**Code Generation**

* **Code generation**
* Source → lexical analysis (scanning)
  + → [tokens] → context-free analysis (parsing)
  + → [parse tree] → context-sensitive analysis (semantic analysis)
  + → [parse tree, symbol table] → code generation
* At this point, we have guaranteed for the source program to be free of compile-time errors
* Now we need to generate equivalent MIPS code
* Ex: int wain(int a, int b) { return a; }
  + Conventions: params in $1, $2; output passed in $3
  + MIPS:

Add $3, $1, $0

Jr $31

* Ex: int wain(int a, int b) { return b; }
  + MIPS:

Add $3, $2, $0

Jr $31

* These two programs have the same parse tree – how to tell one from another?
  + Use symbol table
  + Add column to table for where each symbol is stored (register location)
  + When traversing the parse tree for code generation, when ID is countered → lookup in symbol table to get location
  + For local variables & params of wain – allocate space on the stack
  + Reference local variables by the amount of offset from $30
  + But offsets change as other things are added to/removed from the stack
* Ex: int wain(int a, int b) { return a; }
  + MIPS:

sw $1, -4($30) //store a, b

sw $2, -8($30)

lis $4

.word 8

sub $30, $30, $4 //adjust SP

lw $3, 4($30) //output a (a is stored 4 from SP)

//this offset will change if more items are pushed onto stack

add $30, $30, $4 //adjust SP

jr $31

* Introduce conventions:
  + $4 always contains 4 – do not change it
  + $29 points to the bottom of stack frame (frame pointer) – location of local variables will be calculated using offsets to $29, which remain constant
* Ex: int wain(int a, int b) { int c = 0; return a; }
  + MIPS:

lis $4 //$4 = 4

.word 4

sw $1, -4($30) //store a, b, c & adjust SP

sub $30, $30, $4

add $29, $30, $0 //set up SP to point at first local variable stored on stack

sw $2, -4($30)

sub $30, $30, $4

sw $0, -4($30)

sub $30, $30, $4

//symbol table:

//a = 0 from $29; b = -4 from $29; c = -8 from $29

lw $3, 0($29) //output a

add $30, $29, $4 //restore SP value = FP + 4

* More complicated program:
* In general, for each grammar rule A → γ, build code(A) from code(γ)
* Introduce conventions:
  + Use $3 for output of all expressions
    - But if there are multiple pending operations – where to store those results without overwriting $3?
    - Use the stack
* Ex: a + (b + (c + d))
  + MIPS:

; code(a) $3 ← a

; push $3

; code(b) $3 ← b

; push $3

; code (c) $3 ← c

; push $3

; code(d) $3 ← d

; pop $5 $5 ← c

add $3, $5, $3 ; $3 = c + d

; pop $5 $5 ← b

add $3, $5, $3 ; $3 = b + (c + d)

; pop $5 $5 ← a

add $3, $5, $3 ; $3 = a + (b + (c + d))

* Printing output
  + Runtime environment – a set of procedures supplied by the compiler/OS to assist programs in their execution
  + To make procedure print part of the runtime environment:
    - Use provided print.merl
    - Assemble output & link with print.merl
  + i.e.:
    - ./wlp4gen < source.wlp4i > source.asm
    - java cs241.linkasm < source.asm > source.merl
    - linker source.merl print.merl > full.merl
    - java mips.twoints/array full.merl
  + Assume “print”procedure is available
* Ex: println(expr)
  + MIPS:

; code(expr) $3 ← expr

add $1, $3, $0 ; $1 ← expr

; push $31

lis $5

.word print

jalr $5

; pop $31

* Assignments
  + i.e. lvalue = expr;
  + Assume lvalue is an ID
  + MIPS:

; code(expr) $3 ← expr

sw $3, x($29) ; look up ID in symbol table to determine x

* + Introduce conventions:
    - Store value of 1 in $11
    - Store print procedure in $10
  + Code so far has the form:

.import print ; PROLOGUE

list $4

.word 4

list $10

.word print

lis $11

.word 1

sub $29, $30, $4

… ; allocate stack for all variables

; CODE

add $30, $29, $4 ; EPILOGUE

jr $31

* Boolean testing
  + i.e. expr1 < expr2
    - MIPS:

; code(expr1) ; $3 ← expr1

add $6, $3, $0 ; $6 ← expr1

; code(expr2) ; $3 ← expr2

slt $3, $6, $3 ; $3 ← $6 < $3

* + i.e. expr1 > expr2
    - MIPS:
    - Treat as expr2 < expr1, then do the same as above
  + i.e. expr1 != expr2
    - MIPS:

; code(expr1)

add $6, $3, $0

; code(expr2)

slt $7, $6, $3 ; $7 ← $6 < $3

slt $8, $3, $6 ; $8 ← $3 < $6

add $3, $7, $8 ; $3 ← $7 or $8, i.e. $3 != $6

* + i.e. expr1 == expr2
    - MIPS:
    - Treat as !(expr1 != expr2)
    - To invert 0/1:

sub $3, $11, $3 ; subtract $11 from 1 (recall $3 = 1)

* If statements
  + i.e. if (test) { statements1 } else { statements2 }
  + MIPS:

; code test ; $3 ← test

bne $3, $11, else ; recall $11 = 1

; code statements1

beq $0, $0, endif

else:

; code statements2

endif:

* + Issue: need unique label names for multiple if statements
    - Use a counter X to keep track of new if statements
    - Use elseX, endifX as label names
    - For nested ifs – make sure to generate labels first before recursing
* While
  + i.e. while (test) { statements }
  + MIPS:

loopY: ; label top of loop

; code(test) ; $3 ← test

beq $3, $0, doneY

; code(statements)

beq $0, $0, loopY

doneY:

* Generate comments in output assembly so debugging is easier
* Pointers
  + In order to support pointers, these must be supported:
    - NULL
      * Could use 0 – but dereferencing NULL should result in crash
      * So use 1 as NULL
    - Dereferencing operator
    - Address-of operator
    - Alloc/dealloc (i.e. new/delete)
    - Pointer arithmetic
    - Pointer comparison
    - Assignment through pointers
  + Dereferencing
    - i.e. \* expr
  + Comparison
    - Same for comparing ints
    - Use slt if expr1 & expr2 have type int
    - Use sltu if they have type int\*
    - Add a “type” field in each tree node to record the type
  + Pointer arithmetic
    - i.e. expr + term
    - MIPS:

If expr, term : int

As before

If expr : int\*, term : int

Code(expr)

Push $3

Code(term)

mult $3, $4

mflo $3 ; $3 = 4 \* term

pop $5 ; $5 = expr

add $3, $5, $3 ; $3 = expr + 4 \* term

* + - i.e. expr – term
    - MIPS:

If expr, term : int

As before

If expr, term : int\*

Code(expr)

Push $3

Code(term)

Pop $5

sub $3, $5, $3

div $3, $4

mflo $3 ; $3 = (expr – term) / 4

* + Assign thru pointer dereference
    - i.e. \* expr1 = expr2;
    - LHS = address at which to store the value
    - RHS = the value
    - Calculate the value of expr1
    - Use it as the address at which to store expr2
    - MIPS:

Code(expr2)

Push $3

Code(expr1)

Pop $5

sw $5, 0($3)

* + Address-of
    - i.e. & lvalue
    - MIPS:

if lvalue = ID

lis $3

.word (look up ID’s offset in symbol table)

add $3, $29, $3

if lvalue = \* expr

code(expr)

* + New/delete
    - Part of the runtime environment
    - Use the provided allocation module alloc.merl
    - Alloc.merl must be linked last
    - Add to the prologue:
    - .import init
      * Set up allocator’s data structures
      * Called once at the beginning of the asm file
      * Takes a param in $2
        + If calling program with mips.array, i.e. first param of wain is int\*
        + $2 = length of array
        + Otherwise $2 = 0
    - .import new
      * $1 = # of words needed
      * Returns a pointer to memory in $3
      * Returns 0 if allocation fails
      * i.e. new int[expr]
      * MIPS:

Code(expr)

add $1, $3, $0

call new

bne $3, $0, $11 ; skip next line if result of new != 0

add $3, $11, $0 ; if new failed, $3 = 1 (NULL)

* + - .import delete
      * $1 = address to be freed
      * Deleting NULL is guaranteed to be safe (does nothing)
      * First check the address; only call delete if address is not NULL
      * i.e. delete [] expr
      * MIPS:

Code(expr)

beq $3, $11, skipY ; skip if $3 = 1 (NULL)

add $1, $3, $0

call delete

skipY:

* Procedures
  + Make sure to execute main first
    - Either generate the code for main at the top, followed by procedures
    - Or generate a first line that jumps to main, i.e. beq $0, $0, wain
  + Main prologue/epilogue
    - Save $1, $2 on stack
    - Import print, init, new, delete
    - Set $4, $11 etc. constant registers
    - Save & set $29
    - Call init
    - Restore stack at the end & jr $31
  + Procedure prologues/epilogues
    - Don’t need to import
    - Don’t need to set constants
    - Save & set $29
    - Restore stack at the end & jr $31
  + Procedures should save & restore all registers that are modified
  + How to know which registers to save if the procedure code hasn’t been generated yet?
    - Don’t save $3 – used for output
    - Can just save all the registers
    - Or since our code generation methods don’t use any registers past $8, just save these