M2

Steps to run a program	
1	**Loader** loads program => RAM
2	**PC** = startAddress of program in RAM
3	**Fetch-Execute Cycle** **IR** = MEM [**PC**] **PC** += 4 Execute **IR**

Instruction		Notes
lis $d	1.$d = MEM[PC] 2.PC += 4	1. $d = treats whatever comes after as a 32 bit binary (instruction or not) 2. Skips to the instruction after
div $s, $t divu $s, $t	$s / $t	**lo** = quotient **hi** = remain
beq $s, $t, i bne $s, $t, i	PC+= i*4	positive i: skip i instructions (from branch) negative i: go back |i|-1 instructions (from branch)
slt $d, $s, $t sltu $d, $s, $t	$d = 1 if $s < $t = 0 otherwise	
jalr $s	1.$31 = PC 2.PC = $S	1. $31 = current PC of this instruction (the next instruction) 2. Sets PC to address in $s

Immediate (i)	
branch	decimal, hex, label
word	decimal, hex, label
load/store	decimal, hex

Label Branch Formula: Offset = (LabelAddress − PC) / 4

- where PC is 1 after branch

Example

```
0x00    lis $2
0x04    .word 13
0x08    lis $1
0x0c    .word -1
0x10    add $3, $0, $0
      loop:
0x14    add $3, $3, $2
0x18    add $2, $2, $1
0x1c    bne $2, $0, loop
0x20    jr $31
```

PC = 0x20 = 32

LabelAddress = 0x14 = 20

Offset = 12 / 4 = -3

Input	**Output**
lw ← 0xffff0004 1. Reads one byte (8 bits) 2. Store the byte in the destination register (padded with 0s to turn it into a 32-bit word 3. -1 if error	sw → 0xFFFF000C 1. LSB of register ⇒ standard output

Procedures

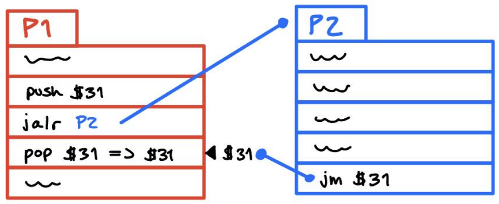
3 Rules

Calling a Procedure: store/restore $31

A Procedure Should:

1. stores/restore parameters
2. stores/restore local variables

Calling a Procedure

 P1 push $31 jalr P2 pop $31 => $31 $31 P2 jm $31	**push $31** ***jalr*** procedure [PROCEDURE RUNS] **pop $31**

Recursive Procedures

Example

```
; input:  $1 = non-negative integer
; output: $3 = n!
```

```
n! = n x (n-1) x ··· x 1

   base : 1
 recursive : m! = m x (m-1)!
```

```
factorial:

   lis $3  .word 4

    Sw  $1, -4($30)        ; Save $1
    Sub $30, $30, $3       ; update $30

   base case
    lis $3   .word 1

    bne $1, $3, recursive

     jr $31

    recursive:

      Sub $1, $1, $3              ; $1= $1-$1

       Sw $31, -4(30)     sub $30, $30, 4

          lis $3    .word factorial
          jalr $3

      lw $31, 0(30)      add $30, $30, 4


     moltu $1, $3                ; $1 • ($1-1)!

     mflo $3
    end

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

      jr $31
```

Assembler

Tokens ⇒ Binary

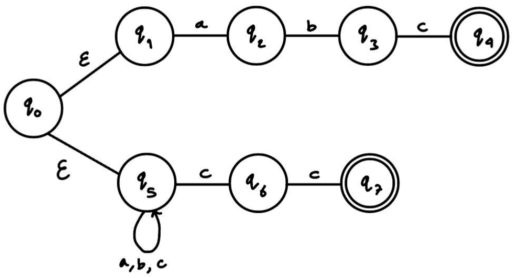
1) Encoding	1. Convert Each **Token** to binary 2. Shift Each binary into position 3. **bitwise(|)** and/or **mask** the result
2) Output	Output Each Word, 1 byte at a time: 1. instr >> 24, 16, 8, 0 2. cout << instr

M3

ε-NFA Recognition

Example

$L = \{abc\} \cup \{\text{ends with } cc\}$



Input: abcaccc

Seen	remaining	S
ε	abcaccc	$\{q_0, q_1, q_5\}$
a	bcaccc	$\{q_2, q_5\}$
ab	caccc	$\{q_3, q_5\}$
abc	accc	$\{q_4, q_5, q_6\}$
abca	ccc	$\{q_5\}$
abcac	cc	$\{q_5, q_6\}$
abcacc	c	$\{q_5, q_6, q_7\}$
abcaccc	ε	Accept

0+ ε closures

Note: zero or more epsilon transitions!
Note: look at step 2

Scanning

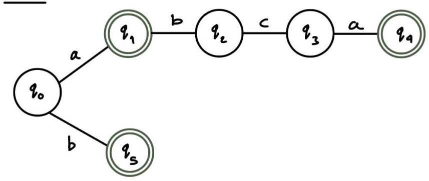
Runs on a DFA

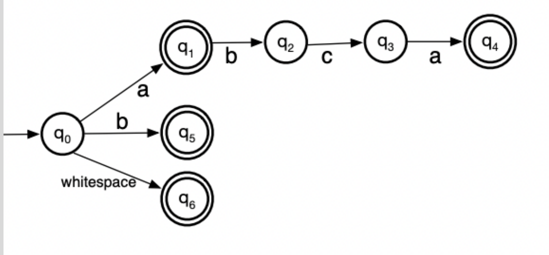
DFA	MAX	SIMPLIFIED
Consumer Each Letter		
Accept State?	1. **flag** as last accept-state	**Be Greedy** (keep going)

	2. **Be Greedy** (keep going)	
Stuck or EOF?		
Otherwise	**Stuck:** 1. **Backtrack** to last seen accept-state 2. **Un-Consume + output** input that was consumed after *accept* 3. **Reset** to start state **EOF: reject**	**reject**
Accept State	1. **Output** what was consumed 2. **Reset** to start state	3. **Output** what was consumed 4. **Reset** to start state

$\Sigma = \{a,b,c\}$, $L = \{a, b, abca\}$

DFA:

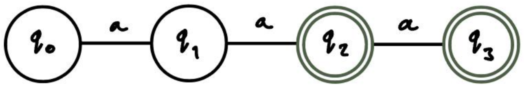




Input	Maximal	Simplified
ababca **a_b_abca**	1. a, b, abca	1. a, error 2. a, _, b, _, abca
baba **ba_ba**	1. b, a, b, a 2. b, a, _, b, a	1. b, error 2. b, a, _, b, a

Ex

$L = \{aa, aaa\}$



Input	Maximal	Simplified
aaaaa	aaa, aa	aaa, aa
aaaa	aaa, error	aaa, error

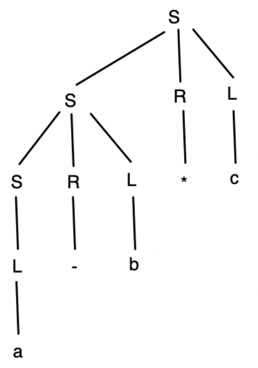
M4

Associativity: When operations have the same precedence, do we associate the left or right terms?

Left	Right
(x-y) + z	x-(y+z)

Associativity vs Recursive

Left recursion	Left associativity
property of the grammar's production rules	**Specifies** order that operators of the same precedence.
"We use left recursion to enforce left associativity in our arithmetic" **left recursive ⇒ left associative** **right recursive ⇒ right associative**	

Example		
Left Recursive: S → SRL | L L → a | b | c R → + | − | ∗ | /	**a-b*c:** • S ⇒ SRL • ⇒ SRLRL • ⇒ aRLRL • ⇒ a-LRL • ⇒ a-bRL • ⇒ a-b*L • ⇒ a-b*c	 S S R L S R L * c L - b a

Other Ways to avoid Ambiguity

Method 1 - Precedence

Force the language syntax to require parentheses

Ex

S → a | b | c | (SRS) R → + | - | * | /	
S ⇒ (SRS) ⇒ ((SRS)RS) ⇒ ((aRS)RS) ⇒ ((a-S)RS) ⇒ ((a-b)RS) ⇒ ((a-b)*S) ⇒ ((a-b)*c)	S ⇒ (SRS) ⇒ (aRS) ⇒ (a-S) ⇒ (a-(SRS)) ⇒ (a-(bRS)) ⇒ (a-(b*S)) ⇒ (a-(b*c))
(a-(b*c))	((a-b)*c)

Method 3 - BEDMAS

Make higher precedence (*, /) appear further down the tree:

S → SPT | T T → TRF | F F → a | b | c |(S) P → + | − R → ∗ | /	S → S + T | T T → T ∗ F | F F → a | b | c | (S)

M5

Parsing Algorithms	
Input	1. CFG 2. Tokens
Steps	**For Each Word(w)** ● Derivation for w ⇒ w is in the language ● No derivation ⇒ w is not in the language ⇒ error
Output	**Parse Tree (leftmost derivation)**

Predict Table

Predict[A][a] : given a non-terminal A and a lookahead terminal a, will predict which rule to choose for A

<table>
<tr><td></td><td></td><td>
① $S` \to \vdash S \dashv$
② $S \to bSd$
③ $S \to pSq$
④ $S \to C$
⑤ $C \to 1C$
⑥ $C \to \varepsilon$
</td></tr>
<tr><td><b>Nullable(A)</b></td><td><b>A</b> ⇒* ε</td><td>
<table>
<tr><th>Iteration</th><th>0</th><th>1</th><th>2</th><th>3</th><th></th></tr>
<tr><td>S`</td><td>F</td><td>F</td><td>F</td><td>F</td><td>$S` \to \vdash S \dashv \to \vdash C \dashv \to \vdash \varepsilon \dashv$</td></tr>
<tr><td>S</td><td>F</td><td>F</td><td>T</td><td>T</td><td>$S \to C \to \varepsilon$</td></tr>
<tr><td>C</td><td>F</td><td>T</td><td>T</td><td>T</td><td>$C \to \varepsilon$</td></tr>
<tr><td></td><td>↑
Start all fals</td><td>↑
directly</td><td>↑
2 rules</td><td>↑
3 rules</td><td></td></tr>
</table>
</td></tr>
<tr><td><b>First(A)</b></td><td><b>Look at the RHS of rule: A → BCD…</b>
If B is nullable: add First(C)</td><td>
<table>
<tr><th>Iteration</th><th>0</th><th>1</th><th>2</th></tr>
<tr><td>S`</td><td>{ }</td><td>{⊢}</td><td>{⊢}</td></tr>
<tr><td>S</td><td>{ }</td><td>{b,p}</td><td>{b,p,1}</td></tr>
<tr><td>C</td><td>{ }</td><td>{1}</td><td>{1}</td></tr>
<tr><td></td><td>↑
Start</td><td>↑
directly</td><td>↑
2 rules</td></tr>
</table>
</td></tr>
<tr><td><b>Follow(A)</b></td><td><b>Look at the RHS of rule:</b> C → …A…
$C \to \cdots A \cdots$
① $C \to \cdots A a$ (add a)
② $C \to A \underbrace{B_1 B_2 B_3}_{\text{nullable}} D \ || \ AD$ (add First(D))
③ $C \to A \underbrace{B_1 B_2 B_3}_{\text{nullable}} \ || \ \cdots A$ (add Follow(C))
</td><td>
<table>
<tr><th>Iteration</th><th>0</th><th>1</th></tr>
<tr><td>S</td><td>{ }</td><td>{⊣,d,q}</td></tr>
<tr><td>C</td><td>{ }</td><td>{⊣,d,q}</td></tr>
<tr><td></td><td>↑
Start</td><td></td></tr>
</table>
</td></tr>
</table>

Predict Table

rule # n: A ⇒ B

1) A : Add First(B)
2) A : If Nullable(B)
    a) add Follow(A)

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

LL(1)

- Top-Down Parsing Algorithm
- 2 ways to get ERROR
    - no match
    - Predict[A][a] = 0 or 2+ rules

Predict Table:

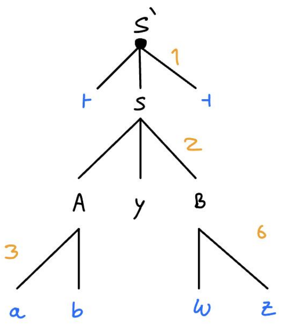
1) S` → ⊢S⊣
2) S → AyB
3) A → ab
4) A → cd
5) B → z
6) B → wx

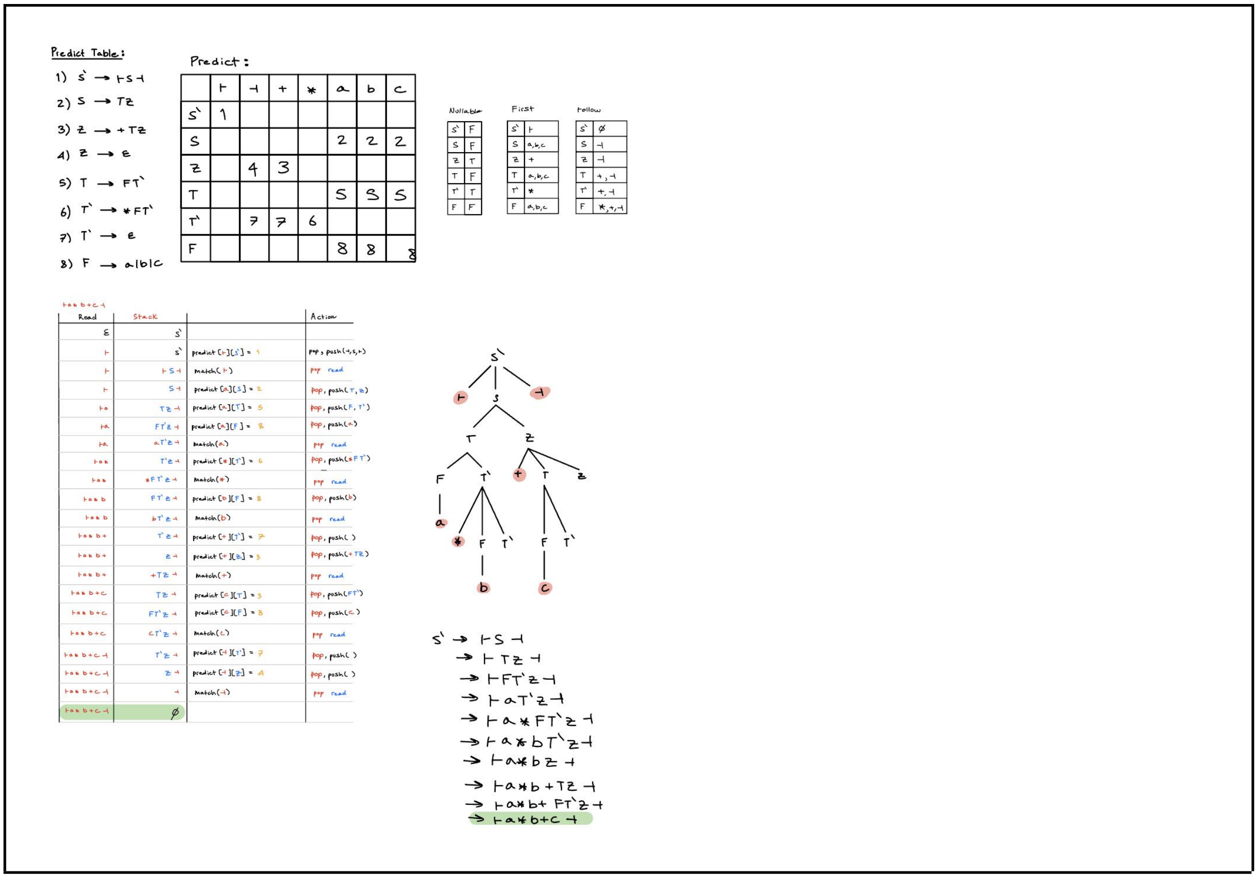
	⊢	⊣	y	a	b	c	d	w	x	z
S`	1									
S				2		2				
A				3		4				
B					5			6		

① push S` → ⊢S⊣

			TOS		
Read	Unread	Stack	non-terminal	terminal	Action
ε	⊢abywx⊣	⊢ S ⊣			
ε	⊢abywx⊣	⊢ S ⊣		⊢ matches ⊢	→ pop read
⊢	abywx⊣	S ⊣	predict [S][a] = 2		→ pop push(B, y, A)
⊢	abywx⊣	A y B ⊣	predict [A][a] = 3		→ pop push(b, a)
⊢	abywx⊣	a b y B ⊣		a matches a	→ pop read
⊢a	bywx⊣	b y B ⊣		b matches b	→ pop read
⊢ab	ywx⊣	y B ⊣		y matches y	→ pop read
⊢aby	wx⊣	B ⊣	predict [B][w] = 6		→ pop push(x, w)
⊢aby	wx⊣	w x ⊣		w matches w	→ pop read
⊢abyw	x⊣	x ⊣		x matches x	→ pop read
⊢abywx	⊣	⊣		⊣ matches ⊣	

Sequence of rules: 1, 2, 3, 6




S`
⊢
⊣
S
T
Z
F
T`
+
T
Z
a
*
F
T`
F
T`
b
c

Predict Table:

1) S` → ⊢S⊣
2) S → TZ
3) Z → +TZ
4) Z → ε
5) T → FT`
6) T` → *FT`
7) T` → ε
8) F → a|b|c

Predict:

	⊢	⊣	+	*	a	b	c
S`	1						
S					2	2	2
Z		4	3				
T					5	5	5
T`		7	7	6			
F					8	8	8

Nullable

S`	F
S	F
Z	T
T	F
T`	T
F	F

First

S`	⊢
S	a,b,c
Z	+
T	a,b,c
T`	*
F	a,b,c

Follow

S`	∅
S	⊣
Z	⊣
T	+, ⊣
T`	+, ⊣
F	*,+,⊣

⊢a*b+c⊣

Read	Stack		Action
ε	S`		
⊢	S`	predict [⊢][S`] = 1	pop, push(⊣,S,⊢)
⊢	⊢S⊣	match(⊢)	pop read
⊢	S⊣	predict [a][S] = 2	pop, push(T, Z)
⊢a	TZ⊣	predict [a][T] = 5	pop, push(F, T`)
⊢a	FT`Z⊣	predict [a][F] = 8	pop, push(a)
⊢a	aT`Z⊣	match(a)	pop read
⊢a*	T`Z⊣	predict [*][T`] = 6	pop, push(*FT`)
⊢a*	*FT`Z⊣	match(*)	pop read
⊢a*b	FT`Z⊣	predict [b][F] = 8	pop, push(b)
⊢a*b	bT`Z⊣	match(b)	pop read
⊢a*b+	T`Z⊣	predict [+][T`] = 7	pop, push( )
⊢a*b+	Z⊣	predict [+][Z] = 3	pop, push(+TZ)
⊢a*b+	+TZ⊣	match(+)	pop read
⊢a*b+c	TZ⊣	predict [c][T] = 3	pop, push(FT`)
⊢a*b+c	FT`Z⊣	predict [c][F] = 8	pop, push(c)
⊢a*b+c	cT`Z⊣	match(c)	pop read
⊢a*b+c⊣	T`Z⊣	predict [⊣][T`] = 7	pop, push( )
⊢a*b+c⊣	Z⊣	predict [⊣][Z] = 4	pop, push( )
⊢a*b+c⊣	⊣	match(⊣)	pop read
⊢a*b+c⊣	∅		

S` → ⊢S⊣
→ ⊢TZ⊣
→ ⊢FT`Z⊣
→ ⊢aT`Z⊣
→ ⊢a*FT`Z⊣
→ ⊢a*bT`Z⊣
→ ⊢a*bZ⊣
→ ⊢a*b+TZ⊣
→ ⊢a*b+FT`Z⊣
→ ⊢a*b+c⊣

Limitations

Left-Associative Grammars are **never** LL(1)
Right-Associative Grammars **can be** LL(1)

Solution

1. **At least** make it right recursive
2. Factor **if needed**
3. Add precedence **if needed**
    a. to introduce parentheses to force the left association

left recursive	right recursive
1 $S \to S+T$	1 $S \to T + S$
2 $S \to T$	2 $S \to T$
3 $T \to T * F$	3 $T \to F * T$
4 $T \to F$	4 $T \to F$
5 $F \to a\|b\|c$	5 $F \to a\|b\|c$

when we make it right-recursive:

predict [a][S] = $S \to S+T$ or $S \to T$

predict [a][T] = $T \to F * T$ or $T \to F$

2 distinct rules w/ same LHS can give us the same terminal

don't have more lookaheads

---

we could increase the lookahead: LL(2)

but the predict table is far more complex

Left Factoring

$$S \to \alpha T_1 \qquad S \to \alpha T_2 \qquad S \to \alpha T_3 \quad \xrightarrow{\text{factor}} \quad S \to \alpha T` \qquad T` \to T_1 \mid T_2 \mid T_3$$

left recursive	right recursive	right recursive, left factored
$S \to S+T$	$S \to T + S$	$S \to TX$ ⟶ predict [a][S] = $S \to TX$ ✓
$S \to T$	$S \to T$	$X \to +S \mid \epsilon$
$T \to T * F$	$T \to F * T$	$T \to FY$ ⟶ predict [a][T] = $T \to FY$ ✓
$T \to F$	$T \to F$	$Y \to *T \mid \epsilon$
$F \to a\|b\|c$	$F \to a\|b\|c$	$F \to a\|b\|c$

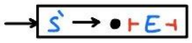
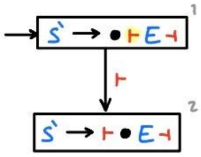
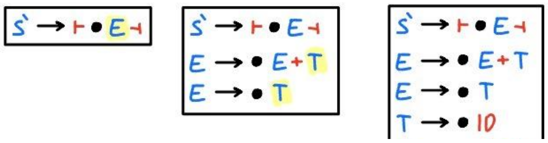
problem: right associative

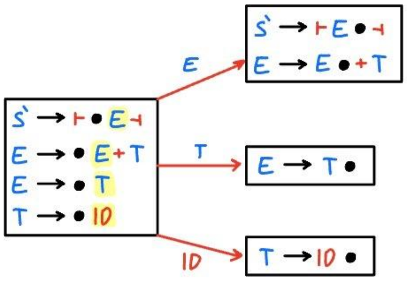
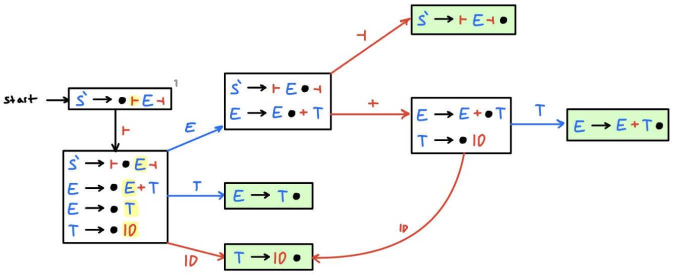
M6

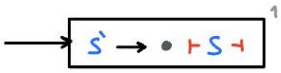
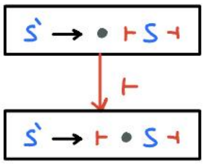
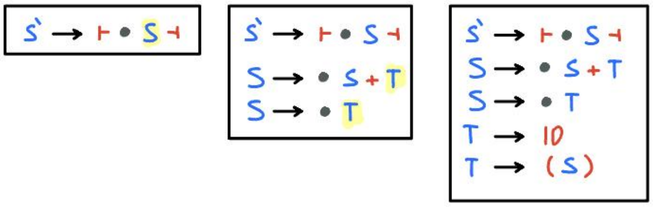
	Bottom Up	Top Down
Goes from:	input string → start symbol	start symbol → input string
Produces:	rightmost derivation	leftmost derivation
Works for:	left-associative grammars	right-associative grammars

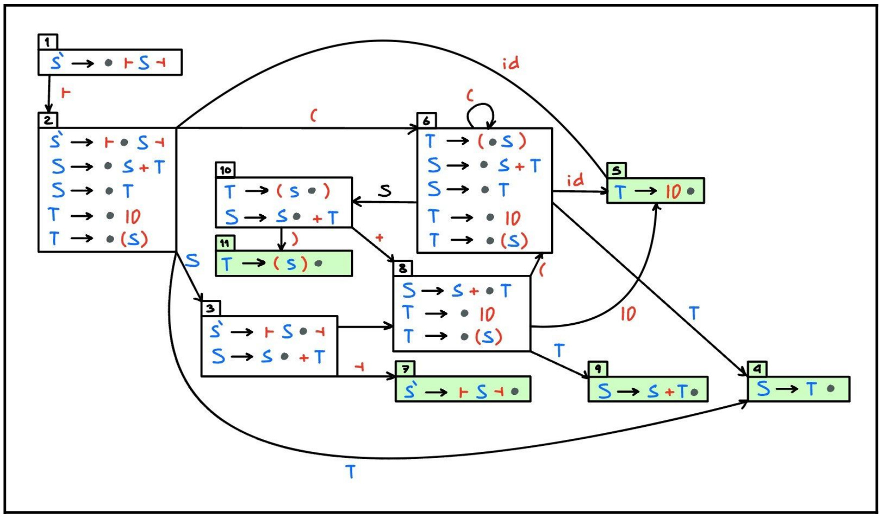
LR(0) Parsing

Building an LR(0) DFA	
1	start state = 1st rule
2	For every NT/T **DIRECTLY TO THE RIGHT** of the bookmark, X: → transition to a new state on symbol X + *bookmark* pushed 1 ahead
5	Mark states containing **reducible items** as accept-states
3	If a NT is to the **DIRECTLY TO THE RIGHT**of a bookmark: ● Add all rules to the state with X on the LHS
4	Repeat step 2-3 until no new states are discovered

EX 1	
1) S` → ⊢E⊣ 2) E → E + T 3) E → T 4) T → 10	
1	
2	
3	 S` → ⊢ • E ⊣ S` → ⊢ • E ⊣ E → • E + T E → • T S` → ⊢ • E ⊣ E → • E + T E → • T T → • 10

2	 S` → ⊢ • E ⊣ E → • E + T E → • T T → • ID E S` → ⊢ E • ⊣ E → E • + T T E → T • ID T → ID •
 start S` → • ⊢ E ⊣ 1 ⊢ S` → ⊢ • E ⊣ E → • E + T E → • T T → • ID E S` → ⊢ E • ⊣ E → E • + T ⊣ S` → ⊢ E ⊣ • + E → E + • T T → • ID T E → E + T • T E → T • ID ID T → ID •	

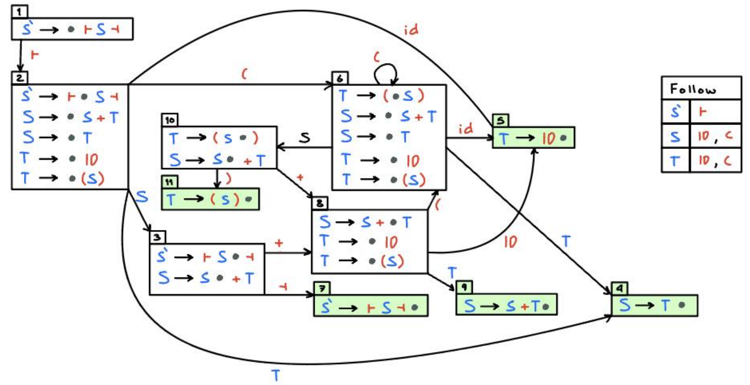
Example 2	
1) S` → ⊢ S ⊣ 2) S → S + T 3) S → T 4) T → ID 5) T → (S)	
1	
2	
3	 S` → ⊢ • S ⊣ S` → ⊢ • S ⊣ S → • S + T S → • T S` → ⊢ • S ⊣ S → • S + T S → • T T → ID T → (S)



Algorithm

1	State stack containing the start state
2	if [**reduce-state**] ● **reduce** x (# symbols on RHS of rule) ○ symbols.pop ○ states.pop ● symbols.push(LHS) ● state ⇒ DFA(new state, new symbol) else: **shift** ● symbols.push(symbol) ● states.push(newState)
3	**Accept:** Pushed start symbol
4	**Reject:** 1. cannot reduce 2. can't shift

Example:



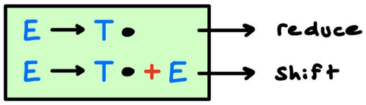
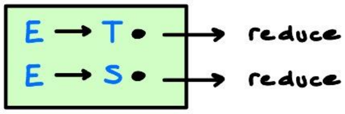
Follow	
S'	⊢
S	ID , c
T	ID , c

Read	Unread	State Stack	Symbol Stack	Action
ε	⊢ (a+b)+c ⊣	1		Shift ⊢
⊢	(a+b)+c ⊣	1, 2	⊢	Shift (
⊢ (	a+b)+c ⊣	1, 2, 6	⊢ (	Shift a
⊢ (a	+b)+c ⊣	1, 2, 6, 5	⊢ (a	reduce(T → ID): q_4
⊢ (a	+b)+c ⊣	1, 2, 6, 4	⊢ (T	reduce(S → T): q_{10}
⊢ (a	+b)+c ⊣	1, 2, 6, 10	⊢ (S	Shift +
⊢ (a +	b)+c ⊣	1, 2, 6, 10, 8	⊢ (S + b	Shift b
⊢ (a+b	)+c ⊣	1, 2, 6, 10, 8, 5	⊢ (S + T	reduce(T → ID): q_9
⊢ (a+b	)+c ⊣	1, 2, 6, 10	⊢ (S	reduce(S → S+T): q_{10} 3 symbols
⊢ (a+b)	+c ⊣	1, 2, 6, 10, 11	⊢ (S)	Shift)
⊢ (a+b)	+c ⊣	1, 2, 4	⊢ T	reduce(T → (S)): 3 symbols
⊢ (a+b)	+c ⊣	1, 2, 3	⊢ S	reduce(S → T): q_3
⊢ (a+b)+	c ⊣	1, 2, 3, 8	⊢ S +	Shift +
⊢ (a+b)+c	⊣	1, 2, 3, 8, 5	⊢ S + c	Shift c
⊢ (a+b)+c	⊣	1, 2, 3, 8, 9	⊢ S + T	reduce(T → ID)
⊢ (a+b)+c	⊣	1, 2, 3	⊢ S	reduce(S → S+T)
⊢ (a+b)+c ⊣	ε	1, 2, 3, 7	⊢ S ⊣	Shift ⊣
⊢ (a+b)+c ⊣	ε	1	S'	reduce(S' → ⊢ S ⊣)

Limitations/Conflicts

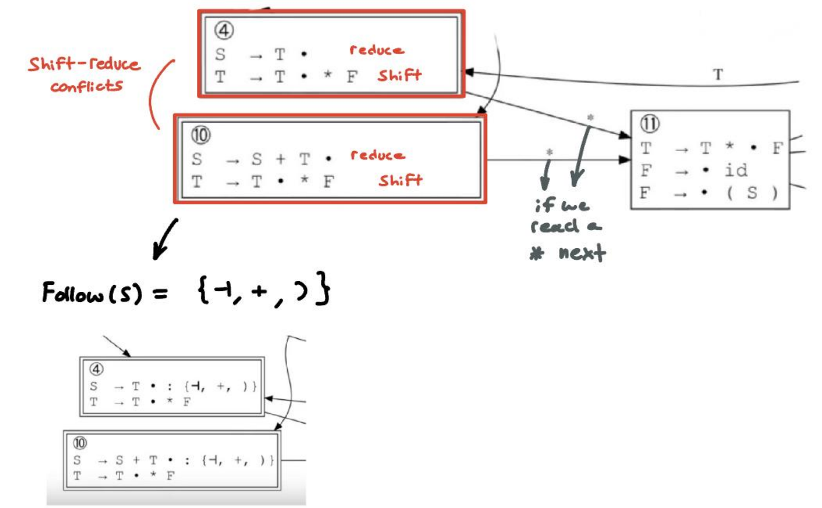
Right-Recursive grammars are not LR(0)

A grammar is LR(0) ⇔ LR(0) DFA does not have any **shift-reduce** or **reduce-reduce conflicts**.

Conflict	Why	Ex
Shift - Reduce	State w' irreducible and reducible items	
Reduce - Reduce	State has two reducible items	

Example

1) S` → ⊢ S ⊣
2) S → S + T
3) S → T
4) T → T * F
5) T → F
6) F → ID
7) F → (S)



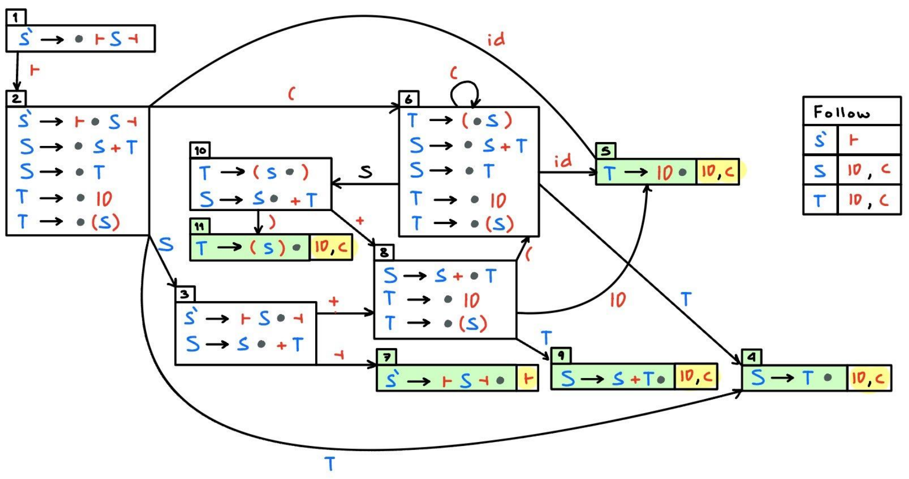
SLR(1)

- LR(0) + 1 lookahead

SLR(1) DFA

⇒ **LR(0) DFA** + Follow Sets for Reducible Items

Example:



Follow	
S`	⊢
S	ID, (
T	ID, (

Algorithm

If State has a Reducible Item

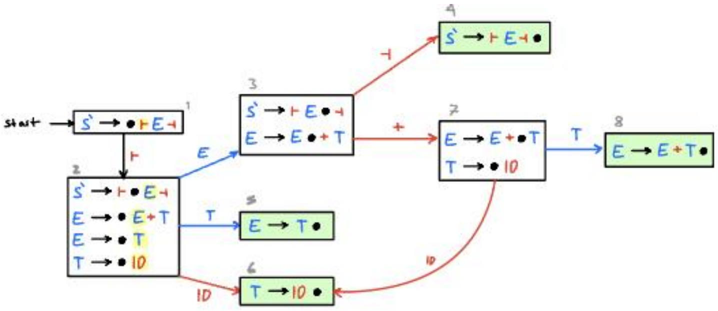
1. **No Conflict**
    - reduce
2. **Reduce-Shift**
    - next input symbol is in the FollowSet ⇒ **Reduce**
    - else: **shift**
3. **Reduce-Reduce**
    - pick reduction that has the next input symbol in its FollowSet

Parse Trees (from LR(0) or SLR(1))

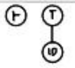
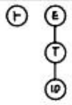
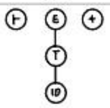
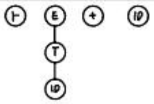
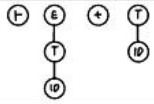
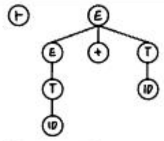
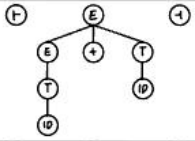
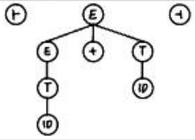
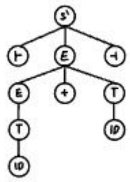
	Tree
shift	Push a node onto the treeStackonto that contains the symbol
reduce	Pops the subtree nodes that represent the RHS of the rule Push a new tree: ● root = LHS ● children = RHS

Example

1) $S' \rightarrow \vdash E \dashv$
2) $E \rightarrow E + T$
3) $E \rightarrow T$
4) $T \rightarrow 10$


start
1
$S' \rightarrow \bullet \vdash E \dashv$
⊢
2
$S' \rightarrow \vdash \bullet E \dashv$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet 10$
E
3
$S' \rightarrow \vdash E \bullet \dashv$
$E \rightarrow E \bullet + T$
⊣
4
$S' \rightarrow \vdash E \dashv \bullet$
+
7
$E \rightarrow E + \bullet T$
$T \rightarrow \bullet 10$
T
8
$E \rightarrow E + T \bullet$
T
5
$E \rightarrow T \bullet$
10
6
$T \rightarrow 10 \bullet$
10

Input: ⊢ 10 + 10 ⊣

Read	States	Symbols	Action	Tree Stack
	1			
⊢	1,2	⊢	shift ⊢	⊢
⊢ 10	1,2, 6	⊢ 10	shift 10	⊢ 10
⊢ 10	1, 2, 5	⊢ T	reduce (T→10)	
⊢ 10	1, 2, 3	⊢ E	reduce (T→E)	
⊢ 10+	1, 2, 3, 7	⊢ E +	shift +	
⊢ 10 + 10	1, 2, 3, 7, 6	⊢E + 10	shift 10	
⊢ 10 + 10	1, 2, 3, 7, 8	⊢E + T	reduce(T→10)	
⊢ 10 + 10	1, 2, 3	⊢ E	reduce(E→E+T)	
⊢ 10 + 10 ⊣	1, 2, 3, 4	⊢E	Shift (⊣)	
⊢ 10 + 10 ⊣	1, 2, 3, 4	⊢E ⊣	Shift (⊣)	
⊢ 10 + 10 ⊣	1	∅	reduce(S'→⊢E⊣)	

M7

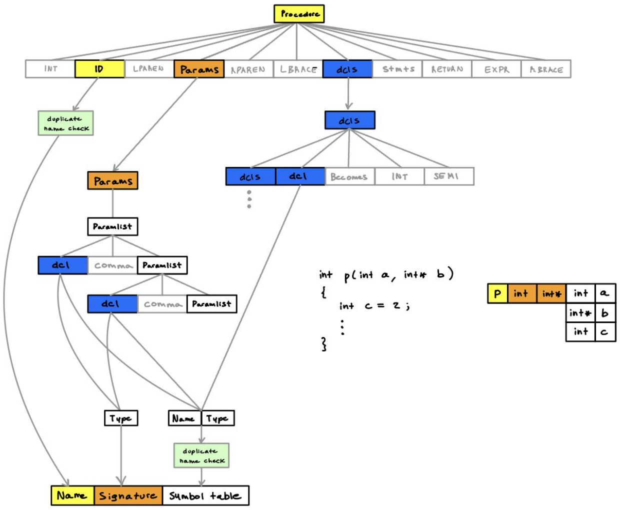
Checked by Parsing	Need To Check
• wain function has been defined • return (appears only once in last statement) • every return type is an integer	1. Type rules **Variables** 2. Can't declare twice **in the same function** 3. Can't use before declared **Procedures** 1. Can't declare twice 2. Can't use before declared

CSA

Implementation (Tree Traversal)

Step 1

1. Duplicate identifier (procedure, variable) checks
2. Build SymbolTable



```
int p(int a, int* b)
{
    int c = 2;
    ⋮
}
```

P	int	int*	int	a
			int*	b
			int	c

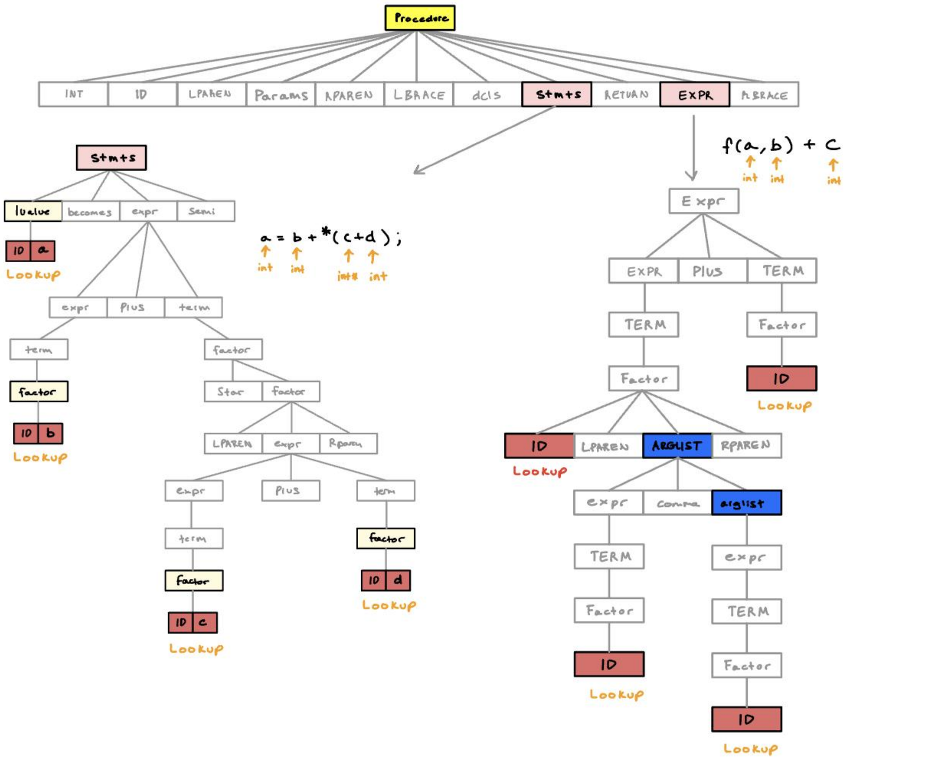
Step 2: Declared before Use

Variables used (in **statements/RETURN**) are **declared**

- factor → ID
- lvalue → ID

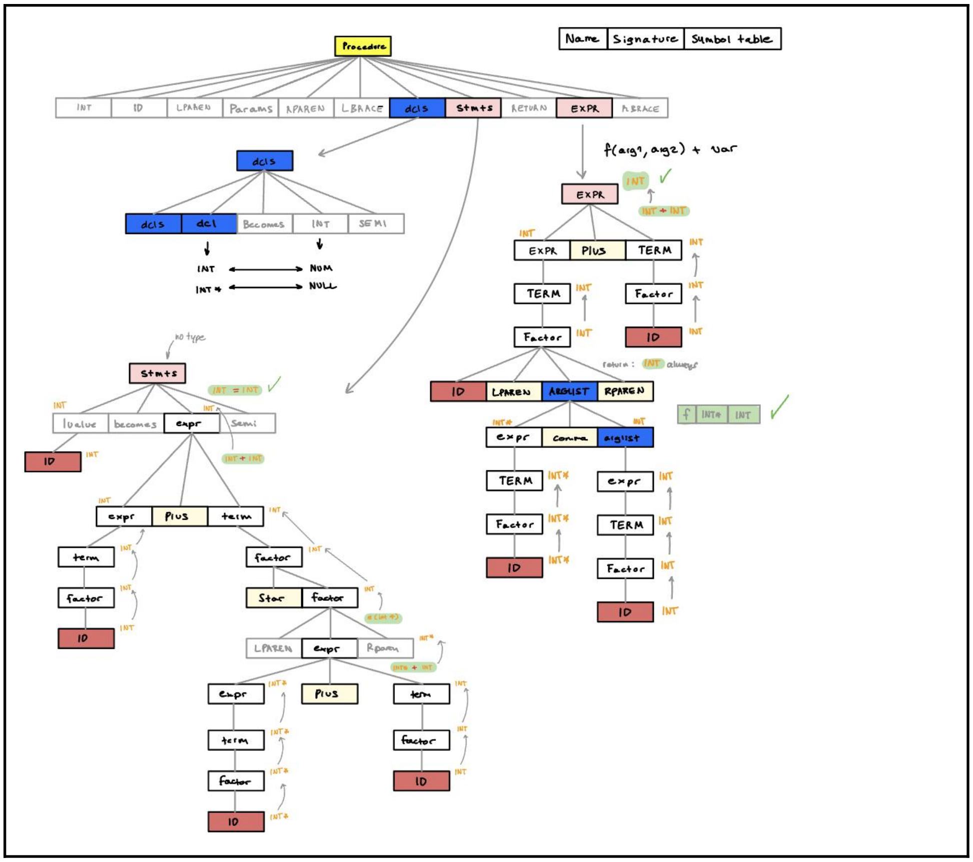
Procedures used (in **statements/RETURN**) **declared** & **signatures match**

- factor → ID LPAREN RPAREN
- factor → ID LPAREN arglist RPAREN



Step 3: Type Checking

1. **dcls**
2. **statements/RETURN**



Type Checking Expressions

```
void typeOf(Tree tree)
{
        for each child c of tree
        {
                typeOf( c )
        }
        // refer to the **type system rule** for this tree, to determine if it’s valid
}
```

Premise	Type
NUM	int
NULL	int*
int* + int	int*
int* - int	int*

int* - int*	int
function	int

Type Checking Statements

Expressions	**Statements (contain expressions)**
We **infer** the types of expressions	Can't **infer** a type
well typed ⇔ type can be inferred	**well type** ⇔ components are well-typed

Statement	**Well Typed ⇔**
println	parameter = int
delete	Parameter = int*
assignment: a = b	type(LHS)=type(RHS)
empty sequence of statements	always well typed
test	operands are of the same type
if statement	components are well-typed
while statement	components are well-typed
empty	well typed
int ID =	declared to integer
int* ID =	declared to NULL
procedure	1. dcls : well typed 2. statements : well types 3. returns : int
wain	1. 2^{nd} parameter : INT 2. dcls : well typed 3. statements : well types 4. returns : int

M8

Extend the symbol table

- With a **location** (offset from **Frame Pointer**) entry for each variable

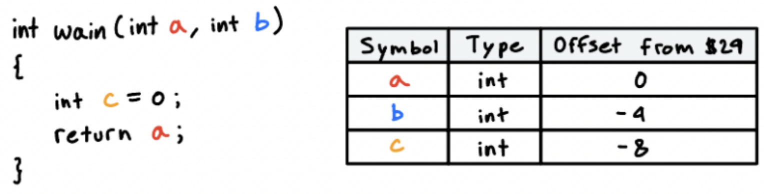
Extend the tree

- **type annotated** (for checking type with pointer arithmetic)

Frame Pointer (FP)

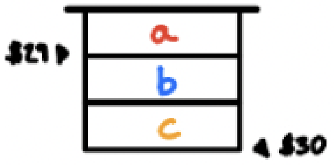
FP = $29

- **location** of the **first value pushed on the stack by a procedure**
- does not change within a procedure

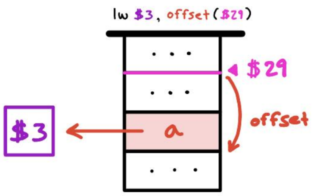


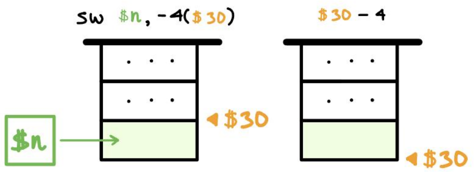
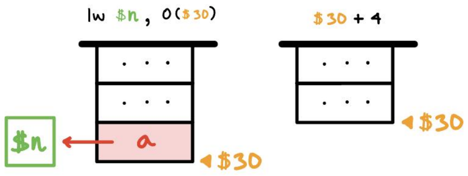
```
int wain(int a, int b)
{
    int c = 0;
    return a;
}
```

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



Shorthand/Conventions

Shorthands	
code(a)	 lw $3, offset($29)

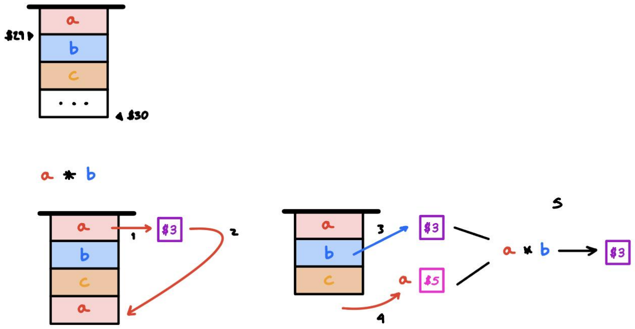
push($n)	
pop($n)	

Conventions

$10	"print" (address)
$11	1
$4	4
$5	Temporary values

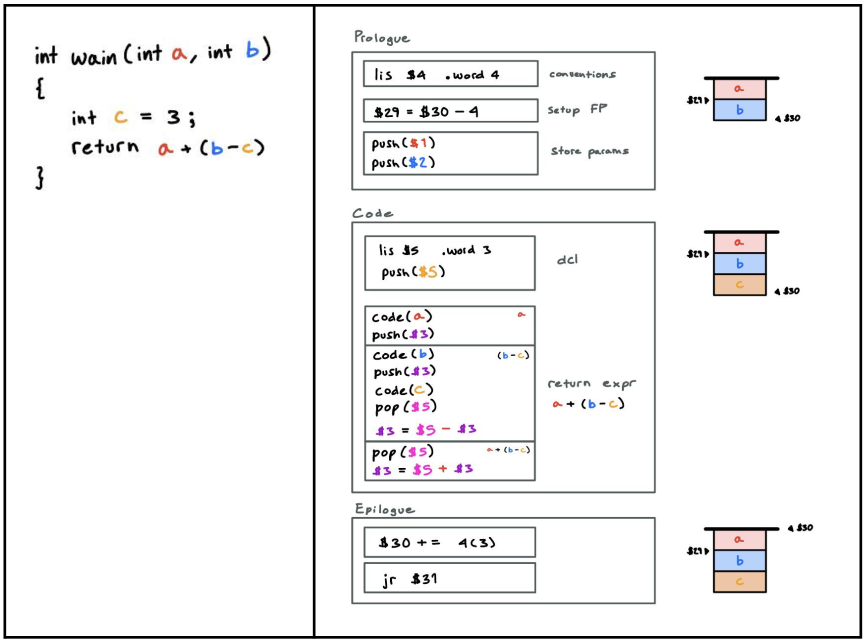
Expressions

Universal Code Gen Scheme

1	**code** (a)	 a $29 b c $30 a * b a b c a 1 $3 2 a b c 3 $3 a $5 4 5 a * b $3
2	**push** ($3)	
3	**code** (b)	
4	**pop** ($5)	
5	*$3* **=** ***F(***$3, $5***)***	

Step 3: 'b' is an expression: repeat steps 1-5

Example



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

Prologue

```
lis $4  .word 4
$29 = $30 - 4
push($1)
push($2)
```

Code

```
lis $5  .word 3
push($5)

code(a)
push($3)
code(b)
push($3)
code(c)
pop($5)
$3 = $5 - $3
pop($5)
$3 = $5 + $3
```

Epilogue

```
$30 += 4(3)
jr $31
```

Easy Ones

S	→	⊢ **procedure** ⊣
code(S)	=	**code** (procedure)

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

factor	→	(**expr**)
code(factor)	=	**code**(expr)

Variable Assignment

statement	→	**lvalue** BECOMES expr SEMI
code(factor)	=	**code**(expr) sw $3, offset($29)

printLn

statement	→	**PRINTLN LPAREN expr RPAREN SEMI**	
code(statement)	=	push ($1)	preserve $1
		code (expr) $1 = $3	expr = input to println
		push ($31) lis $5 .word *print* jalr $5 pop ($31)	call println()
		pop ($1)	restore $1

Comparisons

Pointer Comparison	**Integer Comparison**
sltu	slt

Convention: $3 ⇐ 1 : comparison is true

test	→	**expr$_1$ * expr$_2$**
code(test)	=	**code** (expr$_1$) **push** ($3) **code** (expr$_2$) **pop** ($5) + code for *

*****	**code for ***
<	slt $3 ,$5 ,$3
>	slt $5, $3 ,$3
!=	slt $6 ,$3 ,$5 slt $7 ,$5 ,$3 add $3 , $6 , $7

==	code(!=) sub $3 , $11 , $3
<=	code(>) sub $3 , $11 , $3
>=	code(<) sub $3 , $11 , $3

if statement

statement	→	**IF (test) { stmts_1 } ELSE { stmts_2 }**
code()	=	code (test) beq $3 , $0, *else* code ($\text{stmts}_1$) beq $0 , $0 , *endif* else: code (stmts2) endif:

while statement

statement	→	**WHILE (test) { stmts }**
code()	=	loop: code(test) beq $3 , $0 , *endWhile* code (stmts) beq $0 , $0 , *loop* endWhile:

Pointers

NULL = **0x1**

factor	→	**NULL**

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

Dereferencing

factor	→	STAR factor
code()	=	code (factor$_2$) ; $3 = address lw $3 , 0($3)

Get Address of

	factor	→	AMP lvalue
1	lvalue = ID		
	code()	=	lis $3 .word offset (from symbolTable) $3 = $29 + $3
2	lvalue = STAR factor		
	code()	=	code(factor)

Assignment Through Pointer Dereference

	statement	→	lvalue BECOMES expr SEMI
1	lvalue = ID		
	code()	=	code(expr) sw $3, offset($29)
2	lvalue = STAR factor		

	code()	=	code(expr) //value
			push($3)
			code(factor) //address
			pop $5
			sw $5, 0($3)

Arithmetic

Note: sizeof(int) = 4

	expr	→	**expr + term**
1	int + int*		

	code()	=	**expr + (4 × term):**
			
			code(**expr**)
			push($3)
			code(**term)**
			mult $3, $4
			mflo $3
			pop $5
			$3 = $5 + $3

2	int* + int		
	code()	=	**(expr × 4) + term**

	expr	→	**expr - term**
1	int* + int		
	code()	=	**(expr × 4) - term**
2	int* + int*		
	code()	=	**# elements between 2 addresses:** code (expr) push ($3) code (term) pop ($5) sub $3 , $5 , $3 ; $3 = $expr_2$ - term div $3 , $4 ; $3 / 4 mflo $3

New/Delete

We rely on the runtime environment to provide support for **new** & **delete**

alloc.merl

Exports	import with	Expects	Returns
new	.import *new*	$1 = # words needed	$3 = start address
delete	.import *delete*	$1 = address to be deallocated	
init	.import *init*	$2 = ● length of array (mips.array) ● 0 (mips.twoints)	

new

	→	**new int [expr]**
code()	=	code (expr) $1 = $3 ;new procedure expects value in $1 push ($31) lis $5 .word *new* jalr $5 pop ($31) bne $3 , $0 , 1 ; if call succeeded skip next instruction $3 = 1 ; set $3 to NULL address if allocation fails

delete

	→	**delete [ ] expr**
code()	=	code (expr) beq $3, $11, *skipDelete* ; do NOT call delete on NULL $1 = $3 ; delete expects the address in $1 push ($31) lis $5 .word delete jalr $5 pop ($31) skipDelete :

Procedures

wain

1. **Imports**
2. **conventions**
3. **init()**
4. set FP
    a. code(dcls)
        i. code(stmts)
        ii. code(return expr)
    b. restore(dlcs)
5. jr $31

CALLER

push **$29** push $31
push arguments
lis $31 .word ***PROCEDURE*** jalr $31
pop arguments
pop $31 pop **$29**

PROCEDURE

PROCEDURE
$29 = $30 - 4
code(dcls) push [saved registers]
code(statements)

code(return expr)
pop [saved registers]
$30 = **$29** + $4
jr $31

NOTE:

- code(expr) ⇒ $3 = int
- code(lvalue) ⇒ $3 = address

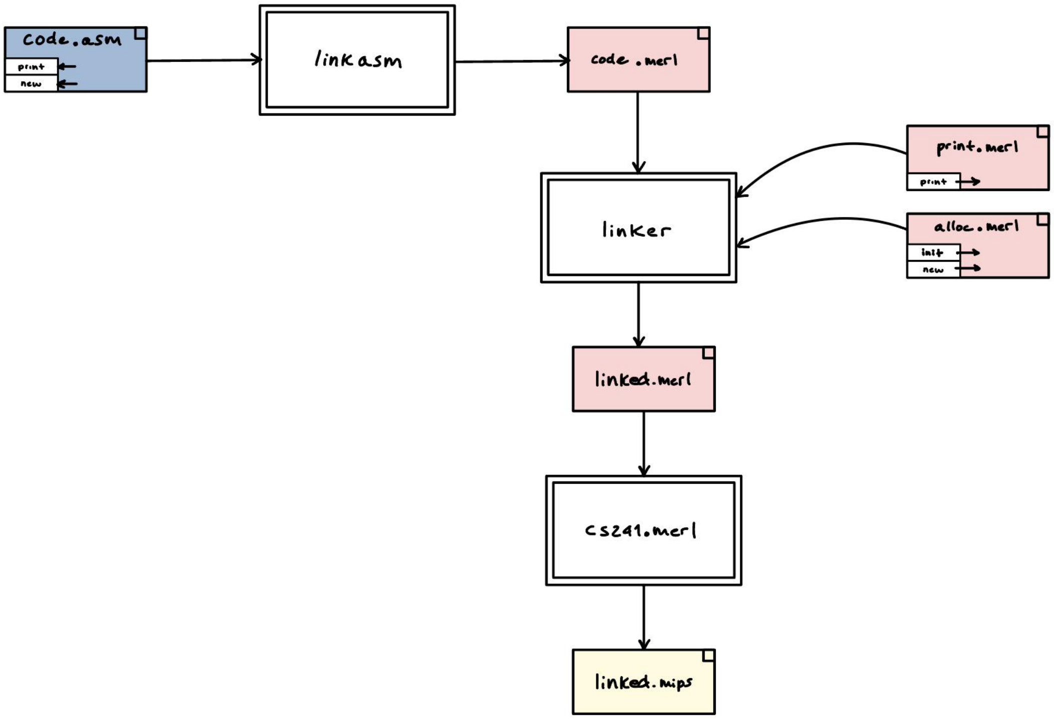
8.5

MERL : Object Files

machine code + **announcements**

- info for *linker* & *loader*

cs241.linkasm	• **Assembler** for MERL (understands **imports**) • produces MERL files **cs241.linkasm** < code.asm > code.merl
cs241.linker	• links object files • produces **linked MERL file** **cs241.linker** code.merl print.merl alloc.merl > linked.merl
cs241.merl	linked.merl - metadata ⇒ pure mips machine code **cs241.merl** 0 < linked.merl > exec.mips



Notes:

SLR(1)

shift-reduce conflict

- A → α • bβ where **b is in the follow set of the complete item**

reduce-reduce conflict

- 2 complete items with **overlapping follow sets**