Regular Language Applications

Monday, October 22, 2007
Revised Version, Wednesday, October 24, 2007
Reading: Stoughton 3.14, Appel Chs. 1 and 2

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Some Applications of Regular Languages

o Efficient string searching

o Pattern matching with regular expressions
(example: Unix grep utility)

o Lexical analysis (a.k.a. scanning, tokenizing) in a compiler
(example: ML-Lex).

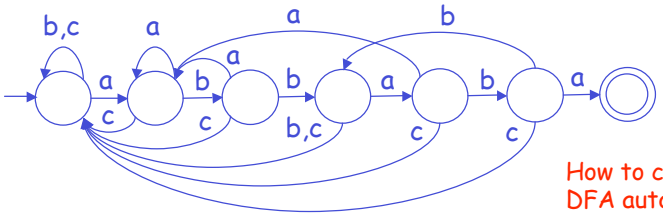
Naïve String Searching

How to search for abbaba in abbabcabbabbaba?

a	b	b	a	b	c	a	b	b	a	b	b	a	b	a
a	b	b	a	b	a									
	a	b	b	a	b	a								
		a	b	b	a	b	a							
			a	b	b	a	b	a						
				a	b	b	a	b	a					
					a	b	b	a	b	a				
						a	b	b	a	b	a			
							a	b	b	a	b	a		
								a	b	b	a	b	a	
									a	b	b	a	b	a

More Efficient String Searching

Knuth-Morris-Pratt algorithm: construct a DFA for searched-for string, and use it to do searching.



How to construct this DFA automatically?

a	b	b	a	b	c	a	b	b	a	b	b	a	b	a
a	b	b	a	b	a									
						a	b	b	a	b	a			
									a	b	b	a	b	a

Pattern Matching with Regular Expressions

Can turn any regular expression (possibly extended with complement, intersection, and difference) into a DFA and use it for string searching.

This idea is used in many systems/languages:

- **grep**: Unix utility that searches for lines in files matching a pattern. ("grep" comes from g/re/p command in the ed editor.)
- **sed**: Unix stream editor
- **awk**: text-manipulation language
- **Perl**: general-purpose programming language with built-in pattern matching
- **Java**: recent versions have support for regular expressions.

Some grep Patterns

Pattern	Matches
c	the character 'c'
.	any character except newline
[a-zA-Z0-9]	any alphanumeric character
[^d-g]	any character except lowercase d,e,f,g
\w	synonym for [a-zA-Z0-9]
\W	synonym for [^a-zA-Z0-9]
^	beginning of line
$	end of line
r_1r_2	r_1 followed by r_2, where r_1, r_2 are reg. exps.
$r_1\|r_2$	r_1 or r_2
r*	zero or more rs, where r a reg. exp.
r+	one or more rs
r?	zero or one rs
$r\{n\}$	exactly n rs
$r\{n,\}$	n or more rs
$r\{n,m\}$	between n and m rs
(r)	r (parens for grouping)
\n	the substring previously matched by the nth parenthesized subexpression of the regular expression

Some grep Examples

As a rule, grep patterns should be double-quoted to prevent Linux from interpreting certain characters specially. (But \ is still a problem, as we'll soon see.)

```
grep "a.*b.*c.*d" words.txt

grep "//.*seed.*" *.java

grep "intBetween(box" *.java

find . | xargs grep "intBetween(box"

find -exec grep "intBetween(box" {} \;
```


Escapes in Grep Patterns

grep patterns use special metacharacters that (at least in some contexts) do not stand for themselves:

? + | ( ) { } . * ^ $ \ [ ]

In order to reference the blue characters as themselves, it is necessary to escape them with a backslash. E.g.,

$ is a pattern that matches the end of line
\$ is a pattern that matches the dollar sign character
\\ is a pattern that matches the backslash character
\\\\ is a pattern that matches two backslash characters in a row

But the backslash character is also an escape character in Linux. To safely pass backslashes from Linux to grep, you should* type *two* backslashes for every backslash you wish to send to grep. E.g.

grep "\\$" searches for the dollar sign character
grep "\\\\" searches for a single backslash
grep "\\\\\\\\" searches for two backslash characters in a row

*In some, but not all cases, a single backslash will suffice.

What About the Red Metacharacters?

The red metacharacters are handled in a rather confusing way:

? + | ( ) {

In the **basic regular expressions** used by **grep**, these characters stand for themselves and must be escaped to have the metacharacter meaning. E.g.

grep "(ab)+" searches for the substring "(ab)+"

grep "\\(ab\\)\\+" searches for any nonempty sequence of abs.

grep "\\(.\\)\\1" searches for two consecutive occurrences of the same character

In the **extended regular expressions** used by **grep -E** and **egrep**, these characters are metacharacters and must be escaped to stand for themselves.

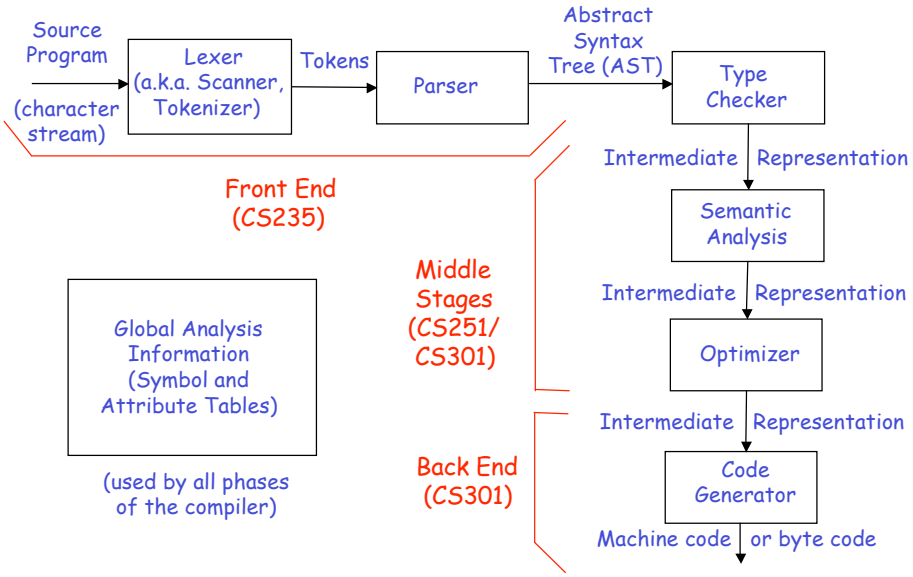
egrep "(ab)+" searches for any nonempty sequence of abs.

egrep "\\(ab\\)\\+" searches for the substring "(ab)+"

egrep "(.)\\1" searches for two consecutive occurrences of the same character

Moral of the story: use egrep instead of grep!

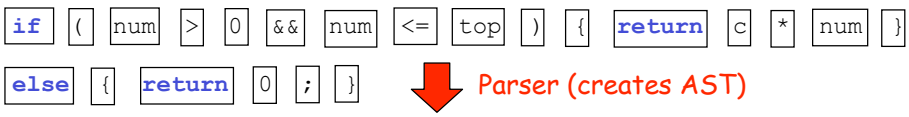
Compiler Structure



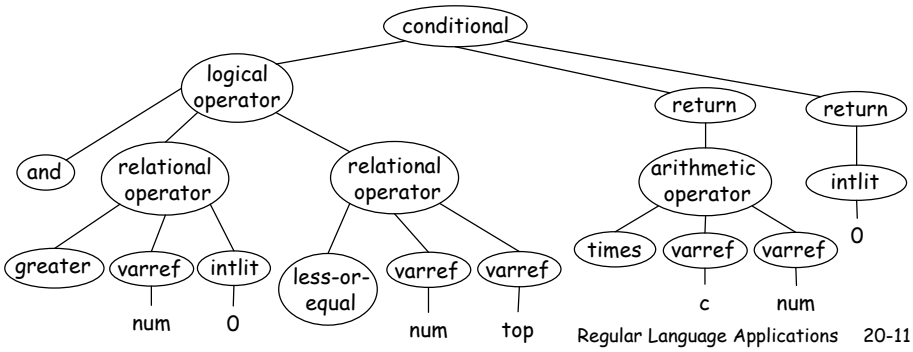
Front End Example

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

Lexer (ignores whitespace, comments)



Parser (creates AST)



Sample English Description of Lexer Rules

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.

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 12×10^{34}).

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 are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

Some ML-Lex Patterns

Pattern	Matches
"abc"	the literal string of characters abc
.	any character except newline
[a-zA-Z0-9]	any alphanumeric character
[^d-g]	any character except lowercase d,e,f,g
r_1r_2	r_1 followed by r_2, where r_1, r_2 are reg. exps.
$r_1 \mid r_2$	r_1 or r_2
r^*	zero or more rs, where r a reg. exp.
r^+	one or more rs
$r?$	zero or one rs
(r)	r (parens for grouping)
{*REName*}	regular expression with name *REName*

Regular expressions for some tokens

`if`	if keyword
`[a-zA-Z_][a-zA-Z0-9_]*`	identifiers (variable names)
`[0-9]+(E[0-9]+)?`	integers

How should the following be split into tokens?

if

if89

12

12E34

12Eat34

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):

```
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:

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

How do we represent these tokens in SML?

Sum-of-Product Data Types in 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

```
(* 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 (4,Cons (6,Nil))) : int mylist
```


A Token Datatype

```
datatype binop = Add | Mul | Sub | Div

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

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

```
[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, EOF]
```

Some Token Operations

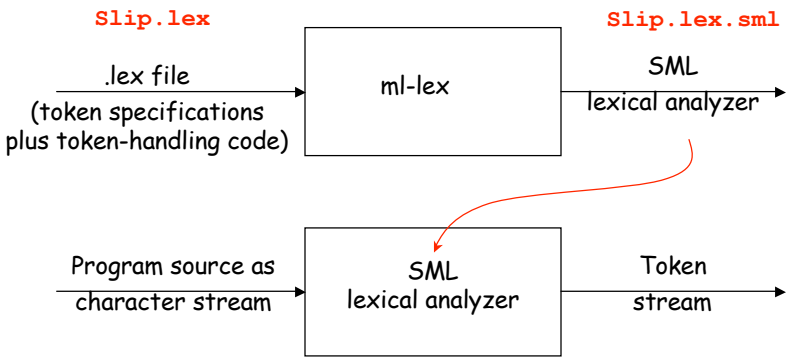
```
fun eof() = EOF

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) = "[:=]"
```


ml-lex



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 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 the current token string and continues lexing
YYBEGIN <state>	Changes state of lexer

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)")
```

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 ^ "\"")
```


Mutable Cells (References) in SML

```
- val c = ref 17;
val c = ref 17 : int ref

- c;
val it = ref 17 : int ref

- !c;
val it = 17 : int

- fun add_c x = x + !c;
val add_c = fn : int -> int

- add_c 10;
val it = 27 : int

- c := 42;
val it = () : unit

- add_c 10;
val it = 52 : int
```

Counters in SML

```
fun makeCounter () =
  let val cell = ref 0
  in  fn () => (cell := !cell + 1; !cell)
  end
```

```
- val a = makeCounter();
val a = fn : unit -> int

- val b = makeCounter();
val b = fn : unit -> int

- a();
val it = 1 : int

- a();
val it = 2 : int

- b();
val it = 1 : int

-a();
val it = 3 : int

- b();
val it = 2 : int
```


Creating a Scanner

```
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
```

Creating a Scanner (Part 2)

```
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

```
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
```

Adding Line Comments

```
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 rule doesn't work. Why?

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

How can we fix it?

Adding Block Comments

```
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 Nested Block Comments

```
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 comments:
    y = prod + 3:}
  z := x * x; }
print(prod); # Finally print prod
```


Lexer States, Part 1

```
(* 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)
%%
%s COMMENT;
alpha=[a-zA-Z];
alphaNumUnd=[a-zA-Z0-9_];
digit=[0-9];
whitespace=[\ \t\n];
any= [^];
%%
```

Lexer States, Part 2

```
<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>":=" => (GETS);
<INITIAL>"#".*"\n" => (lex() (* read a line comment *));
<INITIAL>"{" => (YYBEGIN COMMENT; incrementNesting(); lex());
<INITIAL>{whitespace} => (lex());
<COMMENT>"{" => (incrementNesting(); lex());
<COMMENT>"}" => (decrementNesting(); if (!commentNestingLevel) = 0 then
                    (YYBEGIN INITIAL; lex()) else lex());
<COMMENT>{any} => (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 ^ "\""));
```