A Simple One-Pass Compiler

Compilers: Principles, Techniques, and Tools Ed. 1 (1986)

1. Overview

A programming language is defined by describing what the programs look like (syntax), and what the programs mean (semantics). To describe the syntax, the convention is to represent it in a notation known as context-free grammars (CFG) or Backus-Naur Form (BNF). It is however much more difficult to describe the semantics in such form, thus we will use informal and suggestive descriptions.
It is also useful to translate infix expressions into postfix form, known as reverse polish notation. In this notation, all operands come first, followed by the operations. For example, the postfix form of `9-5+2` is `95-2+` (we do `9-5`, then add `2`). This is useful since this is the exact order that a computer performs computations using a stack. Thus, we will begin by constructing a simple program that translates an expression consisting of digits and addition/subtraction operations into postfix form.
In this compiler, the lexer converts the source code into a stream of tokens, the smallest unit of information for the parser.

2. Syntax Definition

Context-Free Grammar: A form of representing the syntax of a language, which consists of the following components:

1. A set of tokens, known as *terminal* symbols.
2. A set of nonterminals.
3. A set of productions. Each production has the same format: the left side consists of a nonterminal, followed by an arrow, followed by a sequence of tokens and/or nonterminals.
4. A special nonterminal known as the *start* symbol.

For example, an infix notation grammar may look like the following:

$$list \to list + digit \tag{1}$$
$$list \to list - digit \tag{2}$$
$$list \to digit \tag{3}$$
$$digit \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{4}$$

The right hand side for the nonterminal *list* can also be grouped as:

$$list \to list + digit \mid list - digit \mid digit$$

A production is *for* a nonterminal iff the nonterminal appears on the left hand side. A string of tokens is a sequence of zero or more tokens. Note that the string containing zero

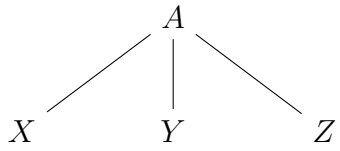
tokens, known as the *empty string*, is also valid, and is notated with ϵ.
We derive strings from a grammar by repeatedly replacing nonterminals on the right side of a given production, until the string consists of only terminals. All token strings that can be derived from the start symbol define the language of the grammar. For example, from the grammar above, we can define `9-5+2` as follows:

1. `9` is a *list* by production (3), since `9` is a digit.
2. `9-5` is a *list* by production (2), since `9` is a list and `5` is a digit.
3. `9-5+2` is a *list* by production (1), since `9-2` is a list and `2` is a digit.

Since our final expression consists of only terminals, the full expression is valid in this language.

2.1. Parse Tree

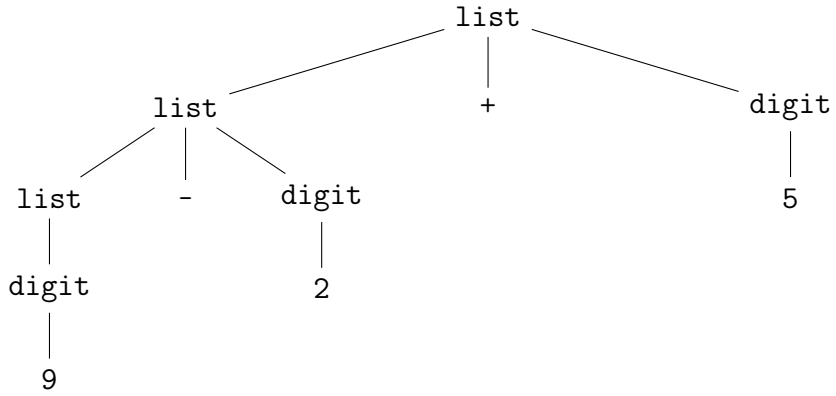
A parse tree is a way to graphically visualize how a string in a language can be derived from the start symbol. Suppose a start symbol A with production $A \to XYZ$, then it's parse tree may look like:



We can generate a parse tree from a CFG with the following properties:

1. The root is the start symbol.
2. Each leaf is a terminal or ϵ
3. Each interior node is a nonterminal.
4. If A is an interior node with children $X_1, X_2 ... X_n$, then $A \to X_1 \mid X_2 \mid ... X_n$ Is a production.

The leaves of a parse tree, read left to right, form the *yield* of the tree, which is the string that is derived from the start symbol. For example, using the production described above,



We can derive the expression `9-2+5`.

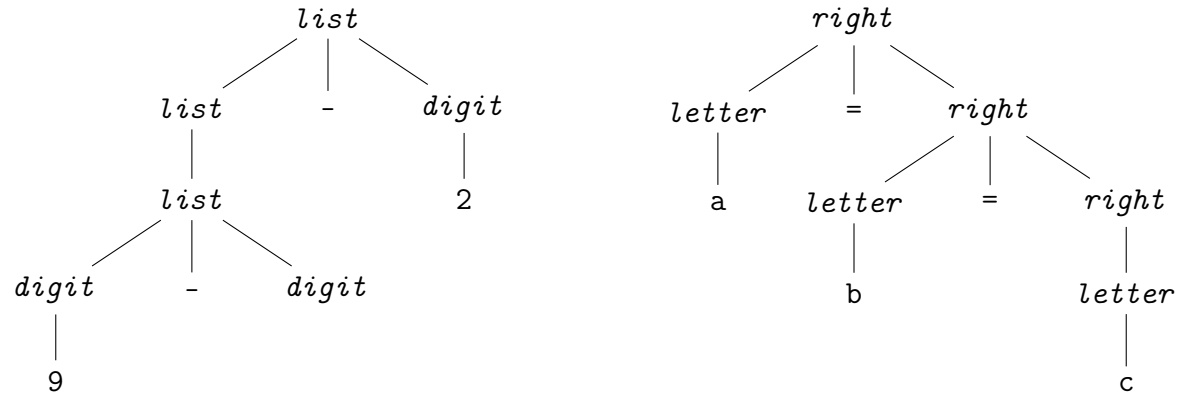
The process of finding a parse tree for a given string is known as *parsing*. We will discuss specific techniques in later sections.

2.2. Ambiguity

An *ambiguous* grammar is one in which a string can be derived with multiple parse trees. To show that a grammar is ambiguous, we need to find at least one token string that has more than one parse tree. Since such a string can be understood by the compiler in different ways, we need to construct an unambiguous grammar, or use ambiguous grammars with additional constraints.

2.3. Associativity of Operators

By convention, the statement `9+5+2` is equivalent to `(9+5)+2`, since addition and subtraction are done from left to right. We say that these operations are *left associative* because an operand with a plus/minus symbol on the left and right will take the left side first. In most languages, the basic arithmetic operations (addition, subtraction, multiplication, and division) are taken to be left associative. There are also some operators that are right associative; for example, consider `a=b=c`. This would be parsed as `a=(b=c)`, instead of `(a=b)=c`. The difference in parse trees for these expressions is as follows:



2.4. Precedence of Operators

Consider the expression `9+5*2`. There are two possible interpretations: `(9+5)*2` or `9+(5*2)`. We say that `*` has *higher precedence* if it takes operands before `+`. In normal arithmetic, multiplication and division are performed before addition and subtraction. Thus, we would take `9+(5*2)`. We can form a table that represents both the associativity and precedence of operations. First, start with a precedence table, where operators are organized in order of increasing precedence:

left associative: `+ -`

left associative: `* /`

Since we have two levels of precedence, we need two nonterminals, *expr* and *term*. We will also create a new nonterminal *factor* to generate basic units. These units are expressions formed by digits or parenthesis.

$$factor \rightarrow \textbf{digit} \mid (\ expr\ )$$

Next, we will consider the operators with highest precedence, `*` and `/`. Sine these associate to the left, their productions will be similar to *list*, which also associated to the left:

$$term \rightarrow term\ *\ factor \mid term\ /\ factor \mid factor$$

expr also generates a list of terms in a similar fashion:

$$expr \rightarrow expr\ +\ term \mid expr\ -\ term \mid term$$

Thus, our final syntax tree:

$$expr \rightarrow expr\ +\ term \mid expr\ -\ term \mid term$$

$$term \rightarrow term\ *\ factor \mid term\ /\ factor \mid factor$$

$$factor \rightarrow \textbf{digit} \mid (\ expr\ )$$

3. Syntax-Directed Translation

A compiler may need to keep track of quantities other than the code generated. For example, it may need to know the location of the first instruction in memory or the number of instructions generated. An *attribute* is an abstract quantity that we associate with a syntax construct.
Syntax-Directed Definition: A definition that specifies the translation of a construct in terms of attributes associated with its syntactic components.

3.1. Postfix Notation

The postfix notation for an expression E is defined as follows:

1. If E is a variable or constant, then the postfix form of E is E.
2. If E is an expression of the form $E_1\ op\ E_2$ where op is a binary operator, then the postfix form is $E_1'\ E_2'\ op$, where E_1' and E_2' are the postfix forms of E_1 and E_2, respectivel y.
3. If E is an expression of the form $(\ E_1\ )$, then the postfix notation for E_1 is also the postfix notation for E.

Note that an expression in postfix notation shouldn't have parenthesis, as the position of the operators along with the number of operands prevents ambiguity. For example, the postfix notation for `(9-5)+2` is `95-2+`, whereas the postfix notation for `9-(5+2)` is `952+-`.

3.2. Syntax-Directed Definitions

A syntax-directed definition uses a CFG to specify a syntactic structure of a given input. For each grammar symbol, it associates it a predefined set of attributes, and with each production, a set of semantic rules for computing values of the attributes associated with the symbols seen in the production. The syntax-directed definition is therefore composed of a grammar and a set of semantic rules.

We can think of a translation as a mapping bewteen some input and some output. for an input x, we can perform the following steps to get the output:

1. Construct a parse tree for x.
2. For a node n with grammar symbol X, we denote $X.a$ as the value of attribute a for X.
3. The value $X.a$ is calculated with the semantic rule for attribute a associated with the X-production.

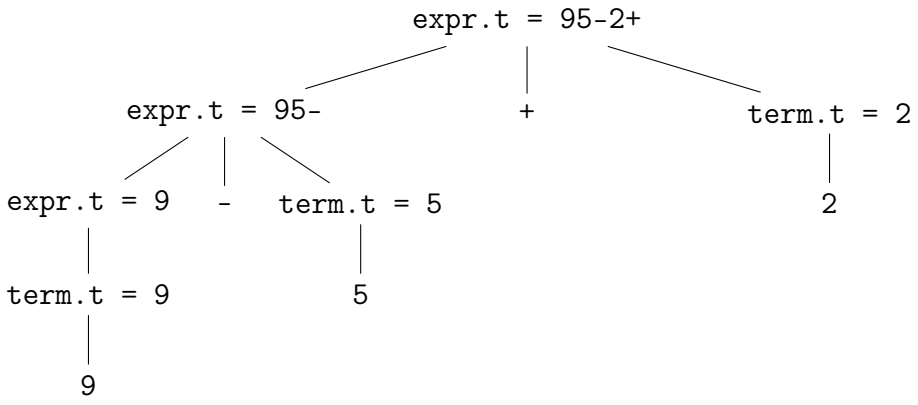
The final parse tree that has all attribute values is known as an annotated parse tree.

3.3. Synthesized Attributes

An attribute is said to be *syntehsized* if its value at a node in a parse tree is derived from it's children nodes' attriubutes. These attributes are desirable, as they can be found in a single bottom-up traversal of the parse tree.
Seen below is an example of a syntax-directed definition for translating digits seperated by plus or minus signs into postfix notation.

Production	Semantic Rule
$expr \rightarrow expr_1 \ + \ term$	$expr.t \ := \ expr_1.t \ \| \ term.t \ \| \ '+'$
$expr \rightarrow expr_1 \ - \ term$	$expr.t \ := \ expr_1.t \ \| \ term.t \ \| \ '-'$
$expr \rightarrow term$	$expr.t \ := \ term.1$
$term \rightarrow 0$	$term.t \ := \ '0'$
$term \rightarrow 1$	$term.t \ := \ '1'$
$\ldots$	$\ldots$
$term \rightarrow 9$	$term.9 \ := \ '9'$

Here, we encode the rules of postfix notation; when we encounter a digit, the postfix is that digit itself. This is shown in, for example, $term \rightarrow 9$ defines $term.t$ to be 9. Whenever we encounter the production $expr \rightarrow term$ in the parse tree, the value of $term.t$ becomes $expr.t$. The semantic rule $expr.t \ := \ expr_1.t \ \| \ term.t \ \| \ '+'$ defines addition; note that the subscript 1 is only there to differentiate the $expr$ on the left and right side. The $\|$ denotes string concatenation. Thus, our annotated parse tree would be as follows:



3.4. Depth-First Traversal

Notice that the syntax-directed definition only defines the language, not how it is evaluated. Thus, there are many different ways we can evaluate the attributes for the parse tree. Any order that computes an attribute a before all attributes that depend on a is valid. The translations in this chapter can all be implemented by evaluating the semantic rules for the attributes in a parse tree in a predetermined order. A *traversal* of a parse tree starts at the root and visits all nodes some order. A "depth-first" traversal is one where the semantic rules for a given node are evluated once all of the descendants of that node are evaluated. We visit an invisited child of a node whenever we can, so we try to visit nodes as far away from the root as quickly as possible. The pseudocode is written below:

prodedure *visit*(n: node);

begin

for each child m of n, from left to right **do**

visit(m);

evaluate semantic rules at node n

end