CS 241 Notes : Foundations of Sequential Programming

Johnew Zhang

April 7, 2012

Contents

1 Lecture 1

1.1 Bits : Binary Digits

A bit is anything that has two states. For example, the switch for light has two state up/down and for the light, it is on/off. We can represent those states as 0/1 or $-/|$.

Single bit cannot do anything so we usually put them into group of k bits.

Consider $k = 2$ and up/down and red/black. Then we can find 4 combinations of these. It is not difficult to find out that for k bits, there are 2^k combinations.

Little ending	big ending		name
0	0	1	fred
2	1	2	barney
1	2	3	wilmort
3	3	4	betty

Above are called unsigned integer representation.

1.2 Two's complement

Negatives are represented as unsigned 2^k greater. For example $-2 = -2 + 2^2 = 2$. Positive are remain the same.

$k=4$	unsigned	2's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
...	...	...
1110	14	-2
1111	15	-1

Computer Arithmetic

In the computer, we type $7+3$ (4 bit arithmetic). Well, in this case, all the calculations are done by modulo 2^k. For $7-6$, it is equivalent to $7+(-6)=7+10=1$.

$$\begin{array}{r} 0111(7) \\ +0011(3) \\ \hline 1010 \end{array} \tag{1}$$

$$\begin{array}{r} 0111(7) \\ +1010(10) \\ \hline 0001 \end{array} \tag{2}$$

Hexadecimal

$0,1,2,\cdots,9,A,B,C,D,E,F$ are numbers to represent it.

1.2.1 Bites

Byte is a group of 8 bits ($k=8$, 256 combinations). In this case, unsigned is between 0 and 255; signed is between -128 and 127; ASCII codes are characters corresponding to numbers between 0 and 127.

bytes	meaning represents unsigned	2's complement	Hex	ASCII
01100001	97	97	61	a

UNICODE There are 1 million symbols in UNICODE.

UTF-8 Sequence of bytes for each character

Extended ASCII

Word 4 bytes (32 bits); unsigned is between 0 and $2^{32}-1$; 2's complement is between -2^{31} and $2^{31}-1$. and 8 bytes (64 bits)

2 Lecture 2

2.1 Stored Program Computer

CS 241 MIPS is a kind of "Von Neumann Architecture" and it has 32-bit instructions. It is Reduced Instruction Set Computer (RISC).

2.2 The operations between CPU and RAM

A RAM is sequence of bits and organized into bytes/words. For a 32-bit machine, 4 bytes/words and each byte has an address and each word has the address of such byte.

CPU : Centered Processing Unit

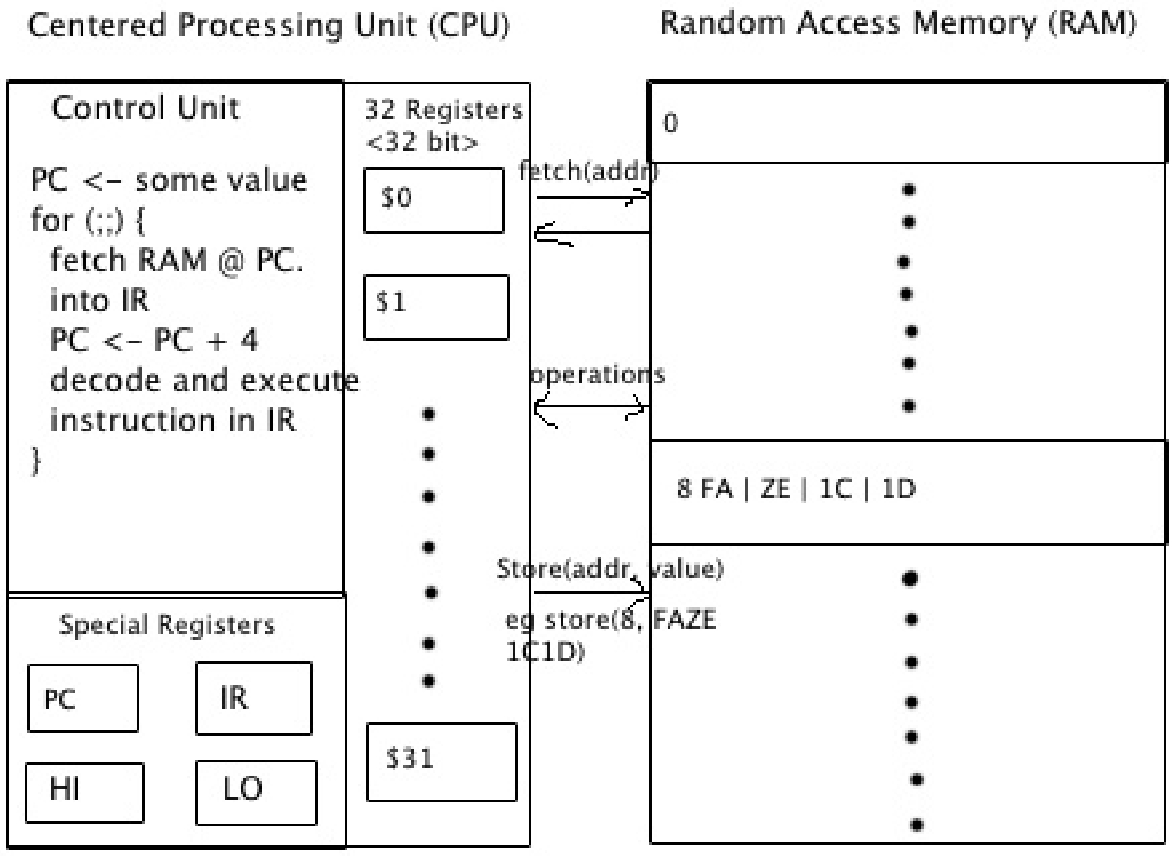
RAM : Random Address Memory

We can have operations between CPU and RAM. fetch(addr) and store (addr, value). For example, store(8, FAZE1C1D).

CPU has registers (from $0 to $31, 32-bit wide), special register (PC(Program Counter), IR (Instruction Register), HI, and LO) and Control Unit.

In Control Unit, it assigns some value to PC and runs through a infinite loop. Inside the loop, it fetches RAM @ PC into IR; $PC \leftarrow PC + 4$. decode and execute instruction in IR.

```
PC = somevalue;
for (;;) {
            fetch RAM @ PC into IR;
            PC = PC +4;
            decode and execute instruction in IR;
}
```



Explanation of MIPS reference sheet ;

1. the first 6 0s means it is a simple instruction
2. the five s and five t and ect. means a register
3. $ 0 is always equal to 0

Here is one example for jr31.hex :

```
; 0000 0011 1110 0000 0000 0000 1000
; 0 3 e 0 0 0 8
.word 0x03e0008
```

3 Lecture 3

3.1 MIPS Programming

Three human readable :

1. binary (ASCII 0's and 1's)
2. hexdecimal text
3. assembly language

C	MIPS
int x,y,z;	\$2 is x, \$7 is y and \$21 is z
unsigned a, b, c;	\$4 is a, \$8 is b and \$19 is c
float w;	no direct MIPS implementation
$x = y + z$;	add \$2 \$7 \$21
$a = b + c$;	add \$4 \$8 \$19
$y = z$;	add \$7 \$0 \$21
$a = b * c$;	multu \$8 \$19, mflo$*$ \$4
$x = 42$;	lis \$42, .word 42
if(t) α;	translate(α)
if $(t)x = y + z$;	$*$
if (t) α; else β	$**$
while (t) α;	$***$
for $(\alpha;\beta;\gamma)\delta$;	α while (β) { $\delta;\gamma$}
do { α } while (t)	$****$

Notes : Hi and Lo represent 64-bit two's compliment

$*$ suppose t is \$10, \$10=0, false; \$10 =1, true. Then beq \$10 \$0 1; add \$2 \$7 \$21. Another example bees \$10 \$0 nnn, translate (α) where nnn is number of instructions in translate(α)

$**$ beq \$10, \$0, nnn, translate(α), beq \$0, \$10, mmm. translate(β), where nnn is 1 plus the number of instructions in translate(α) and mmm is the number of instruction in translate(β)

$***$ translate(t), beq \$10 \$0 nnn; translate(α); beq \$0 \$0 mmm, where nnn is the number of instructions in translate(α) plus 1 and -mmm is the number of instructions in translate(t) plus the number of instructions in translate(α) plus 2

$****$ translate(α); translate(t); ben \$10, \$0, mmm; where -mmm is the number of instructions in translate (α) plus the number instruction in translate(t) plus 1.

3.1.1 Examples

add value in register 5 to value in req 7, and store result in req 3

```
; 00000000101001110001100000100000
; 00A71820
add $3 $5 $7

; 00000011111000000000000000001000
; 03E00008
jr $31
```

add 42 to 57, store result in $3

```
; 00000000000000000001000000010100
; 00001014
lis $2

; 00000000000000000000000000101010
; 0000002A
.word 42

; 00000000000000000001100000010100
; 00001814
lis $3

; 00000000000000000000000000110100
; 00000034
.word 52

; 00000000010000110001100000100000
; 00431820
add $3 $2 $3

; 00000011111000000000000000001000
; 03E00008
jr $ 31
```

Change value in register 1 to its absolute value e.g. $1 = 20 unchanged; $1 = 2, change to 20

If $1 =0, $1 = - $1

```
; 00000010101000010000000000101010
; 02A1002A
slt $21 $1 $0

; 00010010101000000000000000000001
; 12A00001
beq $21 $0 1

; 00000000000000010000100000100010
; 00010822
sub $1 $0 $1

; 00000011111000000000000000001000
; 03E00008
jr $31
```

Suppose int a,b,c, $\cdots$, z, A, B, C, $\cdots$, Z, 52 variables.

Use RAM instead of registers, pick a RAM address that you are not using, say 100000, 100012, 66664, 996.

Suppose x is 100000, y is 100012, and z is 100020.

```
x = y +z;
 lis $7
.word 100012; addr of y
lw $3, 0($7) ;  contents of y
lis $7
.word 100020 ; addr of z
lw $15, 0($7) ; contents of z
add $7, $3, $15  ; y+z
lis $6
.word 100000; adds of x
sw $7,0($6)
```

How to get memory?

for example,

```
 jr $31
.word .... ; x
.word .... ; y
.word .... ; z
```

Well we can put label for .word, e.g. lis $7; .word x

We can do the other way "stack".

4 Lecture 4

Program 1

```
; sum the integers from 1 to N
; input : $1 is N
;output : $3 is the sum
; temp : $2
          add $3, $0, $0 ; zero accumulator
; beginning of loop
          add $3, $3, $1 ; add $1 to $3
          lis $2 ; decrement $2
          .word -1
          add $1, $1, $2
          bne $1, $0, -5 ;branch to beginning of loop (if not done)
          jr $31 ; return
```

Like this we going to create a program first :

```
R = 0;
do {
          R = R + N;
          N = N-1;
} while (N != 0)
return R;
```

Program 2

```
; sum the integers from 1 to N
; input : $1 is N
;output : $3 is the sum
; temp : $2
          add $3, $0, $0 ; zero accumulator
beginLoop: : (label)
          add $3, $3, $1 ; add $1 to $3
          lis $2 ; decrement $2
          .word -1
          add $1, $1, $2
          bne $1, $0,beginLoop ;branch to beginning of loop (if not done)
          jr $31 ; return
```

Program 3

```
; sum the integers from 1 to N
; input : $1 is N
;output : $3 is the sum
; all register except $3 must have initial value

; we will change $1 and $2. Save and restore them.
; Need a label so we can call the procedure.

sumOneToN;
        sw $1, -4($30) ; save $1 on the stack
        sw $2, -8($30) ; save $2 on the stack
        lis $2 ; reset the stack pointer
        .word 8
        sub $30, $30, $2

        add $3, $0, $0 ;zero accumulator

beginLoop:
        add $3, $3, $1 ; add $1 to $3
        lis $2  ; decrement $2
        .word -1
        add $1, $1, $2
        bne $1, $0, beginLoop

        lis $2 ; reset the stack pointer
        .word 8
        add $30, $30, $2
        lw $1, -4($30) ; restore $1 from stack
        lw $2, -8($30) ; restore $2 from stack

        jr $31 ; return from sumOneToN
```

Program 4 : Calling procedure

```
sw $31, -4($30) ; save $31 on stack
lis $31
.word 4
sub $30, $30, $31; call sumOnetoN(13)

lis $1
```

```
.word 13
lis $4
.word sumOneToN
jalr $4 ; put return address in $31
            ; replace PC with contents of $1
lis $31 ; restore $31 from stack
.word 4
add $30, $30, $31
lw $ 31, -4($30)

jr $31 ; return to OS
```

Then you can add the program for sumOneToN to the program above.

Program 5

```
; store $1 in x
; store $2 in y
; i.e. story 42 in $1, $5 in $2
        lis $3
        .word x
        lw $1, 0($3)

        lis $4
        .word y
        lw $2, 0($4)

        add $5, $1, $2
        sw $5, 0($3) ;; x = x +y

        jr $31

x : .word 42
y : .word 55
```

4.1 Input/output

MIPS can read a byte or write a byte to stdin/ stout, i.e. getchar(), putchar().

Program 6

```
; cat
```

```
lis $1
.word 0xffffooo4
lis $3
.word -1

loop :
lw $2, 0($1)
beq $2, $3, quit
sw $2, 8($1)
beq $0, $0, loop
quit
jr $31
```

4.2 Array

Program 7 : Array

```
; read 2nd element of array

lw $3, 4($5)
jr $31

; read 2nd element of array
; input : $1 is the array, $2 is length
; output : $3 contains last element of array

add $5, $2, $2 ;; double $2
add $5 $5, $5 ;; double $5
add $5, $5, $1 ;; add array address
lw $3, -4($5)
jr $31
```

5 Lecture 5

An assembler file is a text file. An assembler reads the source file (MIPS assembly language) and it outputs an object file (MIPS binary machine code) or error report. Well an assembler is just a translator from MIPS assembler language to binary.

Source file example

```
0 ; hello
0 lis $4; load 16
4 .word fortytwo
```

```
8 lw $3, 0($4)
c jr $31
10 fortytwo ; .word 42
```

Object file example(as listed by xxd)

```
0000 : 0000 2014 0000 0010 8c83 0000 019 0008
0010 : 0000 0029
0020 :
0030 :
```

The value of fortytwo is 16 (0x10).

5.1 How to write an assembler?

1. understand the source language
2. understand object language
3. understand the meaning/correspondence between source and object
4. write an assembler
   - divide into components
   - specialized techniques
   - test

A unix file is a sequence of lines with ascii characters and ended by newline (10)

The i is the value of a label minus (4 plus the location of instruction) and then divided by 4.

Well in the assembler, there are two parts : analysis and synthesis. Analysis means to take them into parts. Then synthesis means to put it together. First, to analyze if it is a valid program and then transform into data structures. In the transformation to data structures, we have three parts :a. split program into lines (get line,readline); b, lexical analysis (scanning);c. context-free analysis (parsing); d)context-sensitive analysis (semantic analysis) Secondly we are going to create the MIPS binary from the result of analysis. In the middle, there is an intermediate representation.

5.2 Analysis

split program into lines

lexical analysis Example, translate fred: dick: lw 0($30) ; hello into a sequence of lexical units (tokens). You will be provided a scanner. It is a sequence of pairs < kind, lexeme>. In c++, it is vector<Token> and in scheme it is list.

Context-free analysis

Context-sensitive analysis

6 Lecture 6

6.1 How to write an assembler?

2 pass Assembler :

1. Pass 1 check program for validity; build sumbol table; build intermediate representation of instruction
2. Pass 2 : encode instructions/oprands as MIPS instruction words; output MIPS instructions.

6.1.1 Pass 1

Location is a number,symbol table is symbol.

For every input line, scan line into sequence of tokens (discount comments). for every label definition at start of sequence of tokens. Then we have two choices :

1. Choice 1 : check that there is no <label, value > in symbol table where label is the lexical. if already there : ERROR. Add < label, location > to symbol table. If there is an opecode (following the labels) depending on opecode, determine whether or not the remaining tokens are valid. For example add $1,$2,$5 has five tokens. reg, comma, reg, comma,reg.

   location is location plus 4

   Finally, if we use choice 2, we will check duplicate label in the symbol table.

   For the intermediate representation :

   (a) record source file [not possible in marmoset]
   (b) list/vector of source lines : resort everything in pass;
   (c) only non-null lines;
   (d) only opcodes/operands (of non-null line)
2. Choice 2 : don't check

6.1.2 Pass 2

For every non-null line of input depending on the opecode, encode the opeode and operands into MIPS instruction word and output instruction.

Source

```
location                                                  intermediate
0 -->            ; hello
0 -->           add $1, $2, $3                   add $1, $2, $3
4 -->           foo: zip: ;hi again
4 -->           beq $1, $2, bar                  beq $1, $2, bar
8 -->           bar : .word foo                   .word foo

Symbol Table
/-------------------\
| foo     |   4   |
----------------------
|   zip     |  4    |
----------------------
|   bar    |  8    |
\-------------------/
```

C program to print out bytes

```
// encode add $d, $s, $t  and output it
int s = 2;
int t = 3;
int d = 1;

// 0000 00ss ssst tttt dddd d000 0010 0000
//template for add with 0s for s, t, d

int addintstr = 0x00000020;

// set the sssss stuff to s
int MIPSinstr = addinstr + ( s << 21) + (t << 16) + (d << 11);

// output
printchar(MIPSinstr >> 8); the second last bite
printchar(MIPSinstr); // output the last bite of MIPSinstr
```

Scheme program to print out bytes

```
(define d 1)
(define s 2)
(define t 3)

(define addinstr #0x00000020)

(define MIPSinstr (+ addinstr (arithmetic-shift s 21)
(arithmetic-shift t 16)
(arithmetic-shift t 11)))

(write-byte (bitwise-and (arithmetic-shift MIPSinstr -24) 0xff))
(write-byte (bitwise-and (arithmetic-shift MIPSinstr -16) 0xff))
(write-byte (bitwise-and (arithmetic-shift MIPSinstr -8) 0xff))
(write-byte (bitwise-and MIPSinstr 0xff))
```

7 Lecture 7

MIPS binary file load in the RAM and in the RAM, there is a loader which is loaded by an initial loader. Also in the ROM there is a mini-loader to trigger the initial loader.

According to the example in the class, we can calculate the following :

$$\text{offset } ii = \frac{target(B) - current - 4}{4}$$

Sticky note is two words and has 8 bytes long. The length of the header is 8.

7.1 MERL : MIPS Executable Relocatable Linkable Format

It is formed by three parts : header(cookie, length of the MIPS code, length of the MIPS code plus the length of code in the header), MIPS program and sticky notes.

8 Lecture 8 : Programs that operate on MERL files

```
        MERL
+========+
| header |                              RAM
==========                        +========+
|MIPS    | ========>              |        |
==========                        +========+
|Sticky  |
| notes  |
+========+
```

There is a link file to link them.

8.1 Assembler

java cs241.linkasm consumes a MIPS file and translates it into a MERL file. The difference between CS241.binasm and CS241.linkasm is you don't have to modify the MIPS file.

8.1.1 Modifying Assembler to Produce MERL

1. Pass 1 :

   Relocation list :

   Location counter : 0xc

   For each line of code, we do the normal stuff but if we have .word X, where X is a label, then we need to append location counter to reallocation list.

2. Pass 2 :

   output cookie 0x10000002

   output *length* = location counter + 8 × length of relocation list

   output *codelength* = location counter

   output MIPS program in usual way

   for each location in relocation list

   output 1

   output location from list

```
a.asm :

.import proc (please fill in proc at location at 0x10)
lis $1
.word proc
jalr $1

b.asm :

.export proc (proc is at location 0xc)
proc:
jr $31
```

We put these two files into cat and get ab.asm and then put the assembler and get a MIPS file. If the assembler is a linker then we get a MERL file.

Let's assemble a.asm to a.merl and b.asm to b.merl. Hence we put them through the linker and then get MERL.

8.2 Loader (Relocating) Implementation

- determine α (load the address)
- Copy MERL to RAM starting at address α.
- let Sticky note $= \alpha + \text{RAM}[\alpha + 8]$ (*) if $RAM[SN] = 1$ (add α sticky note),
  RAM[Address to reallocate] $=$ Location to relocate $+ \alpha = RAM[SN + 4] + \alpha$
  else error
  $SN \leftarrow SN + 8$
  repeat (*) if $SN < RAM[\alpha + 4] + \alpha =$ merl length $+ \alpha =$ end of MERL.

8.3 Linker

From 8.1.1, we get a.merl (h1, m1, s1) and b.merl (h2, m2, s2). These two files go through the linker we get a new merl file, ab.merl (h12, m1 m2, s1, s2). Well, the location of m2 is the code length of h1. Then we need to reallocate m2 from c plus code length of h1. Since we moved m2, we also need to move the sticky notes.

External Symbol Table

We put proc in the table and 0x18 in the loc.

9 Lecture 9 : Formal Language (in contrast to ”Natural Language”)

A formal language L is a set of strings from a given alphabet, Σ.

Σ is a finite set of symbols, e.g. $\{a, b, c, \cdots, z\}$ or $\{red, green, blue\}$ or $\{1, 2, 3, 4, 5, 6\}$ or $\{0, 1\}$ or ASCII.

A string x is a finite sequence of symbols from Σ, e.g. abc, 12113, red green ..., 010111010101.

Languages $\Sigma = \{0, 1\}$ $L = \{01, 10, 11, 111\}$.

Each element often called a ”word” or ”sentence”.

ϵ denotes empty string where $\epsilon \notin \Sigma$.

Noam Chonsky tried to describe English and other natural languages using math.

9.1 Binary Integers (no leading zeros)

$\Sigma = \{0, 1\}$

$L = \{0, 1, 10, 11, 100, 101, 110, \cdots\}$.

$L = \{$all valid MIPS assembly programs$\} = \{$All MIPS binary programs that terminate when executed$\}$

This is the halting programs with regard to the undecidability.

9.2 Applications of Formal Language

1. Specify things a communication aid e.g. to program in a programming language to implement programming language
2. Recognize language - recognizer take x as input answers. Is $x \in L$?
3. parsing - parser proves that $x \in L$ such that i.e. constructs a derivation
4. translation - given $x \in L$, denote $y \in L'$ such that y corresponds to x.

Chonsky (1955) hierarchy :

1. Type 0 : Anything you write a computer program in recognize.
2. Type 1: Context-sensitive
3. Type 2: Context-free
4. Regular Language

9.3 Regular Languages

Two definitions :

1. Generative (using set) :
   - finite alphabet Σ,
   - any finite set of strings from Σ is regular. e.g $\{\}$ or $\{\epsilon\}$;
   - union of two regular language is regular;
   - concatenation of two regular language is regular (concatenation of 2 strings $z = xy$ is just the sequence consisting of the symbols in x followed by symbols in y) (concatenation of languages $L_1L_2 = \{xy | x \in L_1, y \in L_2\}$);
   - repetition of two languages is regular. ($L^* = \{\epsilon\} \cup L^*L$)
2. Alternative Definition : alphabet Σ, $L = \{$any string x that can be recognized by a finite state machine (FSM) aka finite automaton$\}$

   A finite automaton is a model of computation.

Finite Automaton

Σ, finite set of states. labels transitions between states; bubble diagrams; set of final states.

Deterministic Finite State Automaton (DFA)

A finite automaton with determinism restriction : for any given state and symbol, at most one transitions. from the state may be labeled with the symbol.

Non-deterministic Finite Automaton

No determinism restriction.

10 Lecture 10 : DFA

$DFA = (\Sigma, S, T, \text{start}, \text{finish})$

e.g. $\Sigma = \{0, 1\}, L =$ binary integers with no useless-zero.

$S = \{\text{fred}, \text{binary}, \text{wilma}\}$

start = barney

finish = $\{\text{fred}, \text{wilma}\}$

T

S	Σ	fred
barney	1	wilma
wilma	0	wilma
wilma	1	wilma
fred	0	err
fred	1	err
err	1	err
err	0	err

Gotcha : T is a partial function to make it total add new state ”err”.

10.1 Program

Σ is the number/name symbols

S is the number/name of states

T is the array $T[state, symbol]$ or a function int T(int, state, int symbol)

start is just a state (number/name)

finish is set (list, array, finish [state] = 1 if state $\in$ finish; otherwise 0.

10.2 DFA

Recognizer

input : DFA, string

$x = a_1, a_2, \cdots, a_n$ where $\forall i, a_i \in \Sigma$.

The algorithm

```
state <- start
for : from 1 to n
state <- T[state, ai]
if finish[state], accept, (x in L)
else reject (x not in L)
```

NFA is the table we list the relationships. Well, it is not a function.

There is a function returning a set

barney	1	{ fred, wilma}
barney	0	{fred }
wilma	0	{wilma}
⋮	⋮	⋮

10.3 NFA

Set up is almost the same as the DFA.

The algorithm

```
state <- start
for : from 1 to n
state <- The union of all the state in S, T[S, ai]
if the intersection of finish and states is not empty, accept, (x in L)
else reject (x not in L)
```

Examples of DFAs

$\Sigma = \{0, 1\}$

$L =$ even binary integers (ends in 0.

$$\Sigma = \{0, 1\}$$

L is divisible by 4

11 Lecture 11 : Using Finite Automata for Translation, Searching & Scanning

11.1 Finite Translation - Finite Automaton with Output

$$\sum_{in} = \text{input alphabet}$$

$$\sum_{out} = \text{output alphabet states}$$

$$T : \Sigma \times S \to S$$

$$O_{\text{State finish}} : \Sigma \times S \to \Sigma_{out}^*$$

For example,

$$\Sigma_{in} = \{a, b, c\}$$

$$\Sigma_{out} = \{A, B, C\}$$

$$L = \{cab, bac\}$$

Translate $cab \to CAB$, $bac \to BAC$.

11.2 Moore Machine

$$O : S \to \Sigma_{out}^*$$

Sometimes Language is unimportant

```
state <- state
fro i in 1 to n
output <- output O[state,ai]
state <- T[state,ai]
if state in final accept
return output
```

As an example, binary integers $\Sigma_{in} = \{0, 1\}$ translate to $\Sigma_{out} = \{-, |\}$

Remove "stutters" - repeated consecutive symbols

$$0 \to 0$$

$$00 \to 0$$

$$11 \to 1$$

$$011000111 \to 0101$$

where $L = \Sigma^*$

$$\Sigma_{in} = \Sigma_{out} = \{0, 1\}$$

11.3 Implementing ϵ transition in NFA

Algorithm for $\epsilon - closure(S)$ - all states reachable by 0 or more ϵ transitions from a member of S.

```
answer ← S
workset ← S
while work set $\neq \emptyset$.
remove some element s from work set
for every $s' \in T[s, \epsilon]$
if $s' \notin answer$.
answer ← answer $\cup \{s'\}$
$workset \leftarrow workset \cup \{S'\}$
return answer
```

11.3.1 ϵ NFA recognizer

Input $x = a_1, a_2, \cdots, a_n$

```
$states \leftarrow \epsilon - closure(\{state\})$
for i from 1 to n
newstates ← $\cup_{s \in state} T[s, a_i]$
$states \leftarrow \epsilon - closure(newstates)$
accept if states$\cap finish \neq \emptyset$
```

Why? Constructive proof that any regular language is recognized by some ϵ NFA.(and hence by some DFA using subset constructions on the ϵ NFA).

12 Lecture 12 : Regular grammar

Regular language is a finite sets that have union, concatenation, and repetition properties.

1. finite sets: empty set, singleton, big set

   empty set $L = \{\}$ - ϵ NFA

   singleton string s $L = \{s\}$-one finish state

2. Concatenation $L_1 L_2$- use ϵ transition to connect the finish state of L_1 and the start state of L_2

3. Union $L_1 \cup L_2$-create a new start state points to the start states of L_1, L_2 and their finish states using epsilon transition to point to a new finish state

4. Repetition $L_1 *$- Similarly, create a new start start and finish state and connect them use epsilon transitions and point the new start to each corresponding old start state and old finish state.

12.1 Regular Expressions - Textual generative specification

regular-language L	regular expression R(L)
$\{\epsilon\}$	ϵ (or just nothing ())
$\{a\}$	a
L_1L_2	$R(L_1)R(L_2)$
$L_1 \cup L_2$	$R(L_1)\|R(L_2)$ or $R(L_1) + R(L_2)$
L^*	$R(L)^*$
$\emptyset$	$\emptyset$

12.1.1 Examples for Regular expressions

1. $\Sigma = \{0, 1\}$ binary number

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

2. product of 1 or more binary numbers

   $\Sigma = \{0, 1, *\}$ $0|1(0|1)^*(*0|1(0|1)^*)^*$

3. $\Sigma = \{0, 1, *, +\}$ sum of 1 or more products

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

4. Note : size of regular expression doubles with each example

12.2 Extended notations - more compact expression of some things

1. Variable: $I = 0|1(0|1)^*$. $P = I(*I)^*.S = P(*P)^*$. As you see, we can apply simple substitution but careful not to use recursion.

2. $R^+ = RR^*$

3. Algol 68 definition : $\{R|S\} = R(SR)^*$

   Hence our regular expression for example 3 could be $\{\{0|1(0|1)^*|*\}|+\}$

4. $\{R\|S\} = R(SR)^+$

Also there exists irregular expression like $(0|1)^*/\times(111|000)\times 0|1^*$

13 Lecture 13 : Context-Free Language (CFL) &Grammars (CFGs)

1. CFL is a language generated by CFG

2. exactly what can be recognized by a non-deterministic push-down automaton (NPDA)

3. generally CF recognizers require multiple stacks or equivalent ($O(n^3)$ time, n is the input size; a better methods will be $O(n^{2.7})$)

4. Deterministic CFLs - recognized by deterministic PDA. ($O(n)$ parser/recognizer, $LR(k)$ parser)

13.1 An example of this grammar

```
G:
S -> AhB
A -> ab
A -> cd
B -> e
B -> fg

L(G):
{ abhe, abhfg, cdhe, cdhfg}
```

Derivation is the substitution process.
Formal Definition of CFG CFG :
N : finite set of non-terminal symbols (variables)
P : finite set of terminal symbols (alphabet)
R : set of rules of the form $A \to \alpha$ where A is a non-terminal symbol and α is terminal or non-terminal symbols ($A \in N, \alpha \in (N \cup T)^*$)
S : start symbol $S \in N$
Conventions :
a,b,c,d, $\cdots$ terminals
$A, B, C, \cdots, S$ non-terminals
X, Y, Z, W terminals or non-terminals $V = N \cup T$ (vocabulary of G)
x, y, z, w strings from T^*
$\alpha, \beta, \cdots$ strings from V^*

Definition. *$\alpha AB\beta \Rightarrow_{\text{"}derives\text{"}} \alpha\gamma\beta$ means $A \to \gamma \in R$*
$\alpha \Rightarrow^* \beta$ "derives in 0 or more steps" means $\alpha = \beta 4 or \exists \gamma, \alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$.
$L(G) = \{x | S \Rightarrow^* x\}$
A derivation of $S \Rightarrow^* x$ proves that $x \in L(G)$

13.2 Parsing

Given $x \in L(G)$, find a derivation.
e.g. Show that $abhe \in L(G)$
top-down parsing $S \Rightarrow Ahb \Rightarrow abhB \Rightarrow able$

bottom-up parse $abhe \Rightarrow abhB \Rightarrow AhB \Rightarrow S$ (reverse derivation)

Suppose I can always choose leftmost nonterminal in a top down parse, I get left most derivation.

righmost : $abhe \Rightarrow Ahe \Rightarrow AhB \Rightarrow S$
leftmost : $abhe \Rightarrow abhB \Rightarrow AhB \Rightarrow S$
This process is called "finding the handle"

More notations :
$xA\beta \Rightarrow_{lm} x\gamma\beta\ A \to \gamma$.
$\alpha Ay \Rightarrow_{rm} \alpha\gamma y\ A \to \gamma$.

13.3 Parse Tree

```
     S
   / | \
   A  h  B
 / |     |
a b       e
```

Strictly speaking, a parser is given $x \in L$ what if $x \notin L$? Modern parsers are also recognizers.

Binary numbers with no useless 0's :

$T = \{0,1\}, N = \{S,D\}, R = \{S \to 0, S \to 1D, D \to \epsilon, D \to D1, D \to D0\}$ Then $L(G) = \{0,1,10,\cdots\}$ This definition is not ambiguous for 101 since the language has a unique parse tree.

Consider the following rule $\{S \to 1, S \to 1D, D \to \epsilon, D \to 1, D \to 0, D \to DD\}$

13.3.1 Canonical derivation

$R = \{S \to 0, S \to D, D \to 1, D \to D0, D \to D1\}$.

This rule is meaningful since we can label each nodes of the parsing tree with meaningful informations.

14 Lecture 14 : Parsing

We talked about Top down, bottom up and canonical parsing last lecture. Today we will learn the parsing algorithms (stacked based).

E.G.

```
G
S -> AhB
A -> ab
A -> cd
B -> e
B -> fg
```

14.1 Generic Top-Down Parser, given $x \in L(G)$

```
$\delta \leftarrow S$
    where $\delta \neq x$ do
    choose any A where $\delta = \alpha A \beta$
    choose $A \rightarrow \gamma \in R$ [oracle]
    $\delta \leftarrow \alpha A \beta$ [expansion]
    done
```

Let's apply our algorithm:
$x = cdhfg$
$\delta = S\ \alpha = \epsilon, \beta = \epsilon, A = S$
$\delta = AhB$, $\gamma = AhB$
$\delta = Ahfg$
$\delta = cdhfg$

14.2 Generic Canonical Top-down Parser, given $x \in L(G)$

```
$\delta \leftarrow S$
    where $\delta \neq x$ do
    choose A where $\delta = zA\beta$
    choose $A \rightarrow \gamma \in R$ [oracle]
    $\delta \leftarrow \alpha A \beta$ [expansion]
    done
```

$x = cdhfg$
$\delta = S\ \alpha = \epsilon, \beta = \epsilon, A = S$
$\delta = AhB$, $\gamma = AhB$
$\delta = cdhB$
$\delta = cdhfg$

14.3 Generic Bottom-Up Parser for $x \in L(G)$

```
$\delta \leftarrow x$
    while $\delta \neq S$ do
    choose $\alpha\gamma\beta$ where $\delta = \alpha\gamma\beta$ and $A \to \gamma \in R$ (the $\gamma$ is the handle)
    $\delta \leftarrow \alpha A \beta$ [reduction]
    done
```

$\delta = cdhfg$
$\delta = cdhB$
$\delta = AhB$
$\delta = S$

14.4 Generic Canonical Bottom-Up Parser for $x \in L(G)$ - yield rightmost derivation in reverse

```
$\delta \leftarrow x$
    while $\delta \neq S$ do
    choose $\alpha\gamma\beta$ where $\delta = \alpha\gamma z$ and $A \to \gamma \in R$ (the $\gamma$ is the handle)
    $\delta \leftarrow \alpha A z$ [reduction]
    done
```

$\delta = cdhfg$
$\delta = Ahfg$
$\delta = AhB$
$\delta = S$

14.5 Stack-Based Parsing

14.5.1 Augmented Grammar G'

It is formed from any CFG.

$N' = N \cup \{S'\}$
$T' = T \cup \{\vdash, \dashv\}$
$R' = R \cup \{S' \to \vdash S \dashv\}$
S' is start symbol
$L(G') = \{\vdash x \dashv | x \in L(G)\}$

```
input is x in L(G)
|- x -| in L(G’)
parse for |- x -|
do the parse by reading x from left to right

at any point we have x = yz
where y = input already seen
z = input not yet seen
```

14.6 Top-down Canonical Stack-based

crux : way to represent δ

use a stack

$\delta =$ input seen stack top bottom

input(seen)	input (not seen)	top of stack	δ
ϵ	$\vdash cdhfg \dashv$	$\vdash S \dashv$	$\vdash S \dashv$
read $\vdash$	$cdhfg \dashv$	$S \dashv$	$\vdash S \dashv$
expand $\vdash$	$cdhfg \dashv$	$AhB \dashv$	$\vdash AhB \dashv$
expand $\vdash$	$cdhfg \dashv$	$cdhB \dashv$	$\vdash cdhB \dashv$
read $\vdash c$	$dhfg \dashv$	$dhB \dashv$	$\vdash cdhB \dashv$
read $\vdash cd$	$hfg \dashv$	$hB \dashv$	$\vdash cdhB \dashv$
read $\vdash cdh$	$fg \dashv$	$B \dashv$	$\vdash cdhB \dashv$
$\vdash cdh$	$fg \dashv$	$fg \dashv$	$\vdash cdhfg \dashv$
read $\vdash cdhf$	$g \dashv$	$g \dashv$	$\vdash cdhfg \dashv$
read $\vdash cdhfg$	$\dashv$	$\dashv$	$\vdash cdhhfg \dashv$
$\vdash cdhfg \dashv$	ϵ	ϵ	$\vdash cdhhfg \dashv$

After the break we will talk about the Oracle. One Oracle : LL(1) parser. You have Predict[top of stack, first symbol of unseen input]

14.7 Bottom-Up Canonical Stack-based Parser

Crux : way to represent δ

use a stack

$\delta =$stack input not seen.

Stack top	seen	unseen	δ
ϵ	ϵ	$\vdash cdhfg \dashv$	$\vdash cdhfg \dashv$
shift $\vdash$	$\vdash$	$cdhfg \dashv$	$\vdash cdhfg \dashv$
shift $\vdash c$	$\vdash c$	$dhfg \dashv$	$\vdash cdhfg \dashv$
shift $\vdash cd$	$\vdash cd$	$hfg \dashv$	$\vdash cdhfg \dashv$
reduce $\vdash A$	$\vdash cd$	$hfg \dashv$	$\vdash Ahfg \dashv$
shift $\vdash Ah$	$\vdash cdh$	$fg \dashv$	$\vdash Ahfg \dashv$
shift $\vdash Ahf$	$\vdash cdhf$	$g \dashv$	$\vdash Ahfg \dashv$
shift $\vdash Ahfg$	$\vdash cdhfg$	$\dashv$	$\vdash Ahfg \dashv$
reduce $\vdash AhB$	$\vdash cdhfg$	$\dashv$	$\vdash AhB \dashv$
$\vdash S$	$\vdash cdhfg \dashv$	$\dashv$	$\vdash S \dashv$
$\vdash S$	$\vdash cdhfg \dashv$	ϵ	$\vdash S \dashv$

Oracle

Conceptually,

Reduce[stack, first unseen symbol] = $\{A \to \alpha\}$ that could be reduced.
Shift[stack, first unseen symbol] = yes/no
If there is exactly one of reduce and shift, for all stacks, grammar os LR(1).

In 1950, Donald Knuth discovered that any canonical parser was a regular language.

15 Lecture 15: LL(1) Parser

We talked about different kind of parser in last lecture and mentioned oracle which has a predict function $N \times T \to R^*$ and $|Predict(A,a)| \leq 1$.

For LL(1) parsing,

1. build predict function:

   $|Predict(A,a)| \leq 1$ for all A, a, then the grammar is $LL(1)$.

2. use predict in stack-based parser

Above all, it is a parser generator.

15.1 $LL(1)$ parser to parse input x, given Predict

```
push $\vdash$
    push $S$
    push $\dashv$
    $\vdash S \dashv$ is the top of the stack.
    for every symbol a in $\vdash x \dashv$ from left to right while top of stack $\in N$.
    if Predict[top of stack, a] = $\{A \to \alpha\}$
    pop[A] from stack
    push every symbol in $\alpha$ from right to left
    if top of stack $\neq a$ then reject
    accept!
```

Correct prefix properly

informally detect error as soon as possible; more formally, if $LL(1)$ rejects let y be input read so far, there exists z such that $yz \in L$

there does not exist z' such that $yaz' \in L$.

$$1. S' \to \vdash S \dashv$$

$$2.S \to AhB$$

$$3.A \to ab$$

$$4.A \to cd$$

$$5.B \to e$$

$$6.B \to fg$$

	a	b	c	d	e	f	g	$\vdash$	$\dashv$
S'							1		
S	2		2						
A	3		4						
B					5	6			

15.2 Example

```
S -> if x then S
S -> if x then S else S
S -> y
```

If you write a parsing tree for if x then if x then y else y, you will get two distinct parsing trees. This problem is called "dangling else problem.

Define
empty: $V^* \to \{true, false\}$
empty(α) $=_{def} \alpha \to^* \epsilon$
first : $V^* \to 2^T$
$first(\alpha) =_{def} \{a|\alpha \to^* a\beta\}$
follow : $N \to 2^T$
$follow =_{def} \{a|S \to^* \alpha A a \beta\}$
$Predict(A, a) = \{A \to \alpha | a \in first(\alpha) || empty(\alpha)$ and $a \in follow(A)\}$

-precompute empty(A) for $A \in N$
-precompute first(A) $A \in N$.
-precompute follow(A) $A \in N$
-compute predict

16 Lecture 16:

Oracle - "have we seen a handle?" "Is this a correct prefix?"

to parse input x
push $\vdash$

```
for each a in $x \dashv$ (from left to right)
while Reduce[stack, a] = $\{A \to \alpha\}$
pop $\alpha$ symbols from stack
push A
if happy[stack, a] = true, push a
otherwise reject
accept
```

Reduce : $V^* \times T \to R^*$ [size of set ≤ 1]
Happy : $V^* \times T \to \{true, false\}$

Observation (Knuth 1965)
-set of all valid stacks is a regular language

16.1 LR Parser (Linear time using states on stack)

```
push T[start, |-]
for each a in x -|
     while Reduce[top of stack, a] = { A -> a}
         pop |a| times from stack
         push T[top of stack, A]
     if T[top of stack, a] in F
          push T[top of stack, a]
     else reject
accept
```

16.2 Not on Exam: how to construct the LR DFA

start with NFA
states in NFA (items)
$A \to \alpha \cdot \beta$ where $A \to \alpha\beta \in R$.

There is online notes from a kind person: https://github.com/matomesc/cs241

17 Context-sensitive Analysis

```
# Context-sensitive Analysis

- semantic analysis
- how to do A9
- all checking of valid input that can’t be done _easily_ by lexical &
context-free analysis

Uses:

- syntax-directed translation

What we need to do:

- check variables are declared once and once only
- check that variables are used correctly

an example grammar:

‘‘‘
// part of WLPP language specification
procedure -> INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements
RETURN expr SEMI RBRACE
type -> INT // typeof(type) = INT
type -> INT STAR // typeof(type) = INT STAR
dcls ->
dcls -> dcls dcl BECOMES NUM SEMI
dcls -> dcls dcl BECOMES NULL SEMI
dcl -> type ID
‘‘‘

## Building a symbol table
- it will hold a collection of symbols, with information about each symbol
  - for now just: ‘type ? { int, int* }‘
- check for duplicates
- check all __uses__ of symbols
- type of every expression & subexpression in input

__multiset__ symbol table: ‘syms(N) = { <id, type>[i] }‘

example symbol table for the above grammar:

‘‘‘
// symbol table for procedure
syms(procedure) = syms(dcls) + syms(dcl1) + syms(dcl2)
```

```
// an easy one symbols for dcls - the empty set
syms(dcls) = {}

// for dcl -> type ID
syms(dcl) = { <lexeme(ID), typeof(type)> }
‘‘‘

- check to make sure that ‘id1 != id2‘ for all distinct ‘<id1, type1>, <id2, type2> ?
 syms(procedure)‘
- now we have the following functions:

‘‘‘
typeof(type) - declared type
typ(E) - type of any particular expression E ? { int, int*, error }
‘‘‘

more grammar:

‘‘‘
statements ->
statements -> statements statement
statement -> lvalue BECOMES expr SEMI

lvalue -> ID
typ(ID) = t if <lexeme(ID), t> in symbol_table else error

lvalue -> STAR factor
typ(lvalue) = if typ(factor) == int* then int else error

expr -> term
typ(expr) = typ(term)

expr -> expr PLUS term
(expr1 = left expr, expr2 = right expr as per usual)
typ(expr1) = int if typ(expr2) == int and typ(term) == int OR
typ(expr1) = int* if typ(expr2) == int and typ(term) == int* OR
typ(expr1) = int* if typ(expr2) == int* and typ(term) == int OTHERWISE error

term -> factor
typ(term) = typ(factor)
```

```
factor -> lvalue
typ(factor) = typ(lvalue)

factor -> NUM
typ(NUM) = int

factor -> ID
typ(ID) = t if <lexeme(ID), t> in symbol_table else error

factor -> STAR factor
‘‘‘
```

18 Code Generating

```
# Code Generation (completing the compiler)

Summary so far:

‘‘‘
WLPP Source -> Scanner -> Token List -> Parser -> WLPPI file -> Context-Sensitive
analysis (using the parse tree) + Code generation
    -> asm file -> CS241.binasm -> MIPS binary -> mips.twoints
‘‘‘

Pretty much every single one can error.

‘‘‘
Procedure -> asm file (stdout):

Prologue
Code fragment code (procedure) from syntax directed translation
Epilogue
‘‘‘

## Prologue

- prologue establishes conventions
- save registers
- capture parameters
- anything else than needs to get setup
```

## Epilogue

- restore registers
- produce the result
- utility procedures (for instance an external print procedure for stdout)

## Conventions

- self imposed rules to make everybody get along
- how is data going to be represented?
  - how are variables going to be represented? WLPP has ‘int‘ and ‘int *‘
  - how are expressions going to be represented?
  - how are registers to be used for __parameters__ and __results__?

## Suggested Conventions
Can also be found [here](http://www.student.cs.uwaterloo.ca/~cs241/))

## Prologue Code Generation

- save registers
- set $11, $4
- allocate memory on stack for __all variables__
  - the __symbol table__ will tell us this information
  - use __stack__
  - subtract ‘$4 * n‘ from the stack pointer ‘$30‘

  ‘‘‘
  // allocating a stack frame

  ====== <-$30 (after)
  (stack frame)
  ====== <-$30 (initial)

  ======
  ‘‘‘

## Epilogue Code Generation

It basically has to undo the prologue.

- add ‘4n‘ to the stack pointer ‘$30‘
- restore registers (except $3 which will store the procedure’s result)
- ‘jr $31‘
- also include library procedures here such as ‘print‘

## Code Generation for Declarations, Statements & Expressions

### factor -> ID

- lookup factor in symbol table
- ‘offset(ID)‘ is assigned offset in frame for ID
- get that result into $3

code for factor:

‘‘‘
lw $3, offset($29)
‘‘‘

### factor -> NUM

‘‘‘
lis $3
.word value(NUM)
‘‘‘

### factor -> NULL

Oh how are we gonna represent pointer? Well an ‘int *‘ is an address and ‘NULL‘
is represented by address 0.
This is just a convention.

Then all we need to do to represent ‘NULL‘ is zero the result register:

‘‘‘
sub $3, $3, $3
‘‘‘

### factor -> (expr)

‘code(factor) = code (expr)‘

### statement -> lvalue = expr;

We know how to generate ‘expr‘. But how do we generate ‘lvalue‘? (lvalue is the left hand side of an expression).
We want the lvalue to return an address in RAM ie. store the address in $3.

```
‘‘‘
code(lvalue) // $3 now holds pointer to lvalue

// push $3 on stack
sw $3, -4($30)
sub $30, $30, $4

code(expr) // result of expr in $3

// pop $3 off stack into $5
add $30, $30, $4
lw $5, -4($30)

// store $3 at address $5
sw $3, 0($5)
‘‘‘
```

This could have easily been done by returning ‘lvalue‘ in register ‘$5‘ to save us from messing with the stack.

### factor -> & lvalue

```
‘‘‘
assert(typ(lvalue) == int)

‘‘‘
```

Since the lvalue has already been generated, there is nothing else to do so ‘
code(factor) = code(lvalue)‘

### factor1 -> * factor2

‘‘‘

~~~~
assert(typ(factor2)) == int *)
lw $3, 0($3)
‘‘‘

### lvalue -> ID

- find the offset of ‘ID‘ in the symbol table. This offset should be relative to
‘$29‘ (stack frame pointer)

‘‘‘
lis $5
.word offset
add $3, $29, $5
‘‘‘

### lvalue -> * factor

‘‘‘
assert(typ(factor) == int *)
// since * factor is a pointer and lvalue is also a pointer, there is nothing to be done
code(lvalue) = code(factor)
‘‘‘

### dcls -> dcls dcl = NUM;

‘‘‘
assert(typeof dcl == int)

code(dcl) // $3 now stores address of dcl’s ID

// load value(NUM) into $5
lis $5
.word value(NUM)
// put $5 at address $3
sw $5, 0($3)

code(dcls)
‘‘‘

### dcl -> type ID
~~~~

```
- same as lvalue -> ID

‘‘‘
lis $5
.word offset(ID)
add $3, $29, $5
‘‘‘

### procedure -> INT WAIN ... dcl1 ... dcl1 ... dcls ... statements ... expr

This is the __meat__

‘‘‘
// this returns an lvalue
// it is also the first parameter to the procedure
code(dcl1)
// store it
sw $1, 0($3)

// 2nd parameter
code(dcl2)
// store it
sw $2, 0($3)

code(dcls)

code(statements)

// final result that goes into $3
code(expr)
‘‘‘

### expr -> expr - or + term

case 1: ‘typ(expr) = typ(term) = int‘

‘‘‘
code(expr) // result in $3
sw $3, -4($30)
sub $30, $30, $4 // push $3 on stack
code(term) // result in $3
```

```
add $30, $30, $4 // pop stack
lw $5, -4($30) // into $5
sub $3, $5, $3 // for -
add $3, $5, $3 // for +
```

case 2: ‘typ(expr) = int*, typ(term) = int‘

```
code(expr)
sw $3, -4($30)
sub $30, $30, $4
code(term)
add $3, $3, $3
add $3, $3, $3 // multiply by 4
add $30, $30, $4
lw $5, -4($30)
add $3, $5, $3
```

case 3: ‘typ(expr) = int, typ(term) = int*‘

```
code(term)
sw $3, -4($30)
sub $30, $30, $4
code(expr)
add $3, $3, $3
add $3, $3, $3 // multiply by 4
add $30, $30, $4
lw $5, -4($30)
add $3, $5, $3
```

```
int *a;
int *b;

a - b = (addr(a) - addr(b)) / 4
```

### term1 -> term2 * or / or % factor

```
‘‘‘
code(term2)
sw $3, -4($30)
sub $30, $30, $4
code(factor)
add $30, $30, $4
lw $5, -4($30)

// for multiplication
mult $5, $3 // result goes into HI/LO so we have to move em from there
mflo $3

// for division
// div $5, $3
// mflo $3

// for remainder
// div $5, $3
// mfhi $3
‘‘‘
```

### factor -> NEW INT [expr]

- create heap in prologue
- allocate ‘n‘ bytes where ‘n‘ is result of expr
- how are we gonna accomplish this? two library functions we’ll write: ‘malloc‘
- the call to ‘malloc‘ needs to be in epilogue

```
‘‘‘
code(expr)
add $1, $3, $3
add $1, $1, $1 // put 4 * $3 in $1
// call malloc, and result is in $3
lis $3
.word malloc
jalr $3
‘‘‘
```

### statement -> DELETE [ ] expr ;

```
‘‘‘
code(expr)
add $1, $3, $0
lis $3
.word free // assume we have a function called free in the epilogue
jalr $3
‘‘‘

### statement -> PRINTLN ( expr ) ;

‘‘‘
code(expr)
add $1, $3, $0
lis $3
.word println // again assume println function in epilogue
jalr $3
‘‘‘

### test -> expr1 == expr2

- test is a boolean expression
- since we don’t have booleans in WLPP, we need to define some conventions
for true and false
  - a word containing ‘0‘ for false
  - a word containing ‘1‘ for true
- one implementation:

‘‘‘
code(expr1)
sw $3, -4($30)
sub $30, $30, $4
code(expr2)
add $30, $30, $4
lw $5, -4($30)
add $1, $11, $0 // put 1 in $1
beq $3, $5, 1
add $1, $0, $0 // put 0 in $1
add $3, $1, $0 // copy $1 to $3
‘‘‘
```

### test -> expr1 != expr2

- same code as above except return values are reversed

```
code(expr1)
sw $3, -4($30)
sub $30, $30, $4
code(expr2)
add $30, $30, $4
lw $5, -4($30)
add $1, $0, $0
beq $3, $5, 1
add $1, $11, $0
add $3, $1, $0 // copy $1 to $3
```

### test -> expr1 < expr2

```
code(expr1)
sw $3, -4($30)
sub $30, $30, $4
code(expr2)
add $30, $30, $4
lw $5, -4($30)
slt $3, $5, $3
```

### test -> expr1 > expr2

- same as above except the comparison:

```
slt $3, $3, $5
```

### test -> expr1 <= expr2

- note: `<=` is the same as `not >`

```text
‘‘‘
code(expr1)
// push $3
code(expr2)
// pop $5
slt $3, $3, $5 // is expr2 < expr1
sub $3, $11, $3 // so we need to NOT this
‘‘‘

### test -> expr1 >= expr2

- same as above but reverse the expressions:

‘‘‘
code(expr2)
// push $3
code(expr1)
// pop $5
slt $3, $3, $5 // is expr2 < expr1
sub $3, $11, $3 // so we need to NOT this
‘‘‘

### statement -> if (test) { statements1 } else { statements2 }

‘‘‘
code(test)
beq $3, $0, falsepart
code(statements1)
beq $0, $0, done
falsepart:
code(statements2)
done:
‘‘‘

This won’t work because if we have nested if-else statements, we will have
duplicate labels.
The easiest way around this is to write a ‘nextLabel()‘ function to generate a unique
number suffix for every label.

Another implementation is:
```

````
‘‘‘
code(test)
bnq $3, $0, truepart
code(statements2)
beq $0, $0, done
truepart:
code(statements1)
done:
‘‘‘

Another limitation is that ‘code(statements1)‘ is limited to ˜32k
instructions due to the ‘beq‘.

### statement -> while (test) { statements }

‘‘‘
loop457:
code(test)
beq $3, $0, done456 // random suffix
code(statements)
beq $0, $0, loop457
done456:
‘‘‘
````

19 Memory Management

````
# Memory Management

- how to represent __variables__, values and data structure
- put variables in __frames__
- what about __precedures__, dynamic data structures?
  - WLPP has only the main procedure
  - C/C++ has user-defined procedures
  - Scheme’s got nested procedures

## Procedures

‘‘‘c
// global
int q;
````

```
// some procedure
int foo(int x, int y) {
  int a = 3;
  int* b = NULL;
  a = a + q;
  return *(a+b);
}

// anohter procedure
int bar(int w) {
  int z;
  z = z + q;
}
‘‘‘

- how to represent ‘x, y, a, b‘?
- what about the __global__ ‘q‘?
- how to represent the procedure?

### Parameters

- our convention: ‘x‘ is passed as ‘$1‘ and ‘y‘ is passed as ‘$2‘
- easiest is to put ‘x, y‘ in RAM

### Local variables

- create a __frame__ for foo allocated on __stack__ whenever foo is _called_!
- create new frame, set the __frame pointer__

### Local frame

The local frame pointer will point to local variables in the __local frame__,
 in this case:

‘‘‘
// foo’s local stack frame
---- <- frame pointer (fp)
b
----
a
```

```
----
y
----
x
----

// bar’s local stack frame
---- <- fp
z
----
w
----
‘‘‘

### Global frame

‘‘‘
----
q
----
‘‘‘

## Nested Procedures

- the most inner stack frame must also have access to previously allocated
stack frames

## Non-stack Memory Management

- allocated storage that __persists__ beyond procedure invocation

‘‘‘c
x = new int[10];  // allocate 10 words
// ....
delete[] x;       // free it up
‘‘‘

## Garbage Collection

What about Scheme (or almost any other modern language)?
```

- we only allocate memory
- the language has built in __automatic garbage collection__ which frees unused memory
- we will discuss this concept in details next lecture
- but the idea is we keep track of everything we allocated and find all the reachable data,
  then get rid of uncreachable data

Another way to get a dangling reference in C:

```c
int* dangle() {
  int x;
  int* y = &x;
  return y;
}

int* a = dangle();

// global fp
----
a
----

// dangle's fp
----
x
----
y
----

// but once dangle executes, the frame will get deleted leaving `a` dangling
```

Anyways the idea is stack based memory management sucks.

## Implementing Non-stack Memory Management

- build a set of library functions that implement the following on a global arena of storage (aka __heap__):
- note this is different that priority queues which is also implemented

```
by a heap (which is a tree data struct)
  - initialization
  - finalization
  - allocation (how we get new memory)
  - reclamation (reuse memory that is no longer in use)
    - identification
    - reuse
- modify the code generator to invoke the library functions as

__simple approach__

- use heap for __fixed-sized__ allocation unsits
  - eg. Scheme: ‘(cons a b)‘

__complex approach__

- __varaible-sized__ allocation units

### Fixed-size Allocation

- done in the initialization (prologue)
- allocate a "big" area of storage from stack (just like the frame):

‘‘‘c
---- <- ap

ARENA

---- <- end

.word 0
‘‘‘

- allocation for fixed size ‘n‘:
  - find available ‘n‘ bytes, or __fail__ (this is important)
  - use the __slice of bread__ algorithm:

‘‘‘c
start                      end
  ^                         ^
  | (USED)  x    (UNUSED)    |
```

~~~~
            ^
      first_available

if (first_available + n > end) {
  fail()
}

avail = avail + n
return avail - n
```

- finalization? get rid of `ARENA`
- reuse is trivial (vacuous): `free() {}`
- wastes space, needlessly fails
  - this is due to fragmentation in the heap ie. `XOXOX` where `X` is used
  and `O` was used but was freed
- fail only if there is not enough unused memory to satisfy the request
- we need to keep track of unused (free) storage using an available space list:

```c
// X = used

// stores pointers to the first byte of each free chunk
// each node points to the next chunk of free space
List = a1 -> a2 -> a3 -> a4;

 a1     a2      a3     a4
|   XXXX     XXX    XXX           |
                              ^
                            avail

if asl != null then
  tmp = asl
  asl = *(asl)
else if avail + n > end then fail
else avail = avail + n
  return avail - n
```
~~~~

20 Optimization

```
# Optimization

Source -> Compiler -> Object

‘assert m[source] = m[object]‘

In general, there is ‘object1 ! = object2 != object3‘ such that ‘m[object1] =
m[object2] = m[object3]‘

## goodness[object]

Ideally, find ‘objecti‘ such that ‘m[objecti] = m[source]‘ and ‘goodness[objecti]‘ is
maximized.

But what is ‘goodness‘?

- typically __speed__ (minimize running time)

## Static Optimization

Done at compile time based on things that we know already.

## Dynamic Optimization

Done at runtime.

## Space

- often smaller = faster
- also smaller and optimized loops results in faster programs due to instruction and data
caching in the CPU
- space is really easy to measured

For example consider optimizing a ‘beq‘:

‘‘‘
beq $0, $0, foo // only works if branch offset <= 32767 words

// a workaround is to do:
```

```
lis $5
.word foo
jr $5
‘‘‘

But consider a lot of flow control (ie. a lot of branches). Optimizing these branches is
an NP complete problem because the optimization of one branch affects the
optimization of the rest. So one solution is approximating optimizations that
are always safe (ie. use long branches everywhere) but some branches
 could definitely be shortened.

## Approaches to Optimization

### Constant expression evaluation (folding)

  - ‘x = x + 2 * 3; => x = x + 6;‘
  - from syntax directed translation, the code we generate can either:
    - puts result in $3
    - computes result as constant
  - ‘code(exec) = < code, representation >‘

### Dead code elimination

Example:

‘‘‘C
if (6 != 2 * 3) {
  // ...
}
‘‘‘

There should be no code generated

### Invariant detection

Treat variables that don’t change as constants.

Example:

‘‘‘C
int x = 2;
```

// ...
// x never changes
// treate x as constant
‘‘‘

How do we prove that x is a constant?

- check for ‘x = ... ‘, definitely not a constant since x was updated
- check for ‘y = &x‘ is tricky because ‘*y‘ now points to ‘x‘ and if you do ‘*y = 25‘, then this will change ‘x‘
  - this is a big thing in compiler optimization (alias detection)
  - you wanna be as safe and complete as possible
  - very tricky stuff going on with these __invariants__

### Common expression elimination

Copy elimination:

‘‘‘C
// this can be optimized
x = 2 * y + 10;
z = 2 * y;

// to the following, provided z doesn’t cahnge
z = 2 * y;
x = z + 10;
‘‘‘

Another example:

‘‘‘C
x = a[10];
a[10] = 2;
‘‘‘

Since we do a lot of work trying to find the address of ‘a[10]‘, the compiler can then cache the address.

### Loop optimization

Example:

```
‘‘‘C
q = 2 * 3 * y;
while (t) {
  x = x + 2 * 3 * y;  // if y is unchaged in loop, move it outside (to q)!
}
‘‘‘

This _might_ make things faster if we’re executing the loop a large
number of times.

### Strength reduction

‘‘‘c
char *a, *b;
strlen(strcat(a,b));
‘‘‘

- doing concatenation for nothing!
- represent strings as their length and value
- if you only manipulate the length, then their value becomes dead code
- consider the following:

‘‘‘c
for (i = 0; i < 10; i++) {
  a[i] = 2;
}
‘‘‘

- when you do a[i], you are storing 2 at ‘addr(a) + 4 * i‘
- the issue here is ‘4 * i‘
- transform it to this:

‘‘‘c
t = addr(a) + 40
for (i = 0, I = addr(a); I < t; i += 1, I += 4) {
  // store 2 at address I
}
‘‘‘

- keeping a copy of the original ‘i‘ is useful if the value of ‘i‘ was used inside the
```

 loop (ie. `a[i] = i`)

### Register usage

We have 32 registers available. Suppose we use 16 to store variables.

This means that these 16 variables won't be in the frame, saving us from messing with the stack frame.

So, for instance, consider we need to return a variable that we have in `$22`.

We could copy it to `$3`: `add $3, $22, $0`, but this is a redundant copy.
We should just leave
it in `$22` and next when we use it we should expect it in `$22`. This requires a change of our __conventions__:

- result of expr can be any register: `<code, resultreg>`

`expr -> expr + term`:

```
// code(expr1):
sw $r, -4($30) // where r is resultreg(expr1)
```

- then we tell code which registers it is allowed to use: `code(expr, allow)`
where `allow` is a set of registers that can be safely used

```
code(expr1, allow)
code(expr2, allow) // result in is resultreg(expr2)
code(term, allow \ resultreg(expr2)) // set difference
// pick a register $t from allow, unless allow is empty
add $t, $r, $s // where $r is resultreg(expr), $s = resultreg(term)
```

- sometimes if we're out of registers, this is called __register pressure__
- managing registers is also a pretty difficult problem

#### Managing registers

Consider the following production:

‘expr -> expr + term;‘

Let needregs(expr) = how many registers we need to allocate good code

So, ‘needregs(expr1) = max(needregs(expr2), needreg(term)) + 1‘. Remeber to save a
 register for the return result
(hence the ‘+ 1‘)