

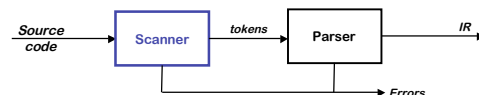
Lexical Analysis

Implementing Scanners

LEX: A Lexical Analyzer Tool

Copyright 2009, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at Instituto Superior Técnico (IST/UTL) have explicit permission to make copies of these materials for their personal use.

Scanners and Parsers



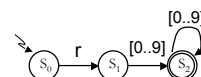
Scanner

- Maps stream of characters into words
 - Basic unit of syntax
 - $x = x + y$; becomes $\langle id, x \rangle \langle eq, = \rangle \langle id, x \rangle \langle pl, + \rangle \langle id, y \rangle \langle sc, ; \rangle$
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Scanners Construction

- Straightforward Implementation
 - Input: REs
 - Construct an NFA for each RE
 - Combine NFAs using ϵ -transitions (alternation in Thompson's construction)
 - Create a DFA using the subset construction
 - Minimize the resulting DFA
 - Create an executable code for the DFA

Table-driven Scanners

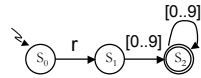


```

char ← nextChar()
state ← s0
while (char ≠ eof)
  state ← δ(state, char)
  char ← nextChar()
end while
if state ∈ SF
  then report acceptance
  else report failure
  
```

δ	r	[0..9]	Other
s ₀	s ₁	s _e	s _e
s ₁	s _e	s ₂	s _e
s ₂	s _e	s ₂	s _e
s _e	s _e	s _e	s _e

Direct-Coded Scanners



```

goto s0
s0: char ← nextChar()
    if(char = 'r')
        then goto s1
    else goto s_e
s1: char ← nextChar()
    if('0' ≤ char ≤ '9')
        then goto s2
    else goto s_e
s2: char ← nextChar()
    if('0' ≤ char ≤ '9')
        then goto s2
    else goto s_e
s_e: report failure
  
```

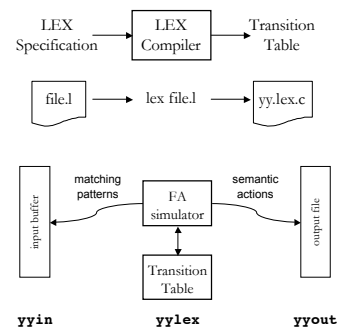
Scanners in Practice

- Uses automated tools to construct a Lexical Analyzer
 - Given a set of tokens defined using regular expressions
 - Tools will generate a character stream tokenizer by constructing a DFA
- Common Scanner Generator Tools
 - lex in C
 - JLex in java

LEX: A Lexical Analyzer Tool

- Input:
 - File with suffix .l
 - Three (3) regions delimited by the marker '%' corresponding to:
 - **Declarations:** macros and language declaration between '%{' e '%}'
 - **Rules:** regular expression and corresponding action between brackets { } to be executed when the analyzer matches the regular expression.
 - **Code:** support functions
- Output:
 - Generates a C function named **yylex** in the file **lex.yy.c**, with the command **lex file.l** and compiled using -ll switch
 - **flex** uses the library -lfl

LEX: A Lexical Analyzer Tool



LEX: Regular Expressions

x	the character x
"x"	the character x, even if it is a special character
\x	the character x, even if it is a special character
x\$	the character x at the end of a line
^x	the character x at the beginning of a line
x?	Zero or one occurrence of x
x+	One or more occurrences of x
x*	Zero or more occurrences of x
xv	the character x followed by the character v
x v	the character x or the character v
[az]	the character a or the character z
[a-z]	from character a to character z
[^a-z]	Any character except from a to z
x{n}	n occurrences of x
x{m,n}	between m and n occurrences of x
x/y	x if followed by y (only x is part of the pattern)
.	Any character except \n
(x)	same as x, parentheses change operator priority
<<EOF>>	end of file

LEX: Regular Expression Short-hands

- Allow Regular Expressions Simplification
- In the declaration region; consists of an identifier followed by a regular expression
- Use between brackets [] in subsequent regular expressions

```

DIGIT    [0-9]
INT      {DIGIT}+
EXP      [Ee][+-]?{INT}
REAL     {INT} "." {INT} ({EXP})?

```

LEX: Example lex.l

```

/* definitions of constants */
#define LT 128
#define LE 129
#define EQ 130
#define NE 131
#define GT 132
#define GE 133
#define IF 134
#define THEN 135
#define ELSE 136
#define ID 137
#define NUM 138
#define RELOP 139

/* regular definitions */
#define (delim) +
#define letter [A-Za-z]
#define digit [0-9]
#define id (letter)(letter)(digit)*
#define number (digit)(+((digit)+)?(E|+)(digit)+)?

/* no action and no return */
if
{ return(IF); }
then
{ return(THEN); }
else
{ return(ELSE); }
(id)
{ install_id(); return(ID); }
(number)
{ install_num(); return(NUM); }
"<"
{ return(LT); }
"<="
{ return(LE); }
"=="
{ return(EQ); }
"!="
{ return(NE); }
">"
{ return(GT); }
">="
{ return(GE); }

%%

int yywrap() { return 1; }

int main()
{
    int n, tok;
    n = 0;
    while(1) {
        tok = yylex();
        if(tok == 0) break;
        printf("token matched is %d\n", tok);
        n++;
    }
    printf("number of tokens matched is %d\n", n);
    return 0;
}

```

LEX: Executing yylex

```

%> more in.txt
a = 4
b = 12.5
if a <= 0 then a = 1

%> ./a.out < in.txt
Installing an identifier a
token matched is 137
Installing a number 4
token matched is 138
Installing an identifier b
token matched is 137
Installing a number 12.5
token matched is 138
token matched is 134
Installing an identifier a
token matched is 137
Installing a number 0
token matched is 138
token matched is 135
Installing an identifier a
token matched is 137
Installing a number 1
token matched is 138
number of tokens matched is 14

```



LEX: Handling of Regular Expressions

- Disambiguating in case of multiple regular expression matching:
 - Longest input sequence is selected
 - If same length, first in specification is selected
- Note: Input sequence length not length of regular expression length:

```
%%  
dependent      printf(Found 'dependent'\n");  
[a-z]+         ECHO;
```



LEX: Context Sensitive Rules

- Set of regular expressions *activated* by a '**BEGIN**' command and identified by '%s' in the declaration region
- The regular expression in each set are preceded by the identifier between < and >. The '**INITIAL**' identifier indicates the global rules permanently active.
- At any given time only the global rules and at most one of the set of context sensitive rules can be active by the invocation of the '**BEGIN**' command.



LEX: Global Variables (C lang.)

- **char yytext[]**, string containing matched input text described by a given regular expression
- **int yyleng**, length of string containing matched input text
- **int yylineno**, line number of input file where the last character of the matched input text lies. With flex use the option -l or include **%option yylineno** or **%option lex-compat** in input file .l
- **FILE *yyin**, file pointer where from the input characters are read
- **FILE *yyout**, file pointer where to the output text is written using the **ECHO** macro or other programmer defined functions
- **YYSTYPE yyval**, internal variable used to carry over the values between the lexical analyzer and other tools, namely YACC



LEX: Auxiliary Functions (C lang.)

- **int yylex(void)**, lex generated function that implements the lexical analysis. Returns a numeric value identifying the matched lexical element (i.e. as identified by **yy.tab.h**) or 0 (zero) when EOF is reached
- **int yywrap(void)**, programmer defined function invoked when the EOF of the current input file is reached. In the absence of more additional input files this function returns 1 (one). It returns 0 (zero) otherwise and the **yyin** variable should be pointing to the new input file
- **void yymore(void)**, function invoked in a semantic action allowing the matched text to be saved and concatenated to the following matched text
- **void yyless(int n)**, function invoked in a semantic action allowing the **n** characters of **yytext** to be reconsidered for processing.



Summary

- Scanner Construction
 - Table-driven
 - Direct-coded
- Lex: A Lexical Analyzer Tool