1506801 Submission 2

Jack Neilson

December 19, 2017

Contents

1 Statement of Compliance

1.1 Implemented

- Semantic Analyser - The submission includes a fully working semantic analyser which takes an annotated abstract syntax tree and populates a symbol table, as well as checking for semantic errors.
- Code Generator - The submission includes a fully working code generator which takes an annotated syntax tree and generates appropriate instructions for the target abstract machine.

1.2 Not Implemented

All parts of the coursework specification have been implemented.

2 Semantic Analysis

The sematic anaylser is implemented as a set of multiple partial classes, each of which is specific to a part of the language definition. These partial classes have been designed using the visitor pattern as a way of implementing double dispatch. When the check() method is called on the instantiated semantic analyser it will traverse the passed abstract syntax tree and populate the symbol table while checking for semantic errors. Note that the annotated abstract syntax tree will not be modified when using this pattern. Since the visitor pattern is used throughout it is simple to implement methods appropriate for different node types on the tree. For example, when the first call to check() is made the semantic analyser calls VisitProgram(), which in turn calls Program.Command.Visit(). The appropriate visit command method is then called depending on which interface the Command object has implemented, identified by their signatures (VisitAssignCommand takes an AssignCommand, VisitCallCommand takes a CallCommand and so on). This continues for every node on the tree until the entire tree has been traversed and checked for errors.

2.1 Symbol Table

The symbol table has been implemented in the IdentificationTable.cs file. The symbol table itself has been represented by a list of dictionaries, with the current scope always being at the head of the list. It allows for the entry and retrieval of new terminals and their attributes, as well as the opening and closing of scopes. When a new scope is opened, it is added at the head of the list so that when scope is closed, all that is needed is a call to list.removeAt(0). The symbol table should only contain entries from the current and previous scopes, and should be empty when semantic analysis is complete.

2.2 Variable Declaration and Use

The symbol table is used whenever a variable is declared or used.

2.2.1 Declaration

When a new variable is declared the symbol table is checked to see if a variable of the same spelling is already declared. If there is, the Duplicated property of the variable declaration is set to true. The new declaration is then added to the table at the current scope (i.e. list[0]), with the key being the spelling of the variable.

2.2.2 Usage

When a variable is used it must first be retrieved from the symbol table. The current scope and all previous scopes are iterated over until a match is found and returned, or if no match found then null is returned.

2.3 Type Checking

Type checking is done as the semantic analyser traverses the tree. When the type of a terminal can be inferred (for example, in the case of an integer literal) the symbol table is not needed. However, when a variable is used the symbol table must be checked to find the type of the variable in order to perform type checking.

2.4 Variable Scope

As described previously, variable scope is checked by using a list of dictionaries. Each dictionary represents a scope level, with the current scope being at the head of the list, the secondmost current scope being at $head + 1$ and so on. If a variable is in scope, it will be an entry in one of the dictionaries. Since the list is iterated over from head to tail, only the most recent matching entry is returned which is the desired behaviour. When a scope is closed, the head of the list is removed, leaving $head + 1$ (the secondmost recent scope) as the new current scope at the head of the list. If variable is referred to after its scope has been closed it will not be found in the symbol table, as the dictionary containing its related entry will have been removed from the list.

3 Code Generation

Once again, the visitor pattern is used during code generation. An annotated syntax tree which has been checked for correctness is passed to the emitter, which will generate code for the target abstract machine using predefined templates which have been elaborated on below.

3.1 Commands

3.1.1 Assign

When an assign command is visited, the expression is visited to find how much memory is needed to store it. While visiting the expression, the emitter should also output the instruction to load the literal value of the expression on to the top of the stack. Encoder.EncodeAssign() is then called with the vname object, a new frame which is larger than the old current frame by the size of the expression, and the size of the expression. This in turn calls KnownAddress.EncodeAssign() with the emitter, current frame, size of the variable and variable name, which finally uses the emitter to output a store instruction to pop the value of the expression off of the top of the stack and store it at the given address in the current register.

3.1.2 Call

When a call command is visited, it's actual parameters are visited and their sizes summed. The function or procedure is then visited, passing a new frame that is the size of the actual parameters.

3.1.3 If

The if command is slightly more complex as it requires memory addresses to be back patched. First, the expression to be evaluated is visited. The emitter then outputs a new JUMPIF false instruction, with the address to jump to set as the code stack base. The address of this instruction is stored for later use. The command to be executed if the expression evaluates to true is then visited. The emitter outputs a JUMP instruction, with the address set again to the code stack base. The JUMPIF instruction is now backpatched with the current memory address, so that if the expression evaluates to false the machine jumps past the command executed if the expression is true. The false command is then visited, and finally the JUMP instruction is backpatched with the current memory address so that the machine jumps past the command to be executed if the expression is evaluated as false once it has completed the command to be executed if the expression is true.

3.1.4 Let

When a let command is visited, any declarations it contains are visited and their sizes summed. The command is then visited, passing a new frame which is larger than the current frame by the summed size of the declarations. Once this command has been completed, the emitter outputs a POP instruction to remove all declarations from the stack.

3.1.5 While

When visiting a while command we must again use backpatching. A JUMP instructions is outputted, with the address to jump to being set to the code stack base. The address of the JUMP statement is stored for later use. The address of the next instruction is also stored, to allow for a jump to the command that is to be looped. The command is then visited, and the current memory address is backpatched to the JUMP statement so that the command is not executed before the expression is evaluated. The expression is then visited, and the emitter outputs a JUMPIF true instruction to jump to the loop address that was stored previously.

3.2 Declarations

3.2.1 Constant

When a constant declaration is visited, the AST is decorated with a new runtime entity that corresponds to the expression child of the ConstDeclaration node. If the expression is not a literal, then it is visited and the size of the expression is included in the decorator. Finally, the extra stack size needed to store the constant is returned.

3.2.2 Variable

When a variable declaration is visited, the size needed to store the variable is stored by checking the type of the variable. The emitter the outputs a PUSH instruction to allocate space of this size to the stack, then the declaration is decorated with a new runtime entity of known address. Finally, the size needed to store the variable is returned.

3.3 Expressions

3.3.1 Literal

For both integer and character literals, the emitter will simply output a LOADL command with the value of the literal.

3.3.2 Binary

When a binary expression is encountered the amount of memory needed to store the type of the expression is first stored. Then, both sides of the expression are visited and the current frame expanded to accommodate. After the right hand side expression is visited, the frame is replaced with a new frame of size $exp1+exp2$. The operator is then visited, and finally the size of the binary expression is returned.

3.3.3 Call

When a call expression is encountered the amount of memory needed to store the type of the call is first stored. Then, the amount of memory needed to store the arguments passed with the call is stored. The identifier for the call is then visited, passing a new frame that is the size of the arguments. Finally, the amount of memory needed to store the type of the call is returned.

3.3.4 If

When visiting an if expression, backpatching must be used. The type and test expression are first visited, then the emitter outputs a JUMPIF false instruction the location of which is stored. The amount of memory needed to store the expression to be evaluated if the test test expression is true is found by visiting the expression. A JUMP instruction is outputted, and then the JUMPIF is patched with the current address so that the expression to be evaluated if the test expression is true is not evaluated when the test expression is false. The false expression is then visited, and finally the JUMP command is backpatched with the current memory address.

3.3.5 Let

When visiting a let expression, declarations are visited and the frame expanded to fit the new declarations. Once the expression has been evaluated, the emitter outputs a POP instruction to remove any declarations from the stack.

3.4 Entities

3.4.1 Known Value

When the value of an entity is known at compile time (e.g. in the case of an integer literal) it can simply be pushed to the top of the stack with the instruction LOADL.

3.4.2 Unknown Value

When the value of an entity is not known at compile time but the address is, it must be loaded with a LOAD command. This takes the size of the entity to be loaded, the register (SB, L1 etc.) the entity is stored in, and the displacement relative to the base of that register.

3.4.3 Known Address

This class allows us to store and retrieve variable values from memory. The first method, EncodeAssign will store whatever value is at the top of the stack at the location in memory referenced by the vname object passed to it. The second, EncodeFetch, will retrieve the value from the address referenced by the vname object and push it to the top of the stack. The final method, EncodeFetchAddress, will push the address referenced by the vname object to the top of the stack.

3.4.4 Unknown Address

Unknown addresses are primarily found when an argument to a function or procedure is bound to a variable. The value of the variable cannot be referenced directly since it's address is unknown, however the address of the variable itself is known. When using the value of the variable, we must first push the variable object itself to the top of the stack, then use the STOREI or LOADI commands to pop that object from the top of the stack, and push the object that it references to the stack.

4 Testing

4.1 sub.tri

This program should compile successfully and when ran in the abstract machine interpreter should ask the user for a number between 0 and 100, then decrement by 5 until the number is less than 0 than exit. The compiler compiled this program successfully.

```
let
  const MAX ~ 100;
  var y: Integer;
  var x: Integer
in
begin
  y := 5;
  put('?');
  getint(var x);
  if (x>0) /\ (x<=MAX) then
    while x > 0 do
    begin
      x := x-y;
      putint(x);
      put(',')
    end
  else
end
```

4.2 gcd.tri

This program should fail upon syntax analysis, as the "func" token cannot start a declaration. The compiler rejected this program at syntax analysis

```
let func gcd (x: Integer, y: Integer) : Integer ~
  if x // y = 0 then y else god(y, x // y);
in
  putint(gcd(321, 81))
```

4.3 type.tri

This program should fail upon semantic analysis, the "put" method should take a character and the "putint" method should take an integer. In both cases, the incorrect type has been supplied. The compiler failed at the semantic analysis stage when supplied with this program.

```
let
        var y: Integer;
        var x: Integer
in
begin
        put(x);
        putint('x')
end
```

4.4 1506801.tri

This program should repeat ”John” infinitely. The compiler succeeded in compiling this program, and it was ran successfully by the Interpreter.

```
while true do
   begin
      put('J');
      put('o');
      put('h');
      put('n');
   end
```