Parsing Notes
2/2011

1 Parsing

Having figured out how to break a program into a list of tokens, we now consider how to build abstract syntax trees representing the program so that it will be easy to process it. Our extended example here will be arithmetic expressions. Later we will parse a very simple programming language called PCF.

```
    <exp> ::= <exp> <addop> <term>
            | <term>
   <term> ::= <term> <mulop> <factor>
            | <factor>
 <factor> ::= ( <exp> )
            | NUM
            | ID
  <addop> ::= + | -
  <mulop> ::= * | /
```

Pure parse trees have too much junk. We use abstract syntax trees instead. In class we looked at parse tree and abstract syntax trees for 2*3+7.

1.1 Recursive descent parsers

Not surprisingly, our parsing program will follow the cfg for the language we are considering. In fact, we will attempt to find the left-most derivation of a term. However, we will find that some grammars work better than other.

The basic idea is to build a recognizing function for each non-terminal of the language and call those to recognize a term. The idea is that each function would return the Thus

```
fun exp input = let
      val inputAfterExp = exp input
      val inputAfterAddop = addOp inputAfterExp
      val rest = term inputAfterAddop
    in
      rest
    end;
```

or alternatively:

```
fun exp input = term(addOp(exp input))
```

There are several problems with this.

1. There are two productions from `<exp>`. How do we decide which to call?
2. The production is “left recursive”. That is, a call of exp immediately makes a call of exp which . . . . Never terminates!

To solve the first problem, we need to find a way of determining which right-hand side to call. Not surprisingly it will depend on what the input token list contains.

It is easy to solve the first if we modify our grammar. In particular, we will write it in EBNF. We can write this instead as:

```
     <exp> ::= <term> {<addop> <term>}*
    <term> ::= <factor> {<mulop> <factor>}*
  <factor> ::= ( <exp> )
             | NUM
             | ID
   <addop> ::= + | -
   <mulop> ::= * | /
```

where {...}* indicates 0 or more occurrences of the material in parentheses.

To get back to BNF, we take the terms in parentheses and give them a name:

```
       <exp> ::= <term> <termTail>                      (1)
  <termTail> ::= <addop> <term> <termTail>              (2)
               | e                                      (3)
      <term> ::= <factor> <factorTail>                  (4)
<factorTail> ::= <mulop> <factor> <factorTail>          (5)
               | e                                      (6)
    <factor> ::= ( <exp> )                              (7)
               | NUM                                    (8)
               | ID                                     (9)
     <addop> ::= + | -                                 (10)
     <mulop> ::= * | /                                 (11)
```

Notice that we no longer have left recursion. We also notice that for those non-terminals that have more than one production, looking at the first token will tell us what production to use. Thus we now have a good grammar.

However, we will do a bit more work before writing the parser. This will be of help with arithmetic expressions (especially in discovering errors), but will be even more helpful with grammars that have more than one production for many non-terminals.

Suppose we are in the middle of a derivation and have so far managed to derive: $\texttt{wX}\alpha$ where `w` contains no non-terminals, so `X` is the first non-terminal still to be expanded (recall that we are performing a left-most derivation). Suppose also that the string we are trying to parse looks like $\texttt{wa}\beta$.

Obviously we would like to apply a production to `X` to perform the next step in the derivation, but which production should be chosen? We look at the right-hand side of the production rules for `X` as follows:

1. If the rhs of a production starts with a terminal symbol, then it must be of the form $\texttt{X} ::= \texttt{a}\gamma$ for it to be a possibility for the next expansion.
2. If the rhs starts with a non-terminal, e.g. $\texttt{X} ::= \texttt{Y}\delta$, then
   (a) Examine the rules for `Y` to see if any of those can derive a string starting with `a`.
   (b) If `Y` can derive ϵ then check to see if δ can derive a string starting with an a.
3. If `X` can derive an ϵ, then we must check to see if the terms following it (e.g., in α) can derive a term starting with `a`.

1.2 First and Follow

The functions FIRST and FOLLOW will help us solve this problem. FIRST is applied to the right-hand sides of production rules to help us resolve the complexities illustrated above in figuring out which production rule to apply. Intuitively, terminal a $\in$ FIRST(X) iff there is a derivation X $\rightarrow^*$ aβ for some β.

1. For any terminal symbol a, FIRST(a) = {a}. Also FIRST(ϵ) = {ϵ}.
2. For any non-terminal A with production rules `A ::=` α_1 `| ...|` α_n,
   FIRST(A) = FIRST(α_1) $\cup \ldots \cup$ FIRST(α_n).
3. For any r.h.s. of the form: $\beta_1 \ldots \beta_n$ (where each β_i is a terminal or a non-terminal) we have:
   (a) FIRST(β_1) is in FIRST($\beta_1 \ldots \beta_n$)
   (b) If $\beta_1 \ldots \beta_{i-1}$ can derive ϵ, then FIRST(β_i) is also in FIRST($\beta_1 \ldots \beta_n$).
   (c) ϵ is in FIRST($\beta_1 \ldots \beta_n$) only if ϵ is in all FIRST(β_i) for all 1 $\leq$ i $\leq$ n.

We can now calculate FIRST for all right-hand sides. FOLLOW is only used if a non-terminal X can derive ϵ (i.e., if $\epsilon \in$ FIRST(X). A terminal a $\in$ FOLLOW(X) iff there is a derivation S $\rightarrow^*$ αXaβ for some α and β.

1. If S is the start symbol, then put EOF into FOLLOW(S) to indicate that the end of file can follow that symbol.
2. For all rules of the form A ::= αXβ, then
   (a) add all elements of FIRST(β) to FOLLOW(X).
   (b) if β can derive ϵ then add all elements of FOLLOW(A) to FOLLOW(X).

We will use FOLLOW in the following circumstances. Suppose X is the current non-terminal, a is the next symbol of the input, and there is a production rule for X which allows it to derive ϵ. Then we can apply that production rule only if a is in FOLLOW(X).

FIRST and FOLLOW help us to determine which rule to apply next. In the case of arithmetic expressions, they aren't really necessary for that purpose, but they will help us to determine relatively early if we have an expression that cannot be derived by the grammar.

1.3 Implementing a Parser

We can build a table that will direct a parser as follows. The rows of the table will correspond to non-terminals, while the columns will correspond to terminals. The entries are productions from our grammar.

Put production X ::= α in entry (X,a) if either

- a $\in$ FIRST(α), or
- $\epsilon \in$ FIRST(α) and a $\in$ FOLLOW(X).

For any non-terminal X and terminal a, the production X ::= α will occur in the corresponding entry if applying this production can eventually lead to a string starting with a.

For this to give us an unambiguous parse, no table entry should contain two productions. (If so, we would have to rewrite the grammar!) Slots with no entries correspond to errors in the parse (e.g., that the string is not in the language generated by the grammar).

We can also write this restriction out as the following two laws for predictive parsing:

1. If `A ::=` α_1 `| ...|` α_n then for all i $\neq$ j, First(α_i) $\cap$ First(α_j) = $\emptyset$.
2. If `X` $\rightarrow^* \epsilon$, then First(`X`) $\cap$ Follow(`X`) = $\emptyset$.

1.3.1 Parsing arithmetic

Recall our final grammar for arithmetic above. We can calculate FIRST and FOLLOW for the grammar as follows:

$$\begin{aligned}
\text{FIRST}(\texttt{<exp>}) &= \{\ (,\ \texttt{NUM},\ \texttt{ID}\ \} \\
\text{FIRST}(\texttt{<termTail>}) &= \{\ +,\ \text{-},\ \epsilon\ \} \\
\text{FIRST}(\texttt{<term>}) &= \{\ (,\ \texttt{NUM},\ \texttt{ID}\ \} \\
\text{FIRST}(\texttt{<factorTail>}) &= \{\ *,\ /,\ \epsilon\ \} \\
\text{FIRST}(\texttt{<factor>}) &= \{\ (,\ \texttt{NUM},\ \texttt{ID}\ \} \\
\text{FIRST}(\texttt{<addop>}) &= \{\ +,\ \text{-}\ \} \\
\text{FIRST}(\texttt{<mulop>}) &= \{\ *,\ /\ \}
\end{aligned}$$

We don't need to do longer strings as no prefix of a right hand side goes to ϵ.

$$\begin{aligned}
\text{FOLLOW}(\texttt{<exp>}) &= \{\ \text{EOF},\ )\ \} \\
\text{FOLLOW}(\texttt{<termTail>}) &= \text{FOLLOW}(\texttt{<exp>}) = \{\ \text{EOF},\ )\ \} \\
\text{FOLLOW}(\texttt{<term>}) &= \text{FIRST}(\texttt{<termTail>}) \cup \text{FOLLOW}(\texttt{<exp>}) \cup \text{FOLLOW}(\texttt{<termTail>}) \\
&= \{\ +,\ \text{-},\ \text{EOF},\ )\ \} \\
\text{FOLLOW}(\texttt{<factorTail>}) &= \{\ +,\ \text{-},\ \text{EOF},\ )\ \} \\
\text{FOLLOW}(\texttt{<factor>}) &= \{\ *,\ /,\ +,\ \text{-},\ \text{EOF}\ \} \\
\text{FOLLOW}(\texttt{<addop>}) &= \{\ (,\ \texttt{NUM},\ \texttt{ID}\ \} \\
\text{FOLLOW}(\texttt{<mulop>}) &= \{\ (,\ \texttt{NUM},\ \texttt{ID}\ \}
\end{aligned}$$

Here is the final table we get from following the rules above. The table entries are production numbers from our grammar

Non-terminals	ID	NUM	"+" or "-"	"*" or "/"	"("	")"	EOF
`<exp>`	1	1			1		
`<termTail>`			2			3	3
`<term>`	4	4			4		
`<factorTail>`			6	5		6	6
`<factor>`	9	8			7		
`<addop>`			10				
`<mulop>`			11				

1.3.2 Recursive descent

Suppose we wish to parse 2*33-(7-2). `<exp>` is start expression, while the lexical analysis results in `[Num 2,*,Num 33,-,(,Num 7,-,Num 2,)]`.

To start, look up `<exp>,Num 2` in table. It tells us to apply production 1, `<exp> ::= <term> <termtail>`.

Next look up `<term>,Num 2` and we see we neet to apply production 4 next: `<term> ::= <factor><factorTail>`. So far we now have:

```
<exp> ::= <term> <termtail>
      =>  <factor><factorTail>
```

If we continue our leftmost derivation, we see the `<factor>,Num 2)` entry indicates production 8, `<factor> ::= NUM`, providing us with

```
      =>  NUM<factorTail>
```

Because NUM matches NUM 2, we move onto the next token, and look up <factorTail>, *, finding production 5. We continue on in this way, looking up productions in the table, throwing away terminals when we find a match. This gives us the following complete derivation:

```
    <exp> ::= <term> <termtail>
          =>  <factor><factorTail> <termtail>
          =>  NUM 2<factorTail> <termtail>
          =>  NUM 2 <mulop> <factor> <factorTail> <termtail>
          =>  Num 2 * <factor> <factorTail> <termtail>
          =>  Num 2 * Num 33 <factorTail> <termtail>
          =>  Num 2 * Num 33 <termtail>
          =>  Num 2 * Num 33 <addop> <term> <termtail>
          =>  Num 2 * Num 33 - <term> <termtail>
          =>  Num 2 * Num 33 - <factor> <factorTail> <termtail>
          =>  Num 2 * Num 33 - ( <exp> ) <factorTail> <termtail>
          =>  ...
```

Finish the derivation on your own by using the table to determine which production to apply next.