CS241 Exam Review W15

1 Introduction

Q: Given a byte or word, how do we know what it means?
A: It could be anything, like any of these:

1. A binary Number / 2's compliment / sign magnitude / Hexidecimal number
2. A character
3. Could be an instruction
4. **Garbage!**

1.1 Fetch-Execute Cycle

Main Idea: How the computer does everything

```
PC = 0;
loop:
    IR = MEM[PC]
    PC = PC+4
    decode + execute the instruction in IR
end loop:
```

2 MIPS

2.1 Creating loops

Main Idea: Use an unconditional branch - beq $0 $0 lbl

```
loop:
YOUR END CONDITION HERE, END
====CODE====
beq $0, $0, loop
END:
```

2.2 Writing Procedures

Main Idea: Save and restore all the registers that we use

```
f: sw $2, -4($30)
   sw $3, -8($30)
   lis $4
   .word 8
   sub $30 $30 $3
   --------
   body of function
   --------
   add $30 $30 $4
   lw $3, -8($30)
   lw $2, -4($30)
   jr $31
```

2.3 Calling & Return

Main Idea:
Save to the contents of reg $31 so we don't lose it

```
main: ---code--
      lis $5
      .word f
      sw $31, -4($30) ;; push $31 onto stack
      lis $31
      .word 4
      sub $30 $30 $31

      jalr $5 ;; call

      lis $31
      .word 4
      add $30 $30 $31
      lw $31 -4($30)  ;; pop $31 off the stack
      ...
      jr $31
```

2.4 Recursion

If we did everything right, then we should be able to recursively call a function without changing anything.

2.5 Printing Out

Main Idea:
Save to the address 0xffff000c

```
;; Printing out things
lis $1
.word 0xffff000c
lis $2
.word 65
sw $2 0($1)
lis $2
.word 10
sw $2 0($1)
```

3 MERL

```
;; Merl Header
beq $0, $0, $2 ;; Cookie
.word endmodule ;; File Length
.word endcode  ;; Code length

;; code goes here

;; Merl Footer (Relocation Table)
    ...
.word 0x1 ;; relocation
.word ADDRESS
.word 0x05 ;; ESD Export (externam symbol definition)
.word ADDRESS
.word LENGTH
.word first char
    ...
.word last char
.word 0x11 ;; ESR Import (external symbol reference)
.word ADDRESS
.word LENGTH
.word first char
    ...
word last char
```

3.1 Relocation

When we relocate a file by α, then we add α to all of our instruction addresses.
This affects our **.word LABEL** instructions and we have to add α to them as well.

3.2 Linking Algorithm

Suppose that we have 2 merl files **m1** and **m2** and we want to make a single merl file.
A = codelength(m1)-12.

1. Relocate m2.code by A. (add A to every address in m2.symbolTable)
2. Resolve symbols - link imports and exports
3. Merge symbol tables - make final relocation table
4. Output final MERL file

final Merl table:

```
Merl cookie
totalCodeLen + total(imports,exports, relocates) + 12
totalCodeLen + 12
m1.code
m2.code
imports/exports/relocates
```

4 Deterministic Finite Automata and NFAS

Main Idea:
A DFA only has one possible choice to take while an NFA may be non-deterministic and have multiple path choices for a single character.

Changing NFA to DFA:
Use the subset construction. First we create our NFA, we start at state 1 and follow the transitions from each of the states at the input character. (if state 1 can go to state 2 or 3 from input char a then the next state from a is 2,3) etc.
⇒ Every DFA is an NFA and every NFA can be converted into a DFA.

4.1 ϵ-NFAs

Main Idea:
A ϵ-transition is used to glue smaller machines together. We may choose to take an ϵ transition or not to.

4.2 Maximal Munch Algorithm

Main Idea:
Run our DFA. If we are at an accepting state, output or else we backup to the most recent accepting state. Resume scanning from there.

4.3 Simplified Maximal Munch

Main Idea:
Run our DFA. If we are not in our accepting state, error.
⇒ No backtrack.

5 Syntactic & Semantic Phases of the Compiler

5.1 Syntactic: Scanning

Main Idea:
We use two passes to scan our code because labels defined further in the program have to be resolved.

1. Check if the text is properly typed
2. Every token is in our dictionary

5.2 Semantic: Parsing

Main Idea:
We use a single pass to check multiple declarations and declaration before use because of WLP4. Grammar guarantees that the declarations come first.

1. Declaration before use and only once
2. Type Checking / Type Errors
3. Scope
4. Every token is in our dictionary

6 Context Free Languages

Main Idea:
A CFG consists of an alphabet Σ of terminal symbols and a finite set of non-terminal symbols N. We have a finite set P of productions that have the form A $\Rightarrow$ B.

Leftmost derivation: Always expand leftmost non terminal first.
Rightmost derivation: Always expand the rightmost non terminal first.
$\Rightarrow$ there will be a uniq parse tree for each
$\Rightarrow$ is unambi if there is only one clear choice to make at every step

7 Parsing

Main Idea:
Finding the derivation (and creating a parse tree)
There are 2 ways for this to be done:

7.1 Forwards: "Top down parsing" and LL(1)

Main Idea:
We start at the start symbol S and expand until you reach w.

In General:
1. When top of stack is a terminal, pop & match against input
2. When top of stack is a non-term A, pop A & push RHS of rule (backwards)
3. Accept when stack & input is empty

Stack	Read input	Unread input	Next Action	Current Input
S'	ϵ	$\vdash abywx \dashv$	top of stack is nonterminal, pop and pick rule $S' \rightarrow \vdash S \dashv$	S'
$\vdash S \dashv$	ϵ	$\vdash abywx \dashv$	read $\vdash$, compare to top of stack, pop	$\vdash S \dashv$
$S \dashv$	$\vdash$	$abywx \dashv$	top of stack is nonterminal, pop and pick rule $S \rightarrow AyB$	$\vdash S \dashv$
$AyB \dashv$	$\vdash$	$abywx \dashv$	top of stack is nonterminal, pop and pick rule $A \rightarrow ab$	$\vdash AyB \dashv$
$abyB \dashv$	$\vdash$	$abywx \dashv$	read a, compare to top of stack, pop	$\vdash aayB \dashv$
$byB \dashv$	$\vdash a$	$bywx \dashv$	read b, compare to top of stack, pop	$\vdash aayB \dashv$
$yB \dashv$	$\vdash ab$	$ywx \dashv$	read y, compare to top of stack, pop	$\vdash aayB \dashv$
$B \dashv$	$\vdash aby$	$wx \dashv$	top of stack is nonterminal, pop and pick rule $B \rightarrow wx$	$\vdash aayB \dashv$
$wx \dashv$	$\vdash aby$	$wx \dashv$	read w, compare to top of stack, pop	$\vdash abywx \dashv$
$x \dashv$	$\vdash abyw$	$x \dashv$	read x, compare to top of stack, pop	$\vdash abywx \dashv$
$\dashv$	$\vdash abywx$	$\dashv$	read $\dashv$, compare to top of stack, pop	$\vdash abywx \dashv$
ϵ	$\vdash abywx \dashv$	ϵ	end of input, accept since stack empty	$\vdash abywx \dashv$

Predictor Table:
we use a predictor table and 1 char of look ahead to help decide which production to use. The empty cells in our table mean there's an error.

`LL(1):`

1. Each cell of the predictor table has one entry
2. left-to-right scan of input (First L)
3. leftmost derivation (Second L)
4. 1 character of look ahead (1)
5. Never backtrack. Can't do ambiguous things

To compute the predictor table we need the following 3 arrays:

`First(`β`):`
set of all terminals that can first letter of a word derived from β.
$= \{\alpha \mid \beta \Rightarrow^* a\gamma\}$

`Follow(A):`
set of terminal symbols that can come immediately after A in derivation starting from S'.
$= \{b \mid S' \Rightarrow^* a\text{A}b\beta\}$

`Nullable(B):`
True if symbol can eventually turn into ϵ.
$= \{B \Rightarrow^* \epsilon\}$

This solves LL(1):
Predict(A,a) = { A $\Rightarrow$ B | a $\in$ First*(B)} $\vee$ { A $\Rightarrow$ B | Nullable(B) $\wedge$ a $\in$ Follow(A)}

7.2 Backwards: "Bottom up parsing" LR(0) and SLR(1)

Main Idea:
We start at the end symbol w and work backwards until we reach S

In General:

1. To shift we read input and follow the transition.
2. If no transition, reduce if state is a reduce state, else error.
3. To reduce we pop the right hand side (# of elements) from the stack and follow the transition from the lefthand side and push it onto the stack.

Stack	Read	Unread	Action
	ε	$\vdash$ *abywx* $\dashv$	Shift $\vdash$
$\vdash$	$\vdash$	*abywx* $\dashv$	Shift *a*
$\vdash$ *a*	$\vdash$ *a*	*bywx* $\dashv$	Shift *b*
$\vdash$ *ab*	$\vdash$ *ab*	*ywx* $\dashv$	Reduce $A \rightarrow ab$ - pop b,a,push A
$\vdash$ *A*	$\vdash$ *ab*	*ywx* $\dashv$	Shift *y*
$\vdash$ *Ay*	$\vdash$ *aby*	*wx* $\dashv$	Shift *w*
$\vdash$ *Ayw*	$\vdash$ *abyw*	*x* $\dashv$	Shift *x*
$\vdash$ *Aywx*	$\vdash$ *abywx*	$\dashv$	Reduce $B \rightarrow wx$ - pop x,w,push B
$\vdash$ *AyB*	$\vdash$ *abywx*	$\dashv$	Reduce $S \rightarrow AyB$ - pop B,y,A,push S
$\vdash$ *S*	$\vdash$ *abywx*	$\dashv$	Shift $\dashv$
$\vdash$ *S* $\dashv$	$\vdash$ *abywx* $\dashv$	ε	Reduce $S' \rightarrow \vdash S \dashv$ - pop $\dashv$,S,$\vdash$,push S'

`LR(0)`:

1. left-to-right scan of input (First L)
2. rightmost derivation (Second L)
3. no character of look ahead (0)
4. we know where we've been because of our stack

We run into 2 problems:
shift-reduce and reduce-reduce
This happens when we don't know whether to shift or reduce at a state, or not sure which rule to reduce by, respectively.

Our soln:
We use 1 char of look ahead: This is now SLR(1) - Simple LR(1).

8 Optimization

Main Idea:
The compiler uses several tricks to cut down on the number of instructions it has to do. Our goal is to minimize the runtime or code size of compiler output.

8.1 Constant Folding

The process of recognizing and evaluating constant expression at compile time rather than at runtime. For example:
$Y = 3 + 3 \Rightarrow Y = 6$

8.2 Constant Propagation

The process of substituting the values of known constants in expressions at compile time. For example:
$X = 5$
$Y = X - 3 \Rightarrow Y = 5 - 3$

8.3 Common subexpression elimination

The process of searching for identical expressions and analyses if it is worth putting them in a single variable holding their value instead of recomputing it. For example:
(a + b) * (a + b) $\Rightarrow$ we could store (a + b) in one reg and multiply itself

8.4 Dead Code elimination

The process of removing code that will never run. Can be code that is never called, or in a branch where the condition will never be met.

8.5 Register allocation

An optimization where we keep local variables as registers instead of loading and storing them in RAM. We will do this for the most frequently used variables.

8.6 Strength Reduction

The process of replacing expensive operations with less expensive ones. For example: adding is faster than multiplication in most architectures.

8.7 Inline Expansion (Inlining)

The process of replacing the function call with the body of the callee. We trade the cost of a little space for the overhead of the call. This may not always be advantageous. If we call the function multiple times. Some functions are harder to incline ie. Recursive calls.

8.8 Tail Call Optimization

If we have a function where the last thing it does is a recursive/function call then we can do tail call optimization. Since there is no more work to do after the call returns, the current stack frame can be reused.

9 Memory Management

Some languages make us do manual memory management, while others (python,java etc) have implicit memory management (garbage collector). First look at 2 definitions:
scope: the part of the program where the variable is visible. In wlp4, the scope is the proc it's defined in.
extent: the part of the program's execution where the variable is live (and can be accessed).

We will be looking at the implimentation of new and delete with **free lists**.
A free list is a linked list of blocks that represent memory that we are free to use. Each node in the linked list is a **block**. Each block at an address α is a word w (length) followed by w bytes(data). We will always have the first block so we can access it.

Allocate memory:
1. Allocated 4 to make a block
2. Walk through our linked list to find which block will fit the size
3. Chop off the size from the block and return a pointer to one word after the new block starts: our data

Deallocate memory:
1. We add deallocated block to the free list
2. Mark the memory is available for allocation
3. Store the next pointer in the block data since it is now unused

We occasionally need to merge adjacent blocks in memory. Our naive solution is $O(n^2)$, comparing each block with the others to see if they are adj.
However, our implementation will suffer from fragmentation since there may be free space between blocks. To fix this we use the best fit algorithm which chooses the smallest suitable block during allocation. We can also do a first-fit algorithm, which inserts at the beginning.

9.1 Garbage Collector

Main Idea:
The implicit memory management reclaims memory when it is no longer accesible. It gives the impression that the machine has an infinite heap.

9.1.1 Mark and Sweep

Main Idea:
We have a flag to mark if an object is being referenced. At the start of a garbage collection cycle, we scan to find every live pointer and set the flag on objects they reference(mark). Then we scan the heap and reclaim memory that doesn't have its flag set and resets flags(sweep)
However, this means we have to stop the program, this is bad.

9.1.2 Reference Counting

Main Idea:
Each object holds a count of how many times it has been referenced. When we reference it we increase and decrease when the reference goes out of scope. If the counter is at 0, we reclaim the object.
However, this does not always work. Cyclic references will cause memory leaks. To solve this we use weak references; references that don't increment the reference count of the object they reference.

9.1.3 Copying Garbage Collection

Main Idea:
We split heap into halves **From** and **To**. We allocate to "From", when it is full, we copy all objects to "To" and switch labels. Now "To" will be freed. The advantage to this is that there is no fragmentation in copying. **However**, we can only use half the heap and we have the same problem as mark and sweep(have to stop the program).