Parser Design

For technical reasons, finite state machines are not powerful enough to deal with most languages; this is because it is impossible to set a viable limit on the number of "parentheses" in advance. It is necessary to move to a more powerful model, **context free languages.** Specification of context free languages is more complicated than for finite state machines, but the concepts are similar. We are unable to us a simple ST graph for context free languages; this is due in part to the need to specify recursion.

Grammars and Production Systems

Background for this Case

We could encode the ST graph information through the use of **production systems.** A *production system* uses the concept of a **grammar** and **algebra.**

A formal grammar *G =(N,T,G,S)*, where

- *N* is a set of states (just like in the ST graph) called **nonterminals symbols** or **categorical symbols.** We will use the **categorial symbol** teminology because it makes the link to grammars in natural language easier.
- *T* is a set of elements called **terminal symbols** or just **terminals.** Terminal symbols are such things as words and punctuation marks. This makes it clear where the scanner comes in.
- *G* is a set of **grammar rules.** Much more below.
- *S* is a distinguished (meaning we know it's identity in each case) element in the set of nonterminals *N*. In keeping with our attempts to link the parser to natural language, *S* is called the **sentential symbol.**

ST Graphs to Production Systems

We could describe the finite state automata representation lexical rules by productions developed by the following rules:

- Name all the nodes of the graph. These names become the categorical symbols.
- For every arc, there is a **source node** and a **sink node.** The **terminal symbols** are the weights of the arcs.
- For every source node, say *A,* and every sink node, say *B,* write down a grammar rule of the form *A->tB,* where *t* is the weight of the arc. If there is no such *t,* then discard the rule.

The second requirement is to move the functions from the ST graphs. The convention that has evolved is that we write the productions as above, then enclose the **semantic actions** in braces.

In effect, we have laid the foundations for a very important theorem in formal languagues: Every language that is specifiable by an ST graph is specifiable by a grammar. The content of the theorem is that many apparently different ways of specifying computations are in fact the same.

Parse Trees

Remember the idea of a derivation

E=>T+E=>T+F*T=>T+F*F=>T+F*3=>T+2*3=>F+2*3=>1+2*3.

The derivation induces a tree structure called a **parse tree.** When thought of as an automaton, the tree is the **justification** for accepting any sequence of tokens. Thought of as a machine, however, the tree is not the whole story.

Productions for Arithmetic Expressions

E→ T+E { $$=$1+$2}
E→ T-E {$$ = $1-$2}
E→ T {$$ =$1}
*T→ F*T {$$=$1*$2}*
T→ F/T {$$ =$1/$2}
T→ F {$$ = $1}
F→ any integer { $$=atoi($1)}
F→ (E) { $$ = $2 }

Now the productions define a *function* that does two things: it recognizes syntactically correct inputs *and* it can compute with the inputs. Instead of the simple derivation above, we can think about tokens for the terminals and values for the categorical symbols. I will use the `?' to indicate a value that is not yet computed. For this discussion to make sense, I will follow the **leftmost canonical derivation rule (LCDR).** LCDR requires that the parse be done by always expanding the leftmost categorical symbol. We do this in a topdown manner. Such a parser is called a **predictive parser.** Let's run it on *1+2*3*.

<E,?>

<E,?>=><T,?><atom,+><E,?>=>

<F,?><atom,+><E,?>=>

Now, we run into the first terminal symbol. The production of interest is the *F→ any integer { $$=atoi($1)}* production. In words, it says "the next input is an integer. Recognize it. If it is there, then perform the function *atoi* on the token *$1*. It the conversion works, assign the result to *$0*"

<F,<int 1>><atom,+><E,?>=>

The parser returns to the previous level, so we have the F replaced by T. The parser now predicts that the atom <atom,+> is next. When the parser sees the `+', it reads on.

<T,<int 1>><atom,+><T,?>=><F,<int,1>><atom,+><F,?><atom,*><F,?>=> ...

Repeating the same reasoning, we end up with a final string of

<T,<int,1>><atom,+><F,<int,2>><atom,*><F,<int,3>>

and we start returning. We return to a level that has the `*' in it. The correct computation takes the 2 and 3 and multiplies them. So we get

<T,<int,1>><atom,+><T,<int,6>>

We can now compute the last operation and get 7. So <E,<int,7>> is the final value.

Asssignment

Take the grammar for the project and produce a production system that parses incoming tokens and builds a tree that represents the actual **parse tree.**