Lexical and Syntax Analysis

Part I

Introduction

- Every implementation of Programming Languages (i.e. a compiler) uses a Lexical Analyzer and a Syntax Analyzer in its initial stages.
- The Lexical Analyzer tokenizes the input program
- The syntax analyzer, referred to as a parser, checks for syntax of the input program and generates a parse tree.
- Parsers almost always rely on a CFG that specifies the syntax of the programs.
- In this section, we study the **inner workings** of Lexical Analyzers and Parsers
- The algorithms that go into building lexical analyzers and parsers rely on **automata and formal language theory** that forms the foundations for these systems.

Lexemes and Tokens

- A lexical analyzer collects characters into groups (**lexemes**) and assigns an internal code (a **token**) to each group.
- Lexemes are recognized by matching the input against **patterns**.
- Tokens are usually coded as integer values, but for the sake of readability, they are often referenced through named constants.

  Example assignment statement (tokens/lexemes shown to the right):

  ```
  result = oldsum - value / 100;
  ```

Token	Lexeme
IDENT	result
ASSIGN	=
IDENT	olds
SUB	-
IDENT	value
DIV	/
INT_LIT	100
SEMI	;

- In earlier compilers, entire input used to be read by Lexical analyzer and a file of tokens/lexemes produced. Modern day lexers provide the **next token** when requested.
- Other tasks performed by Lexers: skip comments and white space; Detect syntactic errors in tokens

Lexical Analysis (continued)

Approaches to building a lexical analyzer:

- Write a formal description of the token patterns of the language and use a software tool such as PLY to automatically generate a lexical analyzer. We have seen this earlier!
- Design a **state transition diagram** that describes the token patterns of the language and **write a program** that implements the diagram. We will develop this in this section.
- Design a state transition diagram that describes the token patterns of the language and **hand-construct a table-driven implementation** of the state diagram.

A state transition diagram, or state diagram, is a **directed graph**. The **nodes** are labeled with state names. The **edges** are labeled with input characters. An edge may also include **actions** to be done when the transition is taken.

Lexical Analyzer: An implementation

- Consider the problem of building a Lexical Analyzer that recognizes lexemes that appear in **arithmetic expressions**, including variable names and integers.
- Names consist of uppercase letters, lowercase letters, and digits, but must begin with a letter. Names have no length limitations.
- To simplify the state transition diagram, we will treat all letters the same way; so instead of 52 transitions or edges, we will have just one edge labeled **Letter**. Similarly for digits, we will use the label **Digit**.
- The following "actions" will be useful to visualize when thinking about the Lexical Analyzer:

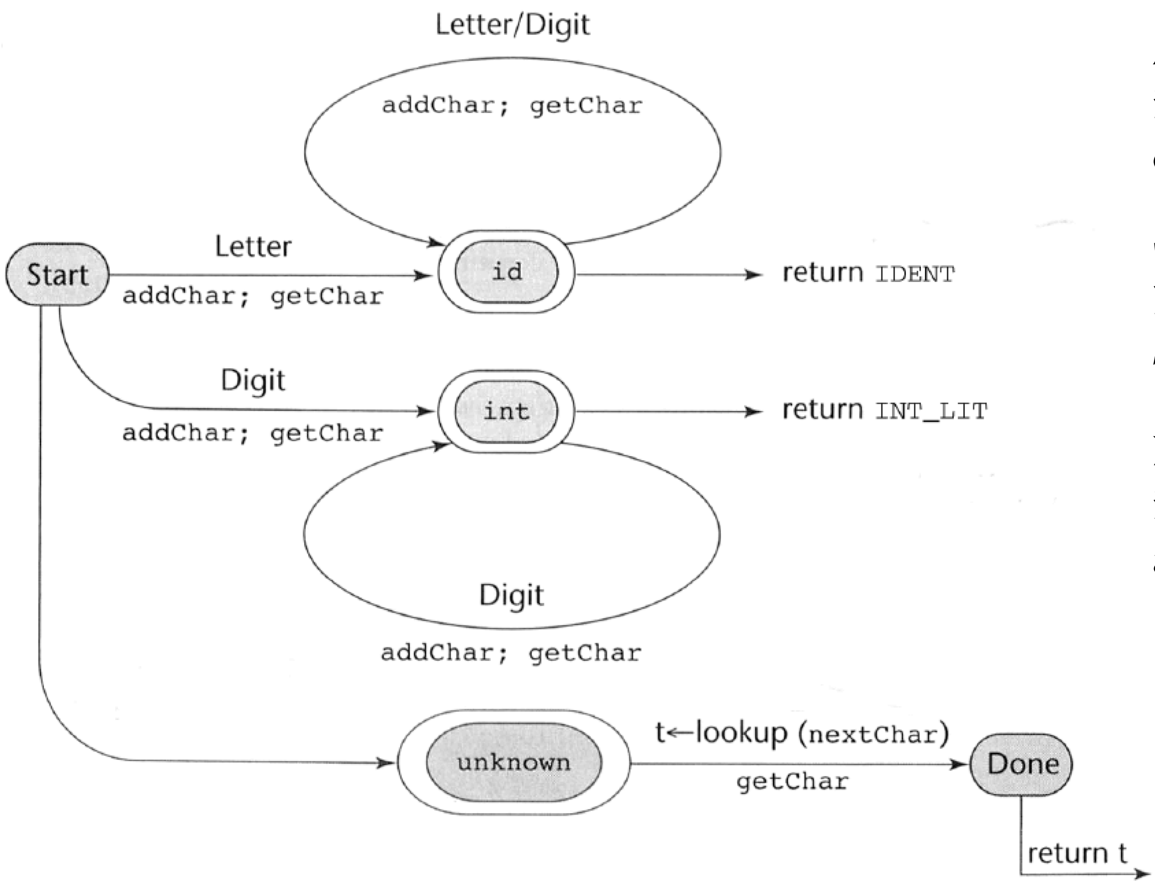
  `getChar`: read the next character from the input

  `addChar`: add the character to the end of the lexeme being recognized

  `getNonBlank`: skip white space

  `lookup`: find the token for single character lexemes

Lexical Analyzer: An implementation (continued)



A state diagram that recognizes names, integer literals, parentheses, and arithmetic operators.

Shows how to recognize **one** lexeme; The process will be repeated until EOF.

The diagram includes actions on each edge.

Next, we will look at a Python program that implements this state diagram to tokenize arithmetic expressions.

Lexical Analyzer: An implementation (in Python; TokenTypes.py)

```
TokenTypes.py

import enum

class TokenTypes(enum.Enum):
  LPAREN = 1
  RPAREN = 2
  ADD = 3
  SUB = 4
  MUL = 5
  DIV = 6
  ID = 7
  INT = 8
  EOF = 0
```

Lexical Analyzer: An implementation (Token.py)

Token.py

```
class Token:

  def __init__(self,tok,value):
    self._t = tok
    self._c = value

  def __str__(self):
    if self._t.value == TokenTypes.ID.value:
      return "<" + self._t + ":"+ self._c + ">"
    elif self._t.value == TokenTypes.INT.value:
      return "<" + self._c + ">"
    else:
      return self._t

  def get_token(self):
    return self._t

  def get_value(self):
    return self._c
```

Lexical Analyzer: An implementation (Lexer.py)

```
import sys
from TokenTypes import *
from Token import *

# Lexical analyzer for arithmetic expressions which
# include variable names and positive integer literals
# e.g. (sum + 47) / total

class Lexer:

  def __init__(self,s):
    self._index = 0
    self._tokens = self.tokenize(s)

  def tokenize(self,s):
    result = []
    i = 0
    while i < len(s):
      c = s[i]
      if c == '(':
        result.append(Token(TokenTypes.LPAREN, "("))
        i = i + 1
      elif c ==  ')':
        result.append(Token(TokenTypes.RPAREN, ")"))
        i = i + 1
      elif c ==  '+':
        result.append(Token(TokenTypes.ADD, "+"))
        i = i + 1
      elif c ==  '-':
        result.append(Token(TokenTypes.SUB, "-"))
        i = i + 1
      elif c ==  '*':
        result.append(Token(TokenTypes.MUL, "*"))
        i = i + 1
      elif c ==  '/':
        result.append(Token(TokenTypes.DIV, "/"))
        i = i + 1
      elif c in ' \r\n\t':
        i = i + 1
        continue
      elif c.isdigit():
        j = i
        while j < len(s) and s[j].isdigit():
          j = j + 1
        result.append(Token(TokenTypes.INT,s[i:j]))
        i = j
```

Lexical Analyzer: An implementation (Lexer.py)

```
    elif c.isalpha():
      j = i
      while j < len(s) and s[j].isalnum():
        j = j + 1
      result.append(Token(TokenTypes.ID,s[i:j]))
      i = j
    else:
      print("UNEXPECTED CHARACTER ENCOUNTERED: "+c)
      sys.exit(-1)
  result.append(Token(TokenTypes.EOF, “-1"))
  return result

def lex(self):
  t = None
  if self._index < len(self._tokens):
    t = self._tokens[self._index]
    self._index = self._index + 1
  print("Next Token is: "+str(t.get_token())+", Next lexeme is "+t.get_value())
  return t
```

Lexical Analyzer: An implementation (LexerTest.py)

LexerTest.py

```
from Lexer import *
from TokenTypes import *

def main():
  input = "(sum + 47) / total"
  lexer = Lexer(input)
  print("Tokenizing ",end="")
  print(input)
  while True:
    t = lexer.lex()
    if t.get_token().value == TokenTypes.EOF.value:
      break

main()
```

Go to live demo.

Lexical Analyzer: An implementation (Sample Run)

```
macbook-pro:handCodedLexerRecursiveDescentParser raj$ python3 LexerTest.py
Tokenizing (sum + 47) / total
Next Token is: TokenTypes.LPAREN, Next lexeme is (
Next Token is: TokenTypes.ID, Next lexeme is sum
Next Token is: TokenTypes.ADD, Next lexeme is +
Next Token is: TokenTypes.INT, Next lexeme is 47
Next Token is: TokenTypes.RPAREN, Next lexeme is )
Next Token is: TokenTypes.DIV, Next lexeme is /
Next Token is: TokenTypes.ID, Next lexeme is total
Next Token is: TokenTypes.EOF, Next lexeme is -1
```

Introduction to Parsing

- Syntax analysis is often referred to as parsing.
- A parser checks to see if the input program is syntactically correct and constructs a parse tree.
- When an error is found, a parser must produce a diagnostic message and recover. Recovery is required so that the compiler finds as many errors as possible.
- Parsers are categorized according to the direction in which they build the parse tree:
  - **Top-down** parsers build the parse tree from the root downwards to the leaves.
  - **Bottom-up** parsers build the parse tree from the leaves upwards to the root.

Notational Conventions

Terminal symbols — Lowercase letters at the beginning of the alphabet (a, b, …)

Nonterminal symbols — Uppercase letters at the beginning of the alphabet (A, B, …)

Terminals or nonterminals — Uppercase letters at the end of the alphabet (W, X, Y, Z)

Strings of terminals — Lowercase letters at the end of the alphabet (w, x, y, z)

Mixed strings (terminals and/or nonterminals) — Lowercase Greek letters (**α, β, γ, δ**)

Top-Down Parser

- A top-down parser traces or builds the parse tree in **preorder**: each node is visited before its branches are followed.
- The actions taken by a top-down parser correspond to a **leftmost derivation.**
- Given a sentential form **xA**α that is part of a leftmost derivation, a top-down parser's task is to find the next sentential form in that leftmost derivation.
  - Determining the next sentential form is a matter of choosing the correct grammar rule that has **A** as its left-hand side (LHS).
  - If the A-rules are A → bB, A → cBb, and A → a, the next sentential form could be xbBα, xcBbα, or xaα.
  - The most commonly used top-down parsing algorithms **choose** an A-rule based on the **token** that would be the **first generated by** A.

Top-Down Parser (continued)

- The most common top-down parsing algorithms are closely related:
  - A **recursive-descent parser** is coded directly from the CFG description of the syntax of a language.
  - An alternative is to use a **parsing table** rather than code.
- Both are **LL algorithms**, and both are equally powerful. The first L in LL specifies a **left-to-right scan of the input**; the second L specifies that a **leftmost derivation** is generated.
- We will look at a hand-written recursive-descent parser later in this section (in Python).

Bottom-Up Parser

- A bottom-up parser constructs a parse tree by **beginning at the leaves** and progressing **toward the root**. This parse order corresponds to the **reverse** of a **rightmost derivation**.
- Given a right sentential form α, a bottom-up parser must determine what substring of α is the right-hand side (RHS) of the rule that must be **reduced** to its LHS to produce the **previous** right sentential form.
- A given right sentential form may include more than one RHS from the grammar. The correct RHS to reduce is called the **handle**. As an example, consider the following grammar and derivation (shown twice):

```
S : aAc
A : aA
A : b
```

```
S => aAc => aaAc => aabc

S => aAc => aaAc => aabc
```

- A bottom-up parser can easily find the first handle, b, since it is the only RHS of a rule. After replacing b by the corresponding LHS, we get aaAc, the previous right sentential form. Finding the next handle is more difficult because both aAc and aA are potential handles.

Bottom-Up Parser (continued)

- A bottom-up parser finds the handle of a given right sentential form by **examining** the symbols on **one or both sides** of a possible handle.
- The most common bottom-up parsing algorithms are in the LR family. The L specifies a left-to-right scan and the R specifies that a rightmost derivation is generated.
- Time Complexity
  - Parsing algorithms that work for any grammar are inefficient. The worst-case complexity of common parsing algorithms is $\mathbf{O(n^3)}$, making them impractical for use in compilers.
  - Faster algorithms work for only a subset of all possible grammars. These algorithms are acceptable as long as they can parse grammars that describe programming languages.
  - Parsing algorithms used in commercial compilers have complexity $\mathbf{O(n)}$.

Recursive-Descent Parsing

- A recursive-descent parser consists of a collection of functions, many of which are recursive; it produces a parse tree in top-down order.
- A recursive-descent parser has one function for each nonterminal in the grammar.
- Consider the expression grammar below (written in EBNF - extended BNF notation):

```
<expr> : <term> {(+|-) <term>}
<term> : <factor> {(*|/) <factor>}
<factor> : ID | INT_CONSTANT |( <expr> )
```

These rules can be used to construct a recursive-descent function named `expr` that parses arithmetic expressions.

The lexical analyzer is assumed to be a function named `lex`. It reads a lexeme and puts its token code in the global variable `next_token`. Token codes are defined as named constants.

Recursive-Descent Parsing (continued)

- Writing the recursive-descent functions are quite simple
- We assume two global variables: lexer_object and next_token; Initially the first token is retrieved into next_token and then the function for the start symbol is called:

```
import sys
from Lexer import *

next_token = None
l = None

def main():
  global next_token
  global l
  l = Lexer(sys.argv[1])
  next_token = l.lex()
  expr()
  if next_token.get_token().value == TokenTypes.EOF.value:
    print(“PARSE SUCCEEDED”)
  else:
    print(“PARSE FAILED”)
```

Recursive-Descent Parsing (continued)

- The function for `<expr>` is shown below. For each terminal symbol on the RHS of the rule, the current value of `next_token` is matched to that terminal and for each non-terminal the corresponding function is called. When the function exits, it is made sure that `next_token` contains the value of the next token beyond what matches `<expr>`

```
# expr
# Parses strings in the language generated by the rule:
# <expr> : <term> {(+|-) <term>}
def expr():
  global next_token
  global l
  print("Enter <expr>")
  term()
  while next_token.get_token().value == TokenTypes.ADD.value or \
        next_token.get_token().value == TokenTypes.SUB.value:
    next_token = l.lex()
    term()
  print("Exit <expr>")
```

Recursive-Descent Parsing (continued)

The function for `<term>` is similar to the function for <expr>

```
# term
# Parses strings in the language generated by the rule:
# <term> : <factor> {(*|/) <factor>}
def term():
  global next_token
  global l
  print("Enter <term>")
  factor()
  while next_token.get_token().value == TokenTypes.MUL.value or \
        next_token.get_token().value == TokenTypes.DIV.value:
    next_token = l.lex()
    factor()
  print("Exit <term>")
```

Recursive-Descent Parsing (continued)

```
# factor
# Parses strings in the language generated by the rules:
#   <factor> -> ID
#   <factor> -> INT_CONSTANT
#   <factor> -> ( <expr> )
def factor():
  global next_token
  global l
  print("Enter <factor>")
  if next_token.get_token().value == TokenTypes.ID.value or \
     next_token.get_token().value == TokenTypes.INT.value:
    next_token = l.lex()
  else: # if the RHS is ( <expr> ), pass over (, call expr, check for )
    if next_token.get_token().value == TokenTypes.LPAREN.value:
      next_token = l.lex()
      expr()
      if next_token.get_token().value == TokenTypes.RPAREN.value:
        next_token = l.lex()
      else:
        error("Expecting RPAREN")
        sys.exit(-1)
    else:
      error("Expecting LPAREN")
      sys.exit(-1)
  print("Exit <factor>")
```

```
def error(s):
  print("SYNTAX ERROR: "+s)
```

The function for `<factor>` checks to see if the next_token matches `ID` or `INT_CONSTANT`; if matched, the function exits.

otherwise it matches a left parenthesis, then calls the function for <expr> and then matches the right parenthesis. This function also makes sure next_token contains the next token beyond the match for <factor>

Show Demo

Recursive-Descent Parsing

Sample run:

```
$ python3 Parser.py "(sum + 20)/30"
Next Token is: TokenTypes.LPAREN, Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next Token is: TokenTypes.ID, Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next Token is: TokenTypes.ADD, Next lexeme is +
Exit <factor>
Exit <term>
Next Token is: TokenTypes.INT, Next lexeme is 20
Enter <term>
Enter <factor>
Next Token is: TokenTypes.RPAREN, Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next Token is: TokenTypes.DIV, Next lexeme is /
Exit <factor>
Next Token is: TokenTypes.INT, Next lexeme is 30
Enter <factor>
Next Token is: TokenTypes.EOF, Next lexeme is -1
Exit <factor>
Exit <term>
Exit <expr>
PARSE SUCCEEDED
```

Recursive-Descent Parsing: if-then-else stmt

```
<ifstmt> → if ( <boolexpr> ) <statement> [else <statement>]
```

```
def ifstmt():
  global next_token
  global l
  if next_token.get_token().value != TokenTypes.IF.value:
    error("Expecting IF")
  else:
    next_token = l.lex()
    if next_token.get_token().value != TokenTypes.LPAREN.value:
      error("Expecting LPAREN")
    else:
      next_token = l.lex()
      boolexpr()
      if next_token.get_token().value != TokenTypes.RPAREN.value:
        error("Expecting RPAREN")
      else:
        next_token = l.lex()
        statement()
        if next_token.get_token().value == TokenTypes.ELSE.value:
          next_token = l.lex()
          statement()
```