Grammars and Things Grammatical

We will use the vocabulary of **formal languages** to describe formal programming languages and their parsers.

A **grammar** *G* is a four-tuple *(V,T,P,S)* where

1. *V* is a set of **categorical symbols;**
2. *T* is a set of **terminal symbols;**
3. *P* is a set of **grammar rules;** and
4. *S* is a member of *V* known as the **sentential symbol.**

In order to have a definite example and to maintain continuity with the case studies, we consider the grammar for arithmetic expressions.

Basic Terminology

The motivation for formal languages is natural language, so the initial concepts are all borrowed from natural language. Grammars describe **sentences** made out of **words** or **symbols.** Both *V* and *T* are sets of such words or symbols.

A **language** would be all the sentences that are judged correct by the grammar: this is called **parsing**. Technically, we define the *Kleene closure operator A** to mean zero or more occurences of *A*. This notation is probably familar to you from the use of regular expressions and as the "wildcard" operator in denoting file names. If *A* is a set, then *A** is every possible combination of any length. If *T* is the set of all possible terminal symbols, then *T** is the set of all possible strings that could appear to a language.

Which strings of *T** are the "legitimate" ones? Only those that are allowed by the rules of *P*. So how does this particular description work?

Productions

The productions are two words, say *Y* and *Z*. Conventionally, a production in *P* is denoted by $Y \rightarrow Z$. In order to have a definite example, we consider

V = { E, T, F}

*T = {+, -, *, /, (,), any integer}*

Productions for Arithmetic Expressions

$E \rightarrow T+E$
$E \rightarrow T-E$
$E \rightarrow T$

*T→ F*T*
T→ F/T
T→ F
F→ any integer
F→ (E)

Note. There is a contextual use of the symbol *T* in two different ways: *T* in the definition for grammars as the terminal set and *T* in our example grammar (short for *term).* Such overloading of symbols is common in theoretical discussions; there are only so many letters. Therefore, you must learn to be cognizant of the context of symbols as well as their definition in context. Ambiguity should arise only when two contexts overlap.

Derivations

A sentence is structured correctly if there is a **derivation** from the starting symbol to the sentence in question. Derivations work as shown in this example. Production are **rewriting rules**. The algorithm works as follows.

1. Write down the sentential symbol.
2. Check the lefthand sides for matches in the word derived so far. For each match, rewrite the word by replacing the matched portion with the righthand side of the rule.
3. Continue until there are no changes.
4. If any word matches the input word, then the input word is grammatically correct.

We use the "=>" symbol to symbolize the statement "derives". That is, read "A=>B" as "A derives B".

Now for an example. Is "1+2*3" grammatically correct?

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

So, yes: "1+2*3" is grammatically correct.

Notice that the parse given is not the only one. For example, I could have done the following:

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

In order to keep such extra derivations out of contention, we impose a rule. We call this the

Leftmost canonical derivation rule: Expand the leftmost non-terminal symbol first.

Production Systems

A useful generalization of grammars are called **production systems.** Production systems are used extensively in artificial intelligence applications. They are also the input concept in parser generating tools like *LEX* and *YACC*. The basic idea is pattern matching followed by execution of program segments as well as rewriting the graph. Instead of a long discussion, here's an example taken from our arithmetic example.

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 }

This is the famous hand calculator problem from the famous "Dragon Book": Alfred V. Aho ,Jeffrey D. Ullman. *Theory of Parsing, Translation and Compiling*. Prentice-Hall. 1972. (Out of Print).

The way to read *E→ T+E { $$=$1+$2}* is as follows: $$ is the semantic content of the lefthand symbol while $n is the semantic value of the nth symbol on the right hand side. The interpreter for this language would first apply the production *E→ T+E;* if the production succeeds, the program segment (after translating $'s to memory locations) is then executed.