1 Building a Parser

Using bison, you are to write a parser for our language. As a reminder, our language is a subset of Pascal. This subset differs from Pascal in the following ways. Program headings contain no list of input and output files. Blocks have only variable and procedure/function declarations and compound statements. Types are limited to the scalars: integer, character, boolean; and single dimensional arrays of scalars indexed by integers. Only the following statements are included: while, if, procedure call, assignment, and compound. Operators are restricted to arithmetic, logical, and relational.

Here’s the grammar for our language:

\[
\begin{align*}
\text{start} & ::= \text{program} \text{ ID} ; \text{ block} . \\
\text{block} & ::= \text{variables} \text{ procdecls} \text{ cmpdstmt} \\
\text{procdecls} & ::= \text{procdecls} \text{ procdecl} | \langle \text{empty string} \rangle \\
\text{procdecl} & ::= \text{procedure} \text{ ID} \text{ parmlist} ; \text{ block} ; \\
& \quad | \text{function} \text{ ID} \text{ parmlist} : \text{ stype} ; \text{ block} ; \\
\text{parmlist} & ::= ( \text{ parms } ) | \langle \text{empty string} \rangle \\
\text{parms} & ::= \text{ parms} ; \text{ parm} | \langle \text{empty string} \rangle \\
\text{parm} & ::= \text{var} \text{ vardcl} | \langle \text{empty string} \rangle \\
\text{variables} & ::= \text{ var} \text{ vardcls} | \langle \text{empty string} \rangle \\
\text{vardcls} & ::= \text{vardcls} \text{ vardcl} ; | \langle \text{empty string} \rangle \\
\text{vardcl} & ::= \text{ idlist} : \text{ type} \\
\text{idlist} & ::= \text{idlist} , \text{ ID} | \langle \text{empty string} \rangle \\
\text{type} & ::= \text{array} [ \text{ integer_constant} .. \text{ integer_constant} ] \text{ of stype} | \langle \text{empty string} \rangle \\
\text{stype} & ::= \text{integer} | \text{char} | \text{boolean} \\
\text{stmtlist} & ::= \text{stmtlist} ; \text{ stmt} | \langle \text{empty string} \rangle \\
\text{stmt} & ::= \text{ifstmt} | \text{wstmt} | \text{astmt} | \text{procstmt} | \text{cmpdstmt} | \text{writestmt} \\
\text{writestmt} & ::= \text{writeln} ( \text{ exp } ) \\
\text{procstmt} & ::= \text{ID} \text{ optexplist} \\
\text{optexplist} & ::= \langle \text{empty string} \rangle | \langle \text{empty string} \rangle \\
\text{explist} & ::= \text{ explist} , \text{ exp} | \langle \text{empty string} \rangle \\
\text{stmt} & ::= \text{ifstmt} | \text{wstmt} | \text{astmt} | \text{procstmt} | \text{cmpdstmt} | \text{writestmt} \\
\text{writestmt} & ::= \text{writeln} ( \text{ exp } ) \\
\text{procstmt} & ::= \text{ID} \text{ optexplist} \\
\text{optexplist} & ::= \langle \text{empty string} \rangle | \langle \text{empty string} \rangle \\
\text{explist} & ::= \text{ explist} , \text{ exp} | \langle \text{empty string} \rangle 
\end{align*}
\]
\[
\begin{align*}
\text{exp} & ::= \text{exp} + \text{exp} \mid \text{exp} - \text{exp} \mid - \text{exp} \mid \text{exp} \times \text{exp} \mid \text{exp} \div \text{exp} \\
& \quad \mid \text{exp} \not= \text{exp} \mid \text{exp} == \text{exp} \mid \text{exp} >= \text{exp} \mid \text{exp} > \text{exp} \mid \text{exp} < \text{exp} \\
& \quad \mid \text{exp} <= \text{exp} \mid \text{exp} \text{ and } \text{exp} \mid \text{exp} \text{ or } \text{exp} \mid \text{exp} \text{ exor } \text{exp} \\
& \quad \mid \text{not } \text{exp} \mid ( \text{exp} ) \mid \text{ID} ( \text{explist} ) \mid \text{lvalue} \mid \text{constant}
\end{align*}
\]

\[
\begin{align*}
\text{lvalue} & ::= \text{ID} \mid \text{ID} [ \text{exp} ] \\
\text{constant} & ::= \text{integer_constant} \mid \text{char_constant} \mid \text{true} \mid \text{false}
\end{align*}
\]

Your project should produce a listing, identify any syntax errors, and print a list of identifiers (not including reserved words) and operators encountered during the parse. Please see the demo files and their output for the format of this interface. You must detect at least the following syntax errors: illegal procedure declaration, illegal parameter, illegal variable declaration, illegal statement, and illegal expression. You must enforce the following precedence and associativity for each operator:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational operators</td>
<td>lowest</td>
<td>not associative</td>
</tr>
<tr>
<td>+ - or</td>
<td></td>
<td>left associative</td>
</tr>
<tr>
<td>* div and</td>
<td></td>
<td>left associative</td>
</tr>
<tr>
<td>not unary-</td>
<td>highest</td>
<td>right associative</td>
</tr>
</tbody>
</table>

2 What To Do

To get started, you should download project2.tar.gz from the class web. This will provide you with:

- A Makefile,
- A skeleton parse.y file that you should modify,
- A simple symbol table that you can use to keep track of identifiers (symtab.h and symtab.c), and
- Several demo files containing sample programs.

You must use the scanner that you built for phase 1 of the project. Please use TRUE_TOK and FALSE_TOK instead of TRUE and FALSE since the latter are predefined in the C compiler.

Similar to the last phase, you compile and link your parser by typing make. For this project you should only need to change the file parse.y. All questions regarding the project should be posted in our news group ru.nb.dcs.class.415.

Remember, a good source of information are the man pages (e.g., man bison) and the recommended book on lex and yacc.