Лекция 3. Язык программирования Рефал (продолжение)

Коновалов А.В.

11 марта 2023 г.

Расширение базисного подмножества

Базисный Рефал — семантическое подмножество языка Рефал, рассмотренное ранее, т.е. предложения функций состоят из образцового и результатного выражений.

Это подмножество общее для всех реализаций Рефала. Различные реализации по-разному расширяют это подмножество.

В Рефале-5 в качестве расширений используются **условия** и **блоки.**

Условия

К образцовому выражению в предложении можно через запятую приписать одно или несколько **условий** — конструкций вида

```
, результатное-выражение : образцовое-выражение
```

В этом случае сопоставление с образцом будет успешным, если условия выполняются. Условие выполняется, если значение результатного выражения можно сопоставить с образцом.

Переменные, получившие значения в условии, можно использовать в последующих условиях и правой части предложения.

Пример. Функция, которая находит в выражении первый знак арифметического действия (+, -, *, /) и разбивает по нему выражение. Если такового не нашлось, функция возвращает слово `Fails`.

```
SplitByArithm {
 e.Before s.Sign e.After, '+-*/' : e.1 s.Sign e.2 = (e.Before) (e.After);

  e.Other = Fails;
}
```

В аргументе функции ищется первое вхождение символа `s.Sign` такое, что можно сопоставить выражение `'+-*/'` с образцом `e.1 s.Sign e.2`. Заметим, что в образце условия переменная `s.Sign` повторная.

Пример. Встроенная функция `<Compare s.X s.Y>` сравнивает два числа и возвращает знак их разности в виде одной из литер `'-'`, `'0'`, `'+'`.

Напишем простую и неэффективную функцию сортировки чисел:

```
Sort {
  e.Before s.X s.Y e.After, <Compare s.X s.Y> : '+'
    = <Sort e.Before s.Y s.X e.After>;

  e.Sorted = e.Sorted;
}
```

Первое предложение функции находит пару соседних макроцифр в неправильном порядке и их обменивает. Цикл (хвостовая рекурсия) продолжается до тех пор, пока имеются пары чисел с нарушением порядка.

Можно показать, что сложность здесь будет $O(n^3)$ и вообще это плохая реализация сортировки пузырьком.

Блок

Блок — вызов вспомогательной анонимной функции внутри правой части предложения. Синтаксис:

```
, результатное-выражение : { предложения };
```

Вычисляется результатное выражение и вызывается безымянная функция с аргументом, равным значению результатного выражения. Результат блока становится результатом функции, содержащей блок.

Пример. Давайте теперь напишем эффективную версию сортировки — быструю сортировку.

```
QuickSort {
  /* пусто */ = /* пусто */;

  s.Pivot e.Items
    , <Partition s.Pivot e.Items> : (e.Less) (e.Equal) (e.Greater)
    = <QuickSort e.Less> e.Equal <QuickSort e.Greater>;
}

Partition {
  s.Pivot e.Items = <DoPartition s.Pivot () () () e.Items>;
}

DoPartition {
```

```
  s.Pivot (e.Less) (e.Equal) (e.Greater) s.Next e.Items
    , <Compare s.Next s.Pivot>
    : {
      '-' = <DoPartition s.Pivot (e.Less s.Next) (e.Equal) (e.Greater) e.Items>;
      '0' = <DoPartition s.Pivot (e.Less) (e.Equal s.Next) (e.Greater) e.Items>;
      '+' = <DoPartition s.Pivot (e.Less) (e.Equal) (e.Greater s.Next) e.Items>;
       };

  s.Pivot (e.Less) (e.Equal) (e.Greater) /* пусто */
    = (e.Less) (s.Pivot e.Equal) (e.Greater);
}
```

В этом примере вспомогательная функция `DoPartition` реализует цикл на хвостовой рекурсии. Переменные цикла — три “корзины” (их ещё называют “карманами”) `(e.Less) (e.Equal) (e.Greater)`, в которые раскидываются несортированные элементы.

Форматы функций

В Рефале все функции формально принимают один аргумент и возвращают одно значение. На практике этого недостаточно — часто требуется иметь функции с несколькими аргументами и часто с несколькими возвращаемыми значениями. Передачу и возврат нескольких значений имитируют при помощи **форматов** функций — соглашений, описывающих упаковку нескольких значений в одно.

Как правило, значения-символы и значения-термы передаются как есть, значения-выражения заворачиваются в скобки. Если требуется передать N значений-выражений, то для однозначного разбора достаточно $N - 1$ из них завернуть в скобки.

Формат как правило описывается комментарием вида

```
<ИмяФункции ОбразецАргумента> == ОбразецРезультата
```

Двойной знак `==` символизирует, что вычисление выполняется за несколько шагов рефал-машины (см. учебник Турчина). `ОбразецАргумента` и `ОбразецРезультата` — образцы, в которых нет открытых и повторных переменных (т.н. жёсткие образцы).

Пример. Форматы функций `Partition` и `DoPartition` можно описать как

```
<Partition s.Pivot e.Items> == (e.Less) (e.Equal) (e.Greater)

<DoPartition s.Pivot (e.Less) (e.Equal) (e.Greater) e.Items>
  == (e.Less) (e.Equal) (e.Greater)
```

Одна пара скобок в выходном формате обеих функций избыточна — для трёх e-значений достаточно только два из них завернуть в скобки. Однако, скобки

написаны здесь из стилистических соображений.

Быстрая сортировка с комментариями-форматами будет выглядеть так:

```
/*
  <QuickSort e.Items> == e.Items
*/
QuickSort {
  /* пусто */ = /* пусто */;

  s.Pivot e.Items
    , <Partition s.Pivot e.Items> : (e.Less) (e.Equal) (e.Greater)
    = <QuickSort e.Less> e.Equal <QuickSort e.Greater>;
}

/*
  <Partition s.Pivot e.Items> == (e.Less) (e.Equal) (e.Greater)
*/
Partition {
  s.Pivot e.Items = <DoPartition s.Pivot () () () e.Items>;
}

/*
  <DoPartition s.Pivot (e.Less) (e.Equal) (e.Greater) e.Items>
    == (e.Less) (e.Equal) (e.Greater)
*/
DoPartition {
  s.Pivot (e.Less) (e.Equal) (e.Greater) s.Next e.Items
    , <Compare s.Next s.Pivot>
    : {
      '-' = <DoPartition s.Pivot (e.Less s.Next) (e.Equal) (e.Greater) e.Items>;
      '0' = <DoPartition s.Pivot (e.Less) (e.Equal s.Next) (e.Greater) e.Items>;
      '+' = <DoPartition s.Pivot (e.Less) (e.Equal) (e.Greater s.Next) e.Items>;
       };

  s.Pivot (e.Less) (e.Equal) (e.Greater) /* пусто */
    = (e.Less) (s.Pivot e.Equal) (e.Greater);
}
```

Комментарий `/* пусто */` в последнем предложении показывает, что компонент формата `DoPartition e.Items` сопоставляется с пустотой.

Ограниченная поддержка высшего порядка — функция Mu

В Рефале-5 функции как полноценные значения отсутствуют — среди типов символов у нас нет символа-функции (а в Рефале-5-лямбда — есть!). Однако, есть встроенная функция `Mu`, которая позволяет вызвать произвольную функцию

по её имени. Её формат:

```
<Mu s.WORD e.Arg> ~~ <Func e.Arg>
<Mu (e.Name) e.Arg> ~~ <Func e.Arg>
```

Здесь `s.WORD` — имя функции `Func`, записанное в виде символа-слова, `e.Name` — имя функции `Func`, записанное в виде последовательности литер.

Пример.

```
<Mu Compare 5 7>
<Mu ('Compare') 5 7>
```

В обоих случаях мы получим `'-'`, т.к. $5 < 7$.

Можно написать, например, функцию `Map`, которая принимает имя функции и последовательность термов и применяет эту функцию к каждому из термов:

```
/*
  <Map s.Func e.Items> == e.Expr
*/
Map {
  s.Func t.Next e.Rest = <Mu s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}
```

Например, `<Map Prout 'hello'>` напечатает 5 строчек по одной букве.

Другой пример:

```
Bracket { t.X = (t.X) }

Map { ... }

$ENTRY Go {
  = <Prout <Map Bracket 'hello'>>
}
```

Напечатается `(h)(e)(l)(l)(o)`.

Грамматики типов для программ на Рефале

Для более точного описания областей определений и значений функций (т.е. точнее, чем комментарии-форматы) используются грамматики типов. Вариантов грамматик у разных программистов на Рефале много, мы будем придерживаться следующего.

В грамматиках терминальные символы — литералы для символов (слов, литер, чисел) и круглые скобки. Нетерминалы — имена типов, которые обозначаются как переменные.

Правила грамматик имеют вид

```
Нетерминал ::= правая-часть
```

где правая часть может содержать обозначения

```
альтернатива | ... | альтернатива
(скобочный терм)
{ группировка }
ноль-или-более-раз*
ноль-или-один-раз?
один-или-более-раз+
```

Зарезервированные имена нетерминалов `s.WORD`, `s.NUMBER`, `s.CHAR` означают соответствующие типы символов. Также будем использовать обозначения `s.ANY`, `t.ANY`, `e.ANY` со следующим смыслом:

```
s.ANY ::= s.WORD | s.NUMBER | s.CHAR
t.ANY ::= s.ANY | (e.ANY)
e.ANY ::= t.ANY*
```

Пример. Опишем тип аргумента функций `Mu`:

```
<Mu t.MuCallee e.ANY>

t.MuCallee ::= s.WORD | (s.CHAR+)
```

Тип функции `Compare`:

```
<Compare s.NUMBER s.NUMBER> == '-' | '0' | '+'
```

Основные встроенные функции Рефала-5

Арифметика

Рефал-5 своеобразно поддерживает длинную арифметику. Длинное число представляется как выражение вида

```
e.LongNumber ::= {'+' | '-'}? s.NUMBER+
```

Необязательный знак в начале в виде литеры, затем несколько макроцифр.

Макроцифры рассматриваются как цифры числа по основанию системы счисления 2^{32}. Т.е., например, запись `'-' 37 1234 999` будет трактоваться как число $-(37 \cdot 2^{64} + 1234 \cdot 2^{32} + 999)$.

Встроенные функции `Add`, `Sub`, `Mul`, `Div`, `Mod` имеют следующий тип:

```
<Функция e.First e.Second> == e.LongResult

e.First ::= {'+' | '-'}? s.NUMBER | (e.LongNumber)
e.Second ::= e.LongNumber
```

```
e.LongResult ::= '-'? s.NUMBER+
```

Т.е. первый аргумент представляет длинное число в скобках, но если он содержит только одну макроцифру, то скобки можно опустить. Результат арифметической функции тоже может быть длинным, но он не может начинаться на знак `'+'`.

В вызове `<Add 1 2 3 4>` первым аргументом будет `1`, вторым `2 3 4`, результатом — `2 3 5`.

Функция `Divmod` одновременно вычисляет частное и остаток, имеет тип

```
<Divmod e.First e.Second> == (e.Div) e.Mod
e.Div, e.Mod ::= e.LongResult
```

Функция `Compare` тоже умеет сравнивать длинные числа:

```
<Compare e.First e.Second> == '+' | '0' | '-'
```

Для преобразования между строкой и числом используются функции `Numb` и `Symb`:

```
<Numb s.CHAR*> == e.LongResult
<Symb e.LongNumber> == s.CHAR+
```

Функция `Numb` ищет префикс аргумента максимальной длины, являющийся записью числа и его парсит, остаток отбрасывает. Например,

```
<Numb '-1234abcdef'> = '-' 1234
```

Если аргумент числом не является, возвращается значение `0`.

Функция `Symb` в ответе сохраняет знак, даже если это `'+'`:

```
<Symb '+' 1234> = '+1234'
```

Ввод-вывод

Вывод на экран (stdout) осуществляется функциями `Print` и `Prout`:

```
<Print e.ANY> == e.ANY
<Prout e.ANY> == пусто
```

Функция `Print` возвращает свой аргумент, функция `Prout` ничего не возвращает.

Чтение с клавиатуры (stdin) выполняется функцией `Card`:

```
<Card> == s.CHAR* 0?
```

В конец результата вызова добавляется макроцифра `0`, если в stdout обнаружился конец файла (EOF).

Для работы с файлом его нужно открыть. В Рефале-5 есть 39 глобальных переменных — номеров файлов, которые можно открыть. Файл открывается функцией `Open`:

```
<Open s.Mode s.FileNo e.FileName> == пусто

s.Mode ::= 'r' | 'w' | 'a' | 'R' | 'W' | 'A' | s.WORD
s.FileNo ::= s.NUMBER
e.FileName ::= s.CHAR*
```

Строчные и заглавные литеры в `s.Mode` ничем не различаются.

В качестве режима символ-слово может быть любой корректной строкой для `fopen()` языка Си, например `wb`.

Номер файла — макроцифра. Если её значение больше 39, берётся остаток от деления на 40. Номер файла `0` зарезервирован — это stdin для ввода и stderr для вывода.

Если имя файла не указано, то по умолчанию открывается файл `REFAL<no>.DAT`, где вместо `<no>` записывается номер файла. Например

```
<Open 'w' 10>
```

откроет файл `REFAL10.DAT`.

Файл закрывается функцией

```
<Close s.FileNo> == пусто
```

Функция чтения

```
<Get s.FileNo> == s.CHAR* 0?
```

аналогична функции `<Card>` — читает из файла строку (знак `\n` в конец не добавляется), `0` добавляется при достижении конца файла.

Функции вывода

```
<Put s.FileNo e.ANY> == e.ANY
<Putout s.FileNo e.ANY> == пусто
```

аналогичны `Print` и `Prout`.

Функция

```
<Write s.FileNo e.ANY> == e.ANY
```

не добавляет знак перевода строки при печати.

Функции вывода (`Prout`, `Print`, `Put`, `Putout`, `Write`) после макроцифр и символов-слов принудительно добавляют пробел:

```
<Prout 1 2 3 'abcd'4 5 6 "abcd" 7 (8) 9>
```

Напечатается:

```
1 2 3 abcd4 5 6 abcd 7 (8 ) 9
```

Если хочется напечатать число без пробела, его нужно сначала сконвертировать в строку при помощи `Symb`:

```
PrintError {
  (e.FileName) s.Line s.Col e.Message
    <Putout
     0 e.FileName ':' <Symb s.Line> ':' <Symb s.Col> ':' e.Message
    >
}
```

Эта функция будет распечатывать сообщение об ошибке в виде файл:строка:колонка:сообщение.

Функции преобразования типов

```
<Chr e.ANY> == e.ANY
<Ord e.ANY> == e.ANY
<Upper e.Any> == e.ANY
<Lower e.Any> == e.ANY

<Chr (('@'))> == ((64))
<Ord (100) 101> == ('d') 'e'

<Explode s.WORD> == s.CHAR*
<Implode e.ANY> == {s.WORD | 0} e.ANY
<Implode_Ext s.CHAR*> == s.WORD
```

Implode откусывает префикс максимальной длины, являющийся записью идентификатора, если не получилось, возвращает 0:

```
<Implode 'abc12-34_56!@#$%^&'> == abc12-34_56 '!@#$%^&'
<Implode '!@#$%^'> == 0 '!@#$%^'

<Type e.Expr> == s.Type s.SubType e.Expr

s.Type s.SubType ::=
    'N0' --- макроцифра
  | 'Wi' --- символ-слово в идентификаторной форме
  | 'Wq' --- символ-слово, который надо записывать в кавычках
  | 'D0' --- литера-цифра ('0' ... '9')
 | 'Lu' | 'Ll' --- литера-буква, соответственно заглавная или строчная
  | 'P' s.X --- печатный символ (isprint(c) вернёт true)
  | 'O' s.X --- любая другая литера
  | 'B0' --- терм в скобках
  | '*0' --- пустое выражение
```

Функция Type позволяет узнать тип первого терма выражения. Она принимает выражение, возвращает пару литер “тип” и “подтип” и исходный аргумент как есть.

```
<Lenw e.Expr> == s.NUMBER e.Expr
```

Вычисляет длину в термах, возвращает длину и само выражение.

```
<First s.NUMBER e.Expr> == (e.Prefix) e.Suffix
<Last s.NUMBER e.Expr> == (e.Prefix) e.Suffix
```

Отделяют, соответственно, префикс и суффикс заданной длины в термах.

Взаимодействие с ОС

```
<System e.Command> == e.RetCode
e.Command ::= s.CHAR+
e.RetCode ::= '-'? s.NUMBER

<System 'ls -l'> == 0

<Arg s.NUMBER> == s.CHAR*

<Arg 0> == 'main.rsl+parse.rsl+generate.rsl'
<Arg 1> == 'source.txt'
<Arg 2> == 'dest.txt'
<Arg 100500> == пусто

<Exit e.RetCode> == нет результата
```

Синтаксический сахар

Функции арифметики можно вызывать как <+ … >, <- … >, <* … >, </ … >, <% … >.

Модули

Запуск

Программа на Рефале-5 может состоять из нескольких независимо компилируемых компонентов. Компилятор `refc` может получать в командной строке несколько имён файлов и независимо их транслировать — для каждого исходника будет создан свой файл .rsl.

Для того, чтобы запустить программу, собранную из нескольких .rsl-файлов, их имена для интерпретатора `refgo` нужно перечислить через знак +:

```
refgo main.rsl+parse.rsl+generate.rsl source.txt dest.txt
```

или

```
refgo main+parse+generate source.txt dest.txt
```

При поиске модулей интерпретатор сначала смотрит в текущую папку, затем в папки, перечисленные в переменной среды `REF5RSL`. Конечно, можно задавать и полный путь до .rsl-ек.

Написание программ

Чтобы вызвать функцию, написанную в другом модуле, её имя нужно пометить как внешнее — указать в списке внешних имён в директиве $EXTERN:

```
$EXTERN Parse, Generate, ReportErrors;
```

Функции по умолчанию имеют локальную область видимости — из других файлов их вызвать нельзя. Чтобы функцию можно было вызвать из другого файла, перед её именем должно быть указано ключевое слово $ENTRY:

```
$ENTRY Parse {
    ...
}
```

Модули и функция Mu

Функция Mu вызывает функцию с заданным именем из того файла, где записан вызов функции Mu:

```
/* файл 1 */

$EXTERN CallMu;

F { = <Print 'Файл 1'> }

$ENTRY Go {
  = <CallMu F>
}

/* файл 2 */

$ENTRY CallMu {
  s.Name = <Mu s.Name>;
}

F { = <Print 'Файл 2'> }
```

В этом примере напечатается Файл 2, т.к. вызов функции Mu записан во втором файле.

Можно условно считать, что компилятор добавляет в каждый файл неявно сгенерированную функцию Mu вот такого вида:

```
$ENTRY Func1 { ... }
Func2 { ... }
Func3 { ... }
$ENTRY Func4 { ... }
```

```
Mu {
   Add e.X = <Add e.X>;
   Arg e.X = <Arg e.X>;
    ...
   Func1 e.X = <Func1 e.X>;
   Func2 e.X = <Func2 e.X>;
    ...
   (e.Name) e.X = <Mu <Implode_Ext e.Name> e.X>;
}
```

Но есть из этого правила исключение: если в текущем файле функции с заданным именем не нашлось, но при этом в программе где-то есть функция с этим именем, помеченная ключевым словом $ENTRY, то вызовется она.

Пример. Напишем модуль `map.ref`, который содержит функцию `Map`:

```
$ENTRY Map {
  s.Func t.Next e.Rest = <Mu s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}
```

А теперь напишем модуль `main.ref`, который эту `Map` вызывает:

```
$EXTERN Map;

Split {
  ' ' e.Rest = <Split e.Rest>;
  e.Word ' ' e.Rest = (e.Word) <Split e.Rest>;
  /* пусто */ = /* пусто */;
  e.Word = (e.Word);
}

$ENTRY ParseInt { (e.Value) = <Numb e.Value> }
$ENTRY Square { s.X = <Mul s.X s.X> }

$ENTRY Go {
  = <Prout
      <Map Square <Map ParseInt <Split <Card>>>>
    >
}
```

Эта программа читает со стандартного ввода строчку, где записано несколько чисел через пробелы и распечатывает квадраты этих чисел. Считаем, что числа небольшие (менее 65536) и положительные, поэтому и само число, и его квадрат представимы в виде одной макроцифры.

Функции `ParseInt` и `Square` помечены как `$ENTRY` для того, чтобы их увидела функция `Mu` в модуле `map.rsl`. Рекомендуется функции, помеченные `$ENTRY` только ради вызова извне при помощи `Mu` называть как `имяфайла_ИмяФункции`:

```
$EXTERN Map;

Split {
  ' ' e.Rest = <Split e.Rest>;
  e.Word ' ' e.Rest = (e.Word) <Split e.Rest>;
  /* пусто */ = /* пусто */;
  e.Word = (e.Word);
}

$ENTRY main_ParseInt { (e.Value) = <Numb e.Value> }
$ENTRY main_Square { s.X = <Mul s.X s.X> }

$ENTRY Go {
  = <Prout
      <Map main_Square <Map main_ParseInt <Split <Card>>>>
    >
}
```

Так мы избегаем конфликта имён и подчёркиваем, что эта функция не часть публичного API данного модуля.

В документации к Рефалу-05 есть детальное обсуждение семантики функции `Mu` как Рефала-5, так и Рефала-05:

https://mazdaywik.github.io/Refal-05/2-syntax

Библиотека `LibraryEx`

Эта библиотека входит в дистрибутив Рефала-5-лямбда, либо её можно взять из следующего репозитория:

https://github.com/mazdaywik/refal-5-framework

Документация к ней:

https://mazdaywik.github.io/refal-5-framework

В библиотеке есть следующие полезные функции:

- Функции высших порядков `Map`, `Reduce`, `MapAccum`, работают через функцию `Mu`, поддерживается каррирование.
- Функции `LoadFile` и `SaveFile` — первая принимает имя файла и загружает из него все строки, вторая принимает имя файла и содержимое и создаёт новый файл (или переписывает имеющийся) с данным содержимым.
- Функция `ArgList`, считывающая все аргументы до первого пустого.

Недавно добавлена в неё функция `LoadExpr`, которой мы будем пользоваться. Документация к ней пока ещё не написана.

```
<LoadExpr e.FileName> == e.ANY
```

Функция принимает имя файла, в котором записано объектное выражение в виде литерала Рефала-5 и парсит его. Если разбор неудался (например, скобка незакрытая), функция завершает программу возвратом кода 1: `<Exit 1>`.

```
<TryLoadExpr e.FileName> == Success e.ANY | Fails e.ErrorMessage
```