Lexing

CS 132: Compiler Design

January 22, 2001

What is Lexing?

- Lexing: breaking program source text into "words" called *tokens*.
  - Other names for process: tokenizing, scanning
  - Unsurprisingly, the part of the compiler that does this is called the lexer (or the scanner, etc.)
- Tokens include keywords, punctuation, identifiers, constants, etc.

Example from Appel

- Input

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
   return 0.0;
}
```

- Output

FLOAT, **ID**(match0), **LPAREN**, **CHAR**, **STAR**, **ID**(s), **RPAREN**, **LBRACE**, **IF**, **LPAREN**, **BANG**, **ID**(strncmp), **COMMA**, **STRING**(0.0), **COMMA**, **INT**(3), **RPAREN**, **RPAREN**, **RETURN**, **REAL**(0.0), **SEMI**, **RBRACE**, **EOF**

Why Do (Separate) Lexing?

- Simplifies the work of the parser
  - Parsing is generally more complex and expensive
  - Hides unnecessary details
    - Source comments
    - Whitespace (sometimes)
    - Case of letters (sometimes)
- Benefits of modularity
  - Easier to understand
  - Easier to modify

Token Definitions

- Recognizing tokens is a pattern-matching problem.
- These patterns can usually be described using *regular expressions.*

Regular Sets over Characters

a	{"a"}	
`"xyz"`	`{"xyz"}`	
ε	{""}	[sometimes denoted λ instead]
M | N	$L(M) \cup L(N)$	
MN	$\{xy \mid x \in L(M), y \in L(N)\}$	
M^*	$\{x_1 \ldots x_n \mid x_i \in L(M), n \geq 0\}$	
M^+	$\{x_1 \ldots x_n \mid x_i \in L(M), n \geq 1\}$	
M?	$L(M) \cup \{\varepsilon\}$	
.	{c | c is a character other than newline}	
`[A-Za-z01]`	{c | c is a character between A and Z or between a and z or a zero or a one}	
`[^A-Za-z01]`	{c | c is a character not between A and Z and not between a and z and not a zero or a one}	

Example [from RX library]

```
M[ou]'?am+[ae]r .*([AEae]l[- ])?[GKQ]h?[aeu]+([dtz][dhz]?)+af[iy]
```

```
Muammar Qaddafi
Mo'ammar Gadhafi
Muammar Kaddafi
Muammar Qadhafi
Moammar El Kadhafi
Muammar Gadafi
Mu'ammar al-Qadafi
Moamer El Kazzafi
Moamar al-Gaddafi
Mu'ammar Al Qathafi
Muammar Al Qathafi
Mo'ammar el-Gadhafi
Moamar El Kadhafi
Muammar al-Qadhafi
Mu'ammar al-Qadhdhafi
Mu'ammar Qadafi
Moamar Gaddafi
Mu'ammar Qadhdhafi
Muammar al-Khaddafi
Mu'amar al-Kadafi
Muammar Ghaddafy
Muammar Ghadafi
Muammar Ghaddafi
Muamar Kaddafi
Muammar Quathafi
Muammar Gheddafi
Muamar Al-Kaddafi
Moammar Khadafy
Moammar Qudhafi
Mu'ammar al-Qaddafi
Mu'ammar Muhammad Abu Minyar al-Qadhafi
```

Defining Tokens with Regexps

- The keyword `if`.
- Valid integer constants
- Real numbers (requiring at least one digit on each side of the decimal point, with optional sign and optional optionally-signed exponent)

Defining Tokens with Regexps

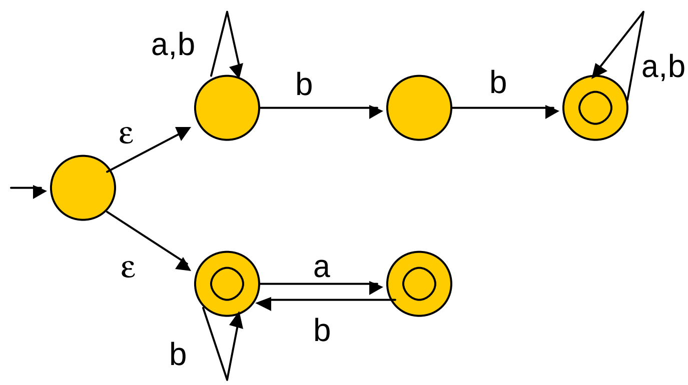
- SML (non-symbolic) identifiers, which must begin with a letter, and then may have any string of letters, digits, underscores, and primes

- Ada identifiers, which must begin with a letter and then may have any string of letters, digits and underscores, with the proviso that underscores may only occur one at a time and cannot be the last character

Nondeterministic Finite Automata

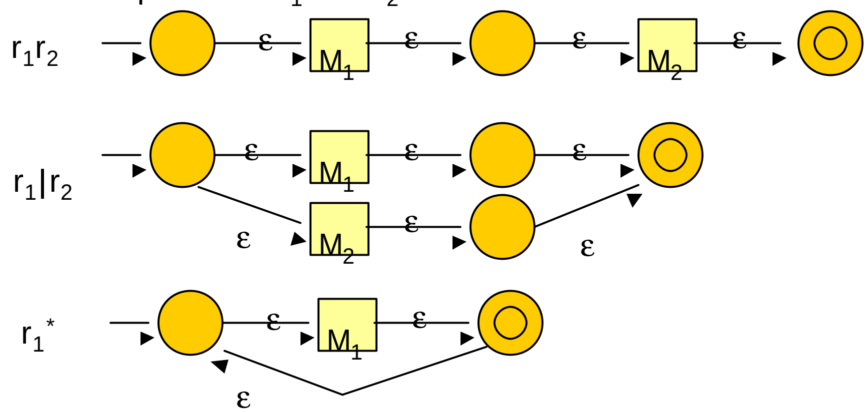
- An NFA is specified by:
  - An alphabet Σ of input symbols (characters)
  - A finite set Q of possible states
  - A transition relation $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$
  - An initial state q_0
  - A set $F \subseteq Q$ of final states.

Picturing NFAs



Regexps to NFAs

- Assume M_1 and M_2 are NFA's that accept regular expressions r_1 and r_2. Then:



NFAs to DFAs

- When simulating an NFA we keep track of the set of possible states
- This can be formalized as *subset construction*
  - Construct a deterministic finite automaton whose states represent sets of NFA states

Notes

- An NFA with *O(n)* states can require a DFA with *O(2ⁿ)* states
  - e.g., $(a|b)^*a(a|b)^n$
- The DFA generated by the subset construction need not be optimal

Lexer Generation

- Straightforward to automate construction of automata
  - Many lexer generators exist: Lex, Flex, ML-Lex, …
- We will use ML-Lex
  - Input: token definitions and actions to take for each token recognized
  - Output: a function `lex` such that the call `lex()` finds the next token in the input stream and performs the corresponding action.

Simple ML-Lex Input File

```
datatype lexresult = IF | ID | INT | REAL | DOT | EOF
fun eof() = EOF
exception Error
%%
%%
"if"                              =>     (IF);
[a-z][a-z0-9]*                    =>     (ID);
[0-9]+                            =>     (INT);
([0-9]+"."[0-9]*)|("."[0-9]+)     =>     (REAL);
"."                               =>     (DOT);
.                                 =>     (raise Error);
```

Disambiguation

- What should happen to `3.0` or `if7` or `if` ?

- Disambiguation rules:
  - **Longest match**: the longest prefix of the input that can match any regular expression is taken as the next token
  - **Rule priority**: if two rules match the same longest prefix, the first rule is chosen.

Larger Example: Setup

```
structure Tokens =
struct
   type pos = int
   datatype token = EOF of pos*pos
                  | IF of pos*pos
                  | ID of string*pos*pos
                  | NUM of int*pos*pos
                  | REAL of real*pos*pos
                  ...
end
```

Larger Example: ML-Lex Input

```
type lexresult = Tokens.token
fun eof() = Tokens.EOF(0,0)
%%
digits=[0-9]+;
%%
if              => (Tokens.IF(yypos, yypos+2));
[a-z][a-z0-9]* => (Tokens.ID(yytext, yypos, yypos+(size yytext)));
{digits}        => (Tokens.NUM(valOf(Int.fromString yytext),
                               yypos, yypos+(size yytext)));
({digits}"."[0-9]*})|([0-9]*"."{digits})
                => (Tokens.REAL(valOf(Real.fromString yytext),
                                yypos, ypos+(size yytext)));
("--"[a-z]*"\n")|(" "|"\n"|"\t") => (continue());
.               => (print "illegal character\n";
                    continue());
```

Reserved Words

- Many languages have *reserved words* which cannot be used as identifiers.
  - **`if`**, **`return`**, etc.
  - Not all languages, e.g., PL/I:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Recognizing Reserved Words

- Some possibilities
  - Lexer directly distinguishes reserved words from identifiers
    - Convenient for automatically-generated lexers (but automaton is larger)
    - Harder to do efficiently in handwritten lexers
  - Lexer only looks for identifiers
    - Then check each identifier found, to see whether it's actually a reserved word

Lexing Challenges

- Non-regular token specifications
  - Fortran Hollerith constants: 9Hcompilers
  - Skipping nested comments
- Layout significance
  - Position of code determines meaning
  - e.g., offside rule

```
if x = 0 then              x = 1
   if y = 0 then           y = x + z
      z := 0                   where
else                           x = 2
   z := 1                      z = 4
                           z = x + y
```

Lexing Challenges

- Context-dependent tokenization
  - May have to use lookahead or other information
  - e.g., Fortran

```
DO 10 I = 1,15
DO 10 I = 1.15
REAL X
REAL X = 3.5
INTEGER FUNCTION A(I)
```

Start States

- It is sometimes convenient to define several lexers and switch between them at run-time
- ML-Lex rules can be prefixed by a list of *start states*
  - Rules only active when the lexer is in these states
  - States need to be predeclared with `%s` line
    - Except for predefined state `INITIAL`
  - Actions can switch states by calling `YYBEGIN` function with the name of the state.

States Example

```
type lexresult = Tokens.token
fun eof() = Tokens.EOF(0,0)
%%
%s COMMENT;
digits=[0-9]+;
%%
<INITIAL>if       => (Tokens.IF(yypos, yypos+2));
<INITIAL>[a-z]+   => (Tokens.ID(yytext, yypos, yypos+(size yytext)));
  ...etcetera...
<INITIAL>"(*"     => (YYBEGIN COMMENT; continue());
<COMMENT>"*)"     => (YYBEGIN INITIAL; continue());
<COMMENT>.        => (continue());
```