Lecture 1:

source code → compiler → object code

Structure of a compiler

source code / program → scanning (lexical analysis)
↳ detect and classify and codify lexical items (= tokens)

→ tokens and ST (= symbol table) → parsing (syntactical analysis

→ syntax tree → semantic analysis →

analysis

synthesis

→ annotated syntax tree → intermediary code generation

→ intermediary code → intermediary code optimization

→ optimized inter. code → object code generation ⇒ object code program

+ error handling

+ symbol table management

1. Scanning = treats the source prg. as a sequence of chars., detect lexical tokens, classify and codify them

- tokens can be: words, (spaces), data ...
  (not really a token, they delimit tokens)
  ⇒ delimiters: • space, newline, tab (= white/ invisible)
  • . , ; ( ) (= visible)

- look ahead for some symbols (=, +, -, >, <) at the next symbol to see if it can form a token

- identifiers: x, y (unknowns)

- reserved word vs. keyword

- ! if a token can't be classified ⇒ lexical error (ex: 2a)

- rule for identifiers:

S → S
'_' or letter '_' or letter or digit

input: source prg. ⇒ output: PIF + ST

Classify: classes of tokens ⇒ ids, constants, separators, operators, reserved words

reserved word → if
separator → (
identifier → x
operator → = =
identifier → y
)
{
x
=
y
+
constant → 2
;
reserved words

Codify

- codification table or code for identifiers and constants
- (identifier, constant) => ST
- PIF = Program Internal Form = array of pairs
- pairs (token, position in ST)

ST	
1	x
2	y

Scanning alg

```
while (not(eof)) do
    detect(token);
    if token is reserved word or operator or separator
        then genPIF(token, 0)
    else
        if token is identifier or constant
            then index = pos(token, ST);
                 genPIF(token, index)
        else message "lexical error"
        endif
    endif
endwhile
```

* genPIF = adds a pair (token, position) to PIF
  pos(token, ST) = searches token in ST; if found then return position
  if not found insert in ST and return position

Lecture # 2:

- most important operations: search, insert
- **symbol table** = contains all information collected during compiling regarding the symbolic names from the source program
  ↑ identifiers, constants etc.
  - variants: unique symbol table - contains all symbolic names
    distinct symbol table - IT (identifiers table) + CT (const table)

- ST organization: 1) unsorted table
  ↳ in order of detection in source code $O(n)$
  2) sorted table - alphabetic (numeric) $O(\lg n)$
  3) binary search tree (balanced) $O(\lg n)$
  4) **hash table** $O(1)$
  ↳ K = set of keys (symbolic names)
  A = set of positions ($|A| = m$; m - prime no.)
  $h : K \to A$, $h(K) = (val(K) \bmod m) + 1$
  conflicts: $K_1 \neq K_2$, $h(K_1) = h(K_2)$
  $h(K) = val(a)$
  ↳ sum of ASCii codes of chars

- visibility domain (scope): each scope - separate ST
  structure ⇒ inclusion tree

Formal languages (basic notions)

FA = Finite Automaton / Automata (sa: automat finit)

PDA = Push-Down Automaton

ex. of langs.: natural, programming, formal (= set) — mathematical view of a program

ex: a boy (subject) / has a dog (predicate)

sentence) S → AV
A → BN
B → a | the
N → boy | dog
V → has A
A → α (= rule)
S, P, V ... (= nonterminal symbols)
a, boy, dog, has (= terminal symbols)

⇒ a dog has a dog
the dog has a boy

Remark: $S \Rightarrow AV \Rightarrow BNV \Rightarrow aNV \Rightarrow \dots$ (until we reach terminal symbols)

1) Sentence = word, sequence (contains only terminal symbols)
   ↳ denoted: w
2) $S \Rightarrow PV \Rightarrow aNV \dots$ = sentential form
   ↳ in general: $w = a_1 a_2 \dots a_n$
3) the rule guarantees syntactical correctness, but not the semantical correctness

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

- N = set of nonterminal symbols and $|N| < \infty$
- Σ = set of terminal symbols (alphabet) and $|\Sigma| < \infty$
- P = finite set of productions (rules), with the property:
  $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- $S \in N$ = start symbol / axiom

remarks: 1) $(\alpha, \beta) \in P$ is a production denoted $\alpha \to \beta$

2) $N \cap \Sigma = \emptyset$

A^* = transitive and reflexive closure $= \{a, aa, aaa, \dots\} \cup \{a^0\}$

$A = \{a\}$

$A^+ = \{a, aa, \dots\}$

$x^0 = \varepsilon$ (empty word)

Binary relations defined on $(N \cup \Sigma)^*$

- direct derivation: $\alpha \Rightarrow \beta$, $\alpha, \beta \in (N \cup \Sigma)^*$, if $\alpha = x_1 \textcircled{x} y_1$, $\beta = x_1 \textcircled{\gamma} y_1$ and $\textcircled{x} \to \textcircled{\gamma} \in P$
- k-derivation: $\alpha \overset{k}{\Rightarrow} \beta$, $\alpha, \beta \in (N \cup \Sigma)^*$ seq. of k derivations $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{k-1} \Rightarrow \beta$; $\alpha, \alpha_1, \alpha_2, \dots, \alpha_{k-1}, \beta \in (N \cup \Sigma)^*$
- + derivation: $\alpha \overset{+}{\Rightarrow} \beta$, if $\exists k > 0$ s.t. $\alpha \overset{k}{\Rightarrow} \beta$ (there is at least one direct derivation)
- * derivation: $\alpha \overset{*}{\Rightarrow} \beta$, if $\exists k \geq 0$ s.t. $\alpha \overset{k}{\Rightarrow} \beta \Leftrightarrow$ $\alpha \overset{*}{\Rightarrow} \beta \Leftrightarrow \alpha \overset{+}{\Rightarrow} \beta$ or $\alpha \overset{0}{\Rightarrow} \beta$ ($\alpha = \beta$)

$S \Rightarrow AV \Rightarrow BNV$

derivation $\Rightarrow A \to BNV$

$S \to AV$

$S \overset{*}{\Rightarrow}$ a boy has a dog

$S \overset{*}{\Rightarrow} w$

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

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

concatenation

$L_1 = \{a, b, aa\}$
$L_2 = \{c, d, cd\}$
$L_1 L_2 = \{ac, ad, acd, bc, bd, \ldots\}$

remarks: 1) $S \overset{*}{\Rightarrow} \alpha$, $\alpha \in (N \cup \Sigma)^*$ (= sentential form)
$S \overset{*}{\Rightarrow} w$, $w \in \Sigma^*$ (= word / sequence)

2) operations: $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$

concatenation: $L = L_1 L_2 = \{ w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2 \}$

3) $|w| = 0$ (empty word - denoted ε)

Def.: two grammar G_1 and G_2 are equivalent if they generate the same language $\Leftrightarrow$ $L(G_1) = L(G_2)$

Chomsky hierarchy (based on form $\alpha \to \beta \in P$)

- type 0: no restriction
- type 1: context dependent grammar ($x_1 A y_1 \to x_1 \gamma y_1$)
- type 2: context free grammar ($A \to \alpha \in P$, $A \in N$ and $\alpha \in (N \cup \Sigma)^*$) (fără nimic "|")
- type 3: regular grammar ($A \to aB \mid a \in P$)

remark: type 3 $\subseteq$ type 2 $\subseteq$ type 1 $\subseteq$ type 0

Notations:

- A, B, C - nonterminal symbols
- $S \in N$ - start symbol (N - set of nonterminal symbols)
- $a, b, c \in \Sigma$ - terminal symbols
- $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ - sentential forms
- ε - empty word
- $x, y, z \in \Sigma^*$ - words
- $X, Y, U \in (N \cup \Sigma)$ - grammar symbols (nonterminal or terminal)

Lecture 3: Regular languages

Def.: A finite automata (FA) is a 5-tuple:

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

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

$q \xrightarrow{\varepsilon} p$ bc. $\varepsilon \in \Sigma^*$

Remarks: 1. $Q \cap \Sigma = \emptyset$

2. $\delta: Q \times \Sigma \to \mathcal{P}(Q)$, $\varepsilon \in \Sigma^*$ - relation $\delta(q, \varepsilon) = p$ NOT allowed
3. if $|\delta(q, a)| \leq 1 \Rightarrow$ deterministic finite automata (DFA)
4. if $|\delta(q, a)| > 1$ (more than a state obtained as result) $\Rightarrow$ NFA
   non-det FA

Property: For any NFA M there exists a DFA M' equiv. to M.

Configuration $C = (q, x)$; q state; x unread seq. from input
$x \in \Sigma^*$

- Initial configuration: (q_0, w), w - whole sequence
- Final configuration: (q_f, ε), $q_f \in F$, ε - empty seq.
  ↳ corresponds to accept

Relations between configs.:

- $\vdash$ move/transition (simple, one step)
  $(q, ax) \vdash (p, x)$, $p \in \delta(q, a)$
- $\vdash^k$ k move = a seq. of k simple transitions: $C_0 \vdash C_1 \vdash \dots \vdash C_k$
- $\vdash^+$ + move: $C \vdash^+ C'$: $\exists k > 0$ s.t. $C \vdash^k C'$
- $\vdash^*$ * move: $C \vdash^* C'$: $\exists k \geq 0$ s.t. $C \vdash^k C'$

$S \to \varepsilon$ reg
$A \to \varepsilon$ not reg
$S \to aA \mid \varepsilon$ not reg
$A \to aS$ reg

Regular grammars:

- regular grammar if - is right linear grammar
  - $A \to \varepsilon \notin P$, with the exception that $S \to \varepsilon \in P$, in which case S does not appear in the rhs of any other production
- right linear language

- right linear grammar

Regular sets = let Σ be a finite alphabet; we define regular sets over Σ recursively in the foll. way:

1. $\emptyset$ is a reg. set over Σ (empty set)
2. $\{\varepsilon\}$ is a reg. set over Σ
3. $\{a\}$ is a reg. set over Σ, $\forall a \in \Sigma$
4. if P, Q are reg. sets over $\Sigma \Rightarrow P \cup Q, PQ, P^*$ are reg. sets over Σ
5. nothing else is a reg. set over Σ

Regular expressions = let Σ be a finite alphabet; we define reg. expressions over Σ recursively in the foll. way:

1. $\emptyset$ is a reg. expression denoting the reg. set $\emptyset$ (empty set)

— u —

2. ε — u — $\{\varepsilon\}$
3. a — u — $\{a\}$, $\forall a \in \Sigma$
4. if p, q are — u — $P, Q \Rightarrow$
   - $p+q$ — u — $P \cup Q$
   - pq — u — PQ
   - p^* — u — P^*
5. Nothing else is a reg. expression

- $a^* = a + a^*$
- $(a^*)^* = a^*$
- $a + a = a$
- $a a^* b + b = (a a^* + \varepsilon) b = a^* b$

Theorem: a) A language is a regular set i.e. a. if is a right linear lang

-) A language is a regular set i.e. a. if it is accepted by a FA.

$$\begin{cases} q_1 = q_3 0 + \varepsilon \\ q_2 = q_1 0 + q_1 1 + q_2 0 + q_3 0 \\ q_3 = q_2 1 \end{cases}$$

,) $\emptyset, \{\varepsilon\}, \{a\}, \forall a \in \Sigma$ are accepted by FA

FA

.) if L_1 and L_2 are accepted by FA $\Rightarrow L_1 \cup L_2, L_1 L_2, L_1^*$ are acc by

Course 5 : Context free grammars (⇒ lhs = a single nonterminal)

Syntax tree = a syntax tree corr. to a cfg. $G = (N, \Sigma, P, S)$ is a tree obtained in the foll. 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 descendants in order from left to right are $X_1, X_2, \dots, X_n$ only if $A \to X_1 X_2 \dots X_n \in P$

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

ex.: $S \xrightarrow{1} aSbS \mid \xrightarrow{2} c$

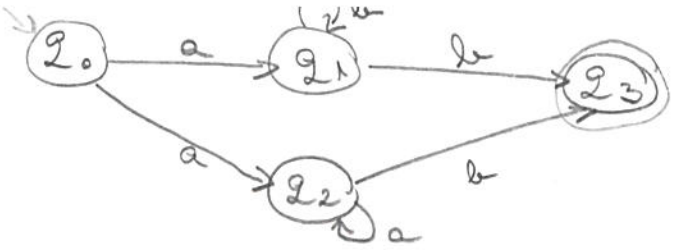
$w = aacbcbc$

- leftmost derivations: $S \Rightarrow aSbS \underset{1}{\Rightarrow} aaSbSbS \underset{2}{\Rightarrow} aacbSbS \underset{2}{\Rightarrow} aacbcbS \underset{2}{\Rightarrow} aacbcbc = w$
- rightmost derivation: $S \Rightarrow aSbS \underset{2}{\Rightarrow} aSbc \underset{1}{\Rightarrow} aaSbSbc \underset{2}{\Rightarrow} aaSbcbc \underset{2}{\Rightarrow} aacbcbc = w$

Def.: A cfg $G = (N, \Sigma, P, S)$ is ambiguous if for a $w \in L(G)$ there exist 2 different syntax trees with frontier w.

Def.: A nonterminal A is unproductive in a cfg if doesn't generate any word: $\{w \mid A \Rightarrow^* w, w \in \Sigma^*\} = \emptyset$.

ex.:



a) $w = abbba$

b) $w' = aba$

a) $w \in L(M)$?

$(q_0, w) \vdash^* (q_f, \varepsilon)$

$(q_0, abbba) \vdash (q_1, bbba) \vdash (q_1, bba) \vdash (q_1, ba) \vdash (q_3, a) \vdash (q_3, \varepsilon)$ $\}$ $q_3 \in F$

$\Rightarrow abbba \in L(M)$

b) $(q_0, aba) \vdash (q_2, ba) \vdash (q_3, a) \vdash (q_1, ba) \vdash (q_1, a) \vdash$

$\vdash (q_3, a)$

$A \to aB \mid b$ right linear

Theorem 1: For any regular grammar $G = (N, \Sigma, P, S)$ there exists a FA $M = (Q, \Sigma, \delta, q_0, F)$ s.t. $L(G) = L(M)$.

RG	FA
$G = (N, \Sigma, P, S)$	$M = (Q, \Sigma, \delta, q_0, F)$
$A \to aB$	$Q = N \cup \{k\}$, $k \notin N$
$A \to b$	$q_0 = S$
	δ : $A \to aB$ then $\delta(A, a) = B$ $\}$!
	$A \to a$ then $\delta(A, a) = k$ $\}$!
	$F = \{k\} \cup \{S \mid S \to \varepsilon \in P\}$

Theorem 2: For any FA $M = (Q, \Sigma, \delta, q_0, F)$ there exists a **right linear grammar** $G = (N, \Sigma, P, S)$ s.t. $L(G) = L(M)$.

$p \mid q$

$p^+ = pp^* = p^*p$

$pq + r$ = p concatenated with q or r

$p(q + r)$

$pr^* = \{p, pr, prr \dots\}$

$(pr)^* = \{\varepsilon, pr, prpr, \dots\}$

$aa^*b + b = a^+b + b = (a^+ + \varepsilon)b$

a^0, a^+

$a^+ = \{a, aa, \dots\}$

$a^* = \{\varepsilon, a, aa, \dots\}$

Lecture 6: Parsing

- if the source program is syntactically correct, then construct syntax tree else "syntax error"
- result - parse tree - representation
  - arbitrary tree - child sybling repres.
  - seq. of derivations $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = w$
  - string of production - index associated to prod - which prod is used at each derivation step: 1, 4, 3, ...

$S \overset{*}{\Rightarrow} w$

$S \overset{(1)}{\Rightarrow} \alpha_1 \overset{(5)}{\Rightarrow} \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{n-1} \Rightarrow w$

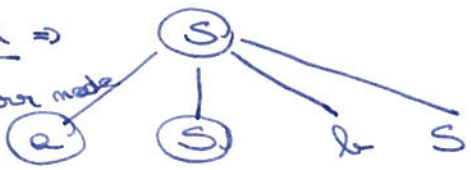
$S \to \alpha_1$; $1, 5, \dots$ + leftmost / rightmost

Descendent recursive parser

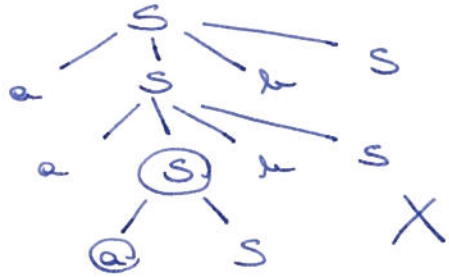
ex.: $S \to aSbS \mid aS \mid c$

$w = aaacbc$

expand ⇒

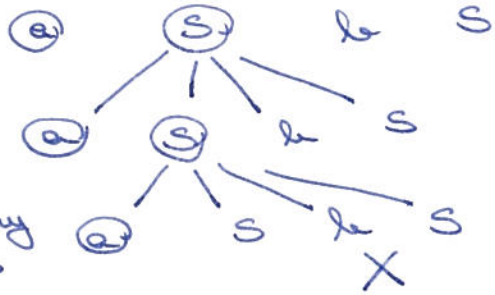


another try:



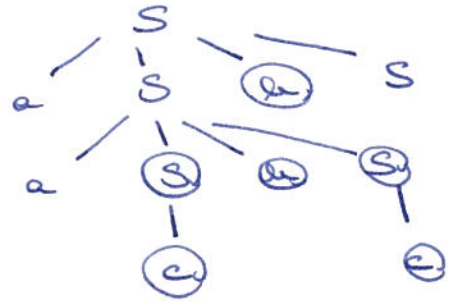
if the curr. node matches with the first of w, we advance

if not, we expand

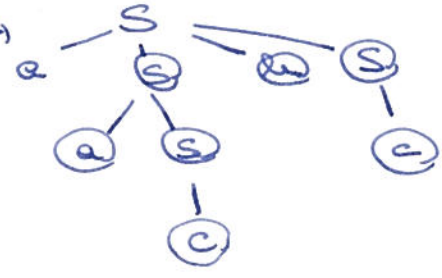


go back another try

another try:



back repeated + another try:



leftmost derivation

$S \Rightarrow aSbS \Rightarrow aaSbS \Rightarrow aacbS \Rightarrow$

$\Rightarrow aacbc \quad (1, 2, 3, 3)$

initial config: $(q, 1, \varepsilon, S)$

final config: $(f, n+1, \alpha, \varepsilon)$

Formal model : - configuration : (s, i, α, β)

- s = state of the parsing
  - q = normal state
  - b = back state
  - f = final state
  - e = error state
- i = pos. of curr. symbol in input seq. ($w = a_1 a_2 \dots a_n$, $i \in \{1, \dots, n+1\}$)
- α = working stack, store the way the parse is built
- β = input stack, part of the tree to be built

LL(1) Parser

= left left
↳ left most derivation
↳ read from left to right

FIRST$_k$ ~ first k terminal symbols that can be generated from α

a a oo
a a o1
a b oo
a b o1
b a oo
b a o1

a o
a 1
o
1

Lecture 4 :

- $\oplus$ = concatenation of length 1

ex: $L_1 = \{a, \varepsilon\}$

$L_2 = \{o, 1\}$ $\quad L_1 \oplus L_2 = \{$

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

ex : $A \to BC$
$B \to DA \mid CC$
$D \to a$
$C \to \varepsilon \mid \varepsilon$

same as F_i so we stop ↓

	F_0	F_1	F_2	F_3	F_4	F_5
A	$\varnothing$	$\varnothing$	ε, ε	ε, ε	$a, \varepsilon, \varepsilon$	
B	$\varnothing$	ε, ε	ε, ε	$a, \varepsilon, \varepsilon$	$a, \varepsilon, \varepsilon$	
C	ε, ε	ε, ε	ε, ε	ε, ε	ε, ε	
D	a	a	a	a	a	

$F_1(A) = F_0(A) \cup \{$

$\{x \mid A \to BC, x \in F_0(B) \oplus F_0(C)\}$ x

$\varnothing$ $\quad \varnothing \oplus$ anything $= \varnothing$

$F_1(B) = F_0(B) \cup \{ x \mid B \to DA, x \in F_0(D) \oplus F_0(A)\}$ ($F_0(A) = \varnothing$)

$B \to CC, x \in F_0(C) \oplus F_0(C)\}$ $\quad \{\varepsilon, \varepsilon\} \oplus \{\varepsilon, \varepsilon\} = \{\varepsilon, \varepsilon\}$

$F_3(B) = F_2(B) \cup \{x \mid B \to DA, x \in F_2(D) \oplus F_2(A)\}$

=> first terminal generated by S } $\Rightarrow$ Fost$(S) = a$
$S \to a$ }

special case ε always add it

FOLLOW:

- Follow(A): $S \overset{*}{\underset{l}{\Rightarrow}} xB\textcircled{y} \Rightarrow xa\textcircled{A}y$ (leftmost derivation; $B \to aA$) what if $B \to uA$
- Follow: $(N \cup \Sigma)^* \to P(\Sigma)$
- $Follow(\beta) = \{ w \in \Sigma \mid S \overset{*}{\Rightarrow} \alpha\beta\gamma, w \in FIRST(\gamma) \}$

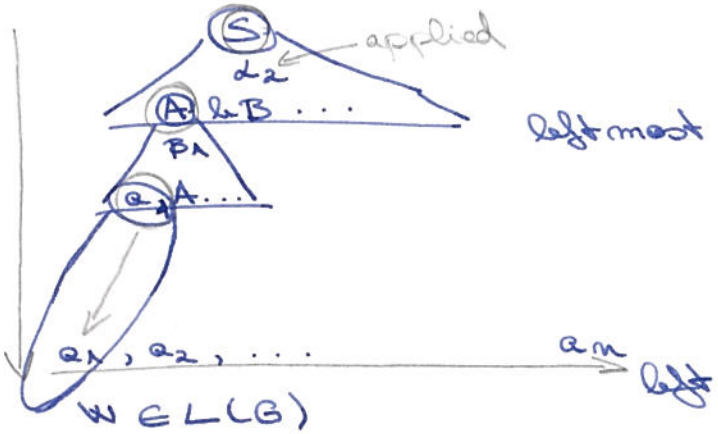
$S \overset{*}{\Rightarrow} S$ true $S \overset{0}{\to} S\varepsilon$

= next symbol generated after / following A

LL(K): L = left (seq. is read from left to right)

L = left (use left most deriv.)

prediction of length K



$S \to \alpha_1 | \alpha_2 | \alpha_3$

leftmost derivations

$A \to \beta_1 | \beta_2 | \beta_3$

+ def of LL(K) and theorem

LL(1) Parser: 1) construct First, Follow

2) construct LL(1) parse table

3) analyse seq. based on moves between configurations

	$\in \Sigma$			$\$ (\notin N \cup \Sigma)$
	a	b	...	
$\in N$ S				
$\in N$ A	α_1, i^*	α_1, i^*		
$\in N$ ...				
$\in \Sigma$ a	pop			
$\in \Sigma$ b		pop		
$\in \Sigma$ ...				
$\$$				acc

$i \quad i+1 \quad i+2$

$A \to \alpha_1 | \alpha_2 | \alpha_3$

*$A \to \alpha_1$, $First(\alpha_1) = \{a, b\}$

2. $M(a,a) = pop$, $\forall a \in \Sigma$
3. $M(\$, \$) = acc$
4. $M(x, a) = err$
1. ...

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

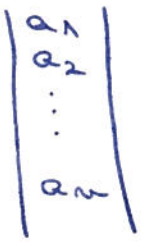
LL(1) configs: (α, β, π) (α: input stack, β: working stack, π: result (next stack))

initial config: $(w\$, S\$, \varepsilon)$

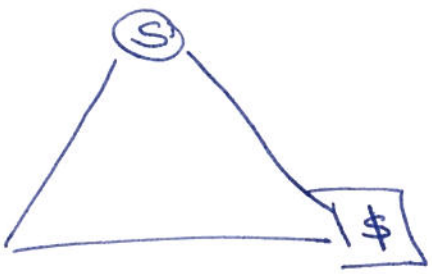
final config: $(\$, \$, \pi)$

working stack

$w = a_1 \dots a_n$



input stack



input is empty

- moves: push, pop, accept, error (+message)

eg. is read from l. to r. ← use rightmost derivations

Lecture 8: LR(k) parsing

- handle = symbols from the head of the working stack that form (in order) a rhp
- shift reduce parser [shift symbols to form a handle; when a rhp is formed, reduce to the corresponding lhp

Enhanced grammar: $G = (N, \Sigma, P, S) \Rightarrow G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S'), S' \notin N$

↳ S' doesn't appear in any rhp

Def.: 1)

- $[A \to \alpha\beta \bullet, u]$ - special case: prefix is all rhp - apply reduce
- otherwise, $[A \to \alpha \bullet \beta, u]$ - apply shift

Consequence 1): state is important - should be stored by parsing method

2): the action takes the parsing process to another state (goto)

LR(k) principle: - current state
- current symbol
- prediction: uniquely determines action to be applied, move to a new state

⇒ LR(k) table has 2 parts: action part + goto part

start
$S \to .aSa, a$ shift
$S \to a.Sa, a$ shift
$S \to aS.a, a$ shift
$S \to aSa., a$ reduce

$[A \to \alpha . \beta, u]$ $A \to \alpha\beta$

$S \overset{*}{\underset{r}{\Rightarrow}} \delta A w \underset{r}{\Rightarrow} \delta \alpha \beta w$

$u \in FIRST(w)$

$[A \to \alpha . B\beta, u]$ $B \to \delta$

$S \overset{*}{\underset{r}{\Rightarrow}} \delta A w \underset{r}{\Rightarrow} \delta \alpha B \beta w \underset{r}{\Rightarrow} \delta \alpha B w' \underset{r}{\Rightarrow} \delta \alpha \delta w'$

⇒ $[A \to \alpha . B\beta, u]$, $[B \to . \delta, u]$

! state = all LR items that core to the same live prefix

goto = how can I go from one state to another state with a symbol

goto $(\Sigma, X) = \Sigma'$
(Σ: state $\in N \cup \Sigma$; Σ': state)

closure = algorithm that puts all the items into one state

LR(k) parsing: LR(0), SLR, LR(1), LALR

- define item
- construct set of states
- construct table
- parse seq. based on

LR(0) Parser • prediction of length 0 (ignored)

- LR(0) item: $[A \to \alpha . \beta]$

↑ closure - what a state contains
goto - have to move from a state to another
canonical collection } construct the set of states

- construct LR(0) table - one line for each state
  - 2 parts: action = one column (for a state, action is unique loc.

* 1) shift-reduce conflict
2) reduce-reduce conflict

LR(0)	action	goto $X \in N \cup \Sigma$
p_0		
p_1	$[S' \to S.]$	empty
p_i (k) $A \to \alpha.]$	reduce k	empty
p_n		

! a grammar is not LR(0) if the LR(0) table contains conflicts

*conflict = went to put 2 diff. values in the same cell (not in goto)

Rules: 1. if $[A \to \alpha . \beta] \in p_i$ then action(p_i) = shift

2. if $[A \to \beta .] \in p_i$ and $A \neq S'$ then action(p_i) = reduce l, l = prod. no. for $A \to \beta$

3. if $[S' \to S.] \in p_i$ then action(p_i) = acc.

4. if goto(p_i, X) = p_j then goto(p_i, X) = p_j

5. otherwise error

! if p is acc. ⇒ goto(p, X) = $\emptyset$

if in state p the action = reduce ⇒ goto(p, X) = $\emptyset$

Lecture 3: ex.:

(why the canonical coll is (not) infinite?)

SLR Parser (= simple LR)

- LR(0) → lots of conflicts (solved if considering prediction)
- prediction = next symbols on input seq.

⇒ 1. LR(0) canonical coll. of states - prediction of length 0

2. table and parsing seq. - prediction of length 1

- parsing steps ↑ define item
  construct set of states
  construct table
  parse seq. - based on moves between configs.

Construct SLR table:

1. prediction = next symbol from input seq. ⇒ FOLLOW
2. structure - LR(K): lines - states, action + goto

action - a column for each prediction $\in \Sigma$

goto - a column for each symbol $X \in N \cup \Sigma$

! optimize table structure: merge action and goto cols. for Σ

Remark (LR(0) table):

- if p is accept state then goto$(p, X) = \emptyset$, $\forall X \in N \cup \Sigma$
- if in state p action is reduce then goto$(p, X) = \emptyset$, $\forall X \in N \cup \Sigma$

Rules for SLR table:

(1) if $[A \to \alpha . \beta] \in p_i$ (dot is not at the end) and goto$(p_i, a) = p_j$ ⇒ action(p_i, a) = shift p_j

(2) if $[A \to \beta .] \in p_i$ (dot is at the end, but not for S') and $A \neq S'$ ⇒ action(p_i, u) = reduce k, where k = no. of production $A \to \beta$, $\forall u \in$ Follow(A)

(3) if $[S' \to S.] \in p_i$ (dot is at the end, prod. of S') ⇒ action$(p_i, \$)$ = acc

(4) if goto$(p_i, X) = p_j$ $\$$ ⇒ goto$(p_i, X) = p_j$, $\forall X \in N$

(5) otherwise error

Remarks: 1. similarity with LR(0)

2. a grammar is SLR if the SLR table does not contain conflicts (= more than one value in a cell)

Parsing sequences:

- **input**
  - grammar $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S')$
  - SLR table
  - input seq. $w = a_1 \dots a_n$
- **output**: if $(w \in L(G))$ then string of productions
  else error and location of error

! SLR = LR(0) configs. (α, β, π), α = working stack
β = input stack
π = output (result)

- initial config.: $(\$ s_0, w\$, \varepsilon)$
- final config.: $(\$ s_{acc}, \$, \pi)$

Moves: 1) **Shift** - if action(s_m, a_i) = shift s_j =>

$(\$ s_0 X_1 \dots X_m s_m, a_i \dots a_n \$, \pi) \vdash (\$ s_0 X_1 \dots X_m s_m a_i s_j, a_{i+1} \dots a_n \$, \pi)$

2) **Reduce** - if action(s_m, a_i) = reduce t and (t) $A \to X_{m-p+1} \dots X_m$
and goto$(s_{m-p}, A) = s_j$ =>

$(\$ s_0 \dots X_m s_m, a_i \dots a_n \$, \pi) \vdash (\$ s_0 \dots X_{m-p} s_{m-p} A s_j, a_i \dots a_n \$, t\pi)$

3) **Accept** - if action$(s_m, \$)$ = accept => $(\$ s_m, \$, \pi) =$ acc

4) **Error** - otherwise

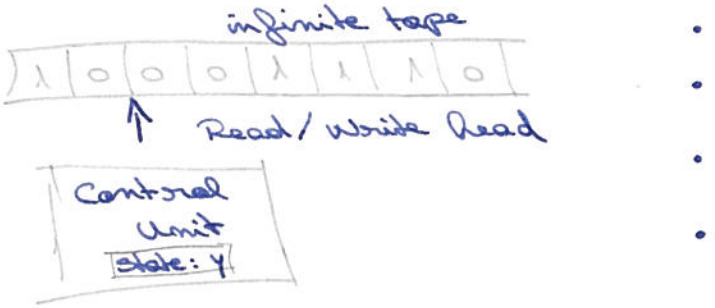
Course 13 :

- Computer with accumulator
- a stack machine consists of ↑

- Computer with registers ↑ registers + memory

- instructions ↑ LOAD v, R
  STORE R, v
  ADD R_1, R_2

Turing Machine - mathematical model for computation
- abstract machine
-



Course 14:

Subject 1: quiz 4 questions = 0.5p/each
- ! regular exp. (0^+, 1^*)
- ! ap. on languages
  - $L_1 \cup L_2, L^*, L^+, L_1 L_2 \ldots$
- ! accepted seq. by gram. or FA

Subject 2: ~~transformations RG-FA-Reg Exp~~ 2p

Subject 3: construct ! PDA 1p

Subject 4: parsing 2p
- descending recursive
- LL(1): First, Follow, constr. table, parse seq
- LR(0), ! SLR, ! LR(1): con. coll., table, parse seq.

Subject 5: ! attribute grammars, intermediary code 2p