Разработка программ с помощью Objective Caml
Developing Applications with Objective Caml

Emmanuel Chailloux
Pascal Manoury
Bruno Pagano

26 августа 2012 г.

Оглавление

Глава 1

Как заполучить Objective CAML

Часть I

Основы языка

Глава 2

Функциональное программирование

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. ASCII 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 приведенных примерах используется контекстно–независимая грамматика. Здесь мы узнаем как реализовать контекстный анализ при помощи Stream. В третьей части мы вернемся 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 предопределенных 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 разузнать 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 окружен (’). 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 какой угодно последовательности букв и цифр, могут включать символ подчеркивания (_) 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"; "-
        ";
```

```
3       "!"; "+"; "−"; "*"; "/"; "%";
4       "="; "<"; ">"; "<="; ">="; "<>";
5       "&"; "|" ] ;;
```

При помощи данного множества, определим функцию лексического анализа:

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

Приведенная функция `line_lexer`, из входящего потока символов создает поток соответствующих лексем.

11.3.2 Использование потоков

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

В следующем примере определен лексический анализатор арифметических выражений. Функции lexer передается поток символов из которого она создает 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.

```
1  # let rec spaces s =
2      match s with parser
3        [<'' ' ; rest >] −> spaces rest
4        | [<''\t' ; rest >] −> spaces rest
5        | [<''\n' ; rest >] −> spaces rest
6        | [<>] −> ();;
7  val spaces : char Stream.t −> unit = <fun>
8  # let rec lexer s =
9     spaces s;
10    match s with parser
11      [< ''(' >] −> [< 'Lsymbol "("; lexer s >]
12      | [< '')' >] −> [< 'Lsymbol ")"; lexer s >]
13      | [< ''+' >] −> [< 'Lsymbol "+" ; lexer s >]
14      | [< ''−' >] −> [< 'Lsymbol "−" ; lexer s >]
15      | [< ''*' >] −> [< 'Lsymbol "*" ; lexer s >]
16      | [< ''/' >] −> [< 'Lsymbol "/" ; lexer s >]
```

[1] тип `lexeme` определен на стр. [?]

```
17      | [< ''0'..'9' as c;
18        i,v = lexint (Char.code c - Char.code('0')) >]
19        ->[<'Lint i ; lexer v>]
20    and lexint r s =
21      match s with parser
22        [< ''0'..'9' as c >]
23        -> let u = (Char.code c) - (Char.code '0') in lexint (10*r + u) s
24        | [<>] -> r,s ;;
25 val lexer : char Stream.t -> lexeme Stream.t = <fun>
26 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 выдает соответствующее значение полученного числа.

11.4 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 рациональные 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 , 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, трех 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 одноэлементным множеством. В таком случае выражение *a| b| c* описывает множество состоящее из трех 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 звездочек.

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 определенное 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, остается проблема распознавания принадлежности строки символов или одной из ее

подстрок этому множеству. Для решения данной задачи необходимо реализовать программу обработки выражений, которая соответствует формальным определениям множества. Для рациональных выражений такая обработка может быть автоматизированна. 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 рациональные выражения, а функция трансформирует рациональное выражение, представленное в виде строки символов, в абстрактное выражение.

Модуль `Str` содержит несколько функций, которые используют рациональные выражения и манипулируют символьными строками. Синтаксис рациональных 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
$\vert$	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 рациональных 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 (линия 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....."
```

11.4.1 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 рациональных 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:

```
1 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:

```
1  {
2    header
3  }
4  let ident = regexp
5            ...
6  rule ruleset1 = parse
7                    regexp { action }
8                    | ...
9                    | regexp { action }
10 and ruleset2 = parse
11           ...
12 and ...
13 {
14   trailer-and-end
15 }
```

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 рациональным выражениям. Эти имена будут использоваться в определении правил.

Пример

Вернёмся к нашему интерпретатору BASIC и усовершенствуем тип возвращаемых лексических единиц. Таким образом, мы можем воспользоваться функцией `lexer`, (см. стр. **??**) которая возвращает такой же тип результата (`lexeme`), но на входе она получает буфер типа `Lexing.lexbuf`.

```
1   {
2     let string_chars s =
3     String.sub s 1 ((String.length s)-2) ;;
4   }
5
6  let op_ar = ['-' '+' '*' '\%' '/']
7  let op_bool = ['!' '\&' '|']
8  let rel = ['=' '<' '>']
9
10 rule lexer = parse
11     [' '] { lexer lexbuf }
12   | op_ar { Lsymbol (Lexing.lexeme lexbuf) }
13   | op_bool { Lsymbol (Lexing.lexeme lexbuf) }
14   | "<=" { Lsymbol (Lexing.lexeme lexbuf) }
15   | ">=" { Lsymbol (Lexing.lexeme lexbuf) }
16   | "<>" { Lsymbol (Lexing.lexeme lexbuf) }
17   | rel { Lsymbol (Lexing.lexeme lexbuf) }
18   | "REM" { Lsymbol (Lexing.lexeme lexbuf) }
19   | "LET" { Lsymbol (Lexing.lexeme lexbuf) }
20   | "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
21   | "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
22   | "IF" { Lsymbol (Lexing.lexeme lexbuf) }
23   | "THEN" { Lsymbol (Lexing.lexeme lexbuf) }
24   | '-'? ['0'-'9']+ { Lint (int_of_string (Lexing.lexeme lexbuf)) }
25   | ['A'-'z']+ { Lident (Lexing.lexeme lexbuf) }
26   | '"' [^ '"']* '"' { 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. **??**).

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

18.1 Параллельное программирование

Глава 19

Распределённое программирование

Глава 20

Приложения

Часть V

Разработка программ с помощью Objective CAML

Часть VI

Приложения