CSC488 Assignment 4 Team-01

Vishrant Vasavada, Tian Ze(Peter) Chen, Anthony Vandikas, Felix Lu, Badrul Chowdhury

CSC488S Code Generation Templates

0.1. Pseudo instructions

These pseudo instruction indicate where we insert the code for the subexpression/statement.

eval expr
    means emit code to compute the value of expr

addr var
    means emit code to compute the address of a variable

exec stmt/stmts
    mean emit code to execute the statements

anchor label
    means set the label value to the current code address

0.2. Macro instructions

These macro instructions will expand to the actual instructions

```
RESERVE x
        PUSH UNDEFINED
        PUSH x
        DUPN
NOT
        PUSH MACHINE_FALSE
        EQ
BR label
        PUSH label
        BR
BF label
        PUSH label
        BF
POPN n
        PUSH n
        POPN
CALL routine
        PUSH LABEL_RETURN
        BR routine.address
        anchor LABEL_RETURN
        POPN routine.paramcount()
```

1. Storage
Strategy for allocating storage for and addressing all kinds of variables and constants

The target machine is a stack machine. Storage is allocated by pushing onto the top of the stack. Variables are initially assigned the value UNDEFINED.

a. **Variables in the main program**
Treat main program as a procedure with no parameters, reserve enough stack space to for variable in the current major scope.
```
RESERVE majorscope.size()
```

b. **Variables in procedures and functions**
Reserve enough stack space for the variable in the current major scope.
```
RESERVE majorscope.size()
```

c. **Variables in minor scopes**
Storage space required for variables in minor scope get reserved in major scope.

d. **Integer and boolean constants**
We don't need to allocate storage space for them, we push them on to the stack as needed.
integer constant x
```
    PUSH x
```
boolean constant TRUE
```
    PUSH MACHINE_TRUE
```
boolean constant FALSE
```
    PUSH MACHINE_FALSE
```

e. **Text constants**
We simply push the text constant onto the stack when needed. Since text constants are only used in put statements, a more detailed explanation can be found in section 4f.

2. Expressions
The result of evaluating each expression is pushed onto the top of the stack.

a. **Accessing values of constants (including text constants).**

- see 1.d and 1.e, we push them on to stack as needed.

b. **Accessing values of scalar variables**
The value of a scalar variable is stored at some physical address on the stack so we can use the ADDR instruction to compute the address of the variable and a LOAD to put it on to the top of the stack.

```
ADDR lexlevel offset_from_base
LOAD
```

c. **Accessing array elements**
We compute the the offset and add that to the base address of array. Do either a load or a store afterward. If the index does not start from 0, subtract the base index and force it to start from 0.
Addressing of 1D array element

```
eval index1
% if lower!=0 then adjusted index: PUSH -lower; ADD
ADDR lexlevel offset_from_base
ADD
```

Addressing of 2D array element

```
eval index1
% if lower1!=0 then adjusted index: PUSH -lower1; ADD
PUSH 2nd_dimension_size
MUL
eval index2
% if lower2!=0 then adjusted index: PUSH -lower2; ADD
ADD
ADDR lexlevel offset_from_base
ADD
```

Load element

```
addr element
LOAD
```

Store value

```
addr element
eval result_value
STORE
```

d. **Arithmetic operators +, -, *, and /**

Arithmetic operators are implemented by recursively computing the two operands and then simply using an arithmetic instruction to compute the result of the operator applied to the result of computing the two operands.

Addition(expr1 + expr2)

```
eval expr1
eval expr2
ADD
```

Subtraction(expr1 - expr2)

```
eval expr1
eval expr2
SUB
```

Multiplication(expr1 * expr2)

```
eval expr1
eval expr2
MUL
```

Division(expr1 / expr2)

```
eval expr1
eval expr2
DIV
```

Unary negation (-expr1)

```
eval expr1
NEG
```

e. **Comparison operators < , <=, =, ! =, >=, >**

All of these operators can be implemented by first recursively computing the operands and then using a combination of LT, SWAP, EQ, and unary_not(3f) instructions.

Less than (expr1 < expr2)

```
eval expr1
eval expr2
LT
```

Less than or equal (expr1 <= expr2)

```
eval expr1
eval expr2
SWAP
LT
NOT
```

Equals (expr1 = expr2)

```
eval expr1
eval expr2
EQ
```

Not equals (expr1 != expr2)

```
eval expr1
eval expr2
EQ
NOT
```

Greater than or equal (expr1 >= expr2)

```
eval expr1
eval expr2
LT
NOT
```

Greater than (expr1 > expr2)

```
eval expr1
eval expr2
SWAP
LT
```

f. **Boolean operators & , |, !**
We use short circuit semantics for AND and OR operators

And (expr1 & expr2)

```
eval expr1
BF LABEL_ELSE
eval expr2
BR LABEL_END
anchor LABEL_ELSE
PUSH MACHINE_FALSE
anchor LABEL_END
```

Or (expr1 | expr2)

```
eval expr1
BF LABEL_ELSE
PUSH MACHINE_TRUE
BR LABEL_END
anchor LABEL_ELSE
eval expr2
anchor LABEL_END
```

Unary not (!expr)

```
eval expr
PUSH MACHINE_FALSE
EQ
```

g. **Anonymous functions**

We embed the anonymous function's local the the outer scope, which essentially makes anonymous a minor scope. And we directly execute the statements when we are "calling" the function, and at the end we execute the expression which will leave the result on the top of stack.

3. Functions and procedures

a. **Activation record for functions and procedures**

```
============ callee msp (top of stack)
expression temps
============ callee msp after reserve space
locals
============ DISPLAY[current_lexlevel]
display
============ caller msp (before calling)
return addr
params n
...
params 1
============ caller msp (end of calling)
[result holder for function]
```

b. **Procedure and function entrance code (activation record setup, allocation of local variables, etc)**

```
ADDR current_lexlevel 0 % save display[current_lexlevel]
PUSHMT
SETD current_lexlevel % setup display
RESERVE majorscope.size() % reserve space for locals
exec body
```

c. **Procedure and function exit code (cleanup, return values, etc)**

store the result for function (given that the result value is on top of the stack)

```
ADDR current_lexlevel -3-routine.paramcount()
SWAP
STORE
```

return to the parent

```
PUSHMT
ADDR current_lexlevel 0
SUB
POPN % destroy the local variables
SETD current_lexlevel % restore display
BR % return to outer scope
```

We emit an error message at the end of the function body that will run if the function ends without returning a value.

```
eval put "Error"
HALT
```

d. **Parameter passing**

push parameter on to the stack one by one, we will use display offset to access them

```
eval param1
... % push parameters to the stack
eval paramn
```

e. **Function calls and function value return**

For caller, push a placeholder for return value and do a normal procedure call.

```
PUSH UNDEFINED % place holder
eval parameters
CALL function
```

For callee, store the return value to that place holder position. (see 3.c)

f. **Procedure calls**

```
eval parameters
CALL procedure
```

g. **Display management strategy**

We use the constant time display update strategy. The update and restore are both done by the callee.

4. Statements

We use forward and backward branches to implement the statements.

a. **Assignment statements(var <= expr)**

```
addr var
eval expr
STORE
```

b. **If statements (if cond then body_then else body_else end)**

```
eval cond
BF LABEL_ELSE
exec body_then
BR LABEL_END
anchor LABEL_ELSE
exec body_else
anchor LABEL_END
```

c. **while and loop statements (while cond do body end; loop body end)**

Note that we need to attach the LABEL_END to the embedded statements.

```
anchor LABEL_BEGIN
eval cond % if ‘while’ loop
BF LABEL_END % if ‘while’ loop
exec body
BR LABEL_BEGIN
anchor LABEL_END
```

d. **Exit statements**

Note that the allocation of local variable is done in major scope, so we don’t need to do anything to balance the stack.

No condition (exit)

```
BR LABEL_END % for the nearest while or loop
```

With condition (exit cond)

```
eval cond
NOT % we branch when the cond is true
BF LABEL_END % for the nearest while or loop
```

e. **Return statements**

See 3.c for the code of the return statement. One potential optimization for code size is to generate only one copy of the return code for the function and branch there.

f. **Get and put statements**

We split compounded get and put statements to single argument statement, this include string, which we split string in to each character.

get var

```
addr var
READI
STORE
```

put expr

```
eval expr
PRINTI
```

put skip

```
PUSH ‘\n’
PRINTI
```

example put string:

put “Hello world!”

```
push ‘H’
PRINTC
push ‘e’
PRINTC
...
push ‘!’
PRINTC
```

example multiple get:

get a, b

```
% get a
addr a
GETI
STORE
% get b
addr b
GETI
STORE
```

5. Everything Else

a. **Main program initialization and termination**

We push the stack pointer onto the stack and set the display for the main scope. Then we execute the body of the main scope.

```
PUSHMT
SETD current_lexlevel % setup display
RESERVE majorscope.size() % reserve space for locals
exec body
HALT
```

b. **Any handling of scopes not described above**

- Major scope
    - when entering a scope we make duplicate n copy of UNDEFINED
    - where n is the size of storage need in the scope
- Minor scope
    - we consider minor scope's storage requirement as a part of major scope's

c. **Other relevant information**

We use labels to indicate a address, and we only resolve the label when we are emit binary to the machine.