Andy Yin
Bradley Lushman
CS 241
Lecture - Introduction
Jan 6, 2015

* Information
  - Brad Lushman
  - DC 3110
    bmlushma@uwaterloo.ca

  - tutor: Sean Harrap (cs241@student.cs.uwaterloo.ca)
  - grading
    - Assignments (weekly, 11 total)
    - Midterm
    - Final Exam

* what is sequential program?
  - ordinary program
  - a sequential program is not concurrent or parallel
    - a parallel program is a program utilizes multiple CPUs to do multiple task and combine the results at the end
    - a concurrent program does many things, but only on one CPU

  *"single-threaded", only does one thing at a time

* Goal - Foundations
  - understand how sequential programs "work"
    - "from the ground up"
  - starting point:
    - bare hardware (so to speak)
      - for CS241, a bare hardware means a simulated MIPS machine
      - only interprets 1, and 0

- at the end, we will get programs written in a C-like language to run in MIPS
  - we will work with: assemblers → linkers → loaders → compiler
    - compiler : Assignments 6 – 11

* The Beginning – Binary & Hexadecimal Numbers

- bit : binary digit, ie. 0 or 1
  - high/low voltages, configuration on a magnetic tape
  - this is all the computer understands
    - convenients to group bits together
- byte: group of 8 bits, ie, 11001001, $2^8 = 256$ possible bytes
  - once upon a time, the number of bits required to represent a char
- word: machine-specific grouping of bytes
  - we will assume a 32-bit computer architecture
    - so one word is 32-bits = 4 bytes
- 4 bits (½ byte), sometimes refered as a nibble

- given a byte (or a word) in the computer memory, what does it mean?
  - example: what does 11001001 mean?
    - it could represent many things
      (1) a number – what number is it?

Binary Number System:

$$\begin{array}{cccccccc} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

$$= 2^7 + 2^6 + 2^3 + 2^0 = 201$$

- how can we represent negative numbers?
  - simple way is to reserve the first bit to represent the sign
    - by convention, 0 for +
      1 for –
    - this is called a sign-magnitude representation
  - then,

$$\begin{array}{cccccccc} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ - & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

$$= -(2^6 + 2^3 + 2^0) = -73$$

- but, there are a few problems
  - two representations for 0; 10000000
    00000000
  - arithmetic on sign-magnitude number is tricky
    - example: adding a positive number to negative number
  - if you want to check if something is 0, then you have to check against positive zero, and negative zero

- Better Way: Two's Compliment
  (1) interpret the n-bit number as an unsigned integer
  (2) if the first bit is 0, we are done
  (3) otherwise, subtract 2^n
  - example: for all the 3 bit numbers, $n=3$

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

  - in terms of modular arithmetic:

100	101	110	111	000	001	010	011
-4	-3	-2	-1	0	1	2	3

- so, the n bits represent the numbers of $-2^{n-1} \ldots 2^{n-1}-1$
  - only one zero
  - left bit gives sign
  - addition is clean, just modulo 2^n
  - same addition circuitry works for signed and unsigned
  - so 11001001 = 201 - 256 = -55

* Convenience: Hexadecimal Notation
- uses base 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- more compact than binary, each hex digit is 4 bits (a nibble)
  - example: what is 11001001 in hexadecimal?
    1100 → C
    1001 → A
    > 11001001 = CA
- group 4 bits and convert

- we use 0x notation to represent hexadecimal
  - eg. 0xCA

- so, given a byte 11001001, how can we tell if it is unsigned, sign-magnitude, or two's compliment?
  - we can't, need to remember our intent was when we stored the byte
  - we don't even know that 11001001 is a number, it could be ...,

* The Begining - Characters
  - we need a convention mapping between bit patterns of characters
    - ASCII: American Standard Code for Information Interchange
      - uses 7 bits, to save money for each bit
      - the eighth bit can be used for additional characters
        - there are non-standard, compatibility issues
    - EBCDIC: Extended Binary Coded Decimal Interchange Code
      - dead
  - example: What is 11001001?
    - 11001001 is not 7-bit ASCII
    - 1001001 is decimal 73, and is ASCII for I
  - we will use ASCII

* The Beginning - Instructions (or a part of one)
  - for us, instructions are 32 bit = 4 bytes
    - 1100, 1001 could be a part of an instruction
    - we will discuss this in detail later in the course

* The Beginning - Garbage
  - unused memory

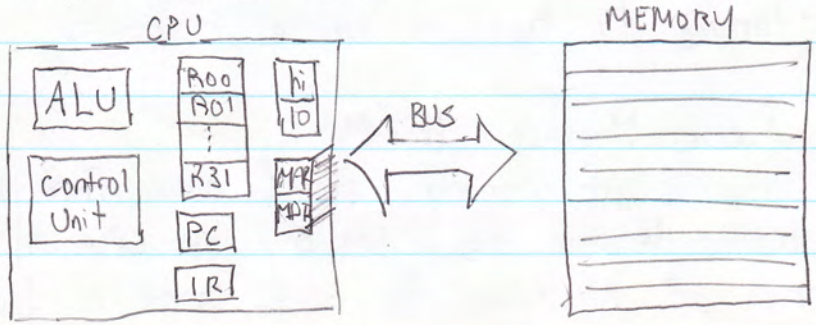
- homework
  - find MIPS reference sheet, print, and bring to all lectures

Andy Yin
Bradley Lushman
CS 241
Lecture –
Jan 8, 2015

*Machine Language
- computer programs operate on data
- computer programs are data
  - historically, programs were seperate from memory (punch cards, and mechanical switches)
    - called the Harvard architecture
- Von Neumann architecture combine the program with data
  - reside in the same memory as the data they operate on
  - makes it possible to write programs that manipulate other programs
    - eg. operating systems, compilers, viruses

- how do we know which parts of memory represent code and which represent data?
  - we don't know

- what does a machine instruction look like? What instructions are there?
  - there are many different machine language
    - processor-specific
  - for us, we will use MIPS (simplified)
    - there are 18 different 32-bit instruction types
    - MIPS Machine CPU



* Central Processing Unit (CPU) "brain of the computer"
  - control unit
    - decodes instructions
    - dispatches to other parts of the computer to carry them out
  - arithmetic logic unit (ALU)
    - does math
  - memory
    - many kinds

CPU cache	Fast
main memory	
disk memory	
network memory	slow

  - on the CPU, there are small amounts of very fast memory
    - registers
  - on a MIPS processor, there are 32 "general purpose" registers
    - "R00 ... R31"
    - each holds 32 bits, ie, one word
    - CPU can only operate directly on data in registers
      - called $0, $1, ..., $31
        - $0 is always 0
        - $31 is special, more later
        - $30 is sort of special
      - example of a register operation
        - add the contents of register s and t, and put the results in register d
    - how many bits does it take to encode a register number?
      - there are 32 registers, so $2^5 = 3$
      - it would take 5 bits
        - so 15 bits to encode 3 register numbers
        - leaves 17 bits to encode operation

* Random Access Memory (RAM)
  - large amounts of memory, stored away from the CPU
  - data travels between the RAM and the CPU on the BUS
  - big array of n bytes for some large n ($n \approx 10^9$)

- each cell has an address 0, ..., n-1
  - each 4 byte block is a word
  - words have addresses: 0, 4, 8, C,
    10, 14, 18, 1C,
    20, 24, 28, 2C, etc
- RAM access is <u>much</u> slower than regular access
  - data in RAM must be transfered to registers before it can be used

- communicating with RAM
  - two commands
    - load (lw)
      - transfers a word from a specified address to a specified register
      - desired address goes into Memory Address Register (MAR)
        - goes out on bus
      - data at that location comes back on the bus and goes into the Memory Data Register (MDR)
    - store (sw)
      - similar to load
- we don't know which words in memory contain data and which contain code
  - then how does the machine execute code?
    - special register called the PC (program counter) holds the address of the next instruction to run
    - by convention, we guarantee that a specific address (ie, 0) contains code, and initialize PC to the specific address (0)
  - the computer runs the fetch-execute cycles
    - the Instruction Register (IR) holds the current instruction

```
PC ← 0
loop
    IR ← MEM[PC]
    PC ← PC + 4
    decode and execute instruction in IR
```

end loop
- only program the machine really runs

- the PC holds the address of the next instruction, while the <u>current</u> instruction is running

- how does a program get executed?
  - program called <u>loader</u> puts your program in memory and sets PC to the address of the first instruction in the program

- what happens when a program ends?
  - need to return control to the loader
  - set PC to the address of the next instruction in the loader
    - which address is that?
      - register $31 will contain the right address back to the loader
      - we need to set the PC to $31
        - use the jr instruction; jr $31

- example
  - add the value in register $5 to the value in $7 and store the result in register $3, and return

MIPS Machine Code

instruction	binary	hex
add $3, $5, $7	0000 0000 1010 0111	0x00A71820
	0001 1000 0010 0000	
jr $31	0000 0011 1110 0000	0x 03E00008
	0000 0000 0000 1000	

- example
  - add 42 to 52, store in $3, return
  - lets put 42 in $5, 52 in $7
    lis $x

Andy Yin
Bradley Lushman
CS 241
Lecture -
Jan 13, 2015

* Example 2
  - add 42 + 52, store result in $3, return (put 42 in $5, 52 in $7, then append example 1's solution)
  - recall
    - lis $d
      - load immediate and skip
      - treat next word as an intermediate value and load it into $d, then skip to the following instruction

  - solution

location	binary	hex	instruction
0000 0000	0000 0000 0000 0000 0010 1000 0001 0100	0x00002814	lis $5
0000 0004	0000 0000 0000 0000 0000 0000 0010 1010	0x0000002a	.word 42
0000 0008	0000 0000 0000 0000 0011 1000 0001 0100	0x00003814	lis $7
0000 000C	0000 0000 0000 0000 0000 0000 0011 0100	0x00000034	.word 52

    + solution for example 1

  - if you want to see the binary representation of the .mips file, use "xxd -bits -col 4 FILENAME"

* Assembly
  - replace tedious binary and hex encoding with easier to use mnemonics
    - less chance of error
    - translation to binary can be automated (assembler)

- one line of assembly equals one machine instruction (word)

- example 2, revisited

```
lis $5                  - load immediate and skip
.word 42                - not an instruction, a directive
lis $7
.word 52
add $3, $5, $7          - always destination first ($3 = $5 + $7)
jr $31                  - set PC = $31
```

  - directives
    - not an instruction, a directive that tells that the value is a word, and it's binary interpretation should be loaded as exact
      - not interpreted by the physical CPU, interpreted by the compiler

- example 3
  - compute the absolute value of $1, store in $1 and return
  - some instructions modify PC (branches and jump (ie, jr))
    - beq, $s, $t, i
      - branch if two registers have equal contents
      - increment the PC by the given number of words
        - can branch backwards
      - there's also bne, which branches if "not equal"
    - slt $d, $s, $t
      - sets $d to 1 if $s is less than $t, sets $d to 0 if $s is not less than $t

address		
0	slt $2, $1, $0	; compare $1 < 0
4	beq $2, $0, 1	; if false, skip
8	sub $1, $0, $1	; $1 = 0 - $1
C	jr $31	; return
10		
14		

- example 4: looping
  - sum the integers 1... 13, store in $3, and return

```
address
   0   lis $2                    ; $2 = 13
   4   .word 13
   8   add $3, $0, $0            ; $3 = 0
   c   add $3, $3, $2            ; $3 += $2   <- -5
  10   lis $1                    ; $1 = 1       -4
  14   .word 1                                  -3
  18   sub $2, $2, $1            ; $2--         -2
  1c   bne $2, $0, -5                          -1
  20   jr $31                                   0
```

*RAM
- two instructions
  - lw $t, i($s)
    - load word from RAM into registers
    - loads MEM[$b + i] into $t
  - sw $t, i($s)
    - store word from register into RAM
    - stores $t into MEM[$s + i]

- example 5
  - we will use mips.array instead of mips.twoints here
    $1 = address of an array
    $2 = number of element in the array
  - place element 5 in $3
  - one possible solution

```
lw $3, 20($1)      ; array[5] -> 5x4 = 20
jr $31
```

  - what if the item number you need isn't known before runtime (ie, a[i] inside a loop)

```
lis $5
.word 5  (or whatever)
```

```
lis $4
.word 4
mult $5, $4          ; $5 *= $4
mflo $5              ; see below
add $5, $5, $1       ; $5 += $1
lw $3, 0($5)         ; $3 = MEM[$5]
jr $31
```

* Multiplication and Division
  - mult $s, $t
    - multiplies $s and $t and stores in hi : lo
      - the product of two 32-bit numbers is 64 bits
      - possibily two big for a register
    - there are two special register, hi and lo
      - the result is stored in the concatenation of hi and lo
        mult $s, $t ≡ hi : lo

  - mflo $d
    - moves the value from lo to $d (quotient for division)
  - mfhi $d
    - moves the value from hi to $d (remainder for division)
    - we will assume that the result of multiplication won't exceed 32-bits
      - if it does, then hi will be non-zero

* Labels
  - recall from example 4, if we wanted to remove or add an instruction in a loop, what would happen?
    - we would have to update all branches
      - this will be very painful for nested loops and if statements

  - assembler allows labelled instructions
    label: instruction
    foo: add $1, $2, $3

- assembler associates the name Foo with the addresses

address	
0	lis $2
4	.word 13
8	add $3, $0, $0
c	top: add $3, $3, $2 *
10	lis $1
14	.word 1
18	sub $2, $2, $1
1c	bne $2, $0, top *
20	jr $31

- So whenever top is used, it will refer to 0xC
  - the assembler also computes the difference between PC and top, so

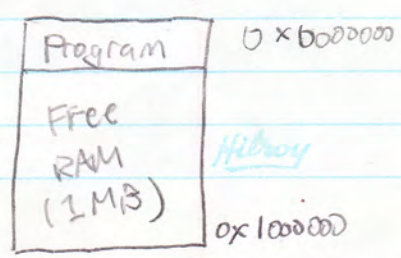
$$\frac{0xC - 20}{4} = \frac{12 - 32}{4} = -5$$

Andy Yin
Bradley Lushman
CS 241
Lecture -
Jan 15, 2015

* Procedures
  - we want to write a procedure f in MIPS assembly
  - however, there are a couple of problems
    (1) call and return
      - how do we transfer control out of f?
      - what if f calls g?
      - how do we pass parameters?
    (2) registers
      - what if f overwrites the registers we were using?
        - eg. some critical data in $3 call procedure f
          f modifies $3, and on return, my data is gone!

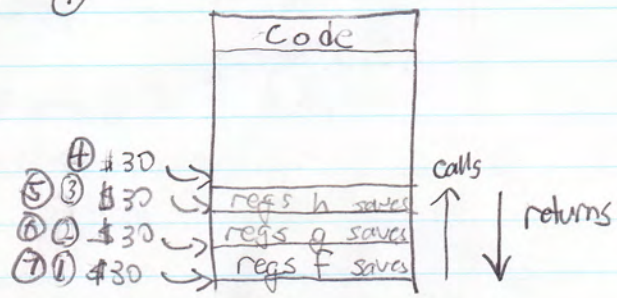
* Procedures - Registers
  - we need a method to store registers
    - we could reserve some of the register for f and some for the main line
    - so they don't interfere with each other
      - what if f calls g? what if f calls f?
        - we will soon run out of registers

    - instead, gaurantee that procedures leave registers unchanged when done
      - we could use RAM to take a snapshot of the registers
        - which RAM? how do we keep procedures from using the same RAM?
          - on the MIPS emulator, we have
          - there are two sensible options:
            - allocate from the top or allocate from the bottom



- we need to keep track of which RAM is used and which one isn't
- the MIPS machine helps us out, specifically the loader
  - $30 is initialized by the loader to just past the last word in memory
    - it points to the first invalid location in memory
    - so it can be used to describe how much RAM is available, but it can be used as a "bookmark" to tell us how much used and unused RAM
    - we will allocate from the bottom for simplicity

- example
  - let f, g, h be functions ①
    - f calls g ②
      - g calls h ③
        - h runs ④
        - h returns ⑤
      - g returns ⑥
    - f returns ⑦



- this is like a stack
  - each procedure stores in RAM the registers it uses and restores the original values on return
  - $30 is the stack pointer, it contains the address of the top of the stack

* Template for Procedures - Registers

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

```
add $30, $30, $3    ; increment $30 (assuming $3
                    ; is still equal to $8)
lw $3, -8($30)      > pop registers
lw $2, -4($30)
```

*Procedures - Calls and Returns

- calling a function

```
main: ....
      lis $5
      .word f        ; address of the line labelled f
      jr $5          ; jump to the line
      (HERE)         ; continue execution here
        .
        .

f:    ...            ; address of this line in memory
```

- returning
  - we need to set PC to the line after the jr (ie, to "HERE" above)
  - how do we know which address that is?

- we have a solution to this:
  - jalr $5
    - jump and link register
    - like jr, but sets $31 to the address of the next instruction (which is the already incremented PC)
  - so, replace jr $5, with jalr $5
    - then return is jr $31
    - but, jalr overwrites $31, then how can we return to the loader
      - and what if f calls g?

- we need to save $31 on the stack first and restore when the call returns

* Template For Procedures - Calls and Returns

```
main: ...
      lis $5                  > function pointer to f at $5
      .word f
      sw $31, -4($30)         > push $31 on the stack
      lis $31
      .word 4                 > decrement stack pointer, note
      sub $30, $30, $31       > that $31 is now free to be used
      jalr $5                 ; call f
      lis $31                 > increment stack pointer
      .word 4
      add $30, $30, $31       > pop $31
      lw $31, -4($30)
        :
      jr $31                  ; return to loader

f:    ; same as main,
      ; push registers onto the stack
      ; decrement $30
      ; body code
      ; increment $30
      ; pop registers off the stack
      ; return to caller
```

- for parameters, we generally use registers (document!)
  - if there are too many registers, can push passing into the stack

* Example

```
; sumOneToN : adds the numbers 1 to N
; registers:
;     $1 - working (we should save this)
;     $2 - input (value of N) (the caller might still use this)
;     $3 - output (the caller knows that register $3 is being
;                  overwritten)
;
```

```
sumOneto N:
    sw $1, -4($30)      ; save $1
    sw $2, -8($30)      ; save $2
    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
```

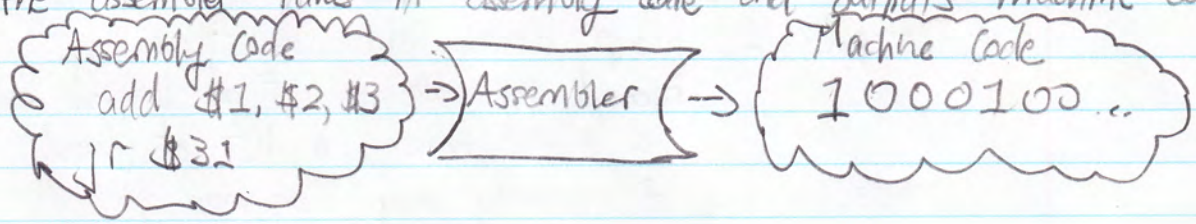
- if you call yourself (recursion), and registers, parameters, and stack is managed properly, then recursion will just work
  - no extra machinery is needed
  - if you get <u>one</u> thing wrong, recursion is almost gaurenteed not to work

* Input/Output (I/O)
- output
  - use sw to store a word in the location 0xffff000c
  - least significant byte will be printed
    - historically, a wire is connected to the screen and anything written to that location would have been printed to the screen
- example

```
print "A\n"
↳ lis $1
  .word 0xFFFF000C
  lis $2
  .word 65          ; A(65)
  sw $2, 0($1)
  lis $2
  .word 10          ; newline(10)
  sw $2, 0($1)
```

Andy Yin
Bradley Lushman
CS 241
Lecture
Jan 20, 2015

* The Assembler
  - the assembler takes in assembly code and outputs machine code



  - any translation process involves two phases
    (1) analysis - understand what is meant by the source string
    (2) synthesis - output the equivalent target string

  - an assembly file is just a string of characters
    - the first step is to generally group characters into meaningful token
      - eg, label, hex number, register number, .word, etc.
    - thankfully, this is done for you in asm.cc, or asm.ss
      - your job is to group tokens into instructions (if possible) as the analysis step
      - output equivalent machine code as the synthesis step
        - if the token are not arranged into sensible instructions, output ERROR to stderr
        - advice, there are many more wrong token configurations than right cases
          - so focus on finding the right tokens, anything else is error

  - later, we will have a more structured way of grouping tokens as parsing

* Biggest Problem with Writing Assemblers

- how do we assemble this?

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

- we can't assemble the beq because we don't know what "abc" is
- the standard solution is to assemble in two passes
  - in pass 1, group tokens into instructions
    record the addresses of all labelled instructions as a "symbol table", a list of (label, address) pairs
  - note, a line of assembly may have more than one label

```
f: g: mult $1, $2
```

```
a:
b:
    jr $31
```

a, b label this instructions (this is one address)

  - you can also label addresses past the end of the program
    - ie,

```
    jr $31
end:
```

will jump to the word after the instruction "jr $31"

- in pass 2, translate each instruction into machine code,
  if an instruction refers to a label, lookup the associated address in the symbol table

- your assembler should output the assembled MIPS to stdout
  output the symbol table to stderr

* Assembly Example

```
main: lis $2
      .word 13
      add $3, $0, $0
  top:
      add $3, $3, $2
      lis $1
      .word 1
      sub $2, $2, $1
```

```
    bne $2, $0, top
    jr $31
beyond:
```

- in pass 1, group tokens into instructions
  build symbol table

label	address
main	0x00
top	0x0c
beyond	0x24

- in pass 2, translate each instruction

```
lis $2      ⟼ 0x00001014
.word 13    ⟼ 0x0000000d
    ⋮
```

bne $2, $0, top (replace top with 0x0c − 0x1c in two's complement)

- note that you need to negate the bits and add 1

⟼ 0x1440fffb

- so, we have −5 = 0xfffb
  - to negate a two's complement, flip the bits and add 1
  - example,

```
 5 = 00000000  00000101
-5 = 11111111  11111010  +1
   = 1111 1111 1111 1011
   = 0x f    f    f    b
```

* Bit-level Operations

- to assemble "bne $2, $0, top" where $\frac{\text{top} - \text{PC}}{4} = -5$

opcode = 000101 = 5	(6 bits)
first register = $2 = 2	(5 bits)
second register = $0 = 0	(5 bits)

offset = -5 (16 bits)

6 bits	5 bits	5 bits	16 bits

32 bits

- to put 000101 in the first 6 bits, need to append 26 zeros, ie left-shift by 26 bits

  5 << 26 C/C++

  (arithmetic-shift 5 -26) Racket

  ⇒ 000101,00 ..... 0 (26)

- move $2, 21 bits to the left

  2 << 21

  ⇒ 00010,00 ..... 0 (21)

- move $0, 16 bits to the left

  0 << 16

- what about -5 = 0xffffffb?
  - this is 32 bits, 8 hex digits; we want only the last 16 bits
  - we can use bitwise operations

a	b	a & b	a | b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

- example

```
  10101110   - bitwise and (&) with 1, original value stays
& 11110000     the same
  10100000   - bitwise and (&) with 0, result is 0, bits
                are turned off

  10101110   - bitwise or (|) with 1, bits are turned on
| 11110000   - bitwise or (|) with 0, original value
  11111110
```

- bitwise & and | can be used to turn bits on and off
- so do a bitwise-"and" with 0xffff

```
-5 & 0xffff                    C/C++
(bitwise-add -5 #xffff)        Racket
```

- then bitwise-"or" the four pieces together

```
(5 << 26) | (2 << 21) | (0 << 16) | (-5 & 0xffff)
```

- this is equal to 339804155 in decimal

```
unsigned int instruction = ....;
cout << instruction << endl;
```

bad!!

this is 10 bits!

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

this prints out the actual bit representation.

- the problem with chars is that they are only one byte

```
unsigned char c;
c = instruction >> 24;
cout << c;
c = instruction >> 16;
cout << c;
c = instruction >> 8;
cout << c;
c = instruction;
cout << c;
```

Andy Yin
Bradley Lushman
CS241
Lecture
Jan 22, 2015

* Loaders
  - they are used to initialize your program
  - can be seen as the operating system (OS)

    OS Code:

```
repeat:
    P ← next program to run
    ① copy P into memory, starting at 0x00
    jalr $0
    beq $0, $repeat
```

① note that mips.twoints, and mips.array are loaders

- problems
  - OS is a program
  - where does it sit in memory?
    - other programs in memory can't all be at address 0x00
  - how can 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 use the same address?
  - let the loader decide where to put the program
    - in this case the loader does the following:
      - takes a program P as input
      - find a memory location α in memory for P
      - copies P to memory starting at α
      - returns α to the OS

OS Code 2.0:

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

- Input: words $w_1, \ldots, w_k$ ← the machine code P
  $n = k$ + space for stack ← how much stack space? pick something.
  n = first address of n contiguous words of unused RAM
  for $i = 1, \ldots, k$
    $MEM[\alpha + (i-1) * 4] = w_i$
  $30 ← $\alpha + 4 * n$
  return

- problems
  - labels may be resolved to the wrong address
  - the loader will have to fix this somehow

```
  lis $3
  .word 0xabc      ok
  lis $A
  .word A          *
  jr $1
B:
  jr $31
A:
  beq $0, $0, B    ok
  .word B          *
```

- instead of loading at address 0x00, consider what would happen if we started at address 0x100000
- the problem occurs whenever we use labels; the labels have hard-coded values
- we should find a way to offset the labels by α
- we can assume that 0xabc isn't an address, because if it was, it should have been a label

- what needs to change when we relocate?
  .word id ← add α to id
  .word constant ← do not relocate
  anything else (including beq, bne) ← do not relocate

OS Code 3.0:

```
repeat
  P ← next program to run
  $3 ← load-and-relocate(P)
  jalr $3
  beq $0, $0, repeat
```

- problem
  - the assembled file is a stream of bits
    - how do we know which ones come from .word (with an id) and which are instructions?
      - we don't, and we need more information from the assembler
  - the output of most assemblers is not pure machine code
    - it is object code

* Object Files
- contains binary code and auxiliary code needed by the loader (and linker)
- our object code is called MERL (MIPS Executable Relocatable Linkable)

- what do we need to put in our object file?
  - the code
  - which lines of the code (addresses) that were ".word id"s
  - other stuff later

- MERL Format

header:
- .word 0x10000002 ← cookie (let loader know file type)
  - prevents us from <u>accidentally</u> running a non-program
- .word length of the MERL file
- .word code length = length of header + MIPS

code:
```
...
MIPS Binary
...
```
- assembled to start at address 0x0C

symbol table

.word	format code	- for format code = 1, it means relocation entry
.word	address	- address in MIPS of a relocatable word
.word	format code	
.word	address	
...		

- example

location	assembly language	machine language	
0x00	beq $0, $0, 2	0x10000002	cookie
0x04	.word endmodule	0x0000003c	.merl length
0x08	.word endcode	0x0000002c	code length + header
0x0C	lis $3		
0x10	.word 0xabc		
0x14	lis $1		
0x18	reloc1: .word A		
0x1C	jr $1		
0x20	B: jr $31		
0x24	A: beq $0, $0, B		
0x28	reloc2: .word B		
0x2C	.word 0x1	0x00000001	format code
0x30	.word reloc1	0x00000013	relocation addr
0x34	.word 0x1	0x00000001	format code
0x38	.word reloc2	0x00000028	relocation addr

- why 0x10000002?
  - it's MIPS for beq $0, $, 2
    - the command to skip the header, straight to the code
  - so MERL files can be executed as ordinary MIPS files, if loaded at address 0

- we would like the assembler to help us generate relocatable MERL object code
  - write a relocatable assembler

Andy Yin
Bradley Lushman
CS 241
Lecture
Jan 27, 2015

* Relocation Tool
  - in terminal, java cs241.merl
  - input: merl file + relocation address
  - output: non-relocatable mips file with merl header and footer removed, ready to load at the given address

  - mips.twoints and mips.array have a second, optional argument
    - the argument corresponds to the address to load the mips file

  - example
    - myobj.merl is to be loaded at 0x10000
      java cs241.merl 0x10000 < myobj.merl > myobj.mips
      java mips.twoints myobj.mips 0x1000
    - relocation is typically done by the loader

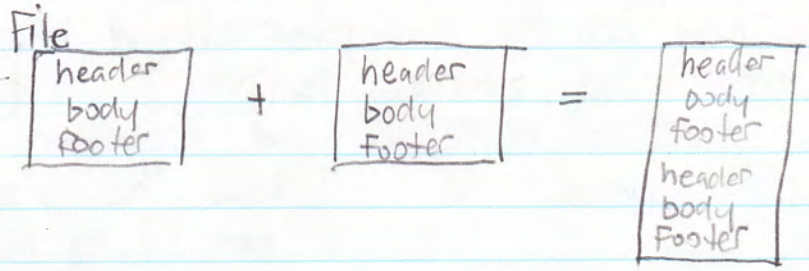
* Loader Relocation Algorithm

```
read()              // skip the cookie
endMod ← read()     // end of MERL file
codeLen ← read() - 12 // subtract header = length of only code

α ← Find_free_ram(codeLen + stackSpace)
for (int i = 0; i < codeLen ; i += 4) {
    Mem[α + i] ← read()
}
i ← codeLen + 12    // start of relocation table

while (i < endMod) {
    Format ← read()
```

```
if (format == 1){
    rel ← read()  // address to be relocated  (1)
    Mem[α + rel - 12] += α-12  // (2)
      // actual location (header not loaded)
        // (1) relative to start of header, not code
        // (2) α - 12 : adjust forward by α
        //      α - 12 : backwards by header length,
                         since header is not loaded
} else {
    ERROR
    it = 8
}
```

*Linkers
- convenient to split large MIPS programs into smaller units
  - code reuse
  - team development

- how can the assembler resolve a reference to a label F if it's in a different file?
  - solution #1
    - concatentate them together and then assemble
      cat a.asm b.asm c.asm | java cs241.binasm > result.mips
  - this will work, but why should we have to assemble these files more than once?
    - ie, every time we cat them into a larger project
  - can't we assemble first, then cat?
    - issues
      - the output needs to be relocatable
        - at most one of the pieces can be loaded at address 0x00?
      - so the assembler needs to output MERL, not just MIPS
        - we can't cat MERL files together to get another MERL



- solution #2
  - we need a tool that understands MERL files and puts them together intelligently
    - this is a linker

- what should the assembler do with references to labels that aren't there?
  - we need to change the assembler
  - when the assembler encounters a .word id, where label id: is not found, it fills in 0 temporarily and it must indicate that the program requires the value of id before it can run
  - example
    - one.asm
      lis $3
      .word a
    - two.asm
      a: sw $4, -4($30)

  - one.asm cannot be executed without being linked to code that contains label a: (eg, two.asm)

- how does the assembler notify us?
  - make an entry in the MERL file

- however, don't we lose a valuable error check?
  eg, lis $3
  .word abc
  ⋮
  abd: ⋮

  - what if we meant abd when we said abc?
  - assembler will ask for abc to be linked in, instead of signalling an error

- how can the assembler know what's an error error and what's intentional?
  - we can make a new assembler directive

```
.import id      ; tells the assembler to ask
                ; for id to be linked
                ; it does not assemble to a
                ; word of MIPS
```

  - so when the assembler encounters a .word abc
    - if label abc: does not occur in the current file and no .import abc directive, then error
  - notifying the system
    - make an entry in MERL table

- Format (code) of ESR entry entry
  - 0x11 means External Symbol Reference (ESR)
  - what info is needed?
    - where (ie, address) is the symbol being use?
    - where is the blank we need to fill in?
    - what is the name of symbol?

  - format

```
(1)   .word 0x11     ; ESR entry
(2)   .word ...      ; location where the symbol is used
(3)   .word ...      ; length of the name in chars (n)
(4)   .word ...      }
(5)   .word ...      } ASCII characters in the symbol's
 ⋮      ⋮            } name (each character in it's own
(3+n) .word ...      } word)
```

- another problem
  - say we have these .asm files

```
- a.asm                  - b.asm
  lis $3                   abc: sw $3, -4($30)
  .word abc                     ⋮
     ⋮                     jr $31
```

external reference

- a.asm

```
⋮
abc: add $1, $2, $3
⋮
beq $2, $0, abc
```

- how can the linker know which abc to link to?
  - can't assume label won't be duplicates

Andy Yin
Bradley Lushman
CS 241
Lecture
Jan 28, 2015

* Linkers
- how can the linker know which labels should be linked?

.import abc	abc:	⋮
lis $3	sw $4, -4($30)	abc:
.word abc	⋮	add $1, $2, $3
	jr $31	⋮
		beq $2, $, abc

  - we can't assume that labels won't be duplicated
- how can we make one abc available and the other not available?
  - solution
    - use another assembler directive of MERL entry
      .export abc
      - makes abc available for linking
      - does <u>NOT</u> assemble to a word of MIPS
      - tells the assembler to make an entry in the MERL symbol table

  - the MERL entry is a External Symbol Definition (ESR)
    - Format
      (1) .word 0x05 - format code
      (2) .word ... - address of the symbol
      (3) .word ... - length of name
      (4) .word ...
      ⋮ ⋮
      (n) .word ...
      } name in ASCII

- in summary, the MERL file contains

- code
- all addresses that need relocation
- addresses and names of any ESR that the linker needs
  - if it's not exported, then an ESR entry will not be made (ie, static)

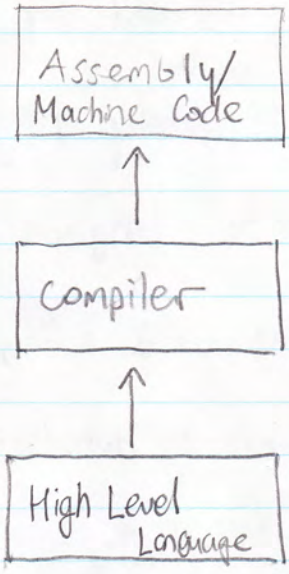
*Linker Algorithm
- input: two MERL files m1 and m2
- output: single MERL file with m2 linked after m1

```
α ← m1.codeLen - 12
relocate m2.code by α
add α to every address in m2.symbolTable

if m1.exports.label intersects m2.exports.label
  ≠ ∅, ERROR (duplicate exports are bad)

foreach <address1, label> in m1.imports
  if there exists <address2, label> in m2.export
    m1.code [address1] ← address2
    remove <address1, label> from m1.imports
    add address1 to m1.relocates

foreach <address2, label> in m2.imports
  if there exists <address1, label> in m1.exports
    m2.code [address2] ← address1
    remove <address1, label> from m2.imports
    add address2 to m2.relocates

imports = m1.imports union m2.imports
exports = m1.exports union m2.exports
relocates = m1.relocates union m2.relocates
```

output MERL cookie
output total code length
+ total (imports, exports, relocates) + 12
output total code length + 12
output M1. code
output M2. code
output imports, exports, relocates

* Formal Languages

- we are now working with formal languages



- simple structures
- easy to translate into machine code
- straight-forward, unambiguous machine language translation

- much more complex
- harder to recognize, not an one-to-one mapping to machine language
- describes the behavior of the program, rather than the program itself

- how can we handle the complexity?
  - we want a formal theory of string recognition
    - general principles that work in the context of any programming language

- definitions

alphabet
- finite set of symbols (eg, $\{a, b, c\}$)
- typically denoted as Σ, as in
  $\Sigma = \{a, b, c\}$

string or word
- finite sequence of symbols (from Σ)
- eg, a, aba, cbca, abc, etc

length of a word $|w|$ = number of characters in w
- eg, $|abba| = 3$

empty string - empty sequence of symbol
ε denotes the empty string
ε is not a symbol ; $|\varepsilon| = 0$

language - set of strings (words)
- eg, $\{a^{2n}b, n \geq 0\}$
- all words with an even number of a's, followed by b

- note that ε is the empty word
$\{\}$ or $\emptyset$ is the empty language (contains no words)
$\{\varepsilon\}$ is the singleton langue that contains only ε
- symbols can be anything, eg $\{@, !\}$

- how can we recognize automatically whether or not a string belongs to a language?
- it depends on the language complexity
- eg, $\{a^{2n}, n \geq 0\}$ - number of even characters
{all valid MIPS programs} - not too hard
{all valid C programs} - much harder
for some languages this is impossible

- characterize languages by complexity of recognition
- Finite languages (languages limited by the number of words)
- regular languages
- context-free languages
- context-sensitive languages
- recursive languages
- etc (impossible)

easy
↓
hard

- we want to start at as easy a level as possible and move down

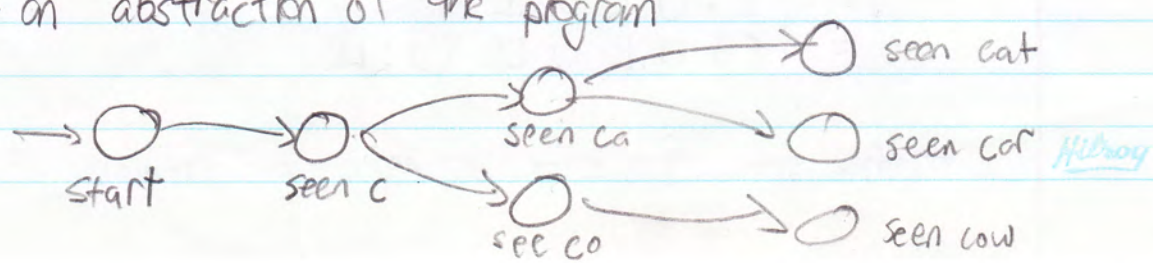
Andy Yin
Bradley Lushman
CS 241
Lecture
Feb 3, 2015

*Finite Languages
- have finitely many words and can recognize a word by comparing with each word in the (finite) set
  - how can we do this more efficiently?

- example
  - consider $L = \{cat, car, cow\}$
    - write code to answer if $w \in L$, such that w is scanned exactly once without storing previously seen chars

```
-scan the input left-to-right
  if first char is c, move on, else error
  if next char is a
     if next char is t
        if input buffer is empty, accept, else error
     else if next char is r
        if input buffer is empty, accept, else error
     else error
  else if next char is o
     if next char is w
        if input is empty, accept, else error
     else error
  else error
```

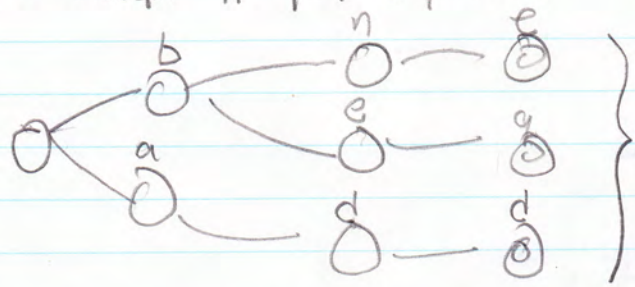
- an abstraction of the program



- based on input seen

  ⓪ = accept if you stop here

- eg,



MIPS instructions

- since programming languages don't usually accept finitely many programs, can't do too much with finite languages

* Regular Language
- built from finite language
  - 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 two languges

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

  - eg, $L_1 = \{ \text{dog}, \text{cat} \}$

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

  $L_1 L_2 = \{ \text{dogfish}, \text{dog}, \text{catfish}, \text{cat} \}$

- repetition

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

  $= \{ \varepsilon \} \cup L$

  $= \{ \varepsilon \} \cup L \cup LL$

  $= \{ \varepsilon \} \cup L \cup LL \cup LLL$

  ...

- the set of zero or more occurences of a word in L

- example
  - let $L = \{a, b\}$

$$L^* = \{\varepsilon\}$$
$$= \{\varepsilon\} \cup \{a, b\}$$
$$= \{\varepsilon\} \cup \{a, b\} \cup \{aa, ab, ba, bb\}$$
$$= \{\varepsilon\} \cup \{a, b\} \cup \{aa, ab, ba, bb\} \cup \ldots$$

- example
  - show that $\{a^{2n}b,\ n \geq 0\}$ is regular

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

*Regular Expressions
- used short hand for regular language

expression	language	
$\emptyset$	$\{\}$	- empty language
ε	$\{\varepsilon\}$	- language consisting of empty
aaa	$\{aaa\}$	- singleton language
$E_1 \mid E_2$	$L_1 \cup L_2$	- alternation (union)
$E_1 \cdot E_2$ (or E_1E_2)	$L_1 \cdot L_2$	- concatenation
E^*	L^*	- repetition

- eg, $\{a^{2n}b,\ n \geq 0\} = (aa)^* b$

- is the C language regular?
  - this is a loaded question (think type checking, order, etc)
  - consider this
    - a C program is a sequence of tokens, each of which comes from a regular languages
      - identifiers

$$[a\text{-}zA\text{-}Z]([a\text{-}zA\text{-}Z0\text{-}9])^*$$

      - integers

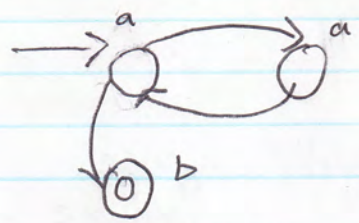
$$([0\text{-}9])^*$$

etc.

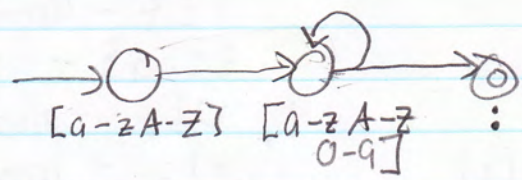
$C \subseteq \{\text{valid } C \text{ tokens}\}^*$

- so maybe

- how can we recognize an arbitrarily regular expression automatically?
  - example
    $\{a^{2n} b, n \geq 0\}$
    - can we harness what we learned about reconizing finite languages?
      - we can represent this with a diagram

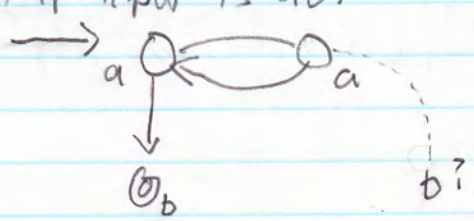


- MIPS labels



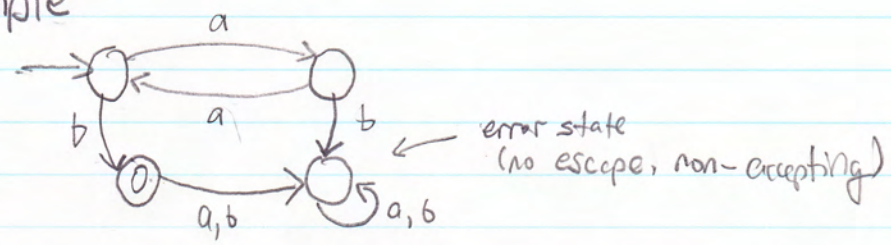
* Deterministic Finite Automatas (DFA's)
- the machines above are called DFA's
  - always start at the start state
  - for each char in the input, follow the corresponding arc to the next state
  - if on an accepting state when input is exhausted, accept, otherwise reject

- what if there is no transition?
  - what if input is ab?

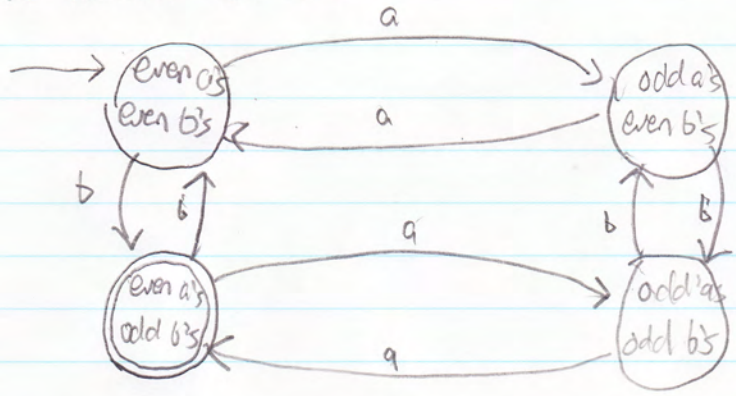


- fall off the machine means rejection
- more formally, an "implicit" error state, all unlabelled transitions go there

- example



- example
  - strings over $\{a, b\}$ with even number of a's and an odd number of b's



Andy Yin
Bradley Lushman
CS 241
Lecture
Feb 5, 2015

* Formal Definition of a DFA
  - a DFA is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$
  where Σ is a finite, non-empty set (alphabet)
  Q is a finite, non-empty set (states)
  $q_0 \in Q$ (start state)
  $A \subseteq Q$ (accepting state(s))
  $\delta : Q \times E \to Q$ (transition function: state + input char → state)
  ↑cross

  δ consumes a single state
  - extend δ to a function that consumes an entire word

  - definition
  $\delta^* : Q \times \Sigma^* \to Q$ (state word → state)
  $\delta^*(q, \varepsilon) = q$
  $\delta^*(q, cw) = \delta^*(\delta(q, c), w)$

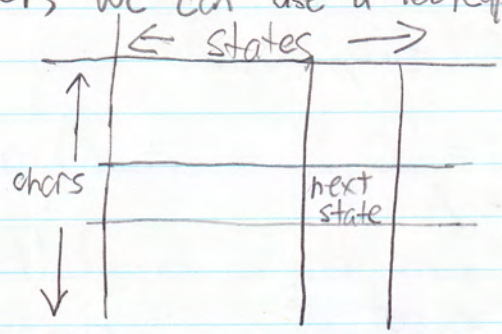
  - we say a DFA $M = (Q, \Sigma, q_0, A, \delta)$ accepts a word w if $\delta^*(q_0, w) \in A$
  - if M is a DFA, we denote by $L(M)$ (the language of M), the set of all strings accepted by M:
  $L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$

* Kleene's Theorem
  L is regular $\iff L = L(M)$ for some DFA M
  - the regular languages are exactly the languages accepted by DFA's

* Implementing a DFA

```
int state = q0
char ch
while not EOF do
   read ch
   Case state of
      q0: case ch of
         a: states ....
         b: states ....
            :
      q1 : case ch of
         a: state = ...
         b: state = ...
            :
      :
      qn : case ch of
         a: state =
         b: state =
end while
return state ∈ A
```

- or, we can use a lookup table

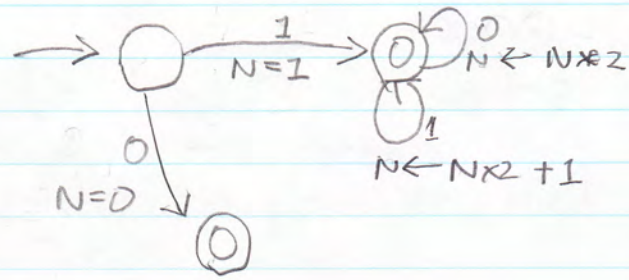


- see assembler starter code, it uses this pattern

* DFA's with actions

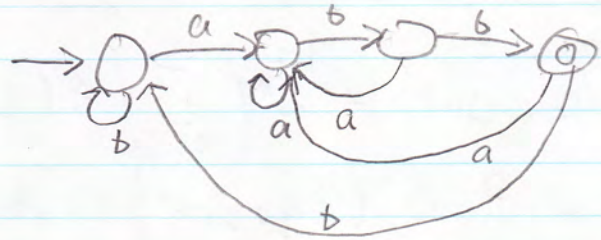
- can attach computations to the arcs of a DFA
- eg, L = { binary numbers with no leading 0's }

$$1(0|1)^*|0$$

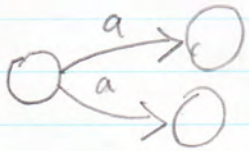


- possible actions could be "emit a token"

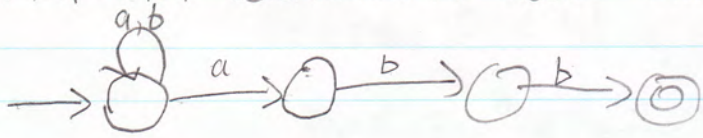
- what do we gain by making DFA's more complex?
  - eg,
    $L = \{w \in \{a, b\}^* \mid w \text{ ends with } abb\}$
    $(a|b)^*abb$
  - DFA for L



- what if we allowed more than one arc from the same state with the same label?



- what would this mean?
  - machine chooses one direction or the other (ie, the machine is non-deterministic)
- accept if some set of choices leads to an accepting state
  - with non-determinism, above machine becomes



- for now, assume the machine knows
- the machine "guesses" to stay in the first state until it reaches the final abb, then transistions to accepting

- non-deterministic finite automatas (NFA's) are often simpler than DFA's

*Formal Definition of NFA

- an NFA is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$

where Σ is a finite, non-empty set (alphabet)

Q is a finite, non-empty set (states)

$q_0 \in Q$ (start state)

$A \subseteq Q$ (accepting)

$\delta : Q \times \Sigma \rightarrow$ subsets of Q (2^Q) (non-deterministic)

- we want to accept if some path through the NFA leads to acceptance for the given word (reject if none do)

δ^* for NFA's : set of states $\times \Sigma^* \rightarrow$ set of states

$$\delta^* : Q^Q \times \Sigma^* \rightarrow 2^Q$$

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

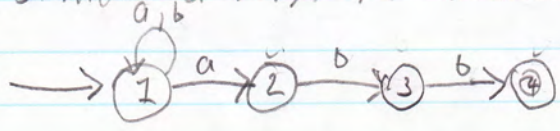
$$\delta^*(q_s, cw) = \delta^*\left(\bigcup_{q \in q_s} (\delta(q, c)), w\right)$$

- then accept w if $\delta^*(\{q_0\}, w) \cap A \neq \emptyset$

- the NFA simulation procedure:

```
states ← {q0}
while not EOF do
    read ch
    states ← ∪ δ(q, ch)
             q∈states
end while
return states ∩ A ≠ ∅
```

- eg, simulate a NFA for baabb



already read input	unread input	states	
ε	baabb	$\{1\}$	1
b	aabb	$\{1\}$	
ba	abb	$\{1, 2\}$	2
baa	bb	$\{1\ 2\}$	

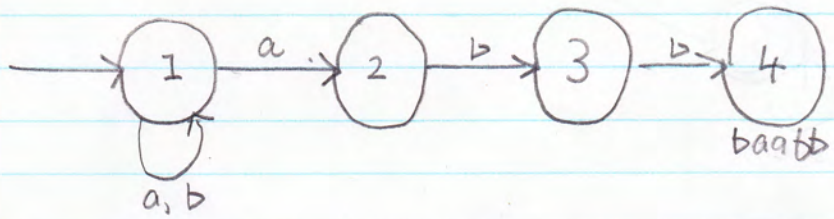
baab	b	$\{1, 3\}$	3
baabb	ε	$\{1, 4\}$	4

$\{1, 4\} \cap \{4\} \neq \emptyset$, and accept

if you give each set of states a name and call these the states, every NFA becomes a DFA!

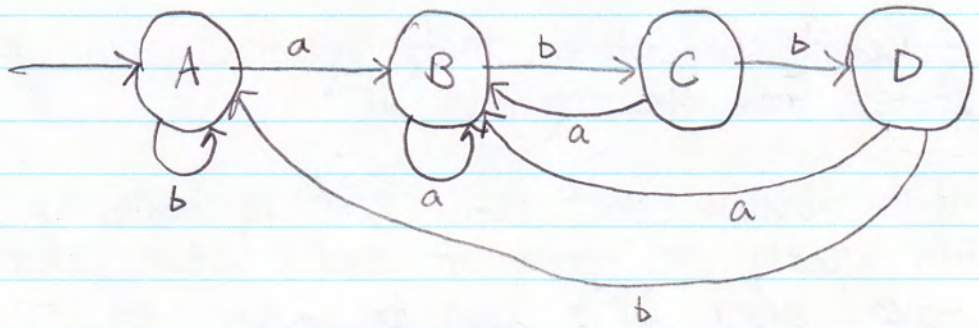
Andy Yin
Bradley Lushman
CS 241
Lecture
Feb 10, 2015

*Non-deterministic Finite State Machines
- recall an NFA



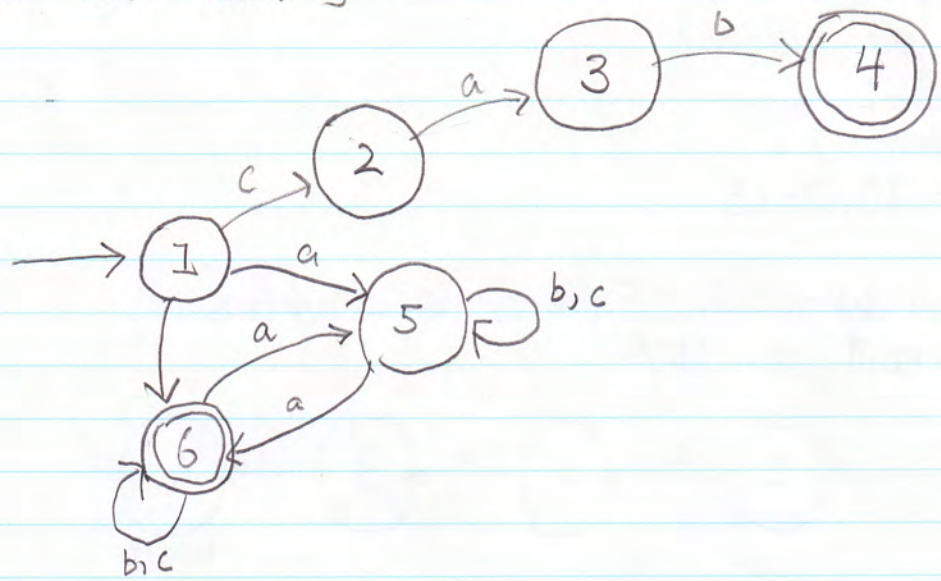
Read	Unread	State	
ε	baabb	$\{1\}$	A
b	aabb	$\{1\}$	
ba	abb	$\{1, 2\}$	B
baa	bb	$\{1, 2\}$	
baab	b	$\{1, 3\}$	C
baabb	ε	$\{1, 4\}$	D

$\{1, 4\} \cap \{4\} \neq \emptyset$, accept
- if you give each set of states a name and call these the states, every NFA becomes a DFA

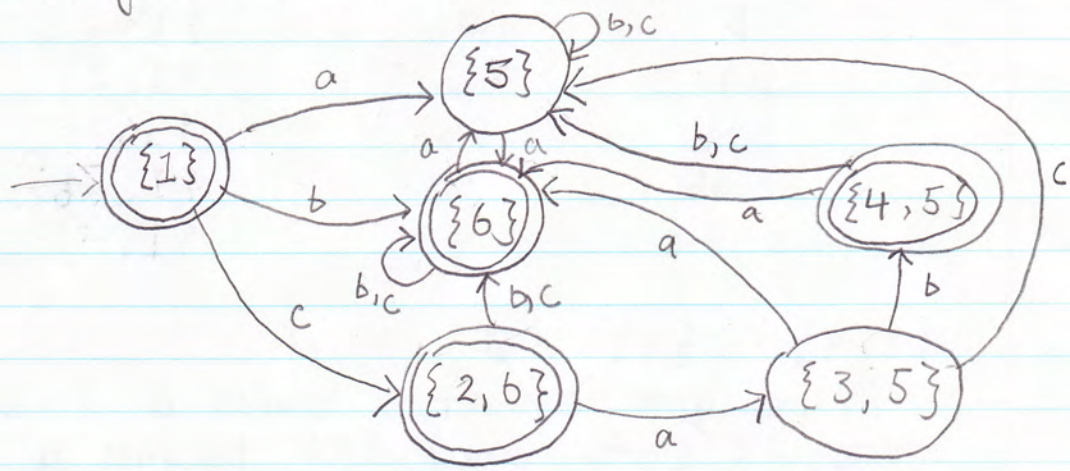


- example
$L = \{cab\} \cup \{w \in \{a, b, c\}^* \mid w$ contains an even number of a's$\}$

- the NFA becomes



- we can build the DFA using subset construction, where can we go from {1} on a, b, c?

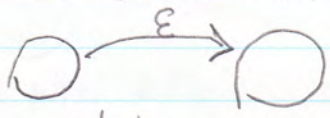


- accept any state that includes an accepting state from the original NFA

- it is obvious that every DFA is implicitly an NFA, with exactly one option at each (state, char)
- also, every NFA can be converted to a DFA for the same language
- so NFA's and DFA's accept the same class of languages (regular language).

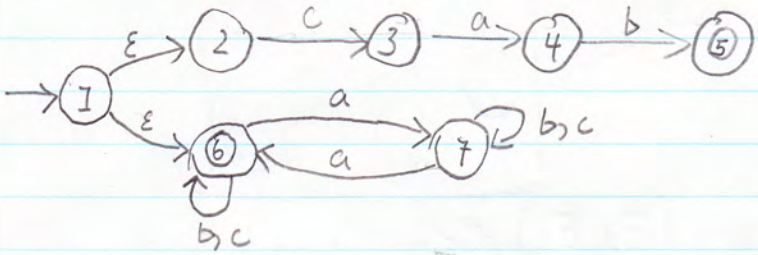
* Epsilon NFAs (ε-NFA)

- what if we can change state without reading a char?
  - ε-transitions



- free pass to a new state without reading a char
  - it is optional, and you do not have to take it
- makes it easier to "glue" smaller machines together

- example
  - let $L = \{cab\} \cup \{\text{even number of } a\text{'s}\}$



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

- so every ε-NFA has an equivalent DFA, and the conversion can be automated

- if we can find an ε-NFA for every regular expression, we have one direction of Kleene's theorem
  - ie, Regex $\rightarrow$ ε-NFA $\rightarrow$ DFA

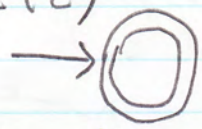
* Regular Expression Types

(1) The Empty Language ($\emptyset$)

ε-NFA:

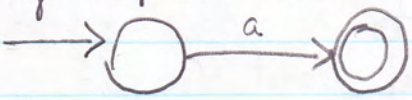
(2) Empty String (ε)

ε-NFA:



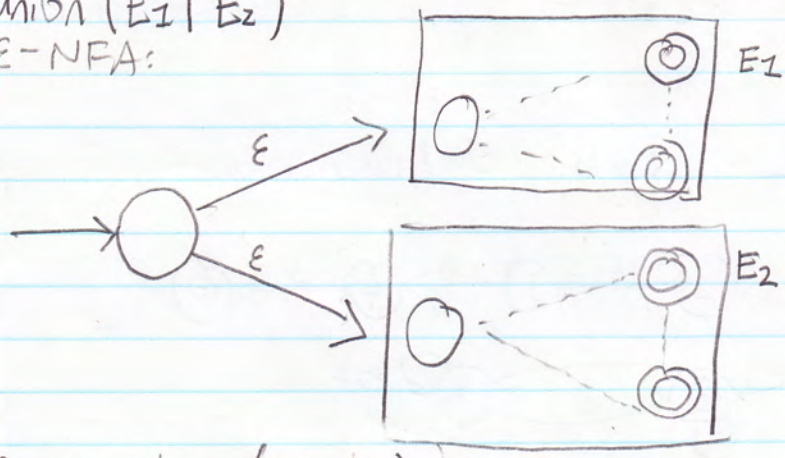
(3) String containing only one word (a)

ε-NFA:



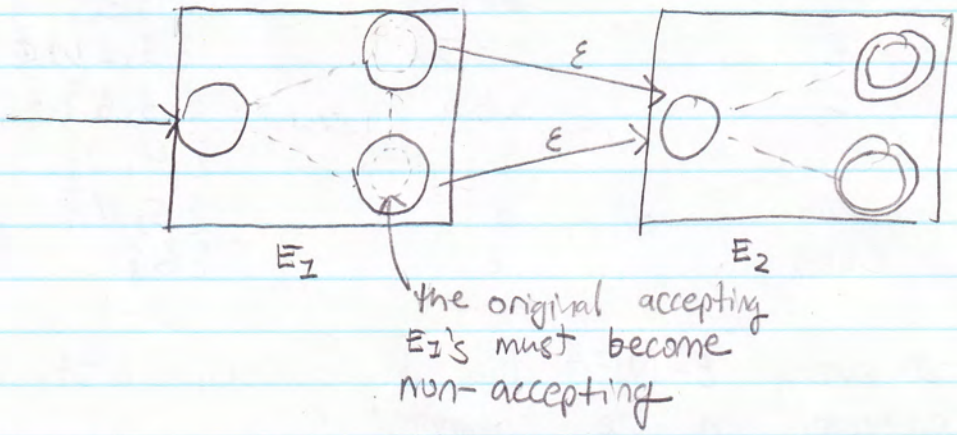
(4) Union ($E_1 | E_2$)

ε-NFA:

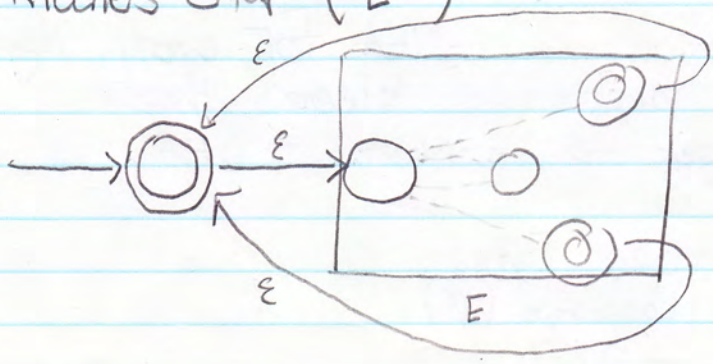


(5) Concatenation ($E_1 E_2$)

ε-NFA:



(6) Kleene's Star (E^*)



- so every regular expression has an equivalent
  - ε-NFA
  - NFA
  - DFA

  } and the conversion can be automated!
- therefore, we can write a tool to convert regular expressions to DFAs

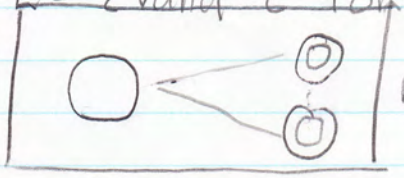
* Scanning (Lexical Analysis)

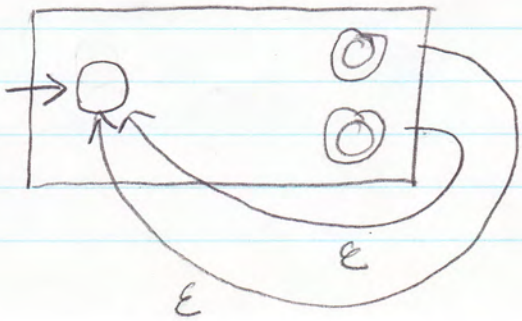
- is C regular?
  - C keywords
  - C ids
  - C literals
  - C operators
  - C comments
  - C punctuation

  } - all of these are regular languages
  - so sequences of these are also regular

- so we can use Finite Automata to do tokenization (scanning)
  - ordinary DFA's answer yes/no to $w \in L$

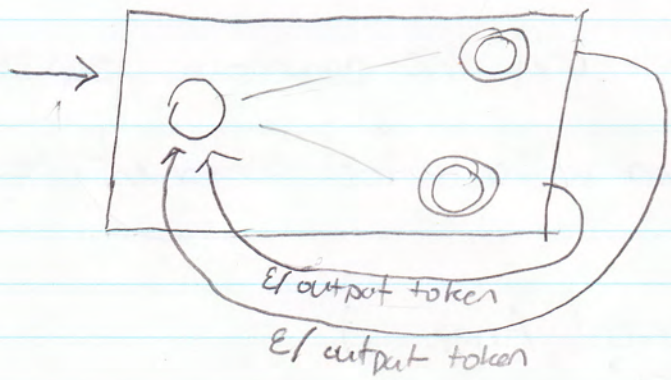
- we need input string w
  - break w into $w_1, w_2, \ldots, w_n$ such that each $w_i \in L$, else error
  - output each w_i

- consider $L = \{\text{valid C tokens}\}$ is regular,
  let $M_L =$  be the DFA that recognizes L,

then 

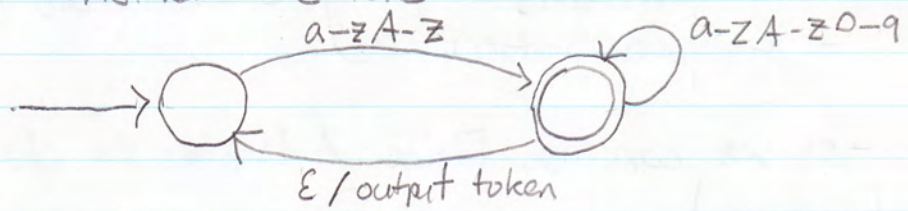
recognizes LL^* (non-empty sequences of tokens)

- add an action to each ε-move



- the machine is now non-deterministic
- since ε-moves are always optional

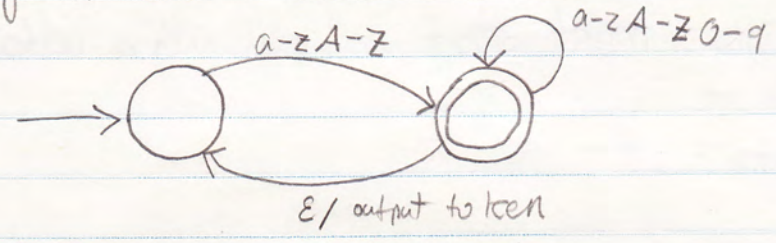
- therefore, we ask the question, does this scheme gaurentee a unique decomposition?
  - ie, $w = w_1, \ldots, w_n$
  - no, consider C id's



  - for input abab, it could be interpreted as 1, 2, 3, or 4 token
    - what do we do about this?

Andy Yin
Bradley Lushman
CS 241
Lecture
Feb 12, 2015

* Parsing
- recall
  - given the ε-NFA



  - for input abab, it could be 1, 2, 3, or 4 tokens
  - what do we do about this?
    - decide to only take the ε-move if there are no other choices
      - so, we look for the longest possible next token

- consider $L = \{aa, aaa\}$, $w = aaaa$
  - take the longest token first, aaa, "a" is left over
    - but $w = aaaa$ could have been matched, however, we will still use it as most langauges do not intend to be ambiguous and difficult
  - the concrete realization is the maximal munch algorithm
    - run DFA (without ε-moves) until no non-error move is possible
    - if in accepting state, output token found, otherwise back up to most recent accepting state (with a variable)
      - input to that point is next token, resume scanning from there

- output token, and ε-move back to q_0

* Simplified Maximal Munch
  - same as above, but if we're not not in an accepting state, and no transistions is possible, error
    - ignore backtracking

- example
  - consider tokens:
    - identifiers: must start and end with a letter, but may contain -
    - operator: --

  - word: ab--
    can't go further, but ab-- is not a valid id, so not in accepting state, so simple maximal munch errors

  - in maximal munch, back up to previous accepting state (ab), and scan from there
    - produces tokens ab, --

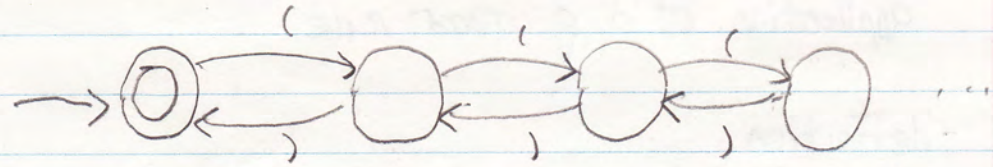
- in practice, simplified MM is usually good enough
  - languages typically designed to faciliate scanning by SMM

- example: C++
  vector <vector< int>> v;
  C++'s longest match scanner scans this as one token '>>', rather than as two '>'s

  - C++ solution is to adapt the language to the scanner

- we must separate '>'s by space to create two token
  - ie, vector<vector<int> > u;

- what (if any) features of C (or Scheme) programs cannot be verified with a DFA?
  - consider $\Sigma = \{ (,) \}$
    $L = \{ w \in \Sigma^* \mid w$ is a string of balanced parenthesis $\}$
  - ie, consider $\varepsilon \in L$, $() \in L$, $()() \in L$, $(())\in L$, $(())() \in L$, $)(\notin L$, $())\notin L$, etc.
  - can we build a DFA for L?



- each new state recognizes one more level of nesting, but no finite number of states recognizes all levels of nesting
  - and DFA's must have finitely many state

* Context Free Languages
- languages that can be described by context-free grammar (set of rewrite rules)

- example
  - consider balanced parenthesis,
    - a word in the language is either:
      - empty
      - a word in the language, surrounded by '()'
      - the concatenation of two words in the language
  - so,

$S \rightarrow \varepsilon$

$S \rightarrow (S)$

$S \rightarrow SS$

- in shorthand, $S \rightarrow \varepsilon \mid (S) \mid SS$ (OR)
  - show this system generates $(())()$

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

- notation

  "$\Rightarrow$" means "derives

  "$\alpha \Rightarrow \beta$" means β can be obtained from α by one application of a grammar rule

- definition
  - a context-free-grammar consists of
    - an alphabet Σ of terminal symbol
    - a finite, non-empty set N of non-terminal symbol, $N \cap \Sigma = \emptyset$ (we use V ("vocabulary") to denote $N \cup E$)
    - a finite set P of productions
      - productions have the form $A \rightarrow B$, where $A \in N$, $B \in V^*$
    - an element $S \in N$ (start symbol)

- conventions
  - $a, b, c, \ldots$ elements of Σ (characters)
  - $w, x, y, \ldots$ elements of Σ^* (strings)
  - $A, B, \ldots, S, \ldots$ elements of N (non-terminal)
  - S start symbol
  - $\alpha, \beta, \gamma, \ldots$ elements of V^* $(= (N \cup \Sigma)^*)$

- we write $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if there is a production $A \rightarrow \gamma$ in P
  - RHS derivable from the LHS in one step

$\alpha A \beta \Rightarrow^* \alpha \gamma \beta$ means
$\alpha A \beta \Rightarrow \delta_1 \Rightarrow \delta_2 \Rightarrow \ldots \Rightarrow \delta_k = \alpha \gamma \beta$
(for $k \geq 0$, ie 0 or more derivations steps)

- definition
  $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$

  language specified by grammar G

  strings of terimals, derivable from S

- definition
  - a language L is context-free if $L = L(G)$ for some context-free grammar G

- example
  - palindromes over $\{a, b, c\}$
    $S \rightarrow aSa \mid bSb \mid cSc \mid M$
    $M \rightarrow a \mid b \mid c \mid \varepsilon$
  - eg,
    $S \Rightarrow aSa \Rightarrow abSba \Rightarrow abMba \Rightarrow abcba$

- expressions
  - let $\Sigma = \{a, b, c, +, -, *, /\}$
    $L = \{$arithmetic expressions using symbols from $\Sigma\}$
    $S \rightarrow a \mid b \mid c \mid S\ Op\ S$
    $Op \rightarrow + \mid - \mid * \mid /$

$\Sigma' = \Sigma \cup \{(,)\}$

$L' = \{$expression with parathesiss$\}$

$S \rightarrow a \mid b \mid S\ Op\ S \mid (S)$

$Op \rightarrow + \mid - \mid * \mid /$

- show $S \Rightarrow^* a + b$

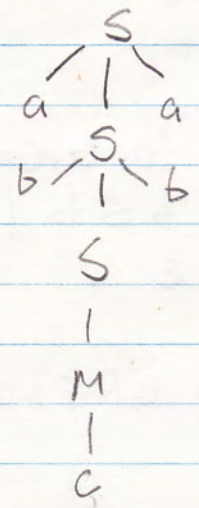
$S \Rightarrow S\ Op\ S \Rightarrow a\ Op\ S \Rightarrow a + S \Rightarrow a + b$

↖ choice of which symbol to expand next

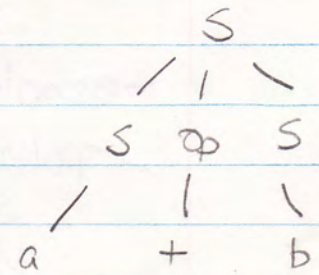
- leftmost derivation
  - always expand leftmost symbol first
- rightmost derivation
  - rightmost first

* Derivations

- derivations can be expressed naturally and succintly as a tree structure
- ie, abcba:



a+b:



- called parse trees

- for every leftmost (or rightmost) derivation, there is an unique parse tree (or vice versa

Andy Yin
Bradley Lushman
CS 241
Lecture
Feb 24, 2015

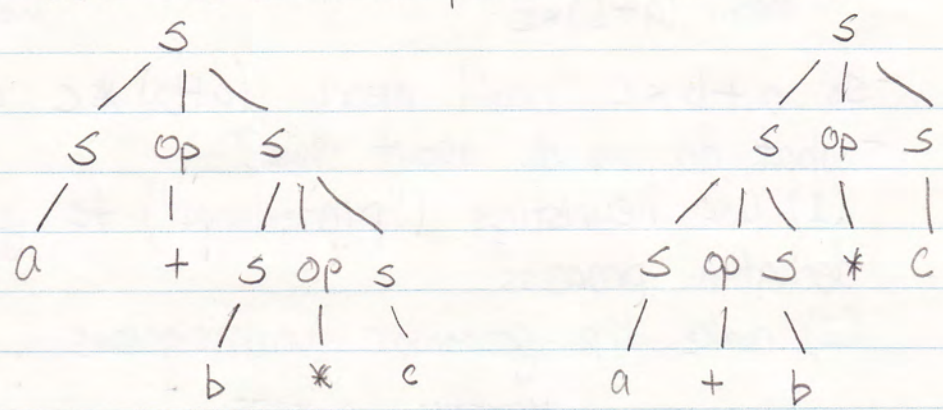
* Grammar

- recall $S \to a \mid b \mid c \mid S \text{ op } S$
  $\text{Op} \to + \mid - \mid * \mid /$
- example
  - leftmost derivation for $a+b*c$

$$S \Rightarrow S \text{ Op } S \Rightarrow a \text{ Op } S \Rightarrow a+S$$
$$\Rightarrow a+S \text{ Op } S \Rightarrow a+b \text{ Op } S \Rightarrow a+b*S$$
$$\Rightarrow a+b*c$$

$$S \Rightarrow S \text{ Op } S \Rightarrow S \text{ Op } S \text{ Op } S \Rightarrow a \text{ Op } S \text{ Op } S$$
$$\Rightarrow a+S \text{ Op } S \Rightarrow a+b \text{ Op } S \Rightarrow a+b*S$$
$$\Rightarrow a+b*c$$

- we have two different parse trees

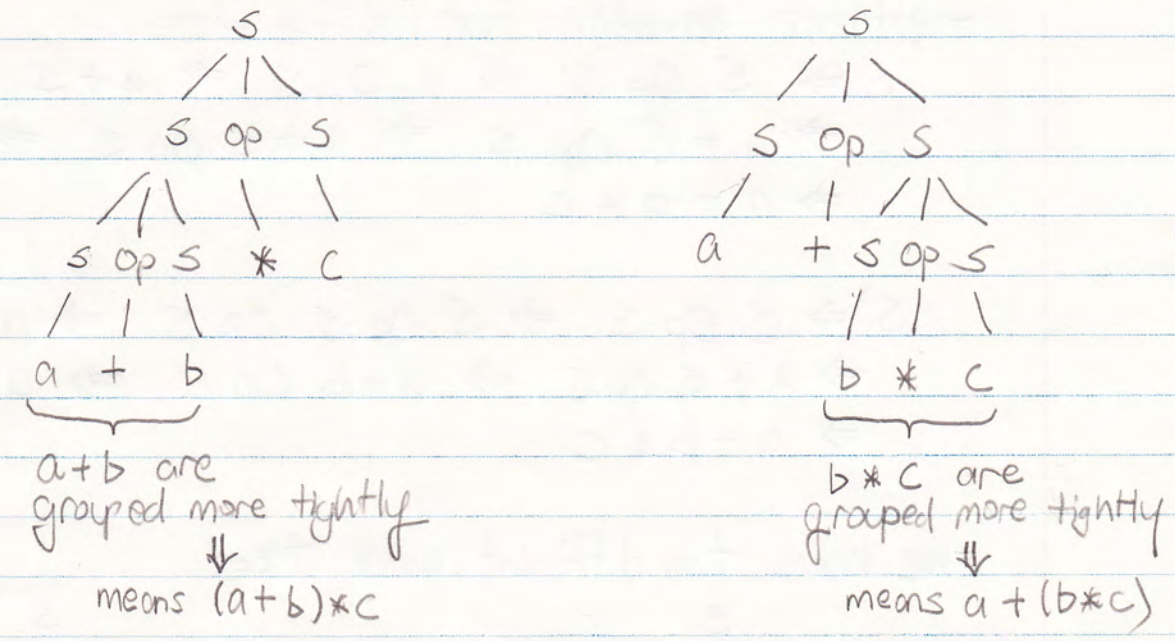


- a grammar for which some word has more than one distinct leftmost derivation (equivalent parse tree) is called ambiguous
  - if we only care whether $w \in L(G)$, ambiguity

doesn't matter

- but as complier writers, we want to know why $w \in L(G)$ (so the derivation matters)
  - the shape of the parse tree describes the meaning of the string
  - so a word with multiple parse tree can have multiple meanings

- example
  - in the last example, we had two parse trees for $a+b*c$



- so $a+b*c$ could mean $(a+b)*c$ or $a+(b*c)$
  - what do we do about this?
    (1) use heuristics ("precedence") to guide the derivation process
    (2) make the grammar unambiguous

MIDTERM CUTOFF

- consider an alternative grammar

$E \rightarrow E \text{ Op } T \mid T$

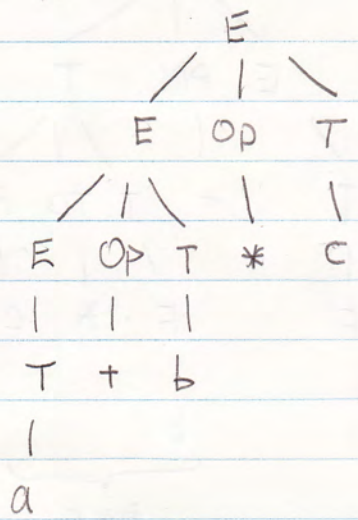
$T \rightarrow a \mid b \mid c \mid (E)$

$Op \rightarrow + \mid - \mid * \mid /$

- example
  - leftmost derivation for $a + b * c$

$$E \Rightarrow E\ Op\ T \Rightarrow E\ Op\ T\ Op\ T \Rightarrow T\ Op\ T\ Op\ T$$
$$\Rightarrow a\ Op\ T\ Op\ T \Rightarrow a + T\ Op\ T$$
$$\Rightarrow a + b\ Op\ T \Rightarrow a + b * T \Rightarrow a + b * T$$

  - only one parse tree



- this is a strict left-to-right precedence
- do whatever comes first

- what if we want to give *, / precedence over +, -?
- things that lower in the parse tree tend to be grouped more tightly
- we could we this as a sum of products, so products get grouped lower in the parse tree

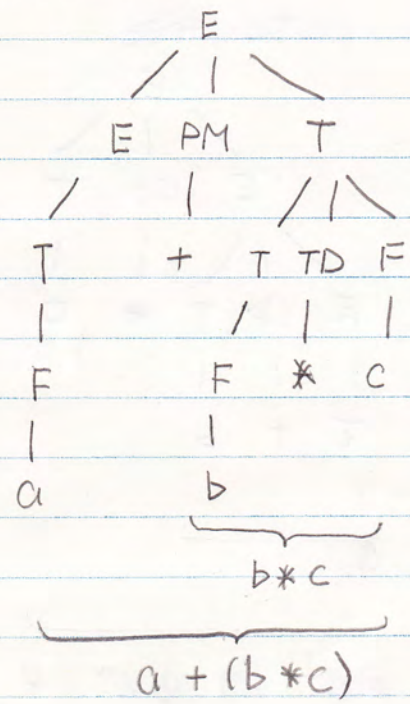
- consider the following grammar

$E \rightarrow E\ PM\ T \mid T$
$T \rightarrow T\ TD\ F \mid F$
$F \rightarrow a \mid b \mid c \mid (E)$
$PM \rightarrow + \mid -$
$TD \rightarrow * \mid /$

- example
  - leftmost derivation for $a+b*c$

$$E \Rightarrow E\ PM\ T \Rightarrow T\ PM\ T \Rightarrow F\ PM\ T$$
$$\Rightarrow a\ PM\ T \Rightarrow a + T \Rightarrow a + T\ TD\ F$$
$$\Rightarrow a + F\ TD\ F \Rightarrow a + b\ TD\ F$$
$$\Rightarrow a + b * F \Rightarrow a + b * c$$

  - the parse tree becomes



- if L is context-free, is there always an unambiguous grammar G such that $L = L(G)$?
  - no, there are inherently ambiguous languages that only have ambiguous grammar.

- can we construct a tool that tells us if a grammar is ambiguous?
  - no, it is undecidable
  - the equivalence of grammars G_1 + G_2
    - ie, is $L(G_1) = L(G_2)$ is also undecidable

* Recognizers
- what class of computer programs is needed to recognize a context-free language?
  - regular language : DFA
    - essentially a program with finite memory

- context-free languages: NFA + stack
  - infinite memory, but use is limited to LIFO order

- but we need more than just a yes/no answer
  - we need the derivation (parse tree)
    - the problem of finding the derivation is called parsing

- example
  - given: grammar G, start symbol S, word w
  - find: $S \Rightarrow \underbrace{\ldots}_{\text{these steps}} \Rightarrow w$
    - or report that there is no derivation

- how can this be done?
  - we have 2 choices:
    (1) forwards - "top-down"
      - start at S, expand until you reach w

    (2) backwards - "bottom-up"
      - start at w, figure out how to get back to S

* Top-Down Parsing
- start with S, apply grammar rule, reach w
  $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow w$
- use the stack to store intermediate α_i, in reverse, match against chars in w
- invariant: consumed input + reverse (stack contents) $= \alpha_i$

- example

- consider grammar

$S \rightarrow A\,y\,B$
$A \rightarrow ab$
$A \rightarrow cd$
$B \rightarrow z$
$B \rightarrow wx$

- For simplicity, we will use augmented grammars for parsing
  - ie, invent two new symbol $\vdash$, $\dashv$, new start symbol S'

BOF EOF

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

- example
  - consider $w = \vdash abywx \dashv$

Stack	Read Input	Unread Input	Action
S'	ε	$\vdash abywx \dashv$	Pop S'; Push $\dashv, S, \vdash$
$\dashv S \vdash$	ε	$\vdash abywx \dashv$	Match $\vdash$
$\dashv S$	$\vdash$	$abywx \dashv$	Pop S; Push B, y, A
$\dashv B\,y\,A$	$\vdash$	$abywx \dashv$	Pop A; Push b, a
$\dashv B\,y\,b\,a$	$\vdash$	$abywx \dashv$	Match a
$\dashv B\,y\,b$	$\vdash a$	$bywx \dashv$	Match b
$\dashv B\,y$	$\vdash ab$	$ywx \dashv$	Match y

Andy Yin
Bradley Lushman
CS 241
Lecture
Feb 26, 2015

* Parsing
  - recall
    - when top of stack is a terminal, pop and match against input
    - when top of stack is a non-terminal A, pop A and push α^R (α reversed) where $A \rightarrow \alpha$ is a grammar rule
    - accept when stack and input both empty

  - however, what if there is more than one production with A on the left-hand side?
    - how can we know which one to pick?

  - brute force
    - try all combinations until one works
    - very inefficient, we want a deterministic approach

  - our solution
    - use the next symbol of input (look ahead) to help decide
    - construct a predictor table
      - given a non-terminal on stack and input symbol, tell which production rule to use
      - ie, given a stack and input

RHS

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

(1) S` → ⊢ S ⊣
(2) S → A y B
(3) A → a b
(4) A → c d
(5) B → z
(6) B → w x

- for empty cells, produce an error
- descriptive error message: parse error row, col (use row/col of next input char), expecting one of ....
  - use chars for which the current stack top has entries in the table

- eg, if A is on top, produce message: "expecting a or c"

- what if a cell contains more than one rule?
  - the method breaks down, but we could restrict our grammar
  - a grammar is LL(1) if each cell of the predictor table has at most one entry
    - define LL(1) as LL(1) = left-to-right scan of input
      = leftmost derivations produced
      = one symbol of look-ahead

* Automatically Computing the Predictor Table
- define predict(A, a) as a function that produces the rules that apply when A is on the stack, a is the next input char

predict(A, a) = { A → β | a ∈ first(β) }
first(β), β ∈ U* where U* is the set of characters that

can be the first letter of a word derived from β

- so, $First(\beta) = \{a \mid \beta \Rightarrow^* a\gamma\}$
- and, $predict(A,a) = \{A \to \beta \mid \beta \Rightarrow^* a\gamma\}$
  - but this is not quite right, what if $A \Rightarrow^* \varepsilon$?
    - then "a" might not come from A, but from something after A

- so $predict(A,a) = \{A \to \beta \mid a \in First(\beta)\} \cup \{A \to \beta \mid nullable(\beta),\ a \in follow(A)\}$

$$nullable(\beta) = \begin{cases} true, & \text{if } \beta \Rightarrow^* \varepsilon \\ false, & \text{otherwise} \end{cases}$$

$follow(A) = \{b \mid S' \Rightarrow^* \alpha A b \beta\}$

- terminal symbols that can come immediately after A in a derivation, starting from S'

* Computing Nullable

```
initialize Nullable[A] = false for all A,
  for each rule B -> B1 ... Bn {
   if n = 0 or Nullable[Bi] for all i = 1,..,n
     Nullable[B] = true
       // consider  A -> B   /*need to
       //           B -> C    rerun
       //           C -> ε    procedure */
```

repeat until nothing changes $\Rightarrow O(n^2)$

* Computing First

```
initialize First[A] = {} for all A,
  for each rule A -> B1 ... Bk
   For i = 1..k
     if Bi is terminal a
       First[A] U= {a}      // U= unions
       break
```

repeat until nothing changes

```
else
    First[A] ∪= First[Bi]
    if not Nullable[Bi], break

//consider   A -> B     /* A -> d is
             B -> C       also true,
             C -> d       so repeat */
```

* Computing First*

- the "First set" for a string of symbol

```
first*(Y1 ... Yn) {
  result = φ
  for i = 1 ... n {
    if Yi ∈ Σ   (terminal)
      result ∪= {Yi}
      break
    else        (Yi ∈ N)
      result ∪= First[Yi]
      if not Nullable[Yi], break
  }
  return result
}
```

* Computing Follow

```
initialilize Follow[Ai] = {} for all A ≠ S`
repeat until nothing changes {
  for each rule A -> B1 ... Bn
    for i = 1..n
      if Bi ∈ N   (only do Follow for non-terminals)
        Follow[Bi] ∪= First*(Bi+1 ... Bn)
        if all of Bi+1, ..., B are nullable (including i=n)
          Follow[Bi] ∪= Follow[A]
}
```

Andy Yin
Sean Macrep
CS 241
Review Session
Mar 2, 2015

*Memory
- no meaning is given for some byte in memory
  - we have to remember the meaning

*Binary Numbers
- Unsigned (Positive) Numbers
  - convert 241 to binary
    11100001
- Signed Numbers
  - convert −73 to two's complement
    73 = 01001001
    −73 = 10110110 + 1
    = 10110111
  - for two's complement numbers, we notice
    $x - x = x + (-x) = 0$
    due to overflow
  - ie, $4 - 4 = 4 + (-4)$
    $= 100 + (1100)$
    $= 101000$

*MERL
- example
  fact: ; pass me n in $1
  ; return n! in $3

```
sw $1, -4($30)        ; store registers onto
sw $2, -8($30)        ; the stack, -4 cause
lis $2                ; it's the next valid point
.word -8              ; in memory
add $30, $30, $2
lis $2
.word -1    ; decrement constant
lis $3
.word 1     ; answer
loop:
  beq $1, $0, endloop
  mult $1, $3
  mflo $3
  add $1, $2, $2
  beq $0, $0, loop
endloop:
 lis $2
 .word 8
 lw $1, -4($30)
 lw $2, -8($30)
 jr $31
```

-when multiplying a negative/positive number together, the properties of two's complements carries over from high to low

-ie, multiply -7, 4 => -28

$\underbrace{111\ldots1}_{32\text{ bits}}$ $\quad \underset{\uparrow\, 2^{32}}{111\ldots}100100$

$\uparrow$ 32

*MERL Linking

-example

```
0x00 : 0x10000002        ; cookie
0x04 : 0x00000048        ; length
0x08 : 0x00000028        ; code length / MERL Meta Start
0x0C : 0x10030002
0x10 : 0x00008814
0x14 : 0x0000000C
0x18 : 0x0000f014
0x1C : 0x00000018
0x20 : 0x00000040
0x24 : 0x00000040
0x28 : 0x00000001        ; REL
0x2c : 0x00000020
0x30 : 0x00000001        ; REL
0x34 : 0x00000014
0x38 : 0x00000001        ; REL
0x3C : 0x00000024
0x40 : 0x00000001        ; REL
0x44 : 0x6000001C
```

-For loading the file at location 0x20, add 0x20 the values of the address after REL

-For linking the file with codesize 0x20, add 0x20 to values of the address after REL, and add 0x20 to all the REL (if this file is linked below), and update length, code length

-example

```
.export myproc
.import proc
```

```
self: beq $0, $0, self
myproc: .word proc ; <- 0x00
```

- don't have an ESR and REL on the same line
  - all ESRs will eventually become REL

*DFAs

- we have
  - Q ← states
  - Σ ← alphabet
  - Q_0 ← start state
  - δ ← transistion function
  - A ← accepting states

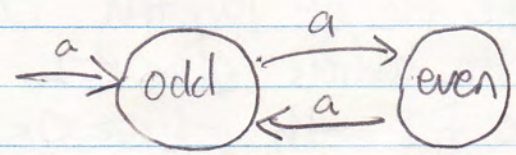
$\delta(v \in Q, c \in \Sigma) \rightarrow v \in Q$

DFA: δ T: $Q \times \Sigma \rightarrow Q$ (one state)

NFA: δ T: $Q \times \Sigma \rightarrow 2^Q$ (any number of states in the power set)

- example
  - consider $\Sigma = \{a\}$, if odd accept
    - DFA

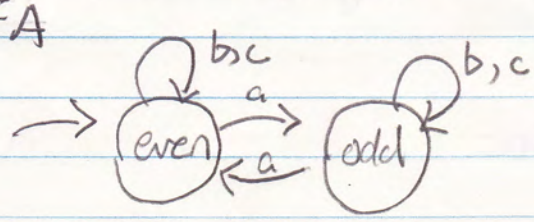
$R = a(aa)^*$



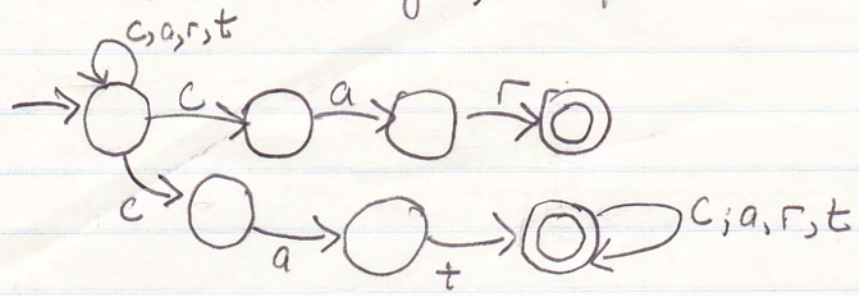
- consider $\Sigma = \{a, b, c\}$, if odd a, accept

  $R = (b|c)^* a ((b|c)^* a (b|c)^* a)^* (b|c)^*$

  - DFA



- consider $\Sigma = \{c, a, r, t\}$, if it ends in car, or has cat substring, accept



Andy Yin
Bradley Lushman
CS 241
Lecture
Mar 3, 2015

* Top-down Parsing

- example

(1) $S' \rightarrow \vdash S \dashv$
(2) $S \rightarrow b\,S\,d$
(3) $S \rightarrow p\,S\,q$
(4) $S \rightarrow c$
(5) $c \rightarrow l\,c$
(6) $c \rightarrow \varepsilon$

- For nullable

iteration	0	1	2	3
S'	false	false	false	false
S	false	false	true	true
c	false	true	true	true

- we note that anything with something on the right-hand side cannot be nullable
  - ie, (1) has a $\dashv$ symbol after S
    (2) has a d symbol after S
    (3) has a q symbol after S

- For First

iteration	0	1	2	3
S'	{ }	{⊢}	{⊢}	{⊢}
S	{ }	{b, p}	{b, p, l}	{b, p, l}
c	{ }	{l}	{l}	{l}

- For follow,

iteration	0	1	2
S	{ }	{⊣, d, q}	{⊣, d, q}
c	{ }	{⊣, d, q}	{⊣, d, q}

- for predict, recall predict(A, a) = $\{A \to \beta \mid a \in first^*(\beta)\} \cup \{A \to \beta \mid nullable(\beta),\ a \in Follow(A)\}$

- predictor table

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

- a grammar is LL(1) if
  - no two distinct productions with the same left-hand sides can generate the same first terminal symbol (ie, $|\{A \to \beta \mid a \in first^*(\beta)\}| \leq 1$)
  - there is only one way to send a nullable symbol to ε (ie, $|\{A \to \beta \mid nullable(\beta),\ a \in Follow(A)\}| \leq 1$)
  - no nullable symbol has the same terminal in both its first set and its follow set (ie, $|\{A \to \beta \mid a \in first^*(\beta)\}| + |\{A \to \beta \mid nullable(\beta),\ a \in follow(A)\}| \leq 1$)

- example

$$\left.\begin{array}{l} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to a \mid b \mid c \end{array}\right\} \text{ not LL(1)}$$

- this grammar has left-recursion
  - ie, we can derive the same symbol for first

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T$

$E \Rightarrow T \Rightarrow F \Rightarrow a$

some first symbol

- left-recursion is always not LL(1), what about right-recursion?

$$\left.\begin{array}{l} E \rightarrow T + E \mid T \\ T \rightarrow F * T \mid F \\ F \rightarrow a \mid b \mid c \end{array}\right\} \text{ still not LL(1)}$$

- in fact, for right recursion, T obviously generates the same first symbol

- what if we factor?

$$\left.\begin{array}{l} E \rightarrow T E' \\ E' \rightarrow \varepsilon \mid + E \\ T \rightarrow F T' \\ T' \rightarrow \varepsilon \mid * T \\ F \rightarrow a \mid b \mid c \end{array}\right\} \text{ is LL(1)}$$

- so LL(1) does not work well with left-associativity

* Bottom-up parsing

- for bottom up parsing, we go from w to S'
- the stack stores partially reduced info read so far
  - ie, $w \Leftarrow \alpha_k \Leftarrow a_{k-1} \Leftarrow \dots \Leftarrow a_1 \Leftarrow S'$
- the invariant: stack + unread input = α_i (or w or S')

- example
  - consider the following grammar

$S' \rightarrow \vdash S \dashv$

$S \rightarrow AyB$

$A \rightarrow ab$

$A \to cd$

$B \to z$

$B \to wx$

- for input: $\vdash abywx \dashv$

stack	read	unread	action
	ε	$\vdash abywx \dashv$	shift $\vdash$
$\vdash$	$\vdash$	$abywx \dashv$	shift a
$\vdash a$	$\vdash a$	$bywx \dashv$	shift b
$\vdash ab$	$\vdash ab$	$ywx \dashv$	reduce $A \to ab$, pop b, pop a push A
$\vdash A$	$\vdash ab$	$ywx \dashv$	shift y
$\vdash Ay$	$\vdash aby$	$wx \dashv$	shift w
$\vdash Ayw$	$\vdash abyw$	$x \dashv$	shift x
$\vdash Aywx$	$\vdash abywx$	$\dashv$	reduce $B \to wx$, pop x, pop w push B
$\vdash AyB$	$\vdash abywx$	$\dashv$	reduce $S \to AyB$, pop B, pop y, pop A push S
$\vdash S$	$\vdash abywx$	$\dashv$	shift $\dashv$
$\vdash S \dashv$	$\vdash abywx \dashv$	ε	reduce $S' \to \vdash S \dashv$
S'	$\vdash abywx \dashv$	ε	Accept

- choice at each step
  (1) shift a char from input to stack
  (2) reduce: the top of the stack is the right-hand side of a grammar rule
    - replace with left-hand side

- accept if stack contains S' when input is ε
  - equivalent to $\vdash S \dashv$ on empty input
    to accepting when you shift $\dashv$
- the challenge is when to use rule (1) and (2)
  - how do we know whether to shift or reduce
    - we can use the next char to help decide, still a hard problem

- theorem (Donald Knuth, 1965):
  - the set $\{ wa \mid \exists x.\ S' \Rightarrow^* wax \}$ is a regular language

    stack next input char

  - can be described by a DFA
  - use the DFA to make shift/reduce decisions
    - the resulting method is called LR parsing
      - left-to-right scan
      - right-most derivation

Andy Yin
Bradley Lushman
CS 241
Lecture
Mar 5, 2015

* LR Parsing

- recall Knuth's Theorem
  {stack + char} is regular
  - use a DFA to make shift/reduce decisions

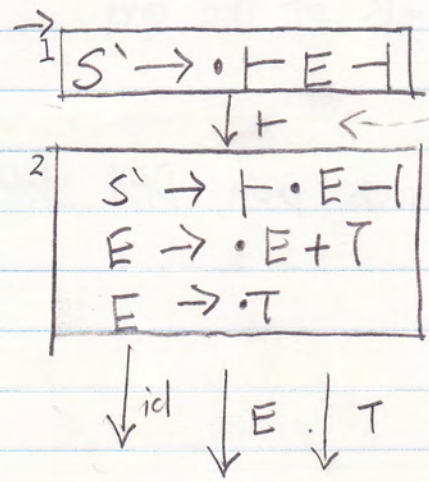
- example
  - consider the following CFG

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

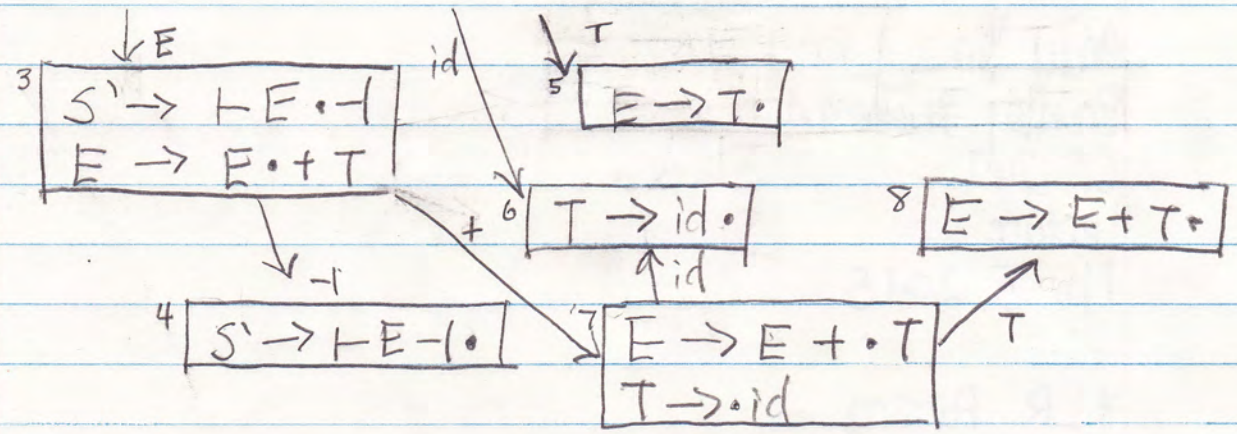
  - we will use LR(0) automation (simplest)

* LR(0) Automation

- an item is a production with a dot (·) somewhere on the right-hand side
  - indicates a partially completed rule



- label an arc with the symbol that follows the dot
  - advance the dot in the next state
- if the dot precedes a non-terminal A, add all items A → · β (dot on the left



- using the machine
  - start in the start state with empty state
- shifting
  - shift a char from input to stack
  - follow the transistion for the char in the machine (if there is one)
    - if none, either reduce or error
- reducing
  - "reduce" states contain a complete item (dot is right-most) alone in the state
    - reduce by the rule in the state
    - pop right-hand side off the stack, backtrack size(RHS) states in the DFA.
      - follow transistion for LHS
      - push LHS onto stack
- accept if S' is on the stack at the end

* Backtracking the DFA

- we must remember DFA states, so push DFA states on the stack as well
- example

  ⊢id + id + id ⊣

Stack	Read	Unread	Action
1	ε	⊢id+id+id⊣	S2 (shift ⊢ go to state 2)
1⊢2	⊢	id+id+id⊣	S6
1⊢2id6	⊢id	+id+id⊣	R, T → id, pop 1 symbol + 1 state, now in state 2, push T, goto 5
1⊢2T5	⊢id	+id+id⊣	R, E → T
1⊢2E3	⊢id	+id+id⊣	S7
1⊢2E3+7	⊢id+	id+id⊣	S6
1⊢2E3+7id6	⊢id+id	+id⊣	R, T → id
1⊢2E3+7T8	⊢id+id	+id⊣	R, E → E+T
1⊢2E3	⊢id+id	+id⊣	S7
1⊢2E3+7	⊢id+id+	id⊣	S6
1⊢2E3+7id6	⊢id+id+id	⊣	R, T → id
1⊢2E3+7T8	⊢id+id+id	⊣	R, E → E+T
1⊢2E3	⊢id+id+id	⊣	S4
1⊢2E3⊣4	⊢id+id+id⊣	ε	R, S` → ⊢E⊣

I S' ⊢id+id+id⊣ ε Accept

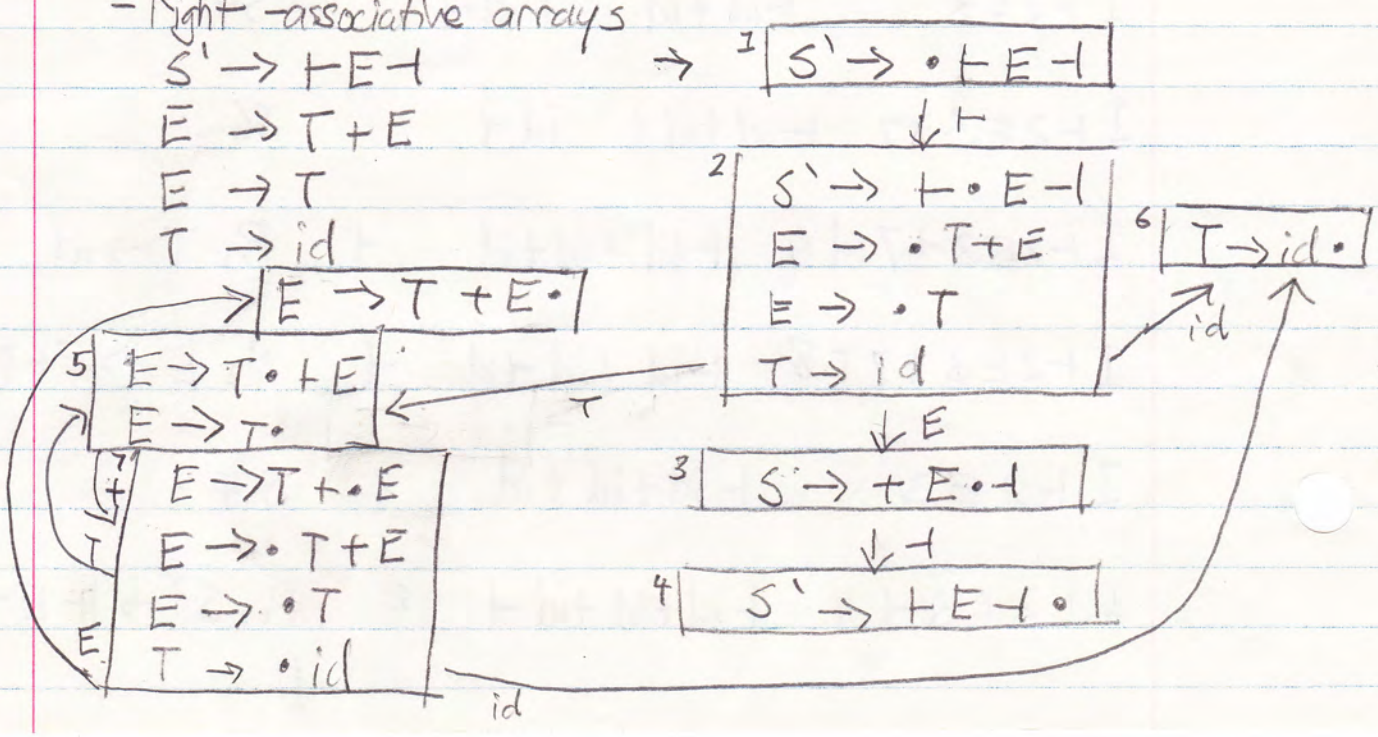
- what can go wrong?
  - what if the state looks like: $\boxed{\begin{array}{l} A \to \alpha \cdot c\beta \\ B \to \gamma \cdot \end{array}}$?
    - do we try to shift c (as suggested by $A \to \alpha \cdot c\beta$), or reduce by $B \to \gamma$ (as suggested by $B \to \gamma \cdot$)?
  - this is a shift-reduce conflict

- what if the state looks like: $\boxed{\begin{array}{l} A \to \alpha \cdot \\ B \to \beta \cdot \end{array}}$?
  - do we reduce by $A \to \alpha$ or by $B \to \beta$?
  - this is a reduce-reduce conflict

- if any complete item $A \to \alpha \cdot$ is not alone in a state, there will be a shift-reduce or a reduce-reduce conflict, and the grammar is not LR(0)

- example
  - right-associative arrays

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



(5) has a shift-reduce conflict
$E \to T \cdot + E$ is a shift
$E \to T \cdot$ is a reduce

- eg, if input starts with $\vdash$ id...

$$\boxed{1} \xrightarrow[S\vdash]{} \boxed{1 \vdash 2} \xrightarrow[S\,id]{} \boxed{1 \vdash 2\, id\, 6}$$

$$\xrightarrow[R,\, T \to id]{} \boxed{1 \vdash 2\, T\, 5}$$

- reduce $\vdash T$.., should we reduce $E \to T$?
  - if input is $\vdash$ id $\dashv$, then yes
  - if input is $\vdash$ id + ..., then No

- add lookahead to the machine to fix the conflict
  - for each $A \to \alpha \cdot$, attach follow(A)
    follow(E) = $\{\dashv\}$
    follow(T) = $\{+, \dashv\}$

$$\boxed{\begin{array}{l} E \to T \cdot + E \\ E \to T \cdot \end{array}} \quad \text{becomes} \quad \boxed{\begin{array}{ll} E \to T \cdot + E & \\ E \to T \cdot & \dashv \end{array}}$$

← sticky note

- interpretation
  - a reduce action $\boxed{A \to \alpha \cdot \quad x}$ (x = follow(A))
  only applies if the next char $\in X$
  - so $\boxed{E \to T \cdot \quad \dashv}$ only applies if next char is $\dashv$
    $\boxed{E \to T \cdot + E}$ applies when the next char is +
  - conflict is resolved

- this result is called a SLR(1) parser
  SLR(1) = Simple LR with 1 char lookahead
- SLR(1) resolves many, but not all conflicts

Andy Yin
Bradley Lushman
CS 241
Lecture
Mar 10, 2015

* LR parsing
  - SLR(1) resolves many, but not all conflicts
  - LR(1) is more sophisticated than SLR(1), and parses strictly more grammar
    - disadvantage of LR(1) is that the machine is complex, and many states
  - LALR(1) is between SLR(1) + LR(1)

  - LR(1), SLR(1), LALR(1) are identical as parsing algorithms
    - difference is entirely in the construction of the machine

* Building the Parse Tree
  - top-down parsing

⊣ S	Pop S; push B, y, A
⊣ B y A	keep S; make the new node its children

```
 \|/
  S
```

  - wlp4
    - the parser should outputs a .wlp4i file
    - eg,

      S → BOF e EOF
      e → e + t
      e → t

t → ID

- input: BOF ID + ID + ID EOF
  (a) (b) (c)

- then, .wlp4i file is

```
6 BOF e EOF
BOF BOF
e e+t
e e+t
e t
t ID
ID a
+ +
t ID
ID b
+ +
t ID
ID c
EOF EOF
```

* Context-Sensitive Analysis

- what properties of valid C programs cannot be enforced by CFG's?
  - declaration-before-use
  - type checking
  - const correctness
  - scoping
  - etc

- we perform context-sensitive analysis (aka semantic analysis) by traversing the parse tree

- example

```
class Tree {
  public:
    vector<string> rule;
    vector<Tree*> children;
};

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

- what semantic analysis is needed for wlp4?
- what errors needs to be caught?
  - type errors
  - variables or procedures declared more than once } scope
  - variables or procedures used, but not declared }

* Declaration Errors
- multiple/missing declaration
- how do we do this?
  - construct a symbol table (3rd occurrence)
  - traverse the parse tree to collect all declaration
  - for each node whose rule is
    dcl → TYPE ID
  - extract name (eg. x) and type (int or int*) and add to symbol table

- if name is already in the symbol table, error
- multiple dcls are now checked

- traverse the parse tree again
  - look for rules of the form
    factor → ID
    lvalue → ID
  - if ID's name ∉ symbol table, error
- undeclared vars are now checked

- these two passes could be merged
  - why can they be merged in the compiler, but not in the assembler?
  - the grammar ensures that declarations are done first

* Symbol Table Implementation
- it can be a global variable

```
map<string, string> symbol_table;
```

name ↗ ↖ type

- but, this doesn't take scope in account
  - doesn't check procedure declarations

- example
  - consider the following

```
int f() {
    int x = 0;
    return 1;
}

int wain(int a, int b) {
```

```
int x=0;            // duplicated, but ok, since in
return 1;           // different scope
}
```

- how can we forbid duplicated declarations in the same procedure, but allow them in different procedures?
  - also,

    int f() {...}
    int f() {...}

    } how do we forbid this?
  - solution
    - a separate symbol table for each procedure?
    - have one "top-level" symbol table that collects all procedure names

      map <string, map <string, string> > top-symbol-table;

      procedure's name — procedure's symbol table
- when traversing the parse tree
  - if the node's rule is

    procedure → INT ID LPAREN

    or

    main → INT WAIN LPAREN
  - new procedure
    - make sure name is not already in symbol table
    - if not, create a new entry (new local symbol table)
  - may want global variable: current_procedure
    - stores the name of the procedure you are currently in
    - update each time you encounter

      procedure →..

      main →..

- For variables, we store its declared type in the symbol table
  - is there type information for procedures?
    - the signature of the procedure
    - note that all procedures in WIP4 returns int, the signature is just the sequence of parameter types
  - store this in the top-level symbol table

```
map < string , pair < vector < string >, ← signature and type
procedure name ↗                  map < string, string > > > ← local symbol table
top_symbol_table;
```

* Computing the Signature
- traverse nodes of the form
  - paramlist → dcl
  - paramlist → dcl COMMA paramlist
- if params → , then signature is empty

- all of this can be done in one pass

Andy Yin
Bradley Lushman
CS 241
Lecture
Mar 12, 2015

*Types
- types let us assign an interpretation to the contents of some memory address
- a good type system prevents us from reinterpreting the bits as something else

- example

```
int* a = NULL;      // "a" denotes a pointer
a = 7;              // attempting to store an int where
// ↑ error          // a pointer is needed
```

- to check type correctness,
  - find a type for each var/expression
  - ensure all operators are applied to operands of the right type

- how do we determine the type of a variable?
  - we will use the symbol table
  - example

```
int wain(int a, int b) {      /*  Symbol Table
    int* c = NULL;            *  Name    Type
    return a;                 *  a       int
}                             *  b       int
                              *  c       int*
                              */
```

* Walk the Tree
  - for each rule, its grammar rule tells you the relevant type rules for that expression
    - if no rule applies, then error

- example

```
string TypeOF(Tree &t){
    for each c ∈ t.children{
        compute Type OF(c)
    // use t.rule to decide which type rules are
       relevant
    // combine types of childen to produce the type of t
    // if not possible, error
    }
}
```

- example
  - ID - get its type from symbol table

$$\frac{\langle id.name, \tau \rangle \in decls}{id.name : \tau}$$ (from CS 245)

  - ie, if id.name is declared to have type τ, then id.name has type τ

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

- example

- singleton productions

$$\left.\begin{array}{l} \text{expr} \rightarrow \text{term} \\ \text{term} \rightarrow \text{factor} \\ \text{factor} \rightarrow \text{id} \end{array}\right\} \text{type of LHS = type of RHS}$$

- simple/trivial cases

$$\frac{}{\text{NUM}: \text{int}} \qquad \frac{}{\text{NULL}: \text{int}*}$$

- parenthesis case

$$\frac{E: \tau}{(E): \tau}$$

- if E has type τ, the (E) has type τ

- address-of case

$$\frac{E: \text{int}}{\&E: \text{int}*}$$

- if E has type int, then &E has type int*

- dereference case

$$\frac{E: \text{int}*}{*E: \text{int}}$$

- if E has type int*, then *E has type int

- allocation case

$$\frac{E: \text{int}}{\text{new int}[E]: \text{int}*}$$

- multipulcation / division cases

$$\frac{E_1: \text{int},\ E_2: \text{int}}{E_1 * E_2: \text{int}} \qquad \frac{E_1: \text{int},\ E_2: \text{int}}{E_1 / E_2: \text{int}} \qquad \frac{E_1: \text{int},\ E_2: \text{int}}{E_1 \% E_2: \text{int}}$$

- addition cases

$$\frac{E_1: \text{int},\ E_2: \text{int}}{E_1 + E_2: \text{int}} \qquad \frac{E_1: \text{int}*,\ E_2: \text{int}}{E_1 + E_2: \text{int}*} \qquad \frac{E_1: \text{int},\ E_2: \text{int}*}{E_1 + E_2: \text{int}*}$$

(int + int) (int* + int) (int + int*)

- subtraction cases

$$\frac{E_1: \text{int},\ E_2: \text{int}}{E_1 - E_2: \text{int}} \qquad \frac{E_1: \text{int}*,\ E_2: \text{int}}{E_1 - E_2: \text{int}*} \qquad \frac{E_1: \text{int}*,\ E_2: \text{int}*}{E_1 - E_2: \text{int}}$$

(int - int) (int* - int) (difference in the address, divided by 4, number array slots)

- additional type constraints
  while (T) { S }
  if (T) { S_1 } else { S_2 }
- the Test T should be boolean, even though there is no bool type in wlp4
  - the grammar ensures that T will be a comparison
  - so as long as the comparison is well typed, it will be boolean
  - any expression with a type is well-typed

$$\frac{E : \tau}{\text{well-typed}(E)}$$

  - tests case

$$\frac{E_1 : \tau, \quad E_2 : \tau}{\text{well-typed}(E_1 == E_2)} \qquad \frac{E_1 : \tau, \quad E_2 : \tau}{\text{well-type}(E_1 \; != E_2)}$$

$$\frac{E_1 : \tau, \quad E_2 : \tau}{\text{well-typed}(E_1 < E_2)}$$

  - assignment case

$$\frac{E_1 : \tau, \quad E_2 : \tau}{\text{well-typed}(E_1 = E_2)}$$

  - print case

$$\frac{E : int}{\text{well-typed}(\text{print } E)}$$

  - deallocation case

$$\frac{E : int*}{\text{well-typed}(\text{delete } [] \; E)}$$

  - if case

$$\frac{\text{well-typed}(T), \quad \text{well-typed}(S_1), \quad \text{well-typed}(S_2)}{\text{well-typed}(\text{if } (T) \{ S_1 \} \text{ else } \{ S_2 \})}$$

  - while case

$$\frac{\text{well-typed}(T), \quad \text{well-typed}(S)}{\text{well-typed}(\text{while}(T) \{ S \})}$$

- declarations case

$$\frac{E : \tau}{\text{well-typed}(\tau\ \text{ID} = E)}$$

- sequence of statements case

$$\frac{}{\text{well-typed}(\varepsilon)} \qquad \frac{\text{well-typed}(S_1),\ \text{well-typed}(S_2)}{\text{well-typed}(S_1\ S_2)}$$

- procedures case

$$\frac{\text{well-typed}(\text{dcls}),\ \text{well-typed}(\text{stmts}),\ E : \text{int}}{\text{well-typed}(\text{INT ID}(\text{args})\ \{\ \text{dcls}\ \ \text{stmts}\ \ \text{return}\ E\ \})}$$

- Main case

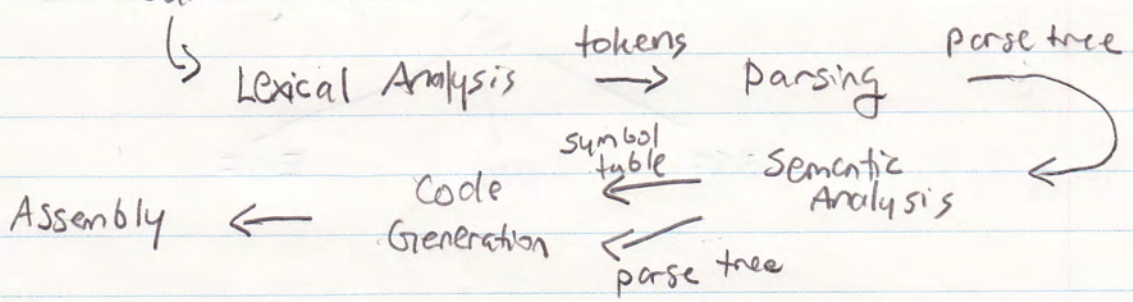
$$\frac{\text{well-typed}(\text{dcls}),\ \text{well-typed}(\text{stmts}),\ E : \text{int},\ \text{dcl}_2 : \text{INT ID}}{\text{well-typed}(\text{INT WAIN}(\text{dcl}_1,\ \text{dcl}_2)\ \{\ \text{dcls},\ \text{stmts},\ \text{return}\ E\ \})}$$

- Function calls

$$\frac{E_1 : \tau_1 \cdots E_n : \tau_n,\ \langle f, \langle \tau_1, \ldots, \tau_n \rangle \rangle \in \text{topDecls}}{f(E_1, \ldots, E_n) : \text{int}}$$

Andy Yin
Bradley Lushman
CS 241
Lecture
Mar 17, 2015

*Code Generation
- by now, the source program has no compiler errors
  - must output equivalent MIPS assembly code



- there are infinitely many equivalent MIPS programs
  - which do we chose?
    - correct (essential)
    - easy (for CS241)
    - most efficient to compile
    - most efficient to run (fastest)
      - for A11, fastest means fewest instructions

- example
  int wain(int a, int b) { return a; }
  - convention
    - parameter of wain are held in $1, $2
      - since mips.twoints and mips.array put them there
    - output will be passed in $3

- MIPS code

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

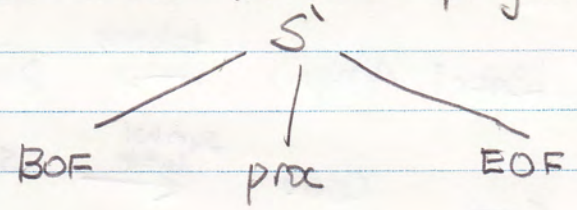
- example

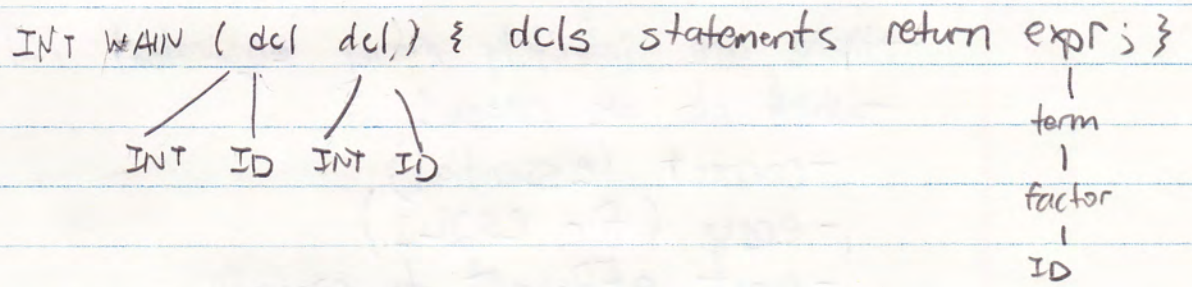
int wain(int a, int b) { return b; }

- MIPS

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

- the parse trees for both programs are the same





- how do we tell one apart from the other?
  - need to determine which of the two declared IDs the returned ID matches
  - how do we do that?
    - use the symbol table
    - add a field to the table that indicates where each symbol is stored

Name	Type	Location
a	int	$1
b	int	$2

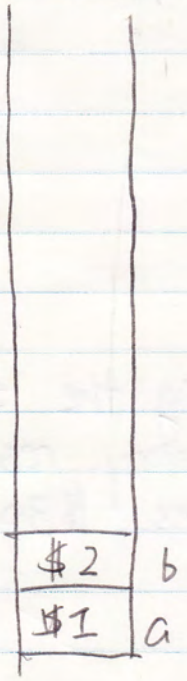
- then, when traversing the parse tree for code generation, when ID is encountered in return expr
  - look up ID in symbol table
  - if it was a, get back to $1
  - if it was b, get back to $2

expr
|
term
|
factor
|
ID

- what about local declarations?
  - can't put all in registers (might not be enough registers)
  - for simplicity, allocate stack space for all local variables, including the parameters of wain

- example

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




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

Name	Type	Offset
a	int	4
b	int	0

- we can't know the offsets until all declarations have been processed, since $30 changes with each new declaration

- example

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

⇓

```
sw $1, -4($30)
sw $2, -8($30)
lis $4
.word 4
sub $30, $30, $4
sub $30, $30, $4
sub $30, $30, $4        ; extra stack space for c
sw $0, 0($30)           ; c=0
lw $3, 8($30)
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
jr $31
```

Name	Type	Offset
a	int	8
b	int	4
c	int	0

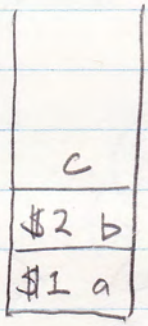
- using the stack for temporary results also changes $30

- conventions
  - $4 will always contain 4
  - $29 points to the bottom of the current stack frame
  - if offsets are calculated with respects to $29, they will be

constant
- this is called the frame pointer or base pointer

- example

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




```
lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sw $0, -4($30)      } int c=0;
sub $30, $30, $4
lw $3, 0($29)
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
jr $31
```

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

- what about more complicated programs?
  - eg, int wain(int a, int b) { return a+b; }
  - in general, for each grammar rule $A \rightarrow \gamma$, build code(A) from code(γ)
  - expand previous convention, use $3 for "output" of all expression
    - ie, a+b

$3 ← eval(a)
$3 ← eval(b) ; $3 is clobbered!
$3 ← $3 + $3

- need a place to store pending computations
  - could use a register

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

- what about a + (b + c)?

code (a)	($3 ← a)	
add $5, $3, $0	($5 ← a)	can't do $5 +
code (b)	($3 ← b)	$3, breaks
add $6, $3, $0	($6 ← b)	association
code (c)	($3 ← c)	
add $3, $3, $6	($3 ← b+c)	
add $3, $3, $5	($3 ← a + (b+c))	

  - this required 2 extra registers
  - similarly, a + (b + (c + d)) would need 3 extra registers
  - this approach would eventually run out of registers

- general solution
  - use the stack

code (a)	($3 ← a)
push ($3)	
code (b)	($3 ← b)
push ($3)	
code (c)	($3 ← c)
push (c)	
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)))
```

- need only 1 extra registers

- in general,

  $expr_1 \rightarrow expr_2 + term$

  $code(expr_1) = code(expr_2) +$
  push($3) +
  code(term) +
  pop($5) +
  add $3, $3, $5

- singleton rules
  - usually easy

  s → BOF procedures EOF
  code(s) = code(procedures)

  expr → term
  code(expr) = code(term)

- print
  - ie, println(expr)
    - prints the expr + \n to screen
    - this was done in A2

- runtime environment
  - set of procedures supplied by the compiler (or OS) to assist programs in their execution

- procedure print is part of the runtime environment
  - printinteger will be provided

Andy Yin
Bradley Lushman
CS 241
Lecture
Mar 19, 2015

*Print
- in print.merl, takes input in $1

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

- note, if using $1 to hold something else, save and restore it
  - also save and restore $31

```
code(println(expr);) = code(expr)
                       add $1, $3, $0
                       push ($31)
                       lis $5
                       .word print
                       jalr $5
                       pop ($31)
```

*Assignment Operator
statement -> lvalue BECOMES expr
- the lvalue denotes an address, and can occur to the left of =

- for now, lvalue = ID (no pointers yet)

```
code(statement) = code(expr)        ($3 ← expr)
                  + sw $3, ___      ($29)
```

- lookup ID's offset in symbol table

* IF and While
  - need boolean testing
  - suggested conventions
    - store 1 in $11
    - store print in $10

  - code so far

```
.import print
lis $4
.word 4
lis $10
.word print
lis $11
.word 1
sub $29, $30, $4
;; allocate space on stack for variables
 -- your code --
add $30, $29, $4
jr $31
```

prologue

epilogue

* Boolean Tests

test → $expr_1$ < $expr_2$

code(test) = code($expr_1$) ($\$3 \leftarrow expr_1$)

add $6, $3, $0 ($\$6 \leftarrow expr_1$)

;; can use this since evaluating expr only uses

;; $3 and $5

code($expr_2$) ($\$3 \leftarrow expr_2$)

slt $3, $6, $3

test → $expr_1 > expr_2$
- treat as $expr_2 < expr_1$

test → $expr_1 \neq expr_2$

```
code(test) = code(expr1)          ($3 ← expr1)
             add $6, $3, $0       ($6 ← expr1)
             code(expr2)          ($3 ← expr2)
             slt $7, $3, $6       ($7 ← $3 < $6)
             slt $8, $6, $3       ($8 ← $6 < $3)
             add $3, $7, $8       ($3 = $7 v $8)
```

at most one of $7, $8 can be 1

test → $expr_1 == expr_2$
- treat as NOT($expr_1 \neq expr_2$)
- take above and add
  sub $3, $11, $3 ($3 ← 1 − $3)

test → $expr_1 >= expr_2$ } exercise
$expr_1 <= expr_2$

*If

statement → if test $statements_1$, else $statements_2$

```
code(statement) = code(test)   ($3 ← test)
bne $3, $11, else  ⇔  beq $3, $0, else
code(statements1)
beq $0, $0, endif
else: code(statements2)
endif:
```

```
code(statement) = code(test)
bne $3, $0, true
beq $0, $0, endif
```

true : code(statements$_1$)
endif

- issues
  - need to generate unique label names
  - keep a counter X
  - use elseX, endifX, trueX for label names
  - increment counter at each new if-statement

*While
statement → while (test) statements
- use a counter Y to generate fresh labels

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

- advice
  - generate comments along with MIPS instructions
  - helps you decipher your output

*Pointers
- needs to support
  - NULL
  - dereference
  - address-of
  - comparison
  - pointer arithmetic
  - allocation / deallocation
  - assignment through pointer dereference

- NULL
  - could use 0, but use 1
    - guarantees the machine will crash if dereferenced

  Factor → NULL
  code(Factor) = add $3, $11, $10

- dereference
  factor → * expr
  code(factor) = code(expr) ($3 ← expr)
  + lw $3, 0($3)

- comparisons
  - same as integer comparison
  - except
    - no negative pointers
    - so pointer comparison should be unsigned, ie, use sltu instead of slt

- when generating code for
  test → $expr_1 < expr_2$
  - use slt if $expr_1$ and $expr_2$ have type int
    sltu if they have type int*
  - rerun "typeof" on $expr_1$ or $expr_2$ to pick the right slt/sltu
    - don't need to do it on both, since types will be the same
    - we did pass type checking

- better
  - add a "type" field to each tree node

- make A8 procedure "type of" store each node's type (if any) in the node itself, then retrieve it now

- pointer arithmetic
  $expr_1 \rightarrow expr_2 + term$
  $expr_1 \rightarrow expr_2 - term$
  - exact meaning of these depends on the types involved
    - addition
      $expr_1 \rightarrow expr_2 + term$
      if $expr_2$, term : int (as before)
      if $expr_2$ = int*, term : int ($expr_2 + 4 \cdot term$)
      code($expr_1$) = code($expr_2$) ($\$3 \leftarrow expr_2$)
      push($\$3$)
      code(term) ($\$3 \leftarrow term$)
      mult $\$3$, $\$4$
      mflo $\$3$
      pop($\$5$) ($\$5 \leftarrow expr_2$)
      add $\$3$, $\$3$, $\$5$

Andy Yin
Bradley Lushman
CS 241
Lecture
Mar 24, 2015

* Pointer Operation
- pointer addition

$expr_1 \rightarrow expr_2 + term$
if $expr_2$: int, term : int {
- as before

}
if $expr_2$: int*, term : int {
code ($expr_1$) = code($expr_2$) +
push($3) +
code(term) +
mult $3, $4 +
mflo $3 +
pop($5) +
add $3, $5, $3

}

- pointer subtraction

$expr_1 \rightarrow expr_2 - term$
if $expr_2$: int, term : int {
- as before

}
if $expr_2$: int*, term : int {
code ($expr_1$) = code($expr_2$) + ($3 ← $expr_2$)
push ($3) +
code(term) + ($3 ← term)
mult $3, $4 +

```
                                mflo $3 +
                                pop($5) + ($5 ← expr)
                                sub $3, $5, $3
}
if expr2 : int*, term: int* {
  // means  expr2 - term
  //         ----------
  //              4
    code(expr1) = code(expr2) +     ($3 ← expr2)
                  push($3) +
                  code(term) +      ($3 ← term)
                  pop($5) +         ($5 ← expr2)
                  sub $3, $5, $3 +  ($5 ← expr2 - term)
                  div $3, $4 +
                  mflo $3
}
```

- assignment through pointer dereferences
  - eg,
```
*x = y;
*(x+z) = y;
*(new int[10]) = y;
```
  LHS = address at which to store the value
  RHS = the value

statement → ID BECOMES expr; // already done
statement → STAR factor BECOMES expr;
- calculate value of factor
  - use that as the address that stores expr
- note that we don't actually dereference the lvalue, and we only care about the address (not value)

code(statement) = code(factor) + ($3 ← factor)

```
push($3) +
code(expr) +    ($3 <- expr)
pop($5) +       ($5 <- factor)
sw $3, 0($5)
```

- address of
  - & applies only to lvalues
  - two cases
    - ID

```
factor1 -> AMP lvalue
  if lvalue = ID {
    code(factor) = lis $3      +
                   .word ...   +// lookup ID offset
                               //in symbol table
                   add $3, $29, $3
}
```

    - STAR factor

```
factor1 -> AMP factor2
  if lvalue = STAR factor2 {
    code(factor1) = code(factor2)
}
```

- new/delete
  - part of the runtime environment
    - alloc, merl
      - must be linked last
      - it must know where the end of your program is to create the heap

- add to the prologue
  .import init

```
.import new
.import delete
/* function init sets up the allocator data structure,
must be called once at the beginning of the
assembly file (ie, in the prologue) */
//takes a param in $2
  if calling with mips.array {
    //ie, first param of wain is int*
    $2 = length of the array
      //ie, 2nd parameter of wain
  } else {
    $2 = 0
  }
/* the second parameter is the offset on
where to start the heap (mips.array puts
the array at the end of the code) */
```

- new/delete should be called as functions, like with print
- new
  - $1 = number of words needed
  - return pointer to memory in $3
  - returns 0 when allocation is not possible
    - if it comes back with 0 in $3, change it to 1

```
code(new int[expr]) = code(expr) + ($3 ← expr)
                      add $1, $3, $0 +
                      call(new) +
                      if $3 == 0, $3 = 1
```

- delete

$1 = pointer to memory to be deallocated

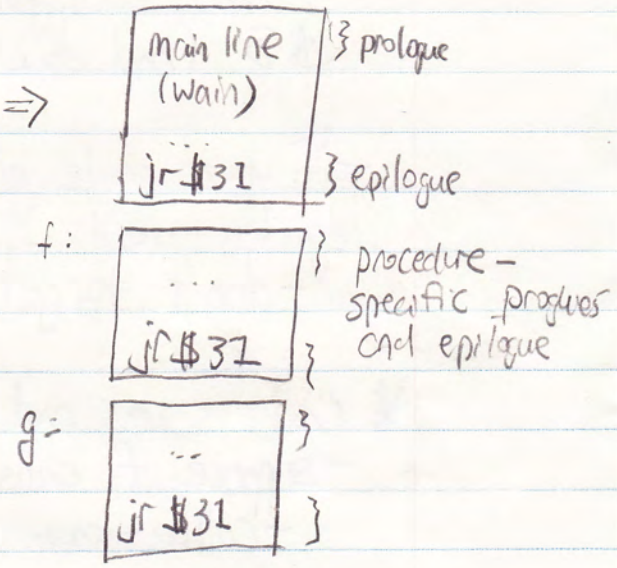
```
code(delete [] expr) = code(expr) + ($3 ← expr)
                       beq $3, $11, skipX +
                       add $1, $3, $0 +
                       call(delete) +
                       skipX:
```

* Compiling Procedures

- big picture

```
int f(...) {...}
int g(...) {...}
int wain(..., ...) {...}
```

⇒



- main prologue/epilogue
  - save $1, $2 on the stack
  - import print, init, new, delete
  - set $4, $11, etc, if desired
  - set $29
  - call init
  - reset stack at end of jr $31

- procedure-specific prologues
  - don't need to import, set constants, etc
  - save registers
  - set $29

- restore registers and reset stack at the end + jr $31

*Saving and Restoring Registers
- procedures should save and restore all registers it will modify
- how do you know which registers?
  - if we're not sure, save them all (except for $3)
  - our codegen scheme doesn't use registers past $8 (and $29 to $31)

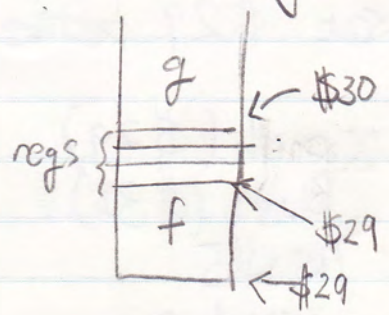
- if your code gen uses more, then keep track of the registers used
  - don't forget to save and restore $29

* Caller-save and Callee-save
- suppose f calls g:
  - callee-save: g saves all register that it modifies
    f doesn't have to worry about what g does
  - caller-save: f saves all registers that contain critical data before calling g
    g doesn't worry about f's registers

- our approach has been: caller-save for $29
  callee-save for everything else

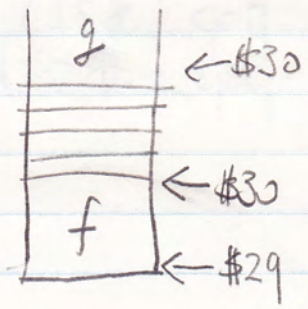
- different approaches also work, but what should we do?

- who should save $29?
  - prologue: save registers, set $29
    - which should you do first?
      - save registers first

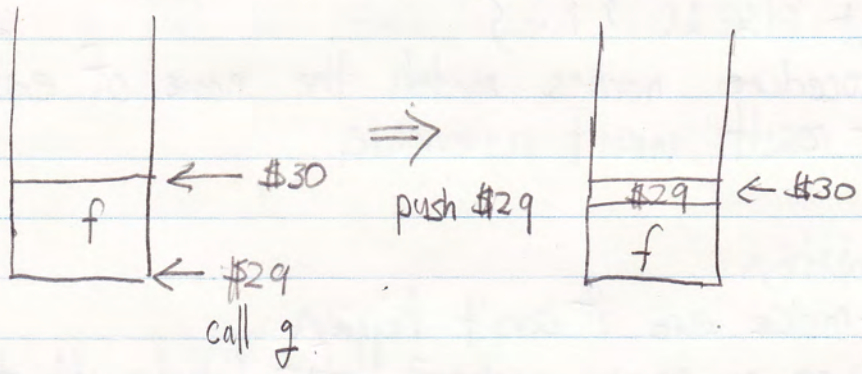


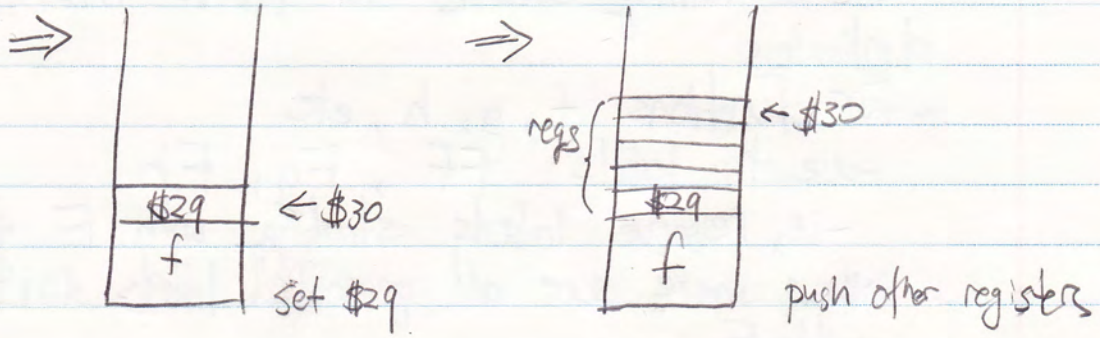
- $30 has already changed
- need to figure out where $29 should point, based on the number of registers

- set $29 first



- $30 hasn't changed yet
- easy to set $29 to $30 - 4 and then save registers
  - if you set $29 first, and then save registers, how do you save $29?
- need to save $29 before you change it





- so push $29
  add $29, $30, $0
  push other registers

- or just let the caller save $29 before the procedure call

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

- a lot cleaner

*Labels

```
int init(...) {...}
int print(...) {...}
int else1(...) {...}
```

- procedure names match the name of existing labels
  - result won't assemble

- solution
  - make sure it can't happen
  - use a naming scheme for labels that prevents duplicates
  - for functions f, g, h, etc
    - use the labels Ff, Fg, Fh
      - ie, reserve labels starting with F for this, then make sure all generated labels don't start with F

*Parameters

- could use registers
  - may not be enough
- push parameters onto the stack

factor → ID(expr$_1$.. expr$_n$)

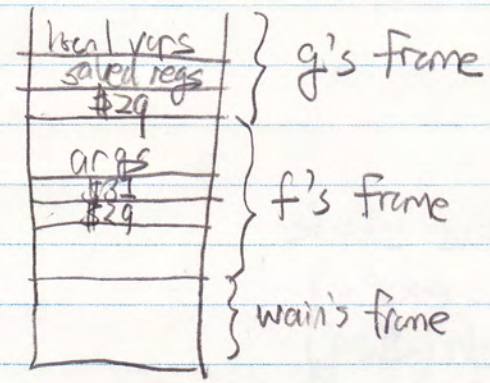
```
code(factor) = push($29)
               push($31)
               code(expr1)
               push($3)
               ...
               code(exprn)
               push($3)
               lis $5
               .word F
               pop all arguments
               pop($31)
               pop($29)
```

procedure → INT ID (params) { dcls statement return expr }

```
code(procedure) = sub $29, $30, $4 +
                  push regs +
                  code(dcls) +
                  code(statements) +
                  code(expr) +
                  pop registers +
                  add $30, $29, $4 +
                  jr $31
```

- what does the stack look like?

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



- suppose g is

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

g's symbol table

	Name	Type	offset
params	a	int	0
	b	int	-4
	c	int	-8
saved regs			
locals	d	int	-12
	e	int	-16
	f	int	-20

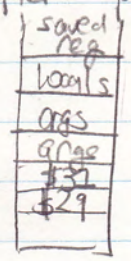
- params a, b, c are below $29
  - local variables d, e, f are above $29
  - in between are saved registers

- so the offsets in the symbol table are wrong
  - params should have positive offsets
  - local variables should have negative offsets

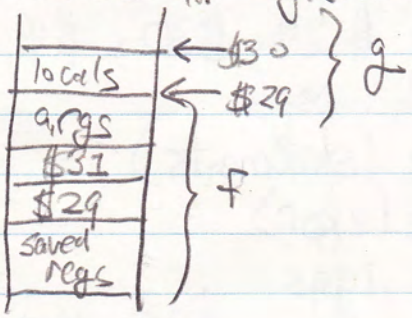
- need to fix the symbol table before we generate code
  - add 4 times the number of arguments to all offsets in the symbol table
    4 * number of args) = 4 * 3 = 12
  - we have a symbol table that looks like:

	Name	Offset (from $29)
Params	a	12
	b	8
	c	4
locals	d	0
	e	-4
	f	-8

- saved registers still between args and local
- save local registers after pushing local variables



- or save all registers in the caller



- caller-save saves registers once per call
- callee-save saves registers per function

- ie,

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

```
g() {
    ...
}
```

callee-save save and restore code goes here, once

caller-save saves and restores around every call to g

- summary

factor → ID (expr1 .. exprn)

```
code(factor) = push($29)
               push($31)
               code(expr1)
               push($3)
```

```
code(expr_n)
push($3)
lis $5
.word FID
jalr $5
pop all arguments
pop $31
pop $29
```

```
code(procedure) = sub $29, $30, $4
                  push regs
                  code(statements)
                  code(expr)
                  pop regs
                  add $30, $29, $4
                  jr $31
```

* Optimization
- very large and difficult topic
  - minimize run-time (or for CS241 code size)
- unsolvable
  - we apply heuristics

- example
  - code for 1+2

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

9 instructions

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

```
lw $5, 0($30)
add $30, $30, $4
add $3, $3, $5
```

- called constant Folding

*Constant Propagation

- example

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

} - recognize that x = 1 and perform the following:

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

- if that is the only place where x is used, it doesn't need a stack entry

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

- watch out for aliasing

```
int* y;
y = &x;
*y = ...;
```

Andy Yin
Stefan Buettcher
CS241
Lecture
Mar 31, 2015

*Constant Propagation

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

} - generated as 13 instructions

-we could recognize x=1 and do
-if x is only used here, don't store it:

```
lis $3
.word 2
jr $31
```

```
lis $3
.word 1
sw $3, -4($3)
sub $30, $30, $4
lis $3
.word 2
jr $31
```

-sometimes might not know x's value can recognize $3 holds the value of x already:

```
lw $3, -12($29)
; ↑assume x is here
add $3, $3, $3
jr $31
```

*Common Subexpression Elimination

(a+b) * (a+b)

-use a register to hold a+b, then multiply it by itself

-stops us from computing a+b twice

*Dead Code Elimination

-if we know for certain that some branch of a program will never be executed, do not produce code for it

```
if (true) {
    ....
} else {
    ....
}
```

*Register Allocation
- register faster than RAM (less lw/sw),
  - eg, $14 - $28 are unused in our code generation scheme
  - could use these to hold 15 variables
    - should use the most frequently used

- issue
  & - address of
  - if we want to take the address of a variable, it needs to be on the stack

*Strength Reduction
- add usually runs faster than mult
  - usually true in the real world, prefer adds over mults
- multiply by 2:

```
li $2
mult $3, $2
mflo $3
```

} add $3, $3, $3

*Procedure specific Optimizations
- inlining:

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

```
    return a+a;
}
```

- replace the function call with the function body in the calling code/caller
  - saves over head of a function call
- is it always a win?
  - it duplicates f's body many times if we call f many times, if f's body is very big, it doesn't save many instructions
  - some functions are harder to inline than others (ie, recursion)
    - if all calls are inlined, then we can remove f

*Tail Recursion

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

↖ the last thing a function does is a recursive call function

- no pending work to do when the call to f returns

- contents of the current stack frame (local variables, arguments) will not be used again
  - can reuse the same stack frame

- Can we do this in WLP4?
  - no, returns must only be at the end of a procedure
    - there's no return in the if statement

- but we could expand the language after parsing by mutating the parse tree

*Basic Transformation

- whenever we return immediately following an if statement, push it inside both branches

```
int f(...) {                      int f(...) {
  int ret = ...;                    int ret = ...;
  if (...) {                        if (...) {
    ...                               ...
  } else {                            return ret;
    ...                           } else {
  }                                 ...
  return ret;                       return ret;
}                                 }
                                }
```

- whenever a return follows an assignment to ret, merge them
  ret = f(...) ⇒ return ret -> return f(...)

- generalization: tail call optimization when a function's last action is any function call (recursive or not), then we can reuse the stack frame

*Memory Management

- WLP4, C, C++: user must do explicit memory management (new/delete, malloc/free)
- Java, Racket, Python: implicit memory management via garbage collection

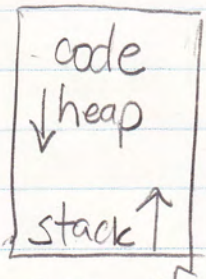
- Scope vs extent

- a variable's scope is the part of the program in which the variable's name is visible
  - scope is static (stack-based), and can be determined at compile time

- if x is declared in f, then x's scope is entirely within f

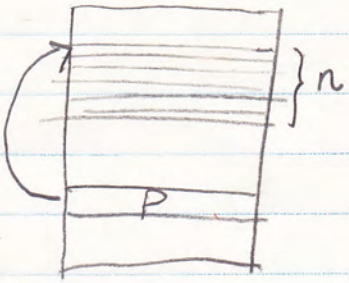
- a variable's extent is the part of the program's execution in which the contents of a variable's memory in live
  - extent is a property of the underlying data
  - for stack-allocated data, extent ends with scope

- sometimes, we want data to "live" past the end of a function
  - use the heap:



typical RAM layout

- the Heap is an area of RAM set aside for dynamic allocation: p = new C[n]
  - allocates space for n C objects on the heap, stack contains pointer to the beginning



- to release heap data (end its extent)

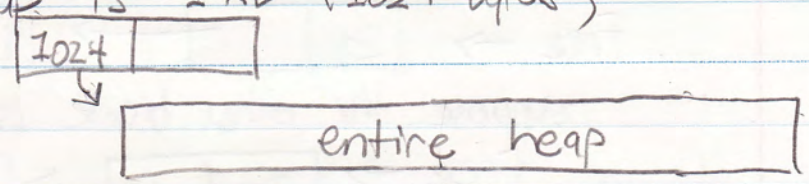
```
delete d;      // non-array
delete [] p;   // array
```

- how does new/delete, malloc/free work?

Andy Yin
Stefan Buettcher
CS 241
Lecture - More Optimizations and Memory Management
Apr 2, 2015

*Memory Management
- how do we implement new/delete and malloc/free?
  - there are many implementations.

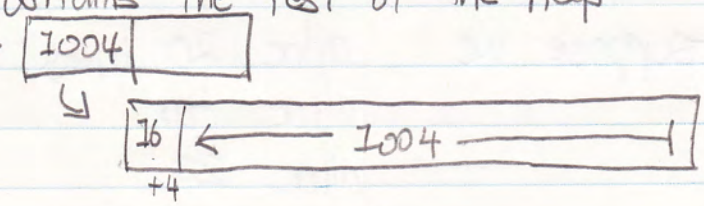
- free lists
  - maintain a linked list of pointers in the heap of free memory
  - initially, free list has one entry
    - suppose heap is 1 KB (1024 bytes)



  - suppose 16 bytes are allocated
    - actually allocate 20 bytes
      → 16 bytes + 1 word (int)
      allocated bytes — book keeping

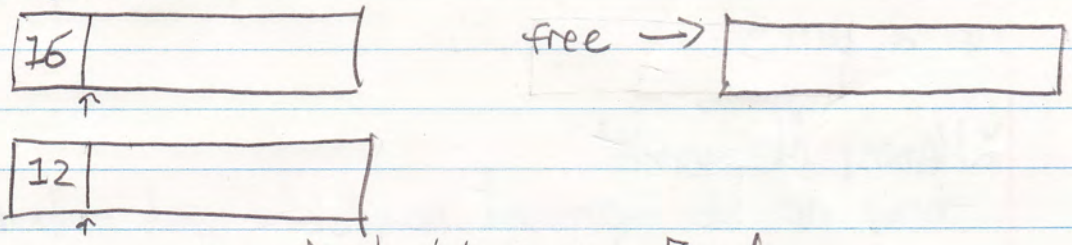
↑ return a pointer to the word just after size

  - free list contains the rest of the heap

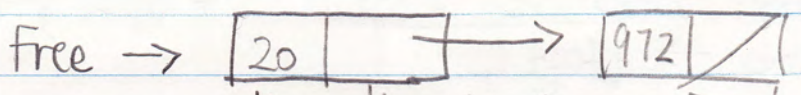


- now suppose we require 28 bytes
  - actually 32 bytes (28 + 4 for bookmarking)

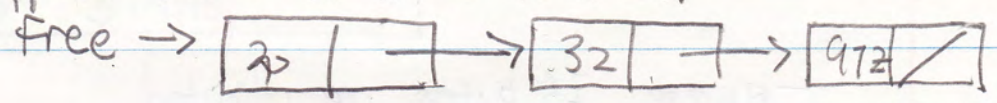
- allocated



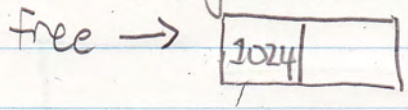
- suppose the first block is freed
  - eg, free(p)
  - free/malloc is going to check p[-1] for size allocated
    - we know how much memory to reclaim
  - so add this memory to the free list



  - suppose the other block is freed



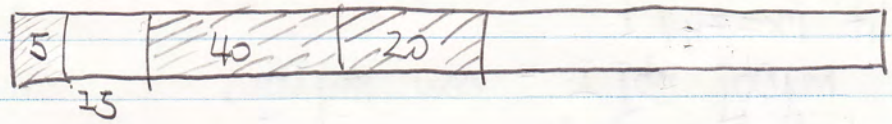
- but we can notice that the blocks are all adjacent in memory
  - we can merge them



  - note, repeated allocations and deallocations can create holes in memory

- suppose we: alloc 20 bytes
  alloc 40
  alloc 20

dealloc 20
alloc 5



- fragmentation
  - even if n bytes of memory free might not be able to allocate a block of n byte
  - to reduce fragmentation, don't choose the first block of memory that fits the request

- suppose we find

- new allocation: alloc 10
  - we do we put it?
    - best fit: put it in the 15 byte block
    - first fit: put it in the 20 byte block
  - issue: searching RAM/heap takes time
    - memory allocation isn't free

*Implicit Memory Management
- garbage collection (Java, Racket, Python)
  - runtime environment reclaims memory when it is no longer accessible

```
int f(){
  myObj obj = new myObj;
  ...
}
```

↖ obj goes out of scope, memory is reclaimed, since no longer accessible

```
int g() {
  myObj obj2;
  if (x==y) {
    myObj obj1 = new myObj;
    obj2 = obj1;
  }
```

↖ obj1 goes out of scope, not reclaimed since obj2 holds the memory

} ↖ obj2 goes out of scope, no longer accessible ⇒ GC reclaims

* Garbage Collection Techniques

(1) Mark and Sweep

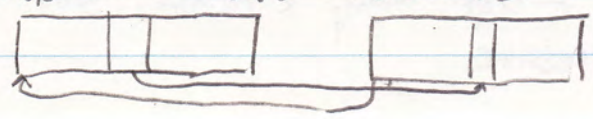
- scan the program/heap for pointers
  → For each pointer found, mark the heap block it points to
    - if the block contains pointers, then repeat

- once done, scan entire heap and reclaim unmarked memory, then clear marks

(2) Reference Counting

- for each heap block, track the number of pointers that point to it (called reference count)
- if block's reference count reaches 0, then we can reclaim it
  - so we must monitor every pointer and update reference counts (decrement old, increment new) when pointers are reassigned

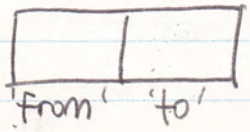
- problem: circular references



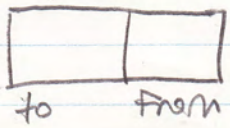
-both have a reference cant of 1, but are otherwise inaccessible

(3) Copying Garbage Collection

- heap is split into halves



  - only allocate from 'from'
  - when from is full, copy all reachable data
    - copy that data from 'from' to 'to', then reverse roles: 'from' becomes 'to'
      'to' becomes 'from'



- benefit: built-in compaction
  - guaranteed to that after the swap, all reachable data is in contiguous memory
    - there is little to no fragmentation

- problem: only half the heap is available at any one time