CS 241 Lecture 20

Graham Cooper

July 15th, 2015

Evaluating Expressions

tests

```
test -> expr1 < expr2

code(test) =
code(expr1) ($3 <- expr1)
add$6, $3, $0
code(expr2) ($3 <- expr2)
slt $3, $6, $3
```

Treat test $\rightarrow$ expr1 $>$ expr2 the same

```
test -> expr1 != expr2
code(test) =
code(expr1) ($3 <- expr1)
add $6, $3, $0 ($6,<- expr1)
code(expr2) ($3 <- expr2)
slt $5, $3, $6 ($5 = $3 < $6)
slt $6, $6, $3 ($6 = $6 < $3)
add $3, $5, $6 ($3 = $6 OR $7)

test -> expr1 == expr2

treat as NOT (expr1 != expr2)
- as above and then (recall $11 = 1)
sub $3, $11, $3 ($3 <- 1 - $3)
```

Leaving $<=$ and $>=$ as excercises

IF

```
stmt -> if test { stmts1 } else {stmts2}
code(stmt) =
code(test) ($3 <- test)
beq $3, $0, else
```

```
code(stmts1)
else: beq $0, $0, endif
code(stmts2)
endif:
```

WE need to generate unique labels

- keep a counter x
- use x, endifx, truex for label names
- incrememnt x for each new if-statement

While:

```
stmt -> while(test){stmts}
-use a counter Y
code(stmt) =
loopY:
code(test) ($3 <- test)
code(stmts)
beq $0, $0, loopY
doneY:
```

Done assn 9

Advice: Generate comments along with mips instructions

Pointers:

Need to do seven?? things

1. NULL
2. Dereferencing
3. address-of
4. Pointer Comparison

5. Pointer Arithmetic
6. alloc/dealloc
7. assignment through pointers

NULL

- could use 0
- Use 1 (not a multiple of 4)
- If 1 is dereferenced, the machine will crash

```
factor -> NULL
code(factor) = add $3, $11, $0 ($3 <- 1)
```

Dereference

```
factor -> *expr
code(factor =
code(expr) ($3 <- expr)
lw $3, 0($3)
```

comparisons

- same as UNSIGNED int comparisons
- (there are no negative pointers)
- use sltu instead of slt

Type Checking 1

So when generating code for:
test $\rightarrow$ expr1 $<$ expr2 etc...
use slt if expr1, expr2 : int
use sltu if expr1, expr2 : int*
- rerun "typeof" (or just on one of them since the types will be the same)

TypeChecking Better

- add a "type" field to each tree node
- make "typed" procedure record each node's type (if it has one) in the node itself
- then the type will be available if needed

Pointer Arithmetic

```
expr1 -> expr2 + term
expr1 -> expr2 - term
```

- the exact meaning depends on the types involved

Addition

```
expr1 -> expr2 + term
if expr2, term : int as before (just add)
if expr2 : int*, term : int
- means expr2 + (4 * term)
code(Expr1) =
code(expr2)
push($3)
code(term) ($3 <- term)
pop($5)
mult $3, $4
mflo $3
add $3, $5, $3

if expr2: int term: int* means (4 * expr2 + term)
```

Subtraction

```
expr1 -> expr2 - term
if expr2 : int, term : int - as before
if expr2 : int*, term : int - means expr2 - 4 * term
- similar to int* + int case
```

```
if expr2 : int*, term : int*
- means (expr2 - term)/4

code(expr1) =
code(expr2)
push($3)
code(term)
pop($5)
sub $3, $5, $3
div $3, $4
mflo $3
```

Assignment

Assnment through pointer dereference

LHS = address at which to store the value RHS = value

```
stmt -> ID = expr - already done
*factor = expr
code(stmt) = code(factor) ($3 <- factor = the address)
push($3)
code(expr) ($3 <- expr)
pop($5) ($5 <- factor)
sw $3, 0($5)
```

Address-of

&lvalue lvalue is ID, or STAR factor

```
if expr = ID
code(&ID) =
lis $3
.word ____ (look up ID in symtable)
add $3, $3, $29

if expr = STAR factor:
code(&*factor) = code(factor)
```

New/Delete

- runtime environment
- allocation module alloc.merl
- link to your assembled output along with print.merl
- MUST link it last