Разработка программ с помощью Objective Caml
Developing Applications with Objective Caml

Emmanuel Chailloux
Pascal Manoury
Bruno Pagano

18 сентября 2012 г.

Оглавление

Глава 1

Как заполучить Objective CAML

Часть I

Основы языка

Глава 2

Функциональное программирование

2.1 Введение

Первый язык функционального программирования LISP появился в конце 1950, в тот же момент, что и Fortran — один из первых императивных языков. Оба этих языка существуют и по сей день, хотя они немало изменились. Область их применения: вычислительные задачи для Фортрана и символьные (symbolic) для Lisp. Интерес к функциональному программированию состоит в простоте написания программ, где под программой подразумевается функция, применённая к аргументам. Она вычисляет результат, который возвращается как вывод программы. Таким образом можно с лёгкостью комбинировать программы: вывод одной, становится входным аргументом для другой.

Функциональное программирование основывается на простой модели вычислений, состоящей из трёх конструкций: переменные, определение функции и её применение к какому–либо аргументу. Эта модель, называемая λ–исчисление, была введена Alonzo Church в 1932, ещё до появления первых компьютеров. В λ–исчислении любая функция является переменной, так что она может быть использована как входной параметр другой функции, или возвращена как результат другой. Теория λ–исчисления утверждает что все то, что вычисляемо может быть представлено этим формализмом. Однако, синтаксис этой теории слишком ограничен, чтобы его можно было использовать его как язык программирования. В связи с этим к λ–исчислению были добавлены базовые типы (например, целые числа или строки символов), операторы для этих типов, управляющие структуры и объявление позволяющие именовать переменные или

функции, и в частности рекурсивные функции.

Существует разные классификации языков функционального программирования. Мы будем различать их по двум характеристикам, которые нам кажутся наиболее важными:

- без побочных эффектов (чистые), или с побочным эффектом (не чистые): чистый язык — это тот, в котором не существует изменения состояния. Все есть вычисление, и как оно происходит, нас не интересует. Не чистые языки, такие как Caml или ML, имеют императивные особенности, такие как изменение состояния. Они позволяют писать программы в стиле близкому к Фортрану, в котором важен порядок вычисления выражений.

- язык типизирован динамически или статически: типизация необходима для проверки соответствия аргумента, переданного функции, типу формального параметра. Это проверка может быть выполнена во время выполнения программы. В этом случае типизация называется динамической. В случае ошибки программа будет остановлена, как это происходит в Lisp. В случае статической типизации проверка осуществляется во время компиляции, то есть до выполнения программы. Таким образом она (проверка) не замедлит программу во время выполнения. Эта типизация используется в ML и в его диалектах, таких как Objective CAML. Только правильно типизированные программы, то есть успешно прошедшие проверку типов, могут быть скомпилированы и затем выполнены.

2.2 План главы

В этой главе представлены базовые элементы функциональной части языка Objective CAML, а именно: синтаксис, типы и механизм исключений. После этого вводного курса мы сможем написать нашу первую программу.

В первом разделе описаны основы языка, начиная с базовых типов и функций. Затем мы рассмотрим структурные и функциональные типы. После этого мы рассмотрим управляющие структуры, а также локальные и глобальные объявления. Во втором разделе мы обсудим определение типов для создания структур и механизм сопоставления с образцом, который используется для доступа к этим структурам. В третьем разделе рассматривается выводимый тип функций и область их применения, после чего следует описание механизма исключений. Четвёртый

раздел объединяет введённые понятия в простой пример программы–калькулятора.

2.3 Функциональное ядро Objective CAML

Как любой другой язык функционального программирования, Objective CAML — это язык выражений, состоящий в основном из создания функций и их применения. Результатом вычисления одного из таких выражений является значение данного языка (value in the language) и выполнение программы заключается в вычислении всех выражений из которых она состоит.

2.3.1 Значения, функции и базовые типы

В Objective CAML определены следующие типы: целые числа, числа с плавающей запятой, символьный, строковый и логический.

Числа

Различают целые `int` и числа с плавающей запятой `float`. Objective CAML следует спецификации IEEE 754 для представления чисел с плавающей запятой двойной точности. Операции с этими числами описаны в таблице 2.1. Если результат целочисленной операции выходит за интервал значений типа `int`, то это не приведёт к ошибке. Результат будет находиться в интервале целых чисел, то есть все действия над целыми ограниченны операцией `modulo` с границами интервала.

Разница между целыми числами и числами с плавающей запятой.

Значения разных типов, таких как `float` и `int`, не могут сравниваться между собой напрямую. Для этого существует функции перевода одного типа в другой (`float_of_int` и `int_of_float`).

```
# 2 = 2.0;;
Error: This expression has type float but an expression was expected of
    type
         int
# 3.0 = float_of_int 3;;
- : bool = true
```

Таблица 2.1: Операции над числами

целые	плавающие
+ сложение	+. сложение
- вычитание и унарный минус	-. вычитание и унарный минус
* умножение	*. умножение
/ деление	/. деление
`mod` остаток целочисленного деления	** возведение в степень

```
# 1 ;;
- : int = 1
# 1 + 2 ;;
- : int = 3
# 9 / 2 ;;
- : int = 4
# 11 mod 3 ;;
- : int = 2
(* limits of the representation
   *)
(* of integers *)
# 2147483650 ;;
- : int = 2
```

```
# 2.0 ;;
- : float = 2
# 1.1 +. 2.2 ;;
- : float = 3.3
# 9.1 /. 2.2 ;;
- : float = 4.13636363636
# 1. /. 0. ;;
- : float = inf
(* limits of the representation
   *)
(* of floating-point numbers *)
# 222222222222.11111 ;;
- : float = 222222222222
```

Аналогично, операции над целыми числами и числами с плавающей запятой различны:

```
# 3 + 2;;
- : int = 5
# 3.0 +. 2.0;;
- : float = 5.
# 3.0 + 2.0;;
Error: This expression has type float but an expression was expected of
    type
         int
# sin 3.14159;;
- : float = 2.65358979335273e-06
```

Неопределенный результат, например получаемый при делении на ноль, приведет к возникновению исключения (см. **??**, стр. **??**), которое остановит вычисление. Числа с плавающей запятой имеют специальные значения для бесконечных величин (`Inf`) и для не определённого результата (`NaN` [1]). Основные операции над этими числами приведены в таблице 2.2

Таблица 2.2: Функции над числами с плавающей запятой

`ceil`	`cos` косинус
`floor`	`sin` синус
`sqrt` — квадратный корень	`tan` тангенс
`exp` — экспонента	`acos` арккосинус
`log` — натуральный логарифм	`asin` арксинус
`log10` — логарифм по базе 10	`atan` арктангенс

```
# ceil 3.4 ;;
- : float = 4.
# floor 3.4 ;;
- : float = 3.
# ceil (-.3.4) ;;
- : float = -3.
# floor (-.3.4) ;;
- : float = -4.
```

```
# sin 1.57078 ;;
- : float = 0.999999999866717837
# sin (asin 0.707) ;;
- : float = 0.707
# acos 0.0 ;;
- : float = 1.57079632679489656
# asin 3.14 ;;
- : float = nan
```

Символы и строки

Символы, тип `char`, соответствуют целым числам в интервале от 0 до 255, первые 128 значений соответствуют кодам ASCII. Функция `char_of_int` и `int_of_char` преобразуют один тип в другой. Строки, тип `string` — это последовательность символов определенной длинны (не длиннее $224^{24} - 6$). Оператором объединения строк (конкатенации) является шапка $\wedge$. Следующие функции необходимы для перевода типов `int_of_string`, `string_of_int`, `string_of_float` и `float_of_string`.

```
# 'B' ;;
```

[1] Not a Number

```
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "est une îchane" ;;
- : string = "est une cha\195\174ne"
# (string_of_int 1987) ^ " est l'éanne de la écration de CAML" ;;
- : string = "1987 est l'ann\195\169e de la cr\195\169ation de
   CAML"
```

Если строка состоит из цифр, то мы не сможем использовать ее в численных операциях, не выполнив явного преобразования.

```
# "1999" + 1 ;;
Error: This expression has type string but an expression was expected of
    type
         int
# (int_of_string "1999") + 1 ;;
- : int = 2000
```

В модуле `String` собрано много функций для работы со строками (стр. **??**)

Булевый тип

Значение типа `boolean` принадлежит множеству состоящему из двух элементов: `true` и `false`. Основные операторы описаны в таблице 2.3.1. По историческим причинам операторы `and` и `or` имеют две формы.

Таблица 2.3: Булевы операторы

not — отрицание	
&& логическое и	& синоним &&
|| логическое или	`or` синоним |

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false
```

Операторы && и || или их синонимы, вычисляют аргумент слева и в зависимости от его значения, вычисляют правый аргумент. Они могут быть переписаны в виде условной структуры (см. **??** стр. **??**).

Таблица 2.4: Операторы сравнения и равенства

= равенство структурное	< меньше
== равенство физическое	> больше
<> отрицание =	<= меньше или равно
!= отрицание ==	>= больше или равно

Операторы сравнения и равенства описаны в таблице 2.3.1. Это полиморфные операторы, то есть они применимы как для сравнения двух целых, так и двух строк. Единственное ограничение это то что операнды должны быть одного типа (см. **??** стр. **??**).

```
# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "one" < "two" ;;
- : bool = true
# 0 < '0' ;;
Error: This expression has type char but an expression was expected of
    type
         int
```

Структурное равенство при проверке двух переменных сравнивает значение полей структуры, тогда как физическое равенство проверяет занимают ли эти переменные одно и то же место в памяти. Оба оператора возвращают одинаковый результат для простых типов: `boolean`, `char`, `int` и константные конструкторы (см. **??** стр. **??**).

Осторожно

Числа с плавающей запятой и строки рассматриваются как структурные типы.

Объединения

Тип `unit` определяет множество из всего одного элемента, обозначается: ()

```
# () ;;
- : unit = ()
```

Это значение будет широко использоваться в императивных программах (см. **??** стр. **??**), в функциях с побочным эффектом. Функции, результат которых равен (), соответствуют понятию процедуры, которое отсутствует в Objective CAML, так же как и аналог типа `void` в языке C.

Декартово произведение, кортежи

Значения разных типов могут быть сгруппированы в кортежи. Значения из которых состоит кортеж разделяются запятой. Для конструкции кортежа, используется символ `*`. `int*string` есть кортеж, в котором первый элемент целое число и второй строка.

```
# ( 12 , "October" ) ;;
- : int * string = (12, "October")
```

Иногда мы можем использовать более простую форму записи.

```
# 12 , "October" ;;
- : int * string = (12, "October")
```

Функции `fst` и `snd` дают доступ первому и второму элементу соответственно.

```
# fst ( 12 , "October" ) ;;
- : int = 12
# snd ( 12 , "October" ) ;;
- : string = "October"
```

Эти обе функции полиморфные, входной аргумент может быть любого типа.

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst ( "October", 12 ) ;;
- : string = "October"
```

Тип `int*char*string` — это триплет, в котором первый элемент типа `int`, второй `char`, а третий — `string`.

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = (65, 'B', "ascii")
```

Осторожно

Если аргумент функций `fst` и `snd` не пара, а другой n–кортеж, то мы получим ошибку.

```
# snd ( 65 , 'B' , "ascii" ) ;;
Error: This expression has type int * char * string
       but an expression was expected of type 'a * 'b
```

Существует разница между парой и триплетом: тип `int*int*int` отличен от `(int*int)*int` и `int*(int*int)`. Методы доступа к элементам триплета (и других кортежей) не определены в стандартной библиотеке. В случае необходимости мы используем сопоставление с образцом (см. **??**).

Списки

Значения одного и того же типа могут быть объединены в списки. Список может быть либо пустым, либо содержать однотипные элементы.

```
# [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "two" ; 3 ] ;;
Error: This expression has type string but an expression was expected of
    type
         int
```

Для того чтобы добавить элемент в начало списка существует следующая функция в виде инфиксного оператора :: — аналог `cons` в Caml.

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

Для объединения (конкатенации) списков существует инфиксный оператор: @.

```
# [ 1 ] @ [ 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; 2 ] @ [ 3 ] ;;
- : int list = [1; 2; 3]
```

Остальные функции манипуляции списками определены в библиотеке `List`. Функции `hd` и `tl` дают доступ к первому и последнему элементу списка.

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Exception: Failure "hd".
```

В последнем примере получить первый элемент пустого списка действительно «сложно», поэтому возбуждается исключение (см. **??**).

2.3.2 Структуры условного контроля

Одна из структур контроля необходимая в каждом языка программирования — условный оператор.

Синтаксис:

```
if expr1 then expr2 else expr3
```

Тип выражения `expr1` равен `bool`. Выражения `expr2` и `expr3` должны быть одного и того же типа.

```
# if 3=4 then 0 else 4 ;;
- : int = 4
# if 3=4 then "0" else "4" ;;
- : string = "4"
# if 3=4 then 0 else "4";;
Error: This expression has type string but an expression was expected of
    type
         int
```

Замечание

Ветка `else` может быть опущена, в этом случае будет подставлено значение по умолчанию равное `else ()`, в соответствии с этим выражение `expr2` должно быть типа `unit` (см. **??** стр. **??**).

2.3.3 Объявление значений

Определение связывает имя со значением. Различают глобальные и локальные определения. В первом случае, объявленные имена видны во всех выражениях, следуют за ним, во втором — имена доступны только в текущем выражении. Мы также можем одновременно объявить несколько пар имя-значение.

Глобальные объявления

Синтаксис:

```
let name = expr ;;
```

Глобальное объявление определяет связь имени `nom` со значением выражения `expr`, которое будет доступно всем следующим выражениям.

```
# let yr = "1999" ;;
val yr : string = "1999"
# let x = int_of_string(yr) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let new_yr = string_of_int (x + 1) ;;
val new_yr : string = "2000"
```

Одновременное глобальное объявление

Синтаксис:

```
let nom1 = expr1
and nom2 = expr2
:
and nomn = exprn;;
```

При одновременном объявлении переменные будут известны только к концу всех объявлений.

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
Error: Unbound value z
```

Можно сгруппировать несколько глобальных объявлений в одной фразе, вывод типов и значений произойдёт к концу фразы, отмеченной «;;». В данном случае объявления будут вычислены по порядку.

```
# let x = 2
  let y = x + 3 ;;
val x : int = 2
val y : int = 5
```

Глобальное объявление может быть скрыто локальным с тем же именем (см. ?? стр. ??).

Локальное объявление

Синтаксис:

```
let nom = expr1 in expr2;;
```

Имя `nom` связанное с выражением `expr1` известно только для вычисления `expr2`.

```
# let xl = 3 in xl * xl ;;
- : int = 9
```

Локальное объявление, которое связывает `xl` со значением `3`, существует только в ходе вычисления `xl*xl`.

```
# xl ;;
Error: Unbound value xl
```

Локальное объявление скрывает любое глобальные с тем же именем, но как только мы выходим из блока в котором была оно определено, мы находим старое значение связанное с этим именем.

```
# let x = 2 ;;
val x : int = 2
# let x = 3 in x * x ;;
- : int = 9
# x * x ;;
- : int = 4
```

Локальное объявление — это обычное выражение, соответственно оно может быть использовано для построения других выражений.

```
# (let x = 3 in x * x) + 1 ;;
- : int = 10
```

Локальные объявления так же могут быть одновременными.

```
let  name1 = expr1
and  name2 = expr2
:
and  namen = exprn
in  expr ;;
```

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
- : float = 5.
# b ;;
Error: Unbound value b
```

2.3.4 Функциональное выражение, функции

Функциональное выражение состоит из параметра и тела. Формальный параметр — это имя переменной, а тело — это выражение. Обычно говорят что формальный параметр является абстрактным, по этой причине функциональное выражение тоже называется абстракцией.

Синтаксис:

```
function p -> expr
```

Таким образом функция возведения в квадрат будет выглядеть так:

```
# function x -> x*x ;;
- : int -> int = <fun>
```

Objective CAML сам определяет тип. Функциональный тип `int->int` это функция с параметром типа `int`, возвращающая значение типа `int`.

Функция с одним аргументом пишется как функция и аргумент следующий за ней.

```
# (function x -> x * x) 5 ;;
- : int = 25
```

Вычисление функции состоит в вычисление её тела, в данном случае `x*x`, где формальный параметр `x`, заменён значением аргумента (эффективным параметром), здесь он равен 5.

При конструкции функционального выражения `expr` может быть любым выражением, в частности функциональным.

```
# function x -> (function y -> 3*x + y) ;;
- : int -> int -> int = <fun>
```

Скобки не обязательны, мы можем писать просто:

```
# function x -> function y -> 3*x + y ;;
- : int -> int -> int = <fun>
```

В простом случае мы скажем, что функция ожидает два аргумента целого типа на входе и возвращает значение целого типа. Но когда речь идёт о функциональном языке, таком как Objective CAML, то скорее это соответствует типу функции с входным аргументом типа `int` и возвращающая функциональное значение типа `int->int`:

```
# (function x -> function y -> 3*x + y) 5 ;;
- : int -> int = <fun>
```

Естественно, мы можем применить это функциональное выражение к двум аргументам. Для этого напишем:

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Когда мы пишем `f a b`, подразумевается применение `(f a) к b`.

Давайте подробно рассмотрим выражение

```
(function x -> function y -> 3*x + y) 4 5
```

Для того чтобы вычислить это выражение, необходимо сначала вычислить значение

```
(function x -> function y -> 3*x + y) 4
```

что есть *функциональное выражение* равное

```
function y -> 3*4 + y
```

в котором `x` заменён на 4 в выражении `3 * x + y`. Применяя это значение (являющееся функцией) к 5, мы получаем конечное значение `3 * 4 + 5 = 17`:

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Арность функции

Арностью функции называется число аргументов функции. По правилам, унаследованным из математики, аргументы функции задаются в скобках после имени функции. Мы пишем: `f(4,5)`. Как было указанно

ранее, в Objective CAML мы чаще используем следующий синтаксис: `f 4 5`. Естественно, можно написать функциональное выражение применимое к `(4,5)`:

```
# function (x,y) -> 3*x + y ;;
- : int * int -> int = <fun>
```

Но в данном случае, функция ожидает не два аргумента, а один; тип которого пара целых. Попытка применить два аргумента к функции ожидающей пару или наоборот, передать пару функции для двух аргументов приведёт к ошибке:

```
# (function (x,y) -> 3*x + y) 4 5 ;;
Error: This function is applied to too many arguments;
maybe you forgot a ';'
# (function x -> function y -> 3*x + y) (4, 5) ;;
Error: This expression has type int * int
       but an expression was expected of type int
```

Альтернативный синтаксис

Существует более компактная форма записи функций с несколькими аргументами, которая дошла к нам из старых версий *Caml*. Выглядит она так:

Синтаксис

```
fun p1 ... pn -> expr
```

Это позволяет не повторять слово `function` и стрелки. Данная запись эквивалентна

```
function p1 -> -> function pn -> expr
# fun x y -> 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y -> 3*x + y) 4 5 ;;
- : int = 17
```

Эту форму можно часто встретить в библиотеках идущих в поставку с Objective CAML.

Замыкание

Objective CAML рассматривает функциональное выражение также как любое другое. Значение возвращаемое функциональным выражением на-

зывается замыканием. Каждое выражение Objective CAML вычисляется в окружении состоящем из соответствий имя–значение, которые были объявлены до вычисляемого выражения. Замыкание может быть описано как триплет, состоящий из имени формального параметра, тела функции и окружения выражения. Нам необходимо хранить это окружение, поскольку в теле функции кроме формальных параметров могут использоваться другие переменные. В функциональном выражении эти переменные называются свободными, нам понадобится их значение в момент применения функционального выражения.

```
# let m = 3 ;;
val m : int = 3
# function x -> x + m ;;
- : int -> int = <fun>
# (function x -> x + m) 5 ;;
- : int = 8
```

В случае когда применение замыкания к аргументу возвращает новое замыкание, оно (новое замыкание) получает в своё окружение все необходимые связи для следующего применения. Раздел **??** подробно рассматривает эти понятия. В главе **??**, а так же в главе **??** мы вернёмся к тому как замыкание представляется в памяти.

До сих пор рассмотренные функциональные выражения были *анонимными*, однако мы можем дать им имя. Объявление функциональных значений

Объявление функциональных значений

Функциональное значение объявляется так же как и другие, при помощи конструктора `let`

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x -> function y -> 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

Для упрощения записи, можно использовать следующий синтаксис:
Синтаксис:

```
let nom p1...pn=expr
```

что эквивалентно:

```
let nom=function p1−>−>function pn−> expr
```

Следующие объявления `succ` и g эквивалентны предыдущим:

```
# let succ x = x + 1 ;;
val succ : int −> int = <fun>
# let g x y = 2∗x + 3∗y ;;
val g : int −> int −> int = <fun>
```

В следующем примере демонстрируется функциональная сторона Objective CAML, где функция `h1` получена применением `g` к аргументу. В данном случае мы имеем дело с частичным применением.

```
# let h1 = g 1 ;;
val h1 : int −> int = <fun>
# h1 2 ;;
− : int = 8
```

С помощью `g` мы можем определить другую функцию `h2` фиксируя значение второго параметра `y`:

```
# let h2 = function x −> g x 2 ;;
val h2 : int −> int = <fun>
# h2 1 ;;
− : int = 8
```

Объявление инфиксных функций

Некоторые бинарные функции могут быть использованы в инфиксной форме. Например при сложении двух целых мы пишем `3 + 5` для применения `+` к `3` и `5`. Для того чтобы использовать символ `+` как классическое функциональное значение, необходимо указать это, окружая символ скобками (`op`).

В следующем примере определяется функция `succ` используя `(+)`:

```
# (+) ;;
− : int −> int −> int = <fun>
# let succ = (+) 1 ;;
val succ : int −> int = <fun>
# succ 3 ;;
```

```
- : int = 4
```

Таким образом мы можем определить новые операторы, что мы и сделаем определив `++` для сложения двух пар целых.

```
# let (++) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;;
val ( ++ ) : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = (2, 3)
# c ++ c ;;
- : int * int = (4, 6)
```

Существуют однако ограничения на определение новых операторов, они должны содержать только *символы* (такие как `*,+,$`, etc.), исключая буквы и цифры. Следующие функции являются исключением:

```
or mod land lor lxor lsr asr
```

Функции высшего порядка

Функциональное значение (замыкание) может быть возвращено как результат, а так же передано как аргумент функции. Такие функции, берущие на входе или возвращающие функциональные значения, называются функциями высшего порядка.

```
# let h = function f -> function y -> (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>
```

Замечание Выражения группируются справа налево, но функциональные типы объединяются слева направо. Таким образом тип функции `h` может быть написан:

```
(int -> int) -> int -> int или(int -> int) -> (int -> int)
```

При помощи функций высшего порядка можно элегантно обрабатывать списки. К примеру функция `List.map` применяет какую-нибудь функцию ко всем элементам списка и возвращает список результатов.

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

Другой пример — функция `List.for_all` проверяет соответствуют ли элементы списка определённому критерию.

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.for_all (function n -> n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# List.for_all (function n -> n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

Видимость переменных

Для вычисления выражения, необходимо чтобы все используемые им переменные были определены, как, например, выражение `e` в определении

```
let p=e
```

Переменная `p` не известна в этом выражении, она может быть использована только в случае если `p` была объявлена ранее.

```
# let p = p ^ "-suffixe" ;;
Error: Unbound value p
# let p = "éprfixe" ;;
val p : string = "pr\195\169fixe"
# let p = p ^ "-suffixe" ;;
val p : string = "pr\195\169fixe-suffixe"
```

В Objective CAML переменные связаны статически. При применении замыкания используется окружение в момент её (замыкания) объявления (статическая видимость), а не в момент её применения (динамическая видимость)

```
# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = (10, 10, 20)
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = (1000, 10, 1010)
```

В функции `k` имеется свободная переменная `p`, которая была определена в глобальном окружении, поэтому определение `k` принято. Связь между именем `p` и значением 10 в окружении замыкания `k` статическая, то есть не зависит от последнего определения `p`.

Рекурсивное объявление

Объявление переменной называется рекурсивным, если оно использует свой собственный идентификатор в своём определении. Эта возможность часто используется для определения рекурсивных функций. Как мы видели ранее, `let` не позволяет делать это, поэтому необходимо использовать специальный синтаксис:

```
let rec nom = expr ;;
```

Другой способ записи для функции с аргументами:

```
let rec nom p1 ... pn = expr ;;
```

Определим функцию `sigma` вычисляющую сумму целых от `0` до значения указанного аргументом:

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

Как заметил читатель, эта функция рискует «не закончится» если входной аргумент меньше 0.

Обычно, рекурсивное значение — это функция, компилятор не принимает некоторые рекурсивные объявления, значения которых не функциональные.

```
# let rec x = x + 1 ;;
Error: This kind of expression is not allowed as right-hand side of 'let
    rec'
```

Как мы увидим позднее, такие определения все таки возможны в некоторых случаях (см. **??** стр. **??**).

Объявление `let rec` может быть скомбинировано с `and`. В этом случае функции определённые на одном и том же уровне, видны всем остальным. Это может быть полезно при декларации взаимно рекурсивных функций.

```
# let rec pair n = (n<>1) && ((n=0) or (impair (n-1))) and impair n
    = (n<>0) &&
```

```
((n=1) or (pair (n−1)));;
val pair : int −> bool = <fun>
val impair : int −> bool = <fun>
# pair 4 ;;
− : bool = true
# impair 5 ;;
− : bool = true
```

По тому же принципу, локальные функции могут быть рекурсивными. Новое определение функции `sigma` проверяет корректность входного аргумента, перед тем как посчитать сумму локальной функцией `sigma_rec`.

```
# let sigma x = let rec sigma_rec x = if x = 0 then 0 else x +
    sigma_rec (x−1)
in if (x<0) then "erreur : argument negatif" else "sigma = " ^ (
    string_of_int
(sigma_rec x)) ;;
val sigma : int −> string = <fun>
```

Замечание

Мы вынужденны были определить возвращаемый тип как `string`, поскольку необходимо чтобы он был один и тот же, независимо от входного аргумента, отрицательного или положительного. Какое значение должна вернуть `sigma` если аргумент больше нуля? Далее, мы увидим правильный способ решения этой проблемы (см. **??** стр. **??**).

2.3.5 Полиморфизм и ограничение типа

Некоторые функции выполняют одни и те же инструкции независимо от типа аргументов. К примеру, для создание пары из двух значений нет смысла определять функции для каждого известного типа. Другой пример, доступ к первому полю пары не зависит от того, какого типа это поле.

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a −> 'b −> 'a * 'b = <fun>
# let p = make_pair "papier" 451 ;;
val p : string * int = ("papier", 451)
# let a = make_pair 'B' 65 ;;
val a : char * int = ('B', 65)
# fst p ;;
− : string = "papier"
```

```
# fst a ;;
- : char = 'B'
```

Функция, для которой не нужно указывать тип входного аргумента или возвращаемого значения называется полиморфной. Синтезатор типов, включённый в компилятор Objective CAML находит наиболее общий тип для каждого выражения. В этом случае Objective CAML использует переменные, здесь они обозначены как `'a` и `'b`, для указания общих типов. Эти переменные конкретизируются типом аргумента в момент применения функции.

При помощи полиморфных функций, мы получаем возможность написания универсального кода для любого типа переменных, сохраняя при этом надёжность статической типизации. Действительно, несмотря на то что `make_paire` полиморфная, значение созданное (`make_paire '6' 65`) имеет строго определённый тип, который отличен от (`make_paire "paire"451`). Проверка типов реализуется в момент компиляции, таким образом универсальность кода никак не сказывается на эффективности программы.

Пример функций и полиморфных значений

В следующем примере приведена полиморфная функция с входным параметром функционального типа.

```
# let app = function f -> function x -> f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

Мы можем применить её к функции `impaire`, которая была определена ранее.

```
# app impair 2;;
- : bool = false
```

Функция тождества (`id`) возвращает полученный аргумент.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id 1 ;;
- : int = 1
```

Следующая функция, `compose` принимает на входе две функции и ещё один аргумент, к которому применяет две первые.

```
# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
# let add1 x = x+1 and mul5 x = x*5 in compose mul5 add1 9 ;;
- : int = 50
```

Как мы видим, тип результата возвращаемого `g` должен быть таким же как и тип входного аргумента `f`.

Не только функциональные значения могут быть полиморфными, проиллюстрируем это на примере пустого списка.

```
# let l = [] ;;
val l : 'a list = []
```

Следующий пример иллюстрирует тот факт, что создание типов основывается на разрешении ограничений, накладываемых применением функций, а вовсе не на значениях, полученных в процессе выполнения.

```
# let q = List.tl [2] ;;
val q : int list = []
```

Здесь тип равен `List.tl 'a list->'a list`, то есть эта функция применяется к списку целых и возвращает список целых. Тот факт что в момент выполнения возвращён пустой список, не влияет на его тип.

Objective CAML создаёт параметризованные типы для каждой функции, которые не зависят от её аргументов. Этот вид полиморфизма называется *параметрическим полиморфизмом* [2].

Ограничение типа

Синтезатор типа Objective CAML образует самый общий тип и иногда бывает необходимо уточнить тип выражения.

Синтаксис ограничения типа следующий:

```
(expr:t)
```

В этом случае синтезатор типа воспользуется этим ограничением при конструкции типа выражения. Использование ограничения типа позволяет

- сделать видимым тип параметра функции
- запретить использование функции вне свой области применения

[2] Некоторые встроенные функции не подчиняются этому правилу, особенно функция структурного равенства (`=`), которая является полиморфной (её тип `'a -> 'a -> bool`), но она исследует структуру своих аргументов для проверки равенства.

- уточнить тип выражения, это окажется необходимым в случае физически изменяемых значений (см. **??** стр. **??**).

Рассмотрим использование ограничения типа.

```
# let add (x:int) (y:int) = x + y ;;
val add : int -> int -> int = <fun>
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int : int -> int -> int * int = <fun>
# let compose_fn_int (f : int -> int) (g : int -> int) (x:int) = compose f
    g x;;
val compose_fn_int : (int -> int) -> (int -> int) -> int -> int = <
   fun>
# let nil = ([] : string list);;
val nil : string list = []
# 'H'::nil;;
Error: This expression has type string list
       but an expression was expected of type char list
```

Это ограничение полиморфизма позволяет лучше контролировать тип выражений, тем самым ограничивая тип определённый системой. В таких выражениях можно использовать любой определённый тип.

```
# let llnil = ([] : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
- : int list list = [[1; 2; 3]]
```

`llint` является списком списков любого типа.

В данном случае подразумевается ограничение типа, а не явная типизация заменяющая тип, который определил Objective CAML. В частности, мы не можем обобщить тип, более чем это позволяет вывод типов Objective CAML.

```
# let add_general (x:'a) (y:'b) = add x y ;;
val add_general : int -> int -> int = <fun>
```

Ограничения типа будут использованы в интерфейсах модулей **??**, а так же в декларации классов **??**

2.3.6 Примеры

В этом параграфе мы приведём несколько примеров функций. Большинство из них уже определены в Objective CAML, мы делаем это только в «педагогических» целях.

Тест остановки рекурсивных функций реализован при помощи проверки, имеющей стиль более близкий к `Lisp`. Мы увидим как это сделать в стиле ML (см. **??**).

Размер списка

Начнём с функции проверяющей пустой список или нет.

```
# let null l = (l=[]);;
val null : 'a list -> bool = <fun>
```

Определим функцию `size` вычисления размера списка (т.е. число элементов).

```
# let rec size l = if null l then 0 else 1+(size(List.tl l));;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```

Функция `size` проверяет список: если он пуст возвращает 0, иначе прибавляет 1 к длине остатка списка.

Итерация композиций (Iteration of composition)

Выражение `iterate n f` вычисляет f^n, соответствующее применение функции `f n` раз.

```
# let rec iterate n f =
    if n = 0 then (function x -> x)
    else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

Функция `iterate` проверяет аргумент n на равенство нулю, если аргумент равен, то возвращаем функцию идентичности, иначе возвращаем композицию `f` с итерацией `f n-1` раз.

Используя `iterate` можно определить операцию возведения в степень, как итерацию умножения.

```
# let rec power i n =
  let i_times = ( * ) i in
    iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
```

```
- : int = 256
```

Функция `power` повторяет n раз функциональное выражение `i_times`, затем применяет этот результат к 1, таким образом мы получаем n-ю степень целого числа.

Таблица умножения

Напишем функцию `multab`, которая вычисляет ряд таблицы умножения соответствующую целому числу переданному в аргументе.

Для начала определим функцию `apply_fun_list`. Пусть `f_list` список функций, тогда вызов `apply_fun_list x f_list` возвращает список результатов применения каждого элемента списка `f_list` к `x`.

```
# let rec apply_fun_list x f_list =
  if null f_list then []
  else ((List.hd f_list) x)::(apply_fun_list x (List.tl f_list)) ;;
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list = <fun>
# apply_fun_list 1 [(+) 1;(+) 2;(+) 3] ;;
- : int list = [2; 3; 4]
```

Функция `mk_mult_fun_list` возвращает список функций умножающих их аргумент на i, `0<=i<=n`.

```
# let mk_mult_fun_list n =
    let rec mmfl_aux p =
      if p = n then [ ( * ) n ]
      else (( * ) p) :: (mmfl_aux (p+1))
    in (mmfl_aux 1) ;;
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

Подсчитаем ряд для 7

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10) ;;
val multab : int -> int list = <fun>
# multab 7 ;;
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

Итерация в списке

Вызов функции `fold_left f a [e1; e2; ...; en]` возвращает `f...(f (f a e1) e2) ... en`, значит получаем n применений.

```
# let rec fold_left f a l =
    if null l then a
    else fold_left f ( f a (List.hd l)) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

С помощью функции `fold_left` можно компактно определить функцию вычисления суммы элементов списка целых чисел.

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

Или, например, конкатенация элементов списка строк.

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

Глава 3

Императивное программирование

Глава 4

Функциональный и императивный стиль

Глава 5

Графический интерфейс

Глава 6

Приложения

Часть II

Средства разработки

Глава 7

Компиляция и переносимость

Глава 8

Библиотеки

Глава 9

Автоматический сборщик мусора

Глава 10

Средства анализа программ

Глава 11

Средства лексического и синтаксического анализа

11.1 Введение

Определение и реализация средств лексического и синтаксического анализа являлись важным доменом исследования в информатике. Эта работа привела к созданию генераторов лексического и синтаксического анализа `lex` и `yacc`. Команды `camllex camlyacc`, которые мы представим в этой главе, являются их достойными наследниками. Два указанных инструмента стали de–facto стандартными, однако существуют другие средства, как например потоки или регулярные выражения из библиотеки `Str`, которые могут быть достаточны для простых случаев, там где не нужен мощный анализ.

Необходимость подобных инструментов особенно чувствовалась в таких доменах как компиляция языков программирования. Однако и другие программы могут с успехом использовать данные средства: базы данных, позволяющие определять запросы или электронная таблица, где содержимое ячейки можно определить как результат какой–нибудь формулы. Проще говоря, любая программа, в которой взаимодействие с пользователем осуществляется при помощи языка, использует лексический и синтаксический анализ.

Возьмём простой случай. Текстовый формат часто используется для хранения данных, будь то конфигурационный системный файл или данные табличного файла. Здесь, для использования данных, необходим лексический и синтаксический анализ.

Обобщая, скажем что лексический и синтаксический анализ преобразует линейный поток символов в данные с более богатой структурой:

последовательность слов, структура записи, абстрактное синтаксическое дерево программы и т.д.

У каждого языка есть словарный состав (лексика) и грамматика, которая описывает каким образом эти составные объединяются (синтаксис). Для того, чтобы машина или программа могли корректно обрабатывать язык, этот язык должен иметь точные лексические и синтаксические правила. У машины нет «тонкого чувства» для того чтобы правильно оценить двусмысленность натуральных языков. По этой причине к машине необходимо обращаться в соответствии с чёткими правилами, в которых нет исключений. В соответствии с этим, понятия лексики и семантики получили формальные определения, которые будут кратко представлены в данной главе.

11.2 План главы

Данная глава знакомит нас со средствами лексического и синтаксического анализа, которые входят в дистрибутив Objective CAML. Обычно синтаксический анализ следует за лексическим. В первой части мы узнаем о простом инструменте лексического анализа из модуля `Genlex`. После этого ознакомимся с формализмом рациональных выражений и тем самым рассмотрим более детально определение множества лексических единиц. А так же проиллюстрируем их реализацию в модуле `Str` и инструменте `ocamllex`. Во второй части мы определим грамматику и рассмотрим правила создания фраз языка. После этого рассмотрим два анализа фраз: восходящий и нисходящий. Они будут проиллюстрированы использованием `Stream` и `ocamlyacc`. В приведённых примерах используется контекстно–независимая грамматика. Здесь мы узнаем как реализовать контекстный анализ при помощи Stream. В третьей части мы вернёмся к интерпретатору BASIC (см. стр **??**) и при помощи `ocamllex` и `ocamlyacc` добавим лексические и синтаксические функции анализа языка.

11.3 Лексика

Синтаксический анализ это предварительный и необходимый этап обработки последовательностей символов: он разделяет этот поток в последовательность слов, так называемых лексические единицы или лексемы.

11.3.1 Модуль `Genlex`

В данном модуле имеется элементарное средство для анализа символьного потока. Для этого используются несколько категорий предопределённых лексических единиц. Эти категории различаются по типу:

```
# type token = Kwd of string
    | Ident of string
    | Int of int
    | Float of float
    | String of string
    | Char of char ;;
```

Таким образом мы можем распознать в потоке символов целое число (конструктор `Int`) и получить его значение (аргумент конструктора `int`). Распознаваемые символы и строки подчиняются следующим общепринятым соглашением: строка окружена символами ("), а символ окружён (’). Десятичное число представлено либо используя запись с точкой (например 0.01), либо с мантиссой и экспонентой (на пример 1E-2). Кроме этого остались конструкторы `Kwd` и `Ident`.

Конструктор `Ident` предназначен для определения идентификаторов. Идентификатором может быть имя переменной или функции языка программирования. Они состоят из любой последовательности букв и цифр, могут включать символ подчёркивания (_) или апостроф (’). Данная последовательность не должна начинаться с цифры. Любая последовательность следующих операндов тоже будет считаться идентификатором: +,*,> или -. И наконец, конструктор `Kwd` определяет категорию специальных идентификаторов или символов.

Категория ключевых слова — единственная из этого множества, которую можно сконфигурировать. Для того, чтобы создать лексический анализатор, воспользуемся следующей конструкцией, которой необходимо передать список ключевых слов на место первого аргумента.

```
# Genlex.make_lexer ;;
- : string list -> char Stream.t -> Genlex.token Stream.t = <fun>
```

Тем самым получаем функцию, которая принимает на вход поток символов и возвращает поток лексических единиц (с типом `token`).

Таким образом, мы без труда реализуем лексический анализатор для интерпретатора BASIC. Объявим множество ключевых слов:

```
# let keywords =
    [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN"; "-";
    "!"; "+"; "-"; "*"; "/"; "%";
```

```
    "="; "<"; ">"; "<="; ">="; "<>";
    "&"; "|" ] ;;
```

При помощи данного множества, определим функцию лексического анализа:

```
# let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l) ;;
val line_lexer : string -> Genlex.token Stream.t = <fun>
# line_lexer "LET x = x + y * 3" ;;
- : Genlex.token Stream.t = <abstr>
```

Приведённая функция `line_lexer`, из входящего потока символов создаёт поток соответствующих лексем.

11.3.2 Использование потоков

Мы также можем реализовать лексический анализ «в ручную» используя потоки.

В следующем примере определён лексический анализатор арифметических выражений. Функции lexer передаётся поток символов из которого она создаёт поток лексических единиц с типом `lexeme Stream.t` [1]. Символы пробел, табуляция и переход на новую строку удаляются. Для упрощения, мы не будем обрабатывать переменные и отрицательны целые числа.

```
# let rec spaces s =
    match s with parser
      [<'' ' ; rest >] -> spaces rest
      | [<''\t' ; rest >] -> spaces rest
      | [<''\n' ; rest >] -> spaces rest
      | [<>] -> ();;
val spaces : char Stream.t -> unit = <fun>
# let rec lexer s =
   spaces s;
   match s with parser
     [< ''(' >] -> [< 'Lsymbol "("; lexer s >]
     | [< '')' >] -> [< 'Lsymbol ")"; lexer s >]
     | [< ''+' >] -> [< 'Lsymbol "+" ; lexer s >]
     | [< ''-' >] -> [< 'Lsymbol "-" ; lexer s >]
     | [< ''*' >] -> [< 'Lsymbol "*" ; lexer s >]
     | [< ''/' >] -> [< 'Lsymbol "/" ; lexer s >]
```

[1] тип `lexeme` определён на стр. **??**

```
    | [< ''0'..'9' as c;
      i,v = lexint (Char.code c - Char.code('0')) >]
      ->[<'Lint i ; lexer v>]
  and lexint r s =
    match s with parser
      [< ''0'..'9' as c >]
      -> let u = (Char.code c) - (Char.code '0') in lexint (10*r + u) s
      | [<>] -> r,s ;;
val lexer : char Stream.t -> lexeme Stream.t = <fun>
val lexint : int -> char Stream.t -> int * char Stream.t = <fun>
```

Функция `lexint` предназначена для анализа той части потока символов, которая соответствует числовой постоянной. Она вызывается, когда функция `lexer` встречает цифры. В этом случае функция `lexint` поглощает все последовательные цифры и выдаёт соответствующее значение полученного числа.

11.3.3 Регулярные выражения

Оставим ненадолго практику и рассмотрим проблему лексических единиц с теоретической точки зрения.

Лексическая единица является словом. Слово образуется при конкатенации элементов алфавита. В нашем случае алфавитом является множество символов ASCII.

Теоретически, слово может вообще не содержать символов (пустое слово [2]) или состоять из одного символа.

Теоретические исследования конкатенации элементов алфавита для образования лексических элементов (лексем) привели к созданию простого формализма, известного как регулярные выражения.

Определение

Регулярные выражения позволяют определить множества слов. Пример такого множества: идентификаторы. Принцип определения основан на некоторых теоретико–множественный операциях. Пусть M и N — два множества слов, тогда мы можем определить:

- объединение M и N, записываемое $M|N$.
- дополнение M, записываемое $\wedge M$: множество всех слов, кроме тех, которые входят в M.

[2] традиционно, такое слово обозначается греческой буквой эпсилон: ε

- конкатенация M и N: множество всех слов созданных конкатенацией слова из M и слова из N. Записывается просто MN.

- мы можем повторить операцию конкатенации слов множества M и тем самым получить множество слов образованных из конечной последовательности слов множеств M. Такое множество записывается $M+$. Он содержит все слова множества M, все слова полученные конкатенацией двух слов множества M, трёх слов, и т.д. Если мы желаем чтобы данное множество содержало пустое слово, необходимо писать $M*$.

- для удобства, существует дополнительная конструкция $M?$, которая включает все слова множества M, а так же пустое слово.

Один единственный символ ассоциируется с одно элементным множеством. В таком случае выражение *a| b| c* описывает множество состоящее из трёх слов *a*, *b* и *c*. Существует более компактная запись: *[abc]*. Так как наш алфавит является упорядоченным (по порядку кодов ASCII), можно определить интервал. Например, множество цифр запишется как $[0-9]$. Для группировки выражений можно использовать скобки.

Для того, чтобы использовать в записи сами символы–операторы, как обычные символы, необходимо ставить перед ними обратную косую черту: \. Например множество $(\backslash *)*$ обозначает множество последовательностей звёздочек.

Пример

Пусть существует множество из цифр (0,1,2,3,4,5,6,7,8,9), символы плюс $(+)$ и минус $(-)$, точки (.) и буквы E. Теперь мы можем определить множество чисел *num*. Назовем *integers* множество определённое выражением $[0-9]+$. Множество неотрицательных чисел *unum* определяется так:

$$integers?(.integers)?(E(\backslash + | -)?integers)?$$

Множество отрицательных и положительных чисел записывается:

$$unum| - unum \quad \text{или} \quad -?unum$$

Распознавание

Теперь, после того как множество выражений определено, остаётся проблема распознавания принадлежности строки символов или одной из её

подстрок этому множеству. Для решения данной задачи необходимо реализовать программу обработки выражений, которая соответствует формальным определениям множества. Для регулярных выражений такая обработка может быть автоматизирована. Подобная автоматизация реализована в модуле `Genlex` из библиотеки `Str` и инструментом `ocamllex`, которые будут представлены в следующих двух параграфах.

Библиотека `Str`

В данном модуле имеется абстрактный тип `regexp` и функция `regexp`. Указанный тип представляет регулярные выражения, а функция трансформирует регулярное выражение, представленное в виде строки символов, в абстрактное представление.

Модуль `Str` содержит несколько функций, которые используют регулярные выражения и манипулируют символьными строками. Синтаксис регулярных выражений библиотеки `Str` приведён в таблице 11.1.

Таблица 11.1: Регулярные выражения

.	любой символ, кроме \
$*$	ноль или несколько экземпляров предыдущего выражения
$+$	хотя бы один экземпляр предыдущего выражения
?	ноль или один экземпляр предыдущего выражения
[..]	множество символов
.	любой символ, кроме \
	интервал записывается при помощи - (пример $[0-9]$)
	дополнение записывается при помощи $\wedge$ (пример $[\wedge A-Z]$)
$\wedge$	начало строки (не путать с дополнением $\wedge$)
$	конец строки
|	вариант
(..)	группировка в одно выражение (можно ссылаться на это выражение)
i	числовая константа i ссылается на i–ый элемент группированного выражения
\	забой, используется для сопоставления зарезервированных символов в регулярных выражениях

Пример

В следующем примере напишем функцию, которая переводит дату из английского формата во французский. Предполагается, что входной файл

состоит из строк, разбитых на поля данных и элементы даты разделяются точкой. Определим функцию, которая из полученной строки (строка файла), выделяет дату, разбивает её на части, переводит во французских формат и тем самым заменяет старую дату на новую.

```
# let french_date_of d =
  match d with
   [mm; dd; yy] -> dd^"/"^mm^"/"^yy
   | _ -> failwith "Bad date format" ;;
val french_date_of : string list -> string = <fun>

# let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
val english_date_format : Str.regexp = <abstr>

# let trans_date l =
  try
  let i=Str.search_forward english_date_format l 0 in
  let d1 = Str.matched_string l in
  let d2 = french_date_of (Str.split (Str.regexp "\.") d1) in
    Str.global_replace english_date_format d2 l
  with Not_found -> l ;;
  val trans_date : string -> string = <fun>

# trans_date
"..............06.13.99............" ;;
- : string = "..............13/06/99............"
```

11.3.4 Инструмент `Ocamllex`

`ocamllex` — это лексический генератор созданный для Objective CAML по модели `lex`, написанном на языке C. При помощи файла, описывающего элементы лексики в виде множества регулярных выражений, которые необходимо распознать, он создаёт файл–исходник на Objective CAML. К описанию каждого лексического элемента можно привязать какое–нибудь действие, называемое семантическое действие. В полученном коде используется абстрактный тип `lexbuf` из модуля `Lexing`. В данном модуле также имеется несколько функций управления лексическими буферами, которые могут быть использованы программистом для того чтобы определить необходимые действия.

Обычно, файлы, описывающие лексику, имеют расширение `.mll`. Для того, чтобы из файла `lex_file.mll` получить файл на Objective CAML,

необходимо выполнить следующую команду:

```
ocamllex lex_file.mll
```

После этого, мы получим файл `lex_file.ml`, содержащий код лексического анализатора. Теперь, данный файл можно использовать в программе на Objective CAML. Каждому множеству правил анализа соответствует функция, которая принимает лексический буфер (типа `Lexing.lexbuf`) и затем возвращает значение, определённое семантическим действием. Значит, все действия для определённого правила должны создавать значение одного и того же типа.

Формат у файла для `ocamllex` следующий:

```
{
  header
}
let ident = regexp
          ...
rule ruleset1 = parse
                  regexp { action }
                | ...
                | regexp { action }
and ruleset2 = parse
          ...
and ...
{
  trailer-and-end
}
```

Части «заголовок» и «продолжение-и-конец» не являются обязательными. Здесь вставляется код Objective CAML, определяющий типы данных, функции и т.д. необходимые для обработки данных. В последней части используются функции, которые используют правила анализа множества лексических данных из средней части. Серия объявлений, которая предшествует определению правил, позволяет дать имя некоторым регулярным выражениям. Эти имена будут использоваться в определении правил.

Пример

Вернёмся к нашему интерпретатору BASIC и усовершенствуем тип возвращаемых лексических единиц. Таким образом, мы можем воспользо-

ваться функцией `lexer`, (см. стр. ??) которая возвращает такой же тип результата (`lexeme`), но на входе она получает буфер типа `Lexing.lexbuf`.

```
  {
    let string_chars s =
    String.sub s 1 ((String.length s)-2) ;;
  }

let op_ar = ['-' '+' '*' '\%' '/']
let op_bool = ['!' '\&' '|']
let rel = ['=' '<' '>']

rule lexer = parse
    [' '] { lexer lexbuf }
  | op_ar { Lsymbol (Lexing.lexeme lexbuf) }
  | op_bool { Lsymbol (Lexing.lexeme lexbuf) }
  | "<=" { Lsymbol (Lexing.lexeme lexbuf) }
  | ">=" { Lsymbol (Lexing.lexeme lexbuf) }
  | "<>" { Lsymbol (Lexing.lexeme lexbuf) }
  | rel { Lsymbol (Lexing.lexeme lexbuf) }
  | "REM" { Lsymbol (Lexing.lexeme lexbuf) }
  | "LET" { Lsymbol (Lexing.lexeme lexbuf) }
  | "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
  | "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
  | "IF" { Lsymbol (Lexing.lexeme lexbuf) }
  | "THEN" { Lsymbol (Lexing.lexeme lexbuf) }
  | '-'? ['0'-'9']+ { Lint (int_of_string (Lexing.lexeme lexbuf)) }
  | ['A'-'z']+ { Lident (Lexing.lexeme lexbuf) }
  | '"' [^ '"']* '"' { Lstring (string_chars (Lexing.lexeme lexbuf
    )) }
```

После обработки данного файла командой `ocamllex` получим функцию `lexer` типа `Lexing.lexbuf -> lexeme`. Далее, мы рассмотрим каким образом подобные функции используются в синтаксическом анализе (см. стр. ??).

11.4 Синтаксис

Благодаря лексическому анализу, мы в состоянии разбить поток символов на более структурированные элементы: лексические элементы. Теперь необходимо знать как правильно объединять эти элементы в син-

таксически корректные фразы какого-нибудь языка. Правила синтаксической группировки определены посредством грамматических правил. Формализм, произошедший из лингвистики, был с успехом перенят математиками, занимающимися теориями языков, и специалистами по информатике. На странице **??** мы уже видели пример грамматики для языка BASIC. Здесь мы снова вернёмся к этому примеру, чтобы более углублённо ознакомится с базовыми концепциями грамматики.

11.4.1 Грамматика

Говоря формальным языком, грамматика основывается на четырёх элементах:

1. Множество символов, называемых терминалами. Эти символы являются лексическими элементами языка. В Бэйсике к ним относятся символы операторов, арифметических отношений и логические ($+$, &, $<$, $\leq$, $\cdots$), ключевые слова языка **(GOTO, PRINT, IF, THEN,** $\cdots$**)**, целые числа (элемент `integer`) и переменные (элемент `variable`).

2. Множество нетерминальных символов, которые представляют синтаксические компоненты языка. Например, программа на языке Бэйсик состоит из строк. Таким образом мы имеем компоненту Line, которая в свою очередь состоит из выражений (Expression), и т.д.

3. Множество так называемых порождающих правил. Они описывают каким образом комбинируются терминальные и нетерминальные символы, чтобы создать синтаксическую компоненту. Строка в Бэйсике начинается с номера, за которой следует инструкция. Смысл правила следующий:

   $$\text{Line} ::= \textit{integer}\ \text{Instruction}$$

   Одна и та же компонента может быть порождена несколькими способами. В этом случае варианты разделяются символом | как в:

   $$\begin{array}{rcl} \text{Instruction} & ::= & \text{LET variable} = \text{Instruction} \\ & | & \text{GOTO integer} \\ & | & \text{PRINT Expression} \\ \text{etc} & & \end{array}$$

Среди всех нетерминальных символов различают один, называемый начальным. Правило, которое порождает эту аксиому является порождающим правилом всего языка.

11.4.2 Порождение и распознавание

С помощью порождающих правил можно определить принадлежит ли последовательность лексем языку.

Рассмотрим простой язык, описывающий арифметические выражения:

Exp	::=	*integer*	(R1)
	|	Exp + Exp	(R2)
	|	Exp * Exp	(R3)
	|	(Exp)	(R4)

здесь *(R1) (R2) (R3) (R4)* — имена правил. По окончании лексического анализа выражение $1 * (2 + 3)$ становится последовательностью следующих лексем:

*integer * (integer + integer)*

Для того, чтобы проанализировать эту фразу и убедиться в том что она принадлежит языку арифметических выражений, воспользуемся правилами справа налево: если часть выражения соответствует правому члену какого-нибудь правила, мы заменяем это выражение соответствующим левым членом. Этот процесс продолжается до тех пор, пока выражение не будет редуцировано до аксиомы. Ниже представлен результат такого анализа [3]:

integer * (integer + integer)	(R1)	Exp * (integer + integer)
	(R1)	Exp * (Exp + integer)
	(R1)	Exp * (Exp + Exp)
	(R2)	Exp * (Exp)
	(R4)	Exp * Exp
	(R3)	Exp

Начиная с последней линии, которая содержит лишь Exp и следуя стрелкам, можно определить каким образом было полученное выражение исходя из аксиомы Exp. Соответственно данная фраза является правильно сформированной фразой языка арифметических выражений, определённого грамматикой.

[3] анализируемая часть выражения подчёркнута, а так же указано используемое правило

Преобразование грамматики в программу, способную распознать принадлежность последовательности лексем языку, который определён грамматикой, является более сложной проблемой, чем проблема использования регулярных выражений. Действительно, в соответствии с математическим результатом любое множество (слов), определённое формализмом регулярных выражений, может быть определено другим формализмом: детерминированные конечные автоматы. Такие автоматы легко реализуются программами, принимающими поток символов. Подобный результат для грамматик в общем не существует. Однако, имеются менее строгие (?) (weaker) результаты устанавливающие эквивалентность между определёнными классами грамматик и более богатыми автоматами: автомат со стеком. Здесь мы не станем вдаваться ни в детали этих результатов, ни в точное определение таких автоматов. Однако, мы можем определить какие классы грамматики могут использоваться в средствах генерации синтаксических анализаторов или для реализации напрямую анализатора.

11.4.3 Нисходящий анализ

Разбор выражения $1 * (2 + 3)$ в предыдущем параграфе не является единственным: мы с таким же успехом могли бы начать редуцирование *integer*, то есть воспользоваться правилом *(R2)* редуцирования $2 + 3$. Эти два способа распознавания являются двумя типами анализа: восходящий анализ (справа налево) и нисходящий слева направо. Последний анализ легко реализуется при помощи потоков лексем модуля `Stream`. Средство `ocamlyacc` использует восходящий анализ, при котором применяется стек, как это уже было проиллюстрировано синтаксическим анализатором программ на Бэйсике. Выбор анализа не просто дело вкуса, в зависимости от используемой для спецификации языка формы грамматики, можно или нет применять нисходящий анализ.

Простой случай

Каноническим примером нисходящего анализа является префиксная запись арифметических выражений, определяемая как:

$$\begin{array}{rcl} \text{Exp} & ::= & \text{integer} \\ & | & + \text{ Exp Exp} \\ & | & * \text{ Exp Exp} \end{array}$$

В данном случае достаточно знать первую лексему, для того чтобы определить какое правило может быть использовано. При помощи по-

добной предсказуемости нет необходимости явно управлять стеком, достаточно положиться на рекурсивный вызов анализатора. И тогда при помощи `Genlex` и `Stream` очень просто написать программу реализующую нисходящий анализ. Функция `infix_of` из полученного префиксного выражения возвращает его инфиксный эквивалент:

```
# let lexer s =
    let ll = Genlex.make_lexer ["+";"*"]
    in ll (Stream.of_string s);;
val lexer : string -> Genlex.token Stream.t = <fun>

# let rec stream_parse s =
    match s with parser
      [<'Genlex.Ident x>] -> x
    | [<'Genlex.Int n>] -> string_of_int n
    | [<'Genlex.Kwd "+"; e1=stream_parse; e2=stream_parse>] -> "("^
        e1^"+"^e2^")"
    | [<'Genlex.Kwd "*"; e1=stream_parse; e2=stream_parse>] -> "("^
        e1^"*"^e2^")"
    | [<>] -> failwith "Parse error" ;;
val stream_parse : Genlex.token Stream.t -> string = <fun>

# let infix_of s = stream_parse (lexer s) ;;
val infix_of : string -> string = <fun>

# infix_of "* +3 11 22";;
- : string = "((3+11)*22)"
```

Однако не стоит забывать о некоторой примитивности лексического анализа. Советуем периодически добавлять пробелы между различными лексическими элементами.

```
# infix_of "*+3 11 22";;
- : string = "*+"
```

Случай посложней

Синтаксический анализ при помощи потоков предсказуем, он облагает грамматику двумя условиями:

- В правилах грамматики не должно быть левой рекурсии. Правило называется рекурсивным слева, если его правый член начинается с

нетерминального символа, который является левой частью правила. Например: `Exp ::= Exp + Exp`

- Не должно существовать правил начинающихся одним и тем же выражением.

Грамматика арифметических выражений, приведённая на стр. 66, не подходит для нисходящего анализа: они не удовлетворяют ни одному из условий. Для того, чтобы применить нисходящий анализ необходимо переформулировать грамматику таким образом, чтобы удалить левую рекурсию и неопределённость правил. Вот полученный результат:

$$
\begin{array}{rcl}
\text{Exp} & ::= & \text{Atom NextExpr} \\
\text{NextExpr} & ::= & + \text{ Atom} \\
 & | & - \text{ Atom} \\
 & | & * \text{ Atom} \\
 & | & / \text{ Atom} \\
 & | & \varepsilon \\
\text{Atom} & ::= & \text{integer} \\
 & | & (\text{ Expr })
\end{array}
$$

Заметьте использование пустого слова ε в определении `NextExpr`. Оно необходимо, если мы хотим чтобы просто целое число являлось выражением.

Следующий анализатор есть просто перевод вышеуказанной грамматики в код. Он реализует абстрактное синтаксическое дерево арифметических выражений.

```
# let rec rest = parser
    [< 'Lsymbol "+"; e2 = atom >] -> Some (PLUS,e2)
  | [< 'Lsymbol "-"; e2 = atom >] -> Some (MINUS,e2)
  | [< 'Lsymbol "*"; e2 = atom >] -> Some (MULT,e2)
  | [< 'Lsymbol "/"; e2 = atom >] -> Some (DIV,e2)
  | [< >] -> None
 and atom = parser
     [< 'Lint i >] -> ExpInt i
   | [< 'Lsymbol "("; e = expr ; 'Lsymbol ")" >] -> e
 and expr s =
   match s with parser
     [< e1 = atom >] ->
       match rest s with
           None -> e1
```

```
        | Some (op,e2) -> ExpBin(e1,op,e2) ;;
val rest : lexeme Stream.t -> (bin_op * expression) option = <fun>
val atom : lexeme Stream.t -> expression = <fun>
val expr : lexeme Stream.t -> expression = <fun>
```

Сложность использования нисходящего анализа заключается в том, что грамматика должна быть очень ограниченной формы. Если язык выражен естественно с использованием левой рекурсии (как в инфиксных выражениях), то не всегда легко определить эквивалентную грамматику, то есть определяющую такой же язык, которая бы удовлетворяла требованиям нисходящего анализа. По этой причине, средства `yacc` и `ocamlyacc` реализуют восходящий анализ, который разрешает определение более естественных грамматик. Однако, мы увидим, что даже в этом случае существуют ограничения.

11.4.4 Восходящий анализ

Мы уже вкратце представили на странице **??** принципы восходящего анализа: сдвиг и вывод [4]. После каждого подобного действия, состояние стека изменяется. Из этой последовательности можно вывести правила грамматики, в случае если грамматика это позволяет, как в примере с нисходящим анализом. Опять же, сложности возникают из-за неопределённости правил, когда невозможно выбрать между продвинутся или сократить. Проиллюстрируем действие восходящего анализа и его недостатки на все тех же арифметических выражениях в постфиксном и инфиксном написании.

Положительная сторона

Упрощённая постфиксная грамматика арифметических выражений выглядит так:

$$\begin{array}{lcll} \text{Exp} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{Exp Exp +} & \text{(R2)} \\ & | & \text{Exp Exp -} & \text{(R3)} \end{array}$$

Данная грамматика является двойственной по отношению к префиксной: для того чтобы точно знать какое правило следует применить, необходимо дождаться окончания анализа. В действительности анализ по-

[4] Обычно на русский язык термины **shift** и **reduce** переводят как сдвиг и вывод, но в данной книге для термина **reduce** будет также использоваться перевод «сокращение» — прим. пер.

добных выражений схож с вычислением при помощи стека. Только вместо проталкивания результата вычисления, проталкиваются грамматические символы. Если в начале стек пустой, то после того как ввод закончен, необходимо получить стек содержащий лишь нетерминальную аксиому. Приведём изменение стека: если мы продвигаемся, то проталкивается текущий нетерминальный символ; если сокращаем, то первые символы стека соответствуют правому члену (в обратном порядке) правила и тогда мы заменяем эти элементы соответствующими нетерминальными элементами.

В таблице 11.2 приведён восходящий анализ выражения 1 2 + 3 * 4 +. Считываемая лексическая единица подчёркивается для более удобного чтения. Конец потока помечается символом $.

Таблица 11.2: Восходящий анализ

Действие	Вход	Стек
	1 2 + 3 * 4 + $	[]
Сдвиг		
	2 + 3 * 4 + $	[1]
Сократить *(R1)*		
	2 + 3 * 4 + $	[Exp]
Сдвиг		
	+ 3 * 4 + $	[2 Exp]
Сдвиг, Сократить *(R1)*		
	+ 3 * 4 + $	[Exp Exp]
Сдвиг, Сократить *(R2)*		
	3 * 4 + $	[Exp]
Сдвиг, Сократить *(R1)*		
	* 4 + $	[Exp Exp]
Сдвиг, Сократить *(R3)*		
	4 + $	[Exp]
Сдвиг, Сократить *(R1)*		
	+ $	[Exp Exp]
Сдвиг, Сократить *(R2)*		
	$	[Exp]

Отрицательная сторона

Вся трудность перехода от грамматики к программе распознающей язык заключается в определении действия, которое необходимо применить. Проиллюстрируем эту проблему на трёх примерах, приводящих к трём неопределённостям.

Первый пример есть грамматика выражений использующих операцию сложения:

$$\begin{array}{lcll} \text{E0} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{E0} + \text{E0} & \text{(R2)} \end{array}$$

Неопределённость данной грамматики проявляется при использовании правила *R2*. Предположим следующую ситуацию:

Действие	Вход	Стек
:		
	$\underline{+}$	E0 + E0 ...
:		

В подобном случае невозможно определить необходимо сдвинуть и протолкнуть в стек + или сократить в соответствии с правилом *(R2)* оба E0 и присутствующий в стеке +. Подобная ситуация называется конфликтом сдвиг-вывод (shift/reduce). Она является следствием того, что выражение *integer* + *integer* + *integer* может быть выведено справа двумя способами.

Первый вариант:

$$\begin{array}{ll} \text{E0} & (\textit{R2})\ \text{E0} + \underline{\text{E0}} \\ & (\textit{R1})\ \underline{\text{E0}} + \textit{integer} \\ & (\textit{R2})\ \text{E0} + \underline{\text{E0}} + \textit{integer} \end{array}$$

Второй вариант:

$$\begin{array}{ll} \text{E0} & (\textit{R2})\ \text{E0} + \underline{\text{E0}} \\ & (\textit{R1})\ \text{E0} + \text{E0} + \underline{\text{E0}} + \textit{integer} \\ & (\textit{R2})\ \text{E0} + \underline{\text{E0}} + \textit{integer} \end{array}$$

Выражения, полученные двумя выводами, могут показаться одинаковыми с точки зрения вычисления выражения,

(*integer* + *integer*) + *integer* и *integer* + (*integer* + *integer*)

но разными для конструкции синтаксического дерева (см. рис. **??** на стр. **??**)

Второй пример грамматики, порождающей конфликт между сдвиг-вывод, содержит такую же неопределённость: явное заключение в скобки. Но в отличии от предыдущего случая, выбор сдвиг-вывод изменяет смысл выражения. Пусть есть грамматика:

$$\begin{array}{lcl} \text{E1} & ::= & \textit{integer}\ (\textit{R1}) \\ & | & \text{E1} + \text{E1}\ (\textit{R2}) \\ & | & \text{E1} * \text{E1}\ (\textit{R3}) \end{array}$$

Здесь мы снова получаем предыдущий конфликт как в случае с + так и для *, но к этому добавляется другой, между + и *. Опять же, одно и то же выражение может быть получено двумя способами, так как у него существует два вывода справа:

integer + *integer* * *integer*

Первый вариант:

$$\begin{array}{ll} \text{E1} & (\textit{R3})\ \text{E0} * \underline{\text{E1}} \\ & (\textit{R1})\ \underline{\text{E1}} * \textit{integer} \\ & (\textit{R2})\ \text{E1} + \underline{\text{E1}} * \textit{integer} \end{array}$$

Второй вариант:

$$\begin{array}{ll} \text{E1} & (\textit{R2})\ \text{E1} + \underline{\text{E1}} \\ & (\textit{R2})\ \text{E1} + \text{E1} * \underline{\text{E1}} \\ & (\textit{R1})\ \text{E1} + \underline{\text{E1}} * \textit{integer} \end{array}$$

В данном выражении обе пары скобок имеют разный смысл:

(*integer* + *integer*) * *integer* $\neq$ *integer* + (*integer* * *integer*)

Подобную проблему, мы уже встречали в выражениях Basic (см. стр. **??**). Она была разрешена при помощи приоритетов, которые присваиваются операторам: сначала редуцируется правило (R3), затем (R2), что соответствует заключению в скобки произведения.

Данную проблему выбора между + и * можно решить изменив грамматику. Для того, введём два новых терминальных символа: член T (*term*) и множитель F (*factor*). Отсюда получаем:

$$\begin{array}{lcll} \text{E} & ::= & \text{E} + \text{T} & (\textit{R1}) \\ & | & \text{T} & (\textit{R2}) \\ \text{T} & ::= & \text{T} + \text{F} & (\textit{R3}) \\ & | & \text{F} & (\textit{R4}) \\ \text{F} & ::= & \text{T} + \textit{integer} & (\textit{R5}) \end{array}$$

После этого, единственный способ получить *integer* + *integer* * *integer*: посредством правила (*R1*).

Третий и последний случай касается условных конструкций языка программирования. На пример в Pascal существует две конструкции: `if .. then` и `if .. then .. else`. Пусть существует следующая грамматика:

$$\begin{array}{lcll} \text{Instr} & ::= & \textit{if}\ \text{EXP}\ \textit{then}\ \text{Instr} & (R1) \\ & - & \textit{if}\ \text{EXP}\ \textit{then}\ \text{Instr}\ \textit{else}\ \text{Instr} & (R2) \\ & - & \text{etc}\ \ldots & \end{array}$$

И в следующей ситуации:

Действие	Вход	Стек
:		
	else	[Instr *then* Epx *if* ...]

Невозможно определить соответствуют ли элементы стека правила (*R1*) и в этом случае необходимо сократить или соответствуют первому `Instr` правила (*R2*) и тогда необходимо сдвинуть.

Кроме конфликтов сдвиг-вывод, восходящий анализ вызывает конфликт вывод-вывод.

Мы представим здесь инструмент `ocamlyacc`, который использует подобную технику может встретить указанные конфликты.

11.5 Пересмотренный Basic

Теперь, используя совместно `ocamllex` и `ocamlyacc`, заменим функцию `parse` для Бэйсика, приведённую на странице **??**, на функции полученные при помощи файлов спецификации лексики и синтаксиса языка.

Для этого, мы не сможем воспользоваться типами лексических единиц, в таком виде как они были определены. Необходимо определить более точные типы, чтобы различать операторы, команды и ключевые слова.

Так же, нам понадобится изолировать в отдельном файле `basic_types.mli` декларации типов, относящиеся к абстрактному синтаксису. В нем будут содержатся декларации типа `sentences`, а так же других типы необходимые этому.

11.5.1 Файл `basic_parser.mly`

Заголовок

Данный файл содержит вызовы деклараций типов абстрактного синтаксиса и две функции перевода строк символов в их эквивалент абстрактного синтаксиса.

```
%{
open Basic_types ;;

let phrase_of_cmd c =
 match c with
   "RUN" -> Run
 | "LIST" -> List
 | "END" -> End
 | _ -> failwith "line : unexpected command"
;;

let bin_op_of_rel r =
 match r with
   "=" -> EQUAL
 | "<" -> INF
 | "<=" -> INFEQ
 | ">" -> SUP
 | ">=" -> SUPEQ
 | "<>" -> DIFF
 | _ -> failwith "line : unexpected relation symbol"
;;

%}
```

Декларации

Здесь содержится три части: декларация лексем, декларация правил ассоциативности и приоритетов, декларация стартового символа `line`, которая соответствует анализу линии программы или команды.

Ниже представлены лексические единицы:

```
%token <int> Lint
%token <string> Lident
%token <string> Lstring
```

```
%token <string> Lcmd
%token Lplus Lminus Lmult Ldiv Lmod
%token <string> Lrel
%token Land Lor Lneg
%token Lpar Rpar
%token <string> Lrem
%token Lrem Llet Lprint Linput Lif Lthen Lgoto
%token Lequal
%token Leol
```

Имена деклараций говорят сами за себя и они описаны в файле `basic_lexer.mll` (см. стр. **??**).

Правила приоритета операторов схожи со значениями, которые определяются функциями `priority_uop` и `priority_binop`, которые были определены грамматикой Бейсика (см. стр. **??**).

```
%right Lneg
%left Land Lor
%left Lequal Lrel
%left Lmod
%left Lplus Lminus
%left Lmult Ldiv
%nonassoc Lop
```

Символ `Lop` необходим для обработки унарных минусов. Он не является терминальным символом, а «псевдо–терминальным». Благодаря этому, получаем перегрузку операторов, когда в двух случаях использования одного и того же оператора, приоритет меняется в зависимости от контекста. Мы вернёмся к этому случаю, когда будем рассматривать правила грамматики.

Здесь нетерминалом является `line`. Полученная функция возвращает дерево абстрактного синтаксиса, которое соответствует проанализированной линии.

```
%start line
%type <Basic_types.phrase> line
```

Правила грамматики

Грамматика делится на 3 нетерминальных элемента: `line` для линии, `inst` для инструкции и `exp` для выражений. Действия, которые привязаны к каждому правилу лишь создают соответствующую часть абстрактного синтаксиса.

```
 %%
line :
    Lint inst Leol { Line {num=$1; inst=$2} }
  | Lcmd Leol { phrase_of_cmd $1 }
  ;

inst :
    Lrem { Rem $1 }
  | Lgoto Lint { Goto $2 }
  | Lprint exp { Print $2 }
  | Linput Lident { Input $2 }
  | Lif exp Lthen Lint { If ($2, $4) }
  | Llet Lident Lequal exp { Let ($2, $4) }
  ;

exp :
    Lint { ExpInt $1 }
  | Lident { ExpVar $1 }
  | Lstring { ExpStr $1 }
  | Lneg exp { ExpUnr (NOT, $2) }
  | exp Lplus exp { ExpBin ($1, PLUS, $3) }
  | exp Lminus exp { ExpBin ($1, MINUS, $3) }
  | exp Lmult exp { ExpBin ($1, MULT, $3) }
  | exp Ldiv exp { ExpBin ($1, DIV, $3) }
  | exp Lmod exp { ExpBin ($1, MOD, $3) }
  | exp Lequal exp { ExpBin ($1, EQUAL, $3) }
  | exp Lrel exp { ExpBin ($1, (bin_op_of_rel $2), $3) }
  | exp Land exp { ExpBin ($1, AND, $3) }
  | exp Lor exp { ExpBin ($1, OR, $3) }
  | Lminus exp %prec Lop { ExpUnr(OPPOSITE, $2) }
  | Lpar exp Rpar { $2 }
  ;
  %%
```

Данные правила не нуждаются в особых комментариях, кроме следующего:

```
exp :
  ...
  | Lminus exp %prec Lop { ExpUnr(OPPOSITE, $2) }
```

Это правило касается использования унарного минуса -. Ключевое слово `%prec` означает, что указанная конструкция получает приоритет от `Lop` (в данном случае наивысший).

11.5.2 Файл `basic_lexer.mll`

Лексический анализ содержит лишь одно множество: `lexer`, которое точно соответствует старой функции `lexer` (см. стр. **??**).

Семантическое действие, которое связано с распознаванием лексических единиц, возвращает результат соответствующего конструктора. Необходимо загрузить файл синтаксических правил, так как в нем декларируется тип лексических единиц. Добавим так же функцию, которая удаляет кавычки вокруг строк.

```
{
 open Basic_parser ;;

 let string_chars s = String.sub s 1 ((String.length s)-2) ;;
}

rule lexer = parse
    [' ' '\t'] { lexer lexbuf }

 | '\n' { Leol }

 | '!' { Lneg }
 | '&' { Land }
 | '|' { Lor }
 | '=' { Lequal }
 | '%' { Lmod }
 | '+' { Lplus }
 | '-' { Lminus }
 | '*' { Lmult }
 | '/' { Ldiv }

 | ['<' '>'] { Lrel (Lexing.lexeme lexbuf) }
 | "<=" { Lrel (Lexing.lexeme lexbuf) }
 | ">=" { Lrel (Lexing.lexeme lexbuf) }

 | "REM" [^ '\n']* { Lrem (Lexing.lexeme lexbuf) }
 | "LET" { Llet }
```

```
| "PRINT" { Lprint }
| "INPUT" { Linput }
| "IF" { Lif }
| "THEN" { Lthen }
| "GOTO" { Lgoto }

| "RUN" { Lcmd (Lexing.lexeme lexbuf) }
| "LIST" { Lcmd (Lexing.lexeme lexbuf) }
| "END" { Lcmd (Lexing.lexeme lexbuf) }

| ['0'−'9']+ { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'−'z']+ { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"' { Lstring (string_chars (Lexing.lexeme lexbuf)
    ) }
```

Заметьте, что нам пришлось изолировать символ =, который используется одновременно в выражениях и приравниваниях.

Для двух рациональных выражений необходимо привести определённые объяснения. Линия комментариев соответствует выражению `("REM"[^'\n']*)`, где за ключевым словом `REM` следует какое угодно количество символов и затем перевод строки. Правило, которое соответствует символьным строкам, `('\ [^ '\]* '\)`, подразумевает последовательность символов, отличных от "и заключённых в кавычки ".

11.5.3 Компиляция, компоновка

Компиляция должна быть реализована в определённом порядке. Это связано с взаимозависимостью деклараций лексем. Поэтому в нашем случае, необходимо выполнить команды в следующем порядке:

```
ocamlc −c basic_types.mli
ocamlyacc basic_parser.mly
ocamllex basic_lexer.mll
ocamlc −c basic_parser.mli
ocamlc −c basic_lexer.ml
ocamlc −c basic_parser.ml
```

После чего получим файлы `basic_lexer.cmo` и `basic_parser.cmo`, которые можно использовать в нашей программе.

Теперь, у нас есть весь необходимый арсенал, для того чтобы переделать программу.

Удалим все типы и функции параграфов «лексический анализ» (стр. ??) и «синтаксический анализ» (стр. ??) для программы Бэйсик. Также в функции `one_command` (стр. ??) заменим выражение:

```
match parse (input_line stdin) with
```

на

```
match line lexer (Lexing.from_string ((input_line stdin)^"\n")) with
```

Заметьте, что необходимо поместить в конце линии символ конца '\n', который был удалён функцией `input_line`. Это необходимо, потому что данный символ используется для указания конца командной линии (`Leol`).

11.6 Exercises

11.7 Резюме

В данной главе были описаны различные средства лексического и синтаксического анализа Objective CAML. По порядку описания:

- модуль `Str` для фильтрования рациональных выражений
- модуль `Genlex` для быстрого создания простых лексических анализаторов
- `ocamllex` представитель семейства `lex`
- `ocamlyacc` представитель семейства `yacc`
- потоки, для построения нисходящих анализаторов, в том числе и контекстных

При помощи инструментов `ocamllex` и `ocamlyacc` мы переделали синтаксический анализ Бэйсика, который проще поддерживать, чем анализатор представленный на стр. ??.

11.8 To Learn More

Глава 12

Взаимодействие с языком C

Глава 13

Приложения

Часть III

Устройство программы

Глава 14

Модульное программирование

Глава 15

Объектно-ориентированное программирование

Глава 16

Сравнение моделей устройств программ

Глава 17

Приложения

Часть IV

Параллелизм и распределение

Глава 18

Процессы и связь между процессами

Глава 19

Параллельное программирование

Глава 20

Распределённое программирование

Глава 21

Приложения

Часть V

Разработка программ с помощью Objective CAML

Часть VI

Приложения