Лекция 2. Язык программирования Рефал-5

Коновалов А.В.

28 февраля 2024 г.

О Рефале (1)

РЕФАЛ (**РЕ**курсивный **Ф**ункциональный **АЛ**горитмический язык, REFAL — **RE**cursive **F**unctional **A**lgorithmic **L**anguage) — семейство функциональных языков программирования, характеризующееся следующими общими чертами:

- данные представлены цепочками (*объектными выражениями*), составленными из символов и скобок, причём скобки образуют правильную структуру;
- конкатенация объектных выражений является фундаментальной операцией;
- анализ значений выполняется путём сопоставления объектного выражения с образцом;
- построение значений выполняется путём интерполяции переменных в выражении с переменными и вызовами функций.

О Рефале (2)

В семейство входит несколько языков, синтаксически несовместимых между собой. На сайте http://refal.ru описаны реализации Рефал-2, Рефал-5, Рефал-6, Рефал Плюс. Можно встретить публикации, в которых описываются разрабатываемые на данный момент реализации Рефала (автор этих строк встречал как минимум 3), тоже несовместимые между собой.

В 2006 году Скоробогатовым С.Ю. был предложен диалект Refal-7 с функциями высшего порядка.

С 2016 года на кафедре ИУ9 разрабатывается диалект Рефал-5λ с функциями высшего порядка, являющийся расширением Рефала-5 (любая программа на Рефале-5 является допустимой программой на Рефале-5λ).

О Рефале (3)

В нашем курсе будет использоваться диалект Рефал-5 (хотя использовать Рефал-5λ не возбраняется ☺). Актуальная документация по этому диалекту доступна только на английском языке по ссылке

http://www.botik.ru/pub/local/scp/refal5/

(На сайте http://refal.ru можно найти перевод на русский язык, однако он описывает устаревший синтаксис, не читайте его.)

Синтаксис Рефала-5

Данные Рефала-5 (1)

Рефал-5 — динамически типизированный язык. Данные представлены **объектными выражениями** — цепочками, составленными из символов и круглых скобок, при этом круглые (их называют **структурные**) скобки должны быть сбалансированы.

Символы бывают трёх видов:

- **символы-литеры** представляют собой запись ASCII-символов, записываются в одинарных кавычках: `'a'`, `'%'`, `'+'`, `'\n'`, `'\x1B'`;
- **символы-числа** или **макроцифры** записываются как десятичные беззнаковые целые числа от `0` до `4294967295` ($2^{32} - 1$);
- **составные символы (*compound symbols*)** или **символы-слова** записываются как последовательности знаков в двойных кавычках: `"Word"`, `"\n\n\n"`, `"+"`, `"<<"`.

Данные Рефала-5 (2)

Несколько символов-литер можно записывать слитно: запись `'hello'` эквивалентна записи `'h' 'e' 'l' 'l' 'o'`.

Если символ-слово является корректной записью идентификатора (имени функции) в Рефале-5 — начинается на букву, состоит из букв, цифр, знаков - и _ — то его можно записывать без двойных кавычек. Запись `Scan-ForName` эквивалента `"Scan-ForName"`.

Заметим, что запись `'abcd'` — это объектное выражение, состоящее из четырёх литер (эквивалентно `'a' 'b' 'c' 'd'`), а запись `"abcd"` — это единственный символ-слово (его можно записать без кавычек: `abcd`).

Выражение, записанное в круглых скобках, называется **скобочным термом**.

Объектный терм (далее, просто **терм**) — это либо символ, либо скобочный терм.

Данные Рефала-5 (3)

Примеры объектных выражений:

```
'abc' 2 3 (4 x 5 y (((6)))) 7 8

(2 "*" (Count "+" 7))

(name 'Alexander') (surname 'Konovalov')

(function gcd (x y)
  (var (rem 1))
  (while ((L y) "<>" 0)
    (rem "=" ((L x) "%" (L y)))
    (x "=" (L y))
    (y "=" (L rem))
  )
  (return (L y))
)
```

Данные Рефала-5 (4)

Данные Рефала-5 напоминают s-выражения языка LISP. Можно даже построить следующую табличку, сопоставляющую понятия Рефала-5 и понятия Scheme:

Рефал-5	Scheme
объектное выражение	список
скобочный терм	вложенный список
литера `'a'`	литера `#\a`
макроцифра	целое число
составной символ	строка `"Hello!"`
составной символ	атом `'hello`

Отличие между данными Лиспа и Рефала — в операциях, применимых к ним. Списки Лиспа однонаправленные (их можно разбирать и наращивать только слева), объектные выражения Рефала можно разбирать с обеих сторон и конкатенировать.

Синтаксис программ на Рефале-5 (1)

Программа на Рефале-5 состоит из определений функций.

Определение функции записывается следующим образом:

```
ИмяФункции {
  предложение;
  предложение;
   ...
}
```

Имя функции должно быть корректным идентификатором Рефала (начинаться с латинской буквы, состоять из латинских букв, цифр и знаков _ и -).

Если функция должна быть доступна из других модулей, перед её именем ставится ключевое слово `$ENTRY`.

Предложений может быть 1 и более (в Рефале-5λ — 0 и более). После последнего предложения точку с запятой можно не ставить.

Синтаксис программ на Рефале-5 (2)

Предложение в простейшем случае имеет вид

```
образец = результат;
```

где `образец` — выражение Рефала, которое может содержать переменные (т.н. **образцовое выражение**), а `результат` — выражение Рефала, которое может содержать и переменные, и вызовы функций (т.н. **результатное выражение**).

Синтаксис программ на Рефале-5 (3)

Переменные в Рефале-5 записываются как `‹вид›.‹индекс›`, где `‹вид›` — одна из трёх букв `s`, `t`, `e`, `‹индекс›` — идентификатор или целое число.

Вид переменной определяет множество значений, на которые переменная может заменяться:

- **s-переменные** могут заменяться на произвольный символ,
- **t-переменные** могут заменяться на произвольный терм (символ или выражение в скобках),
- **e-переменные** могут заменяться на произвольное выражение.

Синтаксис программ на Рефале-5 (4)

Функции принимают ровно один аргумент, он и сопоставляется с образцом.

Вызов функции записывается в угловых скобках:

```
<ИмяФункции аргумент>
```

где `аргумент` — результатное выражение.

Угловые скобки также называются **скобками вызова**, **скобками активации** или **скобками конкретизации**.

Образцы в Рефале-5 (1)

Образцовое выражение (или **образец**) состоит из символов, структурных (круглых) скобок и переменных.

Образец описывает множество объектных выражений, которые можно получить, подставив вместо переменных какие-либо значения соответствующего вида.

Образцы могут содержать несколько вхождений одной и той же переменной (одного и того же вида и с одним и тем же индексом). Такие переменные называются **повторными**. Все вхождения повторной переменной должны иметь одинаковые значения.

Образцы в Рефале-5 (2)

Примеры образцов:

- `s.1 s.2 s.3` — выражение любых трёх символов: `1 2 3`, `'abc'`, `1 "+" 2`, `1 '+' 2`, `One Two Three`.
- `s.A s.A s.A` — выражение из трёх одинаковых символов: `'aaa'`, `1 1 1`, `hello hello hello`.
- `t.A t.B t.A` — выражение из трёх термов, первый и последний должны быть одинаковыми: `'aba'`, `() (a) ()`, `(1 2) 3 (1 2)`.
- `e.X '+' e.Y` — выражение, которое содержит знак `'+'` на верхнем уровне, например: `'+'`, `'2+3=5'`, `() '-' () '+' ()`, `'+++'`.
- `(e.X) e.Y (e.Z)` — выражение как минимум из двух термов, первый и последний — скобочные: `(a b) c d (e f)`.

Образцы в Рефале-5 (3)

Сопоставление с образцом $E : P$ — поиск таких значений переменных, подстановка которых в образец P даёт объектное выражение E.

Пример.

```
'hello' : e.X s.R s.R e.Y
```

При сопоставлении переменная `e.X` получит значение `'he'`, `s.R` — `'l'`, `e.Y` — `'o'`.

Образцы в Рефале-5 (4)

Сопоставление с образцом в Рефале *неоднозначно*. Это означает, что для некоторых сопоставлений $E : P$ можно найти несколько различных подстановок переменных, таких что P превращается в E. Например:

```
'abracadabra' : e.X s.R1 s.R2 e.Y s.R1 s.R2 e.Z
```

Возможные подстановки:

```
e.X → пусто, s.R1 → 'a', s.R2 → 'b', e.Y → 'racad', e.Z → 'ra'
e.X → 'a', s.R1 → 'b', s.R2 → 'r', e.Y → 'acada', e.X → 'a'
e.X → 'ab', s.R1 → 'r', s.R2 → 'a', e.Y → 'cadab', e.Z → пусто
```

Образцы в Рефале-5 (5)

Неоднозначность разрешается по следующему правилу: выбирается подстановка с кратчайшим (в количестве термов) значением самой левой e-переменной. Если это не разрешает неоднозначности, то выбирается следующая e-переменная и так далее.

Т.е. в предыдущем примере будет выбрана подстановка, где `e.X` пустая.

Ещё пример:

```
1 2 3 2 4 3 2 : e.X s.R e.Y s.R e.Z
```

Переменные получат следующие значения: `e.X` — `1`, `s.R` — `2`, `e.Y` — `3`, `e.Z` — `4 3 2`.

Образцы в Рефале-5 (6)

e-переменные, которые участвуют в разрешении неоднозначности, называются **открытыми e-переменными**. Например, в образце `e.X s.R e.Y s.R e.Z` открытыми переменными будут `e.X` и `e.Y`.

В образце `(e.X) e.X A e.Y` открытых переменных нет, т.к. образец описывает множество выражений, начинающихся со скобочного терма, сопоставление скобочного терма с `(e.X)` выполняется однозначно — однозначно определяется значение переменной `e.X`. Второе её вхождение повторное и оно тоже определяется однозначно.

В образце `e.A '*' e.A` первое вхождение `e.A` открытое, второе — повторное.

В образце `(e.X s.R e.Y) (e.U s.R e.V)` открытые только `e.X` и `e.U`.

Образцы в Рефале-5 (7)

Открытые переменные переменные в образцах неявно компилируются в циклы при сопоставлении с образцом. Таким образом, образец с одной открытой переменной имеет линейную сложность сопоставления, с двумя — квадратичную, с тремя — кубическую и т.д.

Кроме того, сопоставления с повторными e- и t-переменными также требуют сравнения на равенство соответствующих фрагментов сопоставимого выражения, что тоже требует времени.

Таким образом, сопоставление с образцом в Рефале может выполняться не за $O(1)$, а зависеть по времени от размера входных данных.

Программы в Рефале-5 (1)

Выполнение программы на Рефале-5 начинается с вызова функции `Go` с пустым аргументном (аналогично функции `main()` языка Си). Т.к. эта функция должна вызываться извне модуля, перед её именем должно быть записано ключевое слово `$ENTRY`.

Встроенная функция, распечатывающая объектное выражение на экране, называется `Prout` (от print out).

Таким образом, кратчайшая программа на Рефале имеет вид

```
$ENTRY Go {
  /* пусто */ = <Prout 'Hello, World!'>
}
```

Программы в Рефале-5 (2)

```
$ENTRY Go {
  /* пусто */ = <Prout 'Hello, World!'>
}
```

Здесь на месте пустого образца написан *комментарий.* Комментарии в Рефале-5 записываются как в языке Си: `/* ... */`. Также допустимы однострочные комментарии — строка текста целиком, начинающаяся со `*`.

Программы в Рефале-5 (3)

Программы записываются в текстовых файлах с расширением `.ref`. Для компиляции программы нужно ввести в командной строке:

```
$ refc hello.ref
```

В результате получится файл байткода (т.н. RASL) с расширением `.rsl`. Его можно запустить интерпретатором байткода

```
$ refgo hello.rsl
```

Программы в Рефале-5 (4)

Для ввода строки текста из `stdin` используется встроенная функция `Card`, вызываемая без аргументов (название восходит к тем временам, когда компьютеры читали перфокарты, функция `Card` считывала очередную перфокарту).

Напишем программу, которая приветствует пользователя:

```
$ENTRY Go {
  = <Prout 'Как тебя зовут?'>
    <Prout 'Привет, ' <Card>>
}
```

Если в выражении имеется несколько вызовов функций, то они выполняются слева направо.

Программы в Рефале-5 (5)

Напишем функцию, которая запрашивает у пользователя строку и заменяет в ней все вхождения слова `cat` на слово `dog`:

```
$ENTRY Go {
  = <Prout <Replace <Card>>>
}

Replace {
 e.Before 'cat' e.After = e.Before 'dog' <Replace e.After>;
  e.NoCat = e.NoCat;
}
```

Программы в Рефале-5 (6)

Напишем функцию, которая запрашивает у пользователя строку и печатает уникальные символы из этой строки.

```
$ENTRY Go {
  = <Prout <Unique <Card>>>
}

Unique {
  e.Begin s.Rep e.Mid s.Rep e.End
    = <Unique e.Begin s.Rep e.Mid e.End>;

  e.Unique = e.Unique;
}
```

Программы в Рефале-5 (7)

Функцию Unique можно немного оптимизировать:

```
$ENTRY Go {
  = <Prout <Unique <Card>>>
}

Unique {
  e.Begin s.Rep e.Mid s.Rep e.End
    = e.Begin <Unique s.Rep e.Mid e.End>;

  e.Unique = e.Unique;
}
```

Программы в Рефале-5 (8)

Напишем программу, которая принимает у пользователя две строки и находит пересечение множеств символов, составляющих эти строки.

```
$ENTRY Go {
  = <Prout <Intersect (<Unique <Card>>) (<Unique <Card>>)>>
}

Intersect {
  (e.Set1-B s.Rep e.Set1-E) (e.Set2-B s.Rep e.Set2-E)
    = s.Rep <Intersect (e.Set1-B e.Set1-E) (e.Set2-B e.Set2-E)>;

  (e.Set1) (e.Set2) = /* пусто */;
}

Unique {
   ... см. ранее ...
}
```

Программы в Рефале-5 (9)

Её тоже можно оптимизировать:

```
$ENTRY Go {
  = <Prout <Intersect (<Unique <Card>>) (<Unique <Card>>)>>
}

Intersect {
  (e.Set1-B s.Rep e.Set1-E) (e.Set2-B s.Rep e.Set2-E)
    = s.Rep <Intersect (e.Set1-E) (e.Set2-B e.Set2-E)>;

  (e.Set1) (e.Set2) = /* пусто */;
}

Unique {
   ... см. ранее ...
}
```

Расширение базисного подмножества

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 (1)

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.

Условия (2)

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.

Условия (3)

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 (1)

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.

Блок (2)

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

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

  s.Pivot e.Items
    , <Partition s.Pivot e.Items> : (e.Le) (e.Eq) (e.Gre)
    = <QuickSort e.Le> e.Eq <QuickSort e.Gre>;
}

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

…продолжение на следующем слайде…

Блок (3)

…продолжение…

```
DoPartition {
  s.Pivot (e.Le) (e.Eq) (e.Gre) s.Next e.Items
    , <Compare s.Next s.Pivot>
    : {
        '-' = <DoPartition s.Pivot (e.Le s.Next) (e.Eq) (e.Gre) e.Items>;
        '0' = <DoPartition s.Pivot (e.Le) (e.Eq s.Next) (e.Gre) e.Items>;
        '+' = <DoPartition s.Pivot (e.Le) (e.Eq) (e.Gre s.Next) e.Items>;
      };

  s.Pivot (e.Le) (e.Equal) (e.Gre) /* пусто */
    = (e.Le) (s.Pivot e.Eq) (e.Gre);
}
```

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

Форматы функций

Форматы функций (1)

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 возвращаемыеми 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.

Форматы функций (2)

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).

Форматы функций (3)

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.

Форматы функций (4)

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.Le) (e.Eq) (e.Gre)
    = <QuickSort e.Le> e.Eq <QuickSort e.Gre>;
}

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

Форматы функций (5)

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

  s.Pivot (e.Le) (e.Equal) (e.Gre) /* пусто */
    = (e.Le) (s.Pivot e.Eq) (e.Gre);
}
```

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

Косвенный вызов функций — функция Mu (1)

В Рефале-5 функции как полноценные значения отсутствуют — среди типов символов у нас нет символа-функции (а в Рефале-5λ — 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.

Косвенный вызов функций — функция Mu (2)

Пример.

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

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

Косвенный вызов функций — функция Mu (3)

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.

Косвенный вызов функций — функция Mu (4)

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

Грамматики типов для программ на Рефале (1)

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.

Грамматики типов для программ на Рефале (2)

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+
```

Грамматики типов для программ на Рефале (3)

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*
```

Грамматики типов для программ на Рефале (4)

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' | '+'
```

Грамматики типов для программ на Рефале (5)

Пример. Ассоциативный список, отображающий имена — цепочки литер на значения — макроцифры:

```
e.NameTable ::= (e.Name s.Value)*
e.Name ::= s.CHAR+
s.Value ::= s.NUMBER
```

Пример значения:

```
('Init' 10) ('Step' 2) ('Limit' 24) ('Sum' 105)
```

Грамматики типов для программ на Рефале (6)

Пример. АВЛ-дерево:

```
t.AVL-Tree ::=
    Leaf
  | (t.AVL-Tree s.Key t.Value s.Balance t.AVL-Tree)
s.Key ::= s.NUMBER
t.Value ::= t.ANY
s.Balance ::= '<' | '=' | '>'
```

Грамматики типов для программ на Рефале (7)

Пример. Абстрактное синтаксическое дерево логики высказываний:

```
t.Formula ::=
    True
  | False
  | (Var s.Name)
  | (Not t.Formula)
  | (t.Formula And t.Formula)
  | (t.Formula Or t.Formula)

s.Name ::= s.WORD
```

Грамматики типов для программ на Рефале (8)

Перевод формулы в нормальную форму отрицания, в которой операция НЕ применяется только к переменным и константам:

```
/*
  <NNF t.Formula> == t.Formula
*/
NNF {
  (Not True) = False;
  (Not False) = True;
  (Not (Not t.Formula)) = <NNF t.Formula>;
  (Not (t.A And t.B)) = (<NNF (Not t.A)> Or <NNF (Not t.B)>);
  (Not (t.A Or t.B)) = (<NNF (Not t.A)> And <NNF (Not t.B)>);
  (t.A And t.B) = (<NNF t.A> And <NNF t.B>);
  (t.A Or t.B) = (<NNF t.A> Or <NNF t.B>);
  t.Other = t.Other;
}
```

Основные встроенные функции Рефала-5

Арифметика (1)

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 $-(37 \cdot 2^{64} + 1234 \cdot 2^{32} + 999)$.

Арифметика (2)

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`».

Арифметика (3)

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' | '-'
```

Имеется синтаксический сахар: функции арифметики можно вызывать как `<+ ... >`, `<- ... >`, `<* ... >`, `</ ... >`, `<% ... >`.

Арифметика (4)

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 (1)

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).

Ввод-вывод (2)

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`.

Ввод-вывод (3)

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
```

Ввод-вывод (4)

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.

Ввод-вывод (5)

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
```

Ввод-вывод (6)

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 (1)

Функции `Chr`, `Ord`, `Upper`, `Lower` производят трансформацию литер, включая вложенные термы.

```
<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'
```

Функции преобразования типов (2)

Преобразование символов-слов в цепочки литер и обратно:

```
<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 '!@#$%^'> == 0 '!@#$%^'
```

Функции преобразования типов (3)

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

```
<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
```

Функции преобразования типов (4)

```
<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
<Arg s.NUMBER> == s.CHAR*
<Exit e.RetCode> == нет результата
```

Семантика понятна из названия. Примеры:

```
<System 'ls -l'> == 0
<Arg 0> == 'main.rsl+parse.rsl+generate.rsl'
<Arg 1> == 'source.txt'
<Arg 2> == 'dest.txt'
<Arg 100500> == пусто
```

Функции закапывания и выкапывания (копилка) (1)

Рефал-5 поддерживает глобальный ассоциативный массив — т.н. **копилку,** которая отображает произвольные объектные выражения на стеки, содержащие объектные выражения.

На верхнем уровне ключ не может содержать знак `'='`.

```
e.Key ::= { t.ANY \ '=' }*
e.Value ::= e.ANY
```

Функции закапывания и выкапывания (копилка) (2)

Доступны следующие функции при работе с копилкой:

```
<Br e.Key '=' e.Value> == пусто
```

Функция «закапывания» (*bury*) — кладёт на стек `e.Key` значение `e.Value`.

```
<Dg e.Key> == e.Value
```

Функция выкапывания (*digg*) — возвращает и удаляет верхушку стека `e.Key`. Если стек был пустым, возвращается пустое выражение.

```
<Dgall> == (e.Key '=' e.Value)*
```

Возвращает содержимое всех стеков с их опустошением.

Функции закапывания и выкапывания (копилка) (3)

```
<Cp e.Key> == e.Value
```

Функция копирует (*copy*) верхушку стека `e.Key` без её удаления. Аналогично, если стек был пустым, возвращается пустота.

```
<Rp e.Key '=' e.Value> == пусто
```

Заменяет верхушку стека `e.Key` на новое значение `e.Value`. Если стек был пустым, то новое значение кладётся на стек.

Использование функций закапывания-выкапывания (1)

Использование глобальных переменных противоречит функциональной парадигме и затрудняет чтение программы.

Глобальные стеки нигде явным образом не определяются (т.е. не предусмотрено никаких синтаксических конструкций вроде `$STACK x;`), поэтому, чтобы понять, какими глобальными переменными пользуется некоторая функция, нужно изучить исходный текст самой функции, а также других функций, которые она вызывает.

Создать стеки, видимые только для отдельной единицы трансляции, невозможно.

Использование функций закапывания-выкапывания (2)

Советы:

- Лучше придерживаться функциональной парадигмы — входные и выходные данные будут непосредственно присутствовать в описании формата/типа функции.
- Если функция пользуется копилкой, в комментарии к ней нужно указывать, с какими стеками она работает.
- Копилкой можно пользоваться в духе динамических переменных Common Lisp:
  - перед выполнением некоторого алгоритма переменные инициализируются при помощи `Br`,
  - в самом алгоритме переменные могут изменяться при помощи `Cp` и `Rp`,
  - после завершения переменные уничтожаются при помощи `Dg`.

Использование функций закапывания-выкапывания (3)

Пример. Функция `NextId` генерирует новый уникальный идентификатор:

```
NextId {
  e.Prefix
    , <Dg NextIdCounter>
    : {
      /* пусто */ = <Br NextIdCounter '=' 0> <NextId e.Prefix>;

        s.N
          = <Implode e.Prefix <Symb s.N>>
            <Br NextIdCounter '=' <+ s.N 1>>;
      };
}
```

Использование функций закапывания-выкапывания (4)

Другой вариант реализации `NextId`:

```
NextId {
  e.Prefix
    , <Dg NextIdCounter> : s.N
    = <Implode e.Prefix <Symb s.N>>
      <Br NextIdCounter '=' <+ s.N 1>>;
}

...

$ENTRY Go {
  = <Br NextIdCounter '=' 0>
    ...
}
```

Модули

Запуск программ

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 (1)

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.

Модули и функция Mu (2)

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>;
}
```

Модули и функция Mu (3)

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 */;
}
```

Примечание. 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

Модули и функция Mu (4)

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>>>>
    >
}
```

Модули и функция Mu (5)

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`.

Модули и функция Mu (6)

Рекомендуется функции, помеченные $ENTRY только ради вызова извне при помощи Mu называть как имяфайла_ИмяФункции:

```
$EXTERN Map;

Split {
  …
}

$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.

Библиотека `LibraryEx`

Библиотека `LibraryEx` (1)

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

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

Документация к ней:

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

Библиотека `LibraryEx` (2)

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.

Библиотека `LibraryEx` (3)

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>`.

Библиотека LibraryEx (4)

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

e.Error ::= (s.Line s.Col) e.Message
s.Line, s.Col ::= s.NUMBER
e.Message ::= s.CHAR*
```

Функция аналогична LoadExpr, но при синтаксической ошибке не завершает программу, а формирует сообщение об ошибке.

Библиотека LibraryEx (5)

Пример использования LoadExpr. Напишем программу, которая считывает из файла формулу логики высказываний и вычисляет нормальную форму отрицания:

```
*$FROM LibraryEx
$EXTERN ArgList, Map, LoadExpr;

$ENTRY Go {
  = <Main <ArgList>>
}

Main {
  (e.ProgName) = <nnf_Process ('nnf-example.txt')>;
  (e.ProgName) e.FileNames = <Map nnf_Process e.FileNames>;
}

$ENTRY nnf_Process {
  (e.FileName) = <Prout <NNF <LoadExpr e.FileName>>>;
}

NNF { /* см. предыдущие слайды */ }
```