CS 340 - Lecture 9: Recursive-Descent Parsing

Example recursive descent parser: recursivedescent.zip

Parsing

Recall that *parsing* is the problem of taking a string of terminal symbols and finding a derivation for that string of symbols in a context-free grammar. Parsing is critical task in implementing an interpreter or compiler for a programming language, since the syntax of a program contains essential information about the *meaning* of the program.

A *parser* is the module of an interpreter or compiler which performs parsing. It takes a sequence of *tokens* from the lexical analyzer (you know how to write one of those!), finds a derivation for the sequence of tokens, and builds a *parse tree* (also known as a *syntax* tree) representing the derivation. We have seen that parse trees are very important in figuring out the meaning of a program (or part of a program).

Recursive Descent

Recursive descent is a simple parsing algorithm that is very easy to implement. It is a *top-down* parsing algorithm because it builds the parse tree from the top (the start symbol) down.

The main limitation of recursive descent parsing (and all top-down parsing algorithms in general) is that they only work on grammars with certain properties. For example, if a grammar contains any *left recursion*, recursive descent parsing doesn't work.

Eliminating Left Recursion

Here's our simple expression grammar we discussed earlier:

```
S → E
E → T | E + T | E - T
T → F | T * F | T / F
F → a | b | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

[S, E, T, and F are nonterminal symbols, and **a**, **b**, and the digits 0-9 are terminal symbols.]

Unfortunately, this grammar is not suitable for parsing by recursive descent, because it uses left recursion. For example, consider the production

```
E → E + T
```

This production is left recursive because the nonterminal on the left hand side of the production, **E**, is the first symbol on the right hand side of the production.

To adapt this grammar to use with a recursive descent parser, we need to eliminate the left recursion. There is a simple technique for eliminating immediate instances of left recursion. [This technique won't handle indirect instances of left recursion.]

Given an occurrence of left-recursion:

```
A → A α
A → β
```

Note that some non-recursive production of the form A → β must exist; otherwise, we could never eliminate occurrences of the nonterminal A from our working string when constructing a derivation.

We can rewrite the rules to eliminate the left recursion as follows:

```
A → β A'
A' → α A'
A' → ε
```

So, for example,

```
E → E + T
E → T
```

becomes

```
E → T E'
E' → + T E'
E' → ε
```

Here's the entire expression grammar, with left-recursion eliminated.

```
E → T E'
E' → + T E'
E' → - T E'
E' → ε
T → F T'
T' → * F T'
T' → / F T'
T' → ε
F → a | b | 0 | 1 | 2 | ... | 9
```

Left-Factoring

Another property required of a grammar to be suitable for top-down parsing is that the grammar is *left-factored*. A left-factored grammar is one where for each

nonterminal, there are no two productions on that nonterminal which have a common nonempty prefix of symbols on the right-hand side of the production.

For example, here is a grammar that is not left-factored:

```
A → a b c
A → a b d
```

Both productions share the common prefix **a b** on the right hand side of the production.

We can left-factor the grammar by making a new nonterminal to represent the alternatives for the symbols following the common prefix:

```
A → a b A'
A' → c
A' → d
```

Our non-left-recursive expression grammar is already left-factored, so we don't need to change it.

Recursive Descent Parsing

Once you have a non-left-recursive, left-factored grammar, recursive descent parsing is extremely easy to implement.

Each nonterminal symbol has a parsing function. The purpose of the parsing function for a nonterminal is to consume a string of terminal symbols that are "generated" by an occurrence of that nonterminal.

Terminal symbols are consumed either by directly reading them from the lexer, or by calling the parse methods of another nonterminal (or nonterminals). In general, the behavior of the parse method for a particular nonterminal is governed by what string of symbols (nonterminal and terminal) is legal for the right-hand side of productions on the nonterminal.

Typically, any parser will construct a parse tree as a side-effect of parsing a string of input symbols. The nodes of a parse tree represent the nonterminal and nonterminal symbols generated in the derivation of the input string. So, we can have the parse method for each nonterminal return a reference to a parse tree node for that particular nonterminal symbol.

Example

Here's what a parse method for the **E** nonterminal in our revised expression grammar might look like:

```
public Symbol parseE() throws IOException {
        NonterminalSymbol e = new NonterminalSymbol("E");

        e.addChild(parseT());
        e.addChild(parseEPrime());
```

```
        return e;
}
```

The **parseE** method internally calls **parseT** and **parseEPrime** methods, because the only production on **E** is

```
E → T E'
```

The **parseT** method is similar.

The **parseEPrime** method is more interesting. There are three productions on **E'**:

```
E' → ε
E' → + T E'
E' → - T E'
```

When **parseEPrime** is called, we should ask the lexical analyzer if we've reached the end of the input string. If so, then the epsilon production must be applied.

If the lexer is not at end-of-input, we should ask the lexical analyzer for a single terminal symbol. If we see a symbol other than **+** or **-**, then we'll again assume that the epsilon production should be applied. Otherwise, we'll add the terminal symbol to the parse node we're creating for the **E'** symbol, and continue by parsing the **T** and **E'** nonterminal symbols by calling their parse methods:

```
private Symbol parseEPrime() throws IOException {
        NonterminalSymbol ePrime = new NonterminalSymbol("E'");

        TerminalSymbol op = lexer.peek(); // look ahead
        if (op == null) {
                // end of input: epsilon production is applied
                System.out.println("end of input");
        } else {
                if (!op.getValue().equals("+") && !op.getValue().equals("-")) {
                        // we saw a terminal symbol other than + or -:
                        // apply epsilon production
                } else {
                        // consume the operator
                        lexer.get();

                        // saw + or -
                        // production is
                        //    E' -> + T E'
                        // or
                        //    E' -> - T E'
                        ePrime.addChild(op);
                        ePrime.addChild(parseT());
                        ePrime.addChild(parseEPrime());
                }
        }

        return ePrime;
}
```

Note a few interesting things going on in **parseEPrime**:

- The lexical analyzer is the **lexer** object. We're using two of its methods: **peek**, which asks for the next token without consuming it, and **get**, which asks for

and consumes the next token. Both methods return a **TerminalSymbol** object. **peek** returns the **null** value when the end of input is reached.

- Applying the epsilon production means we don't add any child symbols to the parse node being created.
- The **parseEPrime** method can call itself recursively, because the

```
E' → + T E'
E' → - T E'
```

productions contain the symbol **E'** on the right hand side. That's why it's called recursive descent!

To use a recursive descent parser to parse an entire input string, simply call the parse method for the grammar's start symbol. It will return a reference to the root node of the parse tree.
