**CS 241 Final Review**

* Compiling = high-level language → assembly language
  + WLP4 code → (lexical analysis/scanning)
    - i.e. checking code consists of valid WLP4 tokens using DFA
    - Lexical errors are detected
  + → WLP4 tokens → (syntax analysis/parsing)
    - i.e. checking that the code structure matches the WLP4 grammar (context-free)
    - Syntax errors are detected
  + → parse tree → (semantic analysis)
    - i.e. checking that the code meaning is correct (context-sensitive)
    - Semantic errors are detected
  + → parse tree + symbol table + type information → (code generation)
    - No errors should be detected
  + → MIPS
* Assembling = assembly language → machine language
  + MIPS → (assembler)
  + → MERL → (linker) → (loader)
  + → binary executable
* **Context-free languages**
* A context-free grammar consists of
  + ∑ = set of terminals – symbols that appear in the output
  + N = set of non-terminals – symbols that do not appear in the output
  + S ∈ N = start symbol
  + P = set of production rules – each expands a non-term → sequence of non-terms/terms
* Derivation: S ⇒\* w
  + i.e. S ⇒ A ⇒ B ⇒ … ⇒ w, w ∈ ∑\*
  + Begin with S, expand single non-terms using production rules, repeat until only terms remain
  + Leftmost/rightmost derivation – expanding left/right-most non-term first
* Context-free language L(G) = {w ∈ ∑\* | S ⇒\* w}
  + i.e. set of all strings that can be derived from the start symbol
* **Simple Maximal Munch Algorithm (scanning)**
  + Begin in start state
  + Consume input and traverse DFA until an error state is reached
  + If last state is an accepting state, output token & return to start state (read next token)
  + Otherwise, ERROR
* **Parse trees**
  + Root = start symbol
  + Internal nodes = non-terms
  + Leaf nodes = terms
  + Parse trees are traversed in:
    - Post-order − process children before processing parent
    - Depth-first − children are processed left to right
  + A CFL is ambiguous if ∃ more than one derivation/parse tree for a word in the language
    - Avoid having >1 instances of a non-term on the RHS of a rule
    - E.g. instead of E → E + E, use E → E + T; recursive expansion only allowed on one side
* **Top-down parsing**
  + Start with S → produce w
  + i.e. replace LHS of rule with RHS
* LL(1) parsing
  + Left-to-right scan of input, leftmost derivation, 1-symbol look-ahead
  + Uses predictor table to predict which rule to use to expand non-terms
  + Uses a symbol stack (top of stack is on the left)
  + First(α) = {a | α ⇒\* aβ} where a ∈ ∑ and α, β ∈ (N ∪ ∑)\*
    - i.e. all the terms that can be derived on the leftmost side from a given sequence
  + Follow(A) = {a | S’ ⇒\* αAaβ} where A ∈ N, a ∈ ∑, and α, β ∈ (N ∪ ∑)\*
    - i.e. all the terms that can immediately follow a given non-term
    - The given non-term must be able to be reduced to ε, i.e. A ⇒\* ε
    - i.e. A will be replaced by a when A is reduced to ε
  + Predict(A, a) = {rule (A → β) | a ∈ First(β) ∨ (β ⇒\* ε ∧ a ∈ Follow(A))
  + A grammar is LL(1) if |Predict(A, a)| ≤ 1 for all A, a
    - In general, if there are >1 rules from non-term A, and one of them is left-recursive (e.g. A → A + B), the grammar will not be LL(1)
  + **LL(1) algorithm:**
    - Start with S’ on the stack
    - If top of stack = term(a), match with input char(c)
      * If a != c, ERROR
      * Pop a off stack
      * Move c from “unread” to “read”, read next char
    - If top of stack = non-term(A), expand
      * Look up rule(A → β) = Predict(A, c)
      * If rule doesn’t exist, ERROR
      * Replace A on stack with β
    - Parse completes when both “unread” and symbol stack contain only −|
    - Derivation = contents of “read” + symbol stack (top to bottom)
  + The expand rules used from top to bottom provides a leftmost derivation
* **Bottom-up parsing**
  + Start with w → produce S
  + i.e. replace RHS of rule with LHS
* LR parsing
  + Left-to-right scan of input, rightmost derivation
  + Uses symbol stack (top of stack is on the right) & state stack
  + Two possible actions at each step:
    - Shift – push an input char into the symbol stack
    - Reduce – replace RHS of rule with LHS
  + Derivation = contents of symbol stack (bottom to top) + “unread”
  + Uses transducer to determine whether to shift or reduce at each step
    - Transducer – a DFA that can also produce output on transitions
  + Item – production rule with ⋅ in RHS
  + Complete item – ⋅ is in the rightmost position; i.e. A → β ⋅
  + Building the LR(0) automaton – refer to **written note**
  + **Using the LR(0) automaton**:
    - If ∃ transition out of current state on current input char → shift
    - If current state has only one item & it’s complete → reduce
  + **Using the shift/reduce table:**
    - Look up using (top of state stack, current input char)
    - Reducing:
      * Pop RHS off symbol & state stacks, push LHS on symbol stack
      * Look up using (top of state stack, top of symbol stack)
    - Accept when “unread” is empty
  + Shift-reduce conflict – a state has both options of shifting or reducing
  + Reduce-reduce conflict – a state has more than one option of reducing
  + In general, if any complete item is not alone in a state, there will be a conflict
* SLR(1) parsing
  + Simple LR + 1-symbol look-ahead
  + For every complete item A → β ⋅ , attach Follow(A)
  + Reduce action only applies if the next input char ∈ Follow(A)
  + The reduce rules used from bottom to top provides a rightmost derivation
  + A S-R conflict still exists if there is an incomplete item A → α ⋅ bβ and b ∈ Follow(A)
  + A R-R conflict still exists if there are >1 complete items and their follow sets overlap
  + i.e. 1-symbol look-ahead may not be enough to resolve all ambiguities
* **Semantic analysis**
* Declarations & procedure scoping
  + Variables & procedures must be declared before being used
    - i.e. a procedure must be declared above its call
    - i.e. mutual recursion not allowed
    - This allows declaration error-checking to be done in one pass of the parse tree (instead of 2 passes as done by the assembler)
  + Variables & procedures cannot be declared multiple times
  + Scoping – forbid duplicate declarations in the same procedure, but allow in different procedures
  + Variables’ types must be kept track of
  + Procedures’ signatures (list of param types) must be kept track of
  + Use symbol table:

// proc name proc signature proc symbol table (var name, var type)

map <string, <vector<string>, map<string, string> > >

* Type checking
  + Types keep track of what kind of data is stored in each block of memory
  + Draw a tree to help determine the type of an expression
  + See WLP4 specs
* Some WPL4 syntax/semantics rules
  + All variables must be declared at the beginning of a procedure
  + Only one return statement allowed at the end of each procedure
  + Procedures cannot return int\*
  + If statements must have both if & else blocks
  + Assignment (=) and comparison operators must be separated by whitespace
* **Code generation**
* Conventions
  + Params of wain are stored in $1 & $2
    - If using mips.twoints, params = (int a, int b)
    - If using mips.array, params = (int\* a, int b)
      * a = starting address of first element in array
      * b = length of array (# of elements)
  + $4 = 4
  + $11 = 1
  + $29 = frame pointer
  + $30 = stack pointer
  + $3 contains result of computation
    - i.e. $3 stores value of expression
    - i.e. procedure returns value into $3
  + Call init at beginning of program
    - Takes $2 as input – leave as is if using mips.array
    - Otherwise $2 must be 0
  + Store commonly used procedure addresses in registers, e.g. print, new, delete
  + Print takes $1 as input (bytes to print)
* Using the stack
  + Push:

sw $3, -4($30)

add $30, $30, $4

* + Pop:

lw $3, 0($30)

sub $30, $30, $4

* Create offset table to keep track of local vars/params relative to FP

// proc name offset count proc offset table (var name, var offset)

map <string, pair<int, map<string, int> > >

* + Init offset count at 0
  + For every dcl, add entry to offset table & offset count -= 4
    - This can be done in the semantic analysis stage (at the same time as creating the symbol table)
  + For every reference/assignment to ID (local var), look up proc & var name in offset table

lw/sw $3, x($29) ; x is offset

* Expressions
  + Code(expr → expr + term) =

code(expr)

push $3 ; save 1st operand on stack

code(term)

pop $5 ; load 1st operand into $5

add $3, $3, $5 ; perform operation

* Boolean tests
  + a < b → slt $3, $5, $3
  + a > b → test b < a
  + a != b → test a < b || b > a
  + a == b → test !(a < b || b > a)
  + a <= b → test !(a > b)
  + a >= b → test !(a < b)
* If statements
  + Use counter to keep if labels unique
  + Code(statement → if …) =

code(test)

beq $3, $0, elseX ; X is counter

code(statements1)

beq $0, $0, endifX

elseX:

code(statements2)

endifX:

* While statements
  + Use counter to keep loop labels unique
  + Code(statement → while …) =

loopX:

code(test)

beq $3, $0, endLoopX

code(statements)

beq $0, $0, loopX

endLoopX:

* Procedures
  + Adjust offset table so that
    - Pushed args → positive offsets
    - Pushed local vars → 0 & negative offsets
  + Add “F” before all procedure labels to avoid conflicts with other program labels
  + The caller must save $31
  + Saving other registers:
    - If there are few procedures called many times → prefer callee saving
    - If there are many procedures called few times (some may be unused) → prefer caller saving
  + Code(procedure definition) =

sub $29, $30, $4 ; set up FP

code(dcls) ; push local vars first

adjust offset table ; add # of args \* 4 to every entry

push regs ; then save used regs

code(statements)

code(expr)

pop regs ; restore saved regs

add $30, $29, $4 ; restore SP

jr $31

* + Code(procedure call) =

push $29, push $31

code(arglist) ; push args on stack

lis $5

.word Fproc

add $30, $30, $9 ; pop args from stack, $9 = # of args \* 4

pop $31, pop $29

* Pointers
  + Use 1 as value for NULL – guarantees the MIPS machine will crash if dereferenced
  + Also need code gen for:
    - Dereferencing
    - Address-of
    - New/delete
      * New takes $1 as input (# of words to allocate)
      * New returns $3 (starting address of allocated memory)
      * Delete takes $1 as input (address of memory to be freed)
    - Pointer assignment (e.g. \*expr = expr)
    - Pointer arithmetic
      * If a(int\*) + b(int) → $3 = a + 4\*b
      * If a(int\*) – b(int) → $3 = a – 4\*b
      * If a(int\*) – b(int\*) → $3 = (a – b)/4
    - Pointer comparison
      * Use sltu instead of slt
* **Compiler optimization**
* Constant folding
  + Arithmetic operations with constants can be done during compile-time instead of generating MIPS for them
* Constant propagation
  + Reuse values of variables (without loading from memory) if they have not changed
* Common subexpression elimination
  + Save values of common expressions so they don’t need to be evaluated for every instance
  + May not work for function calls due to side effects
* Dead code elimination
  + Do not generate code for branch of program that is never executed
* Register allocation
  + Use registers to hold variable & expr intermediate values when possible, instead of the stack
  + Cannot use address-of on a variable stored in register
* Strength reduction
  + Add is faster than multiply
* Procedure inlining
  + Replace a function’s call with the function’s body within the caller
  + If all calls are inlined, function does not have to be declared
  + Not preferable if function’s body is long/function is called many times
* Tail recursion
  + The last thing a procedure does before returning is recursively call itself
  + Thus the current stack frame can be reused infinitely
  + The procedure can be compiled as a loop
* **Memory management**
* Explicit/manual memory management – i.e. new/delete
* Implicit/automatic memory management – i.e. garbage collection
* **List of free blocks method**
  + A linked list of pointers to every free block of memory
  + When a block of k bytes is request, actually allocate k + 4 bytes
    - 1st byte = size of block (k + 4)
    - Return pointer to 2nd byte as the “start” of block
  + Blocks are ordered by address so that they can be merged, if they are adjacent
  + Fragmentation – “holes” of unused memory in the heap
  + Internal fragmentation
    - Occurs in fixed-sized memory allocation – every request receives the same size of memory
    - Not every request requires that size; some is unused & wasted
  + External fragmentation
    - Occurs in variable-sized memory allocation – each request receives the needed amount of memory
    - Free blocks exist between allocated blocks, but are too small to be useful
  + Methods of allocation
    - First fit – use the first available block
    - Best fit – use the smallest available block that is big enough
    - Worst fit – use the largest available block
* **Binary buddy method**
  + Assume size of heap is a power of 2
  + Memory is allocated in blocks with size 2n
  + Split heap into 2 equal halves; choose one half an continue splitting in halves until desired size is reached
    - Return pointer to that block
  + Keep these free blocks in a linked list (as in list method)
  + On next allocation, if requested size is already available as result of splitting, use that block
    - Otherwise, split into halves until requested size is reached
  + Adjacent blocks of equal size can be merged
  + Keep track of size, location & order of free blocks with binary encoding:
    - Entire heap (e.g. 1024 words) = 1
    - 512 blocks = 10, 11
    - 256 blocks = 100, 101, 110, 111
    - etc…
    - For the heap [32] [32] [64] [64] [64] [256] [512]:
    - 100001 (2nd 32-block), 10011 (4th 64-block), 101 (2nd 256-block), 11 (2nd 512-block)