CS 241 – Foundations of Sequential Programs

Brad Lushman
ISA: Akshaya Senthil
cs241@uwaterloo.ca

May 4 2015

$\sum, q_0, \subseteq, \delta, \varepsilon, \notin, \in, \emptyset, \rightarrow, \leftarrow, \neq, \cdot, \Omega, \Rightarrow, \equiv, \alpha, \beta, \gamma, \downarrow, \uparrow, \vdash, \dashv, \forall, \exists, \Leftarrow, \mapsto, \leftrightarrow, \Downarrow$

What is a *sequential program*?

- "Ordinary" program – NOT concurrent or parallel (does more than one things at once)
- Runs typically on a single CPU – goes back –and forth, doing single things at once – seemingly doing more than one thing at a time
- "single-threaded" – only one thing going on at a time

Foundations – understand how sequential "work"

- "From the ground up"

Starting point: bare hardware (sort of)

- For CS241 – simulated MIPS machine
  - Only interprets 1's and 0's

By the end – get programs written in a C-like language to run on the MIPS

- "Will write a compiler"

Binary & Hexadecimal Numbers

Bit: 0 or 1

- "Binary Digit"
- Group bit together

Byte: 8 bits

- e.g.: 11001001
- $2^8 = 256$ possible bytes

Word: machine specific grouping of bytes

- We assume a 32-bit computer architecture
- 1 word = 32 bits = 4 bytes

4 bits (1/2 a byte) – sometimes called a **nibble**

Q: Given a byte (or word) in the computer's memory, what does it mean?
E.g.: 11001001 – means what?

A: It could mean many things:

1) A number – what number is it?
   Binary number system:
   $11001001 = 1x2^7 + 1x2^6 + 0x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 201$
   (Other way: keep diving by 2 and collect the remainders – binary number)
   How can we represent negative numbers?

Simple way – reserve the first bit to represent the sign
– 0 for +, 1 for –
– "**sign-magnitude representation**"
Then 11001001 = -1(64+8+1) = -73
BUT: two 0's: 0000000000, 1000000000

- Wasteful
- Arithmetic on sign-mag number is tricky (e.g. adding a positive & a negative)

Better way: **2's Complemental Notation**

1. Interpret the n-bit number as an unsigned integer
2. If the first bit is 0, done
3. Else subtract 2^n.

E.g. for n = 3

000	001	010	011	100	101	110	111
0	1	2	3	4 – 8 = -4	5 – 8 = -3	6 – 8 = -2	7 – 8 = -1

Number line: 100 101 110 111 000 001 010 011

- Only one 0
- Left bit gives sign
- Arithmetic is cleaner
- E.g. same addition circuitry works for both unsigned and 2's comp

In 2's comp, 11001001 = 201 – 256 = -55

Convenience: Hexadecimal Notation

- Base 16 – 0,…, 9, A,…, F
- More compact than binary
- Each hex digit = 4 bits (1 nibble)

e.g.: $11001001_2 = C9_{16}$
Notation: 0xc9 → "0x" means "hex representation"

Q: Given a byte 11001001, how can we tell if it's unsigned, sign-magnitude, or 2's comp?
A: We can't! Need to remember what our intent was when we stored the byte!
We don't even know for sure that 11001001 is a number!
It could also be:

2) A character – which character?
Need a mapping between numbers and characters – a convention
ASCII – American standard code for information interchange – how computers talk to each other

- Uses 7 bits (the 8th (first) bit is supposed to be a 0)
- 8th bit was typically used for additional characters – non-standard

11001001 is not 7-bit ASCII

01001001 is ASCII for "I" (go on terminal -connected to school- and type man ASCII)
Other encodings:

- EBCDIC: extended binary coded decimal interchange code
- Unicode

3) An instruction (or part of one)
    - Our instructions are 32 bits = 4 bytes

4) Garbage (unused memory)

We can't know without being the ones who put it there.
Download – MIPS (machine language) reference sheet

May 6 2015

Machine Language

Computer programs operate on data.
Computer programs are data. – Van Neumann architecture

- Reside in the same memory space as the data
- Therefore can write programs that manipulate other programs

How does the computer know what is program and what is data? It doesn't.
What does an instruction look like? What instructions are there? Many different machine languages – processor specific
For us: MIPS (simplified): 18 different 32-bit instruction types.

The MIPS Machine: Figure 1
CPU (Central Processing Unit)

- **ALU** (Arithmetic/Logic Unit) – does math (any kind of calculation)
- Control Unit - encodes instructions
    - Dispatches to other parts of the computer to carry them out
- Memory – many kinds – (from fastest to slowest)
    - *CPU*
    - Cache
    - *Main memory (RAM)*
    - Disk memory
    - Network Memory

On the CPU – small amount of very fast memory – **registers** (small number of them, but fast)
MIPS – 32 "general-purpose" registers $0, $1,…, $31

- Each holds 32 bits – 1 word
- CPU can only operate on data in registers
    - $0 always holds 0
    - $31 is also special (page 5)
    - $30 is also sort of special (later)

Register operation

- Add the contents of registers $s + $t, and put results in $d. $d = $s + $t
  How many bits does it take to enable a register #? $2^5 = 32$, therefore 5.
  Therefore 15 bits to encode 3 register #'s.
    - Leaves 17 bits to encode the operation.

RAM

- Large amount of RAM away from the CPU.
- Data travels between CPU and RAM on the BUS.
- Big array of n bytes ($n = 10^{9+}$)
- Each cell has an address 0, …, n-1
- Each 4-byte block
- Words have addresses 0, 4, 8, C, 10, 14, 18, 1C, 20…
- Much slower than registers
- Data in RAM must be transferred to registers before it can be used

Communicating with RAM – 2 commands

- **Load** – transfer a word from RAM to register
    - Desired address goes in the memory address register (**MAR**)
        - This goes out on the BUS
    - Data at that location comes back on the BUS and goes into the Memory Data Register (**MDR**)
    - Value in the MDR moved to the destination register
- **Store** – reverse of load

Computer doesn't know which words contain code and which contain data. Then how does it execute code (and not data)?

Because you tell it - Special register called **PC** (Program Counter) that holds the address of the next instruction to run.

By convention, we guarantee that a specific address (say, 0) contains code, initialize PC to 0.
Computer then runs the **Fetch-Execute Cycle**: Figure 2

IR (Instruction Register) – holds the current instruction

- The only program the machine really runs

Again: PC holds the address of the next instruction, while the current instruction is executing.

Q: How does a program get executed?
A: Program called a **loader** puts program in memory and sets PC to the address of the first instruction in the program.

- Example: In Windows, when you click an icon, the graphical shell tells the loader which program to run and it runs it.

Q: What happens when your program ends?
A: Need to return control to the loader.

- Set PC to the address of the next instruction in the loader – which address is that?
- $31 will contain the right address
- Need to set PC to $31

- **Jump Register** (jr) instruction: jr $31

Example 1: Add the value in $5 to the value in $7, store the result in $3 and return.
MIPS Machine Code:
(s, t and d always occupy 5 bits)

Location	Binary	Hexadecimal	Meaning
00000000	0000 0000 1010 0111 0001 1000 0010 0000	00a71820	add $3, $5, $7
00000004	0000 0011 1110 0000 0000 0000 0000 1000	03e00008	jr $31

May 11, 2015

Example 2: Add 42 + 52, store the sum in $3 and return.

lis $d

- "load immediate & skip"
- Treat the next word as an immediate value and load it into $d – then skip to the following instruction.
- Binary for lis $5
- Binary for 42

Location	Binary	Hexadecimal	Meaning
00000000	0000 0000 0000 0000 0010 1000 0001 0100	00002814	lis $5
00000004	0000 0000 0000 0000 0000 0000 0010 1010	0000002a	.word 42
00000008	0000 0000 0000 0000 0011 1000 0001 0100	00003814	lis $7
0000000c	0000 0000 0000 0000 0000 0000 0011 0100	00000034	.word 52

+ append example 1 solution (with locations 00000010 00000014)

```
$ source ~cs241/setup
$ cat ex2.hex
.word 0x00002814
.word 0x0000002a
.word 0x00003814
.word 0x00000034
$ java cs241.wordams < ex2.hex > ex2.mips
$ cats ex2.mips
(*84_ //if you can read the output, its wrong – looking for a file
that's full of unprintable characters
//need a tool that tells what the bytes are
$ xxd ex2.mips
0000010: 00a7 1820 03e0 0008
$ xxd –cols 4 ex2.mips
00000000: 0000 2814 ..(.
00000004: 0000 002a …*
```

```
00000008: 0000 3814 ..8.
$ xxd –bits –cols 4 ex2.mips
000000000: 000000 000000 0000000 //binary, etc…
$ java mips.twoints ex2.mips
```

Assembly Language

- Replace binary/hex encodings with easier-to-read shorthand – less chance of error
- Translation to binary can be automated (**assembler** – program that turns this code into binary)
- One line of assembly = 1 word of MIPS (roughly speaking)

Example 2:

```
lis $5
.word 42
lis $7
.word 52
add $3, $5, $7
jr $31
```

.word is not an instruction – MIPS chip has no idea what you mean by .word

- It's a **directive** – an instruction that's not given to the machine, but rather given to the *assembler*
- Telling the assembler what to do - to literally put a word in the file
- Tell the assembler that the next word in the file should be literally 42

**Always put destination register first!! ($3 ← $5 + $7)

Example 3: Compute the absolute value of $1, store it in $1 and return.

- Some instructions modify PC (if you change PC, you change what happens next)
  - Branches & jumps, e.g. jr
- **beq** - branch if two registers are equal
  - if they are equal, increment PC by given number of words (can be negative or positive)
  - can branch backwards if negative
- **bne** - branch not equal
- **slt** - set less than

  slt $a, $b, $c ≡ $a ← 1 (if $b < $c) or 0 (otherwise)

Address		
00000000	`slt $2, $1, $0`	is $1 < 0 ?
00000004	`beq $2, $0, 1`	If false, skip
00000008	`sub $1, $0, $1`	$1 = -$1
0000000c	`jr $31`	

Example 4: (looping) sum 1,…, 13, store in $3 and return.

Address		

00000000	`lis $2`	$2 ← 13
00000004	`.word 13`	
00000008	`add $3, $0, $0`	$3 ← $0
0000000c	`add $3, $2, $3`	$3 += $2
00000010	`lis $1`	
00000014	`.word 1`	
00000018	`sub $2, $2, $1`	-- $2
0000001c	`bne $2, $0, -5`	If $2 ≠ 0, loop (to 0c)
00000020	`jr $31`	

RAM

- **lw** – "load word" – from RAM into registers
  lw $a, i($b) ≡ $a ← Mem[$b + i]
- **sw** – "store word" – from registers into RAM
  sw $a, i($b) ≡ Mem[$b + i] ← $a

Example 5:
$1 = address of an array
$ = length of the array
Place element #5 (0-based) into $3 //we're going to assume there enough elements in array

Easy Way:

```
lw $3, 20($3)    //20 because 5x4
jr $31
```

Hard Way: (if the index if not know)
Suppose $5 contains the index of the item you want to fetch ($5 is i – e.g. array[i])

```
lis $4
.word 4
mult $5, $4 ($5 *= 4)
mflo $5
add $5, $1, $5 ($5 += $1)
lw $3, 0($5)
jr $31
```

mult – the answer of the multiplication could be too big (e.g. 64 bits)

- so two special registers hi & lo, store result of mult
- mult $a, $b ≡ hi:lo ← $a * $b
- **mflo** – move from lo
  - mflo $a ≡ $a ← lo
- **mfhi** – move from hi
  - mfhi $a ≡ $a ← hi
- for division: lo holds quotient, hi holds remainder

Moving code into/out of loops, etc… implies you have *to change offsets*

- can be tricky

Instead, the assembler allows labelled instructions

e.g. `foo: add $1, $2, $3`

- assembler associates the name "foo" with the address of the add $1, $2, $3 in memory

Revisit **Example 4:**

Address		
00000000	`lis $2`	$2 ← 13
00000004	`.word 13`	
00000008	`add $3, $0, $0`	$3 ← $0
0000000c	`top: add $3, $2, $3`	$3 += $2
00000010	`lis $1`	
00000014	`.word 1`	
00000018	`sub $2, $2, $1`	-- $2
0000001c	`bne $2, $0, top`	If $2 ≠ 0, loop (to 0c)
00000020	`jr $31`	

- assembler computes the difference between PC & top in words
  - i.e. (top – PC) / 4 = (0c – 20)/4 = (12 – 32)/4 = -5

**Use labels!

May 13, 2015

Procedures in MIPS

Two problems to solve:

1) Call and return – how to get into and out of a procedure f.
   - What if f calls a procedure g?
   - Parameters and results.
2) Registers – what is f overwrites our registers and destroys our data?

For 2) we could: reserve some registers exclusively for f and some for the main line – then they won't interfere.

- What if f calls g, g calls h, etc… - running out of registers

Instead:

- Guarantee that procedures leave registers unchanged when done
- How? Use RAM. Which?
  - How do we keep procedures from using the same RAM?
  - Figure 3
  - Allocate memory from either the top or the bottom of FREE RAM
  - Need to keep track of which RAM is free and which isn't

MIPS loader helps us out

- **$30** is initialized by the loader to just pass the word of free RAM – anything above we can use, anything below is already being used
- Can use $30 as a "bookmark" to separate used and unused RAM, assuming we allocate from the bottom of free RAM

Example 6: Procedures f, g, h

Say f calls g, g calls h, h returns, g returns, f returns

Each procedure stores in RAM the registers it wants to use, restores original values on return

Figure 4

RAM is used in LIFO (last in first out) order – a *stack*

$30 - the stack pointer

- Contains the address of the top of the stack

Template for writing procedures:

(Assume f uses $2 and $3)

```
f: sw $2, -4($30)
   sw $3, -8($30)     ;push the registers f will modify onto stack
   lis $3
   .word 8
   Sub $30, $30, $8   ;decrement $30

     body

   add $30, $30, $3   ;increment $30 (assuming $3 is still 8)
   lw $3, -8($30)
   lw $2, -4($30)     ;pop registers
   jr $31             ;return
```

1) Call and return

Call main:

```
….
lis $5
.word f          ;address of the line labelled f
jr $5            ;jump to that line
HERE
```

Return: need to set PC to the line after the jr (i.e. to HERE, above)

- How does f know what address that is?

Solution: **jalr** instruction ("jump and link to register")

- Like jr, but sets $31 to the address of the next instruction, before jumping
- So replace jr $5 above with jalr $5
- Then return is jr $31

```
…
lis $5
.word f          ;address of the line labelled f
jalr $5          ;jump to that line
HERE
```

Q: jalr overwrites $31, then how can we return to the loader? And what if f calls g?
A: Need to save $31 first (on the stack) and restore when the call returns.

Template for the whole programs:
(Follow this template exactly)

```
main:
      …
      lis $5
      .word f
      sw $31, -4($30)       ; push $31
      lis $31               ; decrement $30
      .word 4               ;
      sub $30, $30, $31     ;
      jalr $5               ;
      lis $31               ; increment $30
      .word 4               ;
      add $30, $30, $31     ;
      lw $31, -4($30)
      …
      jr $31
f:                          ; push registers
                            ; decrement $30
                            ; body
                            ; increment $30
                            ; pop registers
      jr $31                ; return
```

A MIPS machine only knows "where am I" and "what instruction is it", it doesn't know what a procedure is. So you want the first thing it sees, to be something you want to run (not a procedure – that's why it's main first and then the procedure).
99% of times your program isn't working, it's because you've done something wrong with the stack.

Parameters – use registers (document!)

- If too many parameters, push them on the stack
- If you do push them on the stack, remember to take them off when you're done

Example 7: Sum 1 to n

```
; sum1toN: sums 1, …, N
; Registers
; $1 – working  (has to be saved and restored)
; $2 – for input (the value of n - has to be saved and restored)
; $3 – output (do not save this one)
```

```
Sum1toN:    sw $1, -4($30)
            sw $2, -8($30)
            lis $1
            .word 8
            sub $30, $30, $1
            add $3, $0, $0
top:        add $3, $3, $2
            lis $1
            .word 1
            sub $2, $2, $1
            bne $2, $0, top
            lis $1
            .word 8
            add $30, $30, $1
            lw $2, -8($30)
            lw $1, -4($30)
            jr $31
```

Recursion – no extra machinery needed

- If registers, parameters and stack are all managed properly, recursion will just work
- Make one tiny mistake, recursion will blow up

Output – use sw to store a word at address 0xffff000c

- Least significant (last) byte will be printed

Example 8:

```
            ls $1
            .word 0xffff000c
            list $2
            .word 67
            sw $2, 0($1)
            jr $31
```

You will see on the screen: C

May 20, 2015

The Assembler

```
add $1, $2, $3   ->    Assembler  -> Machine Code
jr $31                                  111001
```

Any translation process involves 2 phases

1) **Analysis** – understand what is meant by the source string
2) **Synthesis** – output the equivalent target string

Assembly file → stream of characters (individual characters)

- First step generally, to group the characters into meaningful **tokens** (any group of characters that has meaning)
  - e.g. label, hex number, register number, .word (the are all tokens)
- This is done for you (for now)

Your job is to group the tokens into instructions (if possible) → Analysis

- Output equivalent machine code → Synthesis

If the tokens are not arranged into sensible instructions, output ERROR to `stderr`.

** Advice: there are many more wrong configurations than right ones (more wrong ways to arrange tokens than right ones – most of them won't represent valid instructions).

- Focus on finding all of the right ones – everything else is ERROR

Biggest Problem with Writing Assemblers

How do we assemble:

```
        beq $0, $1, abec
        …
abc:    add $3, $3, $3
```

?

Can't assemble the beq because we don't (yet) know what abc is.

"Standard" solution: assemble in two passes (make 2 passes over the program).

Pass 1:

- Group tokens into instructions
- Record address for all labels
  - You're going to therefore build a **symbol table** which is a list of (label, address) pairs
- Notes:
  1) A line of assembly may have more than one label
     e.g.
     ```
     f:
     g:
           mult $1, $2
     ```
     Both f and g label to the `mult` instruction
  2) You can label after the end of the program
     e.g.
     ```
           jr $31
     z:
     ```
     z would be whatever follows the `jr $31`

Pass 2:

- Translate each instruction into machine code
- If an address refers to a label, look up at the associated address in the symbol table

Your assembler:

- Output the assembled MIPS to `stdout`
- "It should not be readable, or else its wrong"
- Output the symbol table to `stderr`

Example 9:

```
main:       lis $2                (0)
            .word 13              (4)
            add $3, $0, $0        (8)
top:                              (12)
            add $3, $3, $2        (12)
            lis $1                (16)
            .word 1               (20)
            sub $2, $1, $1        (24)
            bne $2, $0, top       (28)
            jr $31                (32)
beyond:
```

Pass 1: Group tokens into instructions – Build symbol table:

Name	Address
`main`	`0x00`
`top`	`0x0c`
`beyond`	`0x24`

Pass 2: Translate each instruction

e.g. lis $2 → 0x00001014

.word 13 → 0x000000d

…

bne $2, $0, top → look up top in symbol table (PC is + 4 of current position)

- 0x0c

Current pos = 28 + 4 = 32

- Calculate $\frac{\text{top}-\text{PC}}{4} = \frac{12-32}{4} = \frac{-20}{4} = -5$

→ 0x1440fffb (-5 = fffb)

Note: To negate a 2's complement number, flip the bits and add 1

e.g. 5 = 0000 0000 0000 0101

-5 = 1111 1111 1111 1010 + 1 = 1111 1111 1111 1011 = fffb

Bit-level Operations

To assembler bne $2, $0, top (where $\frac{\text{top}-\text{PC}}{4} = -5$

opcode = 000101 = 5

first reg = $2 = 2

second reg = $0 = 0

offset = -5

| 6 bits | 5 bits | 5 bits | 16 bits|
32 bits

To put 000101 in the first 6 bits, need to append 26 zeros (32 – 6):
i.e left shift by 26
C: 5 << 26
Racket: (arithmetic-shift 5 -26)
So you get a number whose binary representation is:
000101…. (the … represents 26 0's)
Move $2, 21 places to the left (32 – (6 + 5)):
2 << 21
000010…. (the … represents 21 0's)
Move $0 16 places to the left (32 – (6 + 5 + 5))::
0 << 16
00000…. (the … represents 16 0's)
-5 (need to occupy the last 16 bits)
-5 = 0xfffffffb → 32 bits
Want only the last 16 bits.

Bitwise and & or

a	b	a and b	a or b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

- Extend to a digit-by-digit operation

	1100 1001	- AND with 1	
AND	1111 0000	- get other bit unchanged	Can turn bits off
	----------	- AND with 0	
	1100 0000	- get 0	
	1100 1001	- OR with 1	
OR	1111 0000	- get 1	Can turn bits on
	----------	- OR with 0	
	1111 1001	- get other bit unchanged	

So do a bitwise AND with 0xffff
C: -5 and 0xffff

Then bitwise-or the four pieces together

```
int x = (5 << 26)|(2 << 21)|(0 << 16)|(-5 & 0xffff)
      = 339 804 155
```

This x has the exact bit pattern we're looking for.

```
cout << x << endl;    //prints one by one followed by new line - WRONG
```

- What the computer does: Figure out digits, convert digits to ASCII and print them out one by one

```
int x = 65;
char c = 65;
cout << x << c;
```

Output: 65A

Print chars, not ints

```
int x = …;
char c = x >> 24;     //only want the first 8 bits, so take away the
                      //next 24
cout << c;
c = x >> 16;
cout << c;
c = x >> 8;           //want the next 8 bits
cout << c;
c = x;
cout << c;
```

May 25, 2015

Loaders

Basic OS code:

```
repeat:
     P ← next program to run
     copy P into memory, starting at 0     ;this is the loader
     jalr $0
     beq $0, $0, repeat
```

(mips.twoints & mips.array are the loaders)

Problems:

- OS is a program – it is also loaded in memory – so where does it sit in memory?
- Other programs in memory – can't all be at address 0

How do we fix this?

- Choose different starting addresses for programs at assembly time

    - How will the loader know where to put them?
    - What if two programs have the same load address?
- Let the loader decide where to put the program.
- Loader's job:
    - Take a program P as input
    - Find a location α in memory for P
    - Copy P into memory starting at α (load address)
    - Return a to the OS

OS 2.0

```
repeat:
      P ← next program to run
      $3 ← loader (P)
      jalr $3
      beq $0, $0, repeat
```

Loader pseudo code:

Input: words $w_1,\ldots,w_k$ ← the machine code (P)

Takes in k words (so needs k and room for stack)

```
n = k + space for stack ← How much stack space?
                  (Pick something! Tends to not be a huge amount)
α = first address of n consecutive words of free RAM
for i = 0…k-1
      MEM[α + i*4] ← w_{i+1}
end
$3 = α + 4*n
Return α
```

Problems

- Labels may be resolved to the wrong values!
- Loader will have to fix the problem (somehow)
- What needs to change when we relocate?
    - `.word ID` – add α to id (ex: `.word A`)
    - `.word constant` – no adjustment (ex: `.word 0xabc`)
    - Anything else (including `beq`, `bne`) – no adjustment (ex: `beq $0, $0, A`)

Problems (Doesn't work yet)

- Assembler file is just a stream of bits
- How do we know which ones come from `.word` (.word with an ID) and which are instructions?
- We can't.
    - The loader doesn't know which lines needs to be adjusted – the assembler knows which
- So we need more info from the assembler

The output of most assemblers is not pure machine code – it is **object code**.
Object file:

- Contains the binary code AND whatever auxiliary information is needed by the loader (and later, the **linker**)

Our object code format: **MERL** (made up UW)
MIPS Executable Relocatable Linkable

What do we need to put into our object file?

- The code
- Which lines of the code (which addresses) were originally `.word ID`?

(Other info later)

MERL Format

Header (Always 3 words long)	0x10000002 (1st) - Called a **cookie** - Sanity check - Signals that it really is MERL Length of .merl file (2nd) Code length (3rd) – length of header + MIPS
MIPS Binary (Assembled to start at 0x0c, α = 0x0c)	
Footer (Symbol Table)	Format code (format code = 1 for relocation entry) Address to relocate (address in MIPS binary of a relocatable word) (Comes in pairs)

Note: 0x10000002 is MIPS for `beq $0, $0, 2`, which is a command to skip the header

- So MERL files can be executed as ordinary MIPS programs (if loaded at address 0)

May 27, 2015

Relocation tool: cs241.merl

- Input: MERL file + relocation address
- Output: a non-relocatable MIPS file with MERL header and footer removed, ready to load at α.

Mips.twoints, mips.array – optional second argument = α (load address)
E.g. myobj.merl to be loaded at α = 0x1000

```
java cs241.merl 0x1000 < myobj.merl > myobj.mips
java mips.twoints myobj.mips 0x1000
          ;if this is not done, it is loaded at address 0
```

Loader – algorithm:

```
read()                          //skip cookie
end Mod() <-  read() – 12       //end of module, not counting header
codeLen <- read() – 12          //code length, not counting header
d <- ind free RAM (codeLen + stack)
for (i=0; i<codeLen; i+=4)
      MEM[α=i] <- read()
end
i <- codeLen
while (i<endMod)
      format <- read()
      if (format == 1)
            rel <- read()     //address to be relocated
      //relative to the start of the header, not the code
            MEM[α + rel – 12] += α - 12
      //since header is not laded, everything shifts up by 12
      //we shifted everything down, we're loading at α
      //but it was originally designed to be loaded at 12,
      //so the diff is -12
      //in order to not load the headers
      //[a + rel – 12] is actual location
      else ERROR
      i += 8
```

Linkers

Convenient to split large MIPS files into smaller ones

- Reusable libraries
- Earn development

Issues:

- How can the assembler resolve a reference to a label if it sin a different file

Solution 1: `cat a.asm b.asm c.asm | java cs241.binasm > abc.mips`

- cat all files and then assemble
- This will work. But every time you make a change, you have to cat all files again and re-assemble
- Why re-assemble all files if only one is updated?
- Can we assemble first and then cat?
    - Binaries would need to be relocatable, i.e MERL (if loading all of them, then they would all start at address 0, but at most only one can be loaded at address 0)
    - Catting MERL files does not yield a MERL file – Figure 5

Solution 2: Need a tool that understands MERL files and puts them together intelligently – a **linker**

- But still – we have the issue of what should the assembler do with references to labels that aren't there
- Need to change the assembler

- When the assembler sees .word ID, where label ID: is not found, it outputs 0x00000000, and indicates that the program needs the value of ID before it can run

Example 9: Figure 6

- a.asm cannot execute until the reference to x is resolved
- How does the assembler notify us?
    - Make an entry in the MERL footer (more later)

e.g.
```
lis $3
.word abd
…
abc:
```

But we lost a valuable error check.
What if we mean abc when we said abd?

- Assembler will not see an error
    - Will ask for abd to be linked in.
    - How can the assembler know what is an error and what is intentional?

Need **assembler directive**: (a directive is an instruction that tells the assembler what to do)

- Tells the assembler to ask for ID to be linked in
- This does not translate to a word in MIPS – not a MIPS instruction
- So when the assembler sees .word ID
    if label ID: does not occur
    and no import ID
    Then ERROR

MERL Entry: (which tells you that a reference is not there)
Formal code 0x11 means **External Symbol Reference** (ESR)

- What information must be recorded?
    - Name of symbol
    - Where was the symbol used (i.e. the location of the blank 0x0000000)

ESR Entry:

word 1 – 0x11 ;format code
word 2 – location where the symbol is used
word 3 – length of the name in characters (n)
word 4
…
word (3+n)

word 4 to word (3+n) are ASCII characters in the name of the symbol

- Each char to its own word

The other side:

- (at this point, the file is able to say that it needs symbol and doesn't have it, now we need to have files that say, "I have this symbol, do you want to use it")

Figure 7

- How can the linker know which x to link to?
- Can't guarantee labels won't be duplicated
- How can we make x in b.asm accessible and x in c.asm inaccessible?
- We will have a new directive and MERL entry
    Export abc
    - Make abc available for linking
    - Does not assemble as a word of MIPS
    - Asks for a MERL entry

MERL Entry: External Symbol Definition (ESD)

- Format code 0x05

June 1, 2015

MERL contains

- The code
- All addresses that need relocating
- Addresses and names of every ESR
- Addresses and names of every ESD

Linker Algorithm

Input: MERL files m1 + m2

Output: Single MERL file with m2 linked after m1

```
α <- m1.codeLen – 12
relocate m2.code by α
add α to every address in m2's symbol table (i.e. footer)
if m1.exports.labels ∩ m2.exports.labels ≠ 0, ERROR

for each <addr1, label> in m1.imports
     if ∃ <addr2, label> in m2.exports
          m1.code[addr1] <- addr2
          remove<addr1, label> from m1.imports
          add addr1 to m1.relocates

for each <addr2, label> in m2.imports
     if ∃ <addr1, label> in m1.exports
          m2.code[addr2] <- addr1
          remove<addr2, label> from m2.imports
          add addr2 to m2.relocates
```

```
imports = m1.imports U m2.imports
exports = m1.exports U m2.exports
relocates = m1.reloactes U m2.relocates
output MERL cookie
output total codeLen + length of (imports, exports, relocates) + 12
output total codeLen + 12
output m1.code
output m2.code
output imports, exports, relocates
```

Formal Languages

Compiler takes c program into assembly, assembly turns it into MIPS and then it gets run

Assembly
↑
Compiler
↑
High-level language

Assembly:

- Simple structure
- Easy to recognize
- Straightforward, unambiguous, translation to machine code

High-level language:

- Complex structure
- Harder to recognize
- No single translation to machine language

How do we handle the complexity?

- Want a formal theory of string recognition – general principles that work for any programming language

Definitions:

Alphabet – finite set of symbols (eg. {a, b, c}

- Typically denoted $\sum$, as in $\sum = \{a, b, c\}$

String (or **word**) – finite set of symbols (from $\sum$)

- E.g. a, aba, cbca, abc, …
- Length of a word, $|w|$ = # of characters in w, e.g. $|aba| = 3$

Empty string: an empty sequence of symbols

- **ε (epsilon)** Denotes the empty string
- epsilon is not a symbol, $|\varepsilon| = 0$

Language – set of strings (words)

- e.g. $\{a^{2n}b \mid n >= 0\}$ ({words with an even number of a's followed by b})

Note:

- **ε** – empty word
- {} or 0/ - empty language – contains no words
- **{ε}** – language that has a word, but the word is the empty word
    - Singleton language that contains only **ε**

Our task: how can we recognize automatically whether a given string belongs to a given language?
A: depends on how complex the language is

- $\{a^{2n}b \mid n >= 0\}$ – easy
- {valid MIPS assembly programs} – almost easy
- {valid java programs}
- Some languages – impossible

Characterize languages into classes of languages based on difficulty of answering well

- Finite languages
- Regular
- Context free
- Context-sensitive
- Recursive
- Etc…

(Order form easiest to hardest)

Work at as easy a level as possible, move down as necessary
Finite language – have finitely many words

- Can recognize a word by comparing with every word in the set (finite!)
- Can we do this efficiently?

Exercise: L = {cat, car, cow}
Write code to answer w∈L, such that: w is scanned exactly once. Without sorting previously seen characters

- scan input left-to-right

```
If first char is not c, error
If next char is a
      If next char is t
            If input is empty, accept, else error
      Else if char is r
            If input is empty, accept, else error
      Else error
Else if char is o
```

```
If next char is w
     If input is empty, accept, else error
     Else error
Else error
```

- Need to abstract this

An abstraction of this program…

June 3, 2015

Regular Languages

Build from:

- *finite languages*
- *union*
- *concatenation*
- *repetition*

Union of two languages:

$L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$

Concatenation of languages:

$L_1 \cdot L_2 = \{ xy \mid x \in L_1, y \in L_2\}$

(L1L2}

E.g. $L_1 = \{dog, cat\}$

$L_2 = \{fish, \varepsilon\}$

$L_1L_2 = \{dogfish, catfish, dog, cat\}$

Repetition:

L^* $= \{\varepsilon\} \cup \{xy \mid x \in L^*, y \in L\}$

$= \{\varepsilon\} \cup L \cup LL \cup LLL \cup LLLL \cup \ldots..$

= 0 or more occurrences of a word in L

E.g. L = {a,b}

$L^* = \{\varepsilon, a, b, aa, ab, bb, aaa, aab, aba \ldots\}$

Example 10: Show $\{a^{2n}b \mid n >= 0\}$ is regular (even number of *a*'s followed by a *b*)

$(\{aa\})^k \cdot \{b\}$

Shorthand: **Regular Expressions**

Language	Regular Expression	
{}	Ø	Empty language
{ε}	ε	Language consisting of the empty string
{aaa}	aaa	Singleton language

$L_1 \cup L_2$	$L_1 \mid L_2$	Alternation (union)
$L_1 \cdot L_2$	L_1L_2	Concatenation
L^*	L^*	Repetition

So from previous example, $\{a^{2n}b \mid n >= 0\} = (aa)^*b$

Is C regular?
A C program is a sequence of tokens, each of which comes from a regular language

$C \subseteq \{\text{valid C tokens}\}^k$

So, maybe.

How can we recognize an arbitrary regular language automatically?
E.g. $\{a^{2n}b \mid n >= 0\} = (aa)^*b$

- Can we harness what we learned about recognizing finite languages?
- Yes, if we allow loops in the diagram
- Figure 8 & Figure 9.

These "machines" (*state diagrams*) are called:
Deterministic Finite Automata (**DFA's**)

- Always start at the start gate
- For each character in the input, follow the corresponding arc to the next state
- If on an accepting state when the input is exhausted, accept, else reject

What if there are no transitions?

Figure 9

- Fall off the machine = reject
- So two ways for rejection: falls of the machine, the state is not accepting

More formally, there is an implicit error state (Figure 10) and allunlabelled transitions go there

Example 11: Strings over {a,b} with an even # of a's and an odd # of b's.

Figure 11

Formal definition of a DFA
A **DFA** is a 5-tuple $(\sum, Q, q_0, A, S)$, where

$\sum$ is a finite, nonempty set (alphabet)
Q is a finite set, nonempty set (states)
$q_0 \in Q$ (start state)
$A \subseteq Q$ (accepting state)
$\delta : Q X \sum \rightarrow Q$ (transitioning function: state * input char -> next state)
(If you give me a state and char, I'll you what to do)

δ consumes a single character

- Can extend δ to a function that consumes an entire word

Definition: $\delta^*(q, \varepsilon) = q$
$\delta^*(q, cw) = \delta^*(\delta(q, c), w)$

We say DFA $(\sum, Q, q_0, A, S)$ accepts a word w if
$\delta^*(q_0, cw) \in A$

If M is a DFA, we denote by L(M) ("the language of M") the set of all strings accepted by M
$L(M) = \{w \mid M \text{ accepts } w\}$

Theorem (Kleene):
L is regular iff L=L(M) for some DFA M(the regular languages are exactly the languages accepted by DFA's).

June 8, 2015

Recall:
A **DFA** is a 5-tuple $(\sum, Q, q_0, A, S)$, where
$\sum$, Q finite, nonempty sets (alphabet, states)
$q_0 \in Q$ (start state)
$A \subseteq Q$ (accepting state)
$\delta : Q X \sum \rightarrow Q$ (transitioning function)
It accepts a word w if $\delta^*(q_0, w) \in A$

Implementing a DFA

```
int state = q0
char c
while not EOF do
      read c
      case state of
            q0: case c of
                  a: state = …
                  b: state = …
                  …
            q1: case c of
                  a: state = …
                  b: state = …
                  …
            …
            qn: case c of
                  a: state = …
                  b: state = …
                  …
      end case
end while
return (state ∈ A)
```

OR: Use a lookup table

	←States→
↑ Chars ↓	

Lexer.cc is implemented as a DFA with a lookup table
(see provided assembler starter code)

DFA's with actions

- Can attach computations to the arcs of a DFA

Example 12: L = {binary numbers with no leading 0's}
Compute the value of the number.

1(10)*|0

Figure 12

What do we gain by making DFA's more compex?

Example 13: L = {w ∈{a,b}* | w ends with abb}

(a|b)*abb

DFA for L:

Figure 13

What if we allowed more than one arc with the same char from the same state?

Figure 14

What would this mean?
Machine chooses one direction or the other (i.e. the machine is **non-deterministic**).
Accept if some set of choices leads to accepting state.

- (The machine *always* makes the right choice – it knows what to do)

With non-determinism, the above machine becomes:

Figure 15

Machine "guesses" to stay in the first state until the final abb, then transitions to accepting.

- NFA's often simpler than DFA's

Formal definition of a NFA

An **NFA** is a 5-tuple $(\sum, Q, q_0, A, S)$, where

$\sum$ is a finite, nonempty set (alphabet)
Q is a finite set, nonempty set (states)
$q_0 \in Q$ (start state)
$A \subseteq Q$ (accepting state)
$\delta : Q X \sum \rightarrow$ subsets of Q (2^Q)

Want to accept if <u>some</u> path through the NFA leads to acceptance, (reject if none do)

δ * for NFA's:

$\delta^*(q_s, \varepsilon) = q_s$

$\delta^*(q_s, cw) = \delta^*(U_{q \in qs}\, \delta(q,c), w)$

Then accept w if $\delta^*(\{q_0\}, w) \cap A \neq 0$

NFA Simulation Procedure:

```
states ← { q0}   //initialized to set of q0
while not EOF do
      read c
      states ← Uq∈qs δ(q,c)
            //for every state I could be in, whats the set of states
            //I could go to with c, and put them all together
end do
```

Example 14:

Figure 16

Simulate baabb

Already read input	Unread input	States	Name
ε	baabb	{1}	A
b	aabb	{1}	A
ba	abb	{1, 2}	B
baa	bb	{1, 2}	B
baab	b	{1, 3}	C
baabb	ε	{1, 4}	D

$\{1, 4\}$	$\cap$	$\{4\}$	=	$\{4\} \neq 0$
$(U_{q \in qs}\, \delta(\{1\}, baabb))$		$(A \neq 0)$		(therefore accept)

Figure 17

If you give each set of states a name and call those the states, every NFA becomes a DFA!
This procedure is called the **Subset Construction**

Example 15: (more complex)

L = {cab} U {words over {a, b, c} with an even # of a's}

Figure 18

Trace: caba

Read	Unread	State
ε	Caba	{1}
c	aba	{2, 6}
ca	ba	{3, 5}
cab	a	{4, 5}
caba	ε	{6} Accept

Build the DFA via the subset construction Figure 19

Obvious Fact: Every DFA is implicitly an NFA with exactly one option at all times
And: every NFA can be converted to a DFA for some language.
So NFA's and DFA's accept the same class of languages.

ε - NFA's

What if we can change states without reading a character?
ε-transitions: Figure 20

- "Free pass" to a new state without reading a character.
- Makes it easy to "glue"smaller machines together
  Figure 21

Read	Unread	State
ε	caba	{1, 2, 6}
c	aba	{3, 6}
ca	ba	{4, 7}
cab	a	{5, 7}
caba	ε	{6}

Therefore by the same naming trick as before, every ε-NFA has an equivalent DFA
Therfore ε-NFA's recognize the same class of languages as DFA's

- And the conversion can be automated

If we can find an ε-NFA for every regular expression, then we have one direction of Kleene's Theorem (reg. exp. → ε-NFA → DFA)

Regular Expr. Types

1) Ø ε-NFA: Figure 22
2) ε ε-NFA: Figure 23
3) a ε-NFA: Figure 24
4) $E_1|E_2$ Figure 25
5) $E_1 \cdot E_2$ Figure 26
6) E* Figure 27

Therfore, every regular expression has an equivalent ε-NFA and therefore an equivalent DFA. And the conversion can be automated!

Scanning

Is C a regular language? Well,
C:

- Keywords
- Id's
- Lierals
- Operator
- Comments
- Punctuation

They are all regular languages
Therefore, sequences of these also regular.

So we can use finite automata for tokenization (scanning)
Ordinary DFA's – answer yes/no to w∈L
We need:

- Input string w
- Break w into w_1, …., w_n such that each w∈L, else error
- Output each w_i

Consider: L = {valid C tokens} is regular.
Let M_L (= Figure 28) be the DFA that recognizes L.
Then (Figure 29) recignizes LL* (non-empty sequences of tokens)
Add an action to every ε-move:
Figure 30

- Machine is now non-deterministic
- ε-moves are always optiona;
- So, does this scheme guarantee a unique decomposition w = w_1, …., w_n?
- No!
- Consider just the id portion:
  Figure 31
  - Input abab could be 1, 2, 3 or 4 tokens

What do we do about this?

- Only take the ε-move if no other choice.
  - Always return the longest possible token
- Could mean valid matches missed
  L = {aa, aaa} w = aaaa
- Take aaa, fails
- But could habe taken aa, aa.
- Yes there are possible problems, but no we don't worry about them. Most programs are made easy to scan.

Concrete realization: **Maximal Munch Algorithm**
Run the DFA with no ε-moves until no non-error move is available.

If in an accepting state, token found.
Else
back up to the most recent accepting state ;use variable to track this
The input to that point is the next token
Resume scanning from there (start again with remaining at the very beginning)
End if
Output token ; ε-move back to q_0

Simplified MM Algorithm
As above, but

If not in accepting state when no transitions are possible, error (i.e no backtrack)
(Behaviour when no transitions are possible are also dependent on the programmer)

Example 16:
Identifiers:

- Begin and end with a letter
- can include –

Operator:

- --

Input:

- ab--,
  - scan up until the ","
  - no further characters can be consumed without falling off
  - ab-- is not a valid token, so therefore not in an accepting state
  - Simplified MM: ERROR
  - MM: backup to the previous accepting state, which would have been ab, and scan from there: ab, --
- In this example, MM works, SMM doesn't

In practice, SMM is very often good enough.
(For the next assignment, SMM is fine)

Languages usually designed for easy scanning.

Example 17: C++

vector<vector<int>> v;

- This won't compile, the ">>" is seen as a bit-shift operator
- C++'s longest match scanner scans this as one token >> , rather than as two >'s.

C++ Solution:

vector<vector<int> > v;

- Now there is a space "> >"
- Make it look like two tokens

Q: What (if any) specific features of C (or Scheme) program cannot be verified with a DFA?

Consider: $\sum = \{(,)\}$
$L = \{w \in \sum^* \mid w \text{ is a string of balances parenthesis}\}$

e.g. $\varepsilon \in L$, $()\in L$, $()()\in L$, $(())\in L$, $(())()\in L$, $)(\notin L$, $())\notin L$

Can we build a DFA for L?
Each new state lets us recognize one more level of nesting. But no finite # of states recognizes all levels of nesting. And DFA's have finitely many states.
Time to move on: Context-Free Languages

Context-Free Languages are languages that can be described by a context-free grammar.

Intuition: balanced parenthesis

A word in the language is either:

$S \rightarrow \varepsilon$	either empty
$S \rightarrow (S)$	or a word in the language surrounded by ()
$S \rightarrow SS$	or the concatenation of two words in the language

Shorthand: **$S \rightarrow \varepsilon \mid (S) \mid SS$**
Show: that this system generates (())()

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

Notation: "⇒" ≡ "derives"
"$\alpha \Rightarrow \beta$" means β can be obtained from α by one application of a grammar rule.

Formally: A **Context-Free Grammar** consists of:

- An alphabet $\sum$ of terminal symbols
- A finite, nonempty set N of non-terminal symbols
  $\sum \cap N \neq \emptyset$ (We use V ("vocabulary") to denote NU ε)
- A finite set P of productions
  Productions have the form $\alpha \rightarrow \beta$, where $\alpha \in N$, $\beta \in V^*$
- An element $S \in N$ (start symbol)

(and) are terminal symbols, S is a non-terminal symbol and also start symbol

Conventions: a, b, c, …. – elements of ∑ (characters)
w, x, y, … - elements of ∑* (strings)
A, B, C, … - elements of N
S – start symbol
α, β, γ , … - elements of V* ((∑UN)*)

We write αAβ ⇒ αγβ if there is a production A → γ in P.
(RHS derivable from LHS in oen step)

α ⇒ *β means write α write α ⇒ … ⇒β (0 or more steps)

Definition: L(G) = {w∈∑* | S ⇒ *w}
↑ ↑
Language specified y G Strings of terminals derivable from S

Definition: A language L is <u>context-free</u> if L=L(G) for some context-free grammar G.

Example 18: Palindromes over {a, b, c}
S → aSa | bSb | cSc | M
M → a | b | c | ε
M is the middle, the middle can be anything or nothing

Show:
S ⇒* abcba
S ⇒ aSa ⇒ abSba ⇒ abMba ⇒ abcba

- This is called a **derivation**

<u>Expressions:</u>
∑ = {a, b, c, +, -, /, *}
L = {arithmetic expressions over ∑}
OP (operator can be) → + | - | * | /
S → S OP S | a | b | c
a, b, or c are expressions on their own

∑ = {a, b, c, +, -, /, *, (,)}
L = {arithmetic expressions over ∑}
OP (operator can be) → + | - | * | /
S → S OP S | a | b | c | (S)

<u>Show</u>: S ⇒* a + b
S ⇒ S OP S ⇒ a OP S ⇒ a + S ⇒ a + b
↑

Choice of which symbol to expand

Leftmost derivation – always expand the leftmost symbol
Rightmost derivation – always expand the rightmost symbol

June 17, 2015

Derivations can be expressed naturally and succinctly as trees:
Figure 31

These trees as called **Parse Trees**

- For every eftmost (or rightmost) derivation, there is a unique parse tree.

Example 19: Find leftmost derivation for *a+b*c*
S ⇒ S OP S ⇒ S OP S OP S ⇒ a OP S OP S ⇒ a + S OP S ⇒ a + b OP S ⇒ a + b * S ⇒ a + b * c
OR
S ⇒ S OP S ⇒ a OP S ⇒ a + S ⇒ a + S OP S ⇒ a + b OP S ⇒ a + b * S ⇒ a + b * c
These produce different parse trees:
(Since there are two leftmost derivations, then have two parse trees)
Figure 32

A grammar where some word has more than one distinct leftmost derivation (equivalently more than one distinct parse tree), is called **ambiguous**.

S → S OP S | a | b | c
OP → = | -| *|/ } Is an ambiguous grammar

If we only care about answering w∈L(G), ambiguity does not matter.
But as compiler writers, we wany to know why w∈L(G).
i.e Why does the derivation (parse tree) matter?

Why?
The shape of the parse tree describes (gives us) the meaning of the string.
So if you have a word with more than one parse tree, it may have more than one meaning.
Figure 32
(Some grouped more tightly than others)

What do we do?

1) Use heuristics (“precedence”) to guide the derivation process.
2) Make the grammar unambiguous.

It is ambiguous because you have a choice in the second step (S OP S).

E → E OP T | T expression
T → a | b | c | (E) term
OP → + | - | * | / operator

a + b * c
E ⇒ E OP T ⇒ E OP T OP T ⇒ T OP T OP T ⇒ a OP T OP T ⇒ a + T OP T ⇒ a + b OP T ⇒ a + b * T
⇒ a + b * c

Figure 33
This grammar enforces strict left-to-right precedence

What if we wanted to give *****, **/** precedence over **+**, **-**?
We need to view an expression as a sum of terms, where a term is a product of factors.
E → E PM T | T expression
T → T TD F | F term
F → a | b | c | (E) factor
PM → + | -
TD → * | /

a + b * c
E ⇒ E PM T ⇒ T PM T ⇒ F PM T ⇒ a + T ⇒ a + T TD F ⇒ a + F TD F ⇒ a + b TD F ⇒ a + b * F
⇒ a + b * c

Figure 34

Q: If L is context-free, is there always an unambiguous grammar G such that L(G) = L?
Is it always going to be possible to fix an ambigious program (always one that is unambiguous)?
A: No! There are inherently ambiguous languages that only have ambiguous grammars.

Q: Can we construct a tool that will tell us if a grammar is ambiguous?
A: No! There is no computer program that can tell you that.
This is called **Undecidable**.

Equivalence of grammars G_1 and G_2 (i.e. $L(G_1) = L(G_2)$) is also undecidable (no computer program will tell you that).
Not because something is undecidable, doesn't mean you can't answer parts of the problem you can't answer it.

***END OF MIDTERM MATERIAL

Recognizer – what class of computer programs is needed to recognize a CFL?

- Regular Languages: DFA (essentially a program with finite memory)
  Every program that uses a finite amount of memory can be called a DFA.
- Context-Free Languages: NFA + stack (so an NFA that has access to a stack)
  - Infinite memory, but use of that memory is limited to LIFO

But we need more than just a yes/no answer.

- Need the derivation (parse tree) or error message.

Problem of finding the derivation is called **parsing**.

Gven: Gramma G with start symbol S and word w
Find: $S \Rightarrow \ldots \Rightarrow w$ (find middle "…" setps) or report that there is no derivation.

How? (2 choices, both real)

1) Forwards – "**top down**"
   Start at S, expand non-terminals, obtain w
2) Backwards – "**bottom up**" (DO THIS ONE)
   Start at w, work backwards, obtain S

Top-Down Parsing

Start $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow w$ Finish

Use the stack to store intermediate α_i in reverse, match against chars in w.
An invariant is something that is always true.

Invariant: consumed input + reverse (stack contents) = αi

June 22, 2015

Example 20:
$S \rightarrow A\gamma B$
$A \rightarrow ab$
$A \rightarrow cd$
$B \rightarrow z$
$B \rightarrow wx$

For simplicity, we use augmented grammars, invert two new symbols ⊢, ⊣, new start symbol S'

1) S → ⊢ S ⊣
2) S → AyB
3) A → ab
4) A → cd
5) B → z
6) B → wx

Say **w = ⊢ abywx⊣**

Stack	Read Input	Unread Input	Action
S'	ε	⊢ abywx⊣	Pop S', push ⊣, S, ⊢
⊣ S ⊢	ε	⊢ abywx⊣	match ⊢ (beginning of file)
⊣ S	⊢	abywx⊣	Pop S Push B, y, A
⊣ B y A	⊢	abywx⊣	Pop A Push b, a
⊣ B y b a	⊢	abywx⊣	Match a
⊣ B y b	⊢a	bywx⊣	Match b
⊣ B y	⊢ab	ywx⊣	Match y
⊣ B	⊢aby	wx⊣	Pop B, push x, w
⊣ x w	⊢aby	wx⊣	Match w
⊣ x	⊢abyw	x⊣	Match x
⊣	⊢abywx	⊣	Match ⊣ (end of file)
	⊢abywx⊣	ε	Accept

When top of stack (**TOS**) is a terminal, pop and match against input.
When TOS is a non-terinal A, pop and push αR (α reverse), where A → α is a grammar rule.
Accept when stack , input both empty.
BUT What if there is > 1 production with A on the LHS? How can we know which one to pick?
Brute force (try all combinations) not efficient.

Want a deterministic procedure
Our Solution: Use the next char of input (called **lookahead**) to help decide.
Construct a **predictor table**:

- Given a non-terminal on stack and an input symbol, it tells which grammar rule to use

Example 21:

Rules:

1) S → ⊢ S ⊣
2) S → AyB
3) A → ab
4) A → cd
5) B → z
6) B → wx

	⊢	a	b	c	d	w	x	y	z	⊣
S'	1									
S		2		2						
A		3		4						
B						6			5	

Most of the cells here are empty.
What do empty cells mean? Empty cells = ERROR (parse error)
Descriptive: "Parse ERROR at *row*, *col* (of the next input character), expecting one of _, _, _"

E.g. If TOS = A – "expecting a or c"

What if a cell contains more than one rule? (ex: A → ab, A → ad)

- The predictor table might have more than one entry in the same cell – this wont work
- The method will break down

A grammar is called **LL(1)** if each cell of the predictor table hast at most one entry.

LL(1) = left-to-right sca of the input, leftmost derivations produced 1 symbol of lookahead

Automatically Computing the Predictor Table

Predcit(A, a)

- (build this table) to be the set of rules that apply when A is on the stack, a is the next char

Predict(A, a) = {A → β | a ∈ First(β)}
First(β), β∈V*

- Set of chars that can be the first letter of a string derived from β

First(β) = { a | β ⇒* aδ } //either β already equals a δ, or after some applications you get that
So Predict(A, a) = {A → β | β ⇒* aδ}

- Not quite right
- What if A ⇒* ε? What if we could just make A disappear? And what if a was supposed to match something after? Then a might not come from A, but from something after A.

So really, Predict(A, a) = {A → β | a ∈ First(β)} U {A → β | Nullable(β), a ∈ Follow(A)}

- So you can make A disappear as long as you can legally get a to something that follows

Nullable(β) = true iff $\beta \Rightarrow^* \varepsilon$
Follow(A) = {b | $S' \Rightarrow^* aAb\beta$}

- Terminal symbols that can come immediately after A in a derivation

Nullable:

```
initialize Nullable[A] = false ∀A
//we assumbe nothing nullable until we find evidence that something that actually is
repeat:
    for each rule B → B1…Bk
        if k = 0 or Nullable[Bi] ∀i
            Nullable[B] ← true
repeat until nothing changes
//we know this terminates b/c it only gives from false to true, never true to false
```

First:

```
initialize First[A] = {} ∀A
repeat:
    for each rule B → B1…Bk1
        for i = 1…k
            if Bi is a terminal a
                First[B] U = {a};
                break
            else (non-terminal)
                First[B] U = First[Bi]
                if not Nullable[Bi]
                    break
repeat until nothing changes
```

First*:

Computing First*(β) = first of a string of symbols

```
First*(β)(Y1….Yn)
    result ← Ø
    for i = 1…n
        Yi ∈ ε (non-term)
        result U = First[Yi]
        if not Nullable[Yi] break
        else (term)
            result v = {Yi}
            break
    return result
```

Follow:

Initialize Follow[A] = {} ∀A
repeat
    for each rule B → B_i…B_k
        for i = 1…k
            if B_i ∈ N
                Follow[B_i] U = First*(B_{i+1}…B_n)
            if all of B_i…B_n are Nullable (include the case i=n)
                Follow[B_i] U = Follow(B)
Repeat until nothing changes

June 24, 2015

Nullable[A] ← false ∀A
repeat:
    for each rule B → B_1…B_k
        if k = 0 or Nullable[B_i] ∀i
            Nullable[B] ← true
repeat until nothing changes

First[A] ← {} ∀A
repeat:
    for each rule B → B_1…B_{k1}
        for i = 1…k
            if B_i is a terminal a
                First[B] U = {a};
                break
            else (non-terminal)
                First[B] U = First[B_i]
                if not Nullable[B_i]
                break
repeat until nothing changes

Follow[A] ←{} ∀A ≠ S'
repeat
    for each rule B → B_i…B_k
        for i = 1…k
            if B_i ∈ N
                Follow[B_i] U = First*(B_{i+1}…B_n)
            if all of B_i…B_n are Nullable (include the case i=n)
                Follow[B_i] U = Follow(B)
Repeat until nothing changes

E.g
1) S' → ⊢ S ⊣
2) S → b S d
3) S → p S q
4) S → C
5) C → lC
6) C → ε

Nullable

Iteration:	0	1	2	3
S'	False	False	False	False
S	False	False	True	True
C	false	True	True	True

Since the last (3) didn't change from previous (2), then we know its not going to change anymore so we stop
In principle, S' will never be nullable

First

Iteration:	0	1	2	3
S'	{}	{ ⊢}	{ ⊢}	{ ⊢}
S	{}	{b, p}	{b, p, l}	{b, p, l}
C	{}	{l}	{l}	{l}

On iteration 1 on S, we don't know yet what C can start with

- We only find this out on iteration 1 on C

We stop once the array hasn't changed (just like previous)

Follow

Iteration:	0	1	2
S	{}	{⊣, d, q}	{⊣, d, q}
C	{}	{⊣, d, q}	{⊣, d, q}

Anything that can follow S can follow C
From rule 4 – (because S can become C)

Predict(A, a) =
{A → β | a ∈ First(β)}
//turning A into a string that starts with a
U {A → β | Nullable(β), a ∈ Follow(A)}
//to make a disappear so we can get next symbol

Predict

	⊢	⊣	b	d	p	q	l
S'	1						
S		4	2	4	3	4	4
C		6		6		6	5

C is nullable, so if we want to make C disappear we use rule 6

- We want to make C disappear when end of file dq

A grammar is LL(1) if

- No two distinct productions with the same LHS can generate the same first symbol
- No nullable symbol A can have the same terminal symbol a in both its first set and its follow set
- There is only one way to send a nullable symbol to ε
    - If a is nullable, there better be only one way to make it disappear
    - If there is two ways, the grammar is umbiguous

Example 22:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow id$

This grammar is obviously not LL(1)
(if we don't have any epsilon symbols, that's fine)
This is not LL(1) because of **left recursion**. Left recursion means the recursive epart of the rule is on the left. (I can keep replacing E with E, theres no knowing how many times I can replace E qith something starting with E).

*** Whenever you have a character that can replace itself, it will **always** NOT be LL(1)

So make it **right recursive**:

$E \rightarrow T + E \mid T$
$T \rightarrow F * T \mid F$
$F \rightarrow id$

This is still obviously not LL(1) – its even more obvious now than it was before
The two right hand sides start with the first two symbols ($E \rightarrow T + E \mid T$).
So factor the grammar

$E \rightarrow T\ E'$
$E' \rightarrow \varepsilon \mid + E$
$T \rightarrow F T'$
$T' \rightarrow \varepsilon \mid * T$
$F \rightarrow id$

This now is LL(1)
LL(1) conflict with left-associativity

Bottom-Up Parsing

Go from w to S
Stack stires partially-reduced input read so far
$w \Leftarrow \alpha_k \Leftarrow \alpha_{k-1} \Leftarrow \ldots \Leftarrow \alpha_1 \Leftarrow S$

Example 23:

Rules:

1) $S' \rightarrow \vdash S \dashv$
2) $S \rightarrow AyB$
3) $A \rightarrow ab$
4) $A \rightarrow cd$
5) $B \rightarrow z$
6) $B \rightarrow wx$

$w = \vdash abywx\dashv$

Stack	Read	Unread	Action
	ε	⊢ abywx⊣	Shift ⊢
⊢	⊢	abywx⊣	Shift a
⊢a	⊢a	bywx⊣	Shift b
⊢ab	⊢ab	ywx⊣	Reduce A→ab: Pop b, a, push A
⊢A	⊢ab	ywx⊣	Shift y
⊢Ay	⊢aby	wx⊣	Shift w
⊢Ayw	⊢abyw	x⊣	Shift x
⊢Aywx	⊢abywx	⊣	Reduce B→wx: Pop x, w, push B
⊢AyB	⊢abywx	⊣	Reduce S→AyB: Pop B, y, A push S
⊢S	⊢abywx		Shift ⊣
⊢S⊣	⊢abywx⊣	ε	Reduce S' → ⊢ S ⊣
⊢S'	⊢abywx⊣	ε	Accept

Actions will always be either shift or reduce

Choice at each step:

1) Shift a char from input to stack
2) Reduce – TOP (stack) is RHS of a grammar rule – replace with LHS

Accept if stack contains S' when input is ε
(Equivalently ⊢ S ⊣ on empty input)
(Equivalently, accept when shift ⊣)

How do we know whether to shift or reduce?

- Use next inout char to help decide, but problem is still hard

Theorem: (Donald Knuth, 1965)
The set {wa | ∃x. S' ⇒* wax}
w is the stack
a is next input char
The set of the stacks you can have during one of tehse bottom-ups parsings combined with any input character is a regular language.

June 29, 2015

Recall:

1) S' → ⊢ S ⊣
2) S → AyB
3) A → ab
4) A → cd
5) B → z
6) B → wx

Stack	Read	Unread	Action
	ε	⊢ abywx⊣	Shift ⊢
⊢	⊢	abywx⊣	Shift a
⊢a	⊢a	bywx⊣	Shift b
⊢ab	⊢ab	ywx⊣	Reduce A→ab: Pop b, a, push A
⊢A	⊢ab	ywx⊣	Shift y
⊢Ay	⊢aby	wx⊣	Shift w
⊢Ayw	⊢abyw	x⊣	Shift x
⊢Aywx	⊢abywx	⊣	Reduce B→wx: Pop x, w, push B
⊢AyB	⊢abywx	⊣	Reduce S→AyB: Pop B, y, A push S
⊢S	⊢abywx		Shift ⊣
⊢S⊣	⊢abywx⊣	ε	Reduce S' → ⊢ S ⊣
⊢S'	⊢abywx⊣	ε	Accept

Choice:

1) Shift symbol from input to stack
2) Reduce top of stack by grammar rule

Theorem: (Donald Knuth, 1965)
The set {wa | ∃x. S' ⇒* wax}
w is the stack
a is next input char

- Can be described by a DFA

- Use its DFA to make shift/reduce decisions

LR parsing
- Left to right scan
- Right most derivation

Example:

$S' \rightarrow \vdash E \dashv$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow id$

LR(o) machine (simplest)
Defintion: an item is a production with a dot (.) somewhere on the RHS (indicates partially completed rule)

Figure 35

Using the machine
Start in start state with empty stack

Shifting:
- Shift char from inout to stack
- Follow transitions for that char to next state
- If no transition: error on reduce

Reducing:
- "reduce states" have only one item + dot is rightmost (complete items)
- Reduce by the rule in the state

Reduce: pop RHS off the stack, back track size (RHS) states in DFA, push LHS, follow shift transition for the LHS

Backtracking the DFA – must remember DFA states
Push DFA states as the stack as well

S' → ⊢ E ⊣
E → E + T
E → T
T → id

w = ⊢ id + id + id⊣

Stack	Read	Unread	Action
1	ε	⊢ id + id + id ⊣	S2 (shift +go to 2)
1 ⊢ 2	⊢	id + id + id ⊣	S6
1 ⊢ 2 id 6	⊢id	+ id + id ⊣	R T -> id Pop 1 symbol Pop 1 states Now in state 2 Push T, go to 5
1 ⊢2 T 5	⊢id	+ id + id ⊣	R E -> T Pop 1 symbol Pop 1 state Push E go to 3
1 ⊢ 2 E 3	⊢id + id	+ id⊣	S7
1 ⊢ 2 E 3 + 7	⊢id + id +	id⊣	S6
1 ⊢ 2 E 3 + 7 id 6	⊢id + id + id	⊣	R T -> id go to 8
1 ⊢ 2 E 3 + 7 T 8	⊢ id + id + id	⊣	R E -> E + T S' -> ⊢ E ⊣ Go to 3 E -> E + T
1 ⊢ 2 E 3	⊢id + id + id	⊣	S4
1 ⊢ 2 E ⊣ 4	⊢ id + id + id ⊣	ε	Accept

What can go wrong?
What of a state looks like this?

A → a · cB
B → y ·

Shift c or reduce B → y ? This is a shift-reduce conflict

A → a
B → b

Reduce A → a or B → b? Reduce-reduce conflict

Whenever a complete item A → a is not alone in a stack, there is a conflict and the grammar is not LR(o).

E.g. Right associative

S' → ⊢ E ⊣
E → E + T
E → T
T → id

Figure 36

E.g. Input starts with ⊢ id

Figure 37

Should we reduce E → T first?
Depends: If input is ⊢ id ⊣ then Yes
If input is ⊢ id + … ⊣ then No

Add lookahead to fix the conflict
For each A → a, attach Follow(A)

Follow(E) = {⊣}
Follow(T) = {+, ⊣}

E → E + T	becomes	E → E + T
E → T		E → T ⊣

Interpretation
Reduce action
A → a x (x = follow(A))
Only applies if the nest char is in x
So E → T applies when next char is in x
So E → T applies when next char is ⊣
E → T + E applies when next char is +

Conflict resolved!
Result is called an SLR(1) parser
SLR(1) = simple LR with char lookahead
SLR(1) resolves many, but not all conflicts
LR(1)

- More powerful than (SLR(1))
- Produces many more states

<u>Building a Parse Tree</u>
Top down
Figure 37

July 6, 2015

Context-sensitive analysis
(also called semantic analysis)

What properties of valid programs cannot be enforced by CFG's?

- Declaration-before-use
- type correctness
- scoping

To solve these, need more complex programs
Next language class: **Context-Sensitive Languages**

- specified by a context-sensitive grammar

A more ad hoc approach is more useful

- traverse the parse tree

E.g. parse tree class

```
class Tree{
      public:
            string rule;     //grammar rule, i.e factor ID, ID xyz
            vector<Tree*> children;
};
```

Example: Tree traversal example: print the tree

```
Void print(Tree &t){
      //print t.rule
      for(vector<Tree*>::iterator i = t.children.begin(); i != t.children.end(); ++i){
            print(**i);
      }
}
```

What analysis is needed for WLP4? What errors need to be caught?

- Variables/procedures used but not declared
- Variables/procedures declared more than once
- Type errors

For now, assume WAIN is the only procedure

Declaration errors – multiple/missing declarations
How do we check this? We've doen this before…

Construct a symbol table

- Traverse the parse tree to collect all declarations for each node corresponding to rule:
    - dcl -> type ID

- For each node corresponsing to that rule, extract a name (e.g. x) and a type ("int" or "int*") and add to symbol table
- If name already in the symbol table, its an error
- Once this is done, you have now checked for multiple declarations
- Traverse again
- Check for rules factor -> ID and lvalue -> ID
- If ID's name is not in symbol table, error
- Undeclared variables now checked

These two passes can be merged – since in WLP4 all declarations go first

Symbol table implemenatation - global varibales

map<string, string> symTbl;

↓ ↓

Name type

BUT

- doesn't take scopes into account
- Doesn't check procedure declarations

Issue:

```
int f(){
        int x = 0;
        return x;
}
int wain(int a, int b){
        int x = 0;
        return x;
}
```

Our algorithm will flag this as an error because x is being declared twice -> Issue of scope

We must:

Forbid duplicated declarations in the sam eprocedure but allow them in different procedures

Also we must forbid this:

```
int f(){…}
int f(){…}
```

Solutions:

(one possible solution is to put in the symbol table, the format wain::x)

- Have a separate symbol table for each procedure
- Have one "top level" symbol table that collects all procedure names

Map<string, map<string, string> > topSymTbl;

↓ ↓

Proc name proc's symTbl

Whan traversing:
If the name has rule procedure -> … or main -> …

- New procedure – make sure its name is not alreadt in symTbl, if not, add it
- You may want a global variable, "currentProc"
    - Which procedure you're currently in
    - Keep it up to date

For variables – store the declaration type in the symTbl

- Is there type info for procedures?
- Yes – its the signature

Note: all procedures in WLP4 return int, so the signature is just the sequence of parameter types

- Store in top-level symTbl

map<string, pair<vector<string>, map<String, string> > > topSymTbl;
↓ ↓ ↓
Proc name signature local symbol table

To compute the signature:
Nodes of the form *paramlist -> dcl* or *paramlist->dlc COMMA paramlist*
(if *params->epsilon*, then empty signature)

**Make sure to test procs with no, one, or more than one params, to see if you got it backwards or not

All of this can be done in one pass.
This is all we need to check declarations and scope.

Types

Why do programming languages have types?

- Prevent errors
- Assigns interpreations to RAM locations and remembers those interpretations

Example:

int *a = NULL; a denotes a pointer
a = 7; wrong, attempt to put int where a pointer is needed

- Find a type for every variable/expression
- Ensure the rules are followed

How do we do that?
We traverse the tree.

```
string TypeOf(Tree &t){
        for each c (element) t children, find type of (c)
        use types of children and t.rule to determine type of (t)
}
```

Example: ID

- Get its type from the symbol table

```
<id.name, t>∈ declarations          //if id.name declared to have type t
----------------------------------
id: t                               //then id has type t
```

```
string TypeOf(Tree &t){
        if t.rule == "ID name"
                return symTbl.lookup(name)
        …
}
```

July 8, 2015

Sinlgeton Procedures

Expr → term
Term → factor
Factor → ID

Here the type of LHS = type of RHS

_____ _____ NUM: int NULL: int*	E : t_____ (E) : t "If E has type t, then (E) also has type t"
Address-of E : int _____ &E : int*	**Alloc** E : int_____ New int[E] : int* "If E has type int, then newint[E] has type int*"
Dereferencing E : int*_____ * E : int	**Multiplication** E : int_____ New int[E] : int*
Addition E_1 : int E_2 : int $E_1 + E_2$: int E_1 : int* E_2 : int $E_1 + E_2$: int* E_1 : int E_2 : int* $E_1 + E_2$: int*	int + int → int int * + int → int* int + int* → int* int* + int* → X
Subtraction E_1 : int E_2 : int $E_1 - E_2$: int E_1 : int* E_2 : int $E_1 - E_2$: int* E_1 : int* E_2 : int* $E_1 - E_2$: int	int - int → int int * - int → int* int - int* → X int* - int* → int (because it makes sense to ask by how much two addresses differ – the amount by how much two arrays differ – the number of words they differ by)
$<f,(t_1,\ldots,:t_n)> \in E_1: t_1 \ldots E_n:t_n$ $f(E_1,\ldots,E_2)$: int	

Additional Type Constraints

Loops, if if(T) {S1} else {S2}
while(T) {S}

Test T should be boolean, not in or int*

But there is no boolean type in WLP4

The grammar ensures that T is a comparison
So we just need to check that the comparison is well-typed

$\frac{E : T}{\text{well-typed (E)}}$	
Tests $\frac{}{\text{well-typed } (E_1 == E_2)}$ $\frac{E_1 : t \;\; E_2 : t}{\text{well-typed } (E_1 < E_2)}$	Also != Also >, <=, >=
Assign $\frac{E_1 : t \;\; E_2 : t}{\text{well-typed } (E_1 = E_2)}$	
Print $\frac{E : int}{\text{well-typed (println(E))}}$	
Delete $\frac{E : int *}{\text{well-typed (delete[]E)}}$	
If $\frac{\text{well-typed (T) well-typed}(S_1) \;\; \text{well-typed } (S_2)}{\text{well-typed (if (T) } \{S_1\} \text{ else } \{S_2\}}$	
Loop $\frac{\text{Well-typed (T) well-typed(S)}}{\text{well-typed (while (T) } \{S\})}$	
Sequences $\frac{}{\text{empty sequence}}$ well-typed (ε) $\frac{\text{well-typed}(S_1) \;\; \text{well-typed } (S_2)}{\text{well-typed } (S_1,S_2)}$	
Declarations $\frac{}{\text{well-typed(INT ID = Num)}}$	

_____ well-typed(INT* ID = NULL) well-typed(decls) well-typed(statements) E : int well-typed(INT ID (params) {decls statements return E; }
Wain dcl_2 = INT ID well-typed(decls) well-typed(statements) E : int well-typed(INT WAIN (dcl_1, dlc_2) {decls statements return E; }

End of semantic analysis!!

Code Generation

Source
↓
Lexical Analysis
↓ tokens
Parsing
↓ parse tree
Semantic Analysis
↓ parse tree ↓ symbol table
Code Generation
↓
Assembly

By now the program has no compile-time errors
Now output equivalent MIPS assembly

- There are infinite many equivalent MIPS programs
- Which do we choose?
  - Correct (essential)
  - Easy to write (for cs241)
  - Shorstest program?
  - Fastest program?
    - To run or to compile

Example:

```
int wain(int a, int b) { return a;}
```

Convention:

- Params of wain are in $1 and $2
- Output goes into $3

MIPS:

```
      int wain(int a, int b) { return a;}
```

```
add $3 $1 $0
jr $31
```

```
      int wain(int a, int b) { return b;}
```

```
add $3, $2, $0
jr $31
```

These two programs have the same parse tree
How do you match the use of an ID with the proper decaration?
Symbol Table! Add a field that indicates where each id is stored

Name	Type	Location
a	int	$1
b	int	$2

When traversing for code generation, lookup ID in symbol table
a ↦ $1
b ↦ $2

What about local declarations?

- Can't all be in registers (probably not enough)

For simplicity: (this is recommended)

- Put <u>all</u> vars on the stack, including the variables of wain

```
int wain(int a, int b){ return a;}
⇓
sw  $1, -4($30)
sw $2, -8($30)
lis $4
.word 8
sub $30, $30, $4
lw $3, 4($30)
add $30, $30, $4
jr $31
```

Symbol Table:

Name	Type	Offset from $30
a	int	4
b	int	0

July 13, 2015

Recall:

```
int wain(int a, int b){ return a;}
⇓
sw  $1, -4($30)
sw $2, -8($30)
lis $4
.word 8
sub $30, $30, $4
lw $3, 4($30)
add $30, $30, $4
jr $31
```

Symbol Table:

Name	Type	Offset from $30
a	int	4
b	int	0

Problem:

- Can't compute offsets until all declarations have been seen
- Because register $30 changes with each new declaration

```
int wain(int a, int b){ int c = 0; return a;}
```

Name	Type	Offset from $30
a	int	8
b	int	4
c	int	0

… lw $3, 8($30)…

Introduce two conventions:

- $4 contains 4
- $29 points to the bottom of the stack frame
    - If offsets are calculated with respect to $29, they will be constant
    - $29 called the **frame pointer**

```
int wain(int a, int b){ int c = 0; return a;}
```

Name	Type	Offset from $30
a	int	8
b	int	4
c	int	0

```
int wain(int a, int b){ int c = 0; return a;}

lis $8
.word 4
sub $29, $30, $4
sw $1, -4($30)
sw $2, -8($30)
sw $0, -12($30)
lis $5
.word 12
sub $30, $30, $5
lw $3, 0($29)
add $30, $30, $5
jr $31
```

Name	Type	Offset from $29
a	int	0
b	int	-4
c	int	-8

More complicated Programs

```
int wain(int a, int b) { return a+b; }
```

In general:

- For each grammar rule, A→ δ, build code(A) from code(δ)

Extend previous convention:

- Use $3 for "output" of <u>all</u> expressions

Example:

a+b $3 ← eval(a)
$3 ← eval(b) //value of a is lost
$3 ← $3 + $3

Need a place to store pending computation

We can use a register:

code(a)	($3 ← a)
add $5, $3, $0	($5 ← a)
code(b)	($3 ← b)
add $3, $5, $3	($3 ← a+b)

Example:

a+(b+c)

code(a)	($3 ← a)
add $5, $3, $0	($5 ← a)
code(b)	($3 ← b)

```
add $6, $3, $0          ($6 ← b)
code(c)                 ($3 ← c)
add $3, $6, $3          ($3 ← b+c)
add $3, $5, $3          ($3 ← a+(b+c))
```

Uses 2 extra registers
Similarly, a+(b+(c+d)) would require 3 extra registers
Eventually run out of registers.

More general solution:
Use <u>stack</u>

```
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))
```

Only one extra register!

In general:

$expr_1 \rightarrow expr_2 + term$

```
code(expr1)   = code(expr2)
              + push($3)
              + code(term)
              +pop($5)
              +add $3, $5, $3
```

Recurse this.

Singleton rules: easy
expr → term

code(expr) = code(term)

<u>Print</u>
println(expr) - this prints expression followed by newline

Runtime environment – set of procedures supplied by compiler (or in some cases the operating system) to assist programs in their execution:

msvert.dll

- Put procedure print in the runtime environment
  - We provide print.merl

```
./wlp4gen < prog.wlp4i > prog.asm
java cs241.linkasm < prog.asm > prog.merl
linker prog.merl print.merl > exec.mips
java mips.twoints exec.mips
```

or

```
java mips.array exec.mips
```

So assume "print" is a variable – take sinput in $1

code(println(expr)) = code(expr) ($3 ← expr)
+ add $1, $3, $0
+ call print

Assignment

statement → lvalue = expr

For now, assume `lvalue = ID` (no pointers)

statement → lvavlue = expr
code(statement) = code(expr) ($3 ← expr)
sw $3, ____($29)
↑ look up ID in symbol table

Example: x = y (we need to know the value of y and the location of x)

Whats left? if and while

- Need boolean testing
- Suggested:
  - store 1 in $11
  - store print in $10

Code so far:

```
.import print          //prologue
lis $4
.word 4
lis $10
.word print
lis $11
.word 1
sub $29, $30, $4
        Your Code
```

```
add $30, $29, $4        //epilogue
jr $31
```

test → $expr_1 < expr_2$

```
code(test)    = code(expr1)        ($3 ← expr1)
                add $6, $3, $0           //code gen for exprs only uses $3 and $5
                code (expr2)       ($3 ← expr2)
                slt $3, $6, $3     ($3 ← expr1 < expr2)
```

test → $expr_1 > expr_2$

- treat as $expr_2 < expr_1$

July 15, 2015

test → expr1 != expr2

```
code(test)    = code(expr1)        ($3 ← expr1)
                add $6, $3, $0
                code (expr2)       ($3 ← expr2)
                slt $5, $3, $6     ($5 = $3 < $6)
                slt $6, $6, $3     ($6 = $6 < $3)
                add $3, $5, $6     ($3 = $5 V $6)
```

test → expr1 == expr2

```
code(test)    = code(expr1)        ($3 ← expr1)
                add $6, $3, $0
                code (expr2)       ($3 ← expr2)
                slt $5, $3, $6     ($5 = $3 < $6)
                slt $6, $6, $3     ($6 = $6 < $3)
                add $3, $5, $6     ($3 = $5 V $6)
                sub $3, $11, $3    ($3 ← 1 - $3)
```

Do >=, <=, as exercise

If statement:

if(test) {stmt1} else {stmt2}

```
code(stmt)    =
                code(test)         ($3 ← test)
                beq $3, $0, else
                code(stmt1)
                beq $0, $0, endif
        else:
                code(stmt2)
        endif:
```

Issue: need to generate unique label names

- keep a counter X

- use elseX, endifX, trueX for labels
- increment X for each new if statement
- generate labels before you recurse

While statement

while(test) {stmt}

- use a counter Y for labels

```
code(stmt)      =
        loopY:
                code(test)              ($3 ← test)
                beq $3, $0, doneY
                code(stmt)
                beq $0, $0, loopY
        doneY:
```

**Advice:

- generate comments along with MIPS instructions
- helps debuggings

end of assignment 9 material

Pointers

Need to support:

1. NULL
2. Dereference
3. Address-of
4. Pointer comparison
5. Pointer arithmetic
6. Alloc/dealloc
7. Assignment through pointers

NULL

- can use 0
- Use 1 to represent NULL, so that dereferencing will crash

Factor → NULL:

```
code(factor)        =
                    add $3, $11, $0
```

Address-of (& lvalue)

```
code(&ID)           =
                    lis $3
                    .word _______   //look up ID in symbol table – offset
                    add $3, $29, $3
code(&*factor)      =  code(factor)
```

Dereference

Factor1 → * factor2

```
code(factor1)        =
                     code(factor2)        ($3 ← factor2)
                     lw $3, 0($3)
```

Assign through pointer dereference

LHS – address at which to store value

RHS – the value

Comparsion

- Same as int comparison
- Except – no negative pointers
- So pointer compares should be unsigned
- i.e. use sltu instead of slt

So when generating code:

test → expr1 < expr2

- use slt if expr1, expr2 :int
- use sltu if expr1, expr2: int*
- re-run "type-of" function onf expr1 or expr2 to recompute the types

Better:

- add a "type" field to each tree node
- "type-of" records each node's type (if any) in the node itself
- Then the info is there if needed

Pointer Airthmetic:

expr1 → expr1 +- term

- Exact meaning depends on types

expr1 → expr1 + term

if expr2, term: int- as before

if expr2 : int*, term: int – means expr2+4*term

```
code(expr1)          =
                     code(expr2)
                     push($3)
                     code(term)
                     pop($5)
                     mult $3, $4
                     mflo $3
                     add $3, $5, $3
```

if expr: int, term: int* - 4expr2 + term

- Similar

expr1 → expr1 – term

if exp2, term : int – as before
if expr2 : int*, term:int – means expr2 – 4*term

- Similar to +

If expr2,term: int*

```
code(expr1)     =
                code(expr2)
                push($3)
                code(term)
                pop($5)
                sub $3, $5, $3
                div $3, $4
                mflo $3
```

New/Delete

- Runtime environment alloc.merl
  Link to assembled MIPS
- Must be linked last

Stmt → ID = expr
Stmt → *factor = expr

```
code(stmt)      =
                code(factor)
                push($3)        //this is the data
                code(expr)
                pop($5)         //this is the location    ($5 ← factor)
                sw $3, 0($5)
```

July 20, 2015

Recall: new/delete

- Alloc.merl – must be linked last

Add to prologue:

- Import init
- Import new
- Import delete

Function init

- sets up allocators data structures
- call init once in the prologue

- Takes a param in $2
    - If calling with mips.array (i.e int wain(int * …))
    - $2 = length of the array
    - Else $2 = 0

New & delete – functions like print

- New:
    - $1 = # of words needed
    - Returns pointer to memory in $3
    - Returns 0 if allocations fails

```
Code(new int [expr])  =
                      Code(expr)
                      add $1, $3, $0
                      call new
                      bne $3, $0, 1
                      add $3, $11, $0
```

- Delete:

```
Code(delete [] expr)  =
                      Code(expr)
                      beq $3, $11, skipY
                      add $1, $3, $0
                      call delete
                skipY:
```

<u>Compiling Procedures</u>

int f(…) {…}
int g(…) {…}
int wain(…, …) {…}

Figure from Duaa's notes

Main prologue/epilogue

- Save $1, $2 on stack
- Import print, init, new, delete
- Set $4, $11, etc.. if desired
- Set $29
- Call init
- ….
- Reset $30, jr $31

Procedure-specific prologues

- Don't' need imports, set constants

- Set $29
- Save registers
- …
- Restore registers, reset $30, jr $31

Saving and restoring registers
- Proc should save all registers it will modify
- How do you know which registers to save?
    - If not sure, store them all
    - Our code gen scheme never uses any register fast $6, other than $29, $30, $31
    - Don't forget to sabe and restore $29

<u>Two approaches to saving registers</u>

Caller-save vs callee-save

Supposed f calls g
Callee-save:
- g saves all registers it will modify
- f doesn't worry about what g does

Caller-save:
- f saves all registers that hold critical data before calling g
- g doesn't worry about f's registers

Our approach has been: caller-save for $31
Callee-save for everything else

Q: Who should save $29? Caller or callee?
Supposed we do calee-save.
- G saves its registers, including $29
- G sets $29 to the bottom of the new stack frame
- Maybe – set $29 first then save regs
    - Cant change $29 before it has been saved
    - So: save $29, set $29, save all other registers
    - <u>OR</u> just let the called (f) save $29

```
f(){                          f:    …
…                                   push($29)
     g();                           push($31)
…                                   lis $5
}                                   .word g
                                    jalr $5
                                    pop($31)
                                    pop($29)
```

Labels

int init() {…}
int print() {…}
int else() {…}

procedure names match names of existing labels

- Duplicate labels won't assemble

Solution:

For functions f,g,h etc.. use labels Ff, Fg, Fh

- Reserve labels starting with F as denoting functions
- Then make sure all compiler generated functions don't start with F

Parameters

- Could use registers – may not be enough
  - Or push them on the stack

factor → ID (expr1, …, exprn)

```
code(factor)  =
              push($29)
              push($31)
              code(expr1)
              push($3)
              …
              code(exprn)
              push($3)
              lis $5
              .word F_____
              jalr $5
              //must pop arguments
              pop($31)
              pop($29)
```

proc → INT ID (params) {dcls stmts return expr;}

```
code(proc)    =
              sub $29, $30, $4
              push regs
              code(dcls) (local variables)
              code(stmts)
              code(expr)
              pop regs
              add $30, $29, $4
              jr $31
```

Fugure from Duaa's notes

Suppose g is

```
int g(int a, int b, int c){
      int d = 0; int e = 0; int f = 0;
      …
}
```

Symbol table for g:

Name	Offset from $29
a	0
b	-4
c	-8
d	-12
e	-16
f	-20

Params a,b,c are **below** $29
Locals d,e,f are **above** $29
Offsets are wrong!
Params should have positive offsets
Locals should have negative offsets

Need to fix symbol table
Add 4* (# of args) to every offset

Name	Offset from $29
A	12
b	8
c	4
d	0
e	-4
f	-8

Saved registers still between args and locals

Save local registers after pushing the local variables

Figure from Duaa's notes

(all initializers are constants)

July 22, 2015

Recall: push locals firs then save registers
Figure 1 from textbook
OR
Save registers in caller:

Figure 2 from textbook

But:

```
int f(){
    g();
    g();
    g();
    g();
}
```

- Callee-save (in g) – save registers once per function
- Caller-save (in f) – save registers once per call
    - Probably costs space

Factor → ID (expr1,..., exprn)

```
Code(factor)  =
              push($29)
              push($31)
              code(expr1)
              push($3)
              …
              code(exprn)
              push($3)
              lis $5
              .word FID
              jalr $5
              pop all args
              pop($31)
              pop($29)
```

Optimization

Computationally unsolvable – can only approximate the ideal solution

Example: 1 + 2

```
lis $3
.word 1
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
lw $5, 0($30)
add $30, $30, $4
add $3, $5, $3          //9 words
```

This is what your compiler should output if followed instruction in class

```
lis
.word 3
```

- called constant following

Constant propagation
(when you know the value of a variable is a constant and sub that in)
Costant folding - actual evalutation of constant expression

```
int x = 1;
x + x;

lis $3
.word 1
sw $3, -4($30)
sub $30, $30, $4
lw $3, -12($29)          //assuming x is sitting on -12($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -12($29)
lw $5, 0($30)
add $30, $30, $4
add $3, $5, $3           //11 words
```

Can recognize that x = 1 and simply

```
lis $3
.word 1
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
```

And if that's the only place x is used, ti doesn't need a stack entry:

```
lis $3
.word 2
```

Watch out for aliasing.

```
int x = 1;
int *y = NULL;
y = &x;
*y = 2;
x + x;
```

Even is x's value is not known, could recognize that $3 already contain x:

```
lw $3, -12($29)
add $3, $3, $3
```

This is know as **common subexpression elimination**
Example: (a+b) * (a+b)

- use a register to hold a+b, then multiply it by itself rather than compute it twice

Dead Code Elimination

- if you are certain that some branch of a program will never run, do not output code for it

Register Allocation

- idea that it is cheaper to use registers than stack for variables (saves lw and sw)
- for example registers $12-28 unused by our code gen
    - could use these to hold 17 variables and use the stack for the rest
    - Which 17 vars?
        - probably the most used ones

Issue: & address-of

- if you take the address of a variable, it cant be in registers – mst be in RAM

Strength Reduction

- the idea that add is ususally faster then mult
- more of a speed optomization more than space
- so prefer add than mult

Example: multiply by 2

```
lis $2
.word 2
mult $3, $2
mflo $3
```

OR

```
add $3, $3, $3
```

Procedure-Specific Optimizations

Inlining

```
int f(int x){
        return x + x;
}
int wain(int a, int b){
        return f(a);
}
```

It would be simpler to have:

```
int wain(int a, int b){
        return a + a;
}
```

This is inlining.

Idea: Replace the function call with the body, right in the caller

- saves the overhead of a function call
- Is it always a win?
    - If f is called many times, then you get many copies of its body
    - which is bigger – the body of f or the code to call f?
- Also some functions are easier to inline than others (recursive functions)
    - inlining is then not always possible (recursion)
- If all calls to f are inlined, then you don't need code for f anymore (but be careful with that)

July 27, 2015

Trail Recursion

```
int fact (int n, int a){
        if(n == 0) return a;
        else return fact(n-1, n*a);
}
```

- There is no pending work to do when the recursive call to fact returns
- contents of the current stack frame (local variables, etc..) will not be used again
- So reuse the current frame – don't create a new one – set it up to do the recursive call and start again

Can this optomization be done in WLP4?

- No, because the return is only at the end of the procedure
    - Can't have a return in an if statement

But…

- We could expand the language after parsing and mutate the parse tree

Basic transformation:

- If return immediately follows if-then-else, push inside both branches

```
int f(…){
    if(…){
            if(…){
            } else{
            }
    } else{
    }
    return x;
}
```

Equivalent to :

```
int f(…){
    if(…){
        if(…){
            return x;
        } else{
            return x;
        }
    } else{
        return x;
    }
}
```

Whenever return x follows assignment to x, merge them.

```
x = f(…);
return x;
```

Can turn to: return f(…);

- may create some tail recursive calls

Generalization: Tail call optomization

- when a function's last action is any function call (recursive or not), can reuse the stack frame

Overloading

What would happen:

```
int f(int a) {…}
int f(int a, int* b) {…}
```

- Get duplicate labels for f
- How do we fix this?
- **Name mangling** – encode the types of the parameters as part of the label

Example convention:

F + type information + _ + name

Example:

```
int f(){…}              F_f:
int f(int x) {….}       Fi_f:
int f(int a, int b){…}  Fip_f:
```

C++ compilers will use this because C++ has overloading.

- No standard mangling convention – all compilers are different
- That makes it hard to link code from different compilers

- This is by design because compilers differ in other aspects of calling conventions, so you don't want it to link successfully if the convetions don't truly match

C has no overloading, therefore no mangling
- C and C++ hcode call each other routinely
- How is this done? Need to suppress mangling in C++

Calling C from C++
- extern "C" int f(iny x); //Look for an umangled name when linking

Allowing C to call C++
- extern "C" int g(int x) {…}

```
extern "C" {
…
        int f(int x);
…
}
```
Oviously cant overload extern "C" functions

Explicit Memory Management and the Heap

WLP4, C, C++ - user will do explicit memory management
Java, Scheme – implicit memory management – garbage collection

How do new/delete (or malloc/free) work?
- variety of implementations

1) **List of free blocks**
    - Maintain a linked lists of clocks of free RAM

    Initially, the entire heap is free and the list contains one entry
    Suppose heap is 1K
        free -> [1024 | entire heap = 1020]
    The free pointer tells where the memory starts
    Use the first 4 bytes to store how much memory you have
    Suppose 16 bytes allocated.
    - Actually allocate 20 bytes (16 + 1 int (4 bytes))
    - Return a pointer to the second word (|)
      [20 | 16 left over]
    - Store size just before the returned pointer

    Free list:
        free -> (1004 | 1000)
    Suppose 28 bytes requested – allocate 32
        Allocated : [20 | 16] [32 | 28]

free -> [972 | 968]

Suppose first block is freed: add to free list

figure 1 on notebook

Suppose the other block is free:

figure 2 on notebook

Note: repeated allocdealloc creates "holes" in heap

Example: fig 3

Called **fragmentation** – means even if n bytes are free, may not be able to allocate a block of n bytes.

July 28, 2015

To reduce fragmentation:

- don't always have to pick the first block of RAM big enough to hold the request

Example:

[//// | 20 | //// | 15 | //// | 100]

allocate 10 – Where do we put it?

- first fit – put in 20-byte block
- best fit – out in 15-byte block – better fit, hopefully less waste

Searching available RAM takes time – memory allocation is not free

- using the heap is more expensive than using the RAM, so use RAM when you can and heap when you have to

2) **Binary Buddy System**

Assume: size of heap is a power of 2

Example: heap is 4k (4096 bytes) = 1024 words

<- 1024 ->

[ ]

Suppose:

- user requests 20 words
- we need 1 word for book keeping = 21 words
- memory allocated in blocks of size 2^k, so allocate 32 words

[1024]

too big, so split into two heaps – "buddies"

[512 | 512]

still too big, split again

[256 | 256 | 512], again

[128 | 128 | 256 | 512], keep going until you have a block of 32

[32 | 32 | 64 | 128 | 256 | 512] like this
[/// | 32 | 64 | 128 | 256 | 512] like this

Now request 63 words, 64 (since + 1 for bookkeeping)
[///| 32 |///| 128 | 256 | 512]

Request 50 words, so +1 for bookkeeping = 51, so 64 words
[///| 32 |///|64| 64 | 256 | 512] (128 was split)
[///| 32 |///|///| 64 | 256 | 512]

First 64-block is released:
[///| 32 | 64 |///| 64 | 256 | 512]
32-word block released
[32 | 32 | 64 |///| 64 | 256 | 512], buddy (32) is also free, so merge them
[64 | 64 |///| 64 | 256 | 512], buddy (64) is also free, so merge again
[128 |///| 64 | 256 | 512]

64-word block released
[128 | 64 | 64 | 256 | 512], merge x4
[1024]

Deallocation into (blank space),
[////|]
(^deallocation)

alloc.asm – gives each block a code
entire heap = [ ] (1) that's the code
if it gets split into 512-word buddies = [512 | 512] (10, 11)
if those get split into 256-word buddies = [256 | 256 | 256 | 256] (100, 101, 110, 111)
if those get split (1000 and 10001, 1010 and 1011, 1100 and 1101, 1110 and 1111)

Store a block's buddy code in front of the pointer
Buddy's code = flip the last bit
Merging – buddy code of result = drop last bit (>>1)

Implicit Memory Management: Garbage Collection

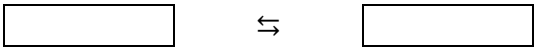
1) **Mark and sweep**
Scan the entire stack, looking for pointers
- for each pointer found, mark the heap block it is pointing at
- then from that heap object, follow any pointers, mark, etc…

Then scan the heap – reclaim any blocks not marked and clear the marks

2) **Reference Counting**

- for each heap block, ount the number of pointers pointing at it (that's what is called its "**reference count**")
- must watch every pointer and update ref counter (decrementing old, incrementing new) each time a poiter is reassigned.
- if a block's reference count hits 0, reclaim it

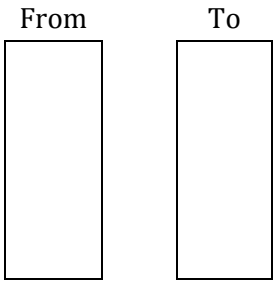
Problem: circular reference



- Both have reference coutn $= 1$, but collectively inaccessible
- They are pointing at each other, but nothing is pointing at them
    - Their reference count is not 0 and will never be 0 but in garabage
    - This creates a memory leak

3) **Copying Garabe Collection**

Heap is split into 2 halves, "from" and "to"



- Only allocate from "from"
- When "from" fills up, all reachable data is copied from "from" to "to", and the role of "from" and "to" is reversed
- Built-in compaction – guaranteed that after the swap, all reachable data is in continuous memory
- No fragmentation
- But can only use half the heap at a time

The End