BNF Processing in Prolog
Language Recognition and Parsing

Introduction

BNF provides a means to formally describe programming languages, communications protocols, and a variety of other aspects of computing that are sequence oriented. The purpose of these descriptions is to serve as a precise basis for the understanding and use of a notation. In this article, we provide a brief introduction to how program language processing is accomplished.

Program language processing is normally divided into four phases: lexical analysis, parsing, compilation, and execution.

- Lexical analysis consists of processing a string of characters and assembling them into a list of "tokens" — basic units such as identifiers — that comprise the program text of interest. This process depends on language conventions about white space and comments that are removed in this phase.
- Parsing is the process of recognizing a valid program and transforming it into the hierarchical structure of its components — the derivation tree. This phase normally operates on the result of the lexical analysis.
- Compiling is the process of transforming the derivation tree into executable code, either for actual computer hardware or for a virtual machine.
- The final phase is the execution of the generated code.

For this discussion, we will treat only simple BNF that includes no white space or comment conventions. Therefore, there will be no discussion of lexical processing, and the parser will act directly on character strings rather than on "tokens".

Language Recognition

Language recognition and parsing have much in common (in fact almost everything). We will begin by examining the recognition problem, and subsequently add the small changes needed to perform parsing. BNF defines a language, and in fact, we can use one BNF to determine several languages by selecting various of the non-terminals as a start symbol. We utilize that perspective in the method we develop. When we write BNF in this discussion, we will use upper case to denote non-terminals, and lower-case to denote terminals. If we are considering the language L(A) for a non-terminal A, then each of its productions provides one of the ways to qualify a string in the language. Suppose one of the productions is A → aAbA. Then this production qualifies a string s into L(A) provided s = axby where x and y are substrings also both in L(A). Given an arbitrary candidate string, it can be tested by seeing if it can be decomposed into axby where x and y are also in L(A). in Prolog, the append utility, together with recursion, is perfectly suited to this task. If we have an (unknown) candidate string Ws, and if the goal append([a|Xs], [b|Ys], Ws) succeeds, then we will have one possible solution for Xs = x, and Ys = y. This leaves only the tests on Xs and Ys. Hence a suitable clause for this production is

```
langA(Ws) :- append([a|Xs], [b|Ys], Ws), langA(Xs), langA(Ys).
```

The fact that Prolog's append processing will search for all possible ways of decomposing a string Ws (there may be numerous points where a potential interior 'b' may key the decomposition) ensures that this process will reliably determine if the given production can apply.

The example in the previous paragraph illustrates a process that is sufficiently general to be adapted to any production. There is always one predicate for each non-terminal. And for each production for a non-terminal, there is exactly one corresponding clause. If the production is terminating, the clause is a fact, and if the production is non-terminating, the clause is conditional. For conditional clauses, the body of the clause is based on the right-hand side of the production. The body of a conditional clause uses the 'append' predicate to split a candidate string into pieces that correspond to the components of the right-hand side of the production. Since 'append' produces only two pieces, we may need to use 'append' repeatedly if there are more than two components. For instance, for A → BCD, a

candidate string Ws must be split into 3 pieces, Bs, Cs, and Ds. This can be done using append twice,

append(Bs,Xs,Ws), append(Cs,Ds,Xs).

The first append goal generates a first component, Bs, and then the second and third components, Cs and Ds, are extracted by splitting the other component, Xs, of the first goal. This process can be repeated if there are more pieces to be extracted. Then the corresponding pieces must be tested to see if they match as required, e.g., langB(Bs), etc. Since append will succeed for every possible way of splitting a candidate string into pieces, this provides an exhaustive search for an appropriate structure in a candidate string.

We illustrate the application of this process to the BNF appearing in Figure 1. This grammar defines the language consisting of *all* sequences of 'a' and 'b' (regardless of character order), where the number of 'a's is equal to the number of 'b's (why?).

```
A → ε
A → aAbA
A → bAaA
```

Figure 1.

Since there is only one non-terminal, there is just one predicate we will call 'langA'. There are three productions for A, so there are three clauses for langA. Each clause is modeled as the right side of a corresponding production. For the terminating production A → ε, we have the following fact: langA(""). That is, according to this production, the null string (i.e., empty list) should be recognized (i.e., succeed). For the production A → aAbA, the candidate string must be split into pieces using append, and the pieces tested. The clause is

langA(Xs) :- append([97|A1], [98|A2], Xs), langA(A1), langA(A2).

Since the parameter Xs will be a Prolog string, the lists will contain ASCII codes. Therefore we used the ASCII codes for 'a' and 'b' to identify the break points to split Xs (the 'name' predicate could be inserted to have the system supply these codes). An entirely similar analysis is used for the third production. The completed predicate definition appears in Figure 2.

```
langA("").
langA(Xs) :- append([97|A1], [98|A2], Xs), langA(A1), langA(A2).
langA(Xs) :- append([98|A1], [97|A2], Xs), langA(A1), langA(A2).
```

Figure 2.

Queries using 'langA' will then succeed precisely when the argument belongs to L(A). And, in fact, if the argument is a variable, elements of the language can be generated one after another!

Parsing

The analysis we have performed for language recognition requires only a small additional step to accomplish parsing — that is, to not only decide if a string is described by the BNF, but if it is, to construct its derivation tree. To accomplish this additional processing, we must provide a place for the derivation tree to be retained. This will be a second parameter for the predicate. Also, to reflect the extended role of the resulting predicate we will change its name (e.g., instead of 'langA', 'parseA'). Other than adding suitable parameters in the recursive calls, the bodies of the clauses of this new predicate just duplicate those of the recognition predicates.

The derivation trees are represented by Prolog terms. Prolog terms provide a very direct representation of trees. The top level operator names the root node of the tree, the number of child nodes is the number of arguments, and each argument term is just the representation of the corresponding subtree. So for example, the term

p2(a, p1, b, p3(b, p1, a, p1)) is the term form of the tree in Figure 3.

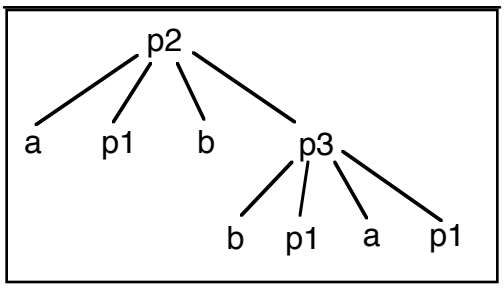

Figure 3.

For purposes of describing derivation trees, we attach a name (or label) to each of the productions of the BNF. These names are arbitrary, but in a programming language context, the names are chosen to be suggestive of their role constructing the language. We continue the example of the previous section, and add labels to the productions as shown in Figure 4. Using these production labels, Figure 3 depicts a derivation tree for the string "abba".

```
p1: A → ε
p2: A → aAbA
p3: A → bAaA
```

Figure 4.

Then in each of the language recognition clauses, the corresponding production name is recorded in the head of the clause, and the calls in the body complete the evaluation of the subtrees as they are used to recognize the candidate string. Thus a parsing predicate is given in Figure 5. It is directly based on the predicate in Figure 2

```
parseA("", p1).
parseA(Xs, p2(a,T1,b,T2)) :- append([97|A1], [98|A2], Xs),
                             parseA(A1,T1), parseA(A2,T2).
parseA(Xs, p3(b,T1,a,T2)) :- append([98|A1], [97|A2], Xs),
                             parseA(A1,T1), parseA(A2,T2).
```

Figure 5.

For instance, in the second clause, the head of parseA records that this step is an application of production p2 and involves the location of characters 'a' and 'b' as indicated, plus subtrees to be constructed by parsing the corresponding pieces of the candidate string Xs as constructed by the append goal. Subsequent parsing failure will cause backtracking to 'append' for another possible decomposition (if any) of string Xs.

In the example presented here, we have included the terminal symbols (a, b) in the derivation trees. In programming language processing, these symbols are needed to construct the derivation tree, but need not be retained after it is constructed. For instance, to process an if-statement,

if (<Boolean>) {<then-part>} **else** {<else-part>}

we need information about <Boolean>, <then-part>, and <else-part>, but once these parts are extracted, it is of no help to retain the '**if**', '**else**' and other punctuation characters. Therefore, in practical programming language processing, the terminal characters are omitted from the derivation trees — these trees are referred to as *abstract syntax trees* since they omit some available information (since it is subsequently of no use).

The above code is in our class directory. Try it out with a variety of queries. Also, other examples illustrating this methodology are included in our class directory.