CS 241, Lecture 18: Code Generation (Continued)

1 MERL, IF, and WHILE

- They covered this today in lecture 18, but we covered it in lecture 17 because I thought they were faster.
- Note that I totally missed both lectures... sleep is nice. I could be missing some info.

2 Pointers

- We need to support all of the following for pointers:
  - NULL
  - Dereferencing
  - Address of
  - Comparisons
  - Pointer arithmetic
  - Allocating and deallocating heap memory
  - Pointer comparisons
  - Pointer assignments and pointer access

2.1 NULL

- You would *think* that we would make NULL 0x0. But this is not valid - 0x0 is a valid memory address!
- We would like NULL to crash if we dereference (which will not happen with 0x0), so we pick a value for NULL that is not word-aligned — not a multiple of 4! Thus, we pick the value of 0x1.
- The code generation is as follows:

```
factor → NULL
code(factor) = add s3, s0, s11
```

- Note that attempting to use NULL with lw or sw will crash since MIPS is expecting a word-aligned address.

2.2 Dereferencing

- Consider code like `factor1 → STAR factor2`. The value in `factor2` is a pointer (otherwise there's a type error, as per A8).
- What we want to do is access the value at `factor2` and load it somewhere, in this case, \$3.
- Since `factor2` is a memory address, we want to load the value in the memory address *at* \$3 and store it in \$3.
- The code generation is as follows:

  ```
  code(factor1) = code(factor2) + lw s3, 0(s3)
  ```

2.3 Addresss of

- Recall an lvalue is something that can appear as the LHS of an assignment rule.
- Note that we have a rule `factor → AMP lvalue`.
- Why this over `factor1 → AMP factor2`? Well, using `factor2` gives us issues if we try to dereference an integer constant, which is a valid factor!
- So, when we have an lvalue, how do we find where it is in memory? Symbol table!
- The code generation is as follows:

  ```
  factor → AMP lvalue
  code(factor) = lis s3 +
                 .word offset +
                 add s3, s3, s29
  ```

  where `offset` is the offset found in the symbol table for the lvalue.
- What if in `factor → AMP lvalue`, the lvalue is actually a `STAR expr`? That is, suppose we said `&*ptr` in C.
- Well, we can just say that `code(factor) = code(expr)`.
- Recall we had code for:

  ```
  statement → lvalue BECOMES expr SEMI:

  code(statement) = code(expr) + ;s3 ← expr
                    sw s3, offset(s29)
  ```

  where `lvalue` is an ID and `offset` is the offset for the ID that is lvalue.
- How can we modify this if lvalue is of the form `STAR factor`?

  ```
  code(statement) = code(expr) +
                    push(s3) +
                    code(lvalue) +
                    pop(s5) + ;s5 ← expr
                    sw s5, 0(s3)
  ```

2.4 Comparisons

- The same as integer comparisons with one exception — pointers cannot be negative and so, `slt` is not what we want to use, but rather, `sltu` (the thing that we never really used).
- But given `test` → `expr COMP expr`, do we use `slt` or `sltu`?
- Just check the type of the first `expr`; by Assignment 8 the two must be the same or else we would have failed the code!
- Tip: Augment the tree node class to include the type of the node itself if it has one. You could also use the symbol table.

2.5 Pointer Arithmetic

- Recall for addition and subtraction, we have several contracts.
- For **int + int** or **int - int**, we proceed as before - but there are four more contracts that use pointers!
- For **int* + int**:

  ```
  expr1 → expr2 + term:

  code(expr1) = code(expr2) +
                push(s3) +
                code(term) +
                mult s3, s4 +
                mflo s3 +
                pop(s5) +  ;s5 <- expr
                add s3, s5, s3
  ```

  Recall we are comuting a different memory address equal to `expr2 + 4 × term`.
- For **int + int***:

  ```
  expr1 → expr2 + term:

  code(expr1) = code(expr2) +
                mult s3, s4 +
                mflow s3 +
                push(s3) +
                code(term) +
                pop(s5) +   ;s5 <- expr
                add s3, s5, s3
  ```

  Which corresponds to `4 × expr2 + term`
- For **int* - int**:

  ```
  expr1 → expr2 - term:
  ```

```
code(expr1) = code(expr2) +
              push(s3) +
              code(Term) +
              mult s3, s4 +
              mflow s3 +
              pop(s5) +    ;s5 <- expr
              sub s3, s5, s3
```

Which is just `expr2 - 4 × term`

- For `int* - int*`:

  `expr1 → expr2 - term:`

```
code(expr1) = code(expr2) +
              push(s3) +
              code(term) +
              pop(s5) +     ;s5 <- expr
              sub s3, s5, s3 +
              div s3, s4 +
              mflo s3
```

Which is just `(expr2 - term) / 4`

2.6 Memory Allocation

- Lastly, we need to handle the commands `new, delete`
- Thankfully, we outsource this to a library, called `alloc.merl`
- This *must* be linked last in our output! That is:

```
./wlp4gen < source.wlp4i > source.asm
cs241.linkasm < source.asm > source.merl
linker source.merl print.merl alloc.merl > exec.mips
```

- Our prologue now has the imports:
  * `.import init`
  * `.import new`
  * `.import delete`
- The command `init` initalizes the heap, and must be called at the beginning!
- `new` finds the number of new words needed as specified in $1.
- It will return a pointer to memory at the beginning of this many words in $3 if successful. Otherwise, it places 0 in $3.
- The code looks a bit like:

```
code(new int [expr]) = code(expr) +
                       add s1, s3, s0 +
                       call(new) +
                       bne s3, s0, 1 +
                       add s3, s11, s0
```

Note that the last line sets \$3 to `NULL` and executes iff the call to new fails!

- For `delete`, it requires that \$1 is a memory address to be deallocated.
- The code looks like:

```
code(delete [] expr) = code(expr) +
                       beq s3, s11, skipDelete: +
                       add s1, s3, s0 +
                       call(delete) +
                       skipDelete:
```

Again, like **`if`**, **`while`**, we need to count the deletion labels. We skip delete if attempting to delete a null pointer.