Tutorial 10

- Switch Statement Code Gen
- Extending ++i & i++
- Extending Pointers as Conditions

Recap: Code Gen Conventions

- For all code(...) we have the following format:

```
; Prologue - Push all vars to the stack & setup temporaries
 .import print      ; println, call with jalr
 lis $4
 .word 4
 lis $10
 .word print        ; jalr $10 calls print
lis $11
.word 1
sub $29, $30, 4     ; Setup frame pointer
; ... Reserve space for vars & push to stack ...
; Begin Code:
code(...)
; Epilouge - Restore stack & return
; ... deallocate vars, reset $30 ...
jr $31
```

☆ Often, we assume $3 holds the return val of code(...) & $5 to hold temporary values. ☆

Conditionals

- Statement → IF (test) { Statements } else { Statements }
- Assume :
  - ↳ code(test) { sets $3 = 1 if the test is true
    " $3 = 0 " " " " false
  - ↳ code(statements) generates code for the statement
  - ↳ genLabelID() = generates a unique id ∈ ℤ
    → ie: Stores & returns a counter that is incremented each call
- With these assumptions, we can code if statements like:

```
void genCode(tree t){
  ...
  if(t.rule == "Statement → IF ( test ) { Statements } else { Statements }"){
    x = genLabelID()
    genCode(t.children[2])   // $3 = test
    beq $3, $0, else+x       // if test is false ($3=0), execute else block
    genCode(t.children[5])   // Statements
    beq $0, $0, endif+x      // test was true ($3=1), skip the else block
    else+x :
    genCode(t.children[9])   // Statements
    endif+x :
  }
}
```

LPAREN RPAREN ...terminals...

eg: How could I add Switch statements to WLP4?

```
switch (expr) {
      case (expr) {
            statements
      }
      case (expr) {
            statements
      }
      ...
      default {
            statements
      }
}
```

- Assumptions:
  - ↳ case statements don't fall through (no need for break;)
  - ↳ default case is mandatory → may not run but always there
  - ↳ expr is an arbitrary expression.
  - ↳ assume all of scanning, parsing & semantic analysis are done.
- Rules to add to WLP4:

  statement → SWITCH (expr) { cases default }

  cases → cases case

  cases → ε

  case → CASE (expr) { statements }

  default → DEFAULT { statements }

- Challenge: Remembering which label the case statements need to jump to after execution
- Soln: Augment the tree t to store an int parentLabelID field on each node

```
void genCode(tree t){
  ...
  if(t.rule == "statement → SWITCH ( expr ) { cases default }"){
      x = genLabelID()
      genCode(t.children[2])        // $3 = expr
      push($3)                      // Pushed to compare with all cases
      t.children[5].parentLabelID = x   // Pass parent ID
      genCode(t.children[5])
      genCode(t.children[6])
      endSwitch + x :
  }
  if(t.rule == "cases → cases case"){
    t.children[0].parentLabelID = t.parentLabelID }
    t.children[1].parentLabelID = t.parentLabelID }  Propogate parent ID
    genCode(t.children[0])
    genCode(t.children[1])                           } generate case statement code
  }
  if(t.rule == "cases → ε "){
      // Do nothing
  }
```

```
if (t.rule == "case → CASE ( expr ) { statements } ") {
    x = genLabelID()
    genCode(t.children[2])      // $3 = expr
    pop($5)                     // $5 = Switch statement's expr
    //Compare switch expr to current case expr
    bne $3, $5, endLabel + x
    genCode(t.children[5])
    jr endSwitch + t.parentLabelID
    endLabel + x :
    push($5)
}
if (t.rule == "default → DEFAULT { statements }") {
    pop($5)     //throw away switch's expr, not needed anymore
    genCode(t.children[2])
}
}
```

$3 = $5, gen case's code & break from switch

$3 ≠ $5, Push Switch's expr back for the next case to try out

eg: Recall from tutorial 9 we implemented ++i & i++ for ints, lets extend these rules to also work for pointers int*

factor → PLUS PLUS lvalue

factor → lvalue PLUS PLUS

① Give type rules for the above grammar rules

② Show modifications to the genCode for these rules to increment int & int*'s

Soln ①:

- If lvalue is an int, then so is factor
- If lvalue is an int*, then so is factor

Soln ②:

- lets modify the line "add $5, $5, $11" to say:

```
if ( typeOf ( lvalue ) == int* ) {
    add $5, $5, $4          // int* increment
} else {
    add $5, $5, $11         // int increment
}
```

eg: Recall C pointers can be used as conditions:

```
int *c = NULL
if (c) {
    // c is not NULL
} else {
    // c is NULL
}
```

Commonly used to check if pointers are null

① Modify the WLP4 grammar so pointers can be used in if & while tests

② Describe how you would modify type checking to account for these new rules

③ Write pseudocode to generate code for these new rules

Soln ①:

- Recall all conditions are under the test non-terminal, lets add our rule there!

  test → expr

Soln ②:

- Verify that expr is type int* for the rule test→expr to be well-typed

Soln ③:

- $3 = { 1 if the expr is not NULL
  0 " " " " NULL
- Assume the value for NULL == 0 (easy to swap out)

```
genCode (tree t) {
    ...
    if (t.rule == "test → expr") {
        genCode (t.children[0])   // $3 = expr
        add $5, $3, $0            // Backup expr
        lis $6
        .word 0                   // Value for NULL
        add $3, $0, $0
        beq $5, $6, 1             // $3=0 if expr = NULL
        add $3, $11, $0           // $3=1 since expr ≠ NULL
    }
}
```