Tutorial 9
- Code Generation

Code Generation
- Idea: Convert WLP4 to MIPS
  → Assemble MIPS to machine code to run our program!
- Convention: WLP4 programs will:
  → Use $1 & $2 for input (wain's parameters)
  → Use $3 for output (wain's return value)

Done via mips.twoints or mips.array

- General Code Gen for Variables:
  → Generate code for the following:
    int wain (int a, int b) {
    int c = 42;
    return a;
  }

  → Store all WLP4 vars on the stack $30
    → All vars must be declared 1st in our wain & procedures. Easy to identify.
    → But what if in our MIPS code another procedure modifies the stack? How do I track my current procedures vars? Need a Frame pointer
Frame pointer ($29$): A dedicated register always fixed to a fixed MEM address within $330$ while executing a procedure.

We can use fixed offsets w.r.t. the unchanging $29 \rightarrow$ access vars in the current stack frame.

Note: Often we set $29 = \text{the 1st value in the frame of a procedure}$.

From our eg:

\[
\begin{array}{ll}
\text{int wain (int a, int b) \{ } & \\
\text{int c = 42; } & \text{lw $x$, -8($29$) = c = 0} \\
\text{return a;} & \text{lw $x$, -4($29$) = b = $2$} \\
\text{\}} & \text{lw $x$, 0($29$) = a = $1$} \\
\end{array}
\]

Wain’s stack frame, access vars from $29$.

\text{Code gen.}

\text{; Prolouge - Push all vars onto the stack}
\text{lis $s4$}

\text{.word 4}

\text{Sub $s29$, $s30$, $s4$; Set $s29 = 1$st var on stack a}
\text{sw $s1$, 0 ($s29$) \{ Push a = $s1$ onto the stack}
\text{sub $s30$, $s30$, $s4$ \{ Update $s30 = s30-4$}
\text{sw $s2$, -4 ($s29$) \{ Push b = $s2$ onto the stack}
\text{sub $s30$, $s30$, $s4$ \{ Update $s30 = s30-4$}
\text{lis $s3$}

\text{.word 42}
\text{sw $s3$, -8 ($s29$) \{ Push c = 42 onto the stack}
\text{sub $s30$, $s30$, $s4$ \{ Update $s30 = s30-4$}
; Begin Code
lw $3, 0($29) ; a's offset to $29 is 0

; Epilogue - Restore stack & return
add $30, $30, $4    ; Programatically we reset $30
add $30, $30, $4    ; to represent "popping off" the
add $30, $30, $4    ; main/procedure
jr  $31

---

Code Gen for expressions

eg: Generate the code for the following WLP4 program:

```c
int wain (int a, int b) {
    int c = 3;  // How do we enforce precedence? (< )
    return a + (b-c);    // How do we store intermediate steps?
}
```

- Problem: How do we store intermediate steps of any expression?
  - Use temp registers? Limited & not scalable
  - Idea: Store steps of an expression in the stack!
    - recursively push non-terminals (push from $3)
    - pop at terminals & calculate expressions (pop to $5)
    - inside-out (according to precedence in our parse tree!)
    - our grammar enforces precedence in our expressions, so all we need to do is traverse our tree with push & pop!
int wain (int a, int b) {  
    int c = 3;
    return a + (b - c);
}

Prologue
leftmost ID's of their tree.

Code Gen:  Apply ops as we pop!

; Prologue — Push wain's vars onto $30
lis $4
.word 4

sub $29, $30, $4  \iff  $29 = $30 - 4
sw $1, -4 ($30)  \iff  ; Push a = $1
sub $30, $30, $4

sw $2, -4 ($30)  \iff  ; Push b = $2
sub $30, $30, $4

lis $5  \iff  ; Push c = 3 (use $5 by convention)
.word 3

sw $5, -4 ($30)
sub $30, $30, $4

; Begin Code
lw $3, 0($29)  \iff  ; load a into $3
sw $3, -4 ($30)  \iff  ; Push a onto $30
sub $30, $30, $4
lw $3, -4($29)  \iff  ; load b into $3
su $3, -4 ($30) \{ \text{Push } b \text{ onto } $30 \} \\
sub $30, $30, $4 \\
li $3, -8 ($24) \}; \; \$3 = c \\
add $30, $30, $4 \{ \text{Pop } b \text{ into } $5 \; ($5=b) \} \\
lw $5, -4 ($30) \\
sub $3, $5, $3 \}; \; \$3 = b - c \\
add $30, $30, $4 \{ \text{Pop } a \text{ into } $5 \; ($5=a) \} \\
lw $5, -4 ($30) \\
add $3, $5, $3 \}; \; \$3 = a + (b-c) \\
\text{/ Epilogue - Unwind Stack} \\
add $30, $30, $4 \{ \text{3 vars = \text{wain}\text{'}s stack frame} \} \\
add $30, $30, $4 \\
jr $31 \\

\text{Adding New Expressions} \\
\text{eg: } \text{C & C++ have pre & post increment operators} \\
\text{++i & i++. If we added the following rules to WLP4:} \\
\text{factor} \rightarrow \text{PLUS PLUS lvalue} \quad (\text{++i}) \\
\text{factor} \rightarrow \text{lvalue PLUS PLUS} \quad (\text{i++}) \\
\text{Assume these are only applied to INT types.} \\
\text{Write pseudocode to generate the correct MIPS output for each rule.}
• `++i` code gen
  \( \rightarrow \) Increases the value of \( i \) by 1, then returns \( i \)
  \( \rightarrow \) Note: this is a factor rule, not an lvalue. We cannot nest \( (++i) \) (invalid)

```c
void genCode (tree t)
{
 ...
    if (t.rule == "factor -> PLUS PLUS lvalue") {
        // gen code to put lvalue in $3
        genCode (t.children[2]); // $3 = lvalue
        // Recall: lvalue is a MEM address, load the actual value
        *lw $5, 0($3); // $5 = RHS of lvalue
        // add one to lvalue
        lis $11
        .word 1
        add $5, $5, $11
        // save new value in MEM
        sw $5, 0($3)
        // Return new value
        add $3, $5, $0
    }
}```
• `i++ code gen`  
  \[ \rightarrow \text{return } i, \text{ increment } i \text{ by } 1 \]  
  \[ \rightarrow \text{need to remember the old value to return in } S3 \]

```c
void genCode (tree t) { 
...
  if (t.rule = "factor -> ivalue PLUS PLUS") { 
    genCode (t.children[0]); // $3 = ivalue
    lw $5, 0($3); // $5 = RHS of ivalue
    add $6, $5, $0; // $6 = $5, copies the ivalue
    lis $11
    .word 1
    add $5, $5, $11 // add one to ivalue
    sw $5, 0($3) // save new value in MEM
    add $3, $6, $0 // Return old value
  }

  
  
* Note: For `lw $5, 0($3)` we assume that the full address in $3 is not offset. Otherwise, we would need to do something like: `add/sub $5, $3, 529` to reorient it to our current frame.