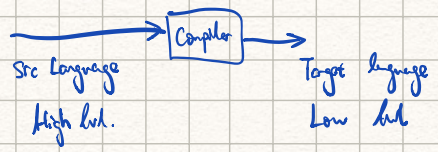
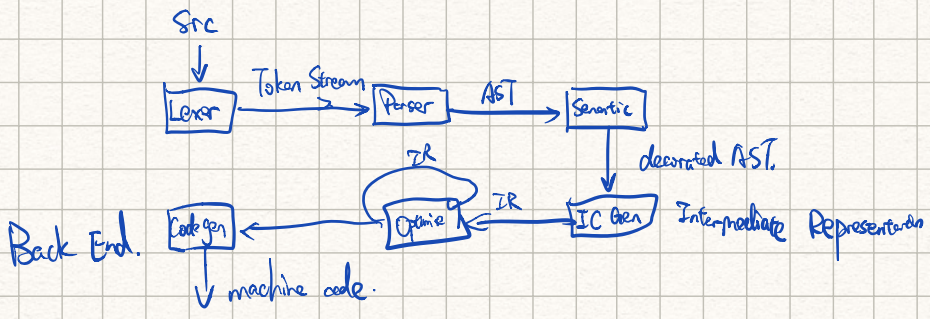
Compiler



Compilers bridge Semantic gap

Front End.

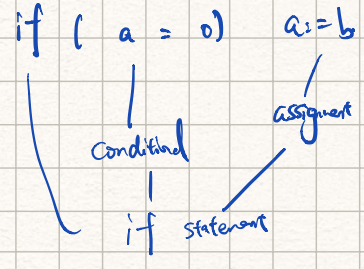


Lexical Analysis.

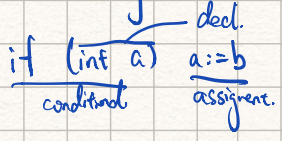
Delimiters define words ␣ \n. \t /...

Lexemes Token Types

Syntax Analysis

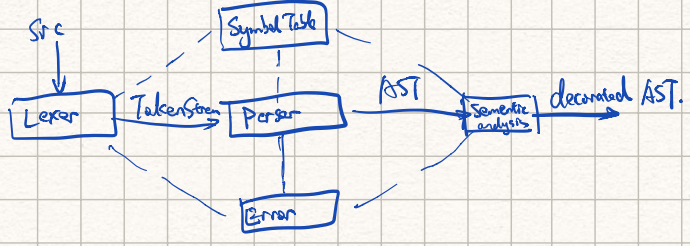


Semantic analysis



Wrong semantics

Error Handling && Symbol Table.



Regular Expressions

Lexeme.

Def: a contiguous seq. of chars that form a lexical unit in the grammar of language.

Token Types.

Lexemes	Token Type	descript
if, while	IF, WHILE, FOR ...	keywords
is, tmp1	ID	identifiers
10, 0, 15	INT	integers
2.0, a3	REAL.	floats
(,),	LPARENS, RPARENS	Symbols

We use Regex to specify lexemes.

Defining a Language

Σ finite set of symbols

Ex. $\Sigma = \{a, b\}$.

Regex R.	L(R).
a	{a}.
a|b	{a, b}.
ab	{ab}.
(ab)*	{ε, ab, abab ...
(a|ε)b	{b, ab}
(a|b*)a	{aa, a, ba, bba ...
(ΣΣΣ)*	{w | |w| mod 3 == 0}

Associativity

Operator priority * > . > |

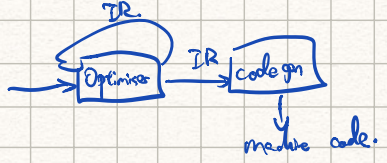
concat usually skips

Syntatic sugar

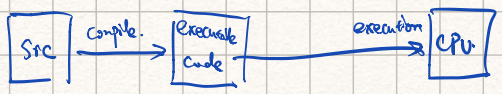
a^+	aa^*
a?	a|ε.
[abc]	a|b|c.
[a-z]	a|b ... |z=Σ
	a|b ... |z=Σ
[^a-c]	Σ - {a, b, c}

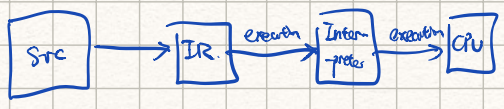
Ambiguity

All phase.



To compile or to interpret.





REPL
Read - Evaluate - Print Loop.
scripting shell languages

if a := 0

if a := 0 or if a := 0
If IDENTIFIER

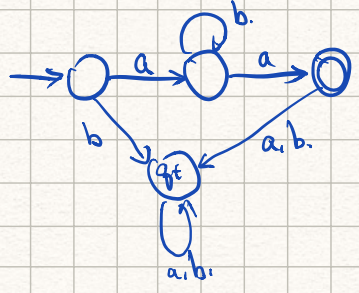
Disambiguation.
Longest match / rule priority

What's Next.?
PSA FSM Finite State Automata

Exercise.
1. $(aa)^*$
2. $//([^/"]^* \mid "\Sigma^*" \mid /[^/]^*)//$

Lexical Analysis.
input → Acceptor for L. → Yes if $i \in L(R)$
No if $i \notin L(R)$

Example DFA for ab^*a over $\Sigma = \{a, b\}$.



NFA. $(\Sigma, Q, \delta, q_0, F)$
Σ: alphabets, Q: states, δ: transitions, q_0: start state, F: set of acc. states

$q_0 \in Q$.
$F \subseteq Q$ set of acc. states

$Q \times \Sigma_\varepsilon \to P(Q)$
$P(Q)$: power set.

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ ε - transition.

$2^Q \Leftrightarrow$ power set of Q.

If $Q = \{a, b, c\}$.
2^Q / $P(Q) = \{a, b, c\}$

$2^Q = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a\,b\}, \{a\,c\}, \{b\,c\}, \{a\,b\,c\}\} \Rightarrow 2^{|Q|}$

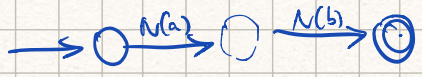
NFA always get the accept route if accepted

parallel interpretation / perfectly lucky

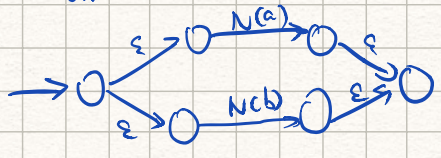
	DFA vs.	NFA.
For each symbol	fully determined.	choose i) ε-transition. ii) which transition.
Accept	ends in $q \in F$	contains a path from $q_0 \to F$.
Implementation	Table driven Speed ↑ Space ↓.	possible states Speed ↓ Space ↑

Regex → NFA. (Thompson's Construction)

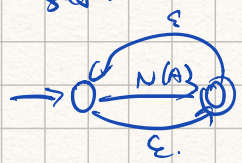
Concat.



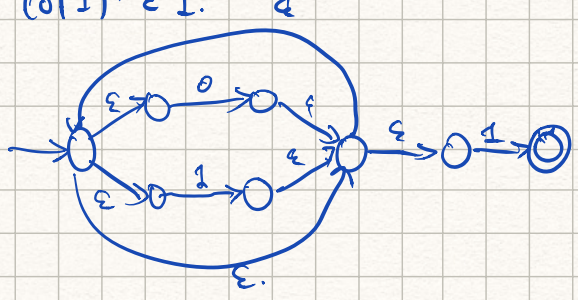
Union.



Star.



$(0|1)^* \varepsilon 1.$



NFA → DFA.

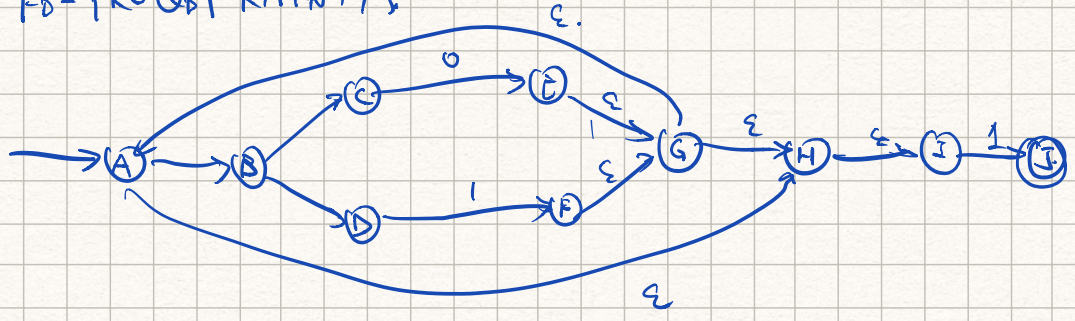
Theorem.

NFA and DFA are equivalent

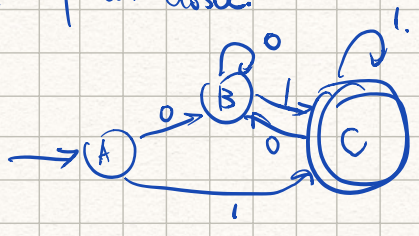
$Q_D \subseteq P(Q_N)$.

$\forall R \in Q_D, \forall a \in \Sigma \quad \delta_D(R,a) = \bigcup_{r \in R} \varepsilon\text{-closure}(\delta_N(r,a))$

$F_D = \{R \in Q_D \mid R \cap F_N \neq \emptyset\}$

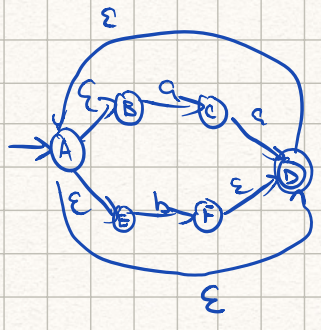


Use epsilon closure.

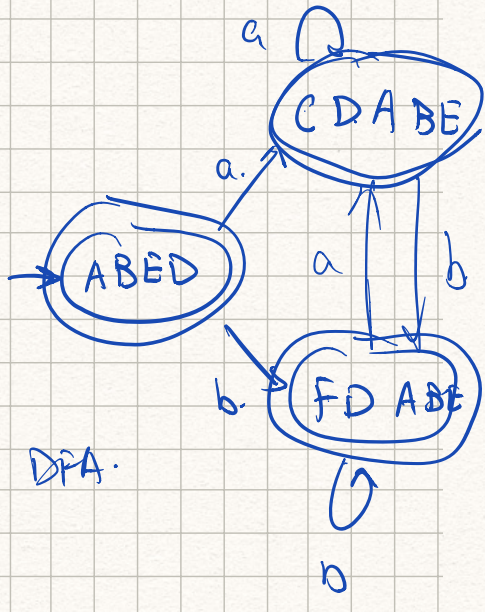


Exercise.

$(a|b)^*$ →

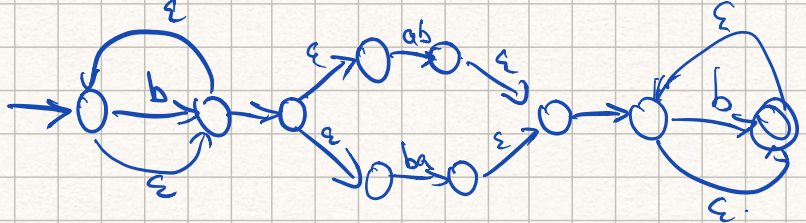


NFA



DFA.

$b^*(ab|ba)b^*$

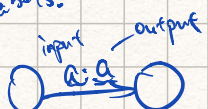


Finite State Transducers

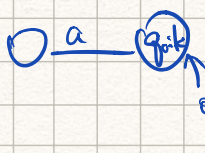
input alphabets

output alphabets.

Mealy machine. $\delta = Q \times \Sigma_\varepsilon \to \Gamma_\varepsilon \times Q$



Moore Machine. $\delta = Q \to \Gamma \varepsilon . \times Q$



FST in lexer label transitions to error states / report lexing errors

Context Free Grammer.

$G = (N, T, S, R)$ — start symbol. — grammar rules

non-terminals

terminals.

EXPR → if EXPR then EXPR else EXPR. fi
| while EXPR loop EXPR pool.
| id.

For a grammar G, $L(G)$ is $\{a_1 \cdots a_n \mid S \xrightarrow{*} a_1 \cdots a_n \text{ where } a_i \in T\}$

all set of strings a grammar can generate

In CFG,

str $\in L(G)$ is a yes-or-no.

also need to produce a parse tree.

handle errors gracefully

need an implementation CUP

The form of a grammar is important. (ambiguities)

The Chomsky Hierarchy

a right-regular grammar is the gram $G=(N,T,S,R)$

Rs all obey the constraints

$A \to a$
$A \to aB$
$A \to \varepsilon$ $\quad A, B \in N$ and $a \in T$.

for left-regular.

$A \to Bc$.

if left and right are mixed $\to$ CFG.

Grammar.	Production Constraint	Automata.
Regular (right)	$A \to a \mid aB \mid \varepsilon$.	FSA. NFA/DFA.
Context free.	$A \to \alpha$.	PDA.
Context Sensitive	$\alpha A \beta \to \alpha \delta \beta$	LBA Linear Bounded Automata
Universal.	$\alpha \to \beta$.	TM.

Derivations

a seq. of sentential forms connected by the application of a single production from S.

$$S \to X_1 \cdots X_n \to \cdots \to Y_1 \cdots Y_m.$$

$$\forall\, x_i \to y_j \in R.$$

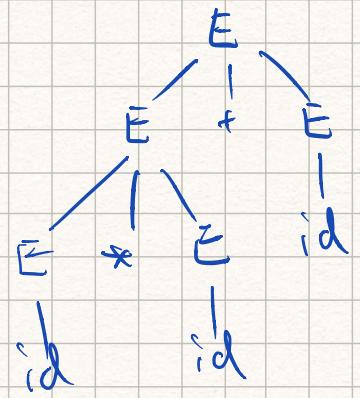
Parse Trees

S is the root.

$X \to Y_1 \cdots Y_n$. $\quad Y_1 \cdots Y_n$ are children of X

E → E+E.
→ E*E
⇒ (E)
→ id.

String id*id+id

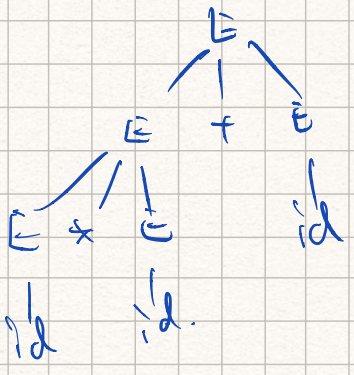


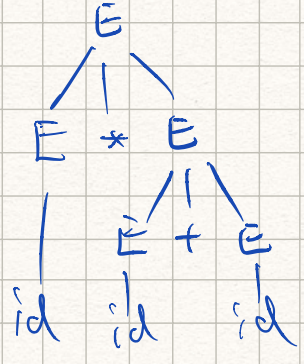
A parse tree has terminals at leaves
non-terminals as interior nodes
in-order gives the original input.

← left-most derivations
at each step / replace left non-terminal.
right - —
— ——— — righ— — -

They produce SAME result.
it's just the production order differences

Two Parse Trees





Ambiguity.
a grammar is ambiguous if
it generates >1 left most
parse tree for some string.

it's BAD can leave
programs ill-defined.

6*4+2
E+E first gives 26
E*E first gives 36

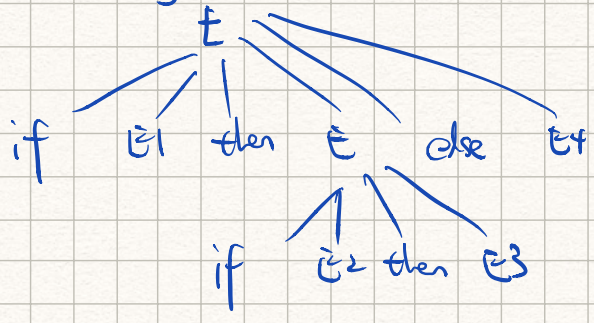
Handling ambiguities

1. $E \to E' + E \mid E'$ ← too complicated

$E' \to id * E' \mid id \mid (E) * E' \mid (E)$

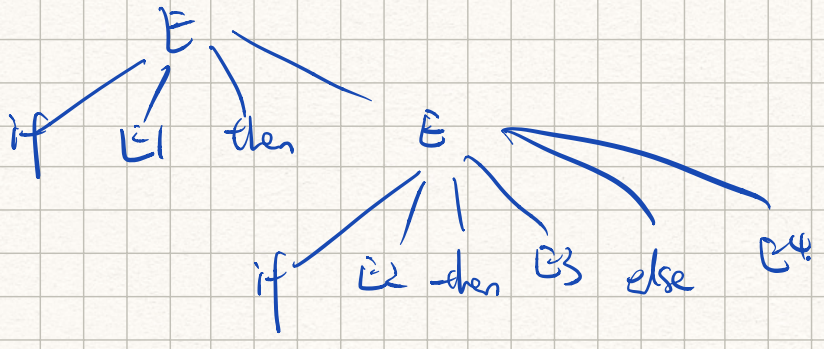
2. Or forces precedence of * over +

Example Dangling Else



← not what we want.

We want



Ambigous definition can simplify a grammer.

Instead of use hacky grammer.

use natural grammer. / with disambiguating annotations.

% left + %left *

precedance decl.

Top Down Parsing

parse tree constructed from Top → down.

left → right.

Recursive Descent Parsing.

Def parsing. find parse tree for tokens $t_1 \cdots t_n$.

Def Recursive Descent Parsing.

From start symbol, try all productions exhaustively

- The fringe of the parse tree is $t_1 \cdots t_k A \cdots$
- Try $A \to BC \mid CD \cdots$

$t_1 \cdots t_k BC \cdots$

- Backtrack when fringe doesn't match prefix of string
- Stop when all terminals

Consider $S \to Sa$.

$S \to Sa \to SSa \cdots$

left recursive Grammar

No fringe to match.

∴ Eliminate left recursion

$S \to S\alpha \mid \beta$

$S \to \beta S'$

$S' \to \alpha S \mid \varepsilon$.

$S \to 1 \mid S0$

$S \to 1S'$

$S' \to 0S' \mid \varepsilon$

In general.

$S \to S\alpha_1 \cdots S\alpha_n \mid \beta_1 \mid \beta_2 \cdots \beta_m$.

$S \to \beta_1 S' \mid \beta_2 S' \cdots \beta_m S'$

$S' \to \alpha_1 S' \mid \alpha_2 S' \cdots \mid \varepsilon$.

Indirect Left Recursion.

$S \to A\alpha \mid \delta$

$A \to S\beta$.

If put A back in $S \to S\beta\alpha \mid \delta$.

$S \to \delta S'$

$S' \to \beta \alpha S' | \varepsilon$.

ε-masked Recursion.

Def. non-terminal is nullable if $N \to^* \varepsilon$.

$S \to YS \mid q$

$Y \to AxB \mid Cc \mid cd \mid \varepsilon$.

Y is nullable. → S is left recursive.

Put Y in S

$S \to AxBS \mid CcS \mid cdS \mid q$.

Don't need $S \to S$ from $Y \to \varepsilon$. ← useless

Repeat.

Eliminate

Unreachable rules and $N \to N$

Direct left recursion via transformation

Indirect via substitution

ε-masked recursion

Until no elimination applies

Survey of Recursive Descent (Top-Down)

- Simple / general.
  - left recursion elimination.
  - can be done automatically
- Unpopular. because of backtracking
  - inefficient.
  - outdated (LOL)

We can avoid back tracking using Predictive Parsers

LL(k) grammars ↑↑↖ LL(1) Languages.

Left to Right — Leftmost derivation. — lookahead

LL(1) means that for each nonterminal and token, only one production leads to a successful parse.

In a 2D lookup Table.

Usually LL(1).

Consider grammer:

$E \to T + E \mid T$

$T \to int \mid int * T \mid (E)$

for T two productions start with int.

E not clear what to do.

S.t. left factoring is needed for LL(1)

$S \to ID \mid ID\ [E] \mid$ is not LL(1) two alternatives start with ID

In general.

$S \to AB_1 \mid AB_2 \mid \cdots \mid AB_n \mid$ <rules not start by A>.

S.t.

$S \to AA' \mid$ <rules not prefix A>

$A' \to B_1 \mid B_2 \cdots \mid B_n$ where B_i can be ε

$E \to T + E \mid T$

$T \to int \mid int * T \mid (E)$

$E \to T\,E'$

$E' \to +E \mid \varepsilon.$

$T \to int\ T' \mid (E)$

$T' \to *T \mid \varepsilon$

$ EOF.

	int	*	+	(	)	$
E	TE'			TE'		
E'			+E		ε	ε
T	intT'			(E)		
T'		*T	ε		ε.	ε.

Blak entries in table error.

No way to get a string start with "*" from E

Method similar to recursive descent, except.

For each nonterminal S, look at next - a.

choose [S, a]

Use stack to keep track of non-terminals

reject on error state.

accept on EOF.

LL(1) Example. int * int.

Stack	Input	Action
$E\$$	int * int $	TE'
$TE'\$$	int * int $	in T'.
$intT'E'\$$	int * int $	termind
$T'E'\$$	* int $	$*T$.
$*TE'\$$	* int $	termind.
$TE'\$$	int $	in T'
$intT'E'\$$	int $	termind
$T'E'\$$	$	ε.
$E'\$$	$	ε
$\$$	$	ACCEPT.

Constructing LL(1) Parsing Table

LL(1) language are those defined by LL(1) parsing table.

- No multiple definition for one table entry.
- If there is a table.
  parsing simple and fast.
  no backtracking necessary

LL(1) parse table is a matrix $|N| \times (|T|+1)$

$T[A, b] = \alpha \to \beta$. $A \to \beta$

$T[A, b] = \to \varepsilon$ discard A.

Consider state $S \to \beta A \gamma$

consider A. b is the lookahead.

1. $b \in$ prefix A. $b \in$ First(A)
2. b is an expansion of A but. $A \to \varepsilon$ b belongs to another expansion of γ.

$b \in$ Follow(A) $\varepsilon \in$ First(A)

Example

$E \to TX$ $X \to +E \mid \varepsilon$

$T \to (E) \mid int\ Y \qquad Y \to *T \mid \varepsilon$

First (T) = {(, int}
First (E) = First (T)
First (X) = {+, ε}
First (Y) = {*, ε}

Follow Sets

$S \to^* \beta A \gamma$ Follow (A) = First (γ) − ε ∪ First (γ_1) − ε if $\gamma \to \varepsilon$

$\varepsilon \notin$ Follow (X)

Fw (E) = {$} ∪ {)} ∪ Fw (X)
= Fw (E) ∪ {$,)}
= {$,)}

Fw (X) = Fw (E) = {), $}.

Fw (T) = {First (X) − {ε}} ∪ Fw (Y)
∪ Fw (E)
= {+} ∪ Fw (T) ∪ {), $}
= {+,), $}

Fw (Y) = Fw (T) = {+,), $}

Construct a parse table.

$A \to \alpha$.
$b \in$ First (α) − {ε}. $T(a, b) = \alpha$.

$a \to^* \varepsilon$ $b \in$ Follow (a) do.
$T(a, b) = \varepsilon$

Notes on LL(1)

G is not LL(1) if
- G is ambiguous
- G is left recursive
- G is not left-factored

Bottom Up Parsing

Handles have grammer.
just as efficient.

Consider $E \to E + (E) \mid int$ ← not LL(1) / due to left recursion.

Idea.
str ← input.
repeat.
Identify β has been excised. in str. (rightmost) $A \to \beta$ is a production
reduce $\beta \to A$. → str becomes $\alpha A \gamma$
until str = S

Example

reversely a right most derivation.

int + (int) + (int)

→ E + (E) + (E)

→ E + (E)

→ E

Bottom-up parsing

Idea:

Split the string into two substrings.

- right substring → a string of terminals
- left substring → NT and Ts.

divide point marked by |

Initially | x1 x2 ... xn.

Two things to do: Shift Reduce.

Shift: move | →

Reduce: Apply inverse. E + (E)| → E|

Implement with a stack

char left of |.

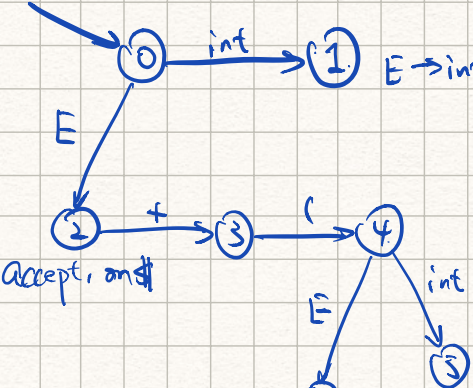
Shift pushes a new element in.
Reduces pops ts and nts out, push a NT in.

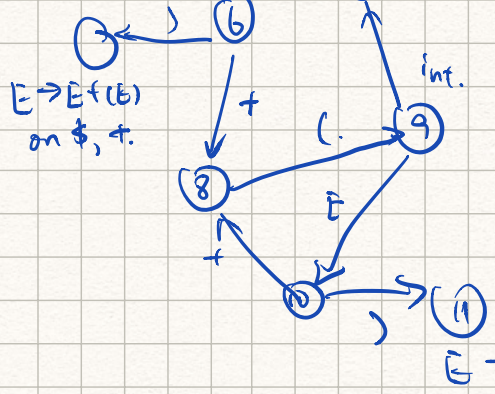
Key Issue: When to shift/reduce?

- Decide based on stack's content and a lookahead.
- Idea: use FSA. Σ = NUT.

run FSA on stack / check state X and lookahead s

if X has a transition label s. shift.
else reduce.





after every reduce, return to 0.

FSA represented as a 2D table.

FSA states are rows

Terminals / NTs are columns

After every shift/reduce, we run FSA again / (ineffiecent)
for each stack element, remember which state FSA on.

$a_i \in \Sigma = NUT.$

$\sigma_i \in Q$

LR parser maintains a stack $\langle \sigma_1, \alpha_1 \rangle \cdots \langle \sigma_n, \alpha_n \rangle$

We jump to state σ_n after shift/reduce

Let I$ be input.
$j = 0$ be index into I, σ_0 is FSA's start state.
stack = push ($\langle \sigma_0, \varepsilon \rangle$)
repeat
$\langle \sigma, \alpha \rangle$:= top stack
Switch action $(\sigma, I[j])$:
shift σ_t: push($\langle \sigma_t, I[j++] \rangle$)
reduce $X \to RHS$:
|RHS|. pop (stack.
push ($\langle X, goto, [\sigma, \alpha] \rangle$)
accept: halt.
error halt & err.

Constructing FSA table

LR(1) item a pair $X \to \alpha \cdot \beta, a$.

$X \to \alpha \cdot \beta$ is a production marked by what is on the stack. α remains to be handled β.

a is a terminal.

LR(1) means 1 lookahead.

$[X \to \alpha \cdot \beta, a]$ describes a context.

X followed by a.

top of stack has α.

Need to find a prefix derived from βa next.

Convention new start symbol $S' \to \cdot S, \$$ empty stack

LR(1) items

In $E \to E + \cdot (E), +$

we can shift. if next is (

$E \to E + (\cdot E), +$.

In $E \to E + (E) \cdot, +$.

we can reduce $E \to E + (E)$ if a + follows

Consider $E \to E + (\cdot E), +$

↑

expect has a prefix from E and suffix is) +.

E has two productions:

↓

$E \to \cdot int,)$

$E \to \cdot E + (E),)$.

The Closure Operation.

For $G = \langle N, T, R, S \rangle$

Closure (items) =

repeat

$\forall [X \to A \cdot Y\beta, a] \in items$

$\forall Y \to \gamma \in R$.

$\forall b \in First(\beta a)$

$items = item \cup [Y \to \cdot \gamma, b]$

until items unchanged.

Example

$S \to E$

$E \to int \mid E + (E)$

Closure($S \to \cdot E, \$$)

$S \to \cdot E, \$$

$E \to \cdot E + (E), \$$

$E \to \cdot int, \$$

$E \to E + (E), +$

$E \to int, +$

$\Rightarrow$

$S \to \cdot E, \$$

$E \to \cdot E + (E), \$ / +$

$E \to int \quad \$ / +$

FSA state is a closed set of LR(1) items

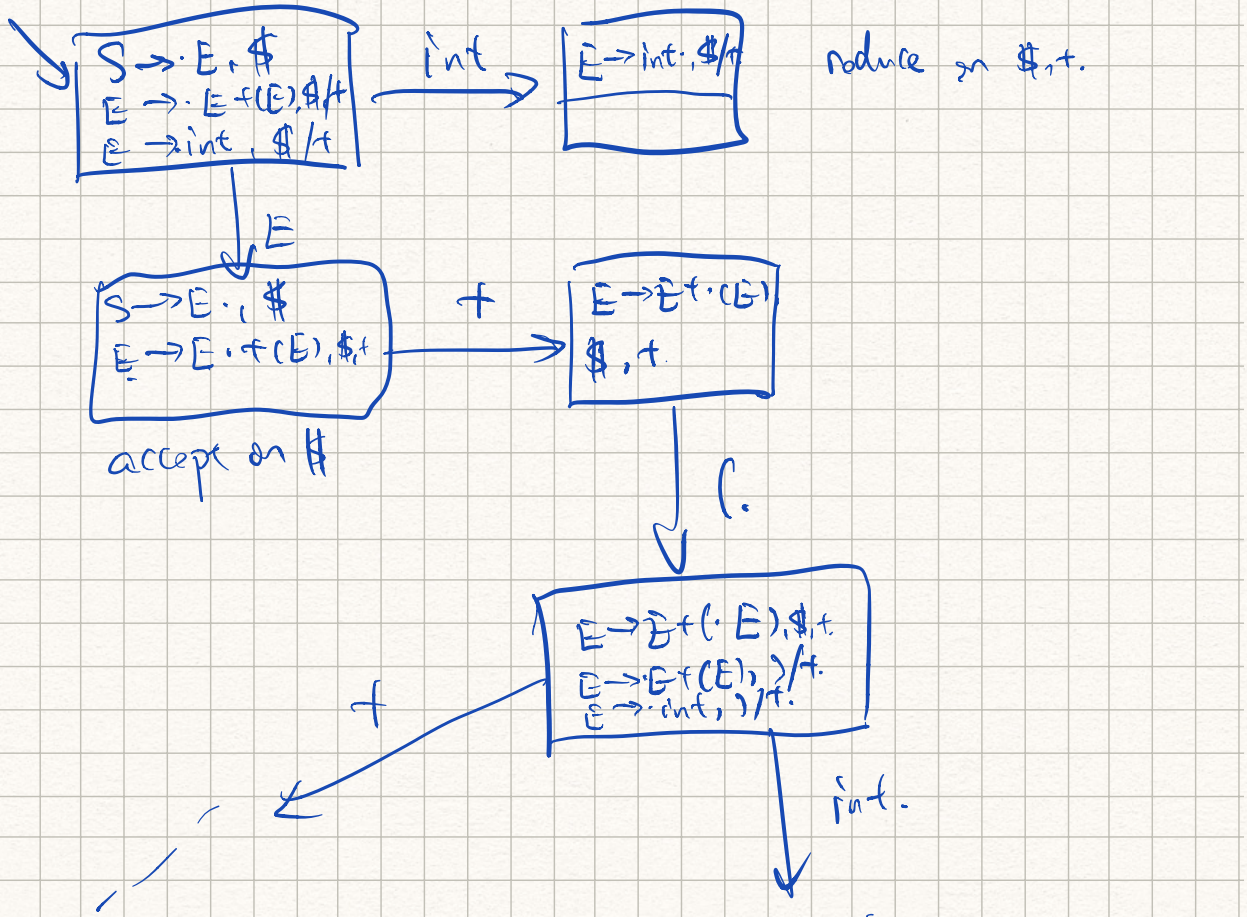
Start state contains $[S \to \cdot E, \$]$

a state that contains $[X \to a \cdot, b]$

reduce $X \to a$ on b lookahead

Building LR(1) Transitions

$[X \to A \cdot \gamma\beta, b]$ has a transition labeled γ.



Also write-up

workSet := S_0 = closure $([S' \to \cdot S, \$])$; $Q = \emptyset$ $\delta = \emptyset$

while (workSet $\neq \emptyset$)

S_s = choose(workSet) removes an item set.

$Q := Q \cup \{S_s\}$

$\forall [X \to \alpha \cdot y\beta, t] \in items(S_s)$

S_t = transition (S_s, y)

$\delta = \delta \cup \{S_s \xrightarrow{y} S_t\}$

workSet = workSet $\cup \{S_t\}$

$F := \{q \in Q \mid \exists i \in items(q) \quad i = [Y \to \alpha \cdot, \$]\}$

return $\langle Q, \delta = N \cup T, \delta, F, S_0 \rangle$

labels are implicitly items in a node

For $G = \langle N, T, R, S \rangle$

1. $\forall n \in N$, compute $F(n)$
2. $G' = \langle N \cup \{S'\}, T, R \cup \{S' \to S\}, S' \rangle$ add Rule $S' \to \cdot S, \$$.
3. Run FSA.
   - create item sets.
   - add transitions
4. Turn FSA to parse Table

The algo builds a NFA

δ doesn't have all transitions to all states

no-ε transitions / ambiguities

It's trivially a DFA if we connect all unwritten transitions to trash.

Shift reduce conflict.

$X \to [A \cdot \alpha\beta, b]$.
$Y \to [\Gamma \cdot, \alpha]$?? Shift or reduce on α.

[S → if E then S., else]
[S → if E then S. else S, x]

Default is not shift. (correct in this case)

Ex. le

Example

$E \to E+E \mid E*E \mid int$

$[E \to E+E\cdot, +]$
$[E \to E\cdot +E, +]$

reduce. + have same precedence.
$E \to E+E\cdot$ rule
+ is terminal. left assoc.
∴ reduce.

$[E \to E*E, +]$
$[E \to E+E, +]$

reduce $E \to E*E$ Higher precedence than +.
terminal.

Dagling else

else precedence > then.

$S \to$ if E then S. | else
if E then S else S, x.

$S \to$ if E then S lower then. else

Shift

Best avoid use of too much precedence.

Reduce - Reduce Conflicts

Example. Given Grammer.

$S \to \varepsilon \mid id \mid id\ S$

$S' \to \cdot S, \$$
$S \to \cdot id, \$$
$S \to \cdot id\ S, \$$
$S \to \cdot\ , \$$

S
id.

$S \to id\cdot , \$$
$S \to id\cdot S, \$$

$S \to \cdot\ , \$$
$S \to \cdot id\ S, \$$

Reduce/Reduce Conflict $S \to \cdot id, \$$

Solve this by rewriting $S \to idS \mid \varepsilon$.

Using parser generators

precedence declarations

best parser gen do not construct FSA (DFA/NFA)

LR(1) Tables are big.

Many states are similar. ⟶ merge together.

The Core of a set of LR items.

$\{[X \to \alpha \cdot \beta, b], [Y \to \gamma \cdot \delta, d]\} \xrightarrow{\text{Coreify}} \{X \to \alpha \cdot \beta, Y \to \gamma \cdot \delta\}$

Same core / can be merged.

⇓

LALR(1) states.

Lookahead LR States (statistically 10 times fewer)

LALR can have more reduce-reduce conflicts (in practice rare)

LALR are efficiency hack.

Reasonable programming languages have a LALR(1) grammar.