COP5621 Programming Project
Part II: Syntax analysis

Purpose:

This project is intended to give you experience in using a parser generator(yacc) and bring together various issues of syntax specification and parsing discussed in class.

Project Summary:

Your task is to write a YACC specification that when input into YACC will produce an LALR(1) parser. The parser will parse the token stream, calling the lexical analyzer for the next token, and produce a syntax tree representation of the PASC program. The syntax tree representation will be used later by the semantic analyzer and code generator.

Syntax Specification:

The syntax of the PASC language is described by a set of syntax diagrams which accompanies this handout. A syntax diagram is a directed graph with one entry and one exit. Each path through the graph defines an allowable sequence of symbols. For example, the structure of a valid PASC program is defined by the first syntax diagram. The occurrence of the name of another diagram, such as variable declaration, function declaration, or compound statement indicates that any sequence of symbols defined by the other diagram may occur at that point.

Your first task in this assignment is to develop a context free grammar for the PASC language from the syntax diagrams with careful consideration to the form of the generated syntax tree for each construct. In particular, construction of the syntax trees will be much easier if a left (right) recursive grammar rule is set up for a construct that has a corresponding left(right) recursive syntax tree.

YACC specification:

Your second task in this assignment is to express your grammar as a YACC specification. You will want to run your specification through YACC to ensure that the grammar produces no parsing conflicts. If conflicts are indicated by YACC, you should alter your grammar to eliminate them without changing the language accepted by your grammar, or ensure that YACC's handling of the conflict agrees with the PASC language specification. Initially, you may want to generate a parser merely prints out ACCEPT for syntactically correct PASC programs and REJECT with error messages for incorrectly structured PASC programs. Your parser will call your lexical analyzer from phase 1 to obtain the next token when needed.

Your third task is to extend your YACC specification to construct a parser that not only accepts a syntactically correct PASC program, but also generates the correct syntax tree for the source program. This is done by adding semantic actions to the YACC specification of each grammar production. These semantic actions will be tree building operations. **Warning:** Debugging is much easier if the addition of semantic actions is done incrementally, a few grammar rules at a time.

Syntax Tree Representation:

A description of the syntax subtrees for each syntactic construct is given in an additional handout. Leaf nodes will contain a *kind* field and a *semantic value* field. The kind field will be one of IDNode, NUMNode, CHARNode, STRINGNode, or DummyNode corresponding to an identifier, integer constant, character constant, string constant, or null node, respectively. The semantic value field will be the value returned in yylval or 0 (in the case of a null node). An interior node will be designated by the kind field being an EXPRnode. The interior node has a NodeOp value that is one of the Ops depicted throughout the syntax tree description and pointers to a left and right child. Thus, NodeOpType = (AddOp, AndOp, RArgOp,...) and NodeKindOP = (IDNode, NumNode,..., ExprNode, Dummynode).

The tree node data structure and its related utility routines are provided in the file proj2.c for your use in building the abstract syntax trees. Adhering strictly to the NodeKind type, NodeOp type, and these tree routines will enable you to exploit a tree checker(checktree) that checks that your generated syntax tree is valid tree and a tree printer (treeprint). You may need to modify the getname() and getstring() routines in proj2.c (check the file for details). You will also need to declare and initialize *FILE *treelst* in your main routine for outputing the tree.

Testing the Parser:

It is next to impossible to debug your parser by entering grammar rules and action statements for the whole PASC language and attempting to debug it all at once. You should enter actions for a few rules and test, then continue to add more rules and test.

The tree printing routine will be quite useful in your debugging. Using the tree printing routine, your syntax tree will be printed out as follows:

(1) Information about nodes is to be printed as follows:

```
NUMNode:      [INT: integer value]
CHARNode:     [CHAR: character symbol]
IDNode:       [ID: token value, lexeme]
STRINGNode:   [STRING: token value, lexeme]
DummyNode:    [dummy]
Other:        [nodeoptype]
```

where nodeoptype is one of the Op's given in the syntax tree description, token value is an integer, and lexeme is a string.

(2) The syntax tree will be printed, starting at the root node. The routine traverses the nodes of the tree in a preorder traversal printing the right subtree before the left subtree. This improves readability of the output during debugging. Each node will be printed on a separate line. The root node will be printed starting in the first column. All nodes at the same level of the tree will be printed with the same indentation. If level i is printed starting in column j, then level $i + 1$ will be printed starting at column $j + 1$.

Error Handling:

Your parser should print appropriate error messages. You do not have to implement any error recovery.

Assignment Submission and Deadline:

You will do a demo on March 7, 2008. Grading policy:

- Recognize correct programs and detect incorrect programs (50).
- Correct abstract syntax tree (40)
- Error reporting (10).