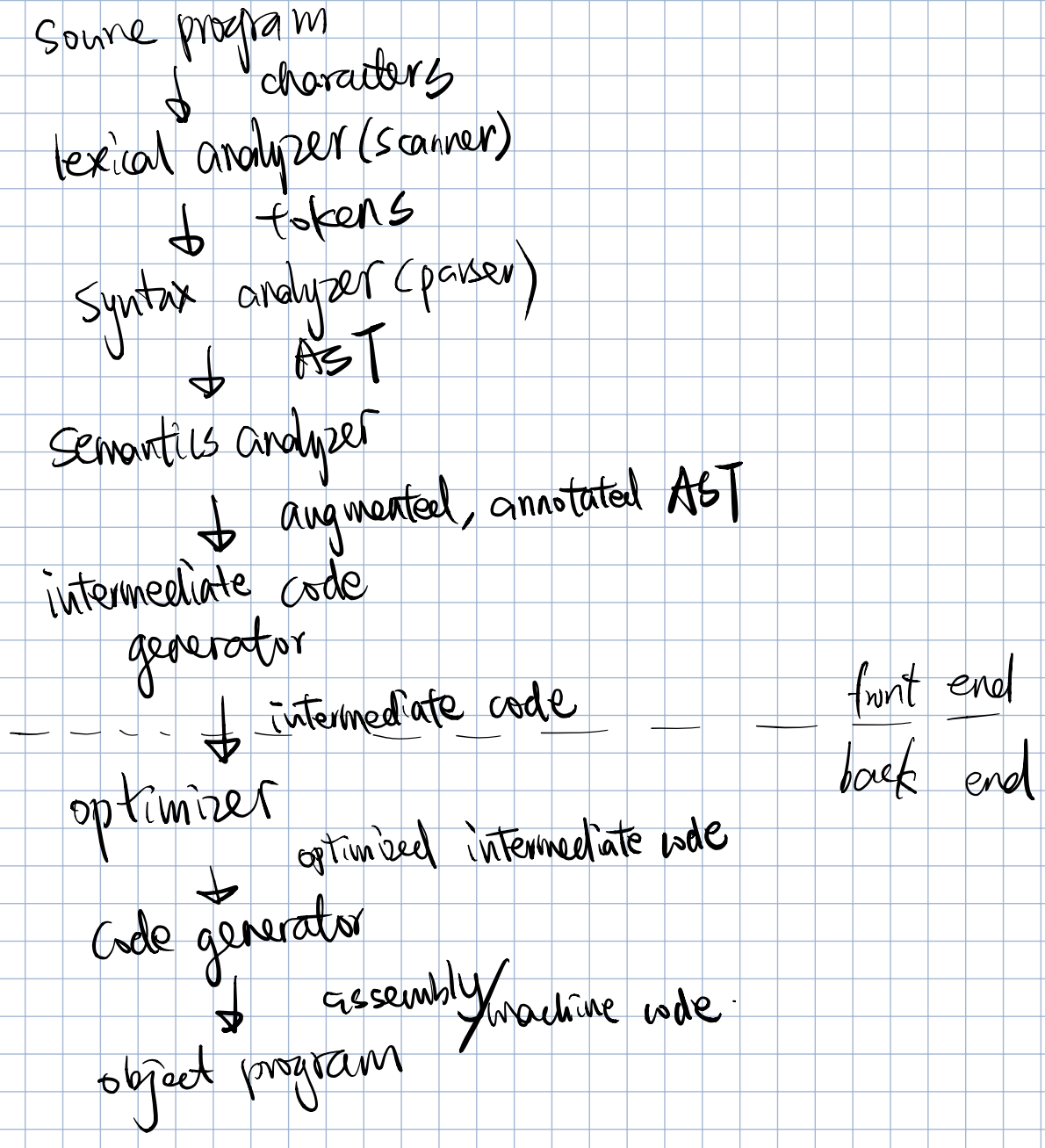
Lec 1 Compiler Overview



Errors to catch:

scanner	parser	Semantics analyzer
illegal character	Syntax error	Semantic error
unterminated string	Semantic error	Type checking: wrong type
too long int literal		Name analysis: undeclared var

Errors can only be found by scanner, parser and semantics analyzer. After that, no errors would be found.

Lec 2 DFA

Scanner: Sequence of chars → Sequence of tokens.

Find longest sequence of chars corresponding to a token

Finite State Machine (FSM), a.k.a. finite automata

Deterministic finite Automata (DFA)

$M \equiv (Q, \Sigma, \delta, q, F)$, where

Q: finite set of states

Σ: set of characters

$\delta : Q \times \Sigma \to Q$, transition functions

q: start state, $q \in Q$.

F: final states, $F \subseteq Q$.

Language of FSM M, $L(M)$. FSM accepts string $x_1 x_2 \cdots x_n$ iff.

$$(\delta(\ldots \delta(\delta(q, x_1), x_2) \ldots), x_n) \in F$$

Deterministic v.s. Non-deterministic

DFA : no state has >1 outgoing edge with the same label

NFA : may have >1 outgoing edges with same label.
edges may be labelled with ε

Lec 3 DFA vs. NFA ; RegEx

NFA: $M \equiv (Q, \Sigma, \delta, q, F)$

Q: finite set of states

Σ: set of characters

$\delta : Q \times \Sigma \to 2^Q$

q: start state $q \in Q$

F: final state(s) $F \subseteq Q$

For DFA: $\delta : Q \times \Sigma \to Q$
For NFA: $\delta : Q \times \Sigma \to 2^Q$. We have 2^Q as a state can have multiple outgoing edges with the same label.

Transition table

DFA

	c_1	c_2	...
s_1	s_2		
s_2		s_3	
⋮			

Each entry is a single state (or stuck).

NFA

	c_1	c_2	...
s_1	$\{s_2, s_3\}$	$\{s_1\}$	
s_2		$\{s_1, s_4\}$	
⋮			

Each entry is a set of state(s).

For DFA M, $x_1 x_2 \dots x_n \in L(M)$ iff.

$\delta(\dots\delta(\delta(q, x_1), x_2)\dots), x_n) \in F$

where the output of δ is zero/one state.

For NFA M, $x_1 x_2 \dots x_n \in L(M)$ iff

$\delta(\dots\delta(\delta(q, x_1), x_2)\dots), x_n) \cap F \neq \emptyset$

where the output of δ is a (possibly empty) set of states.

NFA and DFA are <u>equivalent</u>.

Two automata M and M' are equivalent iff. $L(M) = L(M')$

Lemma 1: DFA → NFA: given a DFA, you can construct an equivalent NFA.
Trivially correct.

Lemma 2: NFA → DFA

Converting NFA into DFA: ① normal-edge. The new DFA tracks set of states.
② remove epsilon.

Algorithm to remove ε-transition:

eclose(s) = set of all states reachable from s in ≥ 0 epsilon transitions.

Rule #1: To construct ε-free, equivalent FSM M' from M, s is an accepting state of M' iff. eclose(s) contains an accepting state.

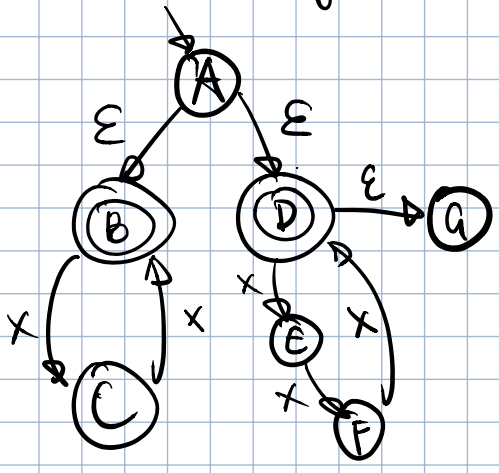
Rule #2: $s \xrightarrow{c} t$ is a transition in M' iff $q \xrightarrow{c} t$ from some $q \in$ eclose(s)

Graphically,

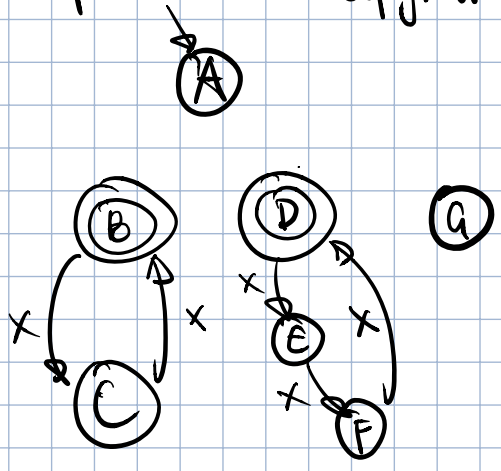
M (NFA): $\circ_s \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \circ_q \xrightarrow{c} \circ_t$, $q \in$ eclose(s)

M' (DFA): $\circ_s \xrightarrow{c} \circ_t$

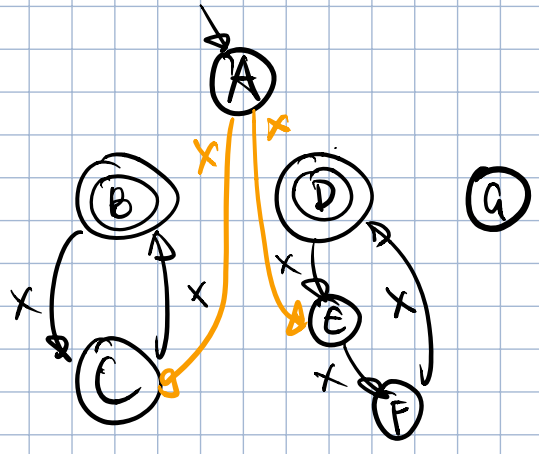
M: accepting x^n where n is even or divisible by 3.



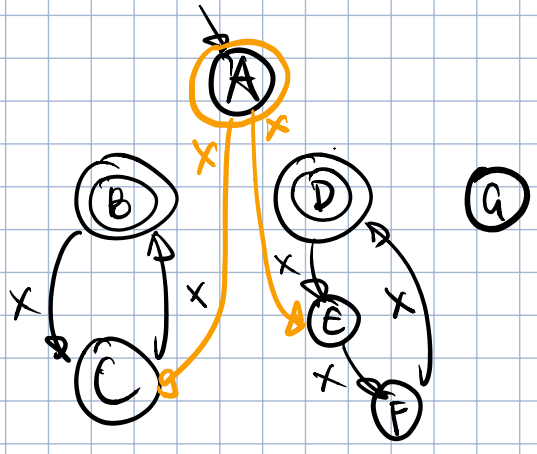
Step 1: Make a copy, with ε-transition removed



Step 2: Follow Rule 1



Step 3: Follow Rule 2



Conclusion: ① DFAs and NFAs are equivalent

② ε-transition don't add expressiveness to NFA.

Lec 4 RegExs & DFAs

Writing regexps differently doesn't affect performance: all compiled into DFAs.

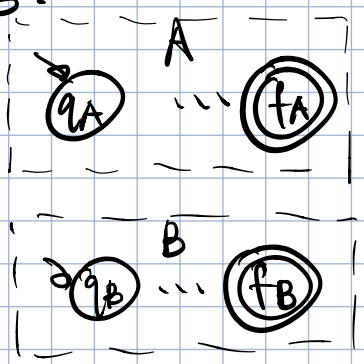
RegExps to NFAs

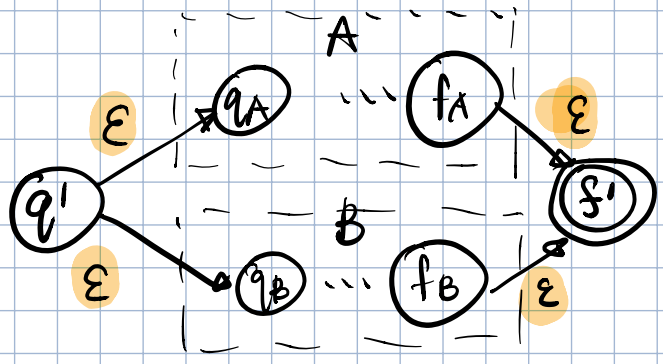
idea = literals/epsilon → simple, small DFAs

Operators → methods of joining DFAs.

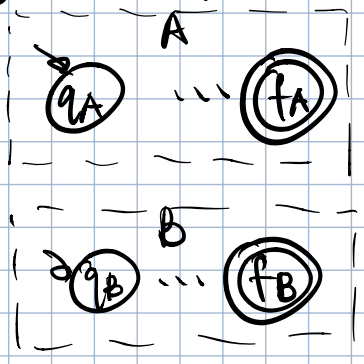
Rule: each expression has a unique start state and a unique accepting state.

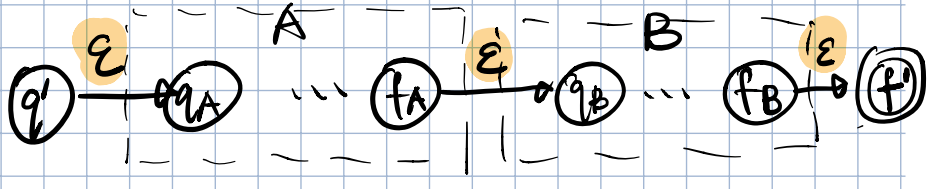
A|B:



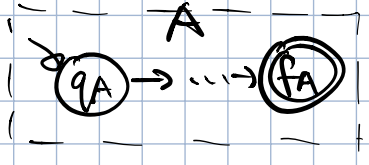


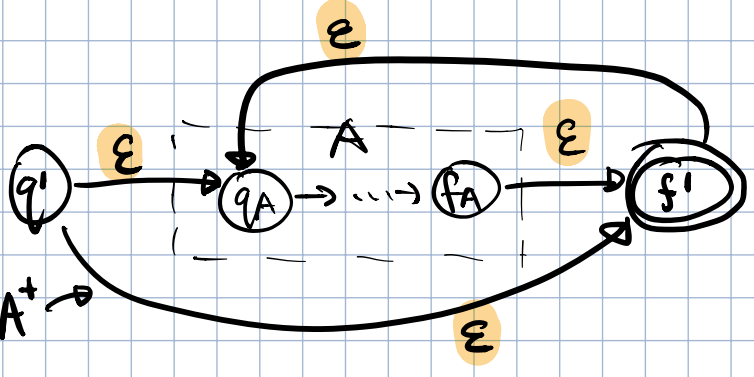
A.B (concatenation):





A* (iteration):





Remove this and you get A^+

RegExp operator precedence: * > . > |

Regexp → tree representation → NFA w/ ε → NFA w/o ε → DFA

Essentially: RegExp == NFA == DFA

Why we want DFA ?

The transitions δ can be expressed as a table and an efficient array representation.

Use FSM for tokenization ?

Up to now, FSM only checks language membership of a string. We need to relate the FSM with some actions.

Solution : action + ability to put char back.

Read/Construct DFA and action table

In this setting, at each run the DFA would return the longest token. Thus, the scanner would keeps running the FSM until it consumes the entire string/gets stuck.

From here (Lec 2 - Lec 4), we finished the discussion on scanner. Next topic: Parser.

Lec 5 CFG

RegExp is great for scanner (tokenization), but insufficient for parsing.

Limitations of regExp:

① cannot specify all structs. E.g "matching": balanced parenthesis.

② doesn't enforce structure on the tokens.

CFG $\equiv (N, \Sigma, P, S)$

N: set of non-terminals

Σ: set of terminals

P: set of production rules

S: $S \in N$, the initial non-terminal symbol

Derivation syntax: $\rightarrow$, $\rightarrow^+$, $\rightarrow^*$

Lec 6 Define Syntax Using CFGs w/o ambiguity

Imposing derivation order doesn't remove ambiguity.

Grammar G is ambiguous if it has >1 parse tree for a string s.

Remove ambiguity: precedence + associativity

- Fix precedence: It's a common practice to put literals and parenthesized terms at the same precedence level.

- Fix associativity: {left / right} recursion → {left / right} associativity

Lec 7 SDT

Syntax Directed Translation: augment CFG rules w/ translation rules.

Suppose we already have the parse tree, then apply SDT rules bottom-up.

Lec 8 Java CUP

Lec9 Bottom-up parsing algorithm: CYK

Chomsky normal form (CNF)

$X \to t$

$X \to A\ B$

The only rule allowed to derive epsilon is the start symbol S.

CYK: build the parse tree in $O(n^3)$, bottom-up

Useless Grammar: ① A nonterminal that cannot derive terminals.

② A nonterminal that can't be derived from the start symbol

Convert grammar into CNF: 4 steps

① eliminate ε rules

F → id(A)
A → ε
A → N

⇒

F → id()
| id(A)
A → N

X → A x A
A → ε
A → Z

⇒

X → A x A
X → x A
X → A x
X → x
A → Z

② eliminate unit rules

F → id()
| id(A)
A → N

⇒

F → id()
| id(N)

③ Fix productions with terminals (and other things) on RHS

F → id ( )
| id (N)

⇒

F → I L R
| I L N R
I → id
L → (
R →)

[Rename each terminal as an non-terminal.]

④ Fix productions with >2 nonterminals on RHS.

A → B C D

⇒

A → B E
E → C D

Lec 10 Top-down parsing for LL(1) (Recursive Descent)

Restricting the grammar to be LL(1), O(n) top-down parser.

Algorithm:

```
stack.push(eof)
stack.push(Start non-term)
t = scanner.getToken()
Repeat
  if stack.top is a terminal y
     match y with t
     pop y from the stack
     t = scanner.next_token()
  if stack.top is a nonterminal X
     get table[X,t]
     pop X from the stack
     push production's RHS in reverse order
Until one of the following:
   stack is empty                          accept
   stack.top = terminal that doesn't match t
   stack.top = non-term and parse table entry is empty
                                           reject
```

Necessary (not sufficient) conditions for LL(1):

① No left recursion
② Left factored

If left-recursion exists, no prediction can be guaranteed to be correct.
If not left factored, obvious ambiguity.

- Remove left-recursion:

$$A \to A\alpha \mid \beta \quad \Rightarrow \quad A \to \beta A' \qquad A' \to \alpha A' \mid \varepsilon$$

$$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \quad \Rightarrow \quad A \to \beta_1 \mid \beta_2 \mid \dots \mid \beta_m A' \qquad A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$$

- Left factoring

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \quad \Rightarrow \quad A \to \alpha A' \qquad A' \to \beta_1 \mid \beta_2$$

$$A \to \alpha\beta_1 \mid \dots \mid \alpha\beta_m \mid \gamma_1 \mid \dots \mid \gamma_n \quad \Rightarrow \quad A \to \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_n \qquad A' \to \beta_1 \mid \dots \mid \beta_m$$

Lec 11 Build the Parse Table

FIRST(α): the set of terminals that begin the strings derivable from α.
If α can derive ε, then ε is also in FIRST(α).

$$FIRST(\alpha) = \{ t \mid (t \in \Sigma \wedge \alpha \Rightarrow^{*} t\beta) \vee (t = \varepsilon \wedge \alpha \Rightarrow^{*} \varepsilon) \}$$

If $\alpha \in \Sigma$, $FIRST(\alpha) = \{\alpha\}$

If $\alpha \in N$, Let $\alpha = Y_1 Y_2 \dots Y_k$
- Put FIRST(Y_1) - {ε} in FIRST(α)
  - If ε is in FIRST(Y_1): add FIRST(Y_2) – {ε} to FIRST(α)
  - If ε is in FIRST(Y_2): add FIRST(Y_3) – {ε} to FIRST(α)
  - ...
  - If ε is in FIRST of all Y_i symbols, put ε into FIRST(α)

FOLLOW(α): For non-terminal α, FOLLOW(α) is the set of terminals that can appear immediately to the right of α.

$$FOLLOW(A) = \{ t \mid (t \in \Sigma \wedge S \Rightarrow^{*} \alpha A t \beta) \vee (t = eof \wedge S \Rightarrow^{*} \alpha A) \}$$

FOLLOW(A) for $X \longrightarrow \alpha\ A\ \beta$
If A is the start, add **eof**
Add FIRST(β) – {ε}
Add FOLLOW(X) if ε in FIRST(β) or β is empty

When building FOLLOW(A), look at every production rule whose RHS contains A.

Build the parse table

```
for each production X ⟶ α {
  for each terminal t in FIRST(α) {
    put α in Table[X][t]
  }
  if ε is in FIRST(α) {
    for each terminal t in FOLLOW(X) {
      put α in Table[X][t]
    }
  }
}
```

← Iterating over each production rule

Whole process:
1. Build FIRST for all nonterminals
2. Build FIRST for all RHSs
3. Build FOLLOW for all nonterminals
4. Fill the parse table

FIRST(α) for $\alpha = Y_1 Y_2 \ldots Y_k$

Add FIRST(Y_1) - {ε}

If ε is in FIRST($Y_{1 \text{ to } i-1}$): add FIRST(Y_i) – {ε}

If ε is in all RHS symbols, add ε

FOLLOW(A) for $X \longrightarrow \alpha A \beta$

If A is the start, add **eof**

Add FIRST(β) – {ε}

Add FOLLOW(*X*) if ε in FIRST(β) or β empty

Table[X][t]

"S → B c | D B" should be considered as {"S → B c", "S → D B"}

```
for each production X ⟶ α
  for each terminal t in FIRST(α)
    put α in Table[X][t]
  if ε is in FIRST(α){
    for each terminal t in FOLLOW(X){
      put α in Table[X][t]
```

FIRST (S) = { **a, c, d** }

FIRST (B) = { **a, c** }

FIRST (D) = { **d**, ε }

FIRST (**B c**) = { **a, c** }

FIRST (**D B**) = { **d, a, c** }

FIRST (**a b**) = { **a** }

FIRST (**c** *S*) = { **c** }

FIRST (**d**) = { **d** }

FIRST (ε) = { ε }

FOLLOW (S) = { **eof, c** }

FOLLOW (B) = { **c, eof** }

FOLLOW (D) = { **a, c** }

Table collision ⟺ Not LL(1)

CFG

S ⟶ **B c** | **D B**

B ⟶ **a b** | **c** S

D ⟶ **d** | ε

Not LL(1)

	a	b	c	d	eof
S	**B c** **D B**		**B c** **D B**	**D B**	
B	**a b**		**c** *S*		
D	ε		ε	**d**	

Now we can use recursive descent, instead of CYK, to build the parse tree for LL(1) grammar.

Lec 12 SDT for Top-Down Parsing

- Last time: remove left-recursion, perform left factoring to get LL(1). However, grammar transformation affect the structure of the parse tree.
- So far, SDT shown as a procedure performed on parse-tree. But for LL(1), the parser never needs to explicitly build the parse tree. Instead, the LL(1) parser uses a **semantic stack** to implicitly track the state. Also, SDT rules become actions (popping, pushing) on the semantic stack.

- Number actions and put them on the symbol stack by adding action number symbols at the end of the production.
- Action numbers go **AFTER** the corresponding nonterminals, **BEFORE** their corresponding terminal. (We put action number before the corresponding terminal so as to consume its value.)

CFG		SDT Actions
Expr →	*Expr* + *Term* #1	#1 tTrans = pop ; eTrans = pop ; push(tTrans + eTrans)
|	*Term*	
Term →	*Term* * *Factor* #2	#2 fTrans = pop; tTrans = pop ; push(fTrans * tTrans)
|	*Factor*	
Factor →	#3 **intlit**	#3 push(**intlit**.value)

The order matters! In reverse order.

Benefits of using Action Numbers: grammar transformation doesn't affect parse tree. Action numbers can go into the parse table.

Now we finished the topic of parser (of LL(1)).
Next topic: Semantic analyzer

Lec 13 Semantic Analysis with Emphasis on Name Analysis.

Static semantic analysis:

① Name analysis
For each scope, { process declarations, add to symbol table
process statements, update IDs to point to entries

② Type analysis
process statements: use symbol table info to determine type of each exp

Practically feasible to catch errors:
undeclared identifiers;
Multiple declared identifiers,
ill-typed terms.

Static scope (compile time) v.s. Dynamic scope (runtime)