

CS423 - 01 Project#1
DUE: Wednesday, March 1, 2006, 11:59 PM

1 Objectives

- Building a lexical analyzer/parser with (flex/flex++) and (yacc/bison)
- Implementing a type-checking system.
- Generating symbol tables.

2 Problem Statement

You are to write a program that:

- reads a filename from the command line
- using the grammar that follows, determines whether or not the filename specified contains text that matches the grammar.
- outputs a new file that contains a symbol table for each of the functions in the program found in the specified filename. It should also have a symbol table for the global variables.
- outputs another file that lists the line number (think about using `yylineno` here) of each assignment statement and indicates whether or not there is a type mismatch somewhere in the assignment statement.

3 Grammar

The following grammar specifies the language you are to use; the start rule is *prog* and all other rules are alphabetized for ease in finding them.

<i>prog</i>	→	<i>decl prog</i> <i>decl</i>
<i>base_type</i>	→	REAL INT VOID
<i>compound</i>	→	LBRACE <i>stmtlist</i> RBRACE
<i>constr</i>	→	LSQR NUM_INT RSQR <i>constr</i> POINTER <i>constr</i> ϵ
<i>decl</i>	→	<i>vardecl</i> SEMICOLON <i>funcdecl</i>
<i>expr</i>	→	<i>simplexpr</i>
<i>exprlist</i>	→	<i>expr</i> COMMA <i>exprlist</i> <i>expr</i>
<i>factor</i>	→	ID ID LPAREN <i>exprlist</i> RPAREN ID LSQR <i>exprlist</i> RSQR NUM_REAL NUM_INT LPAREN <i>expr</i> RPAREN
<i>funcdecl</i>	→	<i>type</i> ID LPAREN <i>params</i> RPAREN <i>vardecl_list</i> <i>compound</i>
<i>idlist</i>	→	ID COMMA <i>idlist</i> ID
<i>paramlist</i>	→	<i>type</i> ID COMMA <i>paramlist</i> <i>type</i> ID
<i>params</i>	→	<i>paramlist</i> ϵ
<i>sign</i>	→	PLUS MINUS
<i>simplexpr</i>	→	<i>term</i> <i>sign</i> <i>term</i> <i>simplexpr</i> PLUS <i>term</i> <i>simplexpr</i> MINUS <i>term</i>
<i>stmt</i>	→	<i>compound</i> <i>variable</i> ASSIGN <i>expr</i> IF <i>expr</i> <i>stmt</i> IF <i>expr</i> <i>stmt</i> ELSE <i>stmt</i> WHILE <i>expr</i> <i>stmt</i>
<i>stmtlist</i>	→	<i>stmt</i> SEMICOLON <i>stmtlist</i> <i>stmt</i> SEMICOLON
<i>term</i>	→	<i>factor</i> <i>term</i> MUL <i>factor</i> <i>term</i> DIV <i>factor</i>
<i>type</i>	→	<i>base_type</i> <i>constr</i>
<i>vardecl</i>	→	<i>type</i> <i>idlist</i>
<i>vardecl_list</i>	→	<i>vardecl</i> SEMICOLON <i>vardecl_list</i> ϵ
<i>variable</i>	→	ID ID LSQR <i>expr</i> RQSR POINTER ID

4 Lexical Analysis

The following table summarizes the tokens used in the grammar:

Token Type	Explanation
ASSIGN	= (assignment operator)
COMMA	, (used as a separator)
DIV	/ (division operand)
ELSE	(keyword)
EQUALTO	== (equality comparison)
ID	<i>letter followed by zero or more letters/digits</i>
IF	(keyword)
INT	<i>integer</i> (keyword for int type)
LBRACE	{ (used for opening a compound statement)
LPAREN	((various uses)
LSQR	[(start of array dimension)
MINUS	- (subtraction operator)
MUL	* (multiplication operator)
NUM_INT	<i>integer constant value</i>
NUM_REAL	<i>floating point constant value</i>
PLUS	+ (addition operator)
POINTER	^ (pointer operator/declaration)
RBRACE	} (end of compound statement)
REAL	<i>real</i> (keyword for floating point type)
RPAREN) (various uses)
RSQR] (end of array dimension)
SEMICOLON	; (terminates a statement)
VOID	<i>void</i> (keyword for void type)
WHILE	(keyword)

5 Type Checking

Here are the basic rules for type checking:

- Everything should be strongly typed, meaning that they should be 100% structurally equivalent, with the following exception:
 - if the left hand side of an assignment statement is a pointer to type X, then the right hand side can either be a pointer to type X or an array of type X.
- when performing operations (i.e. +, -, ...) and there is a type mismatch between operands, then the result's type should be an "error type".
- undefined variables should be thought of as having an "error type", which cannot be structurally equivalent to anything (including other error types).

6 Symbol Tables

Every input file should at least give a (possibly empty) table of global variables. This table should also contain prototypes for all functions in your program.

Beyond the global table, you should have one symbol table for each of your functions. Such a symbol table should contain all parameters and local variables for the function. Parameters should be listed first, followed by local variables.

Each entry in any symbol table should contain the following information:

- the name of the symbol (i.e ID string value).
- the type of the symbol
- a boolean indicating whether or not this symbol is a parameter.

7 Extra Credit

Don't forget – get this to me 48 hours or more early and you get a free 5 point s. So, submit this project by 2/27 at 11:59 PM and you'll get a free 5 points!

8 Grading Breakdown

Correct Submission	10%
Successful Compilation & Makefile	35%
Correct Execution	40%
Comments & read.me File	15%
Extra Credit	5%

9 What to Submit

You will be submitting a tar file containing all source code, a makefile, and a `read.me` file for the project. If necessary, we will discuss the creation of tar files as well as how to build (and use) an appropriate `Makefile` in class. The `read.me` file should be a plain ASCII text file that overviews your project, including:

- your name and any other identifying information.
- how to compile your project (if you have submitted a Makefile as required, this should be something like “type `make` to build the project”).
- how to run your project
- any known bugs your project has, and possible fixes to those bugs (partial credit bounds here).
- an overview of how you solved the project.
- what you named your output files (the symbol table and type checking files).