Chapter 3
Lexical and Syntactic Analysis

"Syntactic sugar causes cancer of the semicolon.
A. Perlis"

Contents

3.1 Chomsky Hierarchy
3.2 Lexical Analysis
3.3 Syntactic Analysis
Lexical Analysis

Purpose: transform program representation
Input: printable Ascii characters
Output: tokens
Discard: whitespace, comments

Defn: A token is a logically cohesive sequence of characters representing a single symbol.

Tokens, Patterns, and Lexemes

Token
- Token name - category of lexical unit (e.g. keyword, identifier, integer, operator, etc.)
- Attribute value - specific representation (e.g. if, myVar, 602, +, etc.)

Pattern – description of the form that the lexemes may take

Lexeme – sequence of characters making up a token instance
Examples of Tokens

<table>
<thead>
<tr>
<th>TOKEN</th>
<th>INFORMAL DESCRIPTION</th>
<th>SAMPLE LEXEMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>characters 'i', 'f'</td>
<td>if</td>
</tr>
<tr>
<td>else</td>
<td>characters 'e', 'l', 's', 'e'</td>
<td>else</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; or &gt; or &lt;= or &gt;= or == or !=</td>
<td>&lt;=, !=</td>
</tr>
<tr>
<td>id</td>
<td>letter followed by letters and digits</td>
<td>pi, score, D2</td>
</tr>
<tr>
<td>number</td>
<td>any numeric constant</td>
<td>3.14159, 0.602e23</td>
</tr>
<tr>
<td>literal</td>
<td>anything but &quot;,&quot;, surrounded by &quot;'s&quot;</td>
<td>&quot;core dumped&quot;</td>
</tr>
</tbody>
</table>

Figure 3.2: Examples of tokens

Why a Separate Phase?

Simpler, faster machine model than parser
75% of time spent in lexer for non-optimizing compiler
Differences in character sets
End of line convention differs
Tricky Problems When Recognizing Tokens

Fortran 90
- \texttt{DO 5 I = 1.25}
- \texttt{DO 5 I = 1,25}
- \textit{Solution – treat blanks and other white space as significant}

PL/I
- \texttt{IF IF = THEN THEN}
  \hspace{1cm} \texttt{THEN = ELSE}
  \hspace{1cm} \texttt{ELSE}
  \hspace{1cm} \texttt{ELSE = IF}
- \textit{Solution – reserve keywords so they may not conflict with identifiers}

Building a Lexical Analyzer

Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description

Design a state diagram that describes the tokens and write a program that implements the state diagram

Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram
Chomsky Hierarchy of Formal Languages

- Type 0 – Recursively enumerable
- Type 1 – Context-Sensitive
- Type 2 – Context-Free
- Type 3 – Regular

Elements of Regular Languages

- A symbol from the alphabet
- The empty string \( \varepsilon \) (epsilon)
- Any string derivable from one of the operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition and Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of ( L ) and ( M )</td>
<td>( L \cup M = { s \mid s \text{ is in } L \text{ or } s \text{ is in } M } )</td>
</tr>
<tr>
<td>Concatenation of ( L ) and ( M )</td>
<td>( LM = { st \mid s \text{ is in } L \text{ and } t \text{ is in } M } )</td>
</tr>
<tr>
<td>Kleene closure of ( L )</td>
<td>( L^* = \bigcup_{i=0}^{\infty} L^i )</td>
</tr>
<tr>
<td>Positive closure of ( L )</td>
<td>( L^+ = \bigcup_{i=1}^{\infty} L^i )</td>
</tr>
</tbody>
</table>

Figure 3.6: Definitions of operations on languages

Regular Expressions

$L = \{a\}$ is denoted by $a$
$L = \{\varepsilon\}$ is denoted by $\varepsilon$
$L = L_1 \cup L_2$ is denoted by $R_1 \mid R_2$ (sometimes $R_1 + R_2$)
$L = L_1 L_2$ is denoted by $R_1 R_2$
$L = L_1^*$ is denoted by $R_1^*$

Finite Automata

Every regular expression may be converted into an equivalent recognizer called a finite automaton, or finite state machine.

A finite automaton consists of a set of states, one of which is the start state and one or more of which are the final states, an alphabet of symbols, and a state transition function which defines how the automaton moves from one state to another on a symbol in the alphabet.
Example Transition Diagram #1

Figure 3.13: Transition diagram for `relop`

Example Transition Diagram #2

Figure 3.14: A transition diagram for `id`'s and keywords
Example Transition Diagram #3

![Diagram](image1)

Figure 3.15: Hypothetical transition diagram for the keyword then

---

Example Transition Diagram #4

![Diagram](image2)

Figure 3.16: A transition diagram for unsigned numbers
Finite Automata

Every regular expression may be converted into a *finite automaton*.

A finite automaton may be *nondeterministic*, meaning there is more than one transition possible from a given state on a given input, or *deterministic* if there is at most one such transition.

For every NFA, there is an equivalent DFA.
For every DFA, there is an equivalent DFA with a minimum number of states.

Finite Automata for Lexical Tokens

[Diagrams of automata for IF, ID, NUM, REAL, white space, and error]
Combined Finite Automaton

Translation of Regular Expressions to Nondeterministic Finite Automata
Regular Expressions Translated into an NFA

NFA Converted to DFA
Lexical Analyzer Generation in C

Lexical Analyzer Generation in Java

Specification of Tokens
(Clite.jflex)

JFlex

Lexical Analyzer Class
(CliteLexer.java)
Structure of JFlex Programs

declarations
%

translation rules
%

auxiliary functions

Declarations include variables and manifest constants

Translation rules are of the form

Pattern { Action}

JFlex Regular Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Matches</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>the one non-operator character</td>
<td>a</td>
</tr>
<tr>
<td>\c</td>
<td>character c literally</td>
<td>*</td>
</tr>
<tr>
<td><em>s</em></td>
<td>string s literally</td>
<td><em>s</em></td>
</tr>
<tr>
<td>.</td>
<td>any character but newline</td>
<td>.</td>
</tr>
<tr>
<td>$</td>
<td>beginning of a line</td>
<td>$</td>
</tr>
<tr>
<td>[s]</td>
<td>end of a line</td>
<td>[s]</td>
</tr>
<tr>
<td>[s]</td>
<td>any one of the characters in string</td>
<td>[s]</td>
</tr>
<tr>
<td>[^s]</td>
<td>any one character not in string</td>
<td>[^s]</td>
</tr>
<tr>
<td>r*</td>
<td>zero or more strings matching</td>
<td>a*</td>
</tr>
<tr>
<td>r+</td>
<td>one or more strings matching</td>
<td>a+</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one r</td>
<td>a?</td>
</tr>
<tr>
<td>r{m, n}</td>
<td>between m and n occurrences of</td>
<td>a{1, 5}</td>
</tr>
<tr>
<td>r1 r2</td>
<td>an r1 followed by an r2</td>
<td>ab</td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
<td>on r1 or an r2</td>
</tr>
<tr>
<td>r1+</td>
<td>same as r</td>
<td>r1+</td>
</tr>
<tr>
<td>r1/r2</td>
<td>r1 when followed by r2</td>
<td>abc/123</td>
</tr>
</tbody>
</table>

Figure 3.8: Lex regular expressions
**JFlex Example**

```java
faculty% cat Clite.jflex
%%
{
    private void echo ()
    {
        System . out . print (yytext ());
    }

    public int position () { return yycolumn; }
}

%class CliteLexer
%unicode
%line
%column
%function next_token
%type Token
%eofval{
    return new Token (Symbol . EOF);
}%eofval
```

Fig. 2.7  *Clite* grammar: lexical level

```
Identifier → Letter { Letter | Digit }
    Letter → a | b | ... | z | A | B | ... | Z
    Digit → 0 | 1 | ... | 9
Literal → Integer | Boolean | Float | Char
    Integer → Digit { Digit }
    Boolean → true | False
    Float → Integer . Integer
    Char → ‘ ASCII Char ’
```
JFlex Example

Comment = "//".*
WhiteSpace = [ \t\n]
Identifier = [[:letter:]][[:letter:] | [:digit:]]*
Integer = [:digit:] [:digit:]*
Boolean = true | false
Float = {Integer}"."{Integer}
Char = "'"'"'

Fig. 2.7 Clite Grammar: Statements

Program → int main ( ) { Declarations Statements }
Declarations → { Declaration }
Declaration → Type Identifier [[ Integer ]] , Identifier [[ Integer ]]
Type → int | bool | float | char
Statements → { Statement }
Statement → ; | Block | Assignment | IfStatement | WhileStatement
Block → { Statements }
Assignment → Identifier [[ Expression ]] = Expression ;
IfStatement → if ( Expression ) Statement [ else Statement ]
WhileStatement → while ( Expression ) Statement
JFlex Example

```
{Comment} { echo (); }
{WhiteSpace} { echo (); }
"," { echo (); return new Token (TokenClass . COMMA); }
";" { echo (); return new Token (TokenClass . SEMICOLON); }
"{" { echo (); return new Token (TokenClass . LBRACE); }
"}" { echo (); return new Token (TokenClass . RBRACE); }
```

Fig. 2.7 Clite Grammar: Expressions

```
Expression → Conjunction { || Conjunction }
Conjunction → Equality { && Equality }
Equality → Relation [ EquOp Relation ]
EquOp → == | !=
Relation → Addition [ RelOp Addition ]
RelOp → < | <= | > | >=
Addition → Term { AddOp Term }
AddOp → + | -
  Term → Factor [ MulOp Factor ]
MulOp → * | / | %
Factor → [ UnaryOp ] Primary
UnaryOp → - | !
Primary → Identifier [ [ Expression ] ] | Literal | ( Expression ) |
Type ( Expression )
```
JFlex Example

"("  { echo (); return new Token (TokenClass . LPAREN); }
")"  { echo (); return new Token (TokenClass . RPAREN); }
"["  { echo (); return new Token (TokenClass . LBRACK); }
"]"  { echo (); return new Token (TokenClass . RBRACK); }
"=="  { echo (); return new Token (TokenClass . ASSIGN); }
"| |"  { echo (); return new Token (TokenClass . OR); }
"&&"  { echo (); return new Token (TokenClass . AND); }
"+"   { echo (); return new Token (TokenClass . PLUS); }
"-"   { echo (); return new Token (TokenClass . MINUS); }
"*"   { echo (); return new Token (TokenClass . TIMES); }
"/"   { echo (); return new Token (TokenClass . SLASH); }
"%"   { echo (); return new Token (TokenClass . MOD); }
"=="  { echo (); return new Token (TokenClass . EQ); }
"!="  { echo (); return new Token (TokenClass . NE); }
"<"   { echo (); return new Token (TokenClass . LT); }
"<="  { echo (); return new Token (TokenClass . LE); }
">"   { echo (); return new Token (TokenClass . GT); }
">="  { echo (); return new Token (TokenClass . GE); }
"!"   { echo (); return new Token (TokenClass . NOT); }

Fig. 2.7 Clite Grammar: Statements

Program → int main ( ) { Declarations Statements }

Declarations → { Declaration }

Declaration → Type Identifier [[ Integer ]] { , Identifier [[ Integer ]] }

Type → int | bool | float | char

Statements → { Statement }

Statement → ; | Block | Assignment | IfStatement | WhileStatement

Block → { Statements }

Assignment → Identifier [[ Expression ]] = Expression ;

IfStatement → if ( Expression ) Statement [ else Statement ]

WhileStatement → while ( Expression ) Statement
JFlex Example

bool { echo (); return new Token (TokenClass.BOOL); }
char { echo (); return new Token (TokenClass.CHAR); }
else { echo (); return new Token (TokenClass.ELSE); }
float { echo (); return new Token (TokenClass.FLOAT); }
if { echo (); return new Token (TokenClass.IF); }
int { echo (); return new Token (TokenClass.INT); }
main { echo (); return new Token (TokenClass.MAIN); }
while { echo (); return new Token (TokenClass.WHILE); }
{Integer} { echo (); return new Token (TokenClass.INTEGER, yytext()); }
{Boolean} { echo (); return new Token (TokenClass.BOOLEAN, yytext()); }
{Float} { echo (); return new Token (TokenClass.FLOATLIT, yytext()); }
{Char} { echo (); return new Token (TokenClass.CHARLIT, yytext().substring(1, 2)); }
{Identifier} { echo (); return new Token (TokenClass.ID, yytext()); }

JFlex Example

... { echo ();
    ErrorMessage.print
    (yycolumn, "Illegal character"); }
JFlex Example

faculty% jflex Clite.jflex
Reading "Clite.jflex"
Constructing NFA : 179 states in NFA
Converting NFA to DFA :
...........................................................
...........................
80 states before minimization, 72 states in minimized DFA
Writing code to "CliteLexer.java"
faculty% javac CliteLexer.java

JFlex Example

faculty% cat TokenClass.java
public enum TokenClass {
    EOF,
    // keywords
    BOOL, CHAR, ELSE, FLOAT, IF, INT, MAIN, WHILE,
    // punctuation
    COMMA, SEMICOLON, LBRACE, RBRACE,
    // operators
    LPAREN, RPAREN, LBRACK, RBRACK, ASSIGN, OR, AND,
    EQ, NE, LT, LE, GT, GE, PLUS, MINUS, TIMES,
    SLASH, MOD, NOT,
    // ids and literals
    ID, INTEGER, BOOLEAN, FLOATLIT, CHARLIT
}
JFlex Example

faculty% cat Token.java

public class Token {

    private int symbol; // current token
    private String lexeme; // lexeme

    public Token () { }
    public Token (int symbol) { this (symbol, null); }
    public Token (int symbol, String lexeme) {
        this . symbol = symbol;
        this . lexeme = lexeme;
    }

    public int symbol () { return symbol; }
    public String lexeme () { return lexeme; }

    public String toString () {
        switch (symbol) {
            case BOOL :      return "(keyword, bool) ";
            case CHAR :      return "(keyword, char) ";
            case ELSE :      return "(keyword, else) ";
            case FLOAT :     return "(keyword, float) ";
            case IF :        return "(keyword, if) ";
            case INT :       return "(keyword, int) ";
            case MAIN :      return "(keyword, main) ";
            case WHILE :     return "(keyword, while) ";
            case COMMA :     return "(punctuation, ,) ";
            case SEMICOLON : return "(punctuation, ;) ";
            case LBRACE : return "(punctuation, {) ";
            case RBRACE : return "(punctuation, }) ";
        }
    }

"}
JFlex Example

case LPAREN :    return "(operator, () ");
case RPAREN :    return "(operator, )) ";
case LBRACK :    return "(operator, [) ";
case RBRACK :    return "(operator, ]) ";
case ASSIGN :    return "(operator, =) ";
case OR :        return "(operator, ||) ";
case AND :       return "(operator, &&) ";
case PLUS :      return "(operator, +) ";
case MINUS :     return "(operator, -) ";
case TIMES :     return "(operator, *) ";
case SLASH :     return "(operator, /) ";
case MOD :       return "(operator, %) ";
case EQ :        return "(operator, ==) ";
case NE :        return "(operator, !=) ";
case LT :        return "(operator, <) ";
case LE :        return "(operator, <=) ";
case GT :        return "(operator, >) ";
case GE :        return "(operator, >=) ";
case NOT :       return "(operator, !) ";
    
    case ID :    return "(identifier, " + lexeme + ") ";
case INTEGER : return "(integer, " + lexeme + ") ";
case BOOLEAN : return "(boolean, " + lexeme + ") ";
case FLOATLIT : return "(float, " + lexeme + ") ";
case CHARLIT : return "(char, " + lexeme + ") ";
default :
    ErrorMessage . print (0, "Unrecognized token");
    return null;
JFlex Example

faculty% cat ErrorMessage.java
// ErrorMessage class
// This class prints error messages.

class ErrorMessage {
    public static void print (int position, String message) {
        int i;
        System.out.println();
        for (i = 0; i < position; i++)
            System.out.print(" ");
        System.out.println("^");
        System.out.println("***** Error: " + message + " *****");
        System.exit(0);
    }
}

JFlex Example

faculty% cat CliteLex.java
import java.io.*;
public class CliteLex {
    private static final int MAX_TOKENS = 100;

    public static void main (String [] args) throws java.io.IOException {
        int i, n;
        Token [] token = new Token [MAX_TOKENS];
        CliteLexer lexer = new CliteLexer (System.in);

        System.out.println("Source Program");
        System.out.println("-------------");
        System.out.println();
    }
}
JFlex Example

n = -1;
do {
    if (n < MAX_TOKENS)
        token [++n] = lexer . next_token ();
    else
        ErrorMessage .
            print (0,
                "Maximum number of tokens exceeded");
} while (token [n] . symbol () != Symbol . EOF);

System . out . println ();
System . out . println ("List of Tokens");
System . out . println ("----------");
System . out . println ();
for (i = 0; i < n; i++)
    token [i] . print ();
System . out . println ();

JFlex Example

faculty% javac CliteLex.java
faculty% java CliteLex < quotrem.c
Source Program
----------
// Compute quotient and remainder of 32/5.
int main () {
    int q, r, x, y;
    x = 32; y = 5;
    q = 0; r = x;
    while (r >= y) {
        q = q + 1;
        r = r - y;
    }
}
JFlex Example

List of Tokens
-------------
(keyword, int)
(keyword, main)
(operator, ()
(operator, ))
(punctuation, )
(keyword, int)
(identifier, q)
(punctuation, ,)
(identifier, r)
(punctuation, ,)
(identifier, x)
(punctuation, ,)
(identifier, y)
(punctuation, ;)

JFlex Example

(identifier, x)
(operator, =)
(integer, 32)
(punctuation, ;)
(identifier, y)
(operator, =)
(integer, 5)
(punctuation, ;)
(identifier, q)
(operator, =)
(integer, 0)
(punctuation, ;)
(identifier, r)
(operator, =)
(identifier, x)
(punctuation, ;)
JFlex Example

(keyword, while)
(operator, ()
(identifier, r)
(operator, >=)
(identifier, y)
(operator, ))
(punctuation, {})
(identifier, q)
(operator, =)
(identifier, q)
(operator, +)
(integer, 1)
(punctuation, ;)

JFlex Example

(identifier, r)
(operator, =)
(identifier, r)
(operator, -)
(identifier, y)
(punctuation, ;)
(punctuation, {})
(punctuation, {})

Copyright © 2014 Barrett R. Bryant.
Syntactic Analysis

Phase also known as: parser
Purpose is to recognize source structure
Input: tokens
Output: parse tree or abstract syntax tree
A recursive descent parser is one in which each
nonterminal in the grammar is converted to a
function which recognizes input derivable from
the nonterminal.

Recursive Descent Parsing

Method/function for each nonterminal
Recognize longest sequence of tokens derivable from
the nonterminal
Need an algorithm for converting productions to code
Based on EBNF
Recursive-Descent Parsing (cont.)

Assume we have a lexical analyzer named \texttt{GET\_TOKEN}, which puts the next token code in \texttt{TOKEN}.

The coding process when there is only one RHS:

- For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error.
- For each nonterminal symbol in the RHS, call its associated parsing subprogram.

Convention: Every parsing routine leaves the next token in \texttt{TOKEN}.

Recursive-Descent Parsing (cont.)

A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse:

- The correct RHS is chosen on the basis of the next token of input (the lookahead).
- The next token is compared with the first token that can be generated by each RHS until a match is found.
- If no match is found, it is a syntax error.
Augmentation Production

Gets first token
Calls method corresponding to original start symbol
Checks to see if final token is end token

*E.g.: end of file token*

```java
private void match (int t) {
    if (token.type() == t)
        token = lexer.next();
    else
        error(t);
}
```
private void error(int tok) {
    System.err.println(
        "Syntax error: expecting"
        + tok + "; saw: " + token);
    System.exit(1);
}

private void assignment() {
    // Assignment → Identifier = Expression ;
    match(Token.Identifier);
    match(Token.Assign);
    expression();
    match(Token.Semicolon);
}
private void expression( ) {
    // Expression → Term { AddOp Term }
    term( );
    while (isAddOp()) {
        token = lexer.next( );
        term( );
    }
}
Discard from Parse Tree

Separator/punctuation terminal symbols
All trivial root nonterminals
Replace remaining nonterminal with leaf terminal

Abstract Syntax Example

Assignment = Variable target; Expression source
Expression = Variable | Value | Binary | Unary
Binary = Operator op; Expression term1, term2
Unary = Operator op; Expression term
Variable = String id
Value = Integer value
Operator = + | - | * | / | !
abstract class Expression { }

class Binary extends Expression {
    Operator op;
    Expression term1, term2;
}

class Unary extends Expression {
    Operator op; Expression term;
}

Modify T(A \rightarrow w) to Return AST

1. Make A the return type of function, as defined by abstract syntax.
2. If w is a nonterminal, assign returned value.
3. If w is a non-punctuation terminal, capture its value.
4. Add a return statement that constructs the appropriate object.
private Assignment assignment() {
    // Assignment → Identifier = Expression ;
    Variable target = match(Token.Identifier);
    match(Token.Assign);
    Expression source = expression();
    match(Token.Semicolon);
    return new Assignment(target, source);
}

private String match(int t) {
    String value = Token.value();
    if (token.type() == t)
        token = lexer.next();
    else
        error(t);
    return value;
}