Programming Languages

Grammars for Describing Programming Languages

Introduction

The set of rules that tell you whether a sentence is correctly structured is called a *grammar*. Here is a simple, and very incomplete, grammar for English:

```
sentence : nounPhrase verbPhrase PERIOD
nounPhrase : ARTICLE ADJECTIVE NOUN
verbPhrase : VERB nounPhrase
```

In this description, the three lines are *grammar rules*. The token before the colon is the thing you want to build (or deconstruct). The tokens that appear after the token describe how one constructs or breaks down the thing to the left of the colon. The tokens that are all-caps, such as PERIOD and ARTICLE, refer to things that cannot be broken down further and, int this example, refer to English words and punctuation. The tokens that are all (or mostly) lowercase denote higher-level structures that can be broken down. The first of the three rules can be stated as:

A sentence is a noun phrase followed by a verb phrase and terminated with a period

Given these nouns,

```
cow
water
dog
```

these articles,

```
the
a
```

these verbs,

```
drank
chased
```

and these adjectives,

```
large
cool
yappy
```

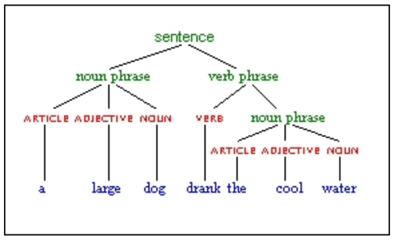
we can generate a number of syntactically correct sentences, such as:

```
The large cow drank the cool water.
A yappy dog chased the large cow.
The cool water chased the cool dog.
The yappy water drank a large cow.
```

Grammars can also be used to recognize syntactically correct sentences. For example, the sentence:

```
A large dog drank the cool water.
```

can be fit to the grammar above in the following way:



Note that the grammar above does not allow for the sentence:

```
The large yappy dog drank the water.
```

since the grammar requires a single adjective in a noun phrase. We can modify the grammar to allow sentences of this form (and still allow for the sentences that were allowed before modification):

```
sentence : nounPhrase verbPhrase PERIOD
nounPhrase : ARTICLE adjList NOUN          //ADJECTIVE changed to adjList
verbPhrase : VERB nounPhrase
adjList : ADJECTIVE | ADJECTIVE adjList
```

The newly added rule, *adjList*, says that:

> *An adjective list is an adjective or it is an adjective followed by more adjectives*

The symbol `|` denotes a choice.

We can also allow for sentences like:

```
The dog drank water.
```

with a further modification:

```
sentence : nounPhrase verbPhrase PERIOD
nounPhrase : optArticle optAdjList NOUN    //modified
verbPhrase : VERB nounPhrase
optArticle : ARTICLE | *empty*             //modified
optAdjList : adjList | *empty*             //modified
adjList : ADJECTIVE | ADJECTIVE adjList
```

The *optArticle* rule says that:

> *An optional article is an article or is omitted*

Syntax versus Semantics

Syntax tells you if the sentence is constructed correctly. Semantics tells you whether a correctly structured sentence makes any sense. The sentence:

```
The box drank water.
```

is correct syntactically speaking (subject then verb then object) but seems wrong since boxes don't normally drink. An oft-made mistake is to attempt to enforce semantics with a grammar, especially when developing grammars for programming languages.

Programming Language Grammars

A grammar simply describes the structure of the sentences in a language. A source code program can be consider a collection of sentences, with the sentences corresponding to programming language expressions. Therefore, a grammar can be used to describe expressions.

As an example, the expression that involves adding two numbers together might be described as: A sum is a `NUMBER` followed by a `PLUS` sign followed by another NUMBER. Noting that we can also add variables bound to numbers as well, we can make the description more general by using two descriptions or rules:

> A *sum* is a *unary* followed by a `PLUS` sign followed by another *unary*

and

> A *unary* is a `NUMBER` or a `VARIABLE`

We use the name *unary* to denote something that is not a binary expression. We can formalize this concept of description and decide upon some conventions for specifying a grammar. The most common descriptive formalization for grammars is the Backus-Naur Form, or BNF, originally developed to formally specify the grammar of FORTRAN. We will use a simplified subset of BNF that is very easy to read and, when the time comes, easy to implement.

BNF is best described through a series of examples. The *sum* pattern described above would be rendered as follows in BNF:

```
sum : unary PLUS unary

unary : NUMBER

unary : VARIABLE
```

In a rule, the item to the left of the colon is known as a *non-terminal*, that is to say, something that can be broken down into finer parts. How the non-terminal is broken down is described to the right of the colon. In this case, a sum can be broken down into a PLUS sign and two primaries. When something cannot be broken down any further, it is denoted a *terminal*, that is to say, the breaking down process terminates. As a convention, non-terminals will be written in mostly lower-case letters, while terminals will be written in mostly upper-case letters. As another convention, the non-terminal to the left of the colon is thought of as the name of the rule. In the grammar above, we might say we have a *sum* rule and two *unary* rules.

Note that both terminals and non terminals can appear on the right-hand-side of a rule. As a consequence, a non-terminal on the right-hand-side of a rule must necessarily appear on the left-hand-side of at least one rule.
From the example, it can be seen the that the same non-terminal can name more than one rule. Multiple rules headed by the same non-terminal indicate a choice. In this case, a unary is a NUMBER or it is a VARIABLE. Rather than have two *unary* rules, the rules can be combined using the vertical bar, or *pipe*, symbol:

```
unary : NUMBER | VARIABLE
```

Note that the vertical bar has very low precedence. Rules with choices are often written to emphasize the low precedence of the vertical bar:

```
unary : NUMBER
      | VARIABLE
```

Concatenation (one thing following another) and alternation (one thing or another) are the only operators we will need from BNF. Other operations such as the *Kleene star* and *Kleene plus*, which indicate zero or more of a thing and one or more of a thing, respectively, as well as parenthetical grouping, make for a more concise grammar, but can get in the way of turning the grammar into an actual program for determining whether a sequence of words are correctly ordered or not. For this reason, we will make do with concatenation and alteration alone.

Recognizing lists

Consider the problem of finding a grammar rule for recognizing a comma separated list of items. With respect to programming languages, lists are often used to supply arguments in a function call. A list of three items is easy:

```
list : item COMMA item COMMA item
```

as is a list of four, five, and six items. We can even write a rule for two or three items:

```
list : item COMMA item
     | item COMMA item COMMA item
```

We could continue this approach by limiting the number of items in a list and providing an alternative for each length under the limit. In addition to being tedious and inelegant, it would be a very poor programming language that restricts the number of arguments to a user defined function. A recursive grammar rule can bail us out of this dilemma, letting us have lists of arbitrary length:

```
list : item
     | item COMMA list
```

This rule reads literally as "a list is either an item or it is an item followed by a comma followed by a list of items".
Following in this vein of recognizing lists with respect to function calls, what about a function call of no arguments or one argument? Here is a rule giving three alternatives for function calls...

```
functionCall : VARIABLE OPEN_PAREN CLOSE_PAREN
             | VARIABLE OPEN_PAREN item CLOSE_PAREN
             | VARIABLE OPEN_PAREN list CLOSE_PAREN
```

where OPEN_PAREN and CLOSE_PAREN are terminals describing open parenthesis and close parenthesis, respectively. Do we need separate rules for describing function calls of zero, one, or multiple arguments? Close inspection of the list grammar rule reveals that a list can be a single item, so the second alternative of *functionCall* is unnecessary. A convenient way to combine the first and third alternatives is to introduce the concept of an optional list, where an optional list is a list or no list at all. The rule for *functionCall* then becomes:

```
    functionCall : VARIABLE OPEN_PAREN optionalList CLOSE_PAREN
```

with *optionalList* defined as:

```
    optionalList : list
                 | *empty*
```

The symbol `*empty*` denotes empty or nothing. Literally, the *optionalList* rule says that "an optional list is either a list or it is nothing at all".

Left versus Right Recursion

Grammar rules themselves, at least the recursive ones, have a handed-ness. The recursive version of the *list* rule, given above, is *right recursive*, as the recursive part of the alternative is on the right. We could have written the *list* rule like this:

```
    list : item
         | list COMMA item
```

or even like this:

```
    list : item
         | list COMMA list
```

In the first of these rewrites, the non-terminal *list* appears immediately after a colon or a pipe. Since *list* appears leftmost on the right hand side of an alternative, we say the rule is *left recursive.* The second rewrite is both left and right recursive.

Theoretically, it does not matter if a rule is left or right recursive, Practically, one way may be preferable to the other. When a grammar is used to build a structured version of the input, known as a *parse trees*, left recursion is preferable for left-associative operations, Similarly, right recursion is preferable for right-associative operations. On the other hand, it is extremely easy to implement code to build parse trees from a purely right recursive grammar, so when it becomes time to do that, we will transform any left recursive rules to right recursion.

A Programming Language Grammar

Recall that expressions can serve as the sentences of a programming language. Once you have a grammar for expressions, 90 percent of the work in developing a language is done. We've already looked at one kind of expression, a sum of numbers. We also might like a product of numbers as well:

```
    product : unary
            | unary TIMES product
```

where `TIMES` is a terminal representing multiplication. Noticing the similarity between the product and sum grammar rules, we can generalize the two into:

```
    expression : unary
               | unary operator expression

    operator : PLUS | TIMES
```

An unintended, yet beneficial, consequence of this generalization is that we can now recognize expressions involving both `PLUS` and `TIMES`, as in: $4 + 3 * 2$. The operator rule is also easily extended by adding alternatives for the terminals representing the subtraction and division operators. We can also extend the *unary* rule to encompass other things that can be combined by operators.

```
    unary : NUMBER
          | VARIABLE
          | OPEN_PAREN expression CLOSE_PAREN
          | MINUS unary
```

The third alternative allows us to group expressions parenthetically, overriding any precedence of operators we wish to implement. The last alternative allows us to the negate the values of variables and other unaries, as in:

$$-a + -(b * -c)$$

For another example, consider expanding the unary rule so that expressions can involve function calls:

```
unary : NUMBER
      | VARIABLE
      | OPEN_PAREN expression CLOSE_PAREN
      | MINUS unary
      | VARIABLE OPEN_PAREN optExpressionList CLOSE_PAREN
```

The third alternative describes a C-like function calls.

The grammar rule and implementation of *optExpressionList* is similar the grammar rules for *list* seen earlier.

Conditionals

The simplest conditional is the *if statement*. To simply things, we will force both the *then* clause and the *else* clause to be *blocks* which we will describe later. Finally, we will allow a generic expression to serve as the test expression.

```
ifStatement : IF OPEN_PAREN expression CLOSE_PAREN block optElse

optElse : ELSE block
        | *empty*
```

This grammar rule forces a chain of *if statements* to be nested:

```
if (a < b)
    {
    result = a;
    }
else
    {
    if (b < c)
        {
        result = b;
        }
    else
        {
        result = c;
        }
    }
```

A simple change to the *else* rule allows *if statements* to be chained:

```
optElse : ELSE block
        | ELSE ifStatement
        | *empty*
```

Now, the above example could be rewritten as:

```
if (a < b)
    {
    result = a;
    }
else if (b < c)
    {
    result = b;
    }
else
    {
    result = c;
    }
```

Iteration

The simplest iterator is a *while loop*. Again, for simplicity's sake, we will the body of the *while loop* to be a *block*.

```
whileLoop : WHILE OPEN_PAREN expression CLOSE_PAREN block
```

Sequence

A sequence of programming language statements is known as a *block*. Suppose we wish to enclose a block with braces, as is done in C-like languages. The *block* rule might look something like:

```
block : OPEN_BRACE statements CLOSE_BRACE
```

A *statements* rule would look very similar to our *list* rule:

```
statements : statement
           | statement statements
```

Finally, a *statement* could be an expression, an if statement, or a *while loop*, at a minimum. In keeping with a C-like programming language, a *statement* rule might look like:

```
statement : expression SEMICOLON
          | ifStatement
          | whileLoop
```

These rules allow code that looks like:

```
{
f(x);
if (a < b) { min = a; } else { min = b; }
while (x >= 0)
    {
    if (x > 0)
        {
        t = t + 1;
        }
    x = x - 2;
    }
}
```

You can see that with a few rules, we can have quite complicated code!

Definitions

The final major piece in our grammar is *definitions*. Programming languages often have two kinds of definitions, variable definitions and function definitions. A variable definition in a dynamically-typed, C-like language might look like:

```
varDef : VAR VARIABLE ASSIGN expression SEMICOLON
```

where the keyword `VAR` introduces a variable definition. In a statically-typed language, keywords representing type, such as `TYPE_INT` or `TYPE_STRING`, would substitute for `VAR`. This rule would allow definitions like:

```
var a = 0;
var b = f(x) * (g(y) + z);
```

The initializer could be made optional with a rule rewrite:

```
varDef : VAR VARIABLE ASSIGN expression SEMICOLON
             | VAR VARIABLE SEMICOLON
```

An equivalent rewrite would be:

```
varDef : VAR VARIABLE optInit SEMICOLON

optInit : ASSIGN expression
        | *empty*
```

Function definitions are straightforward:

```
functionDef : FUNCTION VARIABLE OPEN_PAREN parameterList CLOSE_PAREN block
```

where `FUNCTION` is a keyword introducing a function definition and *parameterList* is a list of variables. Why is *parameterList* not a list of expressions?

Finishing out a grammar

As stated earlier, non-terminals describe collections that can be broken down further. When writing a grammar, make heavy use (and re-use) of non-terminals. When you use a non-terminal in a rule, you must define it if you haven't already done so. When you have written a rule for your last non-terminal, you have finished your grammar.