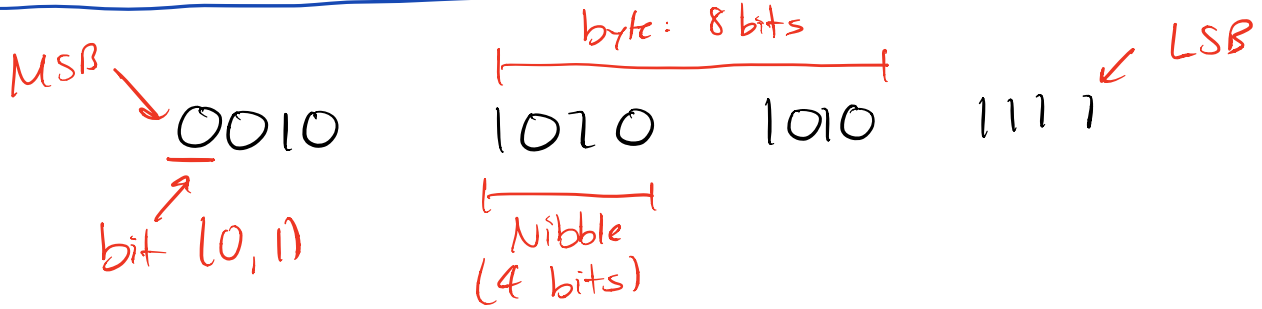
CS 241

DATA REPRESENTATIONS



- Word: dependant processor
  - 32-bit / 64-bit ⇒

Word	32-bit	64-bit
	32 bits	64 bits
	↓ ÷ 8	↓ ÷ 8
	4 bytes	8 bytes

Binary number

- Unsigned int
- Signed int
- Characters

1. Unsigned integer: $n \geq 0$
   a) UI ↔ decimal:
      ↳ UI → decimal:

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 = (2^0 \times b_0) + (2^1 \times b_1) + (2^2 \times b_2) \ldots + (b_7 \times 2^7)$$

      ↳ Decimal → UI
         i) Group into largest power of 2

$$68 = 64 + 4$$
$$= 2^6 + 2^2$$

$$1000100 = 68$$

2^6 2^2 2^1 2^0

ii) Repeated division : use remainder to construct digit

68 $\div 2$

Num	Q	R
68	34	0
34	17	0
17	8	1
8	4	0
4	2	0
2	1	0
1	0	1

Q = 0

68 = 1000100

b) Arithmetic: overflow

```
  1 1 1 1 1 1
... 0 0 0 1 0 0 1 0 0 1
  + 0 1 1 1 1 1 1 1
  -----------------
... 1 1 0 0 1 0 0 0
```

2. Signed integers: neg. + pos.

Negative?

Use this!

MSB as indicator
- ↳ 0 → ⊕
- ↳ 1 → ⊖

10...0

00...0

2's complement

Neg: flip all bits + 1

-8 ⇒ ① Positive integer rep.

1000

② Flip

0111

③ Add 1:

1000 ⇒ −8

- 2's complement trick: flip trick:

  Take the right-most 1 bit ⇒ flip everything to left

Ex: −8 ⇒ 0001000 ⇓ Flip 1...1111000

−68: 10001100 ⇓ Flip ...01110100 ⇒ −68

Proof:

8 bit ⇒ $0-(2^8-1)$

255 < n ⇒ 0

Make $2^8 \approx 0$ ⇒ system mod 2^8 (mod 256)

$$2^8-1 = 2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$$

Represent −68:

$$2^8-1 = 2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$$

$$2^8-(2^6+2^2) = (2^7+2^5+2^4+2^3+2^1+2^0)+1$$

$2^8-(2^6+2^2)$ = $-(2^6+2^2)$ = -68

$(2^7+2^5+2^4+2^3+2^1+2^0)+1$: Every other bit is flipped

Hexadecimal: base 16

0 ⟶ 16

0-9: 0-9

10-11: A-F

$AF8_{16} = 8\times16^0 + F\times16^1 + A\times16^2$

$= 8\times16^0 + 15\times16^1 + 10\times16^2$

$=$ ~ ~ ~

Binary → hexadecimal:

1) Seperate binary into nibbles:

$0101\,0010\,1011 \Rightarrow 0101\ 0010\ 1011$

2) Convert each nibble → hexadecimal

$0101\ 0010\ 1011 \Rightarrow \boxed{52B_{16}}$

3. Characters:

ASCII / UTF / ... ⇒ encodings

- ASCII table (man ascii)

Operations

bitwise AND: AND of bit pair (&)

$1001\ \&\ 1100 \Rightarrow$

$$\begin{array}{r} 1001 \\ \&\ 1100 \\ \hline 1000 \end{array}$$

bitwise OR: OR of bit pair (|)

$$\begin{array}{r} 1001 \\ |\ 1100 \\ \hline 1101 \end{array}$$

bitwise NOT (~)

bitwise XOR (^): $a \wedge b$

Shifts:

- Left-shift (<<): $x << n = x \times 2^n$

$1001 << 3 = 1001000$

- Right-bit (>>): $x >> n = x \div 2^n$ (remainder are save)

$1001 >> 3 = 000\ldots01001 >> 3$

$2^0 = 000\ldots0\ 0001$

- Ex://

```
unsigned char c = 1;   ⇒ 00001   (digit)
unsigned char d = 25   ⇒ 11001
c <<= 3;               ⇒ 001000
c |= 1;                ⇒ 1001
c = c & d
printf("%d", c);
```

integer: 9

```
001000
   - - - 1
---------
   1001
```

```
 01001
&11001
-------
 01001
    |
    9
```

MACHINE & ASSEMBLY LANGUAGE

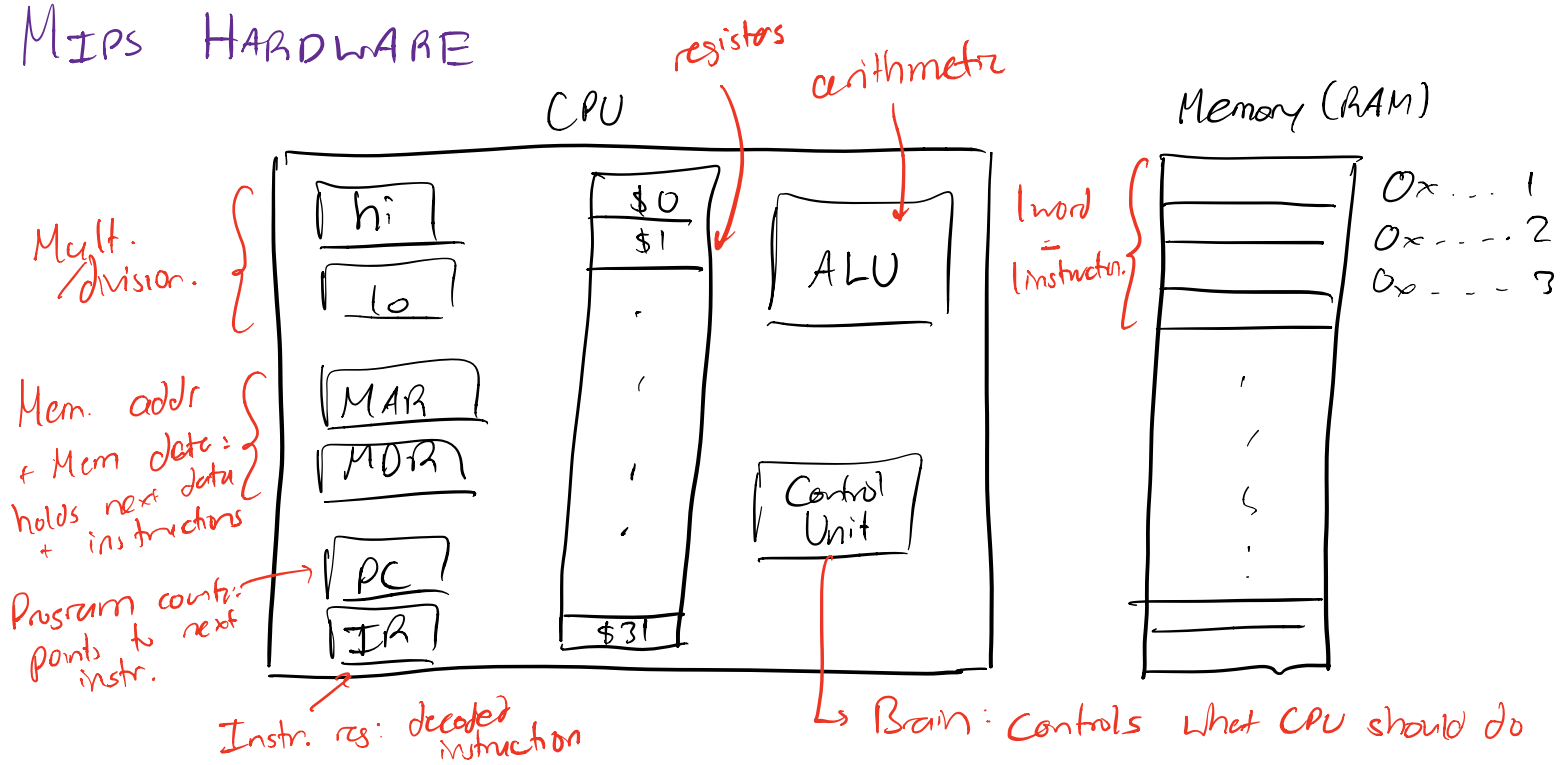
Programming → Assembly language → Machine

Assembly language: Processor dependent

Assembly language ⇓ Somewhat readable code s.t. processor will understand

Machine ⇓ Sequence of 0's of 1's

MIPS HARDWARE



- Special registers:
  - $0: holds 0
  - $29: frame ptr.
  - $30: stack ptr.
  - $31: return register
- Processor doesn't know difference between data and instruction:
  - Different data representations for a sequence of bits, processor makes no assumptions.
  - Know the representation before hand.
- Fetch - increment - execute cycle:
  1. Load into start of memory, set up PC to start of memory
  2. Fetch: IR fetches word from RAM based on PC
     ↳ PC, PC+1, PC+2, PC+3
  3. Increment: PC += 4 (already at next instruction)
  4. Execute: take IR instructions → CU will decide what to do

MIPS LANGUAGE

- In general: every MIPS command has an encoding into bits.
  - Ex: add $s, $t, $d: 000000 ttttt ddddd sssss 00000 100000
    ↳ 5 bits: Range of 5 bits is 0-31
- Addition + subtraction:

  add $s, $t, $d ⇒ $s = $t + $d

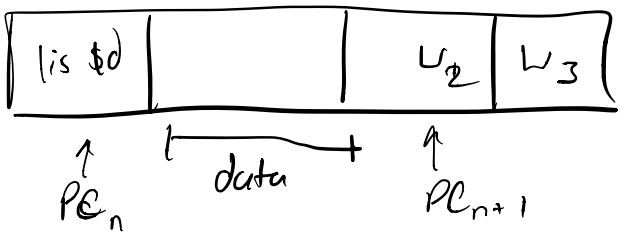
  sub " ⇒ $s = $t - $d
- Jump register: move the PC to the address in $s

jr $s

- jr $31: returning from the program

- Load immediate + skip instruction: loading immediates/constants into a register.

```
lis $d
.word 0 / a / memory addr.
```



- Ex://
```
lis $3
.word 7
lis $7
.word 3
add $2, $3, $7
jr $31
```

- Multiply & divide:
  - Multiplication:
    - Signed: mult $s, $d ⇒ $s x $d
    - Unsigned: multu $s, $d
    - ▷ Storage: hi holds MSW, lo holds LSW (meant for overflow)
  - Division:
    - Signed: div $s, $d ⇒ $s / $d
    - Unsigned: divu $s, $d
    - ▷ storage: quotient is in lo, remainder is in hi
  - Moving from hi/lo: mflo / mfhi $d (move lo/hi → $d)

- Branching:

- Branch if equal: `beq $s, $t, i`
- Branch if not equal: `bne $s, $t, i`

} Skip i instructions if $s == $t (beq) or $s != $t (bne)

- Ex:

```
lis $3
.word 3
lis $2
.word 11
lis $5
.word 2
div $1, $5
mfhi $6
beq $6, $0, 1
add $2, $3, $0
jr $31
```

odd → add $2, $3, $0; even → jr $31

} Is $1 divisible by 2?

- Essentially moves PC i words away based on branching
- Tips:
  1. Opposite case is addressed right after branch
  2. Makes sure comparison is correct

- Set less than:

`slt/sltu $d, $s, $t`

$d = 1 if $s < $t, otherwise $d = 0

- Use in conjunction w/ branching.
- Ex:

```
slt $2, $1, $0
beq $2, $0, 1
sub $1, $0, $1
jr $31
```

} Calculating abs value of $1

- Conditional loops: use beq, bne to loop

```
lis $2
.word 13
add $3, $0, $0
add $3, $0, $2
lis $1
.word -1
add $2, $1, $2
bne $2, $0, -5
jr $31
```

do { ... } while (...)

Offset: +, −

PC_n

- Use labels instead: labels holds memory address

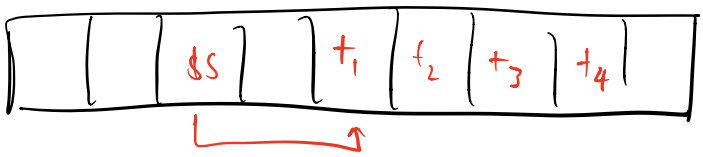
```
lis $2
.word $13
lis $1
.word -1
add $3, $0, $0
```

```
loop:
  add $3, $2, $3
  add $2, $2, $1
  bne $2, $0, loop
  jr $31
```

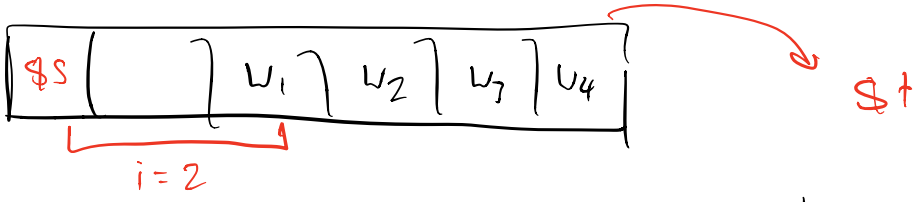
labels are associated w/ address of next instr.

▷ Formula: $\text{offset} = \frac{\text{label} - PC}{4}$

- Storing and loading words
  - Storing: sw $t, i ($s) ⇒ store the value in $t and put it in MEM[$s + i]
    - i: # of bytes! Not words



  - Loading: lw $t, i($s) ⇒ loading in word from MEM[$s+i]-MEM[$s+i+3] and put it into $t



    - Aligned access: memory address should be multiple of 4
      - ▷ ($s + i) % 4 == 0
  - Ex:// $1 contains address to base of arr. $2 contains # of elements. Retrive a[3]

Loading 4th word ⇒ skip 3 words ⇒ 3 × 4 = 12 bytes

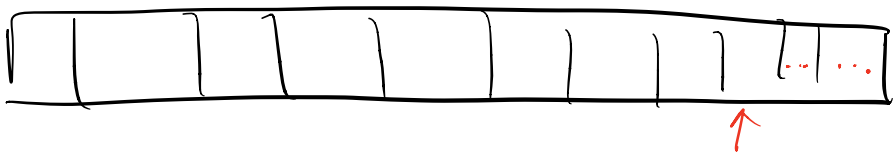
```
lis $5
.word 3
lis $4
.word 4
```

```
mult $5, $4
mflo $5
add $1, $5, $1
lw $3, 0($1)
jr $31
```

already added offset! Tip to do when programming

no specifying necessary

- I/O:
  - Input: load from 0xFFFF0004 ⇒ read 1 character (LSB)
  - Output: store from 0xFFFF000C
- Procedures: functions
  - Issues: registers have global scope, transfer control, recursion
    arguments / return values
  - Addressing register usage:
    - Trick: store registers in RAM → procedure → restore register
    - Use stack pointer ($30): points to memory of last allocated memory.



① Stored in $30 — $30

② store in $30-1

③ store in $30 -1

- Flow:

Pushing to mem. stack
1. Store register in RAM using neg. offsets from $30
2. Update $30 s.t. it is pointing to last word

3. Perform operations
4. Restore registers + $30 → Popping from stack

- Ex://

$30

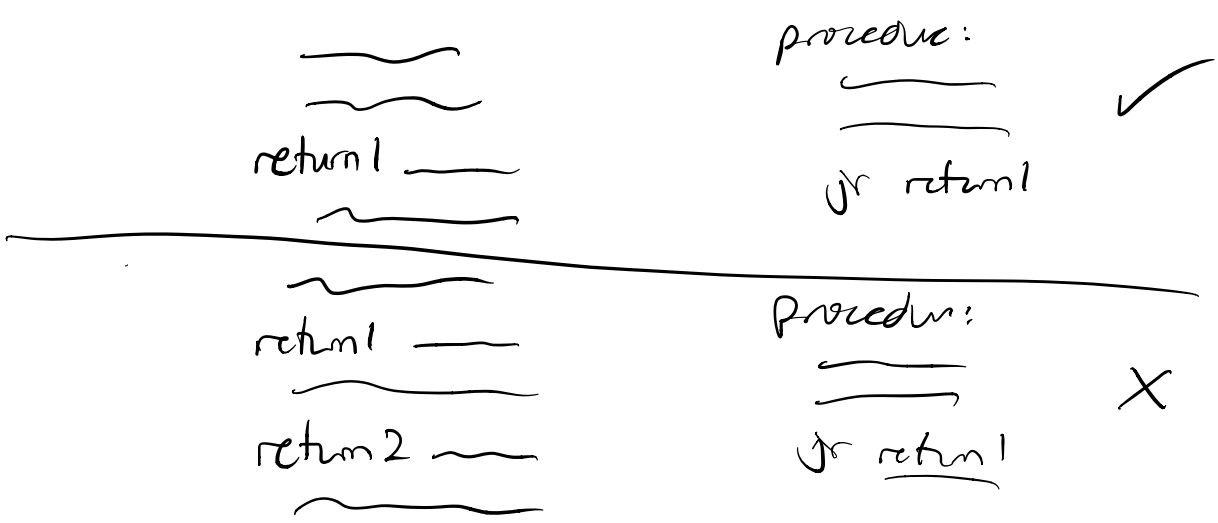
Pushing to memory

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

Pop from Memory?

```
lis $4
.word 4
add $30, $30, $4
lw $1, -4($30)
```

- Calling & returning from registers.
  - Problem w/ labels:



↳ Calling procedure from 2 places does not work w/ labels

- Jump register: same issue
- Use jalr instead
  - Ex: // lis $21 / .word foo / jalr $21 ... jalr $21 ... foo: ... jr $31
    1. Updates $31 to PC (pointing to next instr.)
    2. PC = $21 (to execute procedure)
  - Store $31 in RAM to prevent overwrite.

pushing $31 to mem.
```
sw $31, -4($30)
lis $31
.word 4
sub $30, $30, $31
```

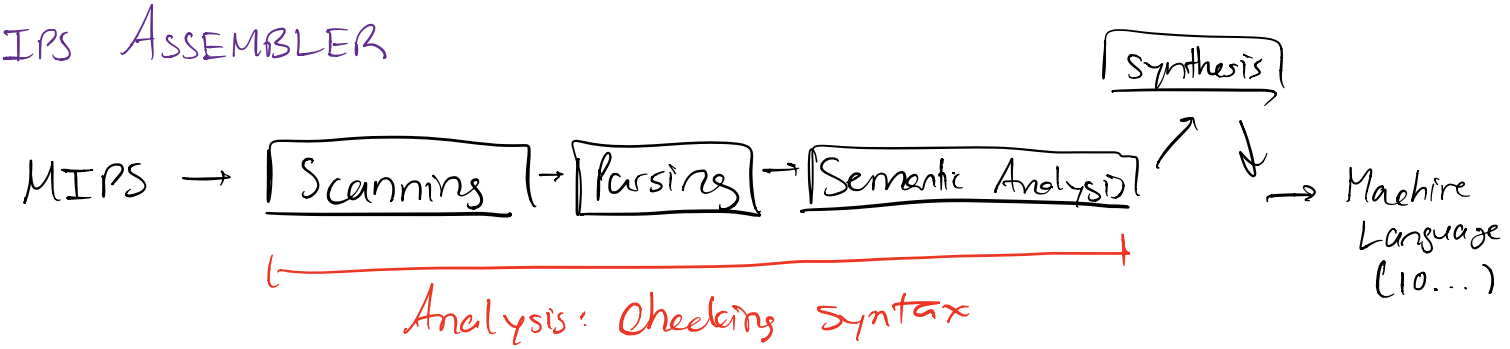
call procedure
```
lis $21
.word foo
jalr $21
```

pop from mem.
```
lis $31
.word 4
add $30, $30, $31
lw $31, -4($30)
```

- Recursion: continually updat stack pointer + push in memory
  - Could potentially run out of stack space

- Pass in parameters: either use registers + persist, use stack just like method before for storing reg.

MIPS ASSEMBLER



- Analysis:
  - Scanning: Characters → tokens
  - Parsing: syntax checking
  - Semantic analysis: unique labels, semantic
- Synthesis: labels, instruction encoding, generating output
  - Label: associate labels w/ addresses
    - Symbol table:

label	addresses
⋮	⋮

⇒ Semantic analysis can check unique labels easily.

    - Problem: labels used before definition
      - ↳ 2 passes: first → scanning, parsing, symbol table
        second pass → semantic analysis, synthesis
    - Ex:

0x00	main: lis $2	
0x04	.word beyond	
0x08	lis $1	
0x0c	.word 2	
0x10	add $3, $0, $0	
0x10	top: begin:	

label	addr.
main	0x00
top	0x10
begin	0x10

- Instruction encoding:
  - Ex: // Generate encoding for add $3, $2, $4
    1) Encoding formally:

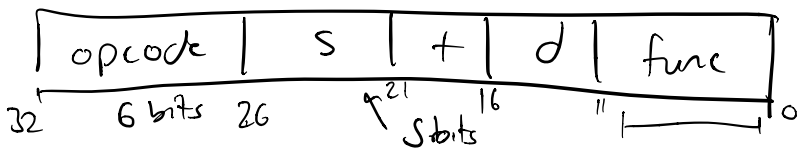
add $d, $s, $t : 000000 sssss ttttt ddddd 00000 100000

2) Assemble mappings:

Value:	Decimal	Binary
Op:	0	0 0 0 0 0 0
s	2	0 0 0 1 0
t	4	0 0 1 0 0
d	3	0 0 0 1 1
func	32	1 0 0 0 0 0

Operations; registers

3) Encoding → correct stop



Op code: shift left by 26 bits
s: shift left by 21
t: shift left by 16
d: shift left by 11
func: don't shift

4) Combine: bitwise or

1100 00. . . 0
| 0 1 1 1 . . .
1100 0 1 1 1 . . .

5) Offsets: 32 bit constants: 16x ↓ 16x ↓

- Masking: bitwise AND w/ 0...01...1

1011
& 0111
0011

Put 1's in places where we mimic the number, 0's where we ignore.

- Generak output: ASCII & stedard output

REGULAR LANGUAGES

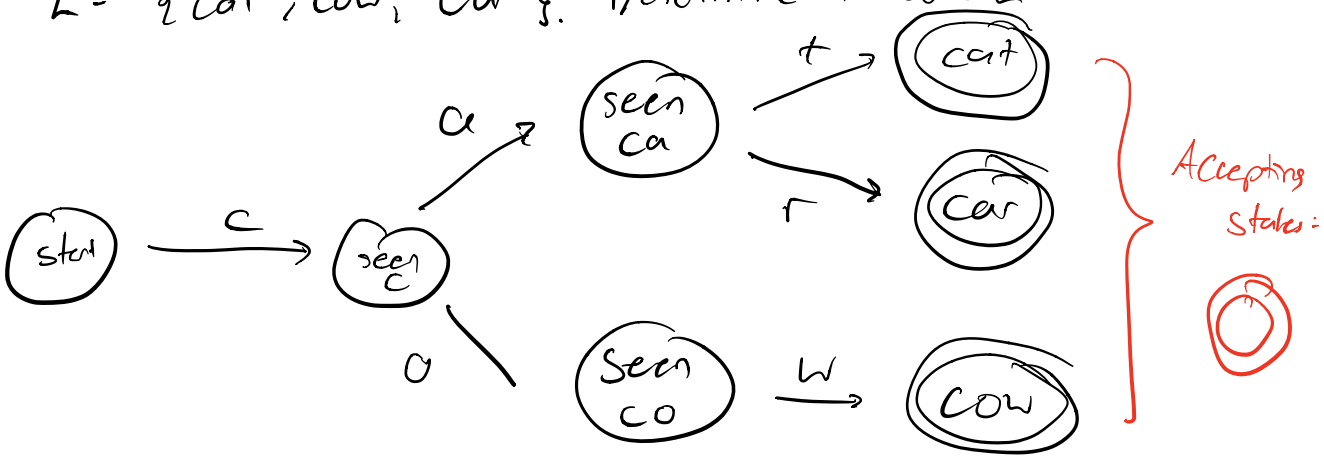
Formal Languages

- Alphabet (Σ): non-empty finite set of symbols
  - Ex: // $\Sigma = \{a, \ldots, z\}$, $\Sigma = \{0, 1\}$
- Word: finite sequence of symbols chosen from Σ
  - Σ^*: set of all words from alphabet Σ
  - $|w|$: length of word
  - ϵ: empty word $\Rightarrow \forall \Sigma, \epsilon \in \Sigma^* \wedge |\epsilon| = 0$
- Language: set of strings w/ some constraint
  - Ex: // 1. $L = \emptyset$, $L = \{\}$ $\Rightarrow$ empty language
    2. $L = \{\epsilon\}$ $\Rightarrow$ language made of empty word
    3. $L = \{ab^n a : n \in \mathbb{N}\}$ $\Rightarrow$ strings from $\Sigma = \{a, b\}$ s.t a followed by 0 or more b followed by an a
- Recognition algorithm: determines if word fits in language

Finite Languages

- Finite # of words $\in L$
- Recognition algorithm:
  1. List all words + check if our word belongs (ineffraint)
  2. state diagrams
- State diagram:
  - Rules:
    1. States: unique order of letters seen so far
       - Start state, accepting states (legit word seen)

2. Transitions: tell us how to go from 1 state to another based on letter seen
   - If no transitions defined for letter ⇒ error state
- Ex: // $L = \{cat, cow, car\}$. Determine if $w \in L$



Regular Languages

- Definition of regular:
  1. $\{\}$ or $\{\epsilon\}$ are regular
  2. $\{a\}$ for some $a \in \Sigma$
  3. Language built using union, concat. or Kleene star of 2 regular languages
- Building regular languages:
  - Union: $L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$
    - Ex: // $L_1 = \{car, cat\}$, $L_2 = \{boat, bar\}$

      $L_1 \cup L_2 = \{car, cat, boat, bar\}$
  - Concatenation: $L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$
    - Ex: // $L_1 = \{car, cat\}$, $L_2 = \{boat, bar\}$

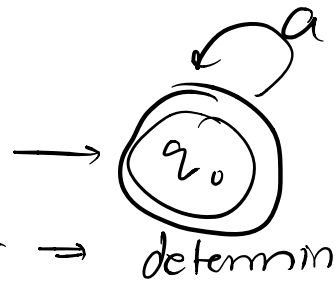
      $L_1 \cdot L_2 = \{carboat, carbar, catboat, catbar\}$
  - Kleene star: L^* ⇒ 0/more concatenating strings from L
    - Ex: // a^* ⇒ $\emptyset, a, aa, aaa \ldots$

- Ex: // $L = \{a, b\}$. $L^* = \{\epsilon, a, b, ab, abb, aab, \ldots\}$
- Best way to think about this: regex *

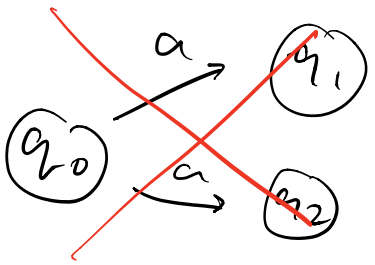
- Order of operations: *, ·, ∪
- Tips:
  1. If we ever need $a^n \Rightarrow a^*$
  2. Even/odd: consider $2n+1$

     Odd # of a: $b^* a b^* (a b^* a b^*)^* \Rightarrow$ a letter $2n+1$!

- Deterministic finite automata: state diagram for reg. L
  - Rules:
    - Self loops allowed:

      $a^* \Rightarrow$ 

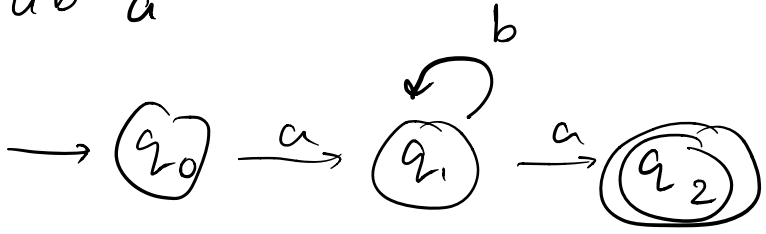
    - 1 transition / character $\Rightarrow$ deterministic



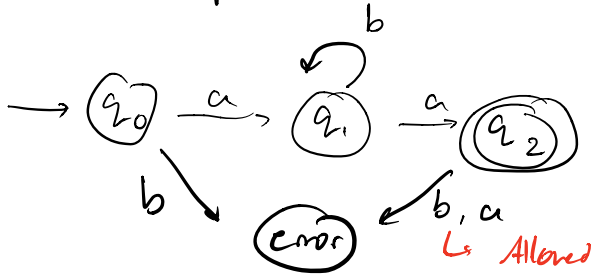
$\Rightarrow$ Non deterministic

  - Formal definition: DFA is a 5 tuple: $(\Sigma, Q, q_0, A, \delta)$
    - Σ: alphabet
    - Q: set of states (finite, non-empty)
    - q_0: starting state
    - A: set of accepting states ($A \subseteq Q$)
    - δ: $\delta(Q, w \in \Sigma) \rightarrow Q_2$ (way to transition state to state)
  - Recognition algorithm: $\delta^*(q_0, w) \in A$
    - Take word character by character and use transitions to determine whether word belongs in DFA

- Ex:// ab^*a

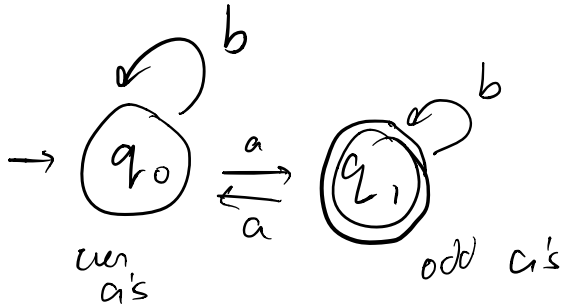


- Ex:// ab^*a w/ explicit error state ($\Sigma = \{a, b\}$)

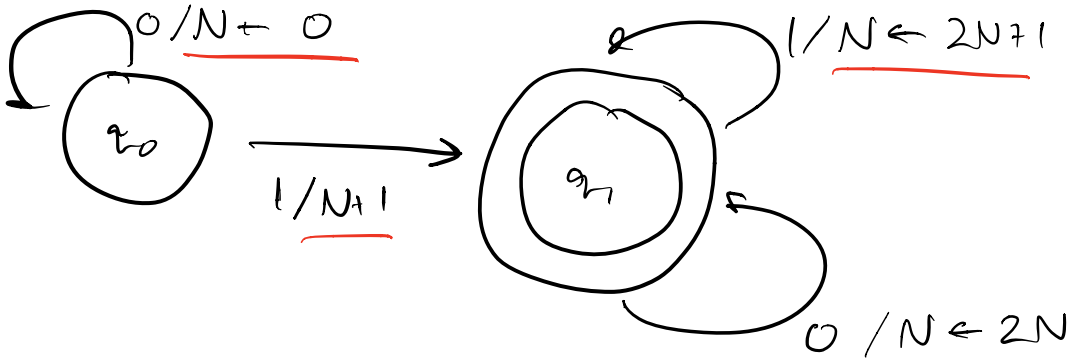


↳ Allowed: 1 transition for different letters

- Ex:// $L = b^*a(b^*ab^*a)^*$



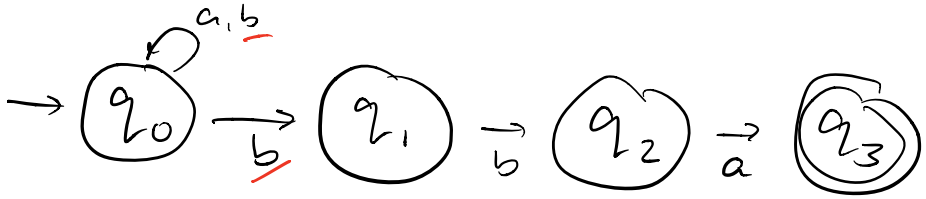
- L is regular ⇔ DFA for language
- Finite transducers: action / transition



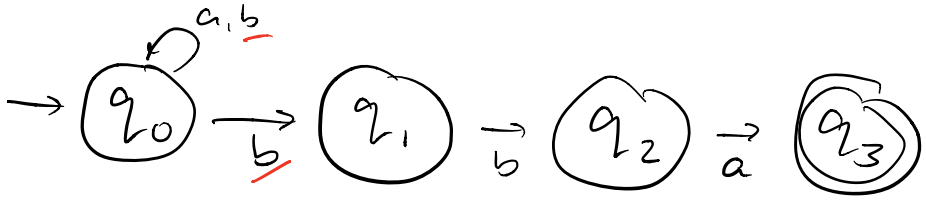
- Used in scanning stage of compiler

– Non-deterministic finite automata:

- Non-deterministic: multiple transitions for a character in a state.
- Ex://



- Formal definition: NFA is a $ tuple $(\Sigma, Q, q_0, A, \delta)$
  - Diff: δ: stack, character → set of states
- Recognition algorithm:
  1. Move to initial state
  2. Get set of states based on δ
  3. For each state in set of states, apply transition
  4. If end in set of states w/ accepting state, accept
  - Ex:// Determine if abbba will be accepted



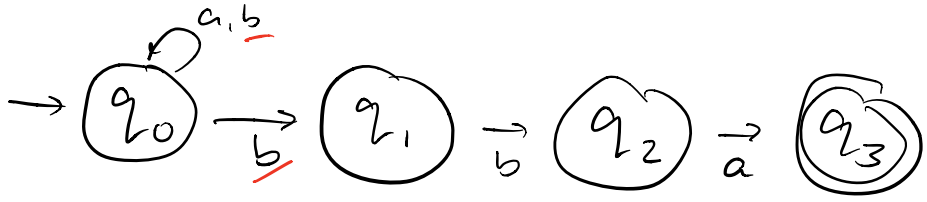
	Seen	Remaining	Set of States
1.	ϵ	abbba	$\{q_0\}$
2.	a	bbba	$\{q_0\}$
3.	ab	bba	$\{q_0, q_1\}$
4.	abb	ba	$\{q_0, q_1, q_2\}$
5.	abbb	a	$\{q_0, q_1, q_2\}$
6.	abbba	ϵ	$\{q_0, q_3\}$ ✓

  - L is regular ⇔ NFA represented

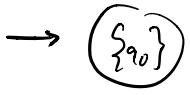
- NFA → DFA:
  - Basic idea: Create a state for each unique set of NFA states
  - Algo:
    1. Start w/ $S_0 = \{q_0\}$
    2. Note down resulting states for each character

3. Note down new set of states and make them accepting states if they contain an NFA accept state.

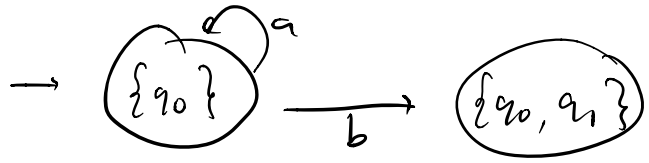
- Ex://



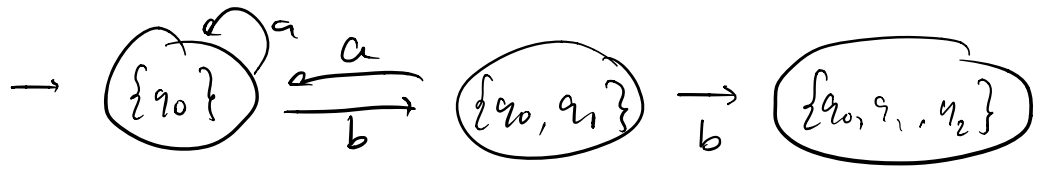
1.



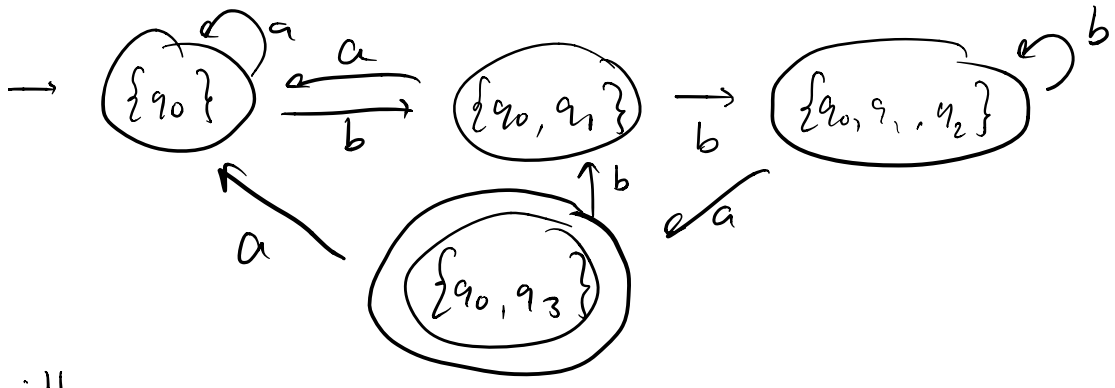
2.



3.



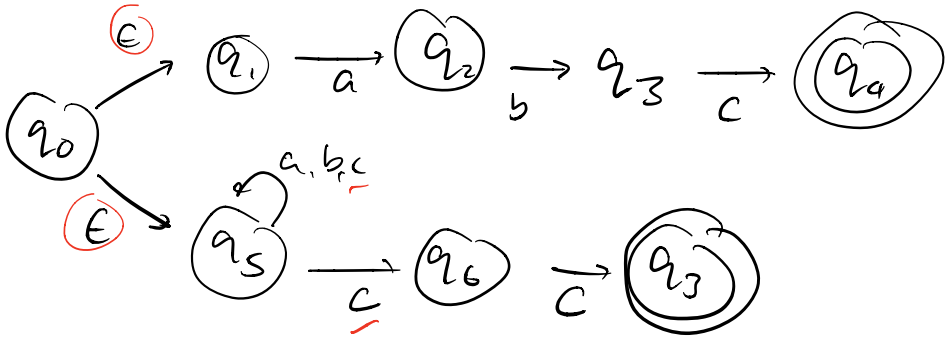
4.



- My algorithm:

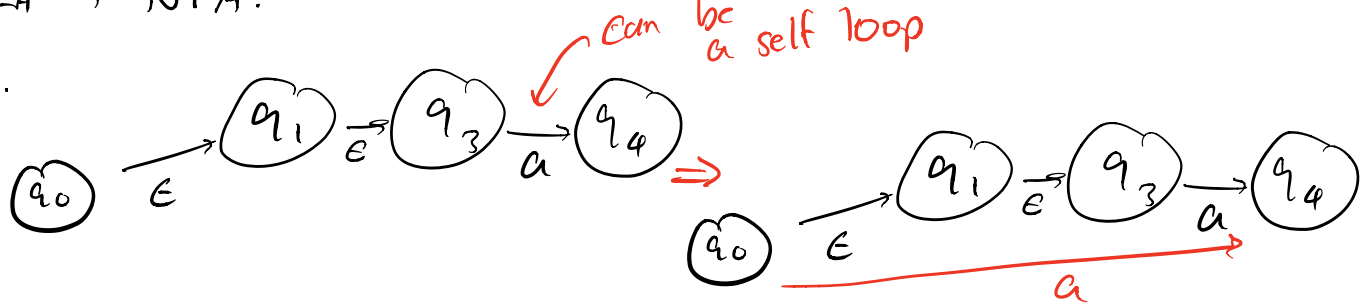
```
for (a ∈ Σ) {
    set = {}
    for (q ∈ S){
        set.add (δ (q, a))
    }
    if set already exists:
        transition to existing set
    else:
        draw a new state
}
```

- ϵ - Non-deterministic finite automata:

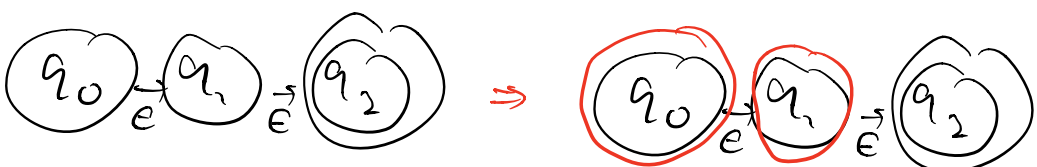
- ε transition: transition without character needed
- Usage: concatenate NFA
- Ex:// 

- Definition change: ε cannot be part of alphabet
- Epsilon closure: E(S) = set of states reachable from S using 0/more epsilon transitions.
- Recognition algorithm: look at epsilon closures for &

- ε NFA → NFA:

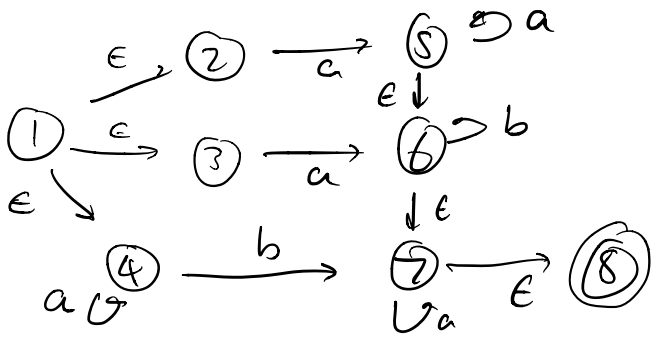
1. 

Given sequence of ε transitions ending w/ non ε transition, write a transformation from src to dest on last non ε trans.

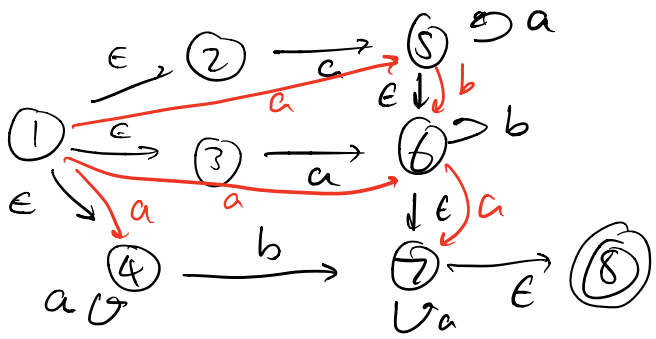
2. 

3. Remove ε transitions + unreachable states.

- Ex:// Convert the following to NFA:

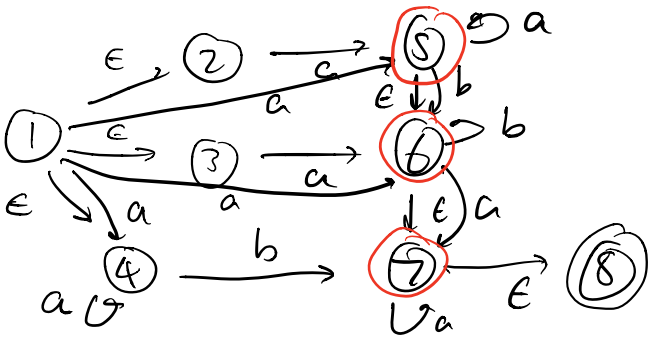


1.



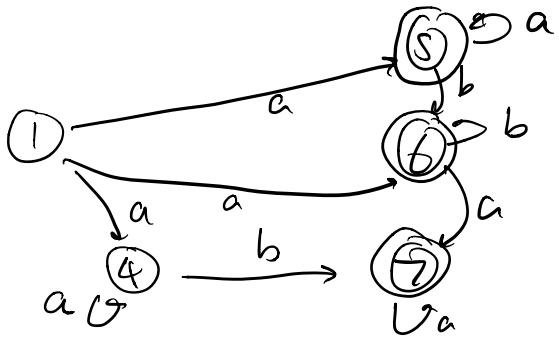
Edge creation

2.



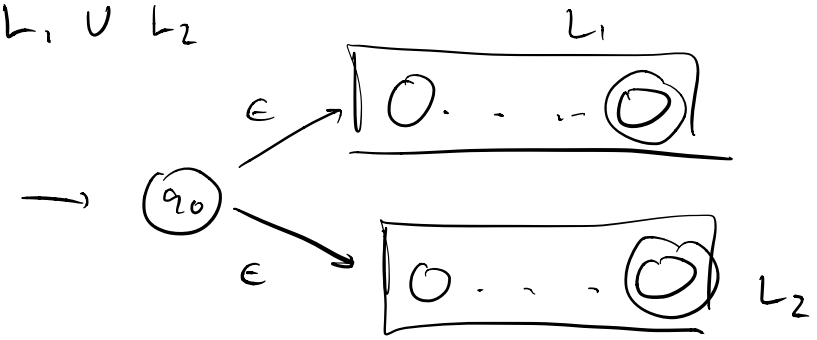
Accepting state propagation

3.

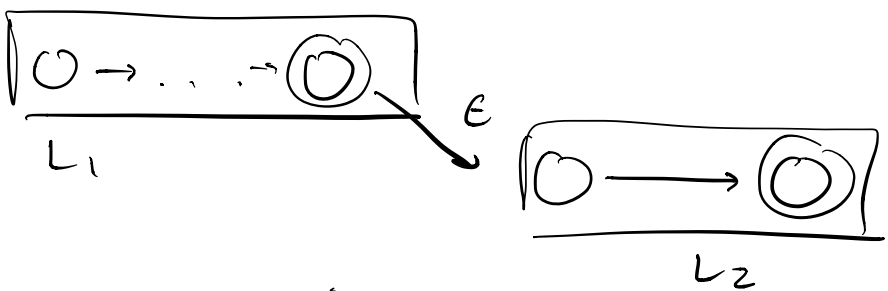


Cleanup

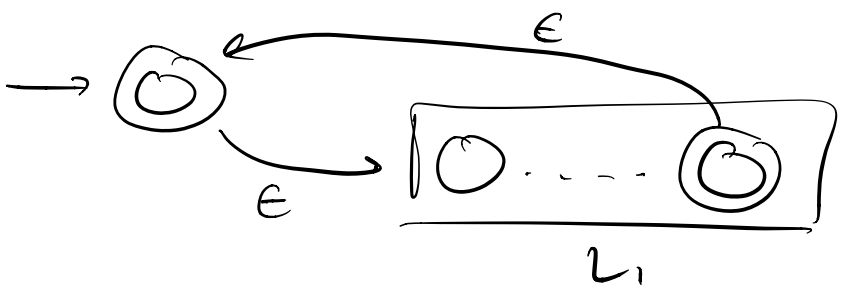
- Kleene's Theorem: showing regular languages → ϵ-NFAs
  - Union: $L_1 \cup L_2$



  - Concatenation: $L_1 L_2$



  - Kleene Star: L^*



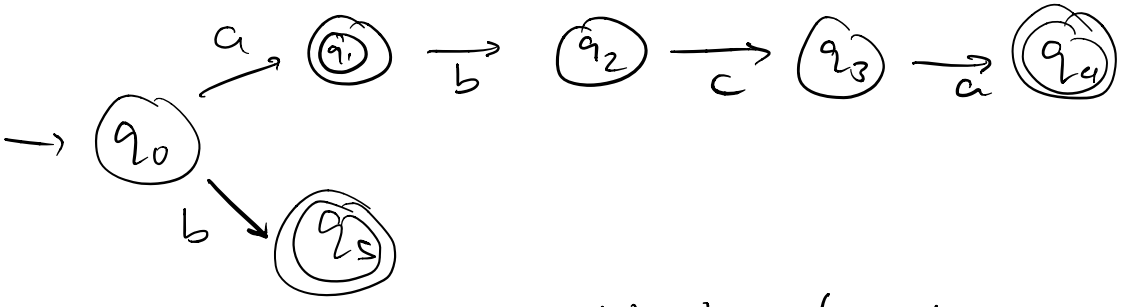
Scanning

- Use DFA to determine if word belongs to a language
  - Issue: no clarity on when to output token

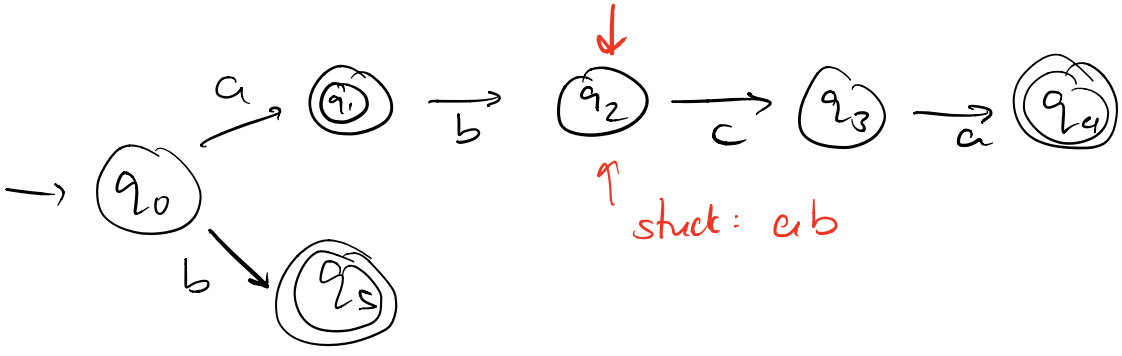
- Maximal munch algorithm: greedy algorithm.

$\Sigma = \{a, b, c\}$. $L = \{a, b, abcc\}$, $s = ababca$

① NFA:

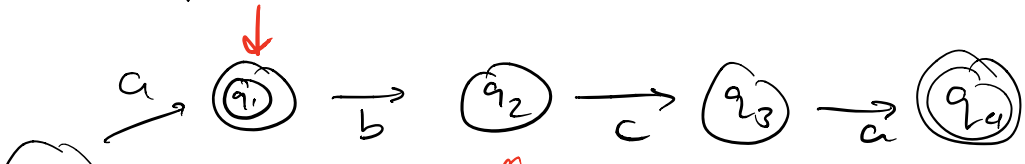


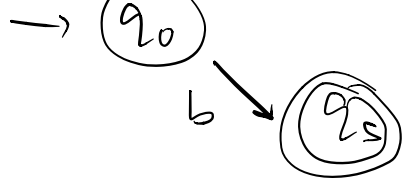
② Go as far as possible until stuck (no transitions left (end), or transitions don't match characters)



$s = abca$

③ Back to last emitted state + emit token





stuck: a

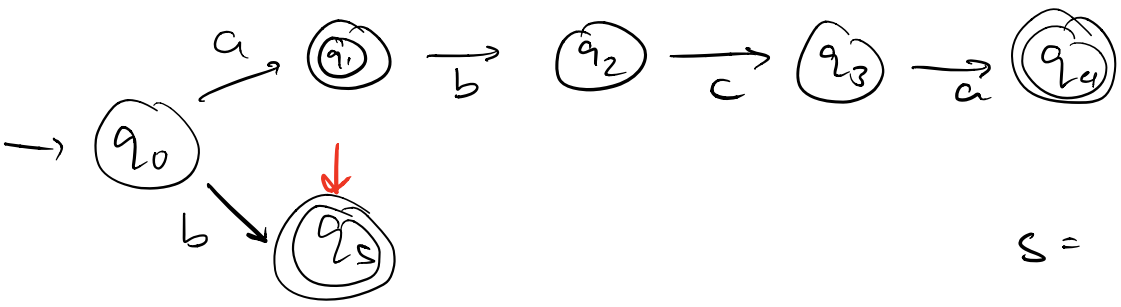
⇓

Remove character like backtracking

output = 'a'

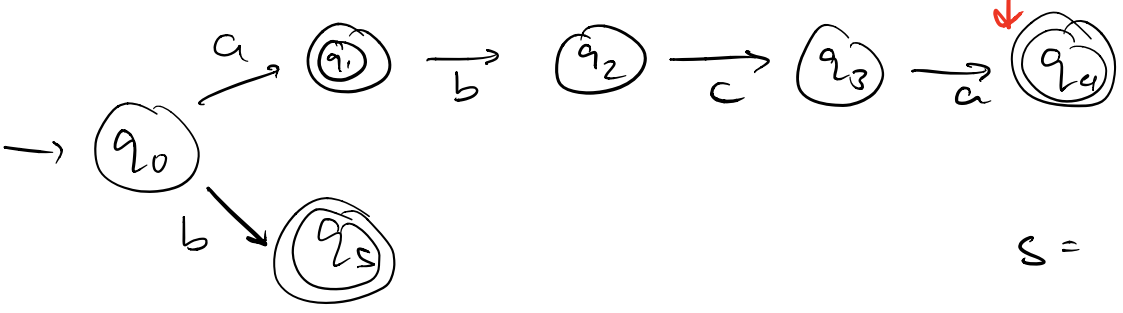
s = bcabca

④ Repeat until exhausted string



s = cbca

Output = 'a', 'b'

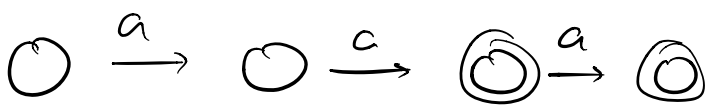


s = cbca

Output = 'a', 'b'

'cbca'

- Greedy:
  - Ex:



s = "aaaa" ⇒ Expected token: 'aa', 'aa'

Maximal much = 'aaa' → reject!

- $O(n^2)$ b/c of backtracking, optimization possible
- Real life:

```
int y = 10;
int *z = &y
int x = y /*z
```

Rejected:

/* = start of comment

- Simplied maximal much:
  - Don't back track ⇒ if stuck at some unaccepting state ⇒ reject
  - $O(n)$ ⇐ no backtracking

Context-Free Grammars

- DFAs are quite restrictive: nested statements, infinite states

Context Free Grammar

CFG

MIPS → [scanner] → [parser] → [semantic analysis]

- Formal defn: (N, T, P, S)
  - N: finite, $\not\exists$ of non-terminals symbols ⇒ can be expanded
  - T: alphabet (finite, $\not\exists$ of terminal symbols)
    ↳ Cannot be expanded
  - P: set of productions/rules

    Non terminals → $A \longrightarrow B$ ← Non-terminals/terminals

    Vocabulary $(N \cup T)$
  - S: start symbol

- CFG converting non-terminals (variables) into another expression via production rules

- Ex:// N: {expr}
  T: {ID, OP, LPAREN, RPAREN}
  P: {expr → ID, expr → expr OP expr, expr → LPAREN/RPAREN}
  S: {expr}

- Remix start symbol into another expression via production rules + continue this until reach goal
  - Ex:// ID OP LPAREN ID OP ID RPAREN in CFG?

expr ⇒ expr OP expr
⇒ expr OP LPAREN expr RPAREN
⇒ ID OP LPAREN expr OP expr RPAREN
⇒ ID OP LPAREN ID OP ID RPAREN

- Context-free language: all words that can be derived w/ CFG
  - Language is context-free $\Leftrightarrow$ $\exists$ CFG G such that $L = L(G)$
  - Regular language is also a context-free language
  - Ex:// $T = \{a, b\}$. Find a CFG that represents $\{a^n b^n : n \in \mathbb{N}\}$.
    Find a derivation for $aaabbb$

① Define parts of CFG

N: $\{S\}$ S: $\{S\}$

T: $\{a, b\}$

② Define our productions

$S \rightarrow aSb$
$S \rightarrow \epsilon$

Look @ patterns of our language + make it recursive

To combine: $S \rightarrow \epsilon \mid aSb$

Derivation:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$

Reduced Grammars

Non-reduced grammars

Rules that cannot be used

no way of arriving at B $\Rightarrow$ $B \rightarrow \alpha$

Repeating rules

$B \rightarrow B \mid \alpha$

Parse Trees

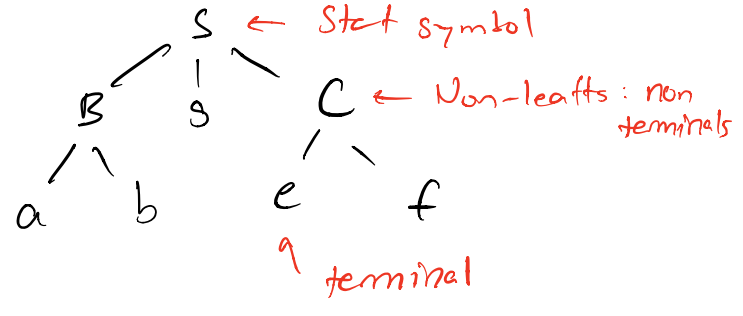
- Representation of derivation of a particular string
- Ex://

$N = \{S, B, C\}$

T: $\{a, b \ldots h\}$

P: $S \rightarrow BgC$
$B \rightarrow ab \mid cd$
$C \rightarrow h \mid ef$

Derivation of "dbgef"



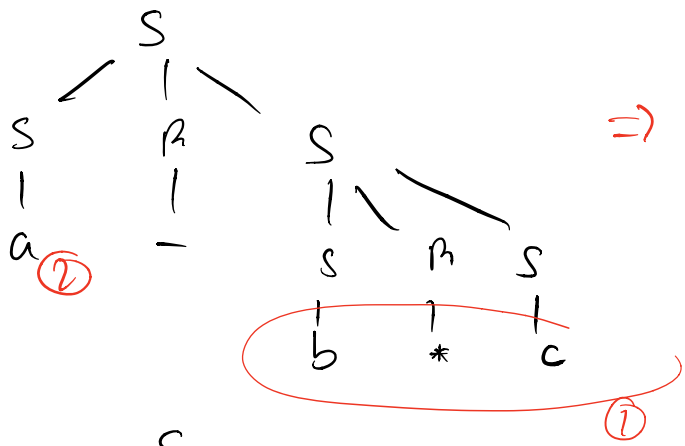
- Properties:
  1. A derivation uniquely defines a parse tree
     ↳ 2 diff. derivations ⇒ 2 diff. parse trees
  2. Multiple derivations for string ⇒ multiple parse trees for string
- Order of derivation matters:
  1. Leftmost derivation: expand left-most non terminal
  2. Rightmost derivation: " right most "

Ambiguous Grammars

- Ambiguous grammar: >1 leftmost/rightmost derivations
  - Ex: // $S \rightarrow a \mid b \mid c \mid SRS$
    $R \rightarrow + \mid - \mid * \mid /$

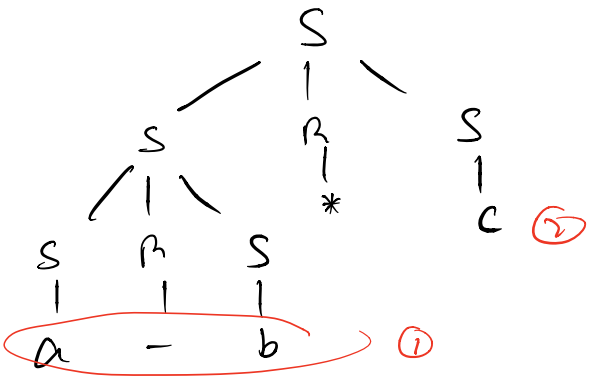
    $a - b * c$ (leftmost derivation)

1)



⇒ $a - (b * c)$

2)



⇒ $(a-b) * c$

  - Constructing meaning out of parse trees that are ambiguous is hard

- Ways to solve:

1. Use parantheses to force precedence
2. Make left/right associative based on location of nonterminal on RHS of production rule.

- 2nd method: remember, DFS traversal
  - Ex: // $a - b * c$

Right associative

$S \rightarrow L\ R\ S \mid L$

$L \rightarrow a \mid b \mid c$

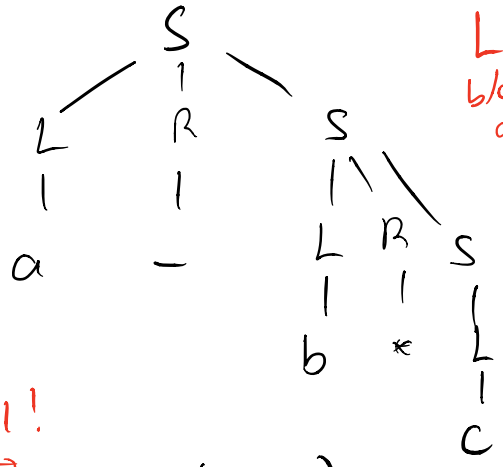
$R \rightarrow + \mid - \mid * \mid /$

Left associative

$S \rightarrow S\ R\ L \mid L$

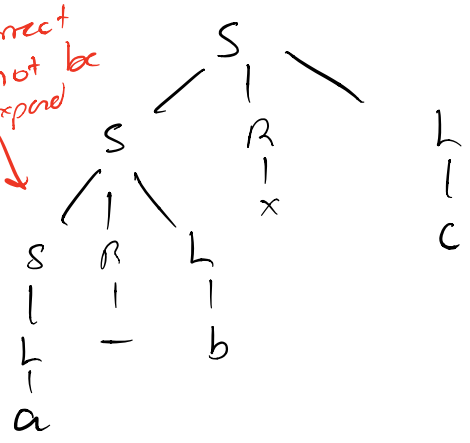
$L \rightarrow a \mid b \mid c$

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



DFS traversal! ⇒ $a - (b * c)$

L not correct b/c will not be able to expand



$(a - b) * c$

  - Reason why it works:
    1. Split recursive non-terminal s.t. it only leads to other non-terminals OR only terminals
    2. Order: expansive non-terminal in the desired position (left associative → S in left most, right assoc. → S in right)

- No guarantee that a non-ambiguous grammar exists for CFL
- No algorithm to determine if grammar is ambiguous
- No way of knowing that 2 CFGs represent same language

Parsing

- Parsing: find derivations ⇒ define parsing tree
- Method of derivations: pushdown automata
  - Not practical for converting different pushdown automata

Top Down Parsing

Start symbol ⟶ leftmost rules ⟶ show string in lang.

- Augmented grammar:
  - Start symbol has 1 production rule

CFG $G = \{N, T, P, S\} \Longrightarrow G' = \{N', T', P', S'\}$

$N' = N \cup \{S'\}$

$T' = T \cup \{\vdash, \dashv\}$

$P' = P \cup \{S' \rightarrow \vdash S \dashv\}$

- Ex// Non-augmented:

$S \rightarrow P$

$S \rightarrow T$

$\Longrightarrow$ Augmented Grammar

$S' \rightarrow \vdash S \dashv$

$S \rightarrow P$

$S \rightarrow T$

Algorithm

- Vague description
  1. Start w/ $S' \rightarrow \vdash S \dashv$
  2. Loop through all derivations + determine if non-term or term
     - i) If terminal: match terminal character w/ character at n
       - a) If match: move to next string character
       - b) No match: parse error
     - ii) Non-terminal: determine if correct derivation rule + substitute.

3. Repeat until no more non-terminals / error

- Ex:// 1) $S' \rightarrow \vdash S \dashv$ String: $\vdash aby wx \dashv$

2) $S \rightarrow A\ y\ B$

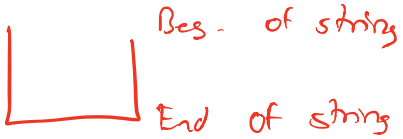
3) $A \rightarrow ab \mid cd$

4) $B \rightarrow z \mid wx$

1. $\alpha_1 = \vdash S \dashv$ ⇒ Used rule 1
2. $\alpha_2 = \vdash A\ y\ B \dashv$ ⇒ Used rule 2
3. $\alpha_3 = \vdash aby\ B \dashv$ ⇒ Used rule 3
4. $\alpha_4 = \vdash aby\ wx \dashv$ ⇒ Used rule 4

} Looked at which NT gives character needed

- How did we come up w/ appropriate rule?
  - Lookahead to find next unmatched character
  - Predict table: (unmatched char, NT) → derivation to apply
- Optimization: stack that has truncated part of string
  - String to match = read string + stack
- Optimized also:



1) Stack w/ ⊢ char
2) While TOS is a terminal, pop + match w/ string
   i) If match: move to next unmatched char
   ii) Else: error
3) If TOS is a non-terminal: pop and query Predict[A][a] (NT, unmatch char)
   i) If Predict[A][a] == rule: push symbols in reverse on stack
   ii) Else: error
4) Repeat until error / stack is empty

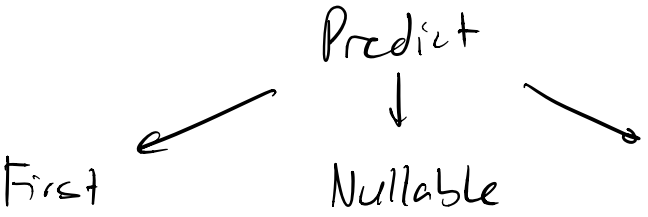
- LL(1):

LL(1)
- L: left → right
- L: leftmost derivation
- (1): # of lookahead

° LL(1) ⇔ Each cell in Predict contains ≤ 1 rule

Predict Table

- Formal: Predict [A] [a] ⇒ $A \to B$ s.t. $B \Rightarrow a\gamma$
  - Other words: derive a rule s.t. first character is lookahead



- First: $First(\beta) = \{a \in T' : \beta \Rightarrow a\gamma \text{ for some } \gamma \in V^*\}$
  - Set of first characters in β
  - Ex: $A \to ab \mid cd \mid zy \Rightarrow First(A) = \{a, c, d\}$
  - ∴ Predict [A] [a] = $\{A \to B : a \in First(B)\}$
- Dealing w/ nullable non-terminals

Nullable (β)

true ⇔ $B \Rightarrow^* \epsilon$

"can β go to ε"

Follow (A)

$\{b \in T' : S' \Rightarrow \alpha A b \beta \text{ for some } \alpha, \beta \in V^*\}$

"Set of terminals that occur right after A"

- ∴ Predict [A] [a] = $\{A \to B : a \in First(B)\} \cup \{A \to B : Nullable(B) \wedge a \in Follow(A)\}$
  - Non-nullable rules: find rules s.t. a is first char
  - Nullable rule: Find rule s.t. a follows nulled char

- Nullable construction:
  - Observations
    1. Nullable (ε): true
    2. Nullable (B): false if B has terminal
    3. Nullable (AB): Nullable (A) ∧ Nullable (B)

- Algo:
  1. Init Nullable (A) = false for all $A \in N'$
  2. Loop through rules:
     i) If: (P is $A \to \epsilon$) v (P is $A \to B_1 \dots B_n$ and $\bigwedge_{i=1}^{n}$ Nullable (B_i) = true):
     Nullable (A) = true
  3. Repeat until no changes

- Ex: //

1) $S' \to \vdash S \dashv$
2) $S \to b\ S\ d$
3) $S \to p\ S\ q$
4) $S \to C$
5) $C \to lC$
6) $C \to \epsilon$

Nullable

	Iter 0	Iter 1	Iter 2	Iter 3
S'	F	F	F	F
S	F	F	T	T
C	F	T	T	T

- First construction
  - Algo:
    1. Init First (A) = {} for all $A \in N'$
    2. For each rule $A \to B_1 \dots B_k$
       3. For each $i \in \{1, \dots, k\}$
          4. If $B_i \in T'$: First (A) = First (A) ∪ $\{B_i\}$. Break ⇒ B_i will be first terminal in rule b/c breaking to next rule
          5. Else: First (A) = First (A) ∪ First (B_i)
             i) Nullable (B_i) == False ⇒ break.
    6. Repeat until no change
       ↳ If B_i is N' and is the first + not nullable, First (A) += First (B_i)
  - Generalized algo to find First (B):
    1. Result = ∅
    2. for $i \in \{1, \dots, n\}$:
       3. If $B_i \in T'$: result = result ∪ $\{B_i\}$. break
       4. Else: result = result ∪ First $\{B_i\}$.
          5) Nullable (B_i) == false : break

- Ex: //

1) $S' \to \vdash S \dashv$
2) $S \to b\ S\ d$
3) $S \to p\ S\ q$
4) $S \to C$
5) $C \to lC$
6) $C \to \epsilon$

First

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

Fast way:

S'	t
S	b, p, l
C	l

1. Go through each rule + add first terminals
2. If rule has a non-terminal first, note for next pass
3. Add the terminal of non-terminal into First

- Follow construction
  - Also:
    1. Init Follow(A) = {} for all $A \in N$ ⇒ Not including S' !!
    2. For each rule $A \to B_1 \dots B_k$
       3. For each $i \in \{1 \dots k\}$
          4. If $B_i \in N$: Follow(B_i) = Follow(B_i) ∪ First($B_{i+1} \dots B_k$)
             5. If $\bigwedge_{m=i+1}^{k}$ Nullable(B_m) = True or i == k
                6. Follow(B_i) = Follow(B_i) ∪ Follow(A)
    7. Repeat until no change

Set of terminals that follow B_i

If $B_{i+1} \dots B_k \Rightarrow \epsilon$ or at end, use Follow(A)

$A \to B_i \dots B_k$
$S \to Az$
Follow(B_k) = z!

  - Ex:// 1) S' → t S t
    2) S → b S d
    3) S → p S q
    4) S → C
    5) C → lC
    6) C → ϵ

Follow

	0	1	
S	{}	{t, d, q}	=
C	{}	{t, d, q}	=

In Rule 4, C is last ⇒ Follow(C) = Follow(S)

- Predict table construction.
  - Also:
    1. Init P[A][a] = {} for all $A \in N'$, $a \in T'$
    2. For each rule A → B:
       3. For each $a \in$ First(B): add A → B in Predict[A][a]
       4. If Nullable(B):
          5. For each $a \in$ Follow(B), add A → B in Predict[A][a]

  - Ex:// 1) S' → t S t
    2) S → b S d
    3) S → p S q

	Nullable	First	Follow
S'	F	{t}	{}
S	T	{b, p, l}	{t, d, q}
C	T	{l}	{t, d, q}

4) $S \rightarrow C$

5) $C \rightarrow 1C$

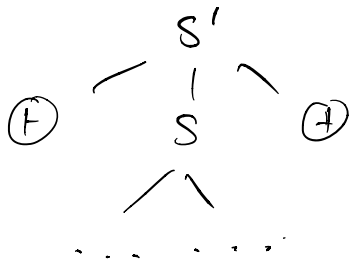
6) $C \rightarrow \epsilon$

		First	Follow
C	T	{1}	{+, d, $}

Predict:

	+	-	b	d	p	$	1
S'	1						
S		4	2	4	3	4	4
C		6		6		6	5

Parse Tree



⇒ Record derivations during algorithm

Limitations of LL(1) Grammar

- Ex:// 1) expr → expr op expr
  2) expr → id
  3) op → +

Predict[expr][id]:
- Rule 1: id ∈ First(expr)
- Rule 2: id ∈ First(expr)

⇒ ! LL(1)

- Grammar is LL(1) if:
  1. No 2 distinct rules s.t. same LHS → same first terminal

~~$A \rightarrow bS$~~
~~$A \rightarrow cS$~~

  2. No nullable symbol A has same terminal in both first + follow sets

~~$A \rightarrow aB$~~ ⇒ ~~First(A) = {a}~~
~~$A \rightarrow a$~~ ~~Follow(A) = {a}~~
~~$B \rightarrow AaC$~~

  3. Only 1 way to derive ϵ from nullable
  4. Left recursive algorithms not LL(1)

- To convert left recursive → right recursive
  1. Switch recursive statements s.t. $N \rightarrow N$ is on Right
     - Common prefix problem
       1) $S \rightarrow T + S$

2) $S \to T$

3) $T \to F * T$

4) $T \to F$

5, 6, 7) $F \to a \mid b \mid c$

$\Rightarrow$ Cannot decide on 1 or 2 b/c lookahead of 1 is not sufficient

2. Increase lookahead / factor

$A \to \alpha B_1 \mid \alpha B_2 \mid \ldots \Rightarrow$

$A \to \alpha B$

$B \to B_1 \mid B_2 \mid \ldots$

- Ex:// Left recursive

1) $S \to S + T$
2) $S \to T$
3) $T \to T * F$
4) $T \to F$

5, 6, 7) $F \to a \mid b \mid c$

① Right recursive

1) $S \to T + S$
2) $S \to T$
3) $T \to F * T$
4) $T \to F$

5, 6, 7) $F \to a \mid b \mid c$

② Factor:

1) $S \to TB$
2) $B \to +S \mid \epsilon$
3) $T \to FX$
4) $X \to *T \mid \epsilon$

5, 6, 7) $F \to a \mid b \mid c$

- Another way:

LEFT

$A \to A\alpha$

$A \to B$ (B does not start w/ A)

RIGHT

$A \to BA'$

$A' \to \alpha A' \mid \epsilon$

- Ex:// Left Recursive

1) $S \to S + T$
2) $S \to T$
3) $T \to T * F$
4) $T \to F$

5, 6, 7) $F \to a \mid b \mid c$

Right recursive

1) $S \to TS'$
2) $S' \to +TS' \mid \epsilon$
3) $T \to FT'$
4) $T' \to *FT' \mid \epsilon$

5, 6, 7) $F \to a \mid b \mid c$

BOTTOM UP PARSING

Input string $\xrightarrow{\text{derivations}}$ start symbol

Algorithm

- High level:

1. Read input symbols from left → right

2. If RHS of rule seen → replace w/ LHS of rule
3. Use a stack to keep track of converted input
4. Repeat until start symbol

Operations

- **Shift**: Consume symbol → push to stack
- **Reduce**: Pop RHS from stack → push LHS

- Ex:// 1. $S' \rightarrow \vdash S \dashv$
  2. $S \rightarrow A\ y\ B$
  3. $A \rightarrow ab \mid cd$
  4. $B \rightarrow z \mid wx$

  String: $\vdash aby\,w\,x \dashv$

Read	Unread	Stack	Actions
ϵ	$\vdash aby\,w\,x \dashv$		Shift $\vdash$
$\vdash$	$aby\,w\,x \dashv$	$\vdash$	Shift a
$\vdash a$	$by\,w\,x \dashv$	$\vdash a$	Shift b
$\vdash ab$	$y\,w\,x \dashv$	$\vdash ab$	Reduce ab
$\vdash ab$	$y\,w\,x \dashv$	$\vdash A$	Shift y
$\vdash aby$	$w\,x \dashv$	$\vdash A\,y$	Shift w
$\vdash aby\,w$	$x \dashv$	$\vdash A\,y\,w$	Shift x
$\vdash aby\,wx$	$\dashv$	$\vdash A\,y\,wx$	Reduce wx
$\vdash aby\,wx$	$\dashv$	$\vdash A\,y\,B$	Reduce $A\,y\,B$
$\vdash aby\,wx\dashv$	ϵ	$\vdash S$	Shift $\dashv$
$\vdash aby\,wx\dashv$	ϵ	$\vdash S\dashv$	Reduce $\vdash S\dashv$
$\vdash aby\,wx\dashv$	ϵ	S'	done

rightmost

- Observations

1. Derivation at any iteration = Unread + Stack
2. Acceptance : unread = ϵ $\wedge$ stack = [start]
3. Reverse derivation
4. Right most derivation

- Item: prod. rule w/ bookmark indicating how far we have read

  $S \rightarrow \cdot abC$, $S \rightarrow ab \cdot C$, $S \rightarrow abC \cdot$

  - Non-reducible: not finished item. Reducible: finished item

- Problem: how to determine whether to shift or reduce.
  - Soln: keep track of how far we are using items

$$E \begin{cases} E \rightarrow \cdot E + T \\ E \rightarrow E \cdot + T \\ + \begin{cases} E \rightarrow E + \cdot T \\ \vdots \end{cases} \end{cases} \qquad \begin{array}{l} S \rightarrow \cdot E - T \\ S \rightarrow E \cdot - T \\ S \rightarrow E - \cdot T \end{array}$$

  - Use a DFA to keep track of items
    1. Start state: fresh rule w/ start symbol

$$\rightarrow \boxed{S' \rightarrow \cdot + E +} \; q_i$$

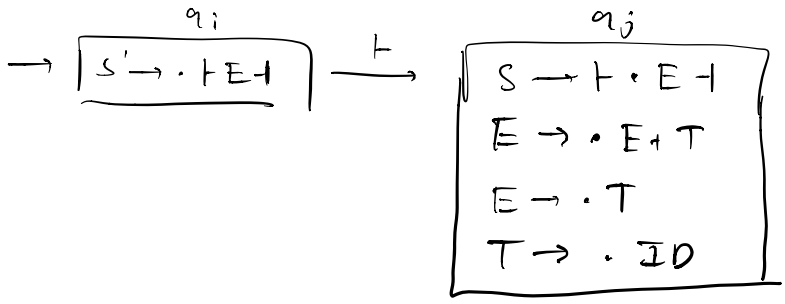
    2. Find state w/ non-reducible item + transition on symbol following bookmark to state q_j

$$\rightarrow \underbrace{\boxed{S' \rightarrow \cdot + E +}}_{q_i} \xrightarrow{+} \boxed{q_j}$$

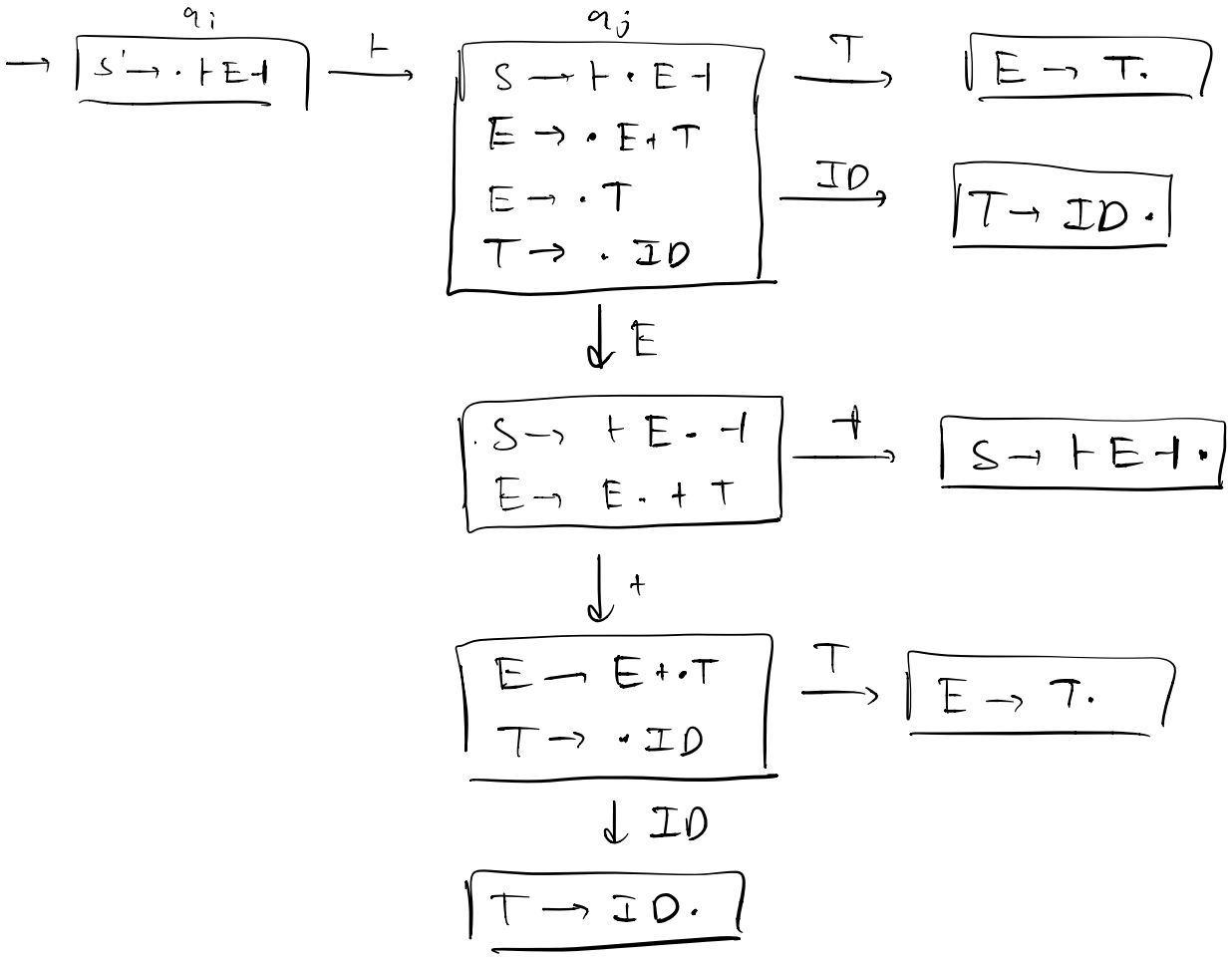
    3. Put all items in q_i w/ bookmark followed by symbol into q_j w/ bookmark updated

$$\rightarrow \overbrace{\boxed{S' \rightarrow \cdot + E +}}^{q_i} \xrightarrow{+} \overbrace{\boxed{S \rightarrow + \cdot E +}}^{q_j}$$

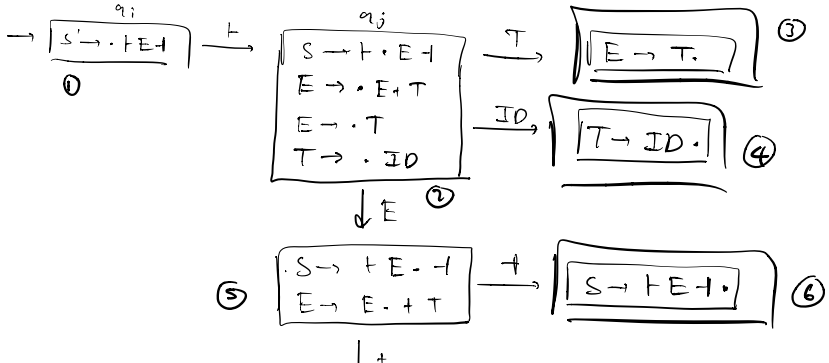
4. For all items in q_j: if bookmark followed by NT, add fresh items for non-terminal in q_j. Repeat

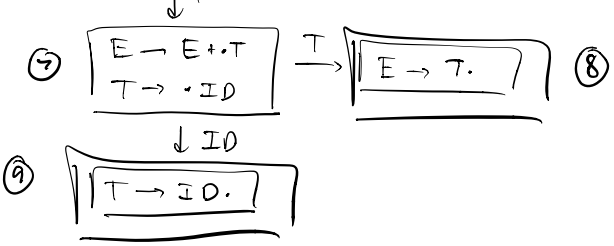


5. Repeat until no new states



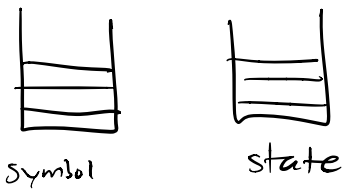
6. Accepting states: states that contain reducible item



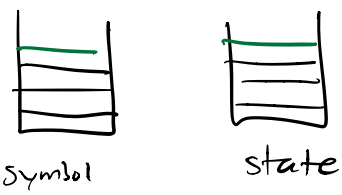


- Algorithm: run stack through DFA. If we land on reducible state → reduce. Else: shift
- Problem: have to run through whole stack every time
  - Soln: state stack (sequence of states till current state) + symbol stack (keeps track of derivations)

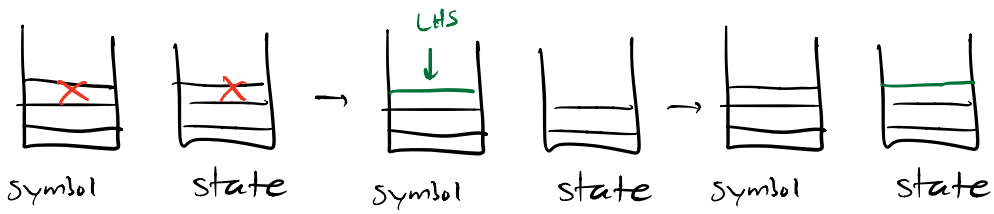
1. Query the state stack



2. State not reducible:
   Shift next input symbol + push new state



3. State is reducible
   a. Pop n items from state + symbol (n = # of RHS symbols)
   b. Push new symbol
   c. Push new state given symbol



- Ex: // + ID + ID -1

Read	Unread	Symbol	State	Action
ϵ	+ ID + ID-1		1	Shift on +
+	ID + ID-1	+	12	Shift on ID
+ ID	+ ID -1	+ ID	1 2 4	Reduce on 4
+ ID	+ ID -1	+ T	1 2 3	Reduce on 3
+ ID	+ ID -1	+ E	1 2 5	Shift on +
⋮	⋮	⋮	⋮	⋮

- LR(0) grammar: left → right, right recursive, no lookahead

Parsing Grammars

Conflicts

Shift-reduce

State in DFA where

$A \to \alpha \cdot a\beta$ ⇒ non-reducible

$B \to \gamma \cdot$ ⇒ reducible

Cannot decide to reduce w/out lookahead

Reduce-reduce conflict

2 reducible rules in state

⇓

Also doesn't know which rule to use in derivation

Using lookaheads

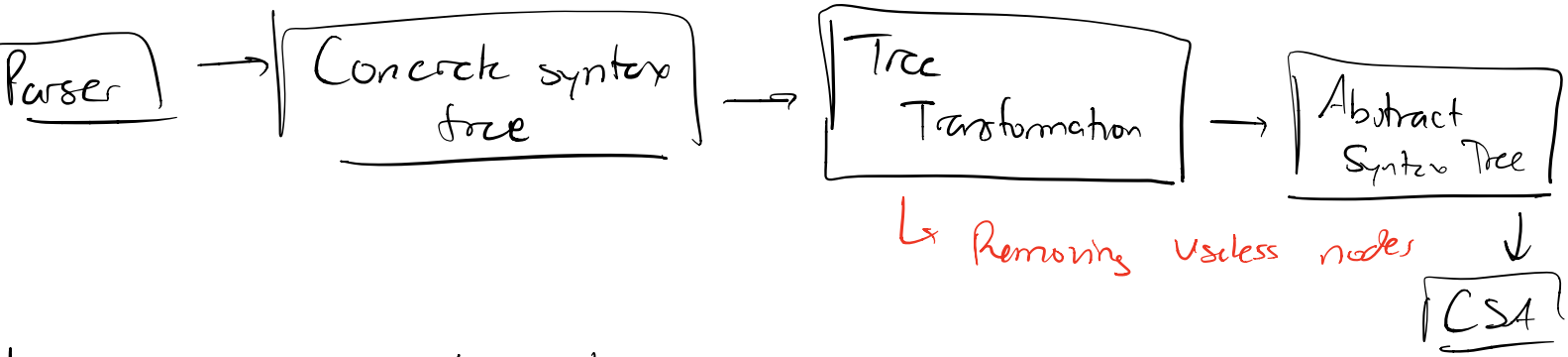
- Methods of solving conflict:
  1. SLR(1): use lookahead and check if in Follow set of LHS of reducible item
     a. Note down Follow sets for all reducible items
     b. Only reduce if lookahead is in Follow set
     - Limitation: if lookahead in Follow set of multiple rules in state
       - Soln: make Follow set local, i.e. ⊆ Follow that only applies to rule in state of question
  2. LR(1): exponential states if large grammar :(
  3. LALR(1): use locally computed follow sets
- Changes in implementation:
  1. DFA constructed accordingly
  2. Determine if state is reducible using current state + lookahead

CONTEXT-SENSITIVE ANALYSIS

Unique to your language! Following: WLP4

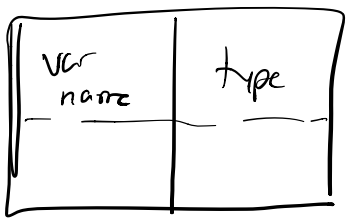
CFG problems

- Type checking
- Unique variables
- Usage before declaration
- Func calls



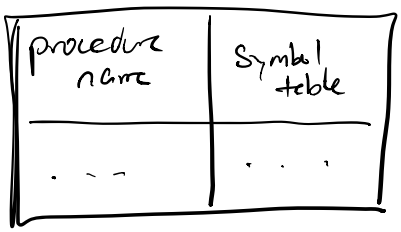
1. Unique variables / procedure names

Symbol Table



1. For each procedure → create symbol table
2. Add variable - type pairs in table

Master symbol table



Ensures unique procedure names!

Note: identifiers within a function with same name as function treated as a variable

```
∴ int p(int p) {
    return p(p);   ⇒ Would not compile
  }
```

Implementation:

1. Go through parse tree & check if producing procedure

   procedure → INT ID ...

   main → INT WAIN ...,

2. Check if procedure already defined (ID name in master table)

   i) Reject if already defined

3. Create new row in master symbol table & keep track of proc. symbol table

4. Continue parse tree. Look for variable declarations.

   dcl → type ID

5. Map in symbol table w/ type : ID lexeme

   i) Catch if already in symbol table

2. Usage before declaration

   Symbol table is finished before statements ← WLP4 spec.

   1. Look for factor → ID or lvalue → ID
   2. If ID lexeme in symbol table → ✓. Otherwise: catch.

3. Functions correctly called

   "correctly called" ← correct # of params. L/ matching types

   1. Function exists:

      a) Look for factor → ID LPAREN RPAREN

         OR

         factor → ID LPAREN arglist RPAREN

      b) Does ID lexeme exist in master symbol?

2. Parameters match:

Masto symbol table:

name	< [arglist types], [symbol table] >
⋮	⋮

Ex:// int foo (int * a, int b) { . . . }

foo	< [int*, int], { . . . } >

To figure out arglist types:

paramlist → dcl OR paramlist → dcl COMMA paramlist

↓

type + add into symbol table

4. Types

Create typechecker for every single production

Ex:// typecheck (statement → lvalue BECOMES expr SEMI) {

ltype := typecheck (lvalue)

exprtype := typecheck (expr) } → Recursively find type

if ltype != exprtype: return error → Check

}

⇓

Not returning a type! Statements have no types

int main () {

int a = b + 10; ← expr

↑ statement

```
typecheck (expr → expr PLUS term) {
    ltype := typecheck (expr)
    rtype := typecheck (term)        } Recursively find types
    Checks { if ltype == int & rtype == int ⇒ return int
             if ltype == int* & rtype == int | ltype == int &
                rtype == int* ⇒ return int*
}
```

Procedure calls

1. Check if function exists
2. Check function type
3. Check params + type match

Booleans: well-typed if operand types are same

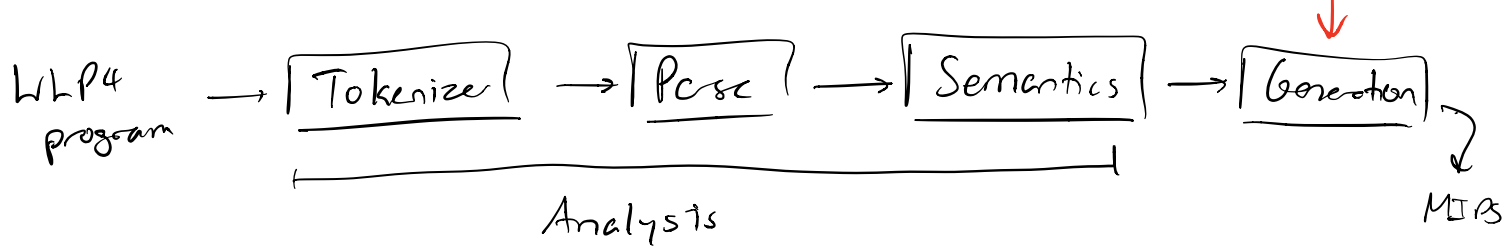
Control: well-typed if Boolean + child statements are well typed

Other checks

- Variables initialized
- Exhaust all possible returns
- Dead code check

} Do not happen in WLP4

CODE GENERATION



Variables

- Convention:

```
int main (int a, int b) {
    :
    return ____ ← $3
}
```

($1 ⇓ a, $2 ↓ b)

- Issue: mapping between variables + register
  - Extend symbol tables

Name	Type	Location
a	int	$1
b	int	$2

- Issue: limited # of registers → limited # of variables
  - Keep track of stack offset for each variable

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

⇒

```
lis $4          } convention
.word 4
sw $1, -4 ($30)     } store on stack
sw $2, -8 ($30)
sub $30, $30, $4    } update stack ptr
sub $30, $30, $4
lw $3, 4($30)   → load into return register
:
<restore>
```

⇒

Name	Type	Offset
a	int	4
b	int	0

- Issue: $30 (stack ptr) constantly moving
  - To use $29 (frame pointer): rel. position of stack for procedure
  - Use offsets from $29 rather than $30

Expressions

- Issue: expressions need temp. registers for storage
  - Use stack! Push everything stack ⇒ pop off sequentially to perform nested operations

- Ex: // int wain (int a, int b) {
  return a-b;
  }

  ⇒

  code (a) ⇒ load a into $3
  push ($3)
  code (b) ⇒ load b into $3
  pop ($5) ⇒ a is in $5
  sub $3, $5, $3

- Ex: // int wain (int a, int b) {
  int c = 3;
  return a + (b-c);
  }

  ⇒

  code (a)
  push ($3)
  code (b)
  push ($3)
  code (c)
  pop ($5)
  sub $3, $5, $3
  pop ($5)
  add $3, $5, $3

- Rule conversions:

1. S → BOF statements EOF:

   Code (S) = code (statements)

2. expr → term:

   code (expr) = code (term)

3. factor → LPAREN expr RPAREN

   code (factor) = code (expr)

4. $expr_1$ → $expr_2$ PLUS term

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

5. statement → lvalue BECOMES expr SEMI; lvalue → ID

   code (statement) = code (expr) + sw $3, offset ($29)

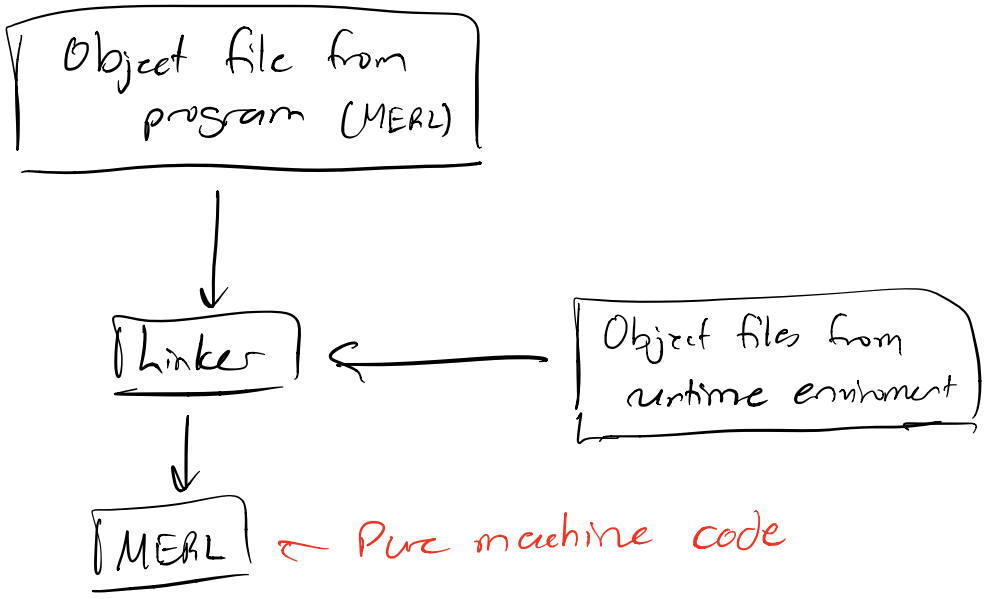
   ↳ ID of symbol table

Print Statements

2 options

- Create print procedure + call everytime
- Import print procedure from runtime environment.

- Runtime environment: exec. environment provided by OS that assists



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

print call

Comparisons

- test → $expr_1 < expr_2$: 1 if true, 0 if false

  code(test) = code($expr_1$) + push ($3) + code($expr_2$) + pop ($5)
  + slt $3, $5, $3

  - Do the same for test → $expr_1 > expr_2$

- test → $expr_1$!= $expr_2$:

code(test) = code(expr$_1$) + push ($3) + code(expr$_2$) + pop ($5)
+ slt $6, $3, $5 + slt $7, $5, $3 + add $3, $6, $7

- For equality: sub $3, $11, $3 ($11 = 1 ⇒ flips)

- >= ⇒ !(<), <= ⇒ !(>)

Control flow statements

- statement → IF (test) {stmts1} ELSE {STMTS2}

code (statement) = code (test) + beq $3, $0, else + code(stmts$_1$)
+ beq $0, $0, endit + else: + code(stmts$_2$)
+ endit:

- To have unique labels, use a counter + append to label

- Statement → WHILE (test) {stmts}

Code (statement) = loop: + code(test) + beq $3, $0, endWhile
+ code (stmts) + beq $0, $0, loop + endWhile:

Pointers

- NULL: misaligned address ⇒ add $3, $0, $1 (0x01)
- Dereferencing ptr:

factor$_1$ → STAR factor$_2$ ⇒ code (factor$_1$) = code (factor 2)
+ lw $3, 0($3)

- Get address:
  1) factor → AMP lvalue (lvalue is ID)
  ⇒ code(factor) = lis $3 + .word [offset From Table] + add $3, $3, $29
  (Addr)
  2) factor → AMP lvalue (lvalue is STAR factor$_2$)
  code (factor1) = code (factor 2)

- Assignment via ptr dereference:

stmt → lvalue BECOMES expr SEMI (lvalue → STAR factor)

code(stmt) = code(expr) + push($3) + code(factor) + pop($5)
+ sw $5, 0($3)

- Comparison: ptrs can be neg ⇒ sltu

- Arithmetic:

1) $expr_1 \rightarrow expr_2 + term$ where type(term) == int + type($expr_2$) == int*

code($expr_1$) = code($expr_2$) + push($3) + code(term) +
mult $3, $4 + mflo $3 + pop($5)
+ add $3, $5, $3

2) $expr_1 \rightarrow expr_2 + term$ but opposite types

Switch order ⇒ multiply expr × 4 ⇒ + to term

3) $expr_1 \rightarrow expr_2 - term$ where type($expr_2$) == int*, type(term) = int

$exp_2 - (4 \times term)$

4) $expr_1 \rightarrow expr_2 - term$ but types switched

$$\frac{exp_2 - term}{4}$$

Heap Allocation

MERL file: alloc.merl

init

Initializing data struct.

new

Allocates mem.
$1: # of word to alloc

delete

Deallocs
$1: addr of block

$2: length of array (or 0) $3: addr of block

- new int [expr]

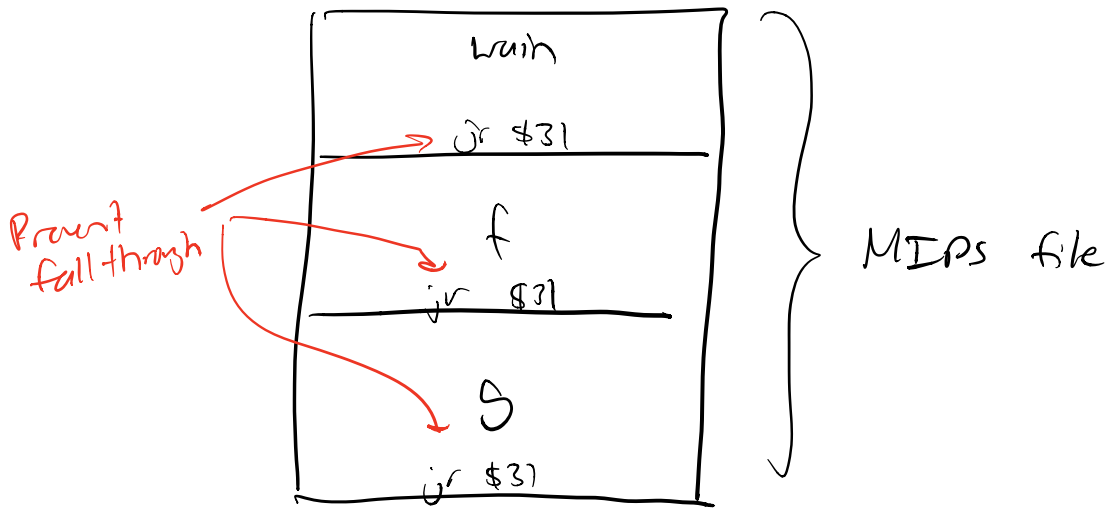
code(expr) + add $1, $3, $0 + push($31) + lis $5 + .word new + jalr($5) + pop($31) + bne $3, $0, 1 + add $3, $1, $0

NULL

- delete [] expr

code(expr) + beq $3, $11, skipDelete + add $1, $3, $0 + push($31) + lis $5 + .word delete + jalr $5 + pop($31) + skipD

Other procedures



- Wain procedure:
  1) Call imports + initialize
  2) Save $1 + $2 on stack
  3) Code ⇒ reset stack ⇒ exit
- Saving + restoring registers: either caller or callee-saved
  - $1-7 needs to be saved. $30 needs to be restored
  - $29:

Callee-saved: push ($29) + add $29, $30, $0

Caller-saved: save $29 before proc. called

- Saving parameters: caller-saves
  - factor → ID (expr1, ..., exprN)

code(factor) = push ($29) + push ($31) + code(expr1) + push($3)
+ ... + code(exprN) + push ($2) + li $5
+ load ID + jalr ($5) + pop n times + pop ($31) + pop ($29)

- Procedure code:

procedure → INT ID(params) {dcls stmt RETURN expr};

code(procedure) = ID:
+ sub $29, $30, $4
+ push registers to save
+ code(dcls)
+ code(expr)
+ pop saved reg.
+ add $30, $29, $4
+ jr $31

  - Problem: params + local variable memory is non contingous

1) Param i at 4(n-i+1), n is # of param
   Local var i -4r - 4(i-1), r is # of saved registers
2) Switch code order
3) Caller save

- Labels: F_ funclabel