Parsing BPL

Writing a *recognizer* for a programming language is pretty easy. We are parsing BPL in order to get parse trees that we can use later in our compiler. The most difficult part of parsing is being sure that you know exactly what sort of tree you need to build in order to be able to do type-checking and code-generation.

Paying attention to 2 properties will help you later:

- You will have lots of different types of nodes which will have different data fields. You need an easy way to tell what kind of node you are at anywhere in the tree.
- It is easy to overlook in the parser things you will need for typechecking. Be sure that types can be found in your tree.

A Game Plan

You have coded enough to know that if you type in a 1000-line program and then try to debug it you will have problems. You need to implement the parser incrementally, but there are a lot of pieces that need to fit together. Remember that your parser doesn't recognize semantic errors (like undeclared variables), so you can parse statements before you handle declarations. You might start by modifying the grammar so the first few rules become

```
PROGRAM -> STATEMENT
STATEMENT -> EXPRESSION_STMT
EXPRESSION_STMT -> EXPRESSION;
EXPRESSION -> <id>
```

This will let you parse "programs" like

```
x;
```

After this is working, extend the statement options to include COMPOUND_STMT, but without the declarations; now you can parse

```
{x; y; z;}
```

Now add a particular kind of statement, like WHILE, so you can parse

```
{while (e) s;}  and  {while (e) {s; t;}}
```

gradually add in the expressions, then the rest of the statements, then variable declarations, then function declarations, and so forth. Try to keep the parser in a working state. Be thorough about checking and debugging each addition.

Each time you add a new rule to the grammar you are parsing, you need to add a new kind of tree node to hold the data for that rule.

For example, the rule for a while loop is

WHILE_STMT -> while (EXPRESSION) STATEMENT

For this you need a tree node that I call a While_Node. It has two children: one that holds the condition (returned by EXPRESSION()) and one that holds the body (returned by STATEMENT()).

You need a way to determine that you are building the tree correctly. As you extend your parser by adding new kinds of treenodes, include print functions for them so that you can walk the tree and print it. I put such a print function in each of my node classes as I create the class. Printing is linear and trees aren't and you don't have extra time to spend on making a pretty picture of your parse tree; just do something that you can follow. If your parse tree is wrong you will never get typechecking to work.

I use a fairly elaborate class system to organize my trees. If you are writing in a non-OO language like C you should think carefully about how to do your typedefs to store all of the information that needs to be in your nodes.

The next set of slides discusses each of the major kinds of treenodes that you will need.

First, *every* treenode I use has at least the following two data items:

- An integer *line_number*. You will want this for error messages and for debugging messages you will use in the typechecking stage, like "Variable x used on line 25 is linked to declaration on line 6." This is problematic because many expressions go on for more than one line. As much as possible I try to put the line number for the start of the expression into this field.
- An integer *kind* that tells me what sort of node I am dealing with. I have 20 different kinds of nodes; the constants for *kind* are defined in my top-level TreeNode class.

I have 3 major types of nodes: Declaration nodes, Statement nodes, and expression nodes.

- A declaration declares a variable, array, or function along with its type, as in
  ```
  int n
  ```
- A statement does something, such as
  ```
  write( x );
  ```
- An expression computes a value, such as
  ```
  x + 1
  ```

A basic declaration node extends the generic TreeNode by adding a *next* pointer that points at the next declaration, because declarations tend to come in series, as in

```
int x;
string y;
```

or the parameter list in

```
int foo( int a, int b)
```

For that matter, a program in BPL is a list of declarations. So all DEC nodes have this *next* field.

I use 3 kinds of declaration nodes: VAR_DEC, FUN_DEC, ARRAY_DEC

A VAR_DEC node handles declarations like

```
int x;
int *y;
```

This has fields for the string that is the name of the variable, and the token that describes its type. You could just as easily use a string for the type name. I handle pointer types (as in int *y) by adding into the VAR_DEC node a boolean that says this is pointer; you can come up with other says to do this if you don't like that.

A FUN_DEC node handles a function declaration. This needs to include:

- A string for the name of the function.
- A token (or string or whatever) for the return type, which can only be int, void, or string.
- A DEC node for the params
- A compound statement node for the body.

I do an ARRAY_DEC node as a subclass of VAR_DEC node. The latter already has the type and name fields; ARRAY_DEC just adds to this an integer (an honest-to-god integer, not an expression node) for the size.

Statements also come in lists, so my top-level statement node class has a *next* pointer. There are lots of different statements nodes (I have distinct EXPRESSION_STATEMENT, IF_STATEMENT, WHILE_STATEMENT, COMPOUND_STATEMENT, RETURN_STATEMENT, READ_STATEMENT, WRITE_STATEMENT, WRITELN_STATEMENT nodes.); each has fields appropriate to the kind of statement it is.

Note that any expression can be treated as a statement; there is a grammar rule

```
        STATEMENT -> EXPRESSION_STATEMENT
        EXPRESSION_STATEMENT -> EXPRESSION; | ;
```

Expressions also come in lists (the arguments for a function call, as in f(3*x, y+1)), so they all have a *next* field. Ok; that means all my nodes have next fields so that could have been part of the TreeNode class. Here are my different kinds of EXP nodes:

- INT_VALUE (holds literal values, like the right side of x + 6)
- STRING_VALUE (literal strings, as in "bob")
- VARIABLE (the left side of x+6)
- OP_EXP (the center of x+6)
- FUN_CALL
- READ_EXP
- ADDRESS_EXP (for &x)
- DEREF_EXP (for *y)

Assignments in BPL are expressions, not statements, which allows you to have

```
        x = y = 5
```

I treat assignments in the parser as OP_NODEs. Note that the grammar rule is right-recursive, so the assignment operator is right-associative, which is what you want. Assignment is the highest node in the expression portion of the grammar, so it has the lowest precedence, so

```
        x = y = z+5
```

will be grouped as

```
        x = (y = (z+5))
```

Exceptions

Every one of parse functions thows a ParseException (that's my own exception class; I also have ScanException, TypeException, etc.)

It is fine to have the top-level parser function called within a try-catch statement where the catch statement prints an error message and halts. You must give some kind of error message that describes the error and the line number on which it occurs.

Some statement forms have required grammar symbols. For example, the rule for while statements is

```
WHILE_STMT -> while ( EXPRESSION ) statement
```

If you are parsing such a statement, the first thing you expect to see is a While-token, then a left parentheses, etc.

You could have code such as

```
if (myScanner.nextToken.kind != T_WHILE)
        throw new ParserException(
                myScanner.currentToken.lineNumber,
                "Bad while statement);
else {
        getNextToken();
        if (myScanner.currentToken.kind != T_LPAREN)
                throw new ParserException(
                        myScanner.nextToken.lineNumber,
                        "missing left parenthesis" );
        else {
                getNextToken();
                ExpressionNode e = Expression();
                ...
        }
}
```

Your code is a bit nicer if you create a procedure

```
            expect( int tokenKind, String message )
```

This procedure compares the kind of the current toke with its argument. If they are different it throws an exception with the current token's line number and the message string. You might (or might not) want procedure Expect to consume the current token if it is of the expected kind.

This changes the code for a while statement to something like

```
WhileNode While_Stmt() throws ParserException {
	expect( Token.T_WHILE, "bad while statement");
	expect( Token.T_LPAREN, "missing left paren in while condition");
	ExpNode e = E();
	expect(Token.T_RPAREN, "missing right parent in while condition");
		etc.
}
```

This is far more readable.

To help you get started, here are a few of my parser functions:

First, the following function parses a list of declarations. The top-level program() function calls this after starting the scanner:

```
public DecNode DeclarationList() throws ParserException{
    DecNode d = Declaration();
    DecNode list = d;
    while (myScanner.nextToken.kind != Token.T_EOF) {
            DecNode d1 = Declaration();
            d.next = d1;
            d = d1;
    }
    return list;
}
```

```
public StatementNode Statement() throws ParserException {
        if (myScanner.nextToken.kind == Token.T_LBRACE)
                return CompoundStatement();
        else if (myScanner.nextToken.kind == Token.T_IF)
                return IfStatement();
        else if (myScanner.nextToken.kind == Token.T_WHILE)
                return WhileStatement();
                .........
        else
                return ExpressionStatement();          }
```

Function declaration( ) is one of the key procedures here. It has many different cases; here is the overall structure:

```
public DecNode Declaration() throws ParserException {
      boolean isPointer = false;                     if
      (!isTypeToken(myScanner.nextToken))
                throw new ParserException( ... );
      Token typeToken = myScanner.nextToken;
      getNextToken();
      if (myScanner.nextToken.kind == Token.T_TIMES) {
                isPointer = true;
                getNextToken();
      }
      expect(Token.T_ID, "Expected an identifier as part of the
                declaration");
      Token id = myScanner.nextToken;
       ......
}
```

```
public ExpNode Factor() throws ParserException
{
        if (myScanner.nextToken.kind == Token.T_LPAREN) {
                getNextToken();
                ExpNode e = Expression();
                expect(Token.T_RPAREN, ...);
                getNextToken();
                return e;
        }
        else if (myScanner.nextToken.kind == Token.T_ID){
                Token id = myScanner.nextToken;
                getNextToken();
                if (myScanner.nextToken.kind == Token.T_LBRACKET ) {
                        getNextToken();
                        ... // get array expression
                else if (myScanner.nextToken.kind == Token.T_LPAREN){
                ... // get functijon call
                else {
                        return new VariableNode(id.string_value, id.lineNumber);
                }
        }
        ......
}
```