Building lexical and syntactic analyzers

Chapter 3

Syntactic sugar causes cancer of the semicolon.
A. Perlis

Chomsky Hierarchy

- Four classes of grammars, from simplest to most complex:
  - Regular grammar
    - What we can express with a regular expression
  - Context-free grammar
    - Equivalent to our grammar rules in BNF
  - Context-sensitive grammar
  - Unrestricted grammar
- Only the first two are used in programming languages

Lexical Analysis

- Purpose: transform program representation
- Input: printable ASCII (or Unicode) characters
- Output: tokens (type, value)
- Discard: whitespace, comments

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

Sample Tokens

- Identifiers
- Literals: 123, 5.67, 'x', true
- Keywords: bool char ...
- Operators: + - * / ...
- Punctuation: ; , ( ) { }
- Whitespace: space tab
- Comments
  - // any-char* end-of-line
- End-of-line
- End-of-file

Lexical Phase

- Why a separate phase for lexical analysis? Why not make it part of the concrete syntax?
  - 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
    - Macs: cr (ASCII 13)
    - Windows: cr/lf (ASCII 13/10)
    - Unix: nl (ASCII 10)

Categories of Lexical Tokens

- Identifiers
- Literals

  Includes Integers, true, false, floats, chars
- Keywords

  bool char else false float if int main true while
- Operators

  = || && == != < <= > >= + - * / % ! [ ]
- Punctuation

  ; . { } ( )

Regular Expression Review

RegExpr	**Meaning**
x	a character x
\x	an escaped character, e.g., \n
{ name }	a reference to a name
M | N	M or N
M N	M followed by N
M*	zero or more occurrences of M
M+	One or more occurrences of M
M?	Zero or one occurrence of M
[aeiou]	the set of vowels
[0-9]	the set of digits
.	Any single character

Clite Lexical Syntax

Category	**Definition**
anyChar	[-~]
Letter	[a-zA-Z]
Digit	[0-9]
Whitespace	[\t]
Eol	\n
Eof	\004

Category **Definition**

Keyword bool | char | else | false | float |
if | int | main | true | while

Identifier {Letter}({Letter} | {Digit})*

integerLit {Digit}+

floatLit {Digit}+\.{Digit}+

charLit ‘{anyChar}’

Category **Definition**

Operator = | || | && | == | != | < | <= | > |
>= | + | − | * | / |! | [|]

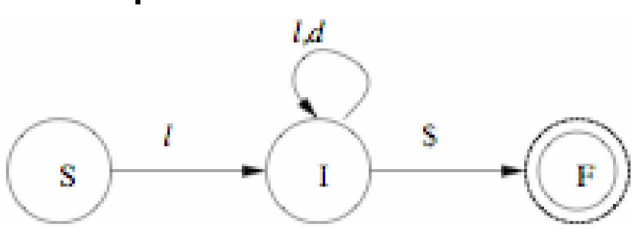
Separator : | . | { | } | (|)

Comment // ({anyChar} |{Whitespace})*{eol}

Finite State Automaton

- Given the regular expression definition of lexical tokens, how do we design a program to recognize these sequences?
- One way: build a deterministic finite automaton
  - Set of states: representation – graph nodes
  - Input alphabet + unique end symbol
  - State transition function
  - Labelled (using alphabet) arcs in graph
  - Unique start state
  - One or more final states

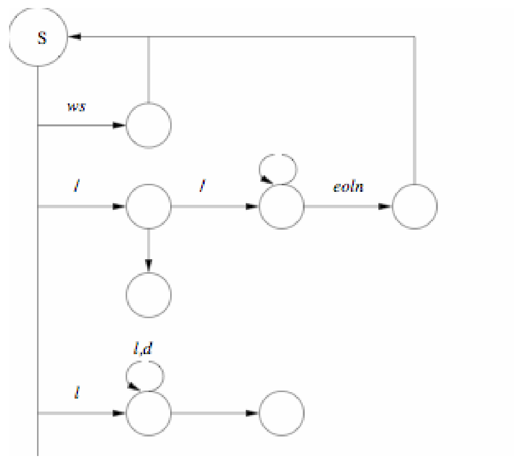
Example : DFA for Identifiers

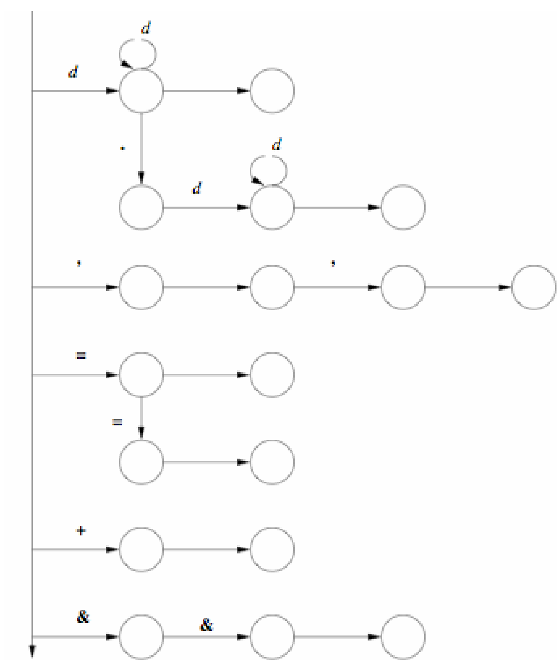


An input is *accepted* if, starting with the start state, the automaton consumes all the input and halts in a final state.

An input is accepted if, starting with the start state, the automaton consumes all the input and halts in a final state.

Overview of DFA's for Clite





Lexer Code

- Parser calls lexer whenever it needs a new token.
- Lexer must remember where it left off.
  - Class variable for the current char (ch)
- Greedy consumption goes 1 character too far
  - Consider: **(foo<bar)** with no whitespace after the foo. If we consume the < at the end of identifying foo, we lose the first char of the next token
    - peek function
    - pushback function
    - no symbol consumed by start state

From Design to Code

```
private char ch = ` `;
public Token next ( )
{
     do {
          switch (ch)
          {
               ...
          }
     } while (true);
}
```

- **Loop only exited when a token is found**
- **Loop exited via a return statement.**
- **Variable ch must be global. Initialized to a space character.**

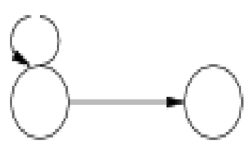
Translation Rules

- We need to translate our DFA into code
  - Relatively straightforward process

  - Traversing an arc from A to B:
    - If labeled with x: test ch == x
    - If unlabeled: else/default part of if/switch. If only arc, no test need be performed.
    - Get next character if A is not start state

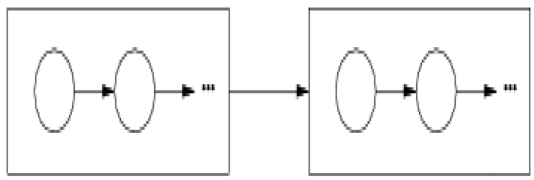
Translation Rules

- A node with an arc to itself is a do-while.



- Otherwise the move is translated to a if/switch:
  - Each arc is a separate case.
  - Unlabeled arc is default case.
- A sequence of transitions becomes a sequence of translated statements.

- A complex diagram is translated by boxing its components so that each box is one node.
  - Translate each box using an outside-in strategy.



Some Code – Helper Functions

```
private boolean isLetter(char c) {
  return ch >= 'a' && ch <= 'z' ||
            ch >= 'A' && ch <= 'Z';
}

private String concat(String set) {
  StringBuffer r = new StringBuffer("");
  do {
            r.append(ch);
            ch = nextChar( );
  } while (set.indexOf(ch) >= 0);
  return r.toString( );
}
```

Code

- See next() method in the Lexer.java source code
- Code is in the zip file for homework #1

Lexical Analysis of Clite in Java

```
public class TokenTester {
     public static void main (String[] args) {
      Lexer lex = new Lexer (args[0]);
      Token t;
      int i = 1;

      do
      {
        t = lex.next();
        System.out.println(i+" Type: "+t.type()
                    +"\tValue: "+t.value());
        i++;
      } while (t != Token.eofTok);
     }
  }
```

Result of Analysis (seen before)

Result of Lexical Analysis:

```
1 Type: Int   Value: int
2 Type: Main  Value: main
3 Type: LeftParen   Value: (
4 Type: RightParen  Value: )
5 Type: LeftBrace   Value: {
6 Type: Int   Value: int
7 Type: Identifier  Value: x
8 Type: Semicolon   Value: ;
9 Type: Identifier  Value: x
10 Type: Assign     Value: =
11 Type: IntLiteral Value: 3
12 Type: Semicolon  Value: ;
13 Type: RightBrace Value: }
14 Type: Eof  Value: <<EOF>>
```

```
// Simple Program
int main() {
   int x;
   x = 3;
}
```

Syntactic Analysis

- After the lexical tokens have been generated the next phase is syntactic analysis, i.e. parsing
- 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.

Parsing Preliminaries

- Skipping, some more detail in the book
- To prep the grammar for easier parsing it is converted into a left dependency grammar:
  - Discover all terminals recursively
  - Turn regular expressions into BNF style grammar
  - For example:

    A → x { y } z   becomes

    A → x A' z
    A' → ε | yA'

Program Structure Consists Of:

- Expressions: x + 2 * y
- Assignment Statement: z = x + 2 * y
- Loop Statements:
  while (i < n) a[i++] = 0;
- Function definitions
- Declarations: int i;

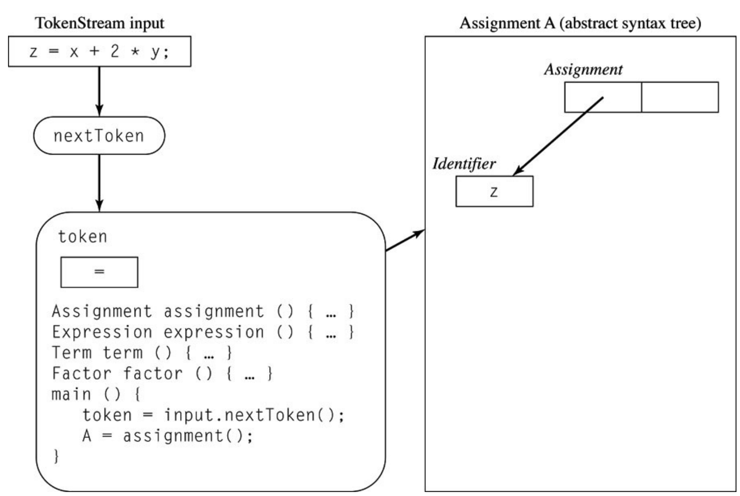
- Assignment → Identifier = Expression
- Expression → Term { AddOp Term }
- AddOp → + | -
- Term → Factor { MulOp Factor }
- MulOp → * | /
- Factor → [UnaryOp] Primary
- UnaryOp → - | !
- Primary → Identifier | Literal | (Expression)

Partial here; skipping &&, ||, etc.

Recursive Descent Parser

- One algorithm for generating an abstract syntax tree
  - Input: lexical, concrete, outputs abstract representation
    - Lexical data a stream of tokens, comes from the Lexer we saw earlier
  - This algorithm is top down
  - Based on an EBNF concrete syntax

Overview of Recursive Descent Process for Assignment



Algorithm for Writing a Recursive Descent Parser from EBNF

For each nonterminal symbol A and set of rules of the form $A \to \omega$:

1. Add a new method definition with A as its return type.
2. Create a new object of class A, say x.
3. For each member y of the sentential form ω,
   a. if y is a nonterminal, call the method associated with y and assign the result to an appropriate field within x.
   b. if y is a terminal, check that the value of that token is identical with y and, if so, call the `nextToken` method. Otherwise the token is in error.
4. If ω contains a series of symbols that is repeated (indicated by *), insert an appropriate while loop that accommodates any number of repetitions of that series.
5. If there is more than one rule of the form $A \to \omega$, insert appropriate `if ... else` statements that distinguishes the alternatives.
6. Return x.

Implementing Recursive Descent

- Say we want to write Java code to parse Assignment (EBNF, Concrete Syntax):
  - Assignment → Identifier = Expression;
  - From steps 1-2, we add a method for an Assignment object:

```
private Assignment assignment () {
    …  // will fill in code here momentarily to parse assignment
    return new Assignment(target, source);
}
```

This is a method named assignment in the Parser.java file… separate from the Assignment class defined in AbstractSyntax.java

Implement Assignment

- According to the syntax, assignment should find an identifier, an operator (=), an expression, and a separator (;)
  - So these are coded up into the method!

```
private Assignment assignment () {
     // Assignment --> Identifier = Expression ;
        Variable target = new Variable
                            (match(Token.Identifier));
        match(Token.Assign);
        Expression source = expression();
        match(Token.Semicolon);
        return new Assignment(target, source);
     }
```

Helper Methods

- Match: retrieves next token or displays a syntax error.
- Syntax Error: Displays error and terminates

```
 private void match (TokenType t) {
      String value = token.value();
      if (token.type().equals(t))
            token = lexer.next();
      else
            error(t);
      return value;
}

private void error(TokenType tok) {
  System.err.println("Syntax error: expecting: " + tok
  + "; saw: " + token);
  System.exit(1);
}
```

Expression Method

- Assignment method relies on Expression method
  - Expression → Conjunction { || Conjunction }*

```
private Expression expression () {
       // Conjunction --> Equality { && Equality }
       Expression e = equality();
       while (token.type().equals(TokenType.And)) {
           Operator op = new Operator(token.value());
           token = lexer.next();
           Expression term2 = equality();
           e = new Binary(op, e, term2);
       }
       return e;
     }
```

Need loop for possible multiple &&'s.
Conjunction method must return expr if there are no &&'s

More Expression Methods

```
private Expression factor() {
       // Factor --> [ UnaryOp ] Primary
       if (isUnaryOp()) {
           Operator op = new Operator(match(token.type()));
           Expression term = primary();
           return new Unary(op, term);
       }
       else return primary();
   }
```

More Expression Methods

```
private Expression primary () {
        // Primary --> Identifier | Literal | ( Expression )
        //             | Type ( Expression )
        Expression e = null;
        if (token.type().equals(TokenType.Identifier)) {
            Variable v = new Variable(match(TokenType.Identifier));
            e = v;
        } else if (isLiteral()) {
            e = literal();
        } else if (token.type().equals(TokenType.LeftParen)) {
            token = lexer.next();
            e = expression();
            match(TokenType.RightParen);
        } else if (isType( )) {
            Operator op = new Operator(match(token.type()));
            match(TokenType.LeftParen);
            Expression term = expression();
            match(TokenType.RightParen);
            e = new Unary(op, term);
        } else error("Identifier | Literal | ( | Type");
        return e;
    }
```

Finished Program

- Finishing recursive descent parser will be available as Parser.java
  - Extending it in some way will be left as an exercise ☺
- What we've done in the resulting program incorporates both the concrete and abstract syntax
  - Concrete syntax used to define the methods, classes, sequence of tokens
  - Abstract syntax is created by setting the class member variables to the appropriate data values as the program is parsed