Lecture 21

Code Generation for Pointers in WLP4

CS 241: Foundations of Sequential Programs
Fall 2014

Troy Vasiga et 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

WLP4 and MIPS

- 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`

WLP4

- 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.

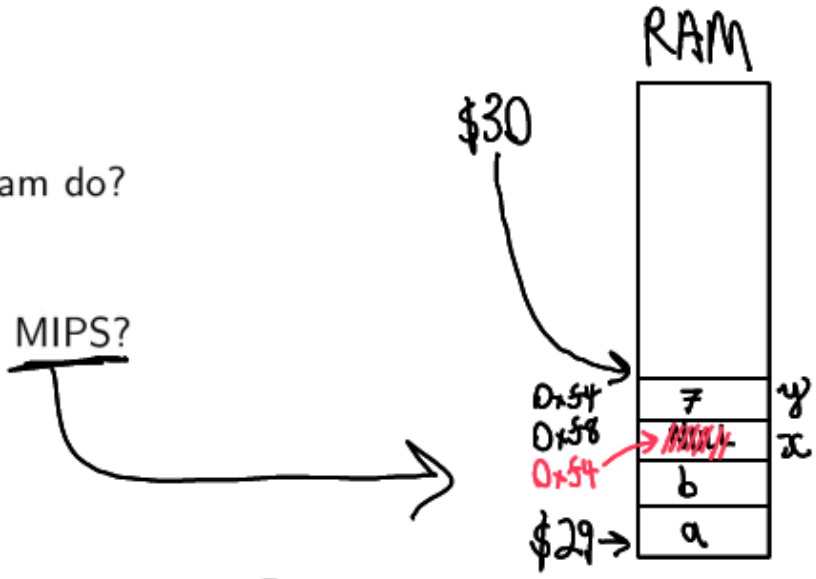
A first example

```
int wain(int a, int b) {
① int *x = NULL;
② int y = 7;
③ x = &y;
④ return (*x);
}
```

What does this program do?

returns 7

How do we do this in MIPS?



Syntax-directed translation

As mentioned in the previous lecture, we use the grammar rules to tell us how to generate the appropriate MIPS code.

Pointer basics (A10P1)

factor → NULL

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

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

↑ not on a word boundary

factor$_1$ → STAR factor$_2$

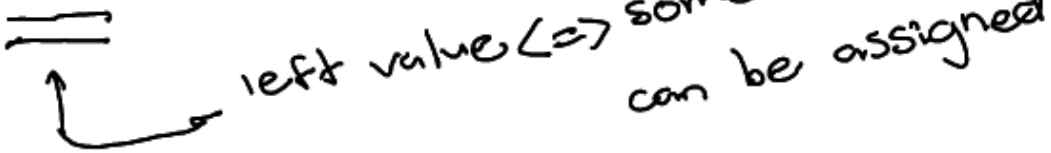
code(factor$_1$) = code(factor$_2$) // $3 = address

lw $3, 0($3)

put the value located at the address

Using Address-of (&)

factor → AMP lvalue



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

I want to disallow &NULL, &7

Look at the grammar:

lvalue → ID] => x=y; ✓

lvalue → STAR factor] => *x=y; ✓

lvalue → LPAREN lvalue RPAREN => (x)=y; ✓

Notice:

3+4=7; ✗

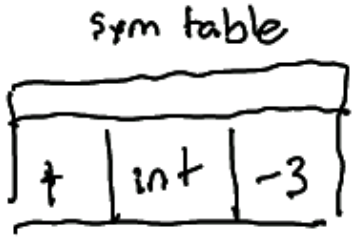
x+2=7; ✗

x+y=7; ✗

Generating code for &

factor → AMP lvalue

Look ahead at the three cases:



(1) if lvalue == ID: code(factor) = lis $3 (&t)

.word <relative location of t>

actual address of variable t in RAM ⇒ add $3, $29, $3

(2) if lvalue == *factor$_2$: code(factor) = code(factor$_2$)

&(*x) = x

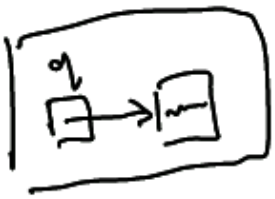
(3) exercise

Assignment to pointer dereference

lvalue → STAR factor

Let's step back and look at the rule:
statement → lvalue BECOMES expr SEMI → already dealt with on A9, if expr is of type int and lvalue is ID

A10



*q = mm ;

code statement = code(expr) // $3 has a value
push($3)
code(factor) // $3 has address
pop($5)
sw $5, 0($3)

A "simple" WLP4 Program

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

mips.array

What does this program do?

returns the first element in the array

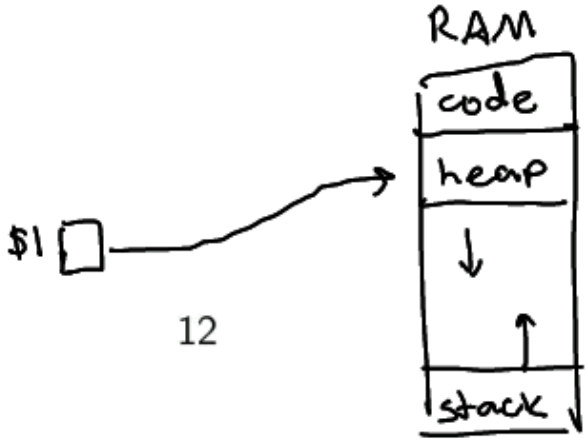
How can we do this in MIPS?

- find the base address for a ($1)
- lw $3, 0($1)

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

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

Meaning:

return the second element

A word about sugar: ← syntactic sugar

$$a[1] \equiv *(a+1) \equiv *(1+a) \equiv 1[a]$$

.import init
.import new
.import delete

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?] link to alloc.merl

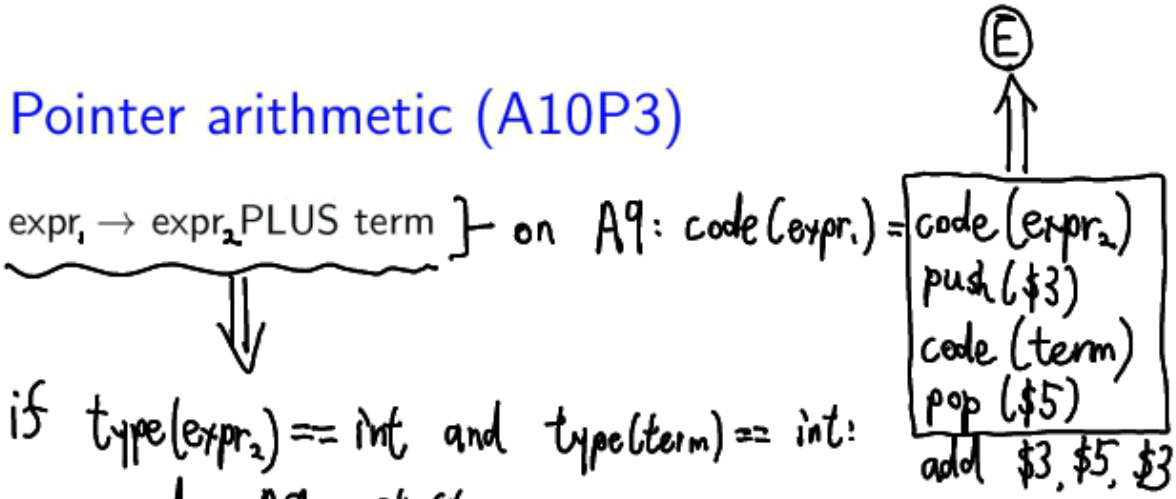
Prologue: call init
↳ initializes the heap
$2 = length of array (wain(int*, int))
$20 (wain(int*,int))

Calling new: (Recall, we want dereferencing NULL to crash.)
↳ $1 = size of array

Notice: new returns 0 if out of memory!
∴ set $3 ← $11

Calling delete: (Hint: don't delete NULL.)
↳ $1 should be the base address

Pointer arithmetic (A10P3)



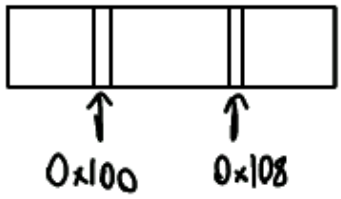
Pointer arithmetic (A10P3)

$expr_1 \rightarrow expr_2$ MINUS term

if type($expr_2$) == int and type(term) == int:
    do A9 stuff

else if type($expr_2$) == int* and type(term) == int:
    code($expr_1$) = (E)
        mult $3, $4
        mflo $3
        sub $3, $5, $3

else if type($expr_2$) == int* and type(term) == int*:
    code($expr_1$) = (E)
        sub $3, $5, $3
        div $3, $4
        mflo $3



Comparison of pointers (A10P4)

As we know, since semantic analysis passed, comparisons like:

test → expr1 LT expr2

must have the same types for expr1 and expr2

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