Formal Languages and Compiler Design

Course 1.:

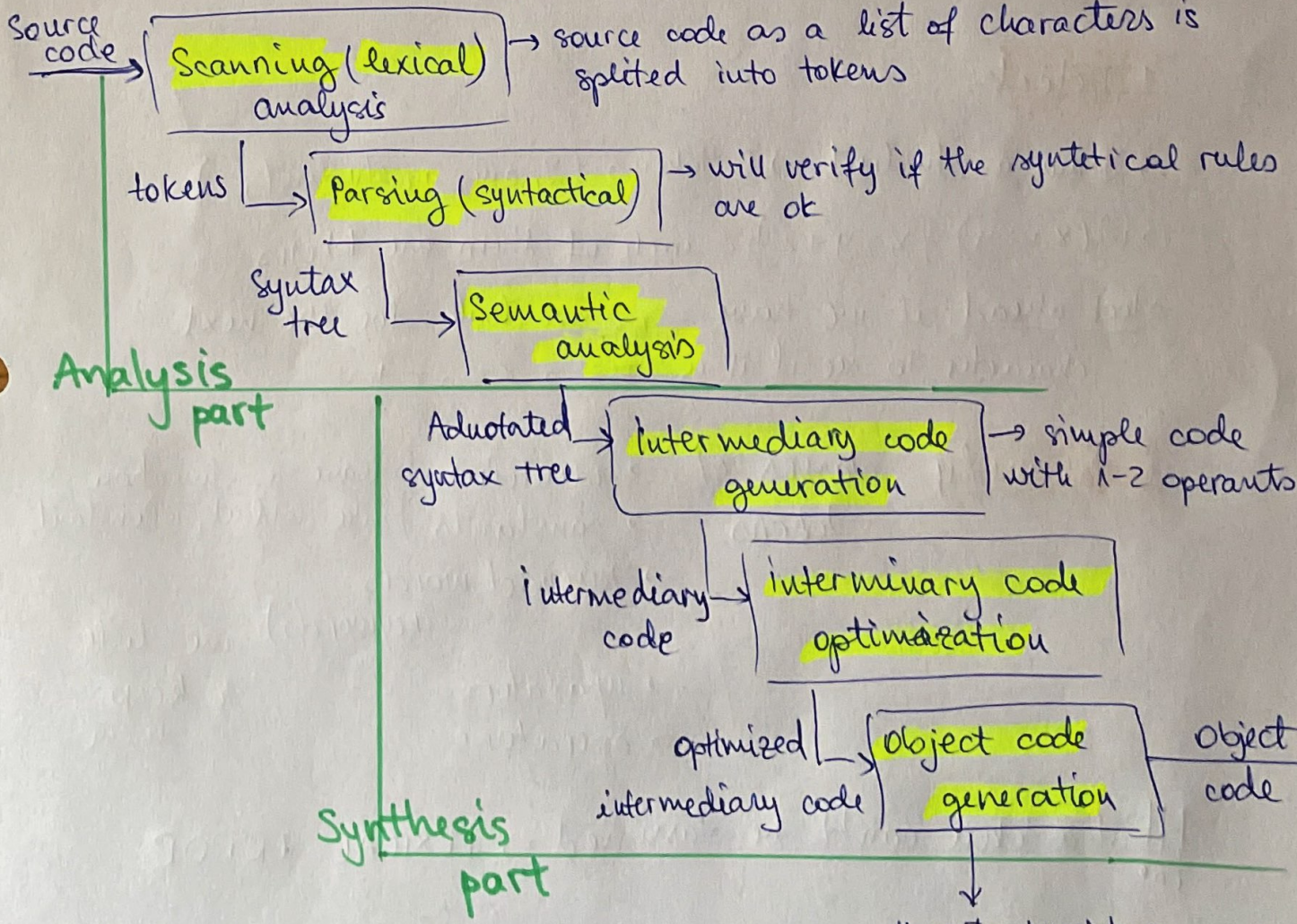
What is a compiler?

Source code/ program → | compiler | → object code/ program

↓ looks at the whole code

interpretor: looks line by line and simulates execution

Structure of a compiler:



Error handling: logical errors are usually not detected by the compiler

Symbol table management:

→ it is important that the identifier is called n or a
→ collects all symbolic names
→ identifier = name to variables or functions
→ symbolic names that are not identifiers id=exp.

Chapter 1: Scanning

Def: Treats the source code as a sequence of characters, detect lexical tokens, clasify and codify them

PIF - program internal form
ST - symbol table

1. Detect:

→ I am a student. => 5 tokens (I, am, a, student, .)
→ if(x==y){x=y+2} each of them are tokens
→ look ahead: if we have =, < we look for the next character to see if it is ==, <=, =+

2. Classify: classes of tokens

- → identifiers } have a function like isIdent, isConstant
- → constants }
- → reserved words (keywords) } use lists to keep them
- → separators
- → operators

!!! If a token can't be classified => LEXICAL ERROR

3. Codify:

→ replace identifier with id

→ identifier, constant ⇒ Symbol Table

→ PIF ⇒ array of pairs <x pairs (token, position in ST)>

Symbol Table:

Def: Contains all information collected during compiling regarding the symbolic names from the sourc program

↓ identifiers, constants

Organization:

* unsorted table: O(n)
* sorted table (alphabetic, numeric): O(lgn)
* Binary search tree (balanced): O(lg n)
* Hash table O(1). [open bucket: with linked lists; close bucket: search for the first open position]

For lab: hash function = sum of ASCII codes of char

Formal Languages
Basic Notations

Grammar:

Definition: A (formal) grammar is a 4-tuple $G=(N, \Sigma, P, S)$ with the following meanings:

- N - set of nonterminal symbols, $|N| < \infty$
- Σ - set of terminal symbols (alphabet), $|\Sigma| < \infty$
- P - finite set of productions (rules)
- $S \in N$ - start symbol / axiom.

Remarks:

1. $(\alpha, \beta) \in P$ is a production denoted $\alpha \to \beta$
2. $A^* = \{a, aa, aaa, \dots\} \cup \{a^0\}$
   $A = \{a\}$
   $A^+ = \{a, aa, aaa\}$
   $x^0 = \varepsilon$
3. Direct derivation:
   $\alpha \Rightarrow \beta$, $\alpha, \beta \in (N \cup \Sigma)^*$ if $\alpha = x_1 x y_1$ and $x \to y \in P$
   $\beta = x_1 y y_1$
   (x is transformed in y).

Definition: Language generated by a grammar $G = (N, \Sigma, P, S)$ is:

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$$

Remarks:

1. Operations defined for languages
   $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$, $L^+ = \bigcup_{k>0} L^k$
   Concatenation: $L = L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$
2. Two grammars G_1 and G_2 are equivalent if they generate the same language $L(G_1) = L(G_2)$
3. Context free grammar: $A \to \alpha \in P$, where $A \in N$ and $\alpha \in (N \cup \Sigma)^*$
4. Regular grammar: $A \to aB \mid a \in P$

Notations:

- $A, B, C, \ldots$ – nonterminal symbols $\rightarrow N$
- $S \in N$ – start symbol
- $a, b, c \in \Sigma$ – terminal symbols
- ε – empty word
- $x, y, z, w \in \Sigma^*$ – words
- $x, y, u \ldots \in (N \cup \Sigma)$ – grammar symbols (term or. nonterm.).

Finite Automata:

Definition: A finite automata (~~FA~~) is a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- Q: finite set of states
- Σ: finite alphabet
- δ: transition function: $\delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$
- q_0: initial state $q_0 \in Q$
- $F \subseteq Q$ - set of final states

Remarks:

1. if $|\delta(q, a)| \leq 1$. $\Rightarrow$ deterministic ~~FA~~ (DFA)
2. if $|\delta(q, a)| > 1$ (more than a state obtained as a result) $\Rightarrow$ nondeterministic ~~FA~~ (NFA)

Configuration: $C = (q, x)$ where $\rightarrow q$ state
$\rightarrow x$ unread sequence from input $x \in \Sigma^*$

Initial configuration: (q_0, w) – w - whole sequence

Final configuration: (q_f, ε), $q_f \in F$, ε - empty word.

Relations between configurations:

- $\vdash$ move / transition (simple, one step)

  $(q, ax) \vdash (p, x)$ if $p = \delta(q, a)$

- $\vdash^k$ k move = sequence of k simple transitions $C_0 \vdash C_1 \dots \vdash C_k$
- $\vdash^+$ + move : $C \vdash^+ C'$: $\exists k > 0$ such that $C \vdash^k C'$
- $\vdash^*$ * move (star move) : $C \vdash^* C'$, $\exists k \geq 0$ such that $C \vdash^k C'$

Definition: Language accepted by FA $M = (Q, \Sigma, \delta, q_0, F)$ is $L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash^* (q_f, \varepsilon), q_f \in F\}$

Regular Languages

Regular grammars:

- $G = (N, \Sigma, P, S)$ right linear

  $\forall p \in P$: $A \to aB$ or $A \to b$,

- $G = (N, \Sigma, P, S)$ regular gra
  - * G is right linear gr and
  - * $A \to \varepsilon \notin P$, with the exception that $S \to \varepsilon \in P$, in which case S does not appear in the right hand side of any other production

Regular grammar:
* right linear grammar
  $A \to aB$ or $A \to b$
* $\varepsilon \in$ only for the starting symbol (S_0)
  if $\varepsilon \in S_0 \Rightarrow S_0$ should not appear in the right hand side of any product.

Example:

- $\begin{cases} S \to aA \\ A \to a \end{cases}$ → regular
- $\begin{cases} S \to aA \\ A \to aA \mid \varepsilon \end{cases}$ → NOT regular
- $\begin{cases} S \to aS \mid aA \\ A \to bS \mid b \end{cases}$ → regular
- $\begin{cases} S \to aA \mid \varepsilon \\ A \to aS \end{cases}$ → NOT regular

Regular sets:

Definition: Let Σ be a finite alphabet. We define **regular sets** over Σ recursively in the following way:

1. $\emptyset$ is a regular set over Σ
2. $\{\varepsilon\}$ is a regular set over Σ
3. $\{a\}$ —"— —"— —"—, $\forall a \in \Sigma$
4. If P, Q reg. sets $\Rightarrow P \cup Q, PQ, P^*$ reg sets over Σ
5. Nothing else is a regular set over Σ

Regular expressions:

Definition: Σ-finite alphabet. We define **regular expressions** over Σ recursively:

1. $\emptyset$ is a regular expression denoting the reg set $\emptyset$
2. ε —"— —"— —"— —"— $\{\varepsilon\}$
3. a —"— —"— —"— —"— $\{a\}, \forall a \in \Sigma$
4. If p, q are reg. expr. denoting P, Q then
   $p+q$ for $P \cup Q$, pq for PQ, $p^* \to P^*$
5. Nothing else is a regular expression

Remarks:

1. $p^+ = pp^*$
2. Priority of operations: $*$, concat, $+$ (from high).
3. Two regular expressions are equivalent iff they denote the same regular set

Example: $(0+1)^*$ denotes $\{\varepsilon, 0, 1, 00, 01, 11, 10 \dots\}$

0^*1^* denotes $\{\varepsilon, 0, 1, 01, 00, 11, \dots\}$

Theorem :

A language is a regular set if and only if it is a linear language

!!! Use to find the Reg Expression from a grammar !!!

$G = (\{S, A, B\}, \{0, 1\}, P, S\}$

$P: S \to 0A \mid 1B \mid \varepsilon$

$A \to 0B \mid 1A$

$B \to 0S \mid 1$

$$X = aX + b \Rightarrow X = a^*b \quad (*)$$

$$\begin{cases} S = 0A + 1B + \varepsilon \\ A = 0B + 1A \\ B = 0S + 1 \end{cases} \Rightarrow A = 0(0S + 1) + 1A$$

$A = 00S + 01 + 1A$ from $(*)$.

$\Rightarrow A = 1^*(00S + 01)$

$S = 0A + 1B + \varepsilon \to$ Replace A, B

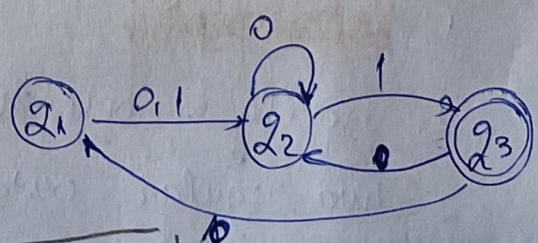
$S = 01^*(00S + 01) + 1(0S + 1) + \varepsilon$

$S = \underbrace{(01^*00 + 10)}_{a} S + \underbrace{01^*01 + 11 + \varepsilon}_{b}$

$\Rightarrow S = (01^*00 + 10)^*(01^*01 + 11 + \varepsilon)$

Theorem :

A language is a regular set if and only if it is accepted by ~~a~~ FA



$$\begin{cases} q_1 = q_3 0 + \varepsilon & (1) \\ q_2 = q_1 0 + q_1 1 + q_2 0 + q_3 0 & (2) \\ q_3 = q_2 1 & (3) \end{cases}$$

$$X = Xa + b \Rightarrow X = ba^*$$

! Solution is the sum of the final states

$3 \to 1 \Rightarrow q_1 = q_2 10 + \varepsilon$

$2 \to q_2 = (q_2 10 + \varepsilon)0 + (q_2 10 + \varepsilon)1 + q_2 0 + q_2 10$

$q_2 = q_2 100 + 0 + q_2 101 + 1 + q_2 0 + q_2 10$

$$q_2 = q_2(100+101+0+10)+0+1$$

$$\Rightarrow q_2 = (0+1)(100+101+0+1)^*$$

$$\Rightarrow q_3 = q_2 1 = (0+1)(100+101+0+1)^* 1.$$

Regular grammars:

Theorem 1: For every regular grammar $G=(N,\Sigma,P,S)$ there exists a FA $M=\{Q,\Sigma,\delta,q_0,F\}$ such that $L(G)=L(M)$

Construct M based on G

$Q = N \cup \{k\}$, $k \notin N$

$q_0 = S$

$F = \{k\} \cup \{S \mid \text{if } S \to \varepsilon \in P\}$

δ: if $A \to aB$ then $\delta(A,a)=B$

if $A \to a$ then $\delta(A,a)=k$

Theorem 2: For any FA $M=\{Q,\Sigma,\delta,q_0,F\}$ there exist a right linear grammar $G=\{N,\Sigma,P,S\}$ such that $L(G)=L(M)$

Construct G based on M

$N = Q$

$S = q_0$

P: if $\delta(q,a)=p$ then $q \to ap \in P$

if $p \in F$ then $q \to a \in P$

if $q_0 \in F$ then $S \to \varepsilon$

Regular sets:

If L_1 and L_2 are right linear languages then $L_1 \cup L_2$, $L_1 L_2$, L_1^* are right linear languages.

Proof:

L_1, L_2 - right linear lang $\Rightarrow \exists\, G_1, G_2$ such that

$G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $L_1 = L(G_1)$

$G_2 = (N_2, \Sigma_2, P_2, S_2)$ and $L_2 = L(G_2)$. assume $N_1 \cap N_2 = \emptyset$

* $L_1 \cup L_2 \rightarrow a+b$

$G_3 = (N_3, \Sigma_3, P_3, S_3)$

$N_3 = N_1 \cup N_2 \cup \{S_3\}$

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$P_3 = P_1 \cup P_2 \cup \{ S_3 \rightarrow \alpha_1 \mid S_1 \rightarrow \alpha_1 \in P_1 \} \cup$

$\cup \{ S_3 \rightarrow \alpha_2 \mid S_2 \rightarrow \alpha_2 \in P_2 \}$

$\Rightarrow G_3$ - right linear language

and

$L(G_3) = L(G_1) \cup L(G_2)$

* $L_1 L_2$ (concatenation) $\rightarrow ab$

$G_4 = (N_4, \Sigma_4, P_4, S_4)$

$N_4 = N_1 \cup N_2$

$\Sigma_4 = \Sigma_1 \cup \Sigma_2$

$S_4 = S_1$

Keep the second

And change

$S \rightarrow a$ with

$S \rightarrow aS_1$, S_1 - from the first

P_4: $\{ A \rightarrow aB \mid \text{if } A \rightarrow aB \in A \} \cup$

$\{ A \rightarrow a S_2 \mid \text{if } A \rightarrow a \in P_1 \} \cup$

$\{ S_1 \rightarrow x \mid \text{if } S_1 \rightarrow \varepsilon \text{ and } S_2 \rightarrow x \} \cup P_2$

$\Rightarrow G_4$ right linear language

and

$L(G_4) = L(G_1) L(G_2)$

if $\varepsilon \in$ second

$\Rightarrow$ if q - final st. of first

q will also be f. state

* L_1^* // IDEA: Concatenate L_1 with itself

$G_5 = (N_5, \Sigma_1, P_5, S_5)$

$N_5 = N_1 \cup \{S_5\}$

P_5 : $P_1 \cup \{S_5 \to \varepsilon\} \cup$

$\{S_5 \to \alpha_1 \mid S_1 \to \alpha_1 \in P_1\} \cup$

$\{A \to aS_1 \mid \text{if } A \to a \in P_1\}$

Tip: • keep everything for the second language
• for every single term. put $S \to aS$

Lemma:

If L_1 and L_2 are accepted by a FA then $L_1 \cup L_2$, $L_1 L_2$ and L_1^* are accepted by FA.

$M_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$ such that $L_1 = L(M_1)$

$M_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$ such that $L_2 = L(M_2)$

* $L(M_3) = L(M_1) \cup L(M_2)$ → $a+b$

$M_3 = (Q_3, \Sigma_3, \delta_3, q_{03}, F_3)$

$Q_3 = Q_1 \cup Q_2 \cup \{q_{03}\}$

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$F_3 = F_1 \cup F_2 \cup \{q_{03} \mid \text{if } q_{01} \in F_1 \text{ or } q_{02} \in F_2\}$

$\delta_3 = \delta_1 \cup \delta_2 \cup \{\delta_3(q_{03}, a) = p \mid \exists\, \delta_1(q_{01}, a) = p\}$

$\cup \{\delta_3(q_{03}, a) = p \mid \exists\, \delta_2(q_{02}, a) = p\}$

* $L(M_3) = L(M_1)\,L(M_2)$ → ab

$M_4 = (Q_4, \Sigma_4, \delta_4, q_{04}, F_4)$

All final states of 1st takes over the initial states of the 2nd

$Q_4 = Q_1 \cup Q_2$

$q_{04} = q_{01}$

$F_4 = F_2 \cup \{q \in F_1 \mid \text{if } q_{02} \in F_2\}$

$\delta_4(q, a) = \delta_1(q, a)$ if $q \in Q_1 - F_1$

$\delta_1(q, a) \cup \delta_2(q_{02}, a)$ if $q \in F_1$

$\delta_2(q, a)$ if $q \in Q_2$ — 11 —

* $L(M_3) = L(M_1)^*$

$M_5 = (Q_5, \Sigma_1, \delta_5, q_{05}, F_5)$

$Q_5 = Q_1$

$q_{05} = q_{01}$

$F_5 = F_1 \cup \{q_{01}\}$

$\delta_5(q,a) = \delta_1(q,a)$ if $q \in Q_1 - F_1$

$\delta_1(q,a) \cup \delta_1(q_{01},a)$ if if $q \in F_1$

Pumping Lemma:

!!! To decide if a language is regular or not

Theorem (Bar-Hillel, Pumping lemma)

Let L be a regular language. $\exists p \in N$, such that if $w \in L$ with $|w| > p$ then

$w = xyz$, where $0 < |y| <= p$

and

$xy^iz \in L, \forall i \geq 0$

(Proof Course 5 - 4).

Example: $L = \{0^n 1^n \mid n >= 0\}$

Suppose L is regular $\rightarrow$ $w = xyz = 0^n 1^n$

$xy^iz, \forall i \geq 0$

Consider all possible decomposition $\Rightarrow$

I. $y = 0^k \Rightarrow xyz = 0^{n-k} 0^k 1^n$

$xy^iz = 0^{n-k} 0^{ik} 1^n \notin L$

II. $y = 1^k \rightarrow xyz = 0^n 1^k 1^{n-k}$

$xy^iz = 0^n 1^{ik} 1^{n-k} \notin L$

III. $y = 0^k 1^\ell \Rightarrow xyz = 0^{n-k} 0^k 1^\ell 1^{n-\ell}$

$xy^i z = 0^{n-k} (0^k 1^\ell)^i 1^{n-\ell} \notin L$

IV. $y = 0^k 1^k \Rightarrow xyz = 0^{n-k} 0^k 1^k 1^{n-k}$

$xy^i z = 0^{n-k} (0^k 1^k)^i 1^{n-k} \notin L$

$\Rightarrow L$ is not regular

Context Free Grammars (cfg)

- Productions of the form: $A \to \alpha$, $A \in N$, $\alpha \in (N \cup \Sigma)^*$
- More powerful
- Can model programming language

Syntax Tree:

Definition: A syntax tree corresponding to a cfg $G = (N, \Sigma, P, S)$ is a tree obtained in the following way:

1. Root is the starting symbol S
2. Nodes $\in N \cup \Sigma$:
   - internal nodes $\in N$
   - leaves $\in \Sigma$
3. For a node A the descendents in order from left to write are $X_1, X_2, \dots, X_n$ only if $A \to X_1 X_2 \dots X_n \in P$.

Remarks:

* Parse tree = syntax tree (result of parsing)
* Derivation tree - condition 2.2 not satisfied
* Abstract syntax tree (AST) $\neq$ syntax tree

Property: In a cfg $G = (N, \Sigma, P, S)$, $w \in L(G)$ if and only if there exists a syntax tree with frontier w.

Example:

$S \to aSbS \mid c$

$w = aacbcbc$

Leftmost derivation:

$S \Rightarrow aSbS \Rightarrow aaSbSbS \Rightarrow aacbSbS \Rightarrow aacbcbS \Rightarrow$

$\Rightarrow aacbcbc$

Rightmost derivation:

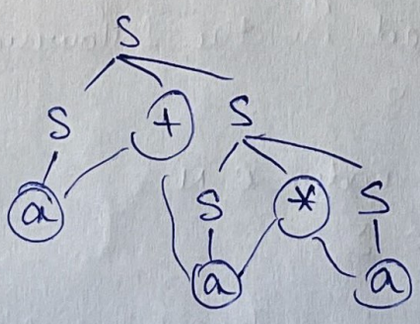
$S \Rightarrow aSbS \Rightarrow aSbc \Rightarrow aaSbSbc \Rightarrow aaSbcbc \Rightarrow$

$\Rightarrow aacbcbc$

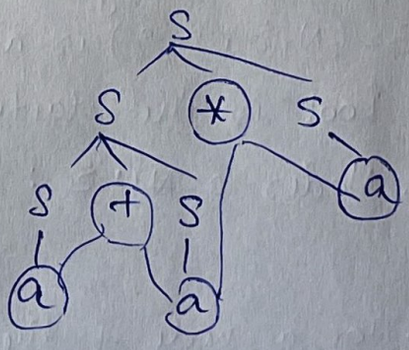
Definition: A cfg $G=(N, \Sigma, P, S)$ is **ambigous** if for a $w \in L(G)$ there exists 2 distinct syntax tree with frontier w.

Example: $S \to S+S \mid S*S \mid a$

$w = a+a*a$



or



Parsing modeled with ctg:

cfg $G=(N, \Sigma, P, S)$:

- N - nonterminal: syntact. constr: declaration, stmt, expression ...
- Σ - terminals, elems of the lang: identifiers, const, reserved words, operators, separators
- P - syntactical rules - expressed in BNF - simple trans
- S - syntactical construct corresp. to program

THEN

Program syntactical correct $\Leftrightarrow w \in L(G)$.

Equivalent transformation of cfg.

Unproductive symbols:

A nonterminal A is **unproductive** in a cfg if it does not generate any word:
$\{w \mid A \overset{*}{\Rightarrow} w \in \Sigma^*\} = \emptyset$

Algorithm 1: Elimination of unproductive symbols

Idea: build $N_0, N_1, \ldots$ recursively (until saturation)

Step 1: $N_0 = \emptyset$ $i = 1$
Step 2: $N_i = N_{i-1} \cup \{A \mid A \to \alpha \in P, \alpha \in (N_{i-1} \cup \Sigma)^*\}$
Step 3: if $N_i \neq N_{i-1}$ then $i = i+1$; go to step 2
else $N' = N_i$
Step 4: if $S \notin N'$ then $L(G) = \emptyset$
else $P' = \{A \to \alpha \mid A \to \alpha \in P \text{ and } A \in N'\}$

Example: $G = (\{S, A, B, C, D\}, \{a, b, c\}, P, S\}$
P: $S \to aA \mid ac$
$A \to AB \mid aB$
$B \to b$
$C \to ac \mid CD$
$D \to b$

$N_0 = \emptyset$
$N_1 = N_0 \cup \{B, D\} = \{B, D\}$
$N_2 = N_1 \cup \{A\} = \{B, D\} \cup \{A\}$
$A \to aB \in \Sigma \in N_1$
$N_3 = \{B, A, D\} \cup \{S\}$
$N_4 = N_3 \cup \emptyset$
$\Rightarrow C$ – unproductive

Inaccesible symbols:

A symbol $X \in N \cup \Sigma$ is inaccesible if X does not appear in any sentential form:
$\forall S \overset{*}{\Rightarrow} \alpha, \; X \notin \alpha$

Algorithm 2: Elimination of inaccesible symbols

Step 1: $V_0 = \{S\}, i = 1$

Step 2: $V_i = V_{i-1} \cup \{X \mid \exists A \to \alpha X \beta \in P, A \in V_{i-1}\}$

Step 3: if $V_i \neq V_{i-1}$ then $i = i+1$, go to step 2
else $N' = N \cap V_i$
$\Sigma' = \Sigma \cap V_i$
$P' = \{A \to \alpha \mid A \to \alpha \in P, A \in N', \alpha \in (N \cup \Sigma)^*\}$

Example:

$S \to aA$
$A \to AB \mid aB$
$B \to b$
~~$D \to b$~~ eliminate D

$V_0 = \{S\}$
$V_1 = \{S, a, A\}$
$V_2 = \{S, a, A, B\}$
$V_3 = \{S, a, A, B, b\}$
$V_4 = \{S, a, A, B, b\}$
$\Rightarrow$ D inaccesible

ε-productions:

A cfg $G = (N, \Sigma, P, S)$ is without ε-productions if
1. $P \not\ni A \to \varepsilon$ (ε-production)
OR
2. $\exists S \to \varepsilon$ si $S \notin rhs(p), \forall p \in P$

Algorithm 3: Elimination of ε-productions

Step 1: construct
$\overline{N} = \{A \mid A \in N, A \overset{*}{\Rightarrow} \varepsilon\}$
1.a. $N_0 = \{A \mid A \to \varepsilon \in P\}$
$i = 1$
1.b. $N_i = N_{i-1} \cup \{A \mid A \to \alpha \in P, \alpha \in N_{i-1}^*\}$
1.c. if $N_i \neq N_{i-1}$ then $i = i+1 \Rightarrow$ go to step 1b
else $\overline{N} = N_i$

Step 2: Let P' - set of productions built:

2.a. if $A \to \alpha_0 B_1 \alpha_1 B_2 \dots B_k \alpha_k \in P$, $k \geq 0$
and for $i = \overline{1,k}$ $B_i \in \bar{N}$ and $\alpha_j \notin \bar{N}$, $j = \overline{0,k}$
$\Rightarrow$ add to P' all prod of the form
$A \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \dots X_k \alpha_k$
where X_i is B_i or ε (not $A \to \varepsilon$)

2b. if $S \in N'$ then add S' to N' and $S' \to S \mid \varepsilon$ to P
else $N' = N$ and $S' = S$

Example:

$S \to aA \mid aAbB$
$A \to aA \mid B$
$B \to bB \mid \varepsilon$

Step 1:
$N_0 = \{B\}$
$N_1 = \{B, A\}$
$N_2 = \{B, A\} \Rightarrow \bar{N} = \{B, A\}$

Step 2:
$S \to aA \mid a$
$S \to aAbB \mid abB \mid aAb \mid ab$
$A \to aA \mid a \mid B$
$B \to bB \mid b$

Single productions:

A production of the form $A \to B$ is called single production or renaming rule

Algorithm 4: Elimination of single productions

For each $A \in N$ build the set $N_A = \{B \mid A \overset{*}{\Rightarrow} B\}$

1.a $N_0 = \{A\}$ $i = 1$
1.b. $N_i = N_{i-1} \cup \{C \mid B \to C \in P \text{ and } B \in N_{i-1}\}$
1.c. if $N_i \neq N_{i-1}$ then $i = i+1$ goto 1b
else $N_A = N_i$

P': for all $A \in N$ do
for all $B \in N_A$ do
if $B \to \alpha \in P$ and not "single" then
$A \to \alpha \in P'$

Parsing

- Cfg $G = (N, \Sigma, P, S)$ check if $w \in L(G)$
- Construct parse tree

How? 1. Top down vs Bottom up
2. Recursive vs linear

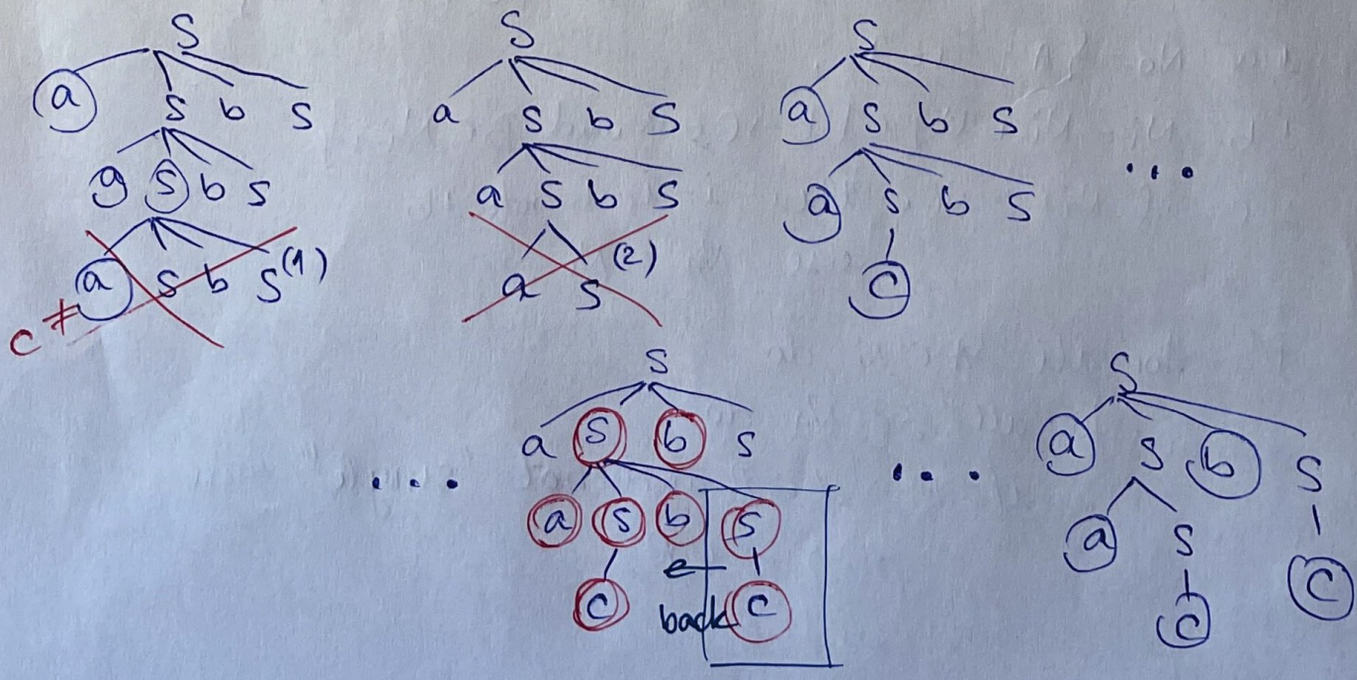
	Descendent	Ascendent
Recursive	Descendent recurs. parser	Ascendent recursive parcer
Linear	LL(k): LL(1)	LR(k): LR(0), SLR, LR(1), LALR

Result – parse tree representation

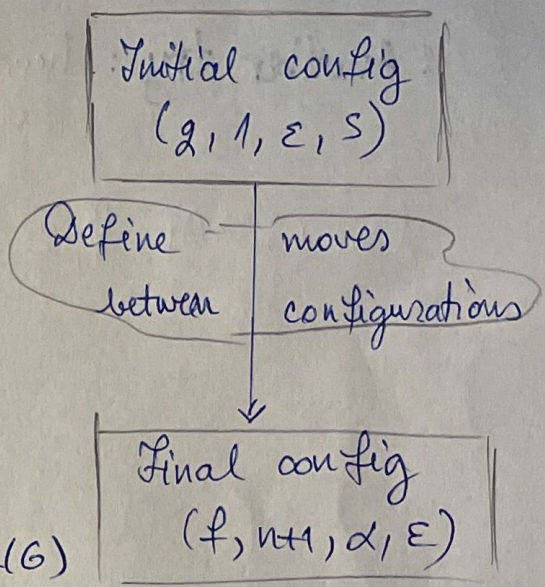
- Arbitrary tree: child nibling repr (left child right sbl)
- Sequence of derivations: $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = w$
- String of productions: index of associated production (which production is used at each derivation step)

Descendent recursive parser

Example $S \to aSbS^{(1)} \mid aS^{(2)} \mid c^{(3)}$ $w = aacbc$



Formal Model:



* Configuration: (s, i, α, β)

where:

- s = state of the parsing, can be
  - → q - normal state
  - → b - back state
  - → f - final state ⟺ succes $w \in L(G)$
  - → e - error state ⟺ insuccess $w \notin L(G)$
- i - position of current symbol in input sequence:
  $w = a_1 a_2 \dots a_n$, $i \in \{1, \dots, n+1\}$
- α = working stack, stores the way the parse is build
- β = input stack, part of the tree to be build

* **Expand**: when head of input stack (β) is a nonterminal

$$(q, i, \alpha, A\beta) \vdash (q, i, \alpha A_1, \gamma_1 \beta)$$

where: $A \to \gamma_1 \mid \gamma_2 \mid \dots$ productions corresp to A
1 - first prod of A

* **Advance**: when head of input stack (β) is a terminal = =current symbol from input

$$(q, i, \alpha, a_i\beta) \vdash (q, i+1, \alpha a_i, \beta)$$

* **Momentary insuccess**: when head of the input stack is a terminal ≠ current symbol from input

$$(q, i, \alpha, a_i\beta) \vdash (b, i, \alpha, \beta)$$

* **Back**: head of working stack (α) is a terminal

$$(b, i, \alpha a, \beta) \vdash (b, i-1, \alpha, a\beta)$$

* **Another try**: head of the working stak is a nonterminal

$(b, i, \alpha A_j, \beta_j B) \vdash (q, i, \alpha A_{j+1}, \beta_{j+1} \beta)$ if $\exists A \to \beta_{j+1}$

$(b, i, \alpha, A\beta)$ otherwise with the exception

(e, i, α, β) if $i=1, A=S$

ERROR

* **Success**: $(q, n+1, \alpha, \varepsilon) \vdash (f, n+1, \alpha, \varepsilon)$

Algorithm

$w \in L(G)$ - ~~HOW~~

- Process α:
  - from left to right (reversed if stored as stack)
  - skip terminal symbols
  - Nonterminals - index of production

Example: $\alpha = S_1\ a\ S_2\ a\ S_3\ c\ b\ S_3\ c$

When the algorithm never stops ???

$S \to S\alpha$ - expand indefinitely (left recursive)

LL(1) Parser

$First_k \approx$ first k terminal symbols that can be generated from α

Definition:

$FIRST_k : (N \cup \Sigma)^* \to \mathcal{P}(\Sigma^k)$

$FIRST_k(\alpha) = \{u \mid u \in \Sigma^k, \alpha \overset{*}{\Rightarrow} ux, |u| = k$ sau $\alpha \overset{*}{\Rightarrow} u, |u| \leq k\}$

Construct **FIRST**:

- If L_1, L_2 are 2 languages over alphabet Σ, then

$L_1 \oplus L_2 = \{w \mid x \in L_1, y \in L_2, xy = w, |w| \leq 1$ or $xy = wz, |w| = 1\}$

($\oplus$: concatenation of lenght 1.)

- $FIRST(\alpha\beta) = FIRST(\alpha) \oplus FIRST(\beta)$

$FIRST(X_1 \dots X_n) = FIRST(X_1) \oplus \dots \oplus FIRST(X_n)$

Algorithm C6 Slide 21

Example:

$E \to TE'$	$First(E) = First(T) = \{(, id\}$
$E' \to +TE' \mid \varepsilon$	$First(E') = \{+, \varepsilon\}$
$T \to FT'$	$First(T) = First(F) = \{(, id\}$
$T' \to *FT' \mid \varepsilon$	$First(T') = \{*, \varepsilon\}$
$F \to (E) \mid id$	$First(F) = \{(, id\}$

(only terminals)

FOLLOW: $FOLLOW_k(A) =$ next k symbols generated after following A

$FOLLOW: (N \cup \Sigma)^* \to \mathcal{P}(\Sigma)$

$FOLLOW(\beta) = \{w \in \Sigma \mid S \overset{*}{\Rightarrow} \alpha\beta\gamma, w \in FIRST(\gamma)\}$

Algorithm C6 Slide 23

Example: Computing follow:

1. If A starting symbol, put ε in FOLLOW(A)
2. Productions of the form $B \to \alpha A \beta$
   $\Rightarrow$ FOLLOW(A) = FIRST(β)
3. Productions of the form $B \to \alpha A$ or $B \to \alpha A \beta$ where $\beta \overset{*}{\Rightarrow} \varepsilon$
   Add FOLLOW(A) = FOLLOW(B)

(a) $E \to TE'$
(b) $E' \to +TE' \mid \varepsilon$
(c) $T \to FT'$
(d) $T' \to *FT' \mid \varepsilon$
(e) $F \to (E) \mid id$

FOLLOW(E) = { $ } ∪ {) } (e → 3, a → 3.)
FOLLOW(E') = FOLLOW(E)
FOLLOW(T) = FIRST(E') = { +, ε } (b → 2)
FOLLOW(T') = FOLLOW(T)
FOLLOW(F) = FIRST(T') = { *, ε } (d → 2)

$E' \to +T \Rightarrow$ β3 $\Rightarrow$ Follow(T) = Follow(E')

$T' \to *F \Rightarrow$ β3 $\Rightarrow$ Follow(F) = Follow(T')

$\Rightarrow$ Follow(E) = { ε,) }
Follow(E') = { ε,) }
Follow(T) = { +, ε } ∪ Follow(E') = { +, ε,) }
Follow(T') = { +, ε,) }
Follow(F) = { *, ε } ∪ Follow(T') = { *, +, ε,) }

Definition:

A cfg is LL(k) if for any 2 leftmost derivations we have

1. $S \overset{*}{\Rightarrow}_{st} wA\alpha \Rightarrow_{st} w\beta\alpha \overset{*}{\Rightarrow}_{st} wx$
2. $S \overset{*}{\Rightarrow}_{st} wA\alpha \Rightarrow_{st} w\gamma\alpha \overset{*}{\Rightarrow}_{st} wy$

such that $FIRST_k(x) = FIRST_k(y) \Rightarrow \beta = \gamma$

Theorem:

A grammar is LL(1) if and only if for any nonterm. A with productions $A \to \alpha_1 | \alpha_2 | \dots \alpha_n$,

$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ and if $\alpha_1 \Rightarrow \varepsilon$,

$\text{FIRST}(\alpha_i) \cap \text{FOLLOW}(A) = \emptyset$, $\forall i,j = \overline{1,n}$, $i \neq j$

STEPS:

1) Construct FIRST, FOLLOW
2) Construct LL(1) parse table
3) Analyse sequence based on moves between configurations

Step 2: Construct LL(1) parse table

Add special char $ - marking for "empty stack"

Rules LL(1) table:

1. $M(A, a) = (\alpha, i)$ $\forall a \in \text{FIRST}(\alpha)$, $a \neq \varepsilon$
   $A \to \alpha$ prod in P with nr i

   $M(A, b) = (\alpha, i)$ if $\varepsilon \in \text{FIRST}(\alpha)$, $\forall b \in \text{FOLLOW}(A)$
   $A \to \alpha$ production in P with nr i

2. $M(a, a) = \text{pop}$, $\forall a \in \Sigma$
3. $M(\$, \$) = \text{acc}$
4. $M(x, a) = \text{err}$

Remark:

A grammar is LL(1) if the LL(1) parse tree table does not contain conflicts - there exists at most one value in each cell of the table $M(A, a)$

Step 3: Define configurations and moves

Configuration: $(\alpha, \beta, \overline{\pi})$

where α = input stack
β = working stack
$\overline{\pi}$ = output (result)

* Initial configuration: $(w\$, S\$, \varepsilon)$
* Final configuration: $(\$, \$, \overline{\pi})$

Moves:

1. Push - put in stack

$(\mu x, A\alpha\$, \overline{\pi}) \vdash (\mu x, \beta\alpha\$, \overline{\pi} i)$
if $M(A, \mu) = (\beta, i)$
(pop A, push symbols of β)

2. Pop - take off from stack (both stacks)

$(\mu x, a\alpha\$, \overline{\pi}) \vdash (x, \alpha\$, \overline{\pi})$ if $M(a, \mu) = pop$

3. Accept: $(\$, \$, \overline{\pi}) \vdash acc$

4. Error: otherwise

Algorithm: Input → LL(1) with no conflicts
→ G - grammar
→ sequence $w = a_1 a_2 \dots a_n$

$\alpha = w\$$; $\beta = S\$$; $\overline{\pi} = \varepsilon$
go = true
while go do:
  if M(head(β), head(α)) = (b, i) then
    pop(β), push(β, b); addTo($\overline{\pi}$, i)
  else
    if M(head(β), head(α)) = pop then
      pop(β); pop(α)
    else
      if M(head(β), head(α)) = acc then
        go = false; s = acc
      else
        go = false, s = err
print s

LR(0) Parser

L = left (seq. is read from left to right)
R = right (use rightmost derivations)

* Enhanced grammar:

$G = \{N, \Sigma, P, S\}$

$G' = \{N \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S'\}$

! S' does not appear in any rhp. (right hand production)

EXAMPLE: $G = (\{S', S, A\}, \{a, b, c\}, P, S)$

P: $S' \to S$

(1) $S \to aA$

(2) $A \to bA$

(3). $A \to c$

$w = abbc$

I. Canonical colection of states:

Initialization: $s_0 = \text{closure}(\{[S' \to .S]\}) = \{[S' \to .S] + \text{all productions of S with . in front of them.}$

Rules:

CLOSURE: if . in front of nonterminal X → add all productions of X with . in front of their rhs

GOTO: search for all items with . in front of them, move . after and compute closure of (item with dot . after)

$S_0 = \text{closure}(\{[S' \to .S]\}) = \{[S' \to .S], [S \to .aA]\}$

$S_1 = \text{goto}(S_0, S) = \text{closure}(\{[S' \to S.]\}) = \{[S' \to S.]\}$

$S_2 = \text{goto}(S_0, a) = \text{closure}(\{[S \to a.A]\}) = \{[S \to a.A],$
$[A \to .bA],$
$[A \to .c]\}$

$S_3 = \text{goto}(S_2, A) = \text{closure}(\{[S \to aA.]\}) = \{[S \to aA.]\}$

$S_4 = \text{goto}(S_2, b) = \text{closure}(\{[A \to b.A]\}) = \{[A \to b.A],$
$[A \to .bA], [A \to .c]\}$

$S_5 = \text{goto}(S_2, c) = \text{closure}(\{[A \to c.]\}) = \{[A \to c.]\}$

$S_6 = \text{goto}(S_4, A) = \text{closure}(\{[A \to bA.]\}) = [A \to bA.]$

$\text{goto}(S_4, b) = \text{closure}(\{[A \to b.A]\}) = S_4$

$\text{goto}(S_4, c) = \text{closure}(\{[A \to c.]\}) = S_5$

II Parsing table: reduce if . at the end.

* Look at S_0 it will be **shift** and for the nonTerm. with . in front look for goto(S_0, nonterm) and put the number of the state in the table

ex. $\text{goto}(S_0, S) = S_1 \Rightarrow$ put 1 at { line 0, column S

* If . at the ent put **reduce** + number of the production

$S_5 = \{[A \to c.]\} \Rightarrow$ put reduce 3 at line 5

(3) $A \to c$

* **Accept** if state $= \{[S' \to S.]\}$

	ACTION	GOTO				
		a	b	C	S	A
0	shift	2			1	
1	accept.					
2	shift		4	5		3
3	reduce 1					
4	shift		4	5		6
5	reduce 3					
6.	reduce 2					

III Parsing the sequence:

! workstack → top at the right
• input stack → top at the left
output → top at the left.

Always look at the top of the stacks

* Shift → move the terminal from input stack to work stack and put the number of shift
* Reduce n → * look for production number n
  * take everything from rhs of prod n till the end of the workstack
  * put on the workstack the lhs of prod n and the number associate with (lhs, prev nr) from the table
  * add n to output band

work stack	input stack	output band
$0	abbc $	ε
$0a2	bbc $	ε
$0a2b4	bc $	ε
$0a2b4b4	c $	ε
$0a2b4b4c5	$	ε
$0a2b4b4A6	$	3
$0a2b4A6	$	23
$0a2A3	$	223
$0S1	$	1223
accept	$	1223

SLR Parser

Example: $G = (\{S', E, T\}, \{+, id, const, (,)\}, P, S')$

P: $S' \to E$

(1) $E \to T$

(2) $E \to E + T$

(3) $T \to (E)$

(4) $T \to id$

(5) $T \to const$

$w = id + const$

I Canonical collection – SAME AS LR(0)

Rules: *CLOSURE: if . in front of nonterminal X ⇒ add all productions of X with . in front of their rhs

* GOTO: search for all items with . in front of them, compute closure (item with . AFTER)

$s_0 = closure(\{[S' \to .E]\}) = \{[S' \to .E], [E \to .T], [E \to .E+T], [T \to .(E)], [T \to .id], [T \to .const]\}$

$s_1 = goto(s_0, E) = closure(\{[S' \to E.], [E \to E.+T]\}) = \{[S' \to E.], [E \to E.+T]\}$

$s_2 = goto(s_0, T) = closure(\{[E \to T.]\}) = \{[E \to T.]\}$

$s_3 = goto(s_0, ( ) = closure(\{[T \to (.E)]\}) = \{[T \to (.E)], [E \to .T], [E \to .E+T], [T \to .(E)], [T \to .id], [T \to .const]\}$

$s_4 = goto(s_0, id) = closure(\{[T \to id.]\}) = \{[T \to id.]\}$

$s_5 = goto(s_0, const) = closure(\{[T \to const.]\}) = \{[T \to const.]\}$

$s_6 = goto(s_1, +) = closure(\{[E \to E+.T]\}) = \{[E \to E+.T], [T \to .(E)], [T \to .id], [T \to .const]\}$

$s_7 = goto(s_3, E) = closure(\{[T \to (E.)], [E \to E.+T]\}) = \{[T \to (E.)], [E \to E.+T]\}$

goto(s_3, T) = closure({[E → T.]}) = {[E → T.]} = s_2

goto(s_3, () = closure({[T → (.E)]}) = s_3

goto(s_3, id) = closure({[T → id.]}) = s_4

goto(s_3, const) = closure({[T → const.]}) = s_5

s_8 = goto(s_6, T) = closure({[E → E+T.]}) = {[E → E+T.]}

goto(s_6, () = closure({[T → (.E)]}) = s_3

goto(s_6, id) = closure({[T → id.]}) = s_4

goto(s_6, const) = closure({[T → const.]}) = s_5

s_9 = goto(s_7,)) = closure({[T → (E).]}) = {[T → (E).]}

goto(s_7, +) = closure({[E → E+.T]}) = s_6

Compute FOLLOW rules:

1. If A starting symbol, put ε in FOLLOW(A)
2. Productions of the form $B \to \alpha A \beta$

   ⇒ FOLLOW(A) = FIRST(β)
3. Productions of the form $B \to \alpha A$ or $B \to \alpha A \beta$ with $\beta \overset{*}{\Rightarrow} \varepsilon$

   ⇒ Add to FOLLOW(A) ← FOLLOW(B)

FOLLOW(E) = {ε, +,)}

FOLLOW(T) = {ε, +,)}

II. Parsing table

ACTION - column for every terminal + $

* Shift : . in front of terminal => shift + nr of state from goto(current state, terminal)
* Reduce : . at the end => reduce + nr of production to all elems from FOLLOW of nonterminal
* Accept : only if . after starting symbol

GOTO : - column for every nonterminals

. in front of nonterminal X => put number of state that has goto(current state, X)

	ACTION						GOTO	
	+	(	)	id	const	$	E	T
0		Shift 3		Shift 4	Sh 5		1	2
1	Shift 6					accept		
2	Reduce 1		Reduce 1			Reduce 1		
3		Shift 3		Shift 4	Shift 5		7	2
4	Reduce 4		Reduce 4			Reduce 4		
5	Reduce 5		Reduce 5			Reduce 5		
6		Shift 3		Shift 4	Shift 5			8
7	Shift 6		Shift 9					
8	Reduce 2		Reduce 2			Reduce 2		
9	Reduce 3		Reduce 3			Reduce 3		

III Parsing the sequence: SAME AS LR(0)

* **Shift** → move terminal from input stack to work stack and put number of shift
* **Reduce n** * look for production nr n
  * take everything from rhs of prod n till the end of workstack
  * put on the work stack the lhs of prod n and the number associated with (lhs, previous nr) from the table
  * add n to output band

work stack	input stack	output band
$0	id + const $	ε
$0id4	+ const $	ε
$0T2	+ const $	4
$0E1	+ const $	14
$0E1+6	const $	14
$0E1+6const5	$	14
$0E1+6T8	$	514
$0E1	$	2514
accept		

$E \underset{2}{\Rightarrow} E+T \underset{5}{\Rightarrow} E+\text{const} \underset{1}{\Rightarrow} T+\text{const} \underset{4}{\Rightarrow}$

$\Rightarrow id+\text{const}$

Push-down automata (PDA)

Definition: A PDA is a 7-tuple $M=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- Q - finite set of states
- Σ - alphabet (input symbols)
- Γ - stack alphabet
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ - transition function
- $q_0 \in Q$ - initial state
- $Z_0 \in \Gamma$ - initial stack symbol
- $F \subseteq Q$ - set of final states

* A transition is determined by:
  - current state
  - current input symbol
  - head of stack
* Reading head → input band: → read one symbol
  ↓ or
  no action
* Stack: → zero symbols ⇒ pop
  → one symbol ⇒ push
  → several symbols ⇒ repeat push

Configuration: (q, x, α) where
- PDA is in state q
- input band contains x
- head of stack is α

Initial configuration: (q_0, x, Z_0)

Moves:

1. If $\delta(q, a, Z) \ni (p, \gamma)$

$\Rightarrow (q, aw, Z\alpha) \vdash (p, w, \gamma\alpha)$

2. If $\delta(q, \varepsilon, Z) \ni (p, \gamma)$ | ε-move |

$\Rightarrow (q, aw, Z\alpha) \vdash (p, aw, \gamma\alpha)$

Language accepted by PDA:

- Empty stack-principle:

$$L_\varepsilon(M) = \{w \mid w \in \Sigma^*, (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

- Final state principle:

$$L_f(M) = \{w \mid w \in \Sigma^*, (q_0, w, Z_0) \vdash^* (q_f, \varepsilon, \gamma), q_f \in F\}$$

Example:

Construct the PDA for $L = \{0^n 1^n \mid n \geq 1\}$

HOW?

For states:

- initial state: q_0 - beginning and process only 0
- when first symbol 1 is found - move to new state -> q_1
- final state: q_2

For stack:

- initial symbol: Z_0
- X - to count symbols:
  - * when reading 0 - push X in stack
  - * when reading 1 - pop X from stack

$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{Z_0, X\}, \delta, q_0, Z_0, \{q_2\})$

$\delta(q_0, 0, Z_0) = (q_0, XZ_0)$

$\delta(q_0, 0, X) = (q_0, XX)$

$\delta(q_0, 1, X) = (q_1, \varepsilon)$

$\delta(q_1, 1, X) = (q_1, \varepsilon)$

$\delta(q_1, \varepsilon, Z_0) = (q_2, Z_0)$ or $\delta(q_1, \varepsilon, Z_0) = (q_1, \varepsilon)$

$w = 0011$ – is it accepted by the language?

$(q_0, 0011, Z_0) \vdash (q_0, 011, XZ_0) \vdash (q_0, 11, XXZ_0)$

$\vdash (q_1, 11, XXZ_0) \vdash (q_1, 1, XZ_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$

final state

$\Rightarrow 0011 \in L_f(M)$.

$\Rightarrow \vdash (q_1, \varepsilon, \varepsilon)$

empty stack

Properties:

Theorem 1: For any PDA M, there exists a PDA M' such that $L_\varepsilon(M) = L_f(M)$

Theorem 2: For any PDA M, there exists a context free grammar such that $L_\varepsilon(M) = L(G)$

Theorem 3: For any context free grammar there exists a PDA M such that $L(G) = L_\varepsilon(M)$.

Semantic analysis

* Parsing ⇒ syntax tree (ST)
* Simplification ⇒ abstract ST (AST)
* Adnotated abstract ST (AAST)
  ↳ attach semantic info in tree nodes

Example

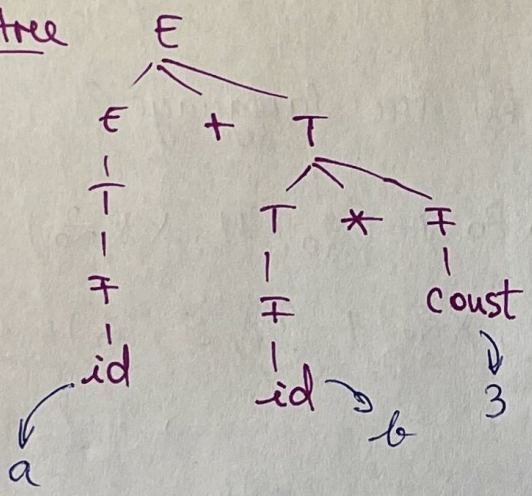
$E \to E+T \mid T$

$T \to T*F \mid F$

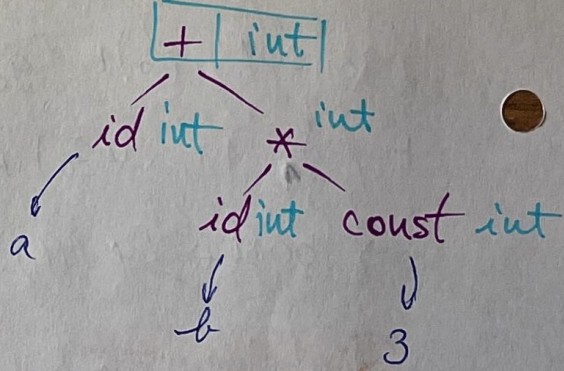
$F \to (E) \mid id \mid const$

parse : a+b*3

Syntax tree



Abstract syntax tree :



Adnotated abstr. ST

ex : type

* Attach meanings to syntactical constructions of a program
* Constructions : → identifiers : values / how to be evaluated
  → statements : how to be executed
  → declaration : det. space to be allocated and location to be stored
* Examples : type checking, verify properties.
* How : → attribute grammars
  → manual methods

Attribute grammar

* For **nonterminal** associate **attributes** (set of attributes)

$\forall X \in N \cup \Sigma : A(X)$

* For **productions** associate **evaluation rules** (set of rules)

$\forall p \in P : R(p)$

Definition: $AG = (G, A, R)$ is called attribute grammar where

* $G = (N, \Sigma, P, S)$ is a context free grammar
* $A = \{A(x) \mid x \in N \cup \Sigma\}$ – finite set of attributes
* $R = \{R(p) \mid p \in P\}$ – finite set of rules to compute/evaluate attributes

* Example → Compute the value of a nr from base 2, to base 10.

$G = (\{N, B\}, \{0, 1\}, P, N)$

P: $N \to NB$

$N \to B$

$B \to 0$

$B \to 1$

2. Write the rules:

$N_1.V = 2 * N_2.V + B.V$

$N.V = B.V$

$B.V = 0$

$B.V = 1$

1. Choose as attribute value of $nr = V$

* **Synthetized attributes**: $A(lhp)$ depends on rhp
* **Inherited attributes**: $A(rhp)$ depends on lhp.

* Special cases of AG:
  - **L-attribute grammars**: for any node the depending attr. are on the left; incorporated in top-down parser LL(1)
  - **S-attribute grammars**: synthetized attributes incorporated in bottom-up. parser (LR).

Steps:

1. What? - decide what you want to compute (type, value)
2. Decide attributes:
   * how many
   * which attribute is defined for which symbol
3. Attach evaluation rules: for each production - which rule/ rules.

More examples course 11 slide 23.

Generate intermediary code

* Forms of intermediary code:
  - Java bytecode: source lang: Java, machine lang: JVM
  - MSIL (Microsoft intermediate language) - source lang: C#, VB, machine lang: Windows
  - GNU RTL (Register transfer language) - source lang: C, C++, Pascal, Fortran, machine lang: dif. platforms

* Representations:
  - Adnotated syntax tree: is generated in semantic analysis
  - Polish postfix form: operators appear in the order of execution

    ex: expr = a+b*c ppf = abc*+

    expr a*b+c ppf = ab*c+
  - 3 address code

3 address code

Def: Sequence of simple format statements, close to object code, with the following general form:

<result> = <arg 1> <op> <arg 2>

Represented as:

* Quadruples: <op> <arg 1> <arg 2> <result>
* Triples: <op> <arg 1> <arg 2> (consider that the triple is storing the result)
* Indirected Triples:

Example: b*b - 4*a*c.

Quadruples:

op	arg 1	arg 2	rez
*	b	b	t_1
*	4	a	t_2
*	t_2	c	t_3
-	t_1	t_3	t_4

Triples

nr	op	arg 1	arg 2
(1)	*	b	b
(2)	*	4	a
(3)	*	(2)	c
(4)	-	(1)	(3)

Optimize intermediary code

- Local optimizations:
  * perform computation at compile time - constant values
  * eliminate redundant computations
  * eliminate inaccessible code
- Loop optimizations:
  * factorization of loop invariants
  * reduce power of operations

See examples course 12 slide 11

Generate object code

Def: Translate intermediary code statements into statements of object code (machine language)

2 aspects:
- **Register allocation**: way in which variable are stored and manipulated
- **Instruction selection**: way and order in which the intermediary code stmts are mapped to machine instructions

Computer with accumulator

* A **stack machine** consists of a **stack** (for storing and manipulating values) and **2 type of statements**:
  - move and copy values in and from head of stack mem.
  - operations on stack head: operands are popped from stack, execute operation and then put the result in stack

* Accumulator: to execute operation
* Stack to store subexpressions and results.

See example course 12 slide 17

Computer with registers – registers + memory

* Instructions:
  - LOAD v, R: load value v in register R
  - STORE R, v: put value v from register R in memory
  - ADD R_1, R_2: add to the value from R_1 the value from R_2 and store the result in R_1

Remarks:

* A register can be available or occupied =>

VAR(R) = set of variables whose values are stored in register R

* For every variable, the place (register, stack or memory) in which the current value of it exists =>

MEM(x) = set of locations in which the value of variable x exists (will be stored in symbol table)

More info course 12 slide 20.