Syntactic Analysis, Recursive Descent Parsing

Syntactic Analysis/Parsing

We previously looked at the scanner, the first phase of a compiler, we will now look at the parser, the second phase of a compiler. It is significantly more complex than the scanner. It assembles the tokens into constructs (e.g. statements, expressions, program, blocks, etc.). The language of constructs is generally context-free and can be represented by a context-free grammar using either BNF or EBNF. The result produced by the parser is, as we saw earlier, typically a parse tree or some equivalent. Parsing typically reconstructs a canonical (i.e. leftmost or rightmost) derivation.

Parsing comes in two "flavors:"

- *top-down*: builds the parse tree from the root down to the leaves. It "looks for" a particular token or a particular construct. It either involves **backtracking** (i.e. investigate all possibilities) or is **predictive** (at each step only one option is possible). Note that predictive parsing requires that the grammar exhibits certain characteristics as we will soon see. Backtracking parsers are not practical as they are too time consuming. The predictive parsers have the property that, at each step, they are able to predict what the next construct in the input string ought to be, based on one or more lookahead tokens. The main parsing technique in that category is **recursive-descent**. The advantage of recursive descent is that it is suitable for hand coding as it follows exactly the grammar. We will see later that to allow for a recursive-descent parser the grammar must satisfy certain constraints.
- *bottom-up*: builds the parse tree from the leaves to the root. The parser looks for substrings that can be "reduced" to a construct. Such parsers are often called shift-reduce parsers since they alternate between shifting tokens from the input to a stack and reducing substrings found on the top of the stack to non-terminals that are then pushed onto the stack. These parsers are not suitable for hand-coding and are created using parser generators (also called compiler-compilers). Those generators are, in part, distinguished by the language of the code they produce. Some examples of such parser generators:

- YACC, although it was preceded by some other system it is the most notable original such tool. It supports C and is part of the Unix development system.
- ANTLR supports generating code in the following languages: C, C++, Java, Python, C#, Objective-C.
- Coco/R has versions for most modern languages (Java, C#, C++, Pascal, Modula-2, Modula-3, Delphi, VB.NET, Python, Ruby and others).
- GNU bison is a parser generator that is part of the GNU project. Bison converts a grammar description for a context-free grammar into a C or C++ program which can parse a sequence of tokens that conforms to that grammar.

All bottom-up parsing is based on **LR (k)** parsing and is table driven. The first L stands for Left to right analysis of the token stream, the R stands for rightmost derivation which is what this type of parser reconstructs. The k denotes the number of token "lookahead" used to make a decision on the appropriate rule to use. In practice k=1 as other values lead to very large parsers.

Recursive-Descent Parsing

A recursive-descent parser consists of a collection of functions each corresponding to a single construct and based on the right hand side of the productions (or on the syntax diagrams). Those right-hand sides are interpreted in the following way:

- tokens are matched directly against the input.
- non-terminals are translated into a call to the procedure associated with the non-terminal.

Assume that we have a variable token that holds the "current" token (could be a variable inside a Scanner Object in which case we would need a method giveToken ( ) to return the token), a procedure nextToken ( ) that loads that variable with the next token. In addition assume a procedure "match" which will compare the input to a desired token. For example a pseudo code for match may be:

```
void match ( tokenType expected )
{    if ( currentToken == expected )
         nextToken ()
     else
         error ( );    }
```

Let us consider the EBNF grammar:

```
<expression>   ::=   <term> { + <term>}
<term>         ::=   <factor> { * <factor>}
<factor>       ::=   identifier | constant
```

We could write the following parser:

```
function expression
{   term ();
    while (token == +)
    {   match (+);
        term ( )  }
}
function term
{   factor ( );
    while (token == *)
    {   match (*);
        factor ( )}
}
```

```
function factor

{   if (token == identifier)
        match (identifier)
    else   if (token == constant)
        match (constant)
    else
        error ( );
}
```

This works well because at each step we know exactly what to do. This is not always true with a given grammar. We will see later that, in general we can ensure that this is true by making sure the grammar has certain properties (see FIRST & FOLLOW below).

In addition, some grammars may give problems with recursive descent. For example:

```
<expr>     ::= <expr> + <term> | <term>
```

This rule is left recursive and will not work with a recursive descent parser (do you see why?). Although the problem could be solved by replacing the left recursion by right recursion as in

```
<expr>     ::= <term>+ <expr> | <term>
```

this changes the associativity of addition turning it from left associative to right associative. In this particular case the use of EBNF alleviates that problem but this may not be the case in other situations

.

First & Follow

As mentioned earlier it is important to be able to decide unambiguously what to do at each step. The idea of FIRST and FOLLOW will help us to do just that.

FIRST

Given a grammar G and a string α, FIRST (α) is defined as the *set of all terminals which can start a string derived from α*. If α is λ, or can generate λ then λ is also in First (α). In other words, if T is the set of terminals:

$$\text{FIRST}(\alpha) = \{ a \in T \mid \alpha \overset{*}{\Rightarrow} a\beta \} \cup \{ \lambda \text{ if } \alpha \overset{*}{\Rightarrow} \lambda \}$$

For example:

FIRST (**identifier**)	=	{ **identifier** }
FIRST (**^ id**)	=	{ **^** }
FIRST (**array [** <simple> **] of** <type>)	=	{ **array** }
FIRST (**(** <exp> **)**)	=	{ **(** }

Given the grammar

<factor> ::= **(** <expr> **)** | <number>
<number> ::= <digit> { <digit> }
<digit> ::= **0** | **1** | **2** | . . . | **9**

FIRST (<digit>)	=	{**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**}
FIRST (<number>)	=	FIRST (<digit>) = {**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**}
FIRST (<factor>)	=	FIRST (**(** <exp> **)**) ∪ FIRST (<number>) = { **(**, **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**}

Algorithm to construct FIRST (X)

1. If X is a terminal, then FIRST (X) = { X }
2. If X ::= λ is a production then add λ to FIRST (X)
3. If X is a nonterminal and X ::= Y_1Y_2 ... Y_k is a production, then place b in FIRST (X) if for all b in FIRST (Y_1), in addition if λ is in FIRST (Y_1), then place d in FIRST (X) for all d in FIRST(Y_2), etc., finally if λ is in all the FIRST (Y_j) then add λ to FIRST (X)

What this means is that everything in FIRST (Y_1), except λ, is certainly in FIRST (X). That is the end of the story if λ is not in FIRST (Y_1). However, if λ is in FIRST (Y_1) it means Y_1 may disappear and that uncovers Y_2 and the elements of FIRST (Y_2), except λ, must be added. This repeats itself until Y_k if necessary. If all the Y_i can disappear, then λ must be included in FIRST (X).
To compute the FIRST of any string X_1X_2 ... X_n we proceed in a similar way as follows:

Add to FIRST (X_1X_2 ... X_n) all the non-λ symbols of FIRST(X_1). If λ is in FIRST(X_1) then the non-λ symbols of FIRST (X_2) are also added, and so on. λ is added if all the FIRST (X_j) contain λ.

FOLLOW

Given a grammar G and a non-terminal A, FOLLOW (A) is defined as the set of all terminals which can appear immediately to the right of A in some string derived from S (those are called sentential form that can include terminals and non-terminals). In other words

$$\text{FOLLOW (A)} = \{ b \in T \mid S \overset{*}{\Rightarrow} \alpha A b \beta \text{ for some } \alpha \text{ and } \beta \} \cup \{ \$ \text{ if A can be the last symbol in a derivation}\}$$

The FOLLOW is useful in case we have a rule A ::= λ, in which case the symbols which can follow A in a sentential form become possible input symbols when that rule is applied.

End-of token-stream marker

It is the role of the parser to figure out the end of the token-stream. To be consistent with what the parser normally does (reads tokens) it uses a special token (here we will use “$”). This token is placed at the end of the token-stream, not by the programmer but by the scanner. It does not appear anywhere else in the stream.

Algorithm to construct FOLLOW (A):

1. Place $ in FOLLOW (S), where S is the start symbol and $ is the token-stream end marker.
2. If there is a production B ::= α A β, then everything in FIRST (β), except λ, is placed in FOLLOW (A).
3. If there is a production B ::= α A, or a production B ::= α Aβ where FIRST (β) contains λ (i.e. $\beta \overset{*}{\Rightarrow} \lambda$), then everything in FOLLOW (B) is in FOLLOW (A).

Based on the FIRSTs and FOLLOWs one can build a table which shows which production should be chosen for a specific non-terminal, given the current input token. We will call this table M(N, T). An entry in such a table might look like:

Nonterminal	**Input Symbol**		
	a	**b**	**c**
E			

Going back to the grammar

<factor> ::= “**(**“ <expr> “**)**” | <number>
<number> ::= <digit> { <digit> }
<digit> ::= **0** | **1** | **2** | . . . | **9**

FOLLOW (<factor>) = { **$** }
FOLLOW (<number>) = FOLLOW (<factor>) = { **$** }
FOLLOW (<digit>) = FIRST (<digit>) ∪ FOLLOW (<number>) = {**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9, $**}

Another example…

Consider the following grammar:

E	➔	TE’	T’	➔	*FT’ | λ
E’	➔	+TE’ | λ	F	➔	(E) | id
T	➔	FT’			

Construct the FIRST and FOLLOW for E, E’, T, T’ and F

	FIRST	FOLLOW
E	FIRST(T) = {(, id }	{$,) }
E’	{+, λ }	FOLLOW (E) = {$,) }
T	=FIRST(F) = {(, id}	FIRST (E’) ∪ FOLLOW (E) = {+,$,) }
T’	{*, λ}	= FOLLOW (T) = {+,$,) }
F	{(, id}	= FIRST (T’) ∪ FOLLOW (T) = {*, +, $,) }

The Two Rules of Predictive Parsing

1. For every production A ::= $\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_n$, we must have

   FIRST (α_i) ∩ FIRST (α_j) = Ø for each pair i, j, i ≠ j

2. For every nonterminal A such that FIRST (A) contains λ, we must have

   FIRST (A) ∩ FOLLOW (A) = Ø

The extensions associated with the EBNF notation require an adaptation of the formal rules:

- optional part of an RHS

$$A ::= \beta \; [\; \alpha \;] \; \gamma$$

  assuming that γ does not generate λ (i.e. FIRST (γ) does not include λ) we must have:

$$\text{FIRST} (\alpha) \cap \text{FIRST} (\gamma) = \emptyset$$

  If FIRST (γ) contains λ we must also have

$$\text{FIRST} (\alpha) \cap \text{FOLLOW} (\beta) = \emptyset$$

- indefinite repeat

$$A ::= \alpha \; \{\; \beta \;\} \; \gamma$$

  assuming that γ does not generate λ (i.e. FIRST (γ) does not include λ) we must have

$$\text{FIRST} (\beta) \cap \text{FIRST} (\gamma) = \emptyset$$

  If FIRST (γ) contains λ we must also have

$$\text{FIRST} (\beta) \cap \text{FOLLOW} (A) = \emptyset$$