Обзор языка Cool

Содержание

1. Введение

В данном курсе по компиляторам в качестве примера используется язык программирования Cool — учебный объектно-ориентированный язык программирования (*C*lassroom *O*bject-*O*riented *L*anguage). Язык Cool был спроектирован с учетом довольно уникального требования: компилятор для него должен быть реализуем за достаточно короткий срок путем приложения разумных усилий (это должно быть по силам студентам в рамках семестрового курса). Важность этого требования очевидна, так как Cool используется главным образом для обучения созданию компиляторов, и число компиляторов для этого языка в мире намного превышает количество программ, написанных на нем. Поэтому требование легкости написания компиляторов для Cool куда важнее требования легкости написания программ на этом языке.

Но одной легкости реализации языка для образовательных целей недостаточно, сосредотачиваясь лишь на ней, мы рискуем получить довольно бесполезный язык, реализация которого мало что даст студенту. Поэтому Cool проектировался также с прицелом на современные языки программирования: он включает такие возможности, как абстракция данных, статическая типизация, повторное использование кода через наследование, автоматическое управление памятью и многие другие.

Этот документ дает краткий обзор языка, рассматривая его на нескольких простых примерах. За полным описанием Cool обратитесь к документу «*Справочное руководство по языку Cool*».

2. Обзор проекта

Проектом, который вы будете реализовывать в рамках курса, является построение компилятора для языка Cool, который будет транслировать программы на Cool в язык ассемблера для MIPS, который гораздо проще языка ассемблера для архитектуры x86. Так как далеко не всем доступны MIPS-машины, в дистрибутиве Cool предоставляется симулятор MIPS, который позволит запускать полученные программы на вашем аппаратном обеспечении.

В каждом задании нужно реализовать одну из фаз компиляции: лексический анализ, синтаксический анализ, семантический анализ и генерацию кода. Следует заметить, что все эти фазы реализуются в Cool в виде совместимых между собой модулей: в дистрибутиве Cool поставляются раздельные и совместимые между собой эталонные реализации каждой из этих фаз, и при разработке своей реализации какой-то из компонент вы легко можете подменить своей компонентой одну из эталонных и получить работающий компилятор Cool. Это облегчает как разработку, так и тестирование: имея эталонные реализации фаз компиляции можно исключить влияние ошибок одной из фаз на последующие при тестировании.

3. Первая программа

Для того, чтобы вы могли компилировать и запускать примеры, приведенные далее, вам нужно скачать дистрибутив Cool для своей платформы с веб-страницы курса и установить его. Инструкции по установке и настройке также приведены на этой веб-странице.

Программа на Cool состоит из списка объявлений классов, каждое из которых заканчивается точкой с запятой. Класс в Cool объявляется путем использования ключевого слова `class`, за которым идут имя класса и его тело в фигурных скобках. Каждая программа на Cool должна содержать класс с именем `Main`, в котором объявлен метод `main`, не принимающий аргументов, с которого и начинается выполнение программы.

Тело класса состоит из списка объявлений элементов (атрибутов и методов), каждое из которых также оканчивается точкой с запятой. Тело метода заключается в фигурные скобки, в которых находится выражение тела. При вызове метода формальные параметры связываются с фактическими, а затем происходит вычисление этого выражения, значение которого и становится значением вызова метода. Обратите внимание, что для возврата значения нет специализированного ключевого слова наподобие `return`: значением вызова метода является значение выражения его тела, а практически все конструкции Cool (даже такие, как, например, `if` или `while`) являются выражениями и вычисляются в какие-либо значения.

В первом примере создадим простейший метод, возвращающий при вызове единицу. Отредактируйте файл `ex1.cl` так, чтобы он содержал следующий код:

```
class Main {
  main() {
    1
  };
};
```

Попробуем скомпилировать программу:

```
$ coolc ex1.cl
"ex1.cl", line 2: syntax error at or near '{'
Compilation halted due to lex and parse errors
```

Мы получили синтаксическую ошибку, так как каждый метод в Cool должен объявлять тип возвращаемого значения. Это объявление идет после списка аргументов. В нашем случае возвращается единица, а значит типом возвращаемого значения является `Int`:

```
class Main {
  main() : Int {
    1
  };
};
```

Скомпилируем исправленную программу:

```
$ coolc ex1.cl
```

На этот раз компилятор не выдал никаких ошибок, а в директории с программой появился файл `ex1.s`, содержащий ассемблерный код, соответствующий нашей программе. Выполним программу, воспользовавшись симулятором MIPS под названием `spim`:

```
$ spim -file ex1.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
COOL program successfully executed
```

Как видно, программа успешно выполнилась и завершилась.

4. Программа "Hello World!"

4.1. Атрибуты класса. Базовый класс IO

В предыдущем примере никакого взаимодействия с пользователем не осуществлялось; метод main просто возвращал в качестве своего значения единицу. В следующем же примере напишем классическую программу, выводящую на экран строку "Hello World!".

Для осуществления операций ввода/вывода в Cool есть базовый класс IO. Для его использования объявим в классе Main атрибут i с типом IO, а затем воспользуемся этим атрибутом для вывода строки, вызвав метод out_string. Для того, чтобы включить в тело метода несколько выражений, воспользуемся новой конструкцией — блоком выражений. Блок выражений содержит в фигурных скобках несколько подвыражений, каждое из которых заканчивается точкой с запятой. При вычислении блока все подвыражения вычисляется в порядке их нахождения в блоке, а значение последнего из них становится значением всего выражения блока. В нашем примере телом метода является блок выражений, первое из которых выводит на экран строку "Hello World!", а второе вычисляется в единицу, которая и будет значением блока, а следовательно и возвращаемым значением метода.

```
class Main {
  i : IO;
  main() : Int {
    {
      i.out_string("Hello World!\n");
      1;
    }
  };
};
```

Скомпилируем и попробуем выполнить программу:

```
$ coolc ex2.cl
$ spim -file ex2.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
ex2.cl:5: Dispatch to void.
```

В этот раз программа скомпилировалась без ошибок, но получена ошибка времени выполнения, говорящая о том, что в 5-й строке программы мы пытаемся отправить сообщение объекту, которого не существует. Действительно, мы забыли выделить память для объекта атрибута i: мы лишь объявили атрибут типа IO, но при этом не создали никакого объекта. Создадим новый объект типа IO и проинициализируем им атрибут i:

```
class Main {
  i : IO <- new IO;
  main() : Int {
    {
      i.out_string("Hello World!\n");
```

```
      1;
    }
  };
};
```

Повторно скомпилируем и выполним программу:

```
$ coolc ex2.cl
$ spim -file ex2.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
Hello World!
COOL program successfully executed
```

На этот раз все выполнилось правильно.

Поэкспериментируем с нашей программой для того, чтобы показать некоторые возможности Cool, а также получше освоиться с синтаксисом этого языка. Для начала отметим, что блок выражений совершенно не нужен в реализации программы `"Hello World!"`. Давайте уберем его, сделав телом метода `main` единственное выражение вывода строки:

```
class Main {
  i : IO <- new IO;
  main() : Int {
    i.out_string("Hello World!\n")
  };
};
```

В этой реализации есть одна проблема: теперь фактический тип тела метода не совпадает с объявленным типом возвращаемого значения. Убедимся, что это так:

```
$ coolc ex2.cl
ex2.cl:3: Inferred return type IO of method main does not conform to declared return type Int.
Compilation halted due to static semantic errors.
```

Сообщение об ошибке говорит, что выражение тела метода `main` имеет тип `IO`, но при этом тип возвращаемого значения этого метода объявлен как `Int`. Для устранения ошибки просто исправим этот объявленный тип на `IO`.

```
class Main {
  i : IO <- new IO;
  main() : IO {
    i.out_string("Hello World!\n")
  };
};
```

Теперь все должно скомпилироваться без проблем.

4.2. Класс Object

На самом деле в данном случае нам не нужно настолько конкретизировать тип возвращаемого значения, так как результат выполнения метода для дальнейших вычислений нигде не используется (программа завершает работу сразу после вывода строки). Поэтому можно просто объявить возвращаемый тип как `Object`. Этот класс является корнем иерархии классов в языке Cool, то есть любой класс является подклассом `Object`.

```
class Main {
  i : IO <- new IO;
  main() : Object {
    i.out_string("Hello World!\n")
  };
};
```

Более того, атрибут `i` в предыдущих примерах необязателен. Он используется лишь однажды, а поэтому легко можно создавать объект класса `IO` в самом методе `main`, где он сразу и используется:

```
class Main {
  main() : Object {
    (new IO).out_string("Hello World!\n")
  };
};
```

4.3. Наследование

Воспользоваться методами `IO` можно также унаследовав его поведение в нашем классе `Main`. Для этого нужно указать, что `Main` является подклассом `IO`.

```
class Main inherits IO {
  main() : Object {
    self.out_string("Hello World!\n")
  };
};
```

Теперь вдобавок ко своим собственным элементам класс `Main` содержит все атрибуты и методы класса `IO`, и вместо выделения памяти для объекта этого типа можно просто вызвать метод `out_string` у переменной `self`, которая содержит текущий объект (аналогом этой переменной в C++ и Java является переменная `this`).

На самом деле при посылке сообщения в данном примере использовать `self` необязательно: если при посылке сообщения объект не указывается, `self` используется по умолчанию.

```
class Main inherits IO {
  main() : Object {
    out_string("Hello World!\n")
  };
};
```

5. Программа вычисления факториала

В этом разделе мы разработаем программу вычисления факториала числа.

5.1. Ввод/вывод

Для ввода строки с клавиатуры базовый класс `IO` предоставляет метод `in_string`. Чтобы проиллюстрировать его использование, напишем программу, которая выводит на экран строку, введеную ранее пользователем.

```
class Main {
  main() : Object {
    (new IO).out_string((new IO).in_string().concat("\n"))
  };
};
```

Как обычно, программа содержит класс `Main`, в котором определен метод `main`. Нас не интересует возвращаемое значение этого метода, поэтому мы просто объявляем его возвращаемым типом `Object`. Метод `concat` класса `String` осуществляет конкатенацию строки, для которой он вызван, с аргументом, переданным методу. Проверим программу:

```
$ coolc ex3.cl -o ex3.s
$ spim -file ex3.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
abcde
abcde
COOL program successfully executed
```

После запуска программа ожидает ввода строки пользователем, после чего выводит на экран введенную строку.

Так как программа вычисления факториала работает с числами, нам нужен способ их ввода и вывода. Для этого воспользуемся библиотекой для осуществления преобразований между строками и числами, написанной на Cool и идущей вместе с ним. В качестве примера рассмотрим программу, принимающую в качестве ввода от пользователя число, а затем выводящую его же, увеличенное на единицу. Предоставим классу `Main` функциональность осуществления преобразований между строками и числами, унаследовав его от класса `A2I`. Преобразование из строки в число осуществляется методом `a2i`, а обратно — методом `i2a`.

```
class Main inherits A2I {
  main() : Object {
    (new IO).out_string(i2a(a2i((new IO).in_string()) + 1).concat("\n"))
  };
};
```

Попробуем скомпилировать программу.

```
$ coolc ex4.cl -o ex4.s
ex4.cl:1: Class Main inherits from an undefined class A2I.
Compilation halted due to static semantic errors
```

Компилятор указывает нам на то, что класс `A2I` не определен, и, действительно, этот класс компилятору предоставлен не был. Для того, чтобы скомпилировать программу, скопируем файл `atoi.cl`, содержащий определение класса `A2I`, из директории примеров Cool в текущую директорию и укажем его в командной строке компилятора.

```
$ cp /home/compilers_course/examples/atoi.cl ./
$ coolc ex4.cl atoi.cl -o ex4.s
$ spim -file ex4.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
42
43
COOL program successfully executed
```

После запуска программа ожидает ввода числа пользователем, после чего выводит это число, увеличенное на единицу.

5.2. Условная конструкция и конструкция цикла. Локальные переменные

Теперь написать программу вычисления факториала очень просто. Для этого добавим новый метод `fact`, рекурсивно вычисляющий факториал переданного числа:

```
class Main inherits A2I {
  main() : Object {
    (new IO).out_string(i2a(fact(a2i((new IO).in_string()))).concat("\n"))
  };

  fact(n : Int) : Int {
    if n < 2 then
      1
    else
      n * fact(n - 1)
    fi
  };
};
```

Метод `fact` принимает один параметр типа `Int` и возвращает значение этого же типа. Для вычисления факториала в теле используется конструкция `if-then-else`.

Проверим программу:

```
$ coolc fact.cl atoi.cl -o fact.s
$ spim -file fact.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
5
120
COOL program successfully executed
```

В образовательных целях перепишем функцию `fact` так, чтобы она вычисляла значение факториала итеративно. Для начала нам нужна локальная переменная-аккумулятор, которая будет накапливать результат вычисления факториала. Локальные переменные в Cool объявляются с помощью выражения `let`, которое состоит из двух частей. Первая содержит объявляемую переменную (переменные), а вторая является телом выражения `let`: выражением, в котором доступны объявленные переменные, и значение которого и будет значением всего выражения `let`.

Далее для вычисления факториала нам нужно выражение цикла `while`, которое также состоит из двух частей. Первая является предикатом, который вычисляется перед каждой итерацией цикла, и на основании значения которого принимается решение, выполнять ли следующую итерацию. Вторая часть — тело цикла — выражение, вычисляемое на каждой итерации. Очередная итерация выполняется, если значение предиката истинно. Если значение ложно, работа цикла прекращается. Значением выражения `while` является `void`.

```
class Main inherits A2I {
  main() : IO {
    (new IO).out_string(i2a(fact(a2i((new IO).in_string()))).concat("\n"))
  };

  fact(n : Int) : Int {
    let fact : Int <- 1 in
      {
        while not (n = 0) loop
          {
            fact <- fact * n;
            n <- n - 1;
          }
        pool;
        fact;
      }
  };
};
```

Обратите внимание, что переменная `fact` имеет то же имя, что и метод. Это не является проблемой, так как в Cool используются два отдельных пространства имен для методов и переменных, поэтому они не конфликтуют. Также стоит заметить, что при объявлении переменной `fact` было указано значение инициализации. Если этого не сделать, значением инициализации для переменной типа `Int` будет нуль.

Телом выражения `let` в примере является блок, состоящий из двух выражений: первое вычисляет значение факториала числа `n`, а второе предоставляет это значение, так что оно становится значением всего выражения `let`.

Обратите внимание, что оператор присваивания в Cool записывается как «`<-`». Оператор же «`=`» используется для сравнения двух значений. Если вы случайно перепутаете эти операторы и воспользуетесь в теле цикла оператором «`=`» вместо присваивания, переменная `n` не будет обновляться на каждой итерации, и вы получите зацикливающуюся программу.

Из остальных операций сравнения значений в Cool также определены «`<`» и «`<=`».

6. Реализация структуры данных списка

В качестве последнего примера реализуем программу работы со списками.

В первом приближении нашей программы просто создадим элементы списка вручную и выведем их на экран.

```
class Main inherits IO {
  main() : Object {
    let hello : String <- "Hello ",
        world : String <- "World!",
        newline : String <- "\n"
    in
      out_string(hello.concat(world.concat(newline)))
  };
};
```

Данный пример иллюстрирует объявление в `let` нескольких переменных: они просто разделяются запятыми. В программе объявляются три переменные-элемента списка, а затем для осуществления вывода этих элементов на экран вручную производится их конкатенация.

Скомпилируем и протестируем написанную программу.

```
$ coolc list1.cl -o list1.s
$ spim -file list1.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
Hello World!
COOL program successfully executed
```

6.1. Реализация класса списка

Теперь вместо того, чтобы раздельно объявлять элементы списка, а затем соединять их вручную, напишем абстракцию для списка, которая будет содержать функцию конкатенации всех своих элементов.

Структура данных списка будет реализовываться классом `List`. Список представим структурой данных из двух элементов: элемента списка, являющегося строкой, и оставшейся части списка. Для

создания списков нам понадобится функция инициализации, которая просто будет инциализировать переданными аргументами две компоненты списка. Кроме того, нам нужна функция, осуществляющая конкатенацию всех элементов списка, назовем ее `flatten`.

```
class List {
  item : String;
  next : List;

  init(i : String, n : List) : List {
    {
      item <- i;
      next <- n;
      self;
    }
  };

  flatten() : String {
    if isvoid next then
      item
    else
      item.concat(next.flatten())
    fi
  };
};

class Main inherits IO {
  main() : Object {
    let hello : String <- "Hello ",
        world : String <- "World!",
        newline : String <- "\n",
        nil : List,
        list : List <-
                 (new List).init(hello,
                   (new List).init(world,
                     (new List).init(newline, nil)))
    in
      out_string(list.flatten())
  };
};
```

Для начала обратите внимание, что в методе `init` возвращается сам проинициализированный объект. Это сделано для того, чтобы вызовы `init` можно было выстраивать в цепочку, создавая список следующим образом: `(new List).init("1", (new List).init("2", ...))`. Именно таким образом и инициализируется переменная `list`, создавая список из трех элементов `hello`, `world` и `newline`.

При создании списка нужен какой-то способ указания того, что атрибут `next` очередного элемента не указывает ни на какой подсписок, то есть данный элемент списка является последним (таким образом, нужен некоторый аналог нулевого указателя). Для этого в Cool используется специальное

значение `void`, являющееся членом всех типов и обозначающее отсутствие объекта. Это значение в Cool не имеет имени, получить его можно путем создания без инициализации переменной любого типа, отличного от `Bool`, `Int` и `String`, что мы и делаем, объявив переменную `nil`.

Как видно по данному примеру, очередная вводимая с помощью `let` переменная может ссылаться на переменные, введенные выше в этой же конструкции.

Рассмотрим реализацию функции `flatten`. Она не принимает аргументов и рекурсивно осуществляет конкатенацию всех элементов списка. Базовым случаем рекурсии является последний элемент списка, когда мы просто возвращаем хранящуюся в этом элементе строку. Для проверки того, является ли элемент последним, мы проверяем атрибут `next` на `void` с помощью специальной конструкции языка Cool `isvoid`, возвращающей `true`, если переданное ей значение является `void`. В случае же непоследнего элемента осуществляется конкатенация текущего элемента и значения, возвращаемого рекурсивным вызовом `flatten` для оставшейся части списка.

Скомпилируем и протестируем полученную программу.

```
$ coolc list2.cl -o list2.s
$ spim -file list2.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
Hello World!
COOL program successfully executed
```

6.2. Анализ типов объектов во время выполнения и конструкция `case`

Теперь попробуем обобщить нашу абстракцию так, чтобы список мог хранить произвольные объекты, а не только строки. Для этого изменим тип атрибута `item` на `Object` и соответствующим образом поменяем функцию `init`.

С функцией `flatten` дело обстоит сложнее, так как на выходе должна получаться строка, состоящая из всех элементов списка, которые теперь могут быть не только строками. Поэтому при прохождении списка нам нужен способ осуществления различных действий в зависимости от того, что именно (значение какого типа) хранится в текущем элементе.

Для выявления типа объекта во время выполнения в Cool существует специальная конструкция `case`, позволяющая осуществлять различные действия в зависимости от типа объекта. В выражении `case` указывается объект, тип которого будет анализироваться, и набор ветвей, каждая из которых завершается точкой с запятой. Каждая ветвь содержит тип, на основании которого принимается решение, будет ли выбрана данная ветвь, переменную этого типа, которая связывается с анализируемым `case` объектом в случае выбора данной ветви, и выражение, которое вычисляется в этом случае, и в котором видима вводимая ветвью переменнная. При анализе объекта `case` выбирает ту ветвь, тип которой наиболее соответствует типу анализируемого объекта. Если ни одна ветвь выбрана быть не может, потому что ни одна из ветвей не соответствует анализируемому объекту, происходит ошибка времени выполнения, приводящая к завершению выполнения программы.

```
class List inherits A2I {
  item : Object;
  next : List;
```

```
  init(i : Object, n : List) : List {
    {
      item <- i;
      next <- n;
      self;
    }
  };

  flatten() : String {
    let string : String <-
      case item of
        i : Int => i2a(i);
        s : String => s;
        o : Object => { abort(); ""; };
      esac
    in
      if isvoid next then
        string
      else
        string.concat(next.flatten())
      fi
  };
};

class Main inherits IO {
  main() : Object {
    let hello : String <- "Hello ",
        world : String <- "World!",
        i : Int <- 42,
        newline : String <- "\n",
        nil : List,
        list : List <-
                 (new List).init(hello,
                   (new List).init(world,
                     (new List).init(42,
                       (new List).init(newline, nil))))
    in
      out_string(list.flatten())
  };
};
```

Рассмотрим реализацию метода `flatten`, в котором произошли основные изменения. Теперь телом метода является выражение `let`, вводящее новую переменную `string`, которая инициализируется строковым представлением текущего элемента списка, получаемым с помощью конструкции `case`. В случае, если текущий элемент содержит число, осуществляется его преобразование в строку. Если

элемент содержит строку, то ничего делать не нужно, именно она и выступает в качестве своего строкового представления. Последняя ветвь `case` является ветвью по умолчанию. Так как класс `Object` является предком всех остальных классов, то именно эта ветвь выбирается во всех случаях, когда хранящийся в элементе объект не является ни числом, ни строкой. В таком случае мы просто останавливаем выполнение программы с помощью функции `abort`.

Обратите также внимание на выражение ветви `Object`. Так как возвращаемым типом функции `abort` является `Object`, а типом значения `case` является наименьший общий тип его ветвей, то выражение данной ветви, состоящее только из вызова `abort`, приведет к тому, что типом выражения `case`, выведенным системой проверки типов, будет `Object`, что не согласуется с объявленным типом переменной `string`. Вы можете проверить это, изменив данную строку на «`o : Object => abort();`» и попробовав скомпилировать программу:

```
$ coolc list3.cl atoi.cl -o list3.s
list3.cl:14: Inferred type Object of initialization of string does not conform to
  identifier's declared type String.
Compilation halted due to static semantic errors.
```

Мы же обходим эту проблему, воспользовавшись блоком выражений, последним элементом которого является строка, поэтому типом выражения `case` будет `String`. Проверим, что все компилируется и работает верно (обратите внимание, что кроме строк мы добавили в список число 42).

```
$ coolc list3.cl atoi.cl -o list3.s
$ spim -file list3.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 for a full copyright notice.
Loaded: /home/compilers_course/lib/trap.handler
Hello World!42
COOL program successfully execute
```

7. Заключение

Мы рассмотрели на приведенных примерах далеко не все возможности языка. За более сложными и интересными примерами обратитесь к программам, идущим с дистрибутивом Cool и находящимся в директории `/home/compilers_course/examples`. За полным описанием Cool обратитесь к документу «*Справочное руководство по языку Cool*».