Compiler guidance

Some simplifications to the project

- There will be one function, main, in test code
- This will simplify managing symbol tables
- Since there is only the main function, there will be no need for call, callr, ret and retr.
- Only integer forms of typed instructions will be used, i.e., printi, pushi, etc.

High level strategy (1)

- Have a main function that opens a file passed as an argument, reads it line by line, and passes the line to a method to parse

```
BufferedReader reader = null;
   try {
      reader = new BufferedReader(new FileReader(fileName));
      String line = reader.readLine();
      while (line != null) {
         parseLine(line);
         line = reader.readLine();
      }
   } . . .
```

Code skeleton for reading the file

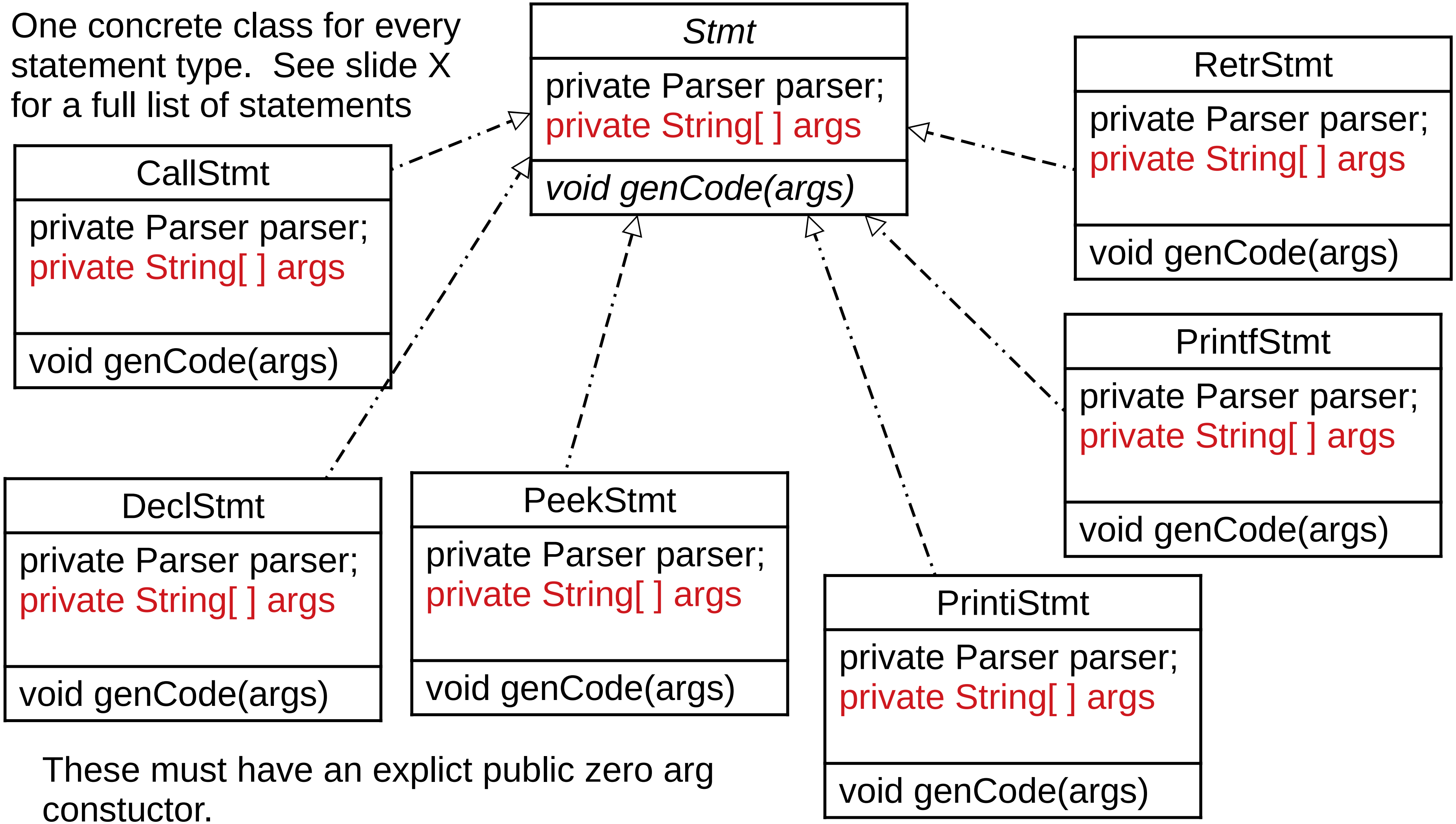
High level strategy (2)

- Parse the line
  - break the line of characters up into *tokens*
  - examine the first token to determine what kind of statement is being parsed
  - scan the tokens finding the arguments needed to generate code
- Have classes that parse different kinds of token sequences

```
line = line.trim( );
line = line.replaceAll(",", " , ");
line = line.replaceAll("\\s+", " ");
String[ ] tokens = line.split("\\s");
token tokens[0];
if (token != null) {
   if (token.matches("decl|retr|call|add|...")) {
      Stmt stmt = StatementFactory.getStatement(token);
      stmt.genCode(tokens);
   } else {
      System.out.println("Unknown stmt: "+token);
   }
}
```

Code for parsing a line

One concrete class for every statement type. See slide X for a full list of statements



These must have an explict public zero arg constuctor.

Let’s use reflection to create all of our *Stmt* objects

First create a map from the name of statements in our language to the name of the class that represents the state

```
private static String[ ] stmtClasses = {"DeclStmt", “CallStmt”, “CallrStmt”,
                                        “RetStmt”, “PushiStmt”, …
                                        };

// used to map statement names onto statement classes
private static String[ ] stmts = {"decl", "call", "callr", "ret","pushi", . . .
                                 };
```

Let’s use reflection to create all of our Stmt objects

Next, let’s create the Stmt object for each statement

```
private static Map<String,Stmt> statements = new HashMap<String,Stmt>( );

for (int i = 0; i < stmtClasses.length; i++) {
   Class<?> cls = null;
   Constructor<?> constructor = null;
   CompilerState state = null;

   cls = Class.forName(stmtClasses[i]); // these statements will all
   constructor = cls.getConstructor(Stmt.class); // need try-catch blocks
   Stmt stmt = (Stmt) constructor.newInstance(); // around them.
   statements.put(commands[i], stmt);
}
```

Now lets see what happens when we execute

```
    Stmt stmt = StatementFactory.getStatement(token);
```

in *parseLine*.

```
public static Stmt getStmt(String stmt) {
    Stmt obj = statements.get(stmt);
    if (obj == null) {
       System.out.println("Error getting state for class "+state);
    }
    return obj;
  }
}
```

- Each *Stmt* object HASA *Parser* object that parses the tokens for that statement
- The parser fills in the fields of an *ArgObj* with the values of the relevant tokens

Sequences of arguments

pattern: variable number of integers followed by string
call (first argument is the count of integer arguments remaining)
callr (first argument is one less than the count of integer arguments remaining)

pattern: string string
decl

pattern: string int
subr (string is a label)
peek
poke

Sequences of arguments

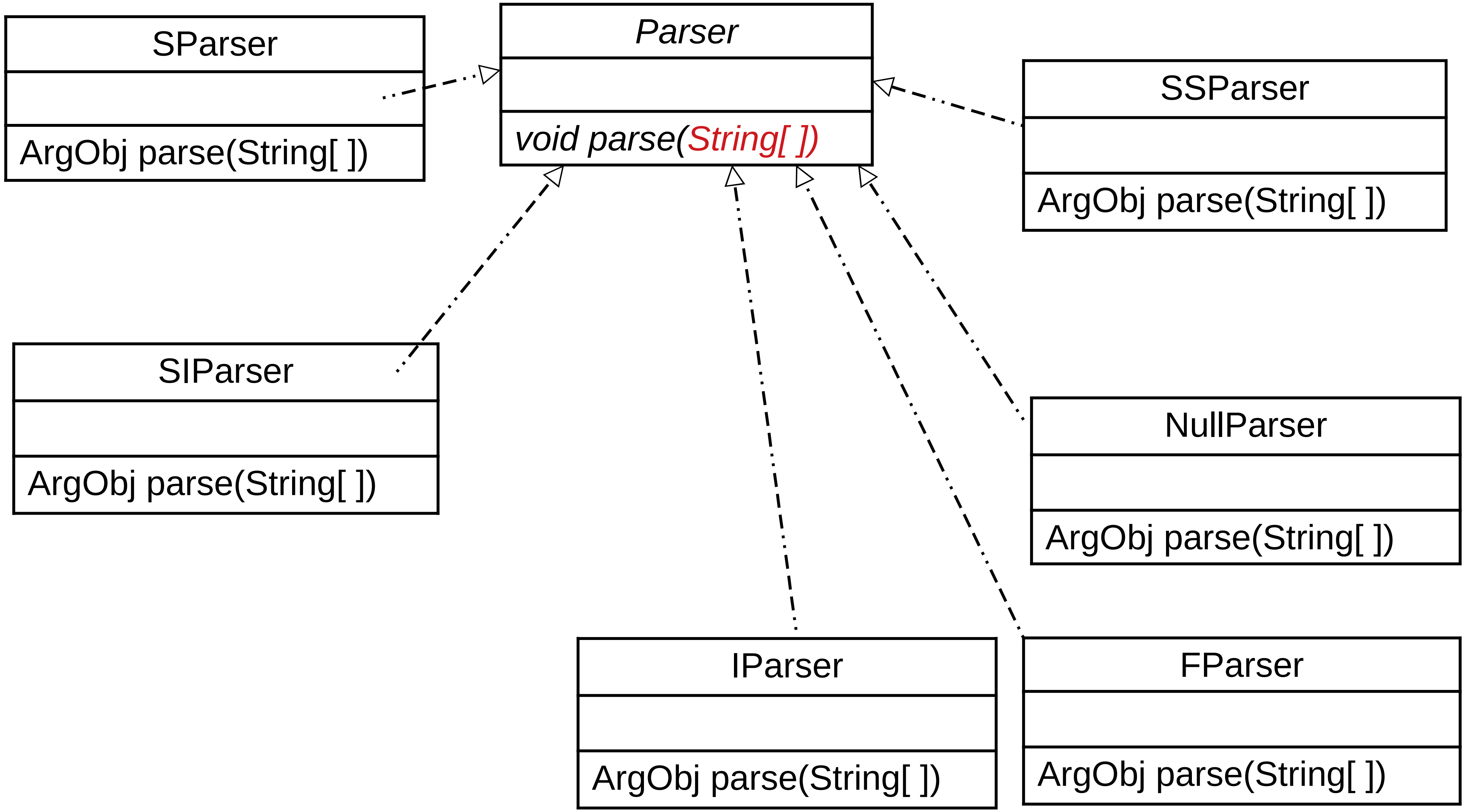
pattern: string
lab (string is 8 characters or less)
printc (string is a single character)
jmp (string is 8 characters or less)
jmpc (string is 8 characters or less)
pushc (string is a single character)

pattern: int
printi
prints
retr
printv
pushi
pushs

Sequences of arguments

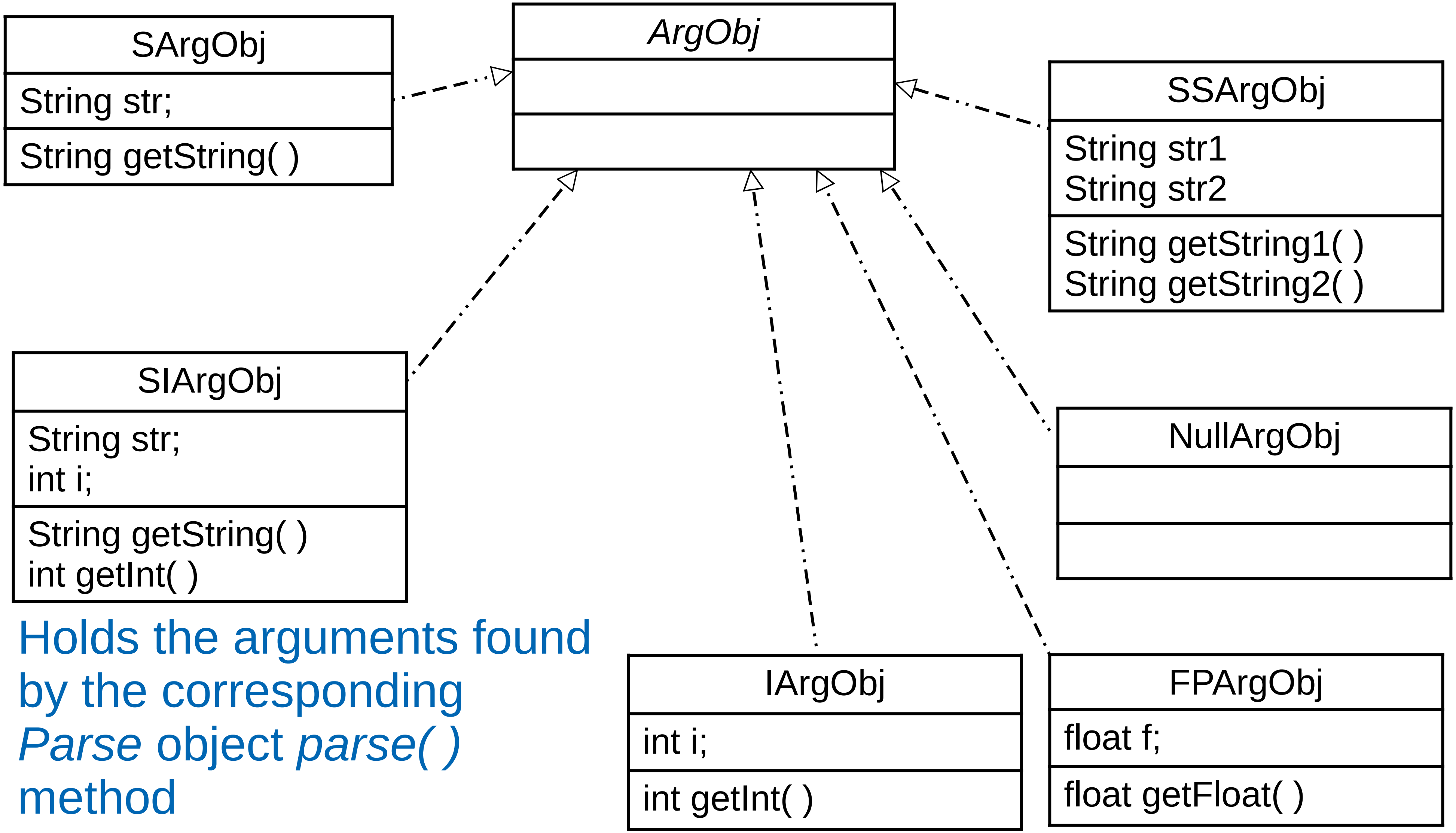
pattern: float
printf
pushf

no argument: null
retr
cmpe
complt
compgt
swp
sub
mul
div


SParser
ArgObj parse(String[ ])
Parser
void parse(String[ ])
SSParser
ArgObj parse(String[ ])
SIParser
ArgObj parse(String[ ])
NullParser
ArgObj parse(String[ ])
IParser
ArgObj parse(String[ ])
FParser
ArgObj parse(String[ ])

Parser class names meaning

- SParser: looks for a string, i.e., *lab foo*
- SSParser: looks for two strings
- SIParser: looks for a Sting and an int *subr 0, func*
- IParser: looks for an int *printi 50*
- FParser: looks for a float *printf 50.5*
- NullParser: looks for no arguments *add*



Holds the arguments found by the corresponding *Parse* object *parse( )* method

Now you have the arguments needed for a statement

- Generate the code needed by the statement
- read the next line
- do this until the end of the file

Other considerations

- You need to maintain a *symbol table*
- The symbol table is simply a map from a symbol (key) (String) to a value
  - For labels, the key is the label string, the value is the position in bytecode where the label is
  - for variables, the key is the variable name, the value is the offset from the sp, i.e., the number of things that will have been pushed onto the stack when the variable is declared

More than one symbol table needed

- A main symbol table that keeps all labels of functions and statements
- A local one for each function that keeps variables
  - This one can probably be gotten rid of when you finish compiling the function