Lecture 20
Code Generation for Pointers in WLP4

CS 241: Foundations of Sequential Programs
Spring 2017

Troy Vasiga et al. (where Dan Holtby ∈ al)
University of Waterloo
Recap

What we know to this point:

▶ **semantic/context-sensitive analysis phase:**
  ▶ symbol table (to keep track of variable/procedure declarations and type)
  ▶ ensure that variables/procedures declared exactly once
  ▶ ensure that types match
  ▶ PRO-TIP: remember the type in your parse tree at *every* node

▶ **code generation phase:**
  ▶ each grammar rule will represent one function/procedure/step in our compilation process
  ▶ the parse tree indicates which subtrees need to produced the code
  ▶ add comments to your generated (MIPS) code
  ▶ it is easy to destroy register values if you are not careful
Compiling WLP4 to MIPS means that MIPS can do everything that a WLP4 program can do.

Recall that MIPS can be executed in two flavours:

- mips.twoints
- mips.array
- Allows arrays to be declared, initialized, allocated and destroyed
  - Nothing more than a pointer: starting address in $1. Size is stored in $2.
- Also, we can use pointers without having arrays.
A first example

```c
int wain(int a, int b) {
    int *x = NULL;
    int y = 7;
    x = &y;
    return (*x);
}
```

What does this program do?

How do we do this in MIPS?
As mentioned in the previous lecture, we use the grammar rules to tell us how to generate the appropriate MIPS code.
factor $\rightarrow$ NULL

(Note: we want dereferencing of NULL to crash.)

Option 1: C style
add $3$, $0$, $0$
Now you need to check every time you dereference a pointer, and crash if it’s NULL. Adds instructions.

Option 2: Exploit alignment
slt $3$, $0$, $4$
Now NULL means 1, and if you lw / sw on address 1, you crash due to an alignment issue.
factor → STAR factor

code(factor) puts the value of the expression into register 3. STAR factor is supposed to put the value stored at that address into register 3, so...
code(factor → STAR factor₁) = code(factor₁) + “lw $3, 0($3)”

What if you need to check for null?
“bne $0, $3, 1 ; if not NULL, skip next instruction”
“.word 0xDEADBEEF ; invalid opcode crash”
(or just “lw $1, 1($0)” for an alignment error again)
Using Address-of (&)

factor → AMP lvalue

Why is the rule written this way, and not “factor → AMP factor”?

Look at the grammar for what lvalue can be:

lvalue → ID

lvalue → STAR factor

lvalue → LPAREN lvalue RPAREN
Generating code for &

factor → AMP lvalue

Look ahead at the three cases:

- lvalue → LPAREN lvalue RPAREN
- lvalue → ID
- lvalue → STAR expr

We dealt with two of those already for assignment statements. The STAR is new though
Assignment to pointer dereference

(Note: I’ve fixed/updated the next two slides because they were rough)
Let’s step back and look at the rule:

statement → lvalue BECOMES expr SEMI

What did we do here when lvalue could only be an ID?
sw $3, i($29) ; where i is the “location” of the ID in the local symbol table.

This won’t work for ampersand though*. We have to do it in 3 instructions (have to add the offset i together with register 29)

* - we don’t have addi in this course
What about Ampersand

code(lvalue) returns MIPS code that puts the **address** of the lvalue into $3$
code(lvalue → ID) needs to add the offset with $29$
code(lvalue → STAR expr) is just code(expr)

“AMP lvalue” is supposed to put the **address** of the lvalue into $3$...sounds familiar

code(AMP lvalue) = code(lvalue)

Assignment statements use code(lvalue), too. For the STAR rule anyway. For ID you might want to stick to the A9 solution because of efficiency.
A “simple” WLP4 Program

```c
int wain(int *a, int n) {
    return *a;
}
```

What does this program do?

How can we do this in MIPS?
What mips.array is doing

@linux.cs [1] % cat ex1.wlpp | wlp4scan | wlp4parse > ex1.wlp4i
@linux.cs [2] % ./wlp4gen < ex1.wlp4i > ex1.mips
@linux.cs [3] % java mips.array ex1.mips
Enter length of array: 3
Enter array element 0: 10
Enter array element 1: 14
Enter array element 2: 19

Allocates memory on the heap, then calls wain with the address and the size as parameters.
A second example

```c
int wain(int *a, int n) {
    return *(a+1);
}
```

Meaning:

Return the second element of the input array.

A word about sugar:
Back to compilation (A10P2)

factor → NEW INT LBRACK expr RBRACK
statement → DELETE LBRACK RBRACK expr SEMI

How do these rules affect compilation of WLP4 programs?

Prologue:

Calling new: (Recall, we want dereferencing NULL to crash.)

Calling delete: (Hint: don’t delete NULL.)
Pointer arithmetic (A10P3)

expr → expr PLUS term
Pointer arithmetic (A10P3)

expr → expr MINUS term
As we know, since semantic analysis passed, comparisons like:

\[ \text{test} \rightarrow \text{expr1 LT expr2} \]

must have the same types for expr1 and expr2

What (if anything) needs to change in how we compare int* values?