CSC488

Compilers and Interpreters

Assignment 4

Nagee Elghassein - elghasse
Theodor Poenaru - poenarut
Yu Sing Wong - wongyu7
Hong Zhi Yew - yewhong
Timur Borkhodoev - borkhodo

Instructor
Dr. Fatemah Panahi

March 15, 2017

Contents

1 Overview

We will walk the AST generated in A3 to create code in A5.

1.1 Visitor Pattern

When building the AST (A3), we chose to implement an `ast_visit` method for each type of node within the class for that node. For A5, we will instead choose to use a visitor pattern. This provides cleaner code that can be easily swapped and adjusted. We will have a `CodeGenVisitor` class that defines how code is generated for each type of AST node. We will also create a `LHSVisitor` class to generate code for the left-hand side of assignment statements (`:=`).

1.2 Termination

We handle termination by processing HALT.

1.3 Initialization

We must initialzie our program counter and pointers. The PC is set to 0, the MSP is set to point to the start of the stack, and the MLP is set to point to the memory chunk just before constants.

2 Storage

We will augment our symbol table to store offsets and lexical levels, as well as the PC for each routine to allow for easy branching. We explain in more detail below.

2.1 Variables

Each variable stores its offset from 0 and increments the scope's offset, so that following variables can calculate their offset. Both `Booleans` and `Integers` will take up one word on the frame. Generally, arrays come in the form `A[len1][len2]`, and so we need to store $len1 \times len2$, when both $len1$ and $len2$ are adjusted to be 0 indexed.

2.1.1 Main

When entering a major scope, we will calculate the size required for all the variables and allocate space accordingly, and we will give the scope an frame/activation record and a block for control information. These variables also get the lexical level of the nearest scope. Each child of `SybmolTableObject` will get their lexical level from the `SymbolTable` they are associated with, and their `offset` is based off of 0 from from that SymbolTable. This remains consistent with our implementatoin of objects in our `SymbolTable`. For arrays, we will add the initial (possibly negative) bounds so that we have a base to do array arithmetic with. Because of this, we have an array index and a machine array index that our classes have to keep track of.

Array Indexing

```
A[-1..10] : Integer
/** The following will be tracked
  { arrayOffset1 = -1
  , arrayOffset2 = None
  , len1 = 11
  , len2 = 1
  , ...
  }
*/
B[5,5..10] : Integer
/** The following will be tracked
  { arrayOffset1 = 0
  , arrayOffset2 = 5
  , len1 = 5
  , len2 = 5
  , ...
  }
*/
```

2.1.2 Procedures and Functions

Procedures and functions may have a nonzero offset, so variables declared in procedures and functions will have their offset calculated treating the initial offset as 0. So a variable v offset o_v from the start of the procedure p at offset o_p has an actual offset of $o_v + o_p$.

Function Offset

```
function hello (a: Integer): Integer
{
  var x: Bool
  var y: Integer

  return a
}
```

When entering a function, we push on MSP and decrement it by the number of paramters, so the return value is under MSP. Here y has an offset of 2 words (argument + var). x), but an actual offset of $o_{\text{hello}} + 2$. For functions in functions, everything is treated as if it belongs to the top function, so the code generation will output the necessary branch.

2.1.3 Minor Scopes

Because minor scopes are independent (things declared in one minor scope cannot be used in another), they may have overlapping space reserved in the frame.

2.2 Integers and Booleans

Integers and Booleans are written as is into the assembly code (with `true` as `MACHINE_TRUE`, and `false` as `MACHINE_FALSE`

2.3 Text

Strings will be pushed character by character starting from the last one.

Text

```
// write ``csc488''
PUSH `8'
PUSH `8'
PUSH `4'
PUSH `c'
PUSH `s'
PUSH `c'
PRINTC
PRINTC
PRINTC
PRINTC
PRINTC
PRINTC
```

3 Expression

Some notation before we start. $< name >$ will refer to an identifier, $< \#name >$ will be its address, and $< @name >$ will refer to its lexical level.

Code used in the templates that are not given in machine.pdf refer to other code in this template, where we replace the code with the code in that block. For example, `GT` is not defined in machine.pdf, but it is used in some places and defined explicity in terms of things in machine.pdf. (Basically, we're saying that this is psuedocode)

3.1 Constants

The constants are Text, Integers, and Booleans. As desribed before, these are directly inserted into code generation (replacing `true` [`false`] with `MACHINE_TRUE` [`MACHINE_FALSE`]). Because of the way we insert text constants, we may end up with multiple copies of the same word, but it's easier to implement.

3.2 Scalars

Scalar variables are accessed as follows:

Accessing scalars

```
// x
ADDR <@x> <#x>
LOAD
```

3.3 Array elements

Because we 0-index arrays intenally, arithmetic operations within array brackets have to be normalized:

Array Elements and Normalization

```
// A[-2] where A[-3..0]
PUSH -2
PUSH 3
ADD
ADDR <@A> <#A>
ADD
LOAD // load A[-2] <=> Machine A[1]

// A[5] where A[3..6]
PUSH 5
PUSH -3
ADD
ADDR <@A> <#A>
ADD
LOAD // load A[5] <=> Machine A[2]

// A[2]  where A[5] (regular)
PUSH 2
PUSH 0
ADD
ADDR <@A> <#A>
ADD
LOAD
```

3.4 Arithmetic Operators

Let `op` $\in$ {`SUB`,`ADD`,`MULT`,`DIV`}. Suppose that we have the steps to get the values of `L` and `R`, then the following is the template for `L op R`

Arithmetic Operators

```
generate_code(L)
generate_code(R)
op
```

3.5 Comparison Operators

Let `comp`$in\{$`LT, GT, EQ, LTE, GTE`$\}$. Suppose that we have steps to get the values of `L` and `R`, then the following is the for `L comp R`

Comparison Operators

```
generate_code(L)
generate_code(R)
comp
```

The following are templates for `GT, GTE, LTE`. (`EQ` and `LT` are available in machine.pdf). These assume that we do these operations on variables. Replace `ADDR <@name> <#name>; LOAD` with `generate_code(name)` if these are not variables.

GT

```
/* a > b */
PUSH -1
ADDR <@b> <#b>
LOAD
MULT
PUSH -1
ADDR <@a> <#a>
LOAD
MULT                    // -b
LT                      // -b < -a <=> a > b
```

LTE

```
/* a <= b equiv a < b+1 */
ADDR <@a> <#a>
LOAD
PUSH 1
ADDR <@b> <#b>
LOAD
ADD
LT
```

GTE

```
/* a >= b equiv a+1 > b
PUSH 1
ADDR <@a> <#a>
LOAD
ADD
ADDR <@b> <#b>
LOAD
GT
```

3.6 Boolean Operators

Let `bop` $\in$ `AND, OR`, then the following are implementations for the two in the set and for `NOT`

Boolean Operators

```
// L bop R
generate_code(L)
generate_code(R)
bop

// NOT A
generate_code(A)
NOT
```

Templates for `AND` and `NOT`

NOT

```
// NOT a
ADDR <@a> <#a>
LOAD
PUSH MACHINE_FALSE
EQ
/*
  a=true, then true=false => false
  a=false, then false=false => true
*/
```

AND

```
// a&b equiv !a|!b
ADDR <@a> <#a> //
LOAD           // a
NOT            // !a
ADDR <@b> <#b> //
NOT            // !b
OR             // !a|!b
```

3.7 Conditionals

Conditionals and If-then-else statements have the same structure. See the Statements section.

4 Functions and Procedures

4.1 Activation Record

4.2 Entrance Code

Entrance Code

```
ADDR 0 F
PUSHMT
PUSH Fbody // label to body of function
```

4.3 Exit Code

Exit Code

```
/*
Swap and pop to preserve the return value while
popping all the variables in the stack
*/
SWAP
POP
...
/*
Put return address on top of stack and goto
the return address
*/
SWAP
BR
```

4.4 Parameter passing

If a procedure or a function takes n parameters, we move the stack pointer back n to grab those parameters before setting the display data. We don't have to store `PrevD` or `n` because it is calculated at compile time.

Parameter Passing

```
PUSHMT
PUSH <n>
SUB
SETD <prevD + 1>
```

4.5 Function Calls and Value Return

Function Calls and Value Return

```
PUSH RETURN_ADDRESS
PUSH PARAMS // label for parameters
PUSH FUNC_BODY // label for function body
/* FUNC_BODY handles exit code */
BR // return value is on top of the stack after the function.
```

4.6 Procedure Calls

This is the same as the function call but there is no return value on top of the stack.

4.7 Display Management

When we enter a new scope, we increment its display. This is handled at compile time.

5 Statements

5.1 Assignment

We load the address at which we wish to store the value and then push the value we want to assign to the stack. If the value is the result of some operation, (e.g.: a := 1 + 2)we perform those now to arrive at the result. Thus the top two elements of the stack wilevel be the value and address, so we use STORE to assign.

Assignment

```
// x := 1
ADDR <@x> <#x>
PUSH 1
STORE
```

5.2 If

We generate the machine code for the predicate and both true and false blocks. After evaluating the condition, we branch to the appropriate block (e.g.: if true, BF wilevel not branch, we execute the true block, then branch to the end).

IfStatement

```
codegen(condition)          // result wilevel be top of stack after this
PUSH <@else>
BF
codegen(true_block)
PUSH <@end_line>
BR
codegen(false_block)        // else_line
...                         // end_line
```

5.3 While and Repeat

We control while loops with branch statements at the end of the loop block that go back to the loop condition (or for repeat loops, go back to the beginning of the code inside the loop).

Loops

```
// while loop
codegen(condition)
PUSH end_line
BF
codegen(inside_loop)
PUSH condition_start_line
BR

// repeat loop
codegen(inside_loop)
codegen(condition)
PUSH end_line
BF
PUSH loop_begin
BR
...                     // end line
```

5.4 Returns

When returning we store any return value at address 0 in that level, and then branch to the end.

IfStatement

```
ADDR level 0
PUSH ret_val
STORE
PUSH end_line
BR
```

5.5 Exit

When we exit we push the next line after the block we are exiting from and then branch to that line (i.e.: when exiting from a loop we want to branch to the line immediately after the loop).

Exit

```
PUSH next_line
BR
```

If exit is caleveled with an integer folevelowing, we clear of that many number of scopes and branch our of the scope. For exit on condition, we simply evaluate the condition first and use BF to skip the exit machine code.

5.6 Reading and Writing

To write a string we push it's characters to the stack in reverse order and folevelow this by one PRINTC for every character in the string. To write an integer, we push the integer and folevelow this with a PRINTI command.

Writing

```
// write "fun"
PUSH "n"
PUSH "u"
PUSH "f"
PRINTC
PRINTC
PRINTC
```

To read a character/integer, we perform a READC/READI and then get an address to store this data and calevel store on the aforementioned values.

Writing

```
READI
ADDR level addr
STORE
```