CS 241 — Lecture 19

Bartosz Antczak *Instructor: Kevin Lanctot* *March 14, 2017*

19.1 Code Generation

Once we scan the program and have our parse tree, we want to generate the actual MIPS assembly language. We'll choose MIPS code that is simplest to use, especially for CS 241 students. We'll translate the syntax from our parse tree into code using *Syntax-directed Translation*.

19.1.1 Overview

Given a parse tree, we generate code for a given node in the tree by first generating the code for the node's children. We then bring all of the children's code together and there we have it — the current node's code.

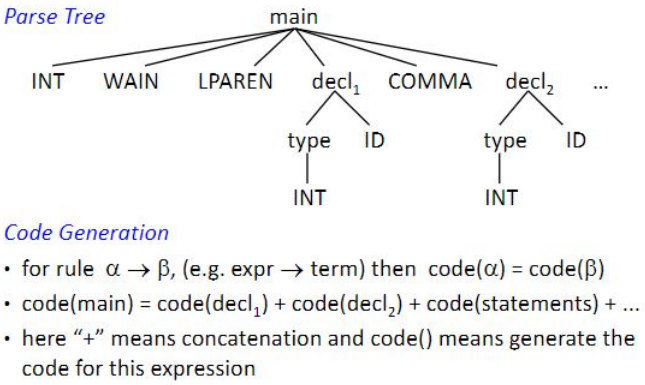

Figure 19.1: *Courtesy of Prof. Lanctot's slides.*

19.1.2 Storing Variables (A9P1)

Example 19.1.1.

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

Output

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

Conventions

We say that \$1 and \$2 hold the parameters for the `wain` function. \$3 holds the return value.

Example 19.1.2.

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

Output

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

Observations

Both of these examples have the same parse tree, so how do we differentiate the programs? We add a Location column to our Symbol Table:

Symbol Table		
Symbol	**Type**	**Location**
a	int	$1
b	int	$2

Figure 19.2: *Courtesy of Prof. Lanctot's slides.*

19.1.3 Methods of Storing Variables

We can use a **stack**, where each variable's location is determined on their position in the stack. An example is shown:

```
int wain(int a, int b) {
  int c = 0;
  return a;
}
```

- store parameters *a*, *b* and local variable *c* on the stack

Symbol Table		
Symbol	**Type**	**Location**
a	int	8
b	int	4
c	int	0

```
;;; prolog
lis $4
.word 4
sw $1,-4($30)   ; push a
sub $30,$30,$4
sw $2,-4($30)   ; push b
sub $30,$30,$4
;;; body
sw $0,-4($30)   ; push c
sub $30,$30,$4
lw $3,8($30)    ; return a
;;; epilog
add $30,$30,$4 ; pop c
add $30,$30,$4 ; pop b
add $30,$30,$4 ; pop a
jr $31
```

Figure 19.3: *Courtesy of Prof. Lanctot's slides.*

A Problem Arises

We don't know the offsets to the parameters and variables until all the variables (for that function) have been defined.

The **solution** is to include a frame pointer. We reserve \$29 to point to the first element of the stack, and any offsets in the symbol table will be relative to the frame pointer. An example utilizing this solution is shown:

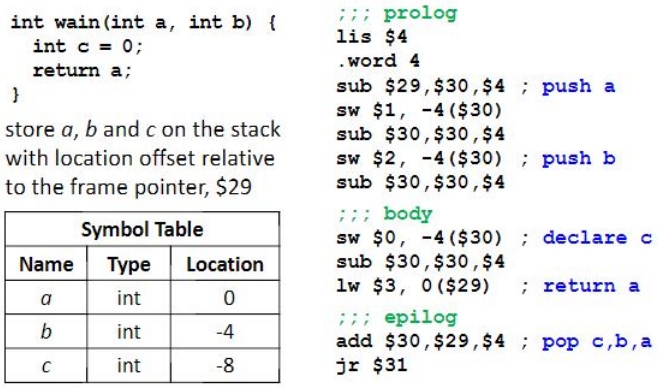

Figure 19.4: The variable's location are in reference to the frame pointer. See that a is zero away from the frame pointer, b is -4 away, and so on. *Courtesy of Prof. Lanctot's slides.*

19.1.4 Some Conventions for Code Generation (for CS 241)

- **\$0:** always 0 (or false)
- **\$1:** value of first parameter / argument
- **\$2:** value of second parameter / argument
- **\$3:** result of calculations
- **\$4:** constant 4 (helpful for pushing on and popping off stack)
- **\$5:** store previous intermediate results
- **\$11:** always 1
- **\$29:** frame pointer (fp)
- **\$30:** stack pointer (sp)
- **\$31:** return address (ra)

Structuring our MIPS Code

It will be subdivided into three sections:

- **Prologue**: push return address and arguments on stack
- **Generated Code**: put local variables on stack and generate code for body of function
- **Epilogue**: pop local variables and arguments off stack; restore previous return address to \$310

19.1.5 Some Examples of Code Generation

Example 19.1.3. *A9P2:*

Here, we simply ignore the parentheses since they don't generate any code. We simply evaluate the expression in it:

```
Code
  int wain(int a, int b) {
     return (a);
  }
Output
  ;; same prolog
  lw $3, 0($29) ; load a from stack(based on fp)
  ;; same epilog
```

Figure 19.5: *Courtesy of Prof. Lanctot's slides.*

Example 19.1.4. *A9P3:*

```
Code
  int wain(int a, int b) {
     return a+b;
  }
Output
  add $3, $1, $2
```

Figure 19.6: *Courtesy of Prof. Lanctot's slides.*

How we we deal with general expressions? (e.g., $a+b+c+d+e+\cdots$)

Solution

We use our special registers \$3 and \$5! For the rule $expr_1 \to expr_2 + term$, we perform the following:

- Evaluate $expr_2$ and store in \$3
- Push the value of \$3 onto the stack
- Now evaluate $term$ and overwrite it into \$3
- Pop the value of $expr_2$ off the stack and store in \$5
- Add \$5 and \$3 together and store in \$3

```
;; code(expr1) =
code(expr2)     ; $3 = expr2
push($3)        ; pseudocode to push $3 onto stack
code(term)      ; $3 = term
$5 = pop()      ; $5 = expr2, pseudocode to pop stack
add $3, $5, $3 ; $3 = expr2 + term
```

Figure 19.7: Pseudo code of the approach. *Courtesy of Prof. Lanctot's slides.*

Example 19.1.5. *A9P4*

For rule *statements* $\rightarrow$ *PRINTLN LPAREN expr RPARENT*, we perform a function call to `print`, which will be provided for us (yes!). We must import `print` in our MIPS language:

- First compile your wlp4i code into assembly
- Then convert your assembly file to a merl file
- Finally, link said merl file with `print.merl` and output the file into your executable merl file

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

Figure 19.8: *Courtesy of Prof. Lanctot's slides.*

Note: the `print` file overwrites whatever content is in $1 and $31. $1 is okay to be overwritten, but we need to store whatever is in $31 onto the stack. Once we finish our code, in our epilogue, we pop the value off the stack and restore it into $31.