Compile and execute variable declarations, assignment statements and basic I/O.

Program

You should write a driver program that takes a text file as a command line argument and then processes it using your compiler.

The compiler should parse the input for a sequence of statements and generate a code tree representing that expression.

The code tree is then used to generate machine code instructions to be executed later.

Statements

One of the main additions to the compiler is the processing of statements.

These should be grouped into statement sequences.

Your parser should have rules (routines) for parsing statement sequences and generic statements.

For now, the statement sequence routine should parse as long as there is input.

The generic statement routine should be able to determine, from the current token, what type of statement is to be parsed.

This is usually done through the recognition of keywords.

Anything else should be an identifier representing the start of an assignment statement (or function call, but that's a different project).

Statements are often terminated with semicolons. This is not generally true.

However, all of the statements implemented in this assignment should be terminated with semicolons.

It's often better to process the semicolon with each statement rather than do catch-all semicolon handling after any type of statement.

The statements that should be handled in this assignment are print(), read(), variable declaration statements for int4 data types, and assignment statements.

Expressions

Arithmetic expressions will be expanded into something much larger, which for lack of a more descriptive term, will just be called expressions.

The parsing for an expression will ultimately go through logical expressions, into relative expressions, and then into arithmetic expressions. For now, your expression parsing routine should

call your arithmetic expression parsing routine. Relational and logical expressions will be added later.

An important change to your code tree creation is that each node involved in an expression should have a value type associated with it.

One way to think of this value type is the type "returned" from a particular operation.

For example, if an addition node has two children who each have a value type of int4, then the resulting value type for the addition should also be an int4.

For this assignment there should be three node value types: a null type, a int4 type, and a string constant type.

At the very end of the arithmetic expression parsing, where integer constants are parsed, you should make changes to handle string constants as well.

The value type of the node generated will be a string constant.

The node generated should contain either the value of the string constant or a reference of some kind to the location where the value is located.

(See string table, below.)

Strings

There are two basic ways to implement strings. Both use arrays of characters.

One terminates the array with a special value, often a zero byte.

The other keeps track of the number of characters used in the array. We will use the null terminated character approach.

Your compiler should have a string table associated with the compiled program at run-time.

This table should contain all of the string constants discovered by the parser during the compilation stage.

The string table should be created and initialized when the compiler starts.

The values in the table can be inserted either during the creation of the code tree or at machine code generation time.

There are no particular advantages of one approach over the other.

Helper Functions

You will need some helper functions, written in C/C++ to avoid a lot of the headaches associated with I/O.

These routines will be called by the generated machine code as needed. These are

A routine that should take a single four-byte integer and print it to the standard output. There should be no return type.

A routine that should take a character pointer and print it to the standard output as a null terminated string. There should be no return type.

A routine that takes no input and returns a 4-byte integer. The value should be read from the standard input.

In addition, if you don't already have them, you should create functions that

Take a four-byte integer and append it to the end of the program array as four bytes in little-endian order

The same thing, but for eight-byte integers. This will be useful for placing pointers in the program code.

Print

The print statement has the following syntax:

print (expr, expr, ...) ;

, i.e., a comma separated list of expressions. The code tree node for this statement should have children for each of the expressions.

The code generation for this statement consists of generating code for each expression in turn and then calling the appropriate helper function according to the value type of the expression.

Read

The read statement has the following syntax:

read (variable);

The statement only has a single operand (or argument). The code tree node for this statement may have 0 or 1 children depending on how you want to store the variable reference. Pick an approach and use it consistently.

The code generation for this statement consists of calling the appropriate helper function and then moving the resulting value into the storage location associated with the variable. Currently, only 4-byte integers can be read.

Assignment

The assignment statement has the following syntax:

variable <- expr ;

The code tree node for this statement should have two children, one for the variable, and one for the expression. The code generation for this statement consists of generating the code for the expression and then taking the resulting value and storing it in the location associated with the variable, which can be found in the symbol table (see below).

Variables and Declarations

Currently only int4 variables can be declared. The declaration statement has the following syntax:

int4 variable_name ;

An entry in the symbol table should be created for each variable if one does not already exist. If an entry does exist, then a duplicate variable error should be returned. The entry should be created by the code tree generation routine at the time that the declaration statement is processed. It is not necessary to create the storage for the variable during code tree processing, but can be done if desired.

In expressions, when encountering a variable name, the symbol table should be checked. If the name does not exist in the symbol table, then an undeclared variable, or non-existant symbol error should be returned.

In variable declaration, it is tempting to not generate a code tree node. However, one is needed. The code generation for this statement consists of creating storage for the integer if it has not already been done during the code tree stage. Then, (and this is the real reason for the node), generate code to set the variable to have an initial value of 0.

Symbol Table

A symbol table keeps track of every symbol used in the program and any information needed to assist in the implementation of that symbol.

A symbol should have a name and a symbol type.

For this assignment, the type should be either a null type or a variable type. Future types include that of function.

Each symbol should also contain information about the location of the symbol.

As this could take place in several different ways, each symbol should have a location type.

These types include memory, register, and stack.

The symbol should also contain additional information needed to support the location type, i.e. memory address, or register label, or stack offset, etc.

The symbol should also contain information about the type associated with the symbol.

There should be two routines associated with the symbol table.

One is for searching for a symbol by name and symbol type.

The other is for inserting a symbol into the symbol table.

Both routines should return somehow a location to the symbol in question, either the one found (or not) or the one inserted.

*NOTE: Be careful as to how the location of the symbol is returned and how the symbol table is implemented.

For example, it would be really easy to push the symbols into a C++ vector and return an address to each symbol as it is pushed into the symbol table.

Unfortunately, as the vector automatically resizes itself it creates larger storage and copies old values into the new array. Pointers to symbols in the old array will then no longer be valid. Possible solutions involve fixed size symbol tables or locations that don't involve pointers. poi

Integer Storage

You should create storage for the integer variables referenced in the symbol table.

This can be done at the time the symbol is created, or later in code generation.

The memory for the variables can be created dynamically, variable by variable or as an offset into a larger block of memory specifically for variables.

For what it's worth, the latter approach more closely mirrors what typically happens in practice.

Code Tree

You should have a node type for a statement sequence. The children of this node will be the individual statements.

It is tempting to have a node type for a generic statement, but this is not needed. You should have node types for the various statement types: print, read, assignment, and variable declaration.