Recursive Descent Parsing

Recursive descent is the easiest, most intuitive parsing technique. It isn't as general as some of the table-driven techniques and for some commonly-used constructions it requires rewriting the grammar, but these are relatively easy problems to get around. This is what I recommend that you use for your BPL parser.

Here is the idea. We write a recursive procedure corresponding to each of the grammar symbols. The role of that procedure is to consume the tokens for the phrase of terminal symbols represented by the grammar symbol, and to build the corresponding portion of the parse tree.

For example, consider the grammar

S ::= aAB | bB
A ::= aA | b
B ::= b

Here the start symbol is S; the only terminal symbols are a and b. An A rule generates a list of 0 or more a's followed by a single b. So this grammar will generate lists of a's followed by 2 b's.

On the next pages is an outline of a recursive descent parser coded in Java. To keep it simple this doesn't build trees; it just recognizes strings in the language generated by this grammar.

I am assuming we have 3 tokens: T_a, T_b, and T_EOF; our scanner has function getToken() and global variable token.

```
void parse( ) {
        getToken();
        S();
        if (token === T_EOF)
                System.out.println( "accept");
        else
                System.out.println( "reject");
}
```

```
void S() {
        if (token == T_a) {
                getToken();
                A();
                B();
        }
        else if (token == T_b) {
                getToken();
                B();
        }
        else
                throw new Exception();

}
```

```
void A() {
        if (token == T_a){
                getToken();
                A();
        }
        else if (token == T_b)
                getToken();
        else
                throw new Exception();
}
```

```
void B() {
        if (token == T-b)
                getToken();
        else
                throw new Exception();
}
```

It is easy enough to make these functions build the parse tree.
Assume we have a top-level TreeNode class and 3 subclasses:

S_Node has a letter ('a' or 'b') and 2 TreeNode links,
which I'll call child1 and child2

A_Node has a letter and 1 link which I'll call child.

B_Node has just a letter

I'll also assume that all of the TreeNodes have a field *kind* that consists of a char value 'S', 'A', or 'B' so we can immediately tell what kind of node a particular value is.

Now each of our recursive functions will return the portion of the parse tree corresponding to the tokens it has consumed.

The parse() function becomes

```
TreeNode parse( ) {
        getToken();
        TreeNode node = S();
        if (token === T_EOF) {
                System.out.println( "accept");
                return node;
        }
        else {
                System.out.println( "reject");
                return null;
        }
}
```

```
S_Node S() {
        if (token == T_a) {
                getToken();
                A_node a = A();
                B_node b = B();
                return new S_Node('a', a, b);
        }
        else if (token == T_b) {
                getToken();
                B_Node b =B();
                return new S_Node( 'b', b, null);
        }
        else
                throw new Exception();

}
```

```
A_Node A() {
        if (token == T_a){
                getToken();
                A_Node a = A();
                return new A_Node( 'a', a);
        }
        else if (token == T_b)
                getToken();
                return new A_Node( 'b', null);
        else
                throw new Exception();
}
```

```
B_Node B() {
        if (token == T_b) {
                getToken();
                return new B_Node('b');
        }
        else
                throw new Exception();
}
```

Here is a function that walks the parse tree and prints the input string:

```
void PrintString( TreeNode t ) {
        if (t.kind == 'S') {
                System.out.print( t.letter );
                PrintString(t.child1);
                if (t.child2 != null)
                        PrintString(t.child2);
        }
        else if (t.kind == 'A') {
                System.out.print( t.letter);
                if (t.child != null)
                        PrintString(t.child);
        }
        else if (t.kind == 'B')
                System.out.print( t.letter );
}
```

Recursive descent is simple. What could possibly go wrong?