Warm Up Problem

How would we encode `switch` statements in our translation of WLP4?

CS 241 Lecture 19

Code Generation Continued Again!

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Pointers

At last we have reached pointers. We need to support all of the following:

- NULL
- Allocating and deallocating heap memory
- Dereferencing
- Address of
- Pointer arithmetic
- Pointer comparisons
- Pointer assignments and pointer access

Here we go!

NULL

The first and obvious choice for us for the value of `NULL` is `0x0`.
Why is this a problem for us?

NULL

The first and obvious choice for us for the value of `NULL` is `0x0`. Why is this a problem for us?

In our language, `0x0` is a valid memory address! (Note, in say C programming, the operating system prevents access to `0x0`).

So we would like our `NULL` to crash if attempting to dereference. We can force this by picking a value for `NULL` that is not **word-aligned**.

NULL

The first and obvious choice for us for the value of `NULL` is `0x0`. Why is this a problem for us?

In our language, `0x0` is a valid memory address! (Note, in say C programming, the operating system prevents access to `0x0`).

So we would like our `NULL` to crash if attempting to dereference. We can force this by picking a value for `NULL` that is not **word-aligned**.

Word-aligned for us means that the address is a multiple of 4. The smallest value for `NULL` therefore that we can (and will) use is `0x1`.

Code

`factor` $\rightarrow$ `NULL`

```
code(factor) = add $3, $0, $11
```

Note that attempts to use NULL with either `lw` or `sw` will result in a crash since MIPS is expecting a word-aligned address.

Dereferencing

What about dereferencing? Consider `factor1` $\rightarrow$ `STAR factor2`

Dereferencing

What about dereferencing? Consider `factor1` $\rightarrow$ `STAR factor2`

The value in `factor2` is a pointer (otherwise you should have a type error!) What we want is to access the value at `factor2` and load it somewhere. As always we load into $3. Since `factor2` is a memory address, we want to load the value in the memory address at $3 and store in $3.

```
code(factor1) = code(factor2) + lw $3, 0($3)
```

Address Of

Recall that an `lvalue` is something that can appear as the LHS of an assignment rule. Note that we have a rule `factor` $\rightarrow$ `AMP lvalue`. Why do we have this instead of `factor1` $\rightarrow$ `AMP factor2`? (Reworded; what values for `factor2` give us issues?)

Address Of

Recall that an `lvalue` is something that can appear as the LHS of an assignment rule. Note that we have a rule `factor` $\rightarrow$ `AMP lvalue`. Why do we have this instead of `factor1` $\rightarrow$ `AMP factor2`? (Reworded; what values for `factor2` give us issues?)

Suppose we have an `ID` value `a`. How do we find out where `a` is in memory?

Address Of

Recall that an `lvalue` is something that can appear as the LHS of an assignment rule. Note that we have a rule `factor` $\rightarrow$ `AMP lvalue`. Why do we have this instead of `factor1` $\rightarrow$ `AMP factor2`? (Reworded; what values for `factor2` give us issues?)

Suppose we have an `ID` value `a`. How do we find out where `a` is in memory?

In our symbol table! We can load the offset first and then use this to find out the location.

Code for Address of

`factor` $\rightarrow$ `AMP lvalue` when `lvalue` is an ID?

```
code(factor) = lis $3
             + .word offset
             + add $3, $3, $29
```

where `offset` is the offset found in the symbol table for `lvalue`.

Code for Address of

What about if in `factor1` $\rightarrow$ `AMP lvalue` we have that `lvalue` is `STAR factor2`?

Code for Address of

What about if in `factor1` $\rightarrow$ `AMP lvalue` we have that `lvalue` is `STAR factor2`? Notice here that the code for `factor1` is just the same as the code for `factor2`!

```
code(factor1) = code(factor2)
```

Revisiting Old Ideas

Recall:

The production rule `statement` $\rightarrow$ `lvalue BECOMES expr SEMI` when `lvalue` is an `ID` becomes

```
code(statement) = code(expr)     ; $3 <- expr
                + sw $3, offset($29)
```

where `offset` is the offset for the `ID` that is `lvalue`.

How do we modify this if `lvalue` is of the form `STAR factor`?

New Translation

The production rule `statement` $\rightarrow$ `lvalue BECOMES expr SEMI` when `lvalue` is `STAR factor` is

```
code(statement) = code(expr)
                + push($3)
                + code(factor)   ;no *
                + pop($5)        ;$5 <- expr
                + sw $5, 0($3)
```

Comparisons

Comparisons work the same as with integers with one exception. What is this exception?

Comparisons

Comparisons work the same as with integers with one exception. What is this exception?

Pointers cannot be negative and so `slt` is not what we want to use! We should use `sltu` instead. Given

$$\texttt{test} \rightarrow \texttt{expr COMP expr}$$

How can we tell which of `slt` or `sltu` to use?

Comparisons

Comparisons work the same as with integers with one exception. What is this exception?

Pointers cannot be negative and so `slt` is not what we want to use! We should use `sltu` instead. Given

$$\texttt{test} \rightarrow \texttt{expr COMP expr}$$

How can we tell which of `slt` or `sltu` to use? Simply check the type of the first `expr`! (Note from Assignment 8 you are guaranteed that the types of the two expressions are the same!)

Pro Tip: You might want to augment your tree node class to include the type of the node itself (if it has one). Could also use symbol table if desired.

Pointer Arithmetic

- Recall for addition and subtraction we have several contracts.
- For `int + int` or `int - int`, we proceed as before. This leaves 4 contracts we need to consider that use pointers

Addition

`expr1` $\rightarrow$ `expr2 + term` where `type(expr2) == int *` and `type(term) == int`:

Addition

`expr1` $\rightarrow$ `expr2 + term` where `type(expr2) == int *` and `type(term) == int`:

```
code(expr1) = code(expr2)
            + push($3)
            + code(term)
            + mult $3, $4
            + mflo $3
            + pop($5)          ;$5 <- expr
            + add $3, $5, $3
```

Recall that we're computing a different memory address corresponding to `expr2` $+ 4 \cdot$ `term`.

Addition

`expr1` $\rightarrow$ `expr2 + term` where `type(expr2) == int` and `type(term) == int *`:

Addition

`expr1` $\rightarrow$ `expr2 + term` where `type(expr2) == int` and `type(term) == int *`:

```
code(expr1) = code(expr2)
            + mult $3, $4
            + mflo $3
            + push($3)
            + code(term)
            + pop($5)          ;$5 <- expr
            + add $3, $5, $3
```

Recall that we're computing a different memory address corresponding to $4 \cdot$ `expr2` $+$ `term`.

Subtraction

`expr1` $\rightarrow$ `expr2 - term` where `type(expr2) == int *` and `type(term) == int`:

Subtraction

`expr1` $\rightarrow$ `expr2 - term` where `type(expr2) == int *` and `type(term) == int`:

```
code(expr1) = code(expr2)
            + push($3)
            + code(term)
            + mult $3, $4
            + mflo $3
            + pop($5)          ;$5 <- expr
            + sub $3, $5, $3
```

Recall that we're computing a different memory address corresponding to `expr2` $- 4 \cdot$ `term`.

Subtraction

`expr1` $\rightarrow$ `expr2 - term` where `type(expr2) == int *` and `type(term) == int *`:

Subtraction

`expr1` $\rightarrow$ `expr2 - term` where `type(expr2) == int *` and `type(term) == int *`:

```
code(expr1) = code(expr2)
            + push($3)
            + code(term)
            + pop($5)          ;$5 <- expr
            + sub $3, $5, $3
            + div $3, $4
            + mflo $3
```

Recall that we're computing a distance between two memory addresses corresponding to (`expr2` $-$ `term`)/4.

Memory Allocation

Lastly, we need to handle calls such as `new` and `delete`.

Memory Allocation

Lastly, we need to handle calls such as `new` and `delete`.

Good news! We're going to outsource this work to a library. For us, we will provide you with a `print.merl` file that you will use to link with your assembled output:

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

Then finally, we can call say

```
mips.twoints exec.mips
```

NOTE: `alloc.merl` *must* be linked last! (Needs to know where end of code is).

Prologue Additions

Now we include

- `.import init`
- `.import new`
- `.import delete`

The command `init` initializes the heap. Must be called at the beginning. Takes a parameter in $2 and initializes the data structure.

- If called with `mips.array` then $2 is the length of the array.
- Otherwise, we want $2 = 0$.

`new`

- Finds the number of new words needed as specified in $1.
- Returns a pointer to memory of beginning of this many words in $3 if successful.
- Otherwise, it places 0 in $3.

new

- Finds the number of new words needed as specified in $1.
- Returns a pointer to memory of beginning of this many words in $3 if successful.
- Otherwise, it places 0 in $3.

```
code(new int [expr]) = code(expr)
                     + add $1, $3, $0
                     + call(new)
                     + bne $3, $0, 1
                     + add $3, $11, $0
```

Note, the last line sets $3 to be NULL and executes if and only if the call to new failed.

`delete`

- Requires that $1 is a memory address to be deallocated.

delete

- Requires that $1 is a memory address to be deallocated.

```
code(delete [] expr) = code(expr)
                     + beq $3, $11, skipDelete:
                     + add $1, $3, $0
                     + call(delete)
                     + skipDelete:
```

Again, as with `if` and `while` statements, you will need to number your deletion labels. We skip delete when attempting to delete a null pointer.

Procedures

We need to now deal with multiple function calls. There are a bunch of factors to consider:

- Who should save which registers? The caller? The callee (the function being called)?
- What do functions need to update/initialize?
- How do we have to update our symbol table?
- How do other functions differ from `wain`?
- How do we handle parameter passing?

Recall: `wain`

What do we need to do for `wain`?

- Load `import`, `init`, `new`, `delete`
- Initialize $4, $10, $11. (Don't need this for other procedures!)
- Call `init`
- Save $1, $2 (these are our parameters).
- Reset stack (at end)
- Call `jr $31` (at end)

General Procedures

How does the previous slide change for a generic procedures?

General Procedures

How does the previous slide change for a generic procedures?

- Don't need any imports
- Need to update $29
- Save registers
- Restore registers and stack and `jr $31`

Saving and Restoring Registers

Who is responsible for saving and restoring registers? There are two options:

Definition

The **caller** is a function `f` that calls another function `g`.

The **callee** is a function `g` that is being called by another function `f`.

Note that `f` could be `g`. One of the above two must be saving registers. Which one should we pick?