ECE251: Programming Languages & Translators Fall 2010

Lecture 10 — October 5, 2010

Patrick Lam *version 1*

Parsing an Expression. Let's see how we actually construct the parse tree for the expression from the previous set of notes.

1. Since we are trying to parse an expression, call `expression`. We don't have a `+` or a `-`, so `expression` calls `term`, which must come next according to the grammar.
2. Enter `term`. The first symbol in a `term` is a `factor`, so it calls `factor`.
3. Enter `factor`. A factor might either be a parenthesized expression, which starts with `(`, or a literal. Fortunately, we can cleanly distinguish between literals and parentheses.

   In this case, we have a literal 5, so we accept this literal as part of the parse tree, and return to the caller, `term`.
4. `term` checks for a `*` or `/` operator, but we have a `+` operator, so `term` returns to its caller, `expression`.
5. The `expression` next looks for a `+` or `-` symbol. It finds a `+`, so it accepts this symbol and calls `term` for the second operand.
6. Once again, enter `term`, which calls `factor` and accepts the literal 2. It then returns to `term`, which accepts the `/`, calls `factor` and accepts the other literal, `48`. It then returns all the way to `expression`.
7. Back at `expression`, we recognize the `-` and the other term, which is the factor `93`.

[It is hard to write down how I actually build the tree. You'll have to see that during lecture.]

Building Top-Down Parsers

Section 2.3 of the textbook explains how to build "recursive-descent" and "table-driven" top-down parsers. However, it is hard to track down why you'd want to build a table-driven parser. Table-driven parsers seem to have a maintenance advantage. But that's not useful, because most of the time, you will use a parser generator (a domain-specific language!) anyway.

The code above was for a *recursive descent* parser. It is easy to build recursive descent parsers, if the input language is suitable. We will see how to manually build so-called LL(1) recursive descent parsers, which will also show you the conditions under which it is possible to build these parsers. ANTLR supports a generalization of this class of languages.

The basic scheme (as we've seen) is to create a procedure per production in the input language, accepting terminals and making appropriate calls to the nonterminals within that production.

The hard part of building a recursive descent parser is figuring out what tokens to look for:

```
void factor(void) {
    if (accept(lit)) { // why lit?
    } else if (accept(lparen)) { // why lparen?
    // etc
```

At a production, we need to *predict* what the next production is going to be, given the current production (which procedure we are currently in) and look-ahead to the next token (determined by the return value from `accept()`). To do so, we'll construct sets called FIRST and FOLLOW. We will usually use FIRST to decide what to do next, unless there is a ε, in which case we'll use FIRST and FOLLOW together.

For a nonterminal A,

- FIRST(A) is the set of all terminals that could start an A; and
- FOLLOW(A) is the set of all terminals that could come after an A in a valid program.

In our example, we have three nonterminals, E, T and F. The FIRST sets are as follows.

- An E must start with either a `+`, a `-`, a `(` or a `LIT`.
- A T must start with either a `(` or a `LIT`.
- A F must start with either a `(` or a `LIT`.

We don't need FOLLOW for this example. We only need it for grammars that contain εs. However, just so you know, the FOLLOW sets are:

- FOLLOW(E) = {EOF, `')'`};
- FOLLOW(T) = {`'+'`, `'-'`};
- FOLLOW(F) = {`'*'`, `'/'`};

We'll talk about how to construct these sets next time.