Contents

A title for the code goes here

```
SUB def // hello
```

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 (`:=`).

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 `Boolean`s and `Integer`s 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

Because functions and procedures are not the highest scope, they have a nonzero offset. 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
}
```

Here y has an offset of 4 words (return + argument + return address + var x), but an actual offset of $o_{\text{hello}} + 4$. 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 independnent (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.

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

```
...
LOAD L
...
LOAR 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

```
...
LOAD L
...
LOAR R
comp
```

The following are templates for `GT, GTE, LTE`. (`EQ` and `LT` are available in machine.pdf)

GT

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

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` ∈ `AND, OR`, then the following are implementations for the two in the set and for `NOT`

Boolean Operators

```
// L bop R
...
LOAD L
...
LOAD R
bop

// NOT A
...
LOAD 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

4 Functions and Procedures

4.1 Activation Record

4.2 Entrance Code

4.3 Exit Code

4.4 Parameter passing

4.5 Function Calls and Value Return

4.6 Procedure Calls

4.7 Display Management

5 Statements

5.1 Assignment

5.2 If

5.3 While and Repeat

5.4 Returns

5.5 Reading and Writing