Lecture 19+20

Code Generation for WLP4MPMP

CS 241: Foundations of Sequential Programs
Fall 2014

WLP4
Minus Pointers
Minus Procedures

Troy Vasiga et al
University of Waterloo

Code Generation (A9/A10/A11)

- Input: parse tree (semantically valid)
  symbol table
- Output: MIPS
  - equivalent program which does the same thing as the WLP4 program
    - return value
    - output

Number of different outputs:

Append:

```
add $1, $1, $0
```

Code Generation Issues

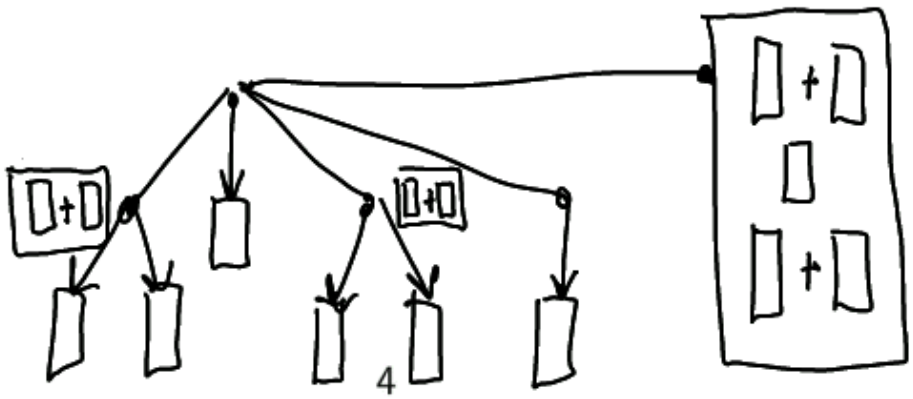
- Correctness → equivalent correct meaning
- Ease of writing compiler
- Efficiency of the compiler → if n lines of WLP4 code ⇒ compile in $O(n)$ time
- Efficiency of the compiled code
  ↳ optimization

Code Generation via Syntax-Directed Translation

Fundamental idea:

- traverse the parse tree and gather information

↳ usually, down then up



First rule

main $\rightarrow$ INT WAIN LPAREN dcl_1 COMMA dcl_2 COMMA...

code(main) = <stuff> "t"
code(dcl_1) "t"
code(dcl_2) "t"
code(dcls) "t"
code(statements) "t"
code(expr) "t"
<other stuff>

Three more simple rules

statements $\rightarrow \epsilon$ code(statements) = ""

expr $\rightarrow$ term code(expr) = code(term)

term $\rightarrow$ factor code(term) = code(factor)

In general:
$\alpha \rightarrow \beta$ code(α) = code(β)

A9P1

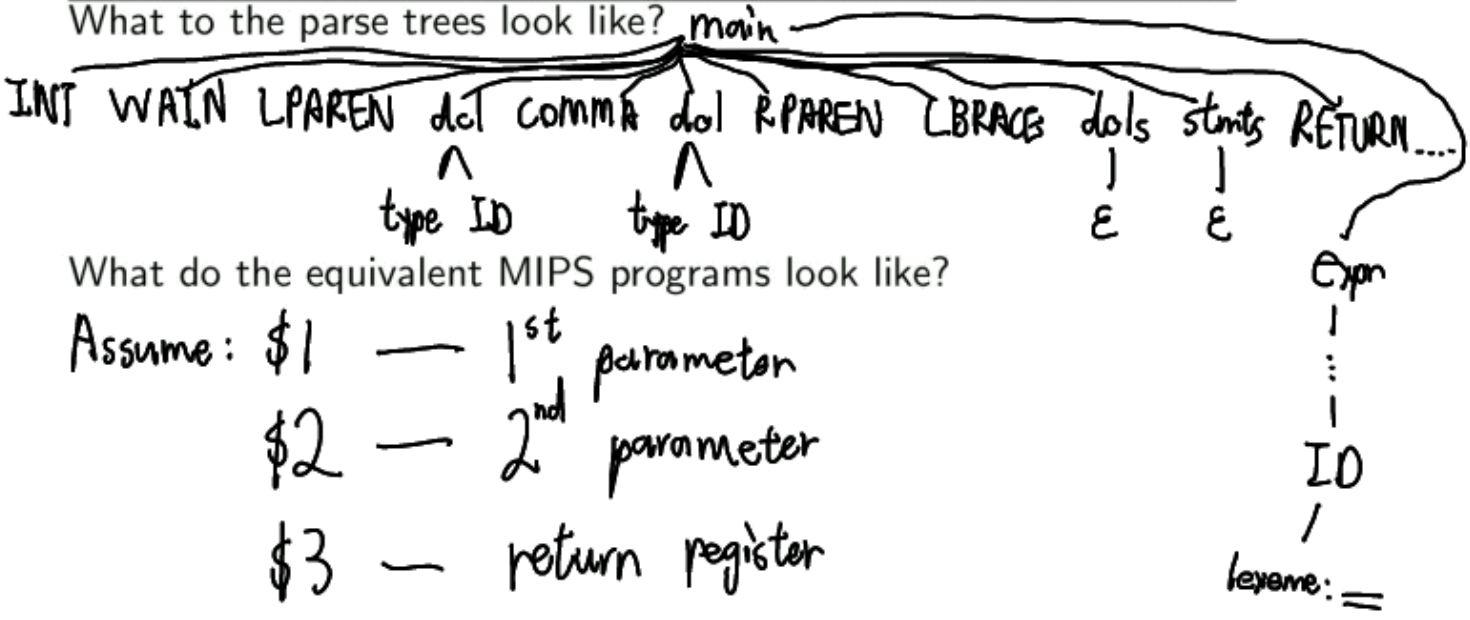
Recall that the input must be *semantically valid*.

What do the WLP4 programs look like?

```
int wain(int a, int b) {
   return a;
}
```
A

```
int wain(int a, int b) {
   return b;
}
```
B

What to the parse trees look like?



What do the equivalent MIPS programs look like?

Assume: $1 — 1st parameter

$2 — 2nd parameter

$3 — return register

A:
```
add $3, $1, $0
jr $31
```

B:
```
add $3, $2, $0
jr $31
```

Changes to the symbol table

name	type	location
a	int	$1
b	int	$2

How to store variables

Option A: Variables in Registers
One variable per register, stored somehow in symbol table.

Problems:

>32 variables

Option B: Variables in RAM using `.word`
Each variable x in WLPP program corresponds to label `x` in MIPS.

Example:

```
code
  :
jr $31
a: .word 0
b: .word 0
c: .word c
```

RAM

(2) costly: get the value of c

```
lis $3
.word c
lw $3, 0($3)
```

Problems:

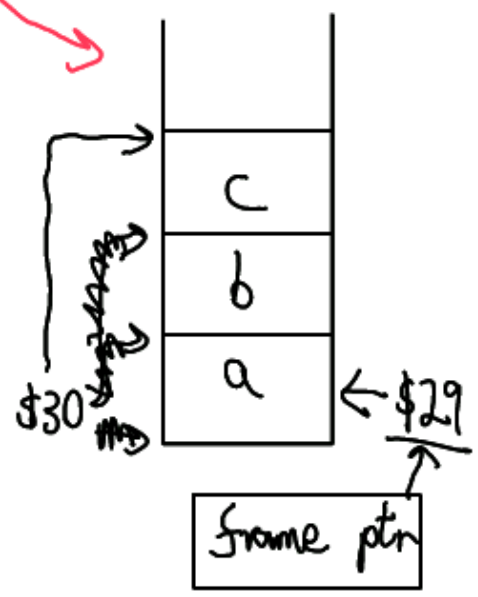
(1) local vs. global ⇒ procedures

Option C

Option C: Variables on Stack

- Suppose we have n variables
- How to get n? → size of symbol table
- Picture in RAM
  $29 + 4 * location
- Where is the ith variable?
  ~~$30~~ + 4 * location
- What about $30 changing?
  - calling a procedure
  - temporary value

name	type	location
a	int	0
b	int	-1
c	int	-2



Some conventions

Remember we have:

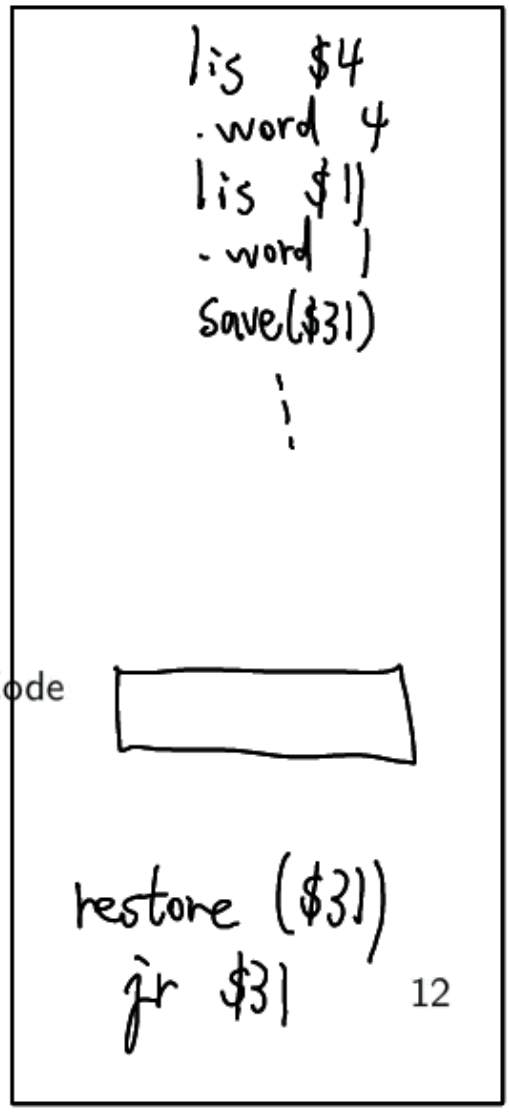
- $0 – 0 false
- $1 – value of 1st parameter (initially)
- $2 – value of 2nd parameter (initially)
- $3 – return value
- $30 – stack pointer
- $31 – return address

plus

- $4 – 4
- $11 – 1
- $29 – frame pointer: bottom of the current stack frame

Code Structure

Prologue



Generated Code

Epilogue

restore ($31)
jr $31

One more rule (A9P2)

factor → LPAREN expr RPAREN

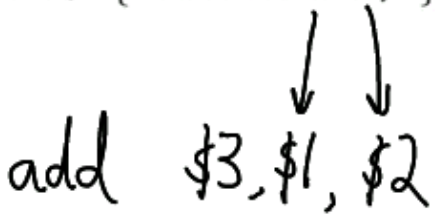
```
int wain(int a, int b){
    return (a);
}
```

- What this gives us:

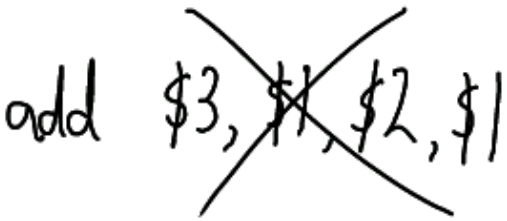
  Same code as before!

A9P3: Full expressions

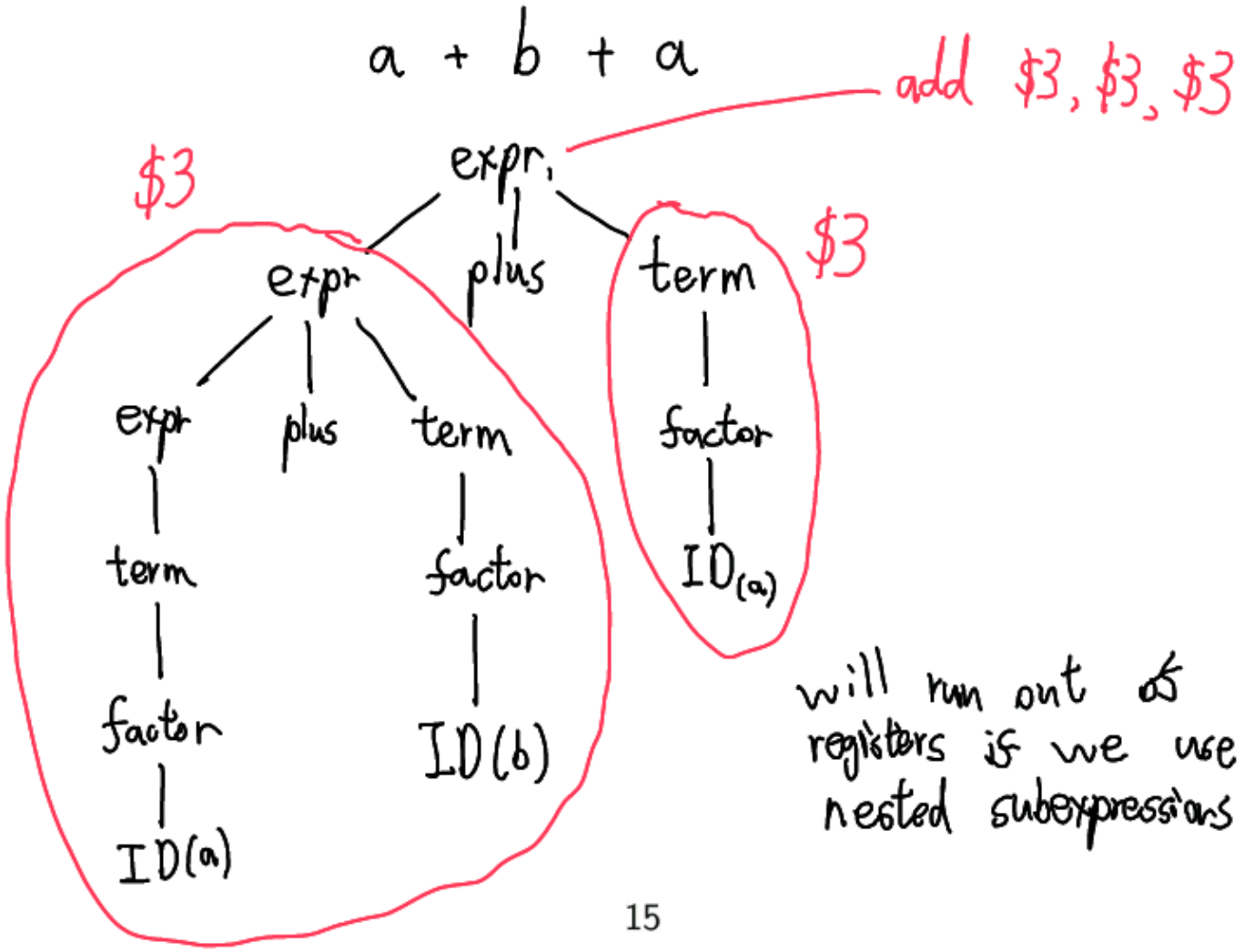
- `int wain(int a, int b) { return a+b; }`



- `int wain(int a, int b) { return a+b+a; }`



A9P3: Parse Tree



A9P3: Pseudocode

$expr_1 \rightarrow expr_2$ PLUS term

code($expr_1$) = code($expr_2$) + // \$3 ← $expr_2$
push(\$3) +
code(term) + // \$3 ← term
pop(\$5) + // \$5 ← $expr_2$
add \$3, \$5, \$3

A9P4: Printing

statements → PRINTLN LPAREN expr RPAREN SEMI

A first attempt:

```
return code(expr) +
       move $3 to $1 +
       call print
```

Problems:

- destroyed $31: ez fix: save it (in prologue)
- where is print?

Where is `print`?

Solution:
`.import print` in Prologue

```
code(statement) = code(expr) +
                  add $1, $0, $3 +
                  lis $5 +
                  .word print +
                  jalr $5 +
```

Problems:

→ $1 gets destroyed

→ expr might be 178392 ← call print once per char.
"1" "7"

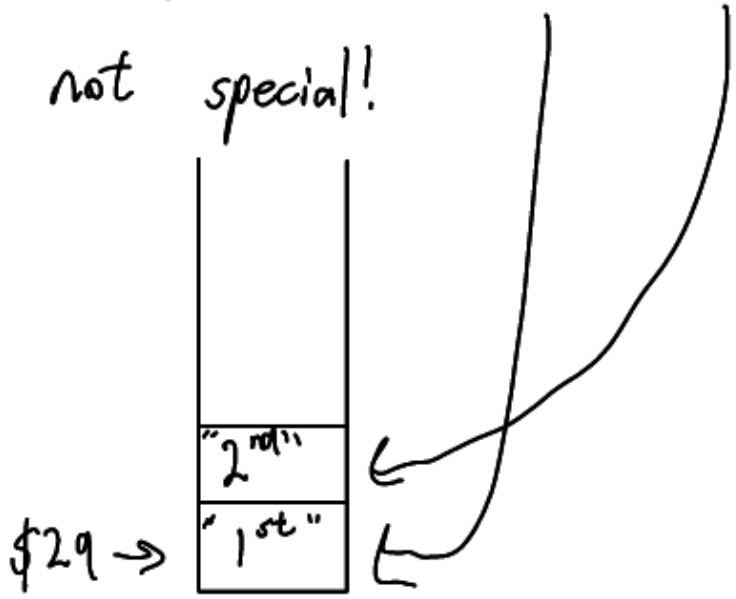
How to not destroy $1?

Three possible solutions (two of which have problems):

- try different calling conventions
  ⇒ make print read its argument in $3
  ] all other code needs changing — Bad Idea
- save + restore each time we use $1] overhead
- destroy $1 ✓

A special note about parameters in $1 and $2

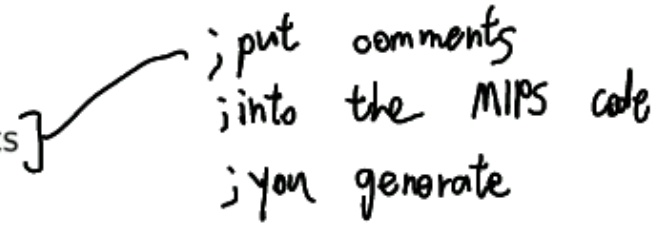
They are not special!



destroy 'em!

Other advice

- comments



- rule-based structure of your code
  - each grammar rule will represent one function/procedure/step in our compilation process
  - the parse tree indicates which subtrees need to produce the code before the "entire" code has been produced
- read the WHOLE assignment before beginning and PLAN ahead
- test, test, test

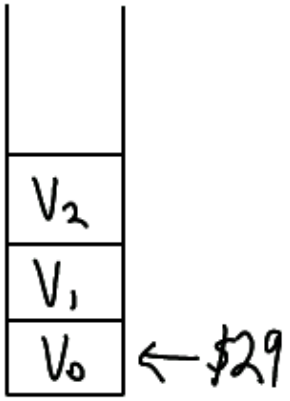
Declarations (A9P5)

Symbol table

V_2	int	-2
V_1	int	-1
V_0	int	0

Rule: dcls → dcls dcl BECOMES NUM SEMI

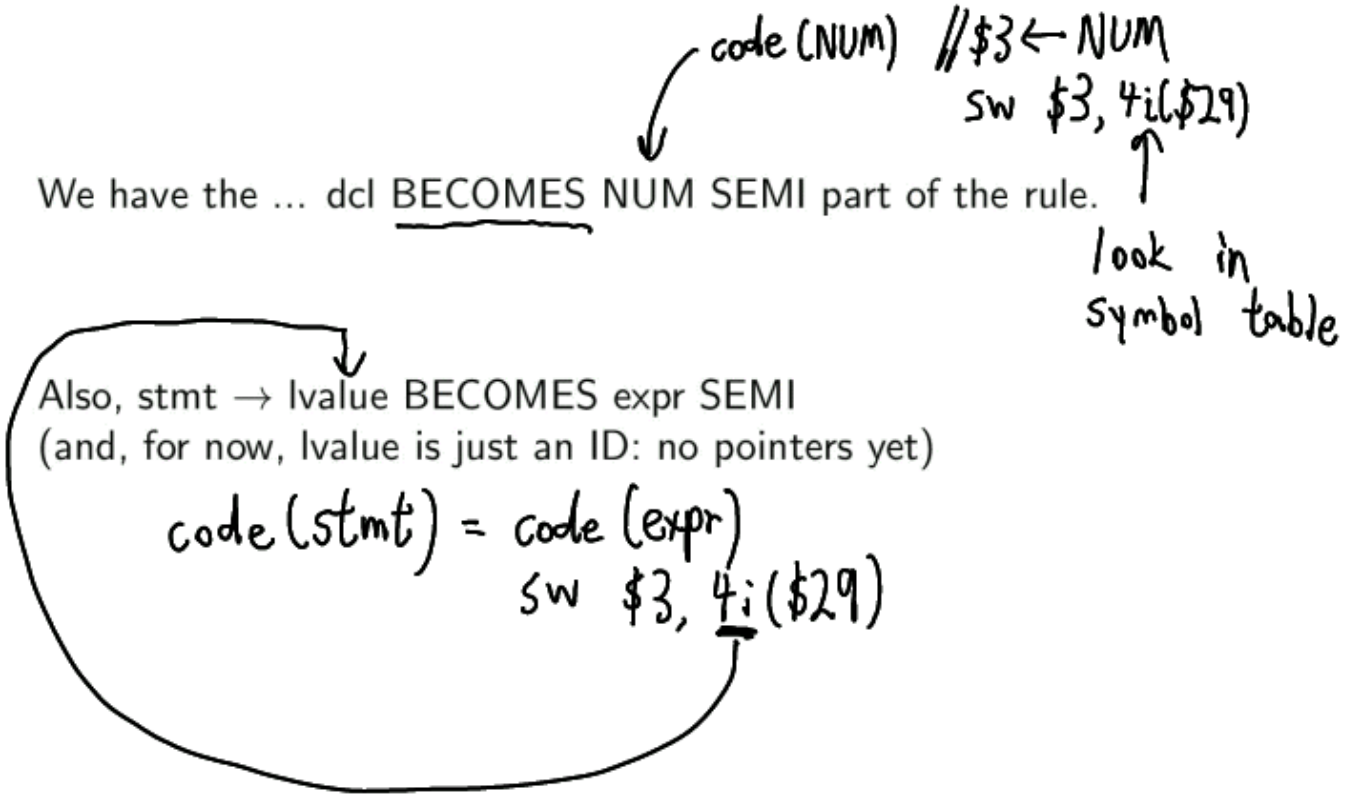
Variables on Stack



Notice location tells us where variable is:

$29 + 4 * i

Giving variables values



Booleans (A9P6)

The only spot where booleans are allowed are in control structures.

if
while

Rule: test → $expr_1$ LT $expr_2$

Conventions: $0 - 0
$11 - 1

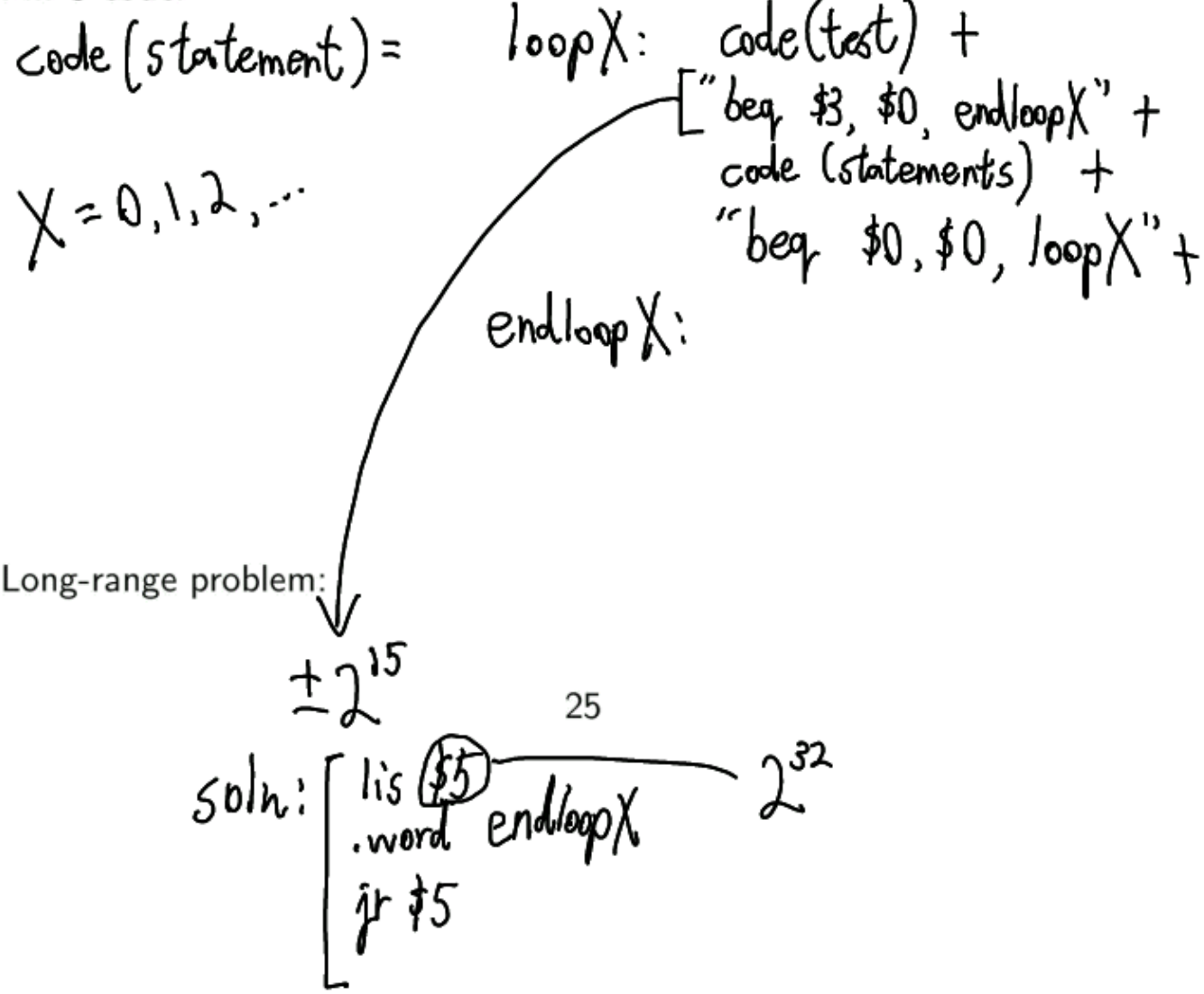
MIPS code:

code(test) = code($expr_1$) + // $3 ← $expr_1$
push ($3) +
code($expr_2$) + // $3 ← $expr_2$
pop ($5) + // $5 ← $expr_1$
slt $3, $5, $3

While loop (A9P6)

statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

MIPS code:



If statement (A9P8)

statement → IF test ... statements$_1$ ... ELSE ... statements$_2$ ...

Two choices: what code to place first

code(statement) = code(test) + // $3 ← test

```
        beq $3, $0, elseY +
        code(statements1) +
        beq $0, $0, endifY +
elseY:  code(statements2) +
endifY:
```

Back to booleans (A9P7)

Rule: test → expr GT expr

$X < Y$ ✓

$t > w \Leftrightarrow w < t$ ⇒ code(test) = code($expr_1$) push($3) code($expr_2$) pop($5) G slt $3, $3, $5

$expr_2 < expr_1$

Rule: test → expr NE expr

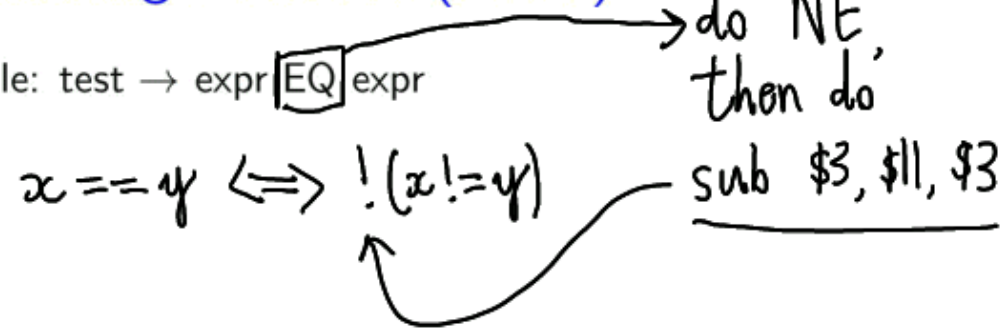
X != Y

In Pascal, Visual Basic, BASIC: x <> y ex:

code(test) = G + slt $6, $3, $5
slt $7, $5, $3
+ add $3, $6, $7

Finishing Booleans (A9P7)

Rule: test → expr EQ expr



Rule: test → expr GE expr

$$x \geq y \Leftrightarrow !(x < y)$$

Rule: test → expr LE expr

$$x \leq y \Leftrightarrow !(x > y)$$