тип для каждого выражения в программе.

Очень важно различать тип, назначенный выражению системой проверки типов во время компиляции, который будем называть *статическим* типом (*static* type) выражения, и тип(ы), в который(ые) выражение может быть вычислено во время выполнения, их будем называть *динамическими* типами (*dynamic* types).

Различие между статическими и динамическими типами нужно, так как система проверки типов не может иметь во время компиляции всей информации о том, во что именно выражения будут вычислены во время выполнения. Поэтому, в общем случае, статический и динамический типы могут отличаться. Требуется лишь то, чтобы статические типы, выведенные системой проверки типов, были *корректны* (*sound*) по отношению к динамическим типам.

Определение 4.2 (Корректность)**.** Для любого выражения `e` пусть $\texttt{D}_\texttt{e}$ будет динамическим типом `e`, а $\texttt{S}_\texttt{e}$ — статическим типом, который был выведен системой проверки типов. Тогда система проверки типов *корректна* (*sound*), если для любого выражения `e` выполняется $\texttt{D}_\texttt{e} \leqslant \texttt{S}_\texttt{e}$.

То есть мы требуем, чтобы система проверки типов ошибалась в сторону переоценки (overestimating) типа выражения в тех случаях, когда идеальная точность невозможна. Такая система проверки типов никогда не примет программу, содержащую ошибки типизации. Однако, ценой, которую за это приходится платить, является то, что система проверки типов не будет принимать некоторые программы, которые на самом деле могут быть выполнены без ошибок.

5. Атрибуты

Определение атрибута имеет следующую форму:

```
<id> : <type> [ <- <expr> ];
```

Выражение тут — опциональная инициализация, которая выполняется при создании объекта. Статический тип выражения должен согласовываться с объявленным типом атрибута. Если инициализация не указана, то используется инициализация по умолчанию (смотрите ниже).

Когда создается новый объект класса, все унаследованные и локальные атрибуты должны быть проинициализированы. Унаследованные атрибуты инициализируются перед локальными в порядке наследования, начиная с атрибутов самого старшего предка. В рамках одного класса атрибуты инициализируются в порядке их появления в исходном тексте.

Атрибуты локальны классу, в котором они определены или унаследованы. Унаследованные атрибуты не могут быть переопределены.

5.1. Void

Все переменные в Cool инициализируются так, чтобы они содержали значения соответствующих типов. Специальное значение `void` является членом всех типов и используется в качестве значения инициализации по умолчанию для переменных, инициализация для которых пользователем указана не была. (`void` используется подобно `NULL` в C или `null` в Java; Cool не имеет ничего эквивалентного типу `void` в этих языках.) Следует обратить внимание на то, что для `void` в Cool нет имени; создать

значение `void` можно лишь путем объявления переменной некоторого класса, отличного от `Int`, `String` и `Bool`, без указания инициализации, а также сохранив результат цикла `while`.

Существует специальная форма `isvoid expr`, которая проверяет, является ли значение выражения значением `void` (смотрите раздел 7.11). Вдобавок значения `void` можно проверять на равенство. Значение `void` может быть передано в качестве аргумента, присвоено переменной и вообще быть использовано любым способом в контексте, где допустимо любое другое значение, за исключением того, что посылка сообщения или использование `case` на `void` сгенерируют ошибку времени выполнения.

Переменные базовых классов `Int`, `Bool` и `String` инициализируются специальным образом; смотрите раздел 8.

6. Методы

Определение метода имеет следующую форму:

```
<id>(<id> : <type>, ..., <id> : <type>) : <type> { <expr> };
```

В нем может быть нуль или более формальных параметров. Идентификаторы, используемые в списке формальных параметров, должны различаться. Тип тела метода должен согласовываться с объявленным возвращаемым типом. При вызове метода формальные параметры связываются с фактическими, и происходит вычисление выражения, итоговое значение которого и является результатом вызова метода. Формальный параметр скрывает любое определение атрибута с тем же именем.

В целях обеспечения типобезопасности существуют ограничения на переопределение унаследованных методов. Правило простое: если класс `C` наследует метод `f` у родительского класса `P`, то `C` может подменить унаследованное определение `f`, если число аргументов, типы формальных параметров и тип возвращаемого значения в точности одинаковы в обоих определениях.

Рассмотрим следующий пример для иллюстрации того, почему на переопределение унаследованных методов наложены ограничения:

```
class P {
  f() : Int { 1 };
};

class C inherits P {
  f() : String { "1" };
};
```

Пусть `p` будет объектом с динамическим типом `P`. Тогда

```
p.f() + 1
```

вполне закономерное выражение со значеним 2. Однако нельзя заменить `p` значением типа `C`, так как это приведет к сложению строки с числом. Таким образом, если допустить произвольное переопределение методов, подклассы не смогут просто расширять поведение своих предков, и большая часть преимуществ наследования, а также типобезопасности будет утеряна.

7. Выражения

Выражения являются самой большой синтаксической категорией в Cool.

7.1. Константы

Наиболее простыми выражениями являются константы. Булевыми константами являются `true` и `false`. Целочисленными константами являются беззнаковые последовательности цифр, такие как 0, 123 и 007. Строковыми константами являются последовательности символов, заключенные в двойные кавычки, такие как `"This is a string"`. Строковые константы могут иметь максимальную длину в 1024 символа. На них также налагаются другие ограничения; смотрите раздел 10.

Константы принадлежат базовым классам `Bool`, `Int` и `String`. Значением константы является объект соответствующего базового класса.

7.2. Идентификаторы

Имена локальных переменных, формальные параметры методов, `self` и атрибуты классов все являются выражениями. К идентификатору `self` можно обращаться, но ошибкой являются присваивание значения этому идентификатору либо его связывание `self` в `let`, `case` или как формального параметра. Также недопустимы атрибуты с именем `self`.

Локальные переменные и формальные параметры имеют лексическую область видимости. Атрибуты видимы повсюду в классе, в котором они определены или унаследованы, хотя они могут быть скрыты локальными объявлениями в выражениях.

7.3. Присваивание

Присваивание имеет форму

```
<id> <- <expr>
```

Статический тип выражения должен согласовываться с объявленным типом идентификатора. Значением присваивания является значение присваиваемого выражения. Статическим типом присваивания является статический тип `<expr>`.

7.4. Посылка сообщения

В Cool существует три формы посылки сообщения (то есть вызова методов). Эти формы отличаются лишь осуществлением выбора вызываемого метода. Наиболее часто используемой формой посылки сообщения является следующая:

```
<expr>.<id>(<expr>, ..., <expr>)
```

Рассмотрим посылку сообщения e_0`.f(`e_1`, ...,` e_n`)`. Для вычисления этого выражения сначала вычисляются аргументы слева направо, начиная с e_1 и по e_n. Затем вычисляется e_0 и запоминается его класс `C` (если e_0 оказывается `void`, генерируется ошибка времени выполнения). И наконец, вызывается метод `f` класса `C` со значением e_0, связанным в теле метода `f` с `self` и фактическими параметрами, связанными с формальными как обычно. Значением всего выражения является значение, возвращенное вызовом метода.

Проверка типов посылки сообщения включает в себя несколько шагов. Пусть e_0 имеет статический тип `A`. (Напомним, что этот тип необязательно является тем же, что и тип `C` выше. `A` является типом, выведенным системой проверки типов; `C` — это класс объекта, вычисленного во время выполнения, который потенциально может быть любым подклассом `A`.) Класс `A` должен иметь метод `f`, посылка сообщения и определение `f` должны иметь одинаковое число аргументов, а статический тип

i-того фактического параметра должен согласовываться с объявленным типом i-того формального параметра.

Если `f` имеет тип возвращаемого значения `B`, и `B` является именем класса, тогда статическим типом посылки сообщения является `B`. Иначе, если `f` имеет тип возвращаемого значения `SELF_TYPE`, тогда статическим типом посылки сообщения является `A`. Для понимания, почему это корректно, нужно обратить внимание на то, что параметр `self` метода `f` согласуется с типом `A`. Следовательно, так как `f` возвращает `SELF_TYPE`, можно заключить, что результат также должен согласовываться с `A`. Именно вывод правильных статических типов для выражений посылки сообщения обосновывает включение `SELF_TYPE` в систему типов Cool.

Другие формы посылки сообщения:

```
<id>(<expr>, ..., <expr>)
<expr>@<type>.<id>(<expr>, ..., <expr>)
```

Первая форма является сокращением для `self.<id>(<expr>, ..., <expr>)`.

Вторая форма предоставляет способ доступа к методам родительских классов, которые были скрыты путем их переопределения в дочерних классах. Вместо использования класса самого левого выражения для определения, какой именно метод нужно вызывать, используется метод указанного явно класса. Например, `e@B.f()` вызывает метод `f` класса `B` для объекта, который является значением `e`. В этой форме посылки сообщения статический тип выражения слева от «@» должен согласовываться с типом, заданным справа от «@».

7.5. Условная конструкция

Условная конструкция имеет форму

```
if <expr> then <expr> else <expr> fi
```

Семантика этой конструкции стандартна. Сначала вычисляется предикат. Если предикат истинен, вычисляется ветвь `then`, если же предикат ложен, вычисляется ветвь `else`. Значением условной конструкции является значение вычисленной ветви.

Предикат должен иметь статический тип `Bool`. Ветви могут иметь любой статический тип. Для специфицирования статического типа условной конструкции определим операцию $\sqcup$ (читается как «объединение», «join») на типах следующим образом. Пусть `A`, `B`, `D` будут любых типов кроме `SELF_TYPE`. Под *наименьшим типом* (*least type*) множества типов будем понимать наименьший элемент относительно отношения согласованности $\leqslant$.

$$\begin{aligned} \mathtt{A} \sqcup \mathtt{B} &= \text{наименьший тип } \mathtt{C} \text{ такой, что } \mathtt{A} \leqslant \mathtt{C} \text{ и } \mathtt{B} \leqslant \mathtt{C} \\ \mathtt{A} \sqcup \mathtt{A} &= \mathtt{A} \qquad \text{(идемпотентность)} \\ \mathtt{A} \sqcup \mathtt{B} &= \mathtt{B} \sqcup \mathtt{A} \quad \text{(коммутативность)} \\ \mathtt{SELF\_TYPE_D} \sqcup \mathtt{A} &= \mathtt{D} \sqcup \mathtt{A} \end{aligned}$$

Пусть `T` и `F` будут статическими типами ветвей условной конструкции. Тогда статическим типом условной конструкции будет $\mathtt{T} \sqcup \mathtt{F}$. (Можно думать об этом, как о движении вверх по дереву наследования к `Object` из обоих `T` и `F`, пока пути не встретятся.)

7.6. Конструкция цикла

Конструкция цикла имеет форму

`while <expr> loop <expr> pool`

Предикат вычисляется перед каждой итерацией цикла. Если предикат ложен, цикл завершается, и возвращается `void`. Если же предикат истинен, вычисляется тело цикла, и процесс повторяется.

Предикат должен иметь статический тип `Bool`. Тело может иметь любой статический тип. Статическим типом выражения цикла является `Object`.

7.7. Блоки

Блоки имеют форму

`{ <expr>; ...; <expr>; }`

Выражения вычисляются по порядку слева направо. Каждый блок имеет по крайней мере одно выражение; значением блока является значение последнего выражения. Выражения блока могут иметь любой статический тип. Статическим типом блока является статический тип последнего выражения.

Источником замешательства в Cool может быть использование точки с запятой («;»). Точки с запятой используются как завершающие выражения символы в списках выражений (например, в синтаксисе блока выше), а не как разделители выражений. Точки с запятой также завершают другие конструкции Cool, смотрите раздел 11 для подробной информации.

7.8. Let

Выражение `let` имеет форму

$$\texttt{let <id}_1\texttt{> : <type}_1\texttt{> [<- <expr}_1\texttt{>], ..., <id}_n\texttt{> : <type}_n\texttt{> [<- <expr}_n\texttt{>] in <expr>}$$

Опциональные выражения — это *инициализации* (*initialization*); последнее выражение — это *тело* (*body*) конструкции `let`. `let` вычисляется следующим образом. Сначала вычисляется $\texttt{<expr}_1\texttt{>}$, и результат связывается с $\texttt{<id}_1\texttt{>}$, затем вычисляется $\texttt{<expr}_2\texttt{>}$, и результат связывается с $\texttt{<id}_2\texttt{>}$, и так далее, пока все переменные в `let` не будут проинициализированы. (Если инициализация для $\texttt{<id}_k\texttt{>}$ не указана, то используется инициализация по умолчанию для типа $\texttt{<type}_k\texttt{>}$.) Затем вычисляется тело `let`, значение которого и будет значением всей конструкции `let`.

Идентификаторы $\texttt{<id}_1\texttt{>}$, ..., $\texttt{<id}_n\texttt{>}$ видимы в теле `let`. Более того, идентификаторы $\texttt{<id}_1\texttt{>}$, ..., $\texttt{<id}_k\texttt{>}$ видимы в инициализации $\texttt{id}_m$ для любого $\texttt{m} > \texttt{k}$.

Если идентификатор определен в `let` несколько раз, более позднее связывание скрывает более ранние. Идентификаторы, вводимые `let`, также скрывают во вложенных областях видимости любые ранее сделанные определения с теми же именами. Каждое выражение `let` должно вводить по крайней мере один идентификатор.

Тип выражения инициализации должен согласовываться с объявленным типом идентификатора. Типом `let` является тип ее тела.

Выражение `<expr>` в `let` простирается так далеко (охватывает столько токенов), сколько позволяет грамматика.

7.9. Case

Выражение `case` имеет форму

```
case <expr0> of
  <id1> : <type1> => <expr1>;
  ...
  <idn> : <typen> => <exprn>;
esac
```

Выражение выбора предоставляет возможность анализа типа объекта во время выполнения. Сначала вычисляется $\texttt{expr}_0$, и запоминается его динамический тип `C` (если $\texttt{expr}_0$ вычисляется в `void`, генерируется ошибка времени выполнения). Затем из всех ветвей выбирается ветвь с таким наименьшим типом $\texttt{type}_k$, что $\texttt{C} \leqslant \texttt{type}_k$. После этого идентификатор $\texttt{<id}_k\texttt{>}$ связывается со значением $\texttt{<expr}_0\texttt{>}$, и вычисляется выражение $\texttt{<expr}_k\texttt{>}$. Результатом выполнения `case` является значение $\texttt{<expr}_k\texttt{>}$. Если для вычисления ни одна ветвь выбрана быть не может, генерируется ошибка времени выполнения. Каждое выражение `case` должно иметь по крайней мере одну ветвь.

Для каждой ветви пусть $\texttt{T}_i$ будет статическим типом $\texttt{expr}_i$. Статическим типом выражения `case` является $\sqcup_{1 \leqslant i \leqslant n} \texttt{T}_i$. Идентификатор `id`, вводимый ветвью `case`, скрывает любое ранее сделанное в объемлющей области видимости определение переменной или атрибута для `id`.

Выражение `case` не имеет специальной конструкции для ветви «по умолчанию». Подобного эффекта можно достигнуть путем добавления следующей ветви

```
x : Object => ...
```

так как любой тип согласуется с `Object`.

Выражение `case` предоставляет способ использования явной проверки типа во время выполнения в случае, когда статические типы, выведенные системой проверки типов, слишком общие. Типовой является такая ситуация, когда программист пишет выражение `e`, и система проверки типов выводит, что `e` имеет статический тип `P`, однако программист знает, что на самом деле динамическим типом `e` всегда является `C` для некоторого $\texttt{C} \leqslant \texttt{P}$. Это знание может быть «реализовано» путем использования выражения выбора:

```
case e of x : C => ... esac
```

В этой ветви переменная `x` связывается со значением `e` и при этом имеет более конкретный статический тип `C`.

7.10. New

Выражение `new` имеет форму

```
new <type>
```

Его значением является новый объект соответствующего класса. Если типом является `SELF_TYPE`, то значением является новый объект класса, объектом которого в текущей области видимости является `self`. Статическим типом выражения является `<type>`.

7.11. Isvoid

Выражение

`isvoid <expr>`

вычисляется в `true`, если `<expr>` есть `void`, и в `false`, если `<expr>` не `void`.

7.12. Арифметические операции и операции сравнения

В Cool есть четыре бинарные математические операции: `+`, `-`, `*` и `/`. Синтаксис следующий:

$\texttt{<expr}_1\texttt{> <op> <expr}_2\texttt{>}$

При вычислении такого выражения сначала вычисляется $\texttt{<expr}_1\texttt{>}$, затем $\texttt{<expr}_2\texttt{>}$. Результатом операции является результат вычисления выражения.

Статическими типами двух подвыражений должен быть `Int`. Статический тип всего выражения также `Int`. В Cool имеется только целочисленное деление.

В Cool есть три операции сравнения: `<`, `<=`, `=`. Для `<` и `<=` правила в точности такие же, как и для бинарных арифметических операций, за исключением того, что результат имеет тип `Bool`. Сравнение же `=` является специальным случаем. Если одно из $\texttt{expr}_1$ или $\texttt{expr}_2$ имеет статический тип `Int`, `Bool` или `String`, то второе должно иметь такой же статический тип. Все остальные типы, включая и `SELF_TYPE`, могут сравниваться совершенно свободно. На небазовых (non-basic) объектах проверка на равенство просто проверяет равенство указателей (то есть, одинаковы ли адреса объектов в памяти). Проверка на равенство также определена для значения `void`.

В принципе нет ничего плохого в том, чтобы разрешить сравнение на равенство для, скажем, `Bool` и `Int`. Однако, такая проверка должна всегда быть ложной и почти наверняка указывает на программную ошибку какого-либо рода. Правила проверки типов Cool отлавливают такие ошибки на этапе компиляции вместо откладывания этого до времени выполнения.

И, наконец, есть один арифметический и один логический унарные операторы. Выражение `~<expr>` есть побитовое отрицание (дополнение) `<expr>`. Выражение `<expr>` должно иметь статический тип `Int`, и все выражение также имеет статический тип `Int`. Выражение `not <expr>` — булево отрицание (дополнение) `<expr>`. Выражение `<expr>` должно иметь статический тип `Bool`, и все выражение также имеет статический тип `Bool`.

8. Базовые классы

8.1. Object

Класс `Object` является корнем графа наследования. В нем определены методы со следующими объявлениями:

```
abort() : Object
type_name() : String
copy() : SELF_TYPE
```

Метод `abort` останавливает выполнение программы и выдает сообщение об ошибке. Метод `type_name` возвращает строку, содержащую имя класса объекта. Метод `copy` создает *поверхностную* (*shallow*) копию объекта[3].

[3]Поверхностное копирование объекта `a` осуществляет копирование самого `a`, но не осуществляет рекурсивное копирование объектов, на которые `a` ссылается.

8.2. IO

Класс `IO` предоставляет следующие методы для осуществления простых операций ввода и вывода:

```
out_string(x : String) : SELF_TYPE
out_int(x : Int) : SELF_TYPE
in_string() : String
in_int() : Int
```

Методы `out_string` и `out_int` печатают свой аргумент на стандартный вывод и возвращают свой параметр `self`. Метод `in_string` считывает строку со стандартного ввода до символа новой строки (невключительно). Метод `in_int` считывает целое число, которое может предваряться пробельными символами. Все символы, следующие за целым числом до последующего символа новой строки (включительно), методом `in_int` отбрасываются.

Класс может использовать методы класса `IO` путем наследования от него. Переопределение класса `IO` является ошибкой.

8.3. Int

Класс `Int` предоставляет класс целых чисел. У `Int` не существует специфичных для него методов. Инициализацией по умолчанию для переменных типа `Int` является `0` (не `void`). Переопределение класса `Int` или наследование от него является ошибкой.

8.4. String

Класс `String` предоставляет класс строк. Определены следующие методы:

```
length() : Int
concat(s : String) : String
substr(i : Int, l : Int) : String
```

Метод `length` возвращает длину параметра `self`. Метод `concat` возвращает строку, сформированную путем конкатенации `self` и `s`. Метод `substr` возвращает подстроку своего параметра `self` начиная с позиции `i` длиной `l`; позиции в строках нумеруются начиная с нуля. Если указанная подстрока выходит за пределы исходной строки, генерируется ошибка времени выполнения.

Инициализацией по умолчанию для переменных типа `String` является пустая строка `""` (не `void`). Переопределение класса `String` или наследование от него является ошибкой.

8.5. Bool

Класс `Bool` предоставляет значения `true` и `false`. Инициализицией по умолчанию для переменных типа `Bool` является `false` (не `void`). Переопределение класса `Bool` или наследование от него является ошибкой.

9. Класс Main

Каждая программа должна иметь класс `Main`. Более того, класс `Main` должен иметь метод `main`, не принимающий формальных параметров, который должен быть определен именно в классе `Main`

(не быть унаследованным от другого класса). Программа выполняется путем вычисления `(new Main).main()`.

Остальные разделы этого руководства предоставляют более формальное определение Cool. Это четыре раздела, охватывающие его лексическую структуру (раздел 10), грамматику (раздел 11), правила типизации (раздел 12) и операционную семантику (раздел 13).

10. Лексическая структура

Лексическими единицами Cool являются целые числа, идентификаторы типов, идентификаторы объектов, специальная нотация, строки, ключевые слова и пробельные символы.

10.1. Целые, идентификаторы и специальная нотация

Целые числа — непустые последовательности цифр 0-9. Идентификаторы — отличные от ключевых слов последовательности, состоящие из букв, цифр и знака подчеркивания. Идентификаторы типов начинаются с заглавной буквы; идентификаторы объектов — со строчной. Также существуют два отдельных идентификатора, `self` и `SELF_TYPE`, которые рассматриваются Cool специальным образом, но не как ключевые слова. Специальные синтаксические символы (например, скобки, оператор присваивания и так далее) перечислены на рисунке 1.

10.2. Строки

Строки заключаются в двойные кавычки `"..."`. Внутри строки последовательность `'\c'` обозначает символ `'c'` за следующими исключениями:

- `\b` возврат на шаг (backspace)
- `\t` табуляция (tab)
- `\n` перевод строки (newline)
- `\f` смена страницы (formfeed)

Неэкранированный символ новой строки не может находиться в строке:

```
"This \
is OK"
"This is not
OK"
```

Строка не может содержать EOF. Строка не может содержать нулевой символ (\0). Любой другой символ может быть помещен в строку. Строки не могут пересекать границы между файлами (то есть каждая строка должна полностью находиться в одном файле).

10.3. Комментарии

В Cool есть две формы комментариев. Любая последовательность символов, находящаяся после двух дефисов «--» и до последующего символа новой строки (или EOF, если последующего символа новой строки нет), рассматривается как комментарий. Комментарии также могут быть написаны путем заключения текста в `(*...*)`. При использовании последней формы комментарии могут быть вложенными. Комментарии также не могут пересекать границы между файлами.

10.4. Ключевые слова

Ключевые слова Cool: `class`, `else`, `false`, `fi`, `if`, `in`, `inherits`, `isvoid`, `let`, `loop`, `pool`, `then`, `while`, `case`, `esac`, `new`, `of`, `not`, `true`. За исключением констант `true` и `false` ключевые слова нечувствительны к регистру. Для соответствия правилам для остальных объектов первые буквы `true` и `false` должны быть строчными; остальные буквы могут быть в любом регистре.

10.5. Пробельные символы

Пробельные символы (white space) — любая последовательность следующих символов: пробел (blank, ASCII 32), `\n` (перевод строки, newline, ASCII 10), `\f` (смена страницы, form feed, ASCII 12), `\r` (возврат каретки, carriage return, ASCII 13), `\t` (табуляция, tab, ASCII 9), `\v` (вертикальная табуляция, vertical tab, ASCII 11).

11. Синтаксис Cool

Рисунок 1 предоставляет спецификацию синтаксиса Cool. Эта спецификация не является чистой нотацией Бэкуса-Наура (Backus-Naur Form, BNF): для удобства здесь также используется нотация регулярных выражений. А именно: A^* означает нуль или более повторений A; A^+ означает одно или более повторений A. Элементы в квадратных скобках $[\ldots]$ являются опциональными. Двойные квадратные скобки $[\![]\!]$ не являются частью синтаксиса Cool: они используются в грамматике в качестве метасимвола для объединения грамматических символов (например, $a[\![bc]\!]^+$ означает, что за a следует одна или более пар bc).

11.1. Приоритет

Приоритеты инфиксных бинарных и префиксных унарных операций, перечисленные от большего к меньшему, даны в следующем списке:

```
.
@
~
isvoid
* /
+ -
<= < =
not
<-
```

Все бинарные операции являются левоассоциативными, за исключением операции присваивания, которая правоассоциативна, и трех операций сравнения, которые не имеют ассоциативности.

12. Правила типизации

Этот раздел формально определяет правила типизации Cool. Правила типизации определяют тип каждого выражения Cool в заданном контексте. Таким контекстом является *окружение типизации* (*type environment*), которое описывает тип каждого несвязанного идентификатора выражения. Окружение типизации описано в подразделе 12.1. Подраздел 12.2 предоставляет правила типизации.

$$
\begin{array}{rcl}
program & ::= & [\![class;]\!]^+ \\
class & ::= & \textbf{class TYPE [inherits TYPE] \{ } [\![feature;]\!]^* \textbf{ \}} \\
feature & ::= & \text{ID([} formal [\![, formal]\!]^* \text{]) : TYPE \{ } expr \text{ \}} \\
 & | & \text{ID : TYPE [<- } expr \text{]} \\
formal & ::= & \text{ID : TYPE} \\
expr & ::= & \text{ID <- } expr \\
 & | & expr\text{[@TYPE].ID([} expr [\![, expr]\!]^* \text{])} \\
 & | & \text{ID([} expr [\![, expr]\!]^* \text{])} \\
 & | & \textbf{if } expr \textbf{ then } expr \textbf{ else } expr \textbf{ fi} \\
 & | & \textbf{while } expr \textbf{ loop } expr \textbf{ pool} \\
 & | & \{ [\![expr;]\!]^+ \} \\
 & | & \textbf{let ID : TYPE [<- } expr \text{] [\![, ID : TYPE [<- } expr \text{]]\!]}^* \textbf{ in } expr \\
 & | & \textbf{case } expr \textbf{ of } [\![\text{ ID : TYPE => } expr;]\!]^+ \textbf{ esac} \\
 & | & \textbf{new TYPE} \\
 & | & \textbf{isvoid } expr \\
 & | & expr \text{ + } expr \\
 & | & expr \text{ - } expr \\
 & | & expr \text{ * } expr \\
 & | & expr \text{ / } expr \\
 & | & \text{~}expr \\
 & | & expr \text{ < } expr \\
 & | & expr \text{ <= } expr \\
 & | & expr \text{ = } expr \\
 & | & \textbf{not } expr \\
 & | & (expr) \\
 & | & \text{ID} \\
 & | & \text{integer} \\
 & | & \text{string} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

Рис. 1. Синтаксис Cool

12.1. Окружения типизации

В качестве первого приближения проверка типов в Cool может рассматриваться как восходящий алгоритм (bottom-up algorithm): тип выражения e вычисляется по (ранее вычисленным) типам подвыражений e. Например, целое 1 имеет тип `Int`; в этом случае нет никаких подвыражений. В другом примере, если e_n имеет тип `X`, то выражение $\{ e_1; \ldots; e_n \}$ также имеет тип `X`.

Сложность возникает в случае, когда выражение `v` является идентификатором объекта. Не существует возможности сказать, каков тип `v` при использовании строго восходящего алгоритма: нам нужно знать тип, объявленный для `v` в бóльшем выражении. В правильной программе на Cool такие объявления должны существовать для всех идентификаторов объектов.

Для сбора информации о типах идентификаторов мы используем *окружение типизации* (*type environment*). Окружение состоит из трех частей: окружения методов M, окружения объектов O и имени текущего класса, в котором находится выражение. Окружения методов и объектов оба являются функциями (также называемыми *отображениями*, *mappings*). Окружение объектов — это функция

$$O(v) = T$$

которая назначает тип T идентификатору объекта v. Окружение методов является более сложным; это функция следующего вида:

$$M(C, f) = (T_1, \ldots, T_{n-1}, T_n)$$

где C — имя класса (тип), f — имя метода, а t_1, $\ldots$, t_n — типы. Кортеж типов — это *сигнатура* (*signature*) метода. Интерпретация сигнатуры состоит в том, что метод f класса C имеет формальные параметры типов (t_1, $\ldots$, t_{n-1}) — в указанном порядке — и тип возвращаемого значения t_n.

Два отображения, а не одно, нужны по причине того, что имена объектов и имена методов не конфликтуют — то есть свободно могут существовать метод и идентификатор объекта с одинаковыми именами.

Третий компонент окружения типизации — это имя текущего класса, которое необходимо для правил типизации, включающих `SELF_TYPE`.

Для каждого выражения e проверка типов осуществляется в окружении типизации; для подвыражений e проверка типов может осуществляться в том же или, если e вводит новый идентификатор объекта, в модифицированном окружении. В качестве примера рассмотрим выражение

```
let c : Int <- 33 in
  ...
```

Выражение `let` вводит новую переменную `c` типа `Int`. Пусть O будет объектным компонентом окружения типизации для `let`. Тогда проверка типов для тела `let` осуществляется в объектном окружении типизации

$$O[\texttt{Int}/\texttt{c}]$$

где нотация $O[T/c]$ определяется следующим образом:

$$O[T/c](c) = T$$
$$O[T/c](d) = O(d) \quad \text{если } d \neq c$$

12.2. Правила проверки типов

Обобщенная форма правила проверки типов такова:

$$\frac{\vdots}{O, M, C \vdash e : T}$$

Правило должно читаться следующим образом: в окружении типизации для объектов O, методов M и в пределах класса C выражение e имеет тип T. Точки над горизонтальной линией обозначают дополнительные утверждения о типах подвыражений выражения e. Эти дополнительные утверждения являются гипотезами правила: утверждение под горизонтальной линией истинно, если гипотезы выполняются. Символ $\vdash$ отделяет контекст (O, M, C) от утверждения $(e : T)$.

Правило для идентификаторов объектов очень просто: если окружение назначает идентификатору Id тип T, то Id имеет тип T.

$$\frac{O(Id) = T}{O, M, C \vdash Id : T} \quad [\textsc{Var}]$$

Правило для присваивания переменной сложнее:

$$\frac{\begin{array}{l} O(Id) = T \\ O, M, C \vdash e_1 : T' \\ T' \leqslant T \end{array}}{O, M, C \vdash Id \leftarrow e_1 : T'} \quad [\textsc{Assign}]$$

Обратите внимание, что это правило типизации — так же, как и остальные — использует отношение согласованности $\leqslant$ (смотрите подраздел 3.2). Правило утверждает, что присваиваемое выражение e_1 должно иметь тип T', который согласуется с типом T идентификатора Id в окружении типизации. Типом всего выражения является T'.

Правила типизации для констант просты:

$$\frac{}{O, M, C \vdash \texttt{true} : \texttt{Bool}} \quad [\textsc{True}]$$

$$\frac{}{O, M, C \vdash \texttt{false} : \texttt{Bool}} \quad [\textsc{False}]$$

$$\frac{i \text{ является целочисленной константой}}{O, M, C \vdash i : \texttt{Int}} \quad [\textsc{Int}]$$

$$\frac{s \text{ является строковой константой}}{O, M, C \vdash s : \texttt{String}} \quad [\textsc{String}]$$

Для `new` имеется два случая, один для `new SELF_TYPE` и второй для любого другого типа:

$$\frac{T' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{если } T = \texttt{SELF\_TYPE} \\ T & \text{иначе} \end{cases}}{O, M, C \vdash \texttt{new } T : T'} \quad [\textsc{New}]$$

Выражения посылки сообщения наиболее сложны для проверки типов:

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T'_0 = \begin{cases} C & \text{если } T_0 = \texttt{SELF\_TYPE}_C \\ T_0 & \text{иначе} \end{cases} \\ M(T'_0, f) = (T'_1, \ldots, T'_n, T'_{n+1}) \\ T_i \leqslant T'_i \quad 1 \leqslant i \leqslant n \\ T_{n+1} = \begin{cases} T_0 & \text{если } T'_{n+1} = \texttt{SELF\_TYPE} \\ T'_{n+1} & \text{иначе} \end{cases} \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}} \quad [\textsc{Dispatch}]$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0 \leqslant T \\ M(T, f) = (T'_1, \ldots, T'_n, T'_{n+1}) \\ T_i \leqslant T'_i \quad 1 \leqslant i \leqslant n \\ T_{n+1} = \begin{cases} T_0 & \text{если } T'_{n+1} = \texttt{SELF\_TYPE} \\ T'_{n+1} & \text{иначе} \end{cases} \end{array}}{O, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}} \quad [\textsc{StaticDispatch}]$$

Для того, чтобы осуществить проверку типов посылки сообщения, сначала должно быть проверено каждое из подвыражений. Тип T_0 выражения e_0 определяет, какое именно объявление метода f использовать. Типы аргументов посылки сообщения должны согласовываться с объявленными типами аргументов. Обратите внимание, что тип результата посылки сообщения — это либо объявленный тип возвращаемого значения, либо T_0 в случае, если тип возвращаемого значения объявлен как `SELF_TYPE`. Единственное отличие проверки типов статической посылки сообщения заключается в том, что класс T метода f указан в самой посылке сообщения, и тип T_0 должен согласовываться с T.

Правила проверки типов для выражений `if` и `{...}` довольно очевидны. Определение операции $\sqcup$ приведено в подразделе 7.5.

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : \texttt{Bool} \\ O, M, C \vdash e_2 : T_2 \\ O, M, C \vdash e_3 : T_3 \end{array}}{O, M, C \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi} : T_2 \sqcup T_3} \quad [\textsc{If}]$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ \vdots \\ O, M, C \vdash e_n : T_n \end{array}}{O, M, C \vdash \{e_1; e_2; \ldots; e_n;\} : T_n} \quad [\textsc{Sequence}]$$

Правило для `let` имеет несколько интересных аспектов.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{если } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{иначе} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leqslant T_0' \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \texttt{let } x : T_0 \leftarrow e_1 \texttt{ in } e_2 : T_2} \quad [\textsc{Let-Init}]$$

Сначала в окружении без нового определения x осуществляется проверка типов для инициализации e_1. Таким образом, переменная x не может использоваться в e_1 за исключением случаев, когда она уже имеет определение в объемлющей области видимости. Затем в окружении O, расширенном типизацией $x : T_0'$, осуществляется проверка типов тела e_2. Обратите внимание, что типом x может быть `SELF_TYPE`.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{если } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{иначе} \end{cases} \\ O[T_0'/x], M, C \vdash e_1 : T_1 \end{array}}{O, M, C \vdash \texttt{let } x : T_0 \texttt{ in } e_1 : T_1} \quad [\textsc{Let-No-Init}]$$

Правило для `let` без инициализации просто опускает требование согласованности.

Мы рассмотрели правила типизации только для `let` с одной переменной. Типизация `let` с несколькими переменными

$$\texttt{let } x_1 : T_1[\leftarrow e_1], x_2 : T_2[\leftarrow e_2], \ldots, x_n : T_n[\leftarrow e_n] \texttt{ in } e$$

по определению такова же, что и типизация

$$\texttt{let } x_1 : T_1[\leftarrow e_1] \texttt{ in (let } x_2 : T_2[\leftarrow e_2], \ldots, x_n : T_n[\leftarrow e_n] \texttt{ in } e\texttt{)}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O[T_1/x_1], M, C \vdash e_1 : T_1' \\ \vdots \\ O[T_n/x_n], M, C \vdash e_n : T_n' \end{array}}{O, M, C \vdash \texttt{case } e_0 \texttt{ of } x_1 : T_1 \Rightarrow e_1; \ldots; x_n : T_n \Rightarrow e_n; \texttt{ esac} : \sqcup_{1 \leqslant i \leqslant n} T_i'} \quad [\textsc{Case}]$$

Проверка типов каждой ветви `case` осуществляется в окружении, где переменная x_i имеет тип T_i. Тип всего `case` является объединением типов его ветвей. Все переменные, объявленные в каждой из ветвей `case`, должны иметь различные типы.

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : \texttt{Bool} \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \texttt{while } e_1 \texttt{ loop } e_2 \texttt{ pool} : \texttt{Object}} \quad [\textsc{Loop}]$$

Предикат цикла должен иметь тип `Bool`; типом всего выражения цикла всегда является `Object`.

Проверка `isvoid` имеет тип `Bool`:

$$\frac{O, M, C \vdash e_1 : T_1}{O, M, C \vdash \texttt{isvoid } e1 : \texttt{Bool}} \quad [\textsc{Isvoid}]$$

За исключением правила для проверки на равенство, правила проверки типов для примитивных логических, арифметических и операций сравнения весьма просты:

$$\frac{O, M, C \vdash e_1 : \texttt{Bool}}{O, M, C \vdash \texttt{not } e_1 : \texttt{Bool}} \quad [\textsc{Not}]$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : \texttt{Int} \\ O, M, C \vdash e_2 : \texttt{Int} \\ op \in \{<, \leqslant\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : \texttt{Bool}} \quad [\textsc{Compare}]$$

$$\frac{O, M, C \vdash e_1 : \texttt{Int}}{O, M, C \vdash \texttt{~}e_1 : \texttt{Int}} \quad [\textsc{Neg}]$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : \texttt{Int} \\ O, M, C \vdash e_2 : \texttt{Int} \\ op \in \{+, -, *, /\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : \texttt{Int}} \quad [\textsc{Arith}]$$

Трудность с правилом для проверки на равенство состоит в том, что любые типы можно свободно сравнивать, за исключением `Int`, `String` и `Bool`, которые могут сравниваться только с объектами своего типа.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{\texttt{Int}, \texttt{String}, \texttt{Bool}\} \vee T_2 \in \{\texttt{Int}, \texttt{String}, \texttt{Bool}\} \Rightarrow T_1 = T_2 \end{array}}{O, M, C \vdash e_1 = e_2 : \texttt{Bool}} \quad [\textsc{Equal}]$$

Остались правила проверки типов для атрибутов и методов. Пусть для класса C окружение типизации объекта O_C дает типы всех атрибутов C (включая любые унаследованные атрибуты). Более формально, если x является атрибутом (унаследованным или нет) класса C, и объявление x есть $x : T$, то

$$O_C(x) = \begin{cases} \texttt{SELF\_TYPE}_C & \text{если } T = \texttt{SELF\_TYPE} \\ T & \text{иначе} \end{cases}$$

Окружение типизации методов M глобально по отношению ко всей программе и для каждого класса C определяет сигнатуры всех методов C (включая все унаследованные).

Два правила проверки типов определений атрибутов подобны правилам для `let`. Существенным различием является то, что атрибуты видимы внутри своих выражений инициализации. Обратите также внимание на то, что в выражениях инициализации связан параметр `self`.

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C[\texttt{SELF\_TYPE}_C/\texttt{self}], M, C \vdash e_1 : T_1 \\ T_1 \leqslant T_0 \end{array}}{O_C, M, C \vdash x : T_0 \leftarrow e_1;} \quad [\textsc{Attr-Init}]$$

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T;} \quad [\textsc{Attr-No-Init}]$$

Правило для проверки типов методов осуществляет проверку тела метода в окружении, где O_C расширено связываниями (bindings) для формальных параметров и `self`. Тип тела метода должен согласовываться с объявленным типом возвращаемого значения.

$$\frac{\begin{array}{l} M(C,f) = (T_1,\ldots,T_n,T_0) \\ O_C[\texttt{SELF\_TYPE}_C/\texttt{self}][T_1/x_1]\ldots[T_n/x_n], M, C \vdash e : T_0' \\ T_0' \leqslant \begin{cases} \texttt{SELF\_TYPE}_C & \text{если } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{иначе} \end{cases} \end{array}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \ \texttt{\{ } e \texttt{ \};}} \quad [\textsc{Method}]$$

13. Операционная семантика

Этот раздел содержит в основном формальное изложение операционной семантики языка Cool. Для каждого выражения Cool операционная семантика определяет, в какое значение это выражение должно вычисляться в указанном контексте. Контекст состоит из трех компонент: окружения, хранилища и self-объекта. Эти компоненты описаны в следующем подразделе. Подаздел 13.2 определяет синтаксис, используемый далее для записи объектов Cool, а подраздел 13.3 — синтаксис для определения классов.

Важно понимать, что формальная семантика — это лишь спецификация, она не описывает реализацию. Цель ее предоставления состоит в том, чтобы сделать понятными все детали вычисления выражений Cool. Но совсем другое дело, как именно эти вычисления реализуются.

13.1. Окружение и хранилище

Чтобы описать семантику Cool, нужно рассмотреть несколько новых концепций и ввести новую нотацию. *Окружение* (*environment*) — это отображение, которое ставит в соответствие идентификаторам переменных *местоположения* (*locations*). То есть, окружение для каждого данного идентификатора просто указывает адрес местоположения в памяти, по которому хранится значение, соответствующее этому идентификатору. Для каждого данного выражения окружение должно указывать местоположения, соответствующие всем идентификаторам, на которые это выражение ссылается. Например, для выражения $a+b$ нужно окружение, которое отображает в некоторые местоположения идентификаторы a и b. Для описания окружений будет использоваться следующий синтаксис:

$$E = [a : l_1, b : l_2]$$

Это окружение отображает a в местоположение l_1 и b в местоположение l_2.

Вторым компонентом контекста, используемого при вычислении выражений, является *хранилище* (*store*, memory), которое ставит в соответствие местоположениям значения, являющиеся объектами Cool. Таким образом, хранилище указывает, какое значение хранится в заданном местоположении памяти.

Для упрощения примеров предположим, что все значения являются целочисленными.

Хранилище записывается подобно окружению:

$$S = [l_1 \to 55, l_2 \to 77]$$

Это хранилище отображает местоположение l_1 в значение 55, а местоположение l_2 в значение 77.

Используя окружение и хранилище, можно найти значение идентификатора путем поиска сначала местоположения идентификатора в окружении, а затем поиска значения, соответствующего этому местоположению, в хранилище:

$$E(a) = l_1$$
$$S(l_1) = 55$$

Вместе окружение и хранилище определяют состояние выполнения на определенном этапе вычисления выражения Cool. Двойная «косвенность» (double indirection) из идентификаторов в местоположения, а затем в значения позволяет моделировать переменные. Рассмотрим пример присваивания значения 99 переменной a в окружении и хранилище, определенных выше. Присваивание переменной означает изменение значения, на которое она ссылается, но не ее местоположения. Поэтому для осуществления присваивания находим в окружении E местоположение a, а затем изменяем отображение полученного местоположения, ставя ему в соответствие новое значение и получая таким образом новое хранилище S'.

$$E(a) = l_1$$
$$S' = S[99/l_1]$$

Синтаксис $S[v/l]$ обозначает новое хранилище, которое идентично хранилищу S, за исключением того, что ставит в соответствие местоположению l значение v. Для всех остальных местоположений l', где $l' \neq l$, по-прежнему $S'(l') = S(l')$.

Таким образом, хранилище моделирует содержимое памяти компьютера во время выполнения программы. Присваивание переменной изменяет хранилище.

Есть и такие ситуации, когда изменяется не хранилище, а окружение. Рассмотрим следующий фрагмент программы на Cool:

```
let c : Int <- 33 in
  c
```

При вычислении этого выражения перед вычислением тела `let` нужно добавить в окружение новый идентификатор c. Если текущими окружением и хранилищем являются E и S, новое окружение E' и новое хранилище S' создаются следующим образом:

$$l_c = newloc(S)$$
$$E' = E[l_c/c]$$
$$S' = S[33/l_c]$$

Первым делом выделяется новое местоположение для переменной c. Местоположение должно быть новым в том смысле, что текущее хранилище не должно иметь для него соответствия. Функция $newloc()$, примененная к хранилищу, дает неиспользуемое местоположение в этом хранилище. Затем создается новое окружение E', которое ставит в соответствие идентификатору c местоположение l_c и, кроме этого, содержит также все соответствия из E для идентификаторов, отличных от c. Обратите внимание, что если c уже имеет соответствие в E, новое окружение E' скрывает старое соответствие. И наконец, нужно обновить хранилище, чтобы оно хранило соответствие указанного значения новому местоположению. В рассматриваемом случае местоположению l_c соответствует значение 33, являющееся начальным значением для c, указанным в выражении `let`.

Приведенные в этом подразделе примеры несколько упрощают окружения и хранилища Cool, так как просто целые числа не являются значениями этого языка. В Cool даже целые числа являются полноценными объектами.

13.2. Синтаксис объектов Cool

Каждое значение в Cool является объектом. Объекты содержат список именованных атрибутов, подобно записям в языке C. Вдобавок каждый объект принадлежит какому-либо классу. Для значений в Cool будем использовать следующий синтаксис:

$$v = X(a_1 = l_1, a_2 = l_2, \ldots, a_n = l_n)$$

Такую запись нужно читать следующим образом: значение v является членом класса X и содержит атрибуты $a_1, \ldots, a_n$, чьими местоположениями являются $l_1, \ldots, l_n$. То, что атрибуты имеют ассоциированные с ними местоположения, очевидно означает, что для каждого из атрибутов объекта резервируется некоторое место в памяти.

Для базовых объектов Cool (типов `Int`, `String` и `Bool`) используется специальная нотация, несколько отличная от вышеописанного синтаксиса. Базовым объектам также соответствуют классы, имеющие имена, но их атрибуты отличаются от атрибутов обычных классов, так как являются неизменяемыми. Поэтому при описании базовых объектов будет использоваться следующий синтаксис:

$$\texttt{Int}(5)$$
$$\texttt{Bool}(\texttt{true})$$
$$\texttt{String}(4, \texttt{"Cool"})$$

Для типов `Int` и `Bool` смысл записи очевиден. Тип `String` содержит две части: длину строки и саму строку ASCII символов.

13.3. Определения классов

В правилах, которые будут рассмотрены в следующем подразделе, нужен способ обращения к определениям атрибутов и методов классов. Предположим, есть следующее определение класса:

```
class B {
  s : String <- "Hello";
  g (y : String) : String {
    y.concat(s);
  };
  f (x : Int) : Int {
    x + 1
  };
};

class A inherits B {
  a : Int;
  b : B <- new B;
  f(x : Int) : Int {
    x + a
  };
};
```

С каждым определением класса связываются два отображения, называемых *class* и *implementation*. Отображение *class* используется для получения атрибутов определенного класса, вместе с типами и инициализацией этих атрибутов:

$$class(A) = (s : \texttt{String} \leftarrow \texttt{"Hello"}, a : \texttt{Int} \leftarrow 0, b : B \leftarrow \texttt{new } B)$$

Обратите внимание, что *class* для класса A кроме его собственных определений содержит все, что было унаследовано этим классом от B. Если и B наследует какие-либо атрибуты, они также будут присутствовать среди этой информации. Атрибуты перечисляются в том порядке, в котором наследуются классы, а в рамках каждого класса — в порядке появления атрибутов в исходном коде: сначала в том порядке, в каком они расположены в исходном коде, идут атрибуты наиболее старшего предка, затем идут атрибуты следующего наиболее старшего предка, и так далее до атрибутов, определенных в конкретном классе. Этот же порядок атрибутов используется при инициализации объектов.

Общая форма отображения *class*:

$$class(X) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n)$$

Обратите внимание, что каждый атрибут сопровождается выражением инициализации, даже если в программе на Cool они не указаны. Инициализацией по умолчанию для переменной или атрибута является значение по умолчанию их типа. Значениями по умолчанию для `Int` является 0, для `String` — `""`, для `Bool` — false, и для всех остальных типов — `void`[4]. Значение по умолчанию для типа T будет записываться как D_T.

Отображение *implementation* предоставляет информацию о методах класса. Для примера, приведенного ранее, *implementation* для A определено следующим образом:

$$implementation(A, f) = (x, x + a)$$
$$implementation(A, g) = (y, y.concat(s))$$

Общая форма этого отображения для класса X и метода m

$$implementation(X, m) = (x_1, x_2, \ldots, x_n, e_{body})$$

указывает, что метод m класса X имеет формальные параметры $x_1, \ldots, x_n$, а телом этого метода является выражение e_{body}.

13.4. Операционные правила

Рассмотрев окружения, хранилища, объекты и определения классов, можно перейти к операционной семантике Cool. Операционная семантика описывается набором правил, подобных правилам, использованным для проверки типов. Обобщенная форма правила такова:

$$\frac{\vdots}{so, S, E \vdash e : v, S'}$$

Правило должно читаться следующим образом: в контексте, где so — это self-объект, S — хранилище и E — окружение, выражение e вычисляется в объект v, а новым хранилищем становится S'.

[4]Небольшое замечание: В этом подразделе в выражениях используется имя `void`, но в самом языке программирования Cool для этого значения имени нет.

Точки над горизонтальной линией обозначают дополнительные утверждения о вычислении подвыражений выражения e.

Как видно, кроме окружения и хранилища контекст выполнения также содержит self-объект so. Self-объект — это просто объект, на который ссылается идентификатор `self` в том случае, если `self` встречается в выражении. Мы не помещаем `self` в окружение и хранилище, так как `self` не является переменной — ей нельзя ничего присвоить. Обратите также внимание на то, что правила задают новое хранилище после вычисления выражения. Новое хранилище содержит все изменения памяти, являющиеся побочными эффектами вычисления выражения e.

Оставшаяся часть этого подраздела рассматривает и кратко описывает различные операционные правила. Некоторые вещи в виде правил не описаны, они обсуждаются в конце подраздела.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ E(Id) = l_1 \\ S_3 = S_2[v_1/l_1] \end{array}}{so, S_1, E \vdash Id \leftarrow e_1 : v_1, S_3} \quad [\textsc{Assign}]$$

Присваивание сначала вычисляет выражение, находящееся с правой стороны, получая значение v_1. Это значение затем сохраняется в памяти по адресу, соответствующему идентификатору.

Правила для обращения к идентификатору, для `self` и констант весьма очевидны:

$$\frac{\begin{array}{l} E(Id) = l \\ S(l) = v \end{array}}{so, S, E \vdash Id : v, S} \quad [\textsc{Var}]$$

$$\frac{}{so, S, E \vdash \texttt{self} : so, S} \quad [\textsc{Self}]$$

$$\frac{}{so, S, E \vdash \texttt{true} : \texttt{Bool(true)}, S} \quad [\textsc{True}]$$

$$\frac{}{so, S, E \vdash \texttt{false} : \texttt{Bool(false)}, S} \quad [\textsc{False}]$$

$$\frac{i \text{ является целочисленной константой}}{so, S, E \vdash i : \texttt{Int}(i), S} \quad [\textsc{Int}]$$

$$\frac{\begin{array}{l} s \text{ является строковой константой} \\ l = length(s) \end{array}}{so, S, E \vdash s : \texttt{String}(l, s), S} \quad [\textsc{String}]$$

Выражение `new` сложнее, чем может показаться:

$$\frac{\begin{array}{l} T_0 = \begin{cases} X & \text{если } T = \texttt{SELF\_TYPE} \text{ и } so = X(\dots) \\ T & \text{иначе} \end{cases} \\ class(T_0) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n) \\ l_i = newloc(S_1) \text{ для } i = 1, \dots, n \text{ и всех } l_i, \text{ являющихся различными} \\ v_1 = T_0(a_1 = l_1, \dots, a_n = l_n) \\ S_2 = S_1[D_{T_1}/l_1, \dots, D_{T_n}/l_n] \\ v_1, S_2, [a_1 : l_1, \dots, a_n : l_n] \vdash \{a_1 \leftarrow e_1; \dots; a_n \leftarrow e_n;\} : v_2, S_3 \end{array}}{so, S_1, E \vdash \texttt{new } T : v_1, S_3} \quad [\textsc{New}]$$

Сложностью выражения `new` является инициализация атрибутов в правильном порядке. Также обратите внимание на то, что во время инициализации атрибуты связываются со значениями по умолчанию для соответствующих классов.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \\ so, S_{n+1}, E \vdash e_0 : v_0, S_{n+2} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \\ implementation(X, f) = (x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+2}) \text{ для } i = 1, \ldots, n \text{ и всех } l_{x_i} \text{ являющихся различными} \\ S_{n+3} = S_{n+2}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ v_0, S_{n+3}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+4} \end{array}}{so, S_1, E \vdash e_0.f(e_1, \ldots, e_n) : v_{n+1}, S_{n+4}} \quad [\textsc{Dispatch}]$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \\ so, S_{n+1}, E \vdash e_0 : v_0, S_{n+2} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \\ implementation(T, f) = (x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+2}) \text{ для } i = 1, \ldots, n \text{ и всех } l_{x_i} \text{ являющихся различными} \\ S_{n+3} = S_{n+2}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ v_0, S_{n+3}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+4} \end{array}}{so, S_1, E \vdash e_0@T.f(e_1, \ldots, e_n) : v_{n+1}, S_{n+4}} \quad [\textsc{StaticDispatch}]$$

Эти два правила для посылки сообщения достаточно очевидны. Сначала вычисляются и сохраняются аргументы. Затем вычисляется выражение, находящееся слева от «.». В случае обычной посылки сообщения класс результата вычисления этого выражения используется для определения метода, который нужно вызывать; иначе класс указывается в самой посылке сообщения.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \texttt{Bool(true)}, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \end{array}}{so, S_1, E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi} : v_2, S_3} \quad [\textsc{If-True}]$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \texttt{Bool(false)}, S_2 \\ so, S_2, E \vdash e_3 : v_3, S_3 \end{array}}{so, S_1, E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi} : v_3, S_3} \quad [\textsc{If-False}]$$

Также все достаточно очевидно для правил `if-then-else`. Обратите внимание, что значением предиката является объект `Bool`, а не просто булево значение.

$$\frac{\begin{array}{c} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \end{array}}{so, S_1, E \vdash \{e_1; e_2; \ldots; e_n;\} : v_n, S_{n+1}} \quad [\textsc{Sequence}]$$

Блоки вычисляются по порядку с первого выражения и до последнего. Результатом является результат последнего выражения.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ l_1 = newloc(S_2) \\ S_3 = S_2[v_1/l_1] \\ E' = E[l_1/Id] \\ so, S_3, E' \vdash e_2 : v_2, S_4 \end{array}}{so, S_1, E \vdash \texttt{let}\ \ Id : T_1 \leftarrow e_1\ \ \texttt{in}\ \ e_2 : v_2, S_4} \quad [\textsc{Let}]$$

Выражение `let` вычисляет выражение инициализации, присваивает результат переменной с новым местоположением, а затем вычисляет свое тело. (Если инициализация не указана, переменная инициализируется значением по умолчанию для типа T_1.)

Мы рассмотрели операционную семантику только для случая `let` с одной переменной. Семантика `let` с несколькими переменными

$$\texttt{let}\ \ x_1 : T_1 \leftarrow e_1, x_2 : T_2 \leftarrow e_2, \ldots, x_n : T_n \leftarrow e_n\ \ \texttt{in}\ \ e$$

по определению такова же, что и семантика

$$\texttt{let}\ \ x_1 : T_1 \leftarrow e_1\ \ \texttt{in (let}\ \ x_2 : T_2 \leftarrow e_2, \ldots, x_n : T_n \leftarrow e_n\ \ \texttt{in}\ \ e\texttt{)}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_0 : v_0, S_2 \\ v_0 = X(\ldots) \\ T_i = \text{ближайший предок } X \text{ среди } \{T_1, \ldots, T_n\} \\ l_0 = newloc(S_2) \\ S_3 = S_2[v_0/l_0] \\ E' = E[l_0/Id_i] \\ so, S_3, E' \vdash e_i : v_1, S_4 \end{array}}{so, S_1, E \vdash \texttt{case}\ \ e_0\ \ \texttt{of}\ \ Id_1 : T_1 \Rightarrow e_1; \ldots; Id_n : T_n \Rightarrow e_n;\ \ \texttt{esac}\ :\ \ v_1, S_4} \quad [\textsc{Case}]$$

Обратите внимание, что для выбора верной ветви в правиле для выражения `case` требуется, чтобы иерархия классов была доступна в каком-либо виде во время выполнения. В остальном это правило также очевидно.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \texttt{Bool(true)}, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash \texttt{while}\ \ e_1\ \ \texttt{loop}\ \ e_2\ \ \texttt{pool} : void, S_4 \end{array}}{so, S_1, E \vdash \texttt{while}\ \ e_1\ \ \texttt{loop}\ \ e_2\ \ \texttt{pool} : void, S_4} \quad [\textsc{Loop-True}]$$

$$\frac{so, S_1, E \vdash e_1 : \texttt{Bool(false)}, S_2}{so, S_1, E \vdash \texttt{while}\ \ e_1\ \ \texttt{loop}\ \ e_2\ \ \texttt{pool} : void, S_2} \quad [\textsc{Loop-False}]$$

Для `while` существуют два правила: одно на случай, когда предикат истинен, и второе на случай, когда предикат ложен. Оба случая весьма просты.

Два правила для `isvoid` также очевидны:

$$\frac{so, S_1, E \vdash e_1 : void, S_2}{so, S_1, E \vdash \texttt{isvoid}\ e_1 : \texttt{Bool(true)}, S_2} \quad \text{[Isvoid-True]}$$

$$\frac{so, S_1, E \vdash e_1 : X(\ldots), S_2}{so, S_1, E \vdash \texttt{isvoid}\ e_1 : \texttt{Bool(false)}, S_2} \quad \text{[Isvoid-Fasle]}$$

Оставшиеся правила для примитивных арифметических, логических и операций сравнения тоже очень просты (за исключением проверки на равенство).

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \texttt{Bool}(b), S_2 \\ v_1 = \texttt{Bool}(\neg b) \end{array}}{so, S_1, E \vdash \texttt{not}\ e_1 : v_1, S_2} \quad \text{[Not]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \texttt{Int}(i_1), S_2 \\ so, S_2, E \vdash e_2 : \texttt{Int}(i_2), S_3 \\ op \in \{\leqslant, <\} \\ v_1 = \begin{cases} \texttt{Bool(true)} & \text{если } i_1\ op\ i_2 \\ \texttt{Bool(false)} & \text{иначе} \end{cases} \end{array}}{so, S_1, E \vdash e_1\ op\ e_2 : v_1, S_3} \quad \text{[Comp]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \texttt{Int}(i_1), S_2 \\ v_1 = \texttt{Int}(-i_1) \end{array}}{so, S_1, E \vdash \tilde{}e_1 : v_1, S_2} \quad \text{[Neg]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \texttt{Int}(i_1), S_2 \\ so, S_2, E \vdash e_2 : \texttt{Int}(i_2), S_3 \\ op \in \{+, -, *, /\} \\ v_1 = \texttt{Int}(i_1\ op\ i_2) \end{array}}{so, S_1, E \vdash e_1\ op\ e_2 : v_1, S_3} \quad \text{[Arith]}$$

Числа `Int` в Cool являются 32-разрядными знаковыми целыми, хранящимися в дополнительном коде; арифметические операции определены соответствующим образом.

Обозначения и правила, приведенные выше, не являются достаточно выразительными для описания того, как осуществляется проверка объектов на равенство, а также как обрабатываются исключительные ситуации времени выполнения. Для объяснения этих вещей прибегнем к описанию на обычном языке.

В выражении $e_1 = e_2$ сначала вычисляется e_1, а затем — e_2. Сначала два объекта проверяются на равенство путем сравнения их указателей (адресов в памяти). Если они одинаковы, объекты равны. Значение `void` не равно никакому объекту за исключением себя самого. Если объекты имеют тип `String`, `Bool` или `Int`, то осуществляется сравнение их соответствующего содержимого.

Далее, операционные правила не указывают, что происходит в случае ошибок времени выполнения. При их возникновении выполнение программы прерывается. В следующем списке перечислены все возможные ошибки такого рода.

1) Посылка сообщения (статическая или динамическая) значению `void`.

2) Применение `case` к значению `void`.

3) Выполнение выражения выбора (`case`), при котором не выбирается ни одна из ветвей.

4) Деление на нуль.

5) Указанная подстрока выходит за пределы строки.

6) Переполнение кучи (heap overflow).

И наконец, вышеприведенные правила не описывают поведение при посылке сообщений, вызывающих примитивные методы, определенные в классах `Object`, `IO` и `String`. Описание этих методов дано в подразделах 8.1, 8.2 и 8.4.

14. Благодарности

Cool основан на языке Sather164, который в свою очередь основан на языке Sather. Некоторые части этого документа были позаимствованы из руководства по Sather164; в свою очередь части руководства по Sather164 основаны на документации по Sather, написанной Stephen M. Omohundro.

В разработку и реализацию Cool свой вклад внесли множество людей, включая Manuel Fähndrich, David Gay, Douglas Hauge, Megan Jacoby, Tendo Kayiira, Carleton Miyamoto и Michael Stoddart. Joe Darcy обновил Cool до его текущей версии.