1 Introduction

The study of compilers provides a direct application of the concepts learned in Theory of Computation. This tutorial presents lexical analysis and parsing, which make heavy use of finite state machines.

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.

Many of you have seen some features of compilers in *Structure and Interpretation of Computer Programs*. The features emphasized there were farther along in the compilation processes. Before we can get to the point of lexical addressing and continuation passing, we need to start with reading in a source file.

This tutorial will focus on the lexical analysis and parsing techniques. 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, 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 separates the two. Throughout this tutorial, we will examine the differences between a lexer and a parser.

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.

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.

function KNP:
$t \leftarrow 0$
$f(1) \leftarrow 0$
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

4.1 Introduction

Parsing is the process whereby we represent the underlying structure of some emission, given an abstract representation of its components. This structure is assumed to be hierarchical. For programming languages this involves taking the output of a lexer and processing it in some way that can later be used in semantic analysis.

4.1.1 Preliminary Notes on Data Structures and Control

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. One exception to this rule is in universal parsing, where the parse is held in an array and returned as a data structure. Though this approach to parsing is not used in today's compilers, it has applications in other disciplines, provides a historical example of approaches to parsing, and will be our first example.

4.1.2 Context-Free vs. Context-Sensitive

Any approach to parsing must ensure that the grammar it is processing is context-free. We know that programming languages are not context-free. In order to allow the parser to recognize them, we must specify grammars in such a way that they temporarily obscure the context-sensitiveness of the language. We defer resolution of the context-sensitive components of the grammar until later semantic analysis. One classic example of a context sensitive grammar is the language $L = \{wcw | w \in [a - z]^*\}$, which we can interpret as variable declaration and instantiation, where variable names are variable-length lower case strings. Grammars will use reserved lexical symbols such as **id** for context-sensitive components

of the grammar. This example illustrates the general approach in grammar construction of abstracting over context-sensitive language features.

The following section introduces parsing techniques. Universal parsing can process any context-free grammar. The other parsing techniques we will consider are top-down §4.3 parsers, which process a subset of CFGs known as LL(k) grammars, and two types of shift-reduce §4.4.1 parsers, known as SLR §4.4.2 and LALR parsers §4.4.3, which process a subset of CFGs known as LR(k) grammars. The names of these parsing techniques and the grammars they parse are often used interchangably. The limitation that these parsers only process a subset of CFGs is sufficient for most programming language grammars.

4.2 Universal Parsing

Unlike the other parsing techniques we will examine, universal parsing techniques can be applied to any kind of grammar. They can successfully parse ambiguous grammars, returning multiple parse trees. You have seen one example of a universal parser in class, the CYK Algorithm. This technique uses a paradigm of algorithm design known as *dynamic programming*. Dynamic programming uses a kind of memoization to store the previously computed partial parses of an input emission. This chart causes the space complexity of the parse to be $O(n^2)$, where n is the length of the emission. The time complexity for the CYK algorithm is typically considered to be $O(n^3)$, since it contains triply nested loops. These loops move compare each of the partial parses of given substrings, in effect (though it is not expressed this way in the algorithm) scanning across the input, back, and across again. Multiple scans occur due to the need to consider ambiguous partial and, potentially, total parses. At each step, the partial parses must then be compared against each applicable production rule. The size of the grammar is also formally included in the complexity, however since it remains constant for any given string in the language and the applicable productions are comparatively much smaller than the size of the string, we usually do not include it in the complexity analysis.

Another instance of universal parsing is Earley's Algorithm, which also uses chart parsing. Earley's moves from left to right through the input emission, looking for applicable production rules. It uses three subroutines known as *predict, scan*, and *complete* to enumerate and hold in memory legal partial parses. Earley's algorithm also runs in $O(n^3)$ time.

It should be noted that while these two algorithms are listed separately from the remaining top-down and bottom-up parsing techniques, they are bottom-up and top-up respectively. Both universal parses can be made to run in less time if their grammars are in CNF. Heuristics knowledge about the grammar can bring the parse down to linear time.

Some sentence parsing tools in Natural Language Processing uses modified versions of universal parsing, which we know to be inherently ambiguous. Today's natural language parsing use probabilistic context-free grammars, which are typical CFGs decorated with probabilities. The parse tree returned is the one with the highest probability. These grammars can be used for both chart parsing and a version of shift-reduce parsing. There are other parsing techniques that may be of interest, which leverage heuristic knowledge of cognitive models of language. These are beyond the scope of this tutorial.

Universal parsing is not typically used in today's compilers, since other parsing techniques having a smaller time complexity. These alternates require their their grammars not produce ambiguous parses. Since we are parsing with the objective of feeding the result into a later stage of a compiler, we know that the formal grammar provided will be designed to limit ambiguity. Since grammars for programming languages can be fairly large, a designer may not know that the grammar they are designing is ambiguous. We cannot however write a program to tell whether or not a grammar is ambiguous; such a problem is undecidable, since it can be reduced to an instance of Post's Correspondence Problem. Instead, the parsing techniques we examine accept a subset of context-free grammars that have certain features and some make changes to the underlying representation in order to overcome ambiguity. We will consider an automatic parser-generator, which issue error messages when it encounters features of a grammar that are problematic.

4.3 Top-Down (Predictive) Parsing

There are two main approaches to top-down parsing that differ in the implementation of their control. They approach the problem of deriving a parse tree in the same way, though, which is by using the leftmost derivation for a grammar, performing a preorder (depth-first) traversal, building the parse tree.

The first technique we will examine is recursive descent parsing with prediction and the second is a kind of PDA.

Both approaches to top-down parsing accept the same class of grammars, known as LL(k). This class of grammars is a proper subset of CFGs and a superset of CNF grammars. LL(k) grammars permit nondeterminism, but enforces a variety of restrictions which will be discussed. Their limited lookahead is what allows them to be more efficient than universal parsing algorithms. Remember that no grammars in this and the following sections are permitted to be ambiguous. Also note that there is a difference between an ambiguous parse and an ambiguous grammar.

The first L in LL(k) corresponds to the way in which we read the input, from left to right. The second L corresponds to the way in which we analyze the production rules, which is from left to right here. That is, for a production of the form $A \to B|C$, we read from the right-hand side of the rule through each symbol in its production. Remember that this is the same as a leftmost derivation. The k corresponds to the predictive part of the parser, the number of grammar symbols we look ahead. That is, with predictive parsing for LL(k) grammars, we consider the right side of the production rule in our lookahead algorithm.

4.3.1 Recursive Descent

A naive implementation of a recursive descent parser might begin by finding applicable productions and doing a depth-first search to find the correct parse. There are two main problems with this approach. The first is that the time complexity risks running as high as our universal parsers. The second is that we risk running into an infinite loop. While we could recover from our infinite loop by doing a breadth-first search, we would still need to implement a recovery strategy for failed branches and would still have the aforementioned complexity issue, making this approach no better than the universal parsers.

We can get around this problem, eliminating the need to backtrack, by using two algorithms which will be important for all subsequent parsing techniques. These algorithms are known as FIRST and FOLLOW and they return sets of terminals.

Informally, FIRST is the set of nonterminals derived by taking the current production and drilling on the first nonterminal symbol until it keys some terminal symbols(s). If ϵ is in this terminal set, we add it to the set and go back up to the current production and repeat the process on the second symbol in. This is repeated until we hit all possible initial terminal symbols.

FOLLOW is the set of nonterminals derived by advancing along the current production, past the initial terminal symbols, to process the next nonterminal production rule. That is to say, if we are looking at some string of grammar symbols the form $\alpha B\beta$ or αB, where α are the already derived terminal symbols and B is some production rule yielding as-yet unknown terminals, then FOLLOW computes this set of nonterminals by drilling down the "middle" of the production rule. We use FOLLOW in our lookahead to see what potential nonterminals might come next.

How are FIRST and FOLLOW actually used in recursive descent parsing? If we think of each production rule as a procedure that processes some input, returns the appropriate nonterminal symbol, and decides which production rule to apply, then these algorithms are used inside the procedures represented by production rules to determine the next recursive call. They are the analogue of moving around a finite state control, while the call stack from the recursion is the analogue to the stack of a PDA.

Recursive descent parsers are sufficient for little languages, but are limited by their k grammar lookahead. They do not consider lookahead in the input. If a recursive descent parser is used, its k is rarely greater than 1; we can derive greater power more efficiently using other techniques. The kinds of constructs they succeed at parsing are typical control flow constructs defined by grammar productions such as:

$$stmts ::= \textbf{do}\ stmts \mid \textbf{while}\ expr\ stmts \mid \{stmts\} \tag{1}$$

Recursive descent parsers fail on what we call the "dangling else" production:

$$stmt ::= \textbf{if}\ expr\ \textbf{then}\ stmt \mid \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \tag{2}$$

We see here an example of the limitation on the type of grammar that can be produced. Note that both productions in the "dangling else" begin with not only the same terminals, but the same string of terminals and nonterminals. Consider the production rule $A \to \alpha \mid \beta$. We can see that, for our parser to work, FIRST(α) and FIRST(β) must be disjoint sets. If they are not disjoint sets, we will have

ambiguity for our limited lookahead and would need to implement a backtracking algorithm. A similar limitation exists for the FOLLOW algorithm, and the sets resulting from both algorithms can only have one ϵ production in them.

4.3.2 Nonrecursive Predictive Parsing

Rather than relying upon the emergent control of procedure calls and the call stack, nonrecursive predictive parsing keeps an explicit stack of grammar symbols and a table of valid production rules. This implementation is directly analogous to a PDA, where the table represents the control of the finite state machines. Aside from its implementation details, nonrecursive predictive parsing is equivalent to recursive descent parsing in its complexity and the limitations of its grammar.

4.4 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.

4.4.1 Shift-Reduce Parsing

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. Shift-reduce parsers also return an accept action or can signal errors.

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 proceeds 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(k) (left read, right recursive) grammars.

Like nonrecursive predictive parsing, shift-reduce parsing maintains an explicit stack as it forms the parse tree. This stack holds as one of its items what we call a handle. The motion of shifting and reducing, combined with the handle, form the control of the program.

The handle refers to the portion of the production rule that we are currently working on. Since we are working from right to left in the rewrite rule, the leftmost substring of the body of the production rule does not need to be in the handle. The handle is just the substring of grammar symbols that we are trying to match with a production rule.

Naive implementations of shift-reduce parsing run into trouble because conflicts may arise when determining whether to shift or reduce or in choosing the next production rule. Identifying the actual substring to match puts some limitation on the kinds of grammars that we can process with shift-reduce. While the set of LR(k) grammars is still a subset of CFGs, it is larger than the set of LL(k) grammars. This is because the lookahead for LR grammars refers to the lookahead of the input strings, rather than the production rules. It allows us to match larger units. It still does not allow us to match the dangling else problem.

4.4.2 SLR Parsing

SLR parsers are simple LR parsers. They use the shift-reduce technique, along with an explicit control and the sets generated by the FIRST and FOLLOW algorithms. The subsequent LALR parsers that we see are just a modification of the SLR parsers.

SLR parsers use a table to store productions in a manner similar to the nonrecursive predictive parsers. Each entry in the table is a production indexed on terminals and nonterminals. Subtrees are stored on the stack, which contains the handle, corresponding to the last completed subtree of the parse. The decide when to shift and when to reduce, SLR parsers build a finite state machine. This FSM is created in two stages. The first stage does not consider the lookahead. This FSM represents all possible production paths for the grammar and is called the canonical LR(0) automaton. A state in the automaton represents a set of production paths which refer to expansions of a production rule, given

we have read in a certain amount of input. These production rules are enumerated using an algorithm called CLOSURE, which we will not discuss in detail here.

The SLR parser then constructs a function called GOTO, which provides the transition rules for the states in this automaton. We can parse a sufficient number of grammars using the SLR(0) automaton. If we want to add lookahead, we add the use of FIRST and FOLLOW to the transition algorithm. Currently we think of each state in the automaton as a set of production rules. We can now build mini state machines inside each state to choose which production rule we need and use FIRST and FOLLOW to determine our transitions for languages in LR(1).

There are many layers to the implementation of LR algorithms, which we will not discuss in detail here. The important idea is that SLR merges many of the techniques we have seen thus far. In creating the canonical automaton, it in essence converts the grammar from one language to another. This representation is complex, even for a small grammar. We follow a process of providing data in one language in a human-readable and general formation and then transforming it into a new representation, or what might even be called a new language, that is optimized for its specific purpose.

4.4.3 LALR Parsing

The canonical LR(1) automaton will end up containing a large number of states. With some choice changes, we can construct another automaton that will parse a large set of grammars. This parser is know as the LALR parser, where the LA stands for "look ahead."

LALR parsers are very complex, but some important points to know about their construction is that, like the states in the control of the Turing machine we examined in class, their states contain a finite amount of memory. This memory is used for disambiguation and is a large part of the reason why the LALR parsers can recognize a larger class of languages than SLR parsers. The LALR parsers also rely on the optimization of the canonical LR(1) automaton. They combine redundant states and merge redundant entries in the parsing table. They can also convert certain nondeterministic operations into deterministic ones, which helps increase the speed of the operations.

4.4.4 Parser Generators

Implementing LALR parsers, like implementing SLR parsers, is complex and tedious. Rather than implementing a new parser for every language we write, most programmers today use a parser-generator program to output a parsing program specific to their needs. Most parser-generators create LALR parsers.

The SLR and LALR parsers both run in linear time and are the choice of parser for today's programmer. The LALR reduces the amount of space needed to represent the parser and so it is typically used in parser generators. One sacrifice we make for speed, though, is in the set of languages that a given parser can accept. Since we write our programming languages to be unambiguous in their grammar, this is not a problem. However, the programmer is fallible. If a programmer enters an ambiguous grammar, there is a rich set of error reporting that can be returned to someone using a parser-generator. Parser-generators also allow programmers to include precedence information for symbols in the grammar, to help disambiguate.

5 References

The classic reference for compiler design is Aho and Ullman's *Principles of Compiler Design*, known as the "dragon book." Various editions exist. The most recent is the "purple dragon book," which includes in its author list Lam and Sethi. Its example code is in Java. The previous edition, known as the "red dragon book," has Sethi as a coauthor and its examples are written in C. You can see more detailed explanations of the algorithsm for implementing compiler techniques in these books.

For examples of emergent data structures from the fusion of physical data structures and control flow mechanisms, see `http://github.com/etosch/graph`. I am currently working on a general graph module in Clojure for some Genetic Programming applications. There are implementations of various graphical schemas here, which you can take a look at.

The Lexer and Parser programs (Flex and Bison) I demonstrated during the tutorial can be found at `http://flex.sourceforge.net` and `http://www.gnu.org/software/bison/`.