Programming Assignment 3

Due Date: 11:59 PM on Mar 31, 2008

For PA3 you will further develop the compiler for a Bronco# programming language. Besides the syntactic errors you have found in PA2, your PA3 program is required to find all semantic errors. The semantic errors include:

- No duplicate names are allowed for variables, types, user-defined type names, and function names at the same level. However, a variable declared at the different levels such as different compound blocks can have a duplicate name. Like ‘C’, you can declare variables within compound blocks.
- Checking for Undefined variables.
- Within a record, enumeration definition, any two fields or two enumeration values should have different names.
- The three types int, float, and short satisfy the following relation: short < int < float, where < represents a subtype relation. If A<B, then A is called subtype and B is called supertype.
- For an assignment, the type of an expression at RHS should be either the same type or subtype of a variable at LHS.
- For all arithmetic binary operations, if the two subexpressions’ types are different, the result type of the expression is the supertype of the two subexpressions’ type. Otherwise, an error should be reported.
- For a function call, the type rule for an assignment can be applied for the types between a real argument and its corresponding parameter.
- The other requirements imposed on Bronco# should be the same as the requirements for C/C++.
- There is one and only one main() method whose return type is void in each input file.

To complete this project, your program must find all semantic errors based on the above requirements:

1. Your program should have some error recovery features.
2. Error Output should be written to STDOUT.
3. Your program should read an input file from a command line.

You can choose to use Linux, cygwin (Windows), UNIX or other operating systems where yacc/bison is supported. However, before you submit your assignment, you are required to run your program in SunOS operating system so we can grade your homework easily. You should use an ssh client to connect to csy[01-12].cs.wmich.edu in order to use SunOS system.
Send your source files to Bilal Abubark at bilal.abubakr@wmich.edu, and cc to zijiang.yang@wmich.edu.
with Subject: *Course Number:PA3: Your Name.*
INPUT GRAMMAR FOR PA2

Module

module → unit { unit }  //{} represents 0 or many repetitions
unit → declaration ‘;’  //global variables
    → function  //functions
    → type_decl ‘;’  //types
    → λ  //NULL

Type Declarations

type_decl → type ident ‘=’ type_def
type_def → array ‘[’ int_const ‘..’ int_const ‘]’ of type_id
    → record field_list end
    → enum ‘{‘ ident_list ‘}’
    → pointer of type_id
field_list → field {‘;’ field}
field → type_id ident_list
type_id → ident | base_type  //‘ident’ is a defined type
base_type → int | float | short  //short takes 2 bytes and others take 4 bytes

Variable Declarations

declaration_list→declaration {‘;’ declaration }
declaration → type_id ident_list
ident_list → ident {‘,’ ident }

Function Declarations

function → function return_type ident ‘(‘ parameter_list ‘)’
    → compound_stmt
return_type → type_id | void  //Function returns nothing
parameter_list → parameter {‘,’ parameter } | λ
parameter → mode type_id ident
mode → ref | λ

Statements

stmt_list → statement {‘;’ statement}
statement → assign_stmt
    → while_stmt
    → if_stmt
    → function_call  //Must return VOID
→ compound_stmt
→ return_stmt
→ print_stmt
→ new_stmt
→ λ //NULL statement

assign_stmt → variable “=” expr
while_stmt → while ‘(’ expr ‘)’ statement
if_stmt → if ‘(’ expr ‘)’ statement
→ if ‘(’ expr ‘)’ statement else statement
function_call → ident ‘(’ expr_list ‘)’
compound_stmt → '{' declaration_list ';' stmt_list '}'
→ '{' stmt_list '}'
return_stmt → return ‘(’ expr ‘)’ | return
print_stmt → print ‘(’ output_list ‘)’
→ println ‘(’ output_list ‘)’ | println
new_stmt → new variable

Output List

output_list → output_element { ‘,’ output_element } output_element → expr | string

Expression List

expr_list → expr { ‘,’ expr } | λ

Variables

variable → ident variable_tail
variable_tail → ‘[’ expr ‘]’ variable_tail
→ ‘.’ ident variable_tail //Ident is defined field
→ ‘->’ ident variable_tail
→ λ

Expressions

expr → expr bin_op expr
→ unary_op expr
→ ‘(’ expr ‘)’
→ variable
→ function_call
→ int_const
→ real_const
bin_op → ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’
Standard mathematical rules of precedence and associativity apply (‘%’ has the same associativity as ‘+’). Binary operators are LEFT associative.

**Note:** All keywords are written in **bold**. All other terminals/tokens are in *italic.*