Parser Actions

This document describes the code that is generated for each statement in the program. The companion document, vmActions.pdf, describes the actions the generated code performs at runtime in the VM.

An example of parsing a small program can be found in ParserExample.pdf

Tokens in the language.

variable: All variables are global, i.e. a variable is visible throughout the entire program. Variables have names of the form *letter { letter | digit}** with a maximum length of 32 characters.

int: ints have the form digit*, with a maximum of 16 characters, where digits are decimal digits.

string: a string has the form "*{character | digit}*"* A *character* cannot be a quote symbol (") or a space character. In the test programs, "_" are used in place of spaces in longer strings. This makes reading strings easier for you.

label: All labels are global, i.e. a label is visible throughout the entire program. Labels have names of the form *letter { letter | digit}** with a maximum length of 32 characters. In test programs labels and variables will never have the same name, so you do not need to handle this case.

Data structures used to parse the language.

Symbol table: The symbol table is a map from a variable or label that is the key to a table entry, described below. The symbol table should be implemented as a *singleton.* See code at the end of this document to see how to use a std::map in C++.

Table entry: This is the data part for the symbol table, and is a <*location, length*> double which gives the location in memory (variable) or in the program (label) of the symbol and the length of the data associated with the symbol. For scalar variables the length is 1, for array variables it is the length of the array, and for labels the length is 0.

Instruction buffer: As code is generated for statements (see below), the code is put into an array of *Stmt* smart unique pointers. *Stmt* will be an abstract class, and different statements will have their own concrete classes that extend the *Stmt* class. All statements (except *label* statements and declaration statements) occupy one location in the program buffer. This should be implemented as a *singleton*.

String buffer: The prints instruction specifies a string to be printed. This buffer holds these strings.

Statements in the language.

Statements are made up of a string of tokens. A statement is completely contained on a line, and there is only one statement on a line. Statements always begin with an operation that will identify the kind of statement it is. All statements can be recognized by a state machine, i.e., the syntax of statements is describable as a regular expression. Thus, our language is significantly simpler to parse than, for example, Java or C++.

declscal *var* : declare a scalar variable. When a scalar variable is declared a symbol table entry is defined, with the name of *var* being the key, and the data being a double with the memory location (which is the same as the instruction buffer location) holding the variable (current value of *bufferIdx*) and the *length*

being 1. All variable declarations occur before any other statements. This will make compiling the program much easier.

declarr *var length* : declare an array variable, where *length* is the length of the array. When an array variable is declared a symbol table entry is defined, with the name of *var* being the key, and the data being a double (two-tuple) with the memory location (which is the same as the instruction buffer location) holding the variable (current value of *bufferIdx*) and the *length* being the length specified in this declaration. All variable declarations occur before any other statements.

label *label* : declare a label. When a label is declared a symbol table entry is defined, with the name of *label* being the key, and the data being the memory location where the label is (location of the next statement in the instruction buffer). Label declarations occur where the label is in the program.

gosublabel *label* : declares a label that can be the target of a *gosub*. When a label is declared a symbol table entry is defined in the current symbol table, with the name of *label* being the key, and the data being the memory location where the label is (location of the next statement in the instruction buffer). A new symbol table level is then created to hold any variables declared in the subroutine.

The gosublabel generates an OP_ENTER_SUBROUTINE instruction in the instruction buffer. The size of the stack frame is not known until the *return* statement (see below) for the gosublabel is encountered, at which time this is added to the OP_ENTER_SUBROUTINE instruction.

A gosublabel will only occur at the top level of the program, i.e., the label will always be placed in the top-level symbol table. Stated differently, we do not have nested subroutines or functions.

start: denotes the start of the source file and the program. It must be the first statement in the program.

end: denotes the end of the source file. If non-blank characters follow the *end* statement an error should be issued. If no end statement exists, and error should be issued.

This statement should fill in the operand field all statements that reference labels with the location of the label. The start statement operand needs to be filled in with the number of variables declared in the outer scope of the program, i.e., the number of variables declared at the beginning of the program before any *gosublabel* instructions are encountered.

exit: exit the program, i.e., terminates execution. It can be anywhere in the program. An OP_EXIT_PROGRAM statement is added to the instruction buffer.

jump *label* : An OP_JUMP statement is added to the instruction buffer.

A jump not in a subroutine will always jump to a label that is not in a subroutine. A jump in a subroutine will always jump to a label declared in the subroutine. At the end of a subroutine the operands for all OP_JUMP statements in the subroutine should be filled in with the label location. At the end of the compile of the program, the operands for all OP_JUMP statements in the program that are not within a subroutine should be filled in with the label location.

jumpzero *label* : An OP_JUMPZERO statement is added to the instruction buffer.

A jumpzero not in a subroutine will always jump to a label that is not in a subroutine. A jumpzero in a subroutine will always jump to a label declared in the subroutine. At the end of a subroutine the operands for all OP_JUMPZERO statements in the subroutine should be filled in with the label location. At the end of the compile of the program, the operands for all OP_JUMPZERO statements in the program that are not within a subroutine should be filled in with the label location.

jumpnzero *label* : An OP_JUMPNZERO statement is added to the instruction buffer.

A jumpnzero not in a subroutine will always jump to a label that is not in a subroutine. A jumpnzero in a subroutine will always jump to a label declared in the subroutine. The OP_JUMPNZERO operand is set to the position of the label as in the jumpzero statement.

gosub *label* : An OP_GOSUB statement is added to the instruction buffer.

When the end statement is processed, the operand of all OP_GOSUB statements is set to the location of the specified label in the program. Note that gosub labels are always included in the symbol table for the main program and not in the symbol table of a subroutine.

return : return from a subroutine call. Generates an OP_RETURN statement in the instruction buffer. Any jump statements in the subroutine referring to labels in the subroutine will have the operands that hold the label address filled in. Any references to variables by instructions in the subroutine will have the location of those variables filled in as the operand of the instruction.

pushscal var_s : An OP_PUSHSCALAR instruction, with the location of var_s as the first operand, is added to the instruction buffer.

pusharr var_a: An OP_PUSHARRAY instruction, with the location of the start of var_a as the first operand, is added to the instruction buffer

pushi *int* : An OP_PUSHI instruction, with the value of *int* as the first operand, is placed into the instruction buffer.

pop : An OP_POP instruction is added to the instruction buffer

popscal var_s : An OP_POPSCAL instruction, with the location of var_s as the first operand, is placed into the instruction buffer.

poparr var_a : An OP_POPARRAY instruction, with the location of var_a as the first operand, is placed into the instruction buffer opcode: xxxx

dup : An OP_DUP instruction is placed into the instruction buffer.

swap : An OP_SWAP instruction is placed into the instruction buffer.

add : An OP_ADD instruction is placed into the instruction buffer.

negate : An OP_NEGATE instruction is placed into the instruction buffer.

mul : An OP_MUL instruction is placed into the instruction buffer.

div: An OP_DIV instruction is placed into the instruction buffer.

printtos : An OP_PRINTTOS instruction is placed into the instruction buffer.

prints : An OP_PRINTS instruction is placed into the instruction buffer. The specified string is added to the string buffer, and the first operand is set to the strings location in the string buffer.

Actions of the Parser, or what you need to program

The parser will read an input file which we'll refer to as *sourcefile* whose name is given on the program command line. The statements of the program will be read line-by-line, and the objects described above will be placed into the *instruction buffer.*

At the end of a subroutine label (denoted by a return statement) any jump statement labels and variable references in the subroutine will be filled in. At the end of the program, any jump statement labels, and variable references will have the location of the label or variable filled in. At the end of the program, all OP_GOSUB instructions will have the operand filled in with the location of the label indicated in the gosub statement.

Should any statement in *sourcefile* not follow the syntax given above, you should issue an error to help you in finding parser errors, *but this will not be graded*. Note that multiple spaces can appear anywhere a single space is legal. Spaces may follow the end of the statement in the *sourcefile*, but the presence of any non-space character after the statement is an error.

Should a variable or label be defined more than once, and error should be given.

When an error is found, a message should be printed and the scanner should terminate normally, i.e, without a segmentation fault. I use *exit(0);* to terminate.

Each Class deriving from *Stmt* will have a *serialize* method that will dump the object to a file in sequential order. This file will have a .out file type.

You should have a dump routine that prints a formatted listing of what is in the instruction buffer. You should print this to stdout (i.e., std::cout). Examples of how this output should look for each input x are shown in x.output.

What you will be graded on

Error messages should be printed for malformed programs, as indicated above

For syntactically correct programs, you should execute the dump routine showing the compiled program

Sample programs will be provided that will test the functionality of your program, and output from these programs will be provided.

Example of using std::map for a symbol table

```
#include <map>
#include <string>
#include "SymbolTable.h"
SymbolTable::SymbolTable( ) {
```

```
  definedMap = new std::map<std::string, int>( );
  idx = 0;
}
make_shared<map<std::string, int>> SymbolTable::makeSymbolTable( ) {
  if (!mapDefined) {
    mapDefined = make_shared<map<std::string, int>>( );
  }
  return mapDefined;
}
int SymbolTable::getData(std::string key) {
  if (!mapDefined) {
    return 2147483647; // just an error value I picked
    for (auto const& e : *definedMap) {
    if (e.first == key) {
      return x.second;
    }
  }
}
bool putEntry(std::string key, int datum);
#endif /* SYMBOLTABLE_H_ */
```