1 Introduction

We often think of compilers in terms of their capability to translate source code in one language into target code of anything. Typically the context for this to move from a higher-level language into assembly language, which is then translated into machine code. Yet when compilers are taught in the classroom setting, the implementation of their components is generally taught such that source and target language are the same. For the student, this may beg the question "what are compilers really about?"

The process we know as compilation analyzes a source, represented as a stream of characters, at the lexical level, the syntactic level, and the semantic level, places the resulting program representation in a data structure and then *does something.* That *something* could be translating the code from one language into another. It could be rewriting the code to make it more efficient. It could even just return the same code, but with different formatting.

This tutorial will focus on the lexical analysis and parsing. It will not cover intermediate code generation, code optimization, type-checking, control flow analysis, garbage collection, or any semantic aspects of compilation. The goal of this tutorial is to demonstrate the application of finite automata and context-free grammars in compiler design.

2 Terms

A *lexeme* is the basic and abstract unit of analysis in a program. The concrete analog is a *token.* Tokens are separated by delimiters, typically whitespace. We do not consider tokens that are used for blocking, such as curly braces in Java, or for application[1], such as the parentheses in Lisp.

A parser performs syntactic analysis on lexemes. A parser could also easily do the work of the lexical analyzer, however modern compiler design seperates the two. Throughout this tutorial, we will examine the differences between a lexer and a parser.

We will consider the following two simple grammars. The first is for a calculator. The second is for a query language:

```
E ::= E OP E | ( E ) | NUMBER
NUMBER ::= NUMBER . NUMBER | NUMBER NUMBER | DIGIT
OP ::= + | - | * | / | ^
DIGIT ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

SQL ::= ( select *ID* from *ID* ) | ( select *ID* from *ID* PRED )
PRED ::= where COND | where COND group by *ID*
COND ::= *ID* = *ID* | count ( *ID* ) = NUMBER
```

3 Lexical Analysis

The objective of the Lexical Analyzer is to read some input from a stream and return a token. The input stream is typically a stream of characters, a compiler may also need to deal with other representations of input data. Typically if we are implmenting a Lexer, we expect the input stream to be of a different type than the output token.

The output token may be a string that is ostensibly "the same" as the item read in, a subclass of a Word object, or any other representation that is germane to the language paradigm being used for the implementation of the compiler.

We will begin by considering a Lexer that reads in a character stream and returns a Token Java Object.

[1] *application* here means something very specific - it is the resolution of binding variables to become bound variables. For example, in the Lisp code `(defunsquare(x)(*xx))`, x is the binding variable. We call this example an abstraction. When we call `(square5)`, we bind 5 to x. This is called an application. Note that when we resolve this expression to 25, we have an atom. Thus, the abstraction is the *expression* corresponding to the function call, not its resolution.

The token is the lexeme that will be used later in the Parser's abstract syntax tree (AST). Since our tokens are implemented as Objects, we can add extra fields to the Token to decorate them with information that will be relevant later in parsing. We may also want to subclass Tokens to produce specific types, such as Words and Numbers.

3.1 Knuth-Morris-Pratt and Aho-Corasick Algorithms

Let us tokenize the input string `(select salamanders from frogs)`. Each identifier begins with a substring that is the same as two of the keywords in our language. To make things more interesting, let's assume that there are two more keywords in our query language, `selectron` and `selselect`. To distinguish these three keywords, we will need an efficient way to backtrack. This is where the Knuth-Morris-Pratt Algorithm comes in.

```
$t \leftarrow 0$
$f(1) \leftarrow 0$
**for** $s \leftarrow 1$ to $n - 1$ **do**
  **while** $t > 0$ and $b_{s+1}! = b_t$ **do**
    $t \leftarrow f(t)$
  **end while**
  **if** $b_{s+1} = b_{t+1}$ **then**
    $t \leftarrow t + 1$
    $f(s + 1) \leftarrow t$
  **else**
    $f(s + 1) \leftarrow 0$
  **end if**
**end for**
```

Fill out the following table to trace the KMP Algorithm:

char	s	e	l	s	e	l	e	c	t
s									
f(s)									

The Aho-Corasick Algorithm is a generalization of the Knuth-Morris-Pratt Algorithm. Instead of, say, looping through any number of individual FA and testing keywords on each of these machines, it combines all keyword automata into a single graph. This graph is a special kind of data structure known as a trie. A trie is a cross between a FA and a tree; we know that FA are graphs and that trees are a type of graph. However, trees traditionally store data "in" their nodes. A trie "stores" its data in its edges, much like a FA.

4 Parsing

Though parsing is often described as the process of forming a parse tree, it is rarely presented as a tree data structure. Instead, the tree is "formed" by the control of the program. We can represent each action taken as a node in the tree, even though in reality these are just calls to functions pushed onto the machine's stack.

We use `*ID*` in the second grammar because we can't keep track of the resolution of `*ID*`s in a context-free grammar. This token could resolve to any string of characters (discounting the character encoding and limits in the language on the string length).

4.1 Top-Down (Predictive) Parsing

Top-down parsing is the approach that is easiest for humans to understand. In natural language, we can think of it as seperating a setence into the subject noun phrase and the predicate verb phrase. We can procede from there towork our way down the syntax tree to the parts of speech themselves and their corresponding tokens in our sentence.

Topmost parsing uses the leftmost derivation for a grammar, performing a preorder traversal on the parse tree.

4.2 Bottom-Up Parsing

Bottom-up parsing begins with the string of terminals, moving through the input from left to right, and constructing subtrees up to the root of the resultant parse tree.

The dominant style of bottom-up parsing is the general class of shift-reduce parsers. These parsers read input until a subtree can be formed, pushing tokens onto a stack. This is the "shift" action. Then when a production rule matches, they pop off the appropriate number of tokens, forming a subtree. This is the "reduce" action.

Recall that top-down parsing finds the appropriate production in sentential form and replaces a nonterminal with a rewrite rule that expands the production. Bottom-up parsing procedes in a different fashion. It instead begins on the right-hand side of a rewrite rule and works towards the left-hand side. The parsing algorithms presented here continue to process the input string from left to right, but the processing of the grammar is now right to left. Consequently, we refer to these as LR (left read, right recursive) grammars.

5 Notes