Lexical Analysis with Regular Expressions

Thursday, October 23, 2008
Reading: Stoughton 3.14, Appel Chs. 1 and 2

CS235 Languages and Automata
Department of Computer Science
Wellesley College

Lecture Overview

Lexical analysis = breaking programs into tokens is the first stage of a compiler.

The structure of tokens can be specified by regular expressions.

The ML-Lex tool can automatically derive a lexical analyzer from a description of tokens specified by regular expressions.

To use ML-Lex, we’ll need to learn a few more ML features:
  • sum-of-product data structures
  • mutable cells
Compiler Structure

Source Program (character stream) → Lexer (a.k.a. Scanner, Tokenizer) → Parser → Abstract Syntax Tree (AST) → Type Checker

Front End (CS235)

Global Analysis Information (Symbol and Attribute Tables)

Intermediate Representation

Middle Stages (CS251/CS301)

Semantic Analysis

Intermediate Representation

Optimizer

Intermediate Representation

Back End (CS301)

Code Generator

Machine code or byte code

Lexical Analysis

Front End Example

```c
if (num > 0 && num <= top) { // Is num in range?
    return c*num
} else {return 0;}
```

Lexer (ignores whitespace, comments)

```
if (num > 0 && num <= top) return c*num
else return 0;
```

Parser (creates AST)

```
if (num > 0 && num <= top) return c*num
else return 0;
```
Sample English Description of Lexer Rules

An integer is a sequence of digits. A nonempty sequence of digits followed by E followed by a nonempty sequence of digits is scientific notation (e.g., 12E34 stands for $12 \times 10^{34}$).

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different.

Certain names are reserved as keywords in the language and cannot be used as identifiers. E.g., Java keywords include while, for, if, else, public, private, static, class, int, void. ML keywords include fun, let, in, end, if, then, else.

If the input character stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines, and comments (known collectively as whitespace) are ignored except as they serve to separate tokens. Some whitespace is required to separate otherwise adjacent identifiers, keywords, and constants.

Some ML-Lex Regular Expression Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>the literal string of characters abc</td>
</tr>
<tr>
<td>.</td>
<td>any character except newline</td>
</tr>
<tr>
<td>[a-zA-Z0-9]</td>
<td>any alphanumeric character</td>
</tr>
<tr>
<td>[^d-g]</td>
<td>any character except lowercase d,e,f,g</td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>r*</td>
<td>zero or more rs, where r a reg. exp.</td>
</tr>
<tr>
<td>r+</td>
<td>one or more rs</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one rs</td>
</tr>
<tr>
<td>(r)</td>
<td>r (parens for grouping)</td>
</tr>
<tr>
<td>{REName}</td>
<td>regular expression with name RENname</td>
</tr>
</tbody>
</table>
Regular Expressions for Some Tokens

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;if&quot;</td>
<td>if keyword</td>
</tr>
<tr>
<td>[a-zA-Z_][a-zA-Z0-9_]*</td>
<td>identifiers (variable names)</td>
</tr>
<tr>
<td>[0-9]+(E[0-9]+)?</td>
<td>integers</td>
</tr>
</tbody>
</table>

How should the following be split into tokens?

- if
- if89
- ifE89
- ifEat34

Disambiguation rules:

- **Longest match.** The longest initial substring of the input that can match any regular expression is taken as the next token.
- **Rule Priority.** For a particular longest initial substring, the first regular expression that can match determines its token.

A SLiP Program

Here is a simple program in the straight-line programming language of Appel Ch. 1 (which I call SLiP):

```slip
sum := 5+3;
prod := (print (sum, sum-1), 10*sum);
print(prod);
```

Imagine that this is in the file `test.slip`.

We expect it to have the following tokens:

```sml
sum := 5 + 3 ;
prod := ( print ( sum , sum - 1 ) , 10 * sum ) ;
print ( prod ) ;
```

How do we represent these tokens in SML?
SML Digression: Sum-of-Product Data Types

```sml
(* contents of the file figure.sml *)
datatype figure =
  Square of int (* <constructor function> of <components>*)
  | Rectangle of int * int
  | Triangle of int * int * int
fun perimeter (Square side) = 4*side
  | perimeter (Rectangle(w,h)) = 2*(w+h)
  | perimeter (Triangle(s1,s2,s3)) = s1+s2+s3
fun scale c (Square side) = Square(c*side)
  | scale c (Rectangle(w,h)) = Rectangle(c*w,c*h)
  | scale c (Triangle(s1,s2,s3)) = Triangle(c*s1,c*s2,c*s3)
```

Use "figure.sml";
[opening figure.sml]
datatype figure
  = Rectangle of int * int | Square of int | Triangle of int * int * int
val perimeter = fn : figure -> int
val scale = fn : int -> figure -> figure
val it = () : unit
- map perimeter [Square 1, Rectangle(2,3), Triangle(4,5,6)];
val it = [4,10,15] : int list
- map (scale 10) [Square 1, Rectangle(2,3), Triangle(4,5,6)];
val it = [Square 10,Rectangle (20,30),Triangle (40,50,60)] : figure list

We Can Define our Own List Data Type

```sml
(* contents of the file mylist.sml *)
datatype 'a mylist = Nil | Cons of 'a * ('a mylist)
fun sum Nil = 0
  | sum (Cons(n,ns)) = n + (sum ns)
fun map f Nil = Nil
  | map f (Cons(x,xs)) = Cons(f x, map f xs)
```

Use "mylist.sml";
[opening mylist.sml]
datatype 'a mylist = Cons of 'a * 'a mylist | Nil
val sum = fn : int mylist -> int
val map = fn : ('a -> 'b) -> 'a mylist -> 'b mylist
val it = () : unit
- sum (Cons(1, Cons(2, Cons(3, Nil))));
val it = 6 : int
- map (fn x => x*2) (Cons(1, Cons(2, Cons(3, Nil))));
val it = Cons (2,Cons (6,Cons (6,Nil))) : int mylist

Lexical Analysis 22-9
Lexical Analysis 22-10
A Token Data Type

datatype binop = Add | Mul | Sub | Div

datatype token = EOF
| ID of string
| INT of int
| OP of binop
| PRINT
| LPAREN | RPAREN | COMMA | SEMI | GETS

token data type definition

Sample program

sum := 5+3;
prod := (print (sum, sum-1), 10*sum);
print(prod);

SML token list for sample program

[ID "sum", GETS, INT 5, OP Add, INT 3, SEMI,
ID "prod", GETS, LPAREN, PRINT, LPAREN, ID "sum", COMMA, OP "sum",
OP Sub, INT 1, RPAREN, COMMA, COMMA, INT 10, OP Mul, ID "sum", RPAREN, SEMI,
PRINT, LPAREN, ID "prod", RPAREN, SEMI, EOF]

Some Token Operations

fun eof() = EOF

fun isEof(EOF) = true
| isEof(_) = false

fun binopToString(Add) = "+"
| binopToString(Sub) = "-"
| binopToString(Mul) = "*"
| binopToString(Div) = "/"

fun toString(EOF) = "[EOF]"
| toString(ID(s)) = "[" ^ s ^ "]"
| toString(INT(i)) = "[" ^ (Int.toString(i)) ^ "]"
| toString(OP(opr)) = "[" ^ (binopToString(opr)) ^ "]"
| toString(PRINT) = "[PRINT]"
| toString(LPAREN) = "[" ^ "]"
| toString(RPAREN) = "[" ^ "]"
| toString(COMMA) = "[" ^ "]"
| toString(SEMI) = "[" ^ "]"
| toString(GETS) = "[" ^ "]"

Lexical Analysis 22-11
ml-lex: A Scanner Generator

Format of a .lex File

Header section with SML code
%
Definitions of named regular expressions with form:
name=regexp
%
Rules with pairs of token patterns & SML code having the form:
regexp => SML-expression

In SML-expression, the following special expressions may be used:

yytext     Stands for the string matching the expression
yypos      Character index of the first character of
           yytext in the input character stream
lex()      Ignores current token string and continues lexing
YYBEGIN <state>    Change state of lexer to <state>
**Slip.lex Header Code**

Open Token

Type lexresult = token

Fun eof () = Token.eof()

Fun pluck (SOME(v)) = v

| pluck NONE = raise Fail ("Shouldn't happen -- pluck(NONE)"")

Note: Functions like eof() and pluck can be put in a separate file and then loaded into header.

---

**Slip.lex Definitions and Rules**

Alpha=[a-zA-Z];

AlphaNumUnd=[a-zA-Z0-9_];

Digit=[0-9];

Whitespace=[\t\n];

Any=["'];

"print" => (PRINT);

(alpha)(alphaNumUnd)* => (ID(yytext));

(digit)+ => (INT(pluck(Int.fromString(yytext))));

"." => (OP(Add));

"_" => (OP(Sub));

"*" => (OP(Mul));

"/'" => (OP(Div));

"(" => (LPAREN);

")" => (RPAREN);

"," => (COMMA);

":" => (SEMI);

":" => (GETS);

(Whitespace) => (lex());

(any) => (" Signal a failure exception when encounter unexpected character. A more flexible implementation might raise a more refined exception that could be handled. ")

raise Fail("Slip scanner: unexpected character \" " ^ yytext ^ "\" ")
Using ml-lex to Generate a Scanner

```
[fturbak@sampras slip]
ls -al Slip.lex.sml
ls: cannot access Slip.lex.sml: No such file or directory

[fturbak@sampras slip] ml-lex Slip.lex

Number of states = 27
Number of distinct rows = 10
Approx. memory size of trans. table = 1290 bytes

[fturbak@sampras slip]
ls -al Slip.lex.sml
-rw-rw---- 1 fturbak fturbak 10277 2008-10-23 09:34 Slip.lex.sml
```

Contents of the file Slip.lex.sml

```
structure Mlex = struct
  structure UserDeclarations = struct ... end
  exception LexError
  structure Internal = struct ... end
  fun makeLexer yyinput = ...
end

fun makeLexer yyainput = ...

fun lex () = ...

end
```

SML Digression: Mutable Cells (References)

```
ref : 'a -> 'a ref
  ref <exp> creates a cell whose contents is the value of <exp>.

! : 'a ref -> 'a
  ! <exp> returns the contents of the cell denoted by <exp>.

:= : 'a ref * 'a -> unit
  <exp1> := <exp2> changes the contents of the cell denoted
  by <exp1> to the value denoted by <exp2>.

; : 'a * 'b -> 'b
  <exp1> ; <exp2> first evaluates <exp1>, then evaluates <exp2>,
  and then returns the value of <exp2>. (The value of <exp1>
  value is discarded.)
```

```
val c = ref 17;
val c = ref 17 : int ref
- c;
val it = ref 17 : int ref
- !c;
val it = 17 : int
- fun add x y = x + !y;
val add = fn : int * int -> int
- add c 10;
val it = 27 : int
- !c;
val it = 17 : int
- add c 10;
val it = 27 : int
- c := 42;
val it = () : unit
- add c 10; !c
val it = 42 : int
```
Incrementing a cell in SML

```
fun inc cell = (cell := !cell + 1; !cell)
```

```ml
- val a = ref 0;
val a = ref 0 : int ref
- val b = ref 0;
val b = ref 0 : int ref
- inc a;
val it = 1 : int
- inc a;
val it = 2 : int
- inc b;
val it = 1 : int
- inc a;
val it = 3 : int
- inc b;
val it = 2 : int
```

Scanner Utilities

```
fun stringToScanner str =
  let val done = ref false
  in Mlex.makeLexer (fn n => if (!done) then ""
    else (done := true; str)
  )
  end

fun fileToScanner filename =
  let val inStream = TextIO.openIn(filename)
  in Mlex.makeLexer (fn n => TextIO.inputAll(inStream))
  end

fun scannerToTokens scanner =
  let fun recur () =
    let val token = scanner()
    in if Token.isEof(token) then
      []
    else
      token::(recur())
    end
    in recur()
  end
```

Lexical Analysis  22-19
More Scanner Utilities

```ml
fun printScanner scanner = 
  let fun loop () = 
    let val token = scanner() 
    in if Token.isEof(token) then 
      () 
    else 
      (print(Token.toString(token) ^ "\n"); 
       loop()) 
    end 
  in loop() 
  end

(* Below, "o" is ML's infix composition operator, *)
val stringToTokens = scannerToTokens o stringToScanner 
val fileToTokens = scannerToTokens o fileToScanner 
val printTokensInString = printScanner o stringToScanner 
val printTokensInFile = printScanner o fileToScanner
```

Testing our Scanner

```ml
sum := 5+3;
prod := (print (sum, sum-1), 10*sum);
print(prod);

- Scanner.fileToTokens "test.slip";
val it = 
[ID "sum", GETS, INT 5, OP Add, INT 3, SEMI, ID "prod", GETS, 
 LPAREN, PRINT, LPAREN, ID "sum", COMMA, ID "sum", 
 OP Sub, INT 1, RPAREN, COMMA, INT 10, OP Mul, ID "sum", 
 RPAREN, SEMI, PRINT, LPAREN, ID "prod", RPAREN, SEMI] : 
  Token.token list
```

Sample program in file named "test.slip"
Adding Line Comments

How to add line-terminated comments introduced by #?

```
sum := 5+3; # Set sum to 8
prod := (print (sum, sum-1), # First print sum and (sum-1),
 10*sum); # then set prod to 10*sum
print(prod); # Finally print prod
```

The following ml-lex rule doesn’t work. Why?

```
"#"{any}"\n" => (lex() (* read a line comment *));
```

How can we fix it?

---

Adding Block Comments

How to add block (multi-line) comments delimited by { and }? (They needn’t be nestable yet.)

```
sum := 5+3; # Set sum to 8
prod := (print (sum, sum-1), # First print sum and (sum-1),
 10*sum); # then set prod to 10*sum
{ Comment out several lines:
  x := sum * 2;
  z := x * x;
}
print(prod); # Finally print prod
```
Adding Nestable Block Comments

How to make block (multi-line) comments nestable?

```plaintext
sum := 5+3;  # Set sum to 8
prod := (print (sum, sum-1), # First print sum and (sum-1),
          10*sum);          # then set prod to 10*sum
{ Comment out several lines:
  x := sum * 2;
  ( Illustrate nested block comments:
    y = prod + 3;)
  z := x * x; }
print(prod);  # Finally print prod
```

Can't do this with regular expressions alone.
Need some extra support!

Using Lexer States for Nested Comments

(* Keeping track of nesting level of block comments *)
val commentNestingLevel = ref 0

fun incrementNesting() =
  (print "Incrementing comment nesting level";
   commentNestingLevel := (!commentNestingLevel) + 1)

fun decrementNesting() =
  (print "Decrementing comment nesting level";
   commentNestingLevel := (!commentNestingLevel) - 1)

%%
% COMMENT;
alpha=[a-zA-Z];
alphaNumUnd=[a-zA-Z0-9_];
digit=[0-9];
whitespace=[\ \t\n];
any=[^];
%%
rules shown on next slide
Lexical Analysis

Lexer Rules for Nested Comments

<INITIAL>"print" => (PRINT);
<INITIAL>(alpha)(alphaNumUnd)* => (ID(yytext));
<INITIAL>{digit}+ => (INT(pluck(Int.fromString(yytext))));
<INITIAL>"*" => (OP(Add));
<INITIAL>"." => (OP(Sub));
<INITIAL>"**" => (OP(Mul));
<INITIAL>"/" => (OP(Div));
<INITIAL>"(" => (LPAREN);
<INITIAL>")" => (RPAREN);
<INITIAL>"->" => (COMMA);
<INITIAL>";" => (SEMI);
<INITIAL>"#.*\n" => (lex() (* read a line comment *));
<INITIAL>"{" => (YBEGIN COMMENT; incrementNesting(); lex());
<INITIAL>{whitespace} => (lex());
<INITIAL>"{" => (incrementNesting(); lex());
<INITIAL>"}" => (decrementNesting(); if (!commentNestingLevel) = 0 then

(YBEGIN INITIAL; lex()) else lex());
<INITIAL>{any} => (lex());
(any) => (lex());
raise Fail("Slip scanner: unexpected character \"^ yytext ^ "\n");

A more flexible implementation might raise a more refined
exception that could be handled. *)