Compiler construction

Mr.	Raphael	Finkel
Dr.	Rafi	Goldstein
Prof.	רפאל	
—		

---

user (client)
——————— specification.
implementation

---

programmer
——————— [syntax
compiler [semantics

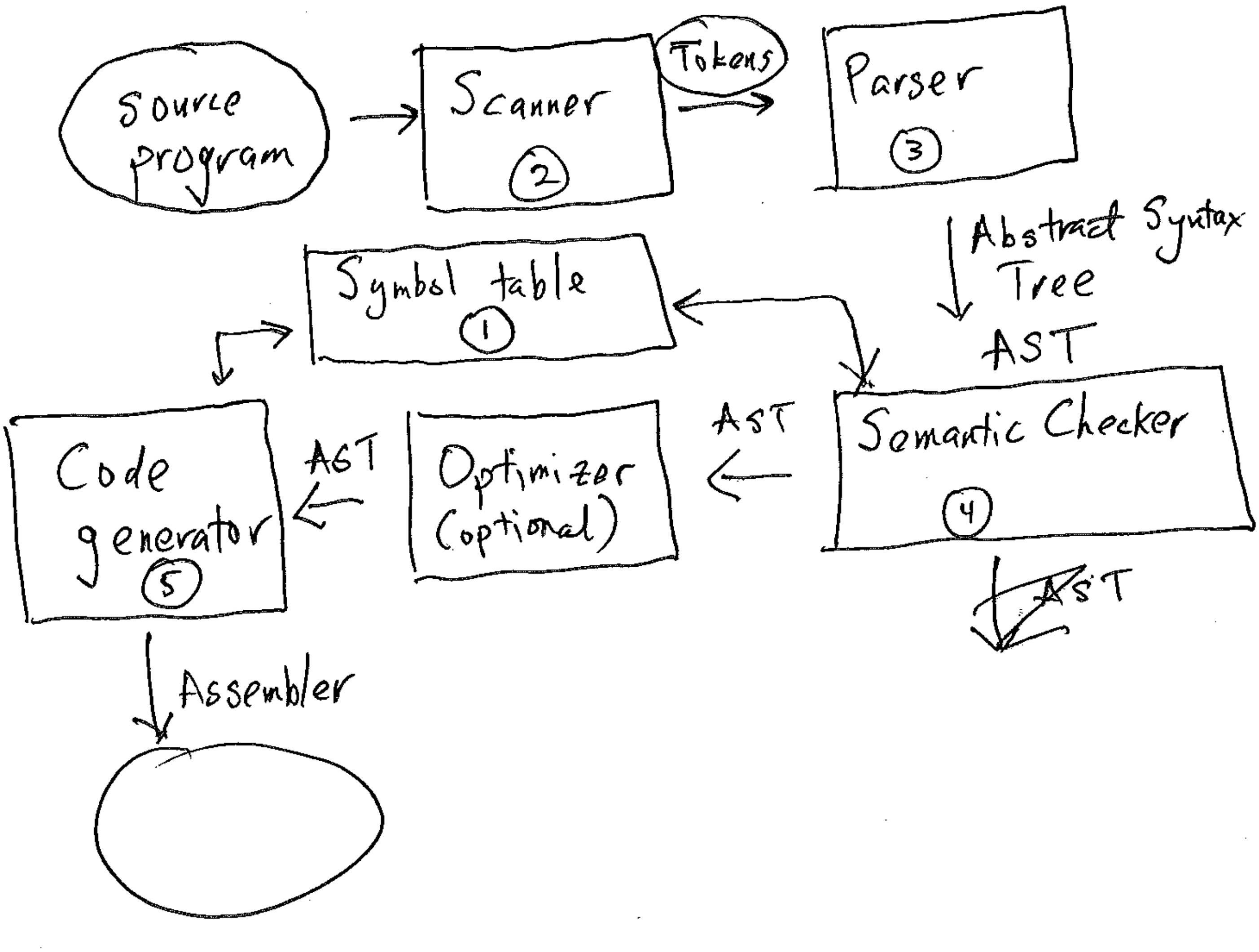
---

how compiler fits into a chain of operations

compiler outputs
- pure machine code (Linux Kernel)
- → augmented machine code
  - libraries
  - operating system calls. (common)
- → virtual machine code (Java) (CSX-go)

To represent the output
→ assembler (compiler does not need to resolve all references)
(modular compilation)
relocatable binary (Unix: .o file)
(defers resolving external references)
absolute binary

Structure of the compiler



Scanner

builds a stream of tokens

reads chars

Example: input: if (a < 39) {

Output:

if (ident, reserved)

"(" (punctuation)

a (ident)

< (punctuation)

39 (integer literal)

")" (punctuation)

{ (punctuation)

Method: regular expressions.

Scanner generator:

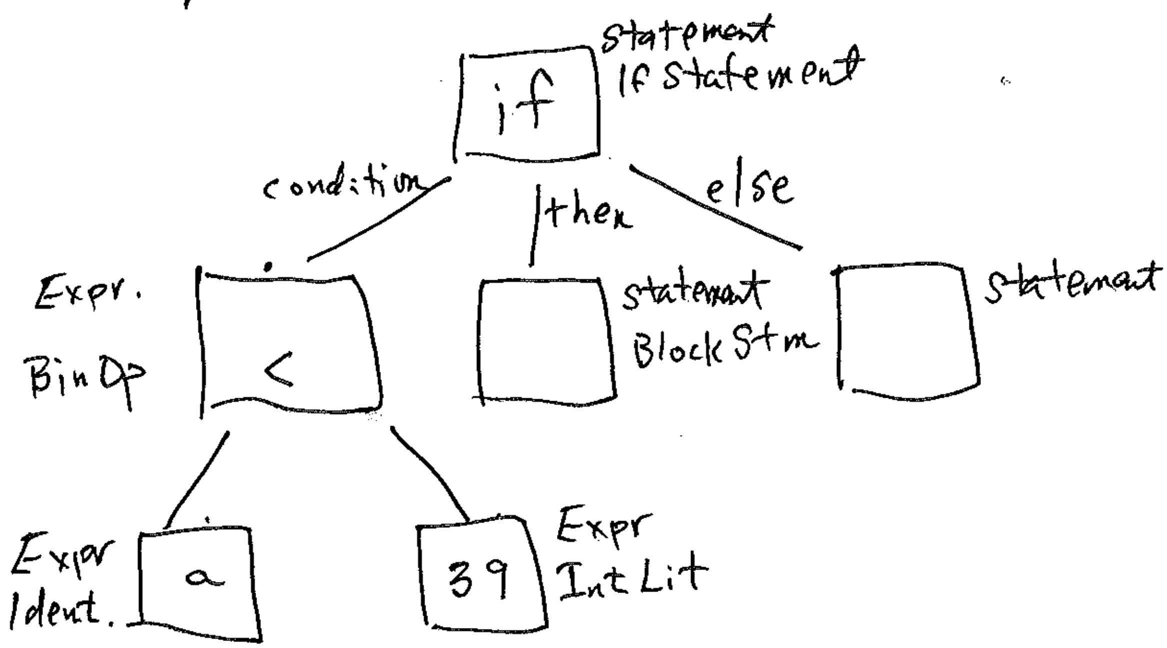
lex, flex, jflex

Parser

reads token stream

builds AST

example: <u>if</u> (a < 39) {



Method: Context-free grammar.

Parser generator:

yacc, bison, javaCup

Semantics Checker

Navigates through entire AST (DFS)

verifying declarations.

binding identifiers to ST (Symbol Table)

verify types of expressions,

adds info to AST. (type)

generate "error type" info.

add nodes to AST. (type conversion)

verify exception handling.

verify reachability.

Optimizer

Analyzes AST or generated code to

improve it (time, space)

many techniques.

simplify expressions.

moving code

eliminating trivial arithmetic.

. . .

Code generator

DFS of AST

generates output

assembler

. . .

Programming Lg Considerations

Special features

- passing parameter "by name"
- dynamic-sized arrays
- nested name scopes. (static chain)
- anonymous functions, first-class functions.
- multiple-yield iterators.
- automatic reclamation of object store (garbage collection)

Computer architecture considerations.

- limited registers.
- cost of operations.
- effect of memory hierarchy.

Specialty compilers

- Debugging support
- Participation in an IDE
- ~~Retarteg~~ Retargetable compilers.

1. $x_2 q = x_1 \wedge y_1$
2. $y_2 = x_2 \wedge y_1 = x_1 \wedge \overbrace{y_1 \wedge y_1}^{0} = x_1$
3. $x_3 = x_2 \wedge y_2 = x_1 \wedge y_1 \wedge \cancel{z} x_1 = x_1 \wedge x_1 \wedge y_1 = y_1$

The AC (adding calculator) language

typos: int, float

Keywords: f i p

variables: lowercase a .. z
except f, i, p

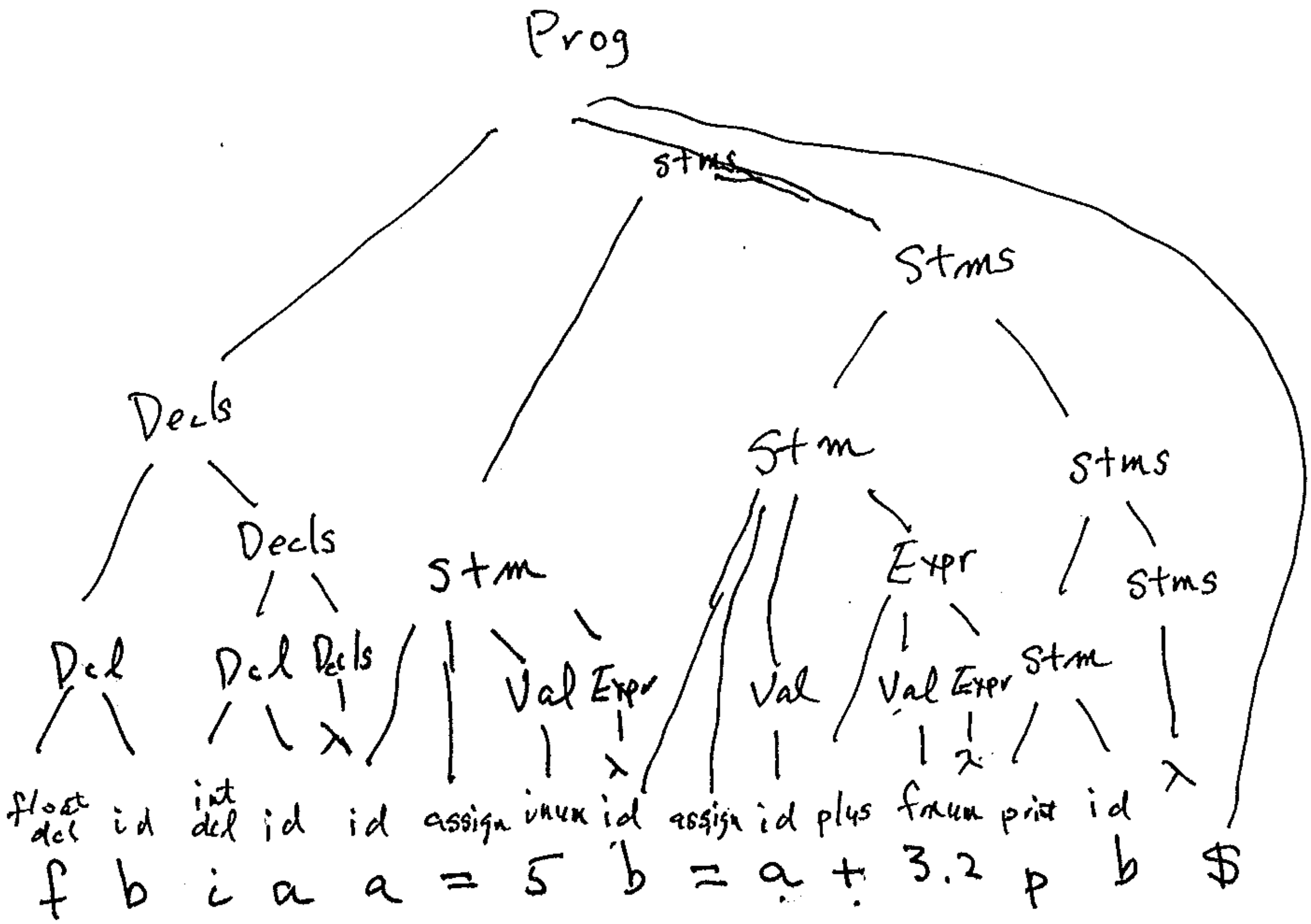
Program:

f b i a a = 5 b = a + 3.2 p b \$

Syntax in: Context-free grammar.
formalism: Backus-Naur Form (BNF)

Prog → Decls Stms \$	rule, production
Decls → Dcl Decls	LHS: a nonterminal
Decls → λ	RHS: nts + ts
Dcl → floatdcl id | intdcl id	any order
Stms → Stm Stms | λ	\$ is terminal
Stm → id assign Val Expr	
| print id	
Val → id | inum | fnum	
Expr → plus Val Expr | minus Val Expr | λ	

Parse Tree



Scanner: translates a stream of chars into a stream of tokens

type(~~operator~~ plus, inum ...)

value ("3.2")

design choices:

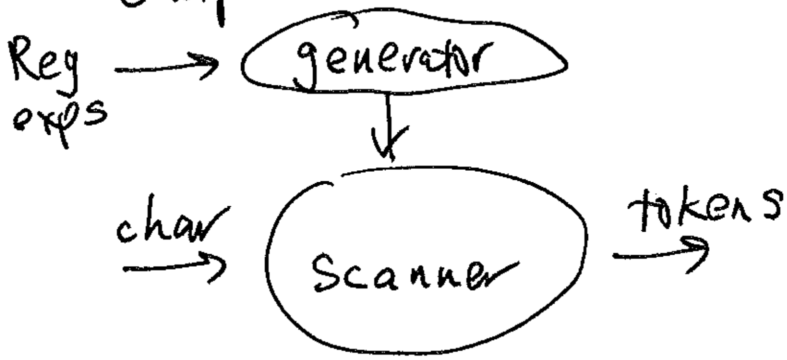
are all operators separate token types? (yes)

are all reserved words " (yes)

To build a scanner

1) Hard-coded (specific to lg)
2) Scanner generator (lex, flex, jflex)

Input: regular expressions.
Output: Scanner

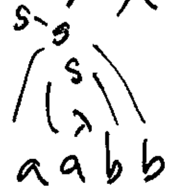


Formal lg hierarchy

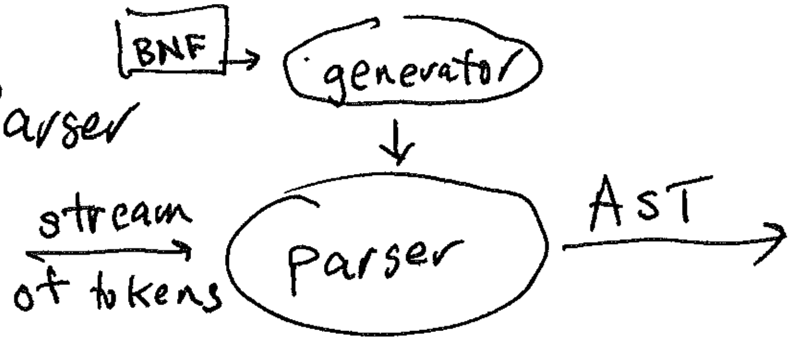
Language type	Formalism	Automaton
* Regular	Regular expression	FSA (finite-state automaton)
* Context-free	BNF CFG context-free grammar	PDA (push-down automaton)
Context-sensitive	CSG	LBA (linear-bounded automaton)
Type 0	various	Turing machine

$S \to aSb$

$S \to \lambda$



Parser



methods:

1) recursive descent (top-down)
   every non-terminal (in BNF) is represented by a procedure.
   case for each RHS for that non-terminal.
   match( ) for each terminal on RHS.
   (recursive) call to a procedure for each non-terminal on RHS.
   to pick which case:
   "predict sets"
2) LR(0) LR(1) SALR(1) (bottom-up)
3) Parser generator (yacc, bison, javaCup, ...)

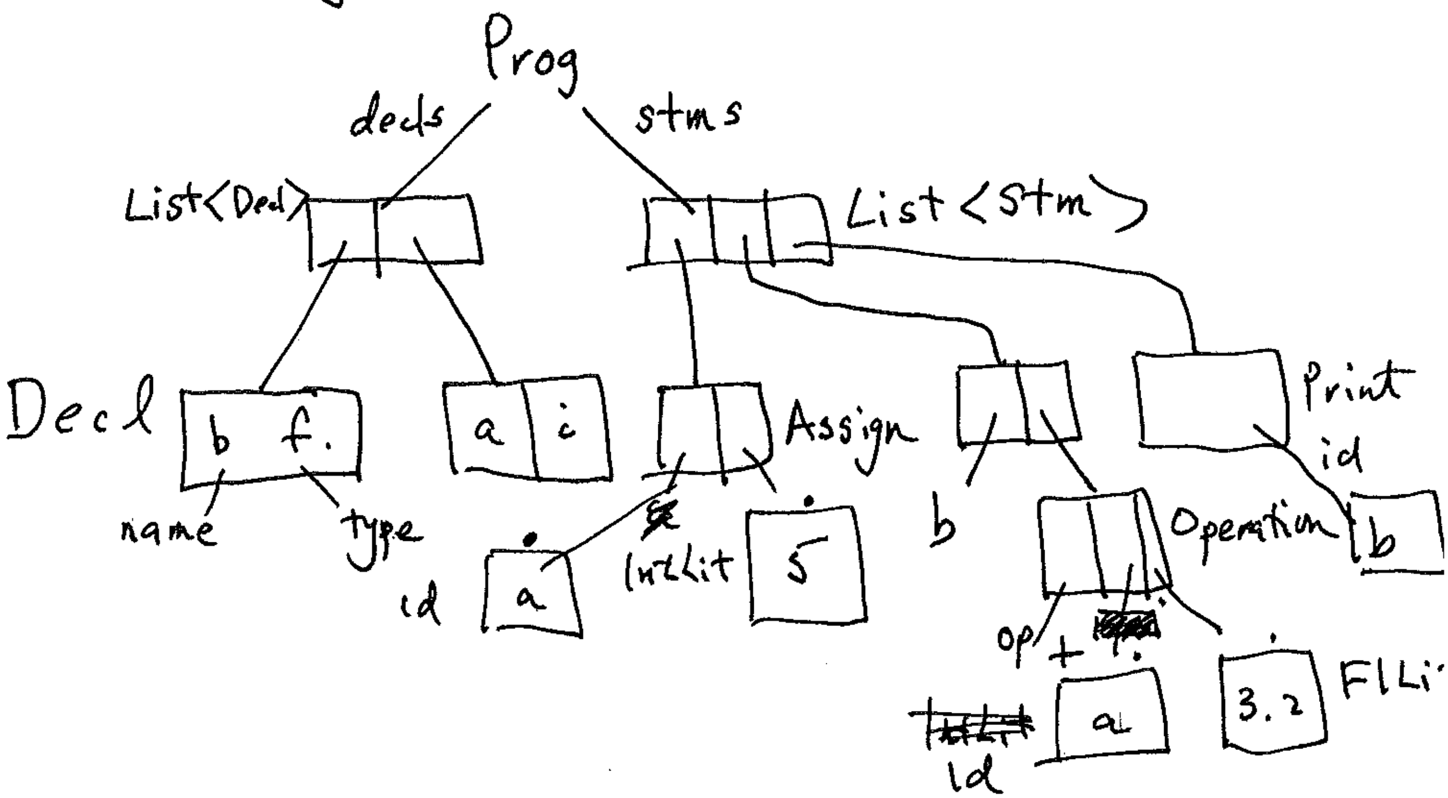
f b i a a = 5 b = a + 3.2 p b $

- Prog•
  - Decls•
    - Dcl•
    - Dcls•
      - Dcl•
      - Dcls•
  - Stms•
    - Stm•
      - Val•
      - Expr•
    - Stms•
      - ~~Val~~ stm•
        - Val•
        - Expr•
          - Val•
          - Expr•
      - ~~stms~~
      - Stms•
        - stm•
        - Stms•

Example:

Tree Prog(~~ts~~ Stream<Token> ts)

Abstract Syntax Tree



Semantic checking: recursively visits AST
Uses symbol table
at declaration: insert: id, type
verifying uniqueness.

at use:
verify existence
mark type

why: there are errors that scanning, parsing cannot catch.
a) strong typing.
b) disambiguate some meanings
Java: x.y.z
c) resolve overloaded operators.

Generating code

AST → code generator → key presses

Calculator:

- has registers, named by letters.
- load (L) store (S)
- precision: K button.
- print: P button.

Method: Visit AST depth-first calling codeGen()

Scanner: full story
formal, systematic

short story:
1) tokens are defined by regular expressions RE
2) which are encoded as non-deterministic finite-state automata (NDFA)
3) which ~~is~~ are converted to ~~a~~ deterministic finite-state automata (DFA)
4) which are described by ~~a~~ tables
state × input ⟹ action × state
5) which are executed by a simple program.

RE → $\boxed{\text{steps}}$ → table

table ↓ scanner

program → $\boxed{\text{scanner}}$ → tokens

shorter story:

RE → $\boxed{\text{scanner generator}}$ → table

complexities
1) string literals: can contain escaped "
2) over-eagerness
3..4 (Pascal)
'a (Ada)
DO 200 I = 1 . 10
3) speed.

Regular expressions

RE defines a <u>language</u>, which is the set of valid strings over an alphabet Σ

Σ: valid input characters.

1) $\emptyset$ [no strings at all]
2) λ [the empty string]
3) any letter in Σ [the string containing just that letter]
4) concatenation of REs. [all 2-part strings]
5) alternation of REs [all of each]
   (notation: | (bar symbol))
   (example: a | baλ)
6) closure operations [*: 0 or more concatenation] [+: 1 or more]
   (notation: * +)
   (example: ab*c)
7) parentheses for grouping
8) escaped metacharacters
   (example: ab*c)

Examples of REs.

// Java comment : $//\,(\text{not}(\underline{eol}))^*\,\underline{eol}$

Decimal literal :

$$(0|1|2|3|4|5|6|7|8|9)^+ \,.\, (0|1|2|3|4|5|6|7|8|9)^+$$

abbrev: $D = (0|1|\ldots|9)$

$$D^+ . D^+$$

Integer literal :

$$-^* D^*$$

$$(-|\lambda)D^+$$

$$(-|+|\lambda)\,D^+$$

Comment delimited by ##

$$\#\#(\Sigma)^*\#\#$$

$$\#\#\,(\text{not}(\#\#))^*\#\#$$

$$\#\#\,((\#|\lambda)\,\text{not}(\#))^*\#\#$$

question: ##a### not in set

Balanced braces

example: { { } { } } not a regular set

Fortran real literal:

need at least one digit on at least one side of the decimal point.

$$D^+ . D^* \mid . D^+$$

Identifier, including underscores (_) but:

never adjacent, frontal, or terminal.

not starting with digit.

$$L\,(L \mid D)^*\,(\_\,(L \mid D)^+)^*$$

$$L\,((\_ \mid \lambda)(L \mid D))^*$$

Hashing: $\underset{\text{string}}{\text{Key}} \longrightarrow \underset{\text{def.}}{\underset{\cancel{\text{int}}}{\text{value}}}$ (lookup function)

$\Theta(1)$

method: introduce a function from Keys to integer indices.

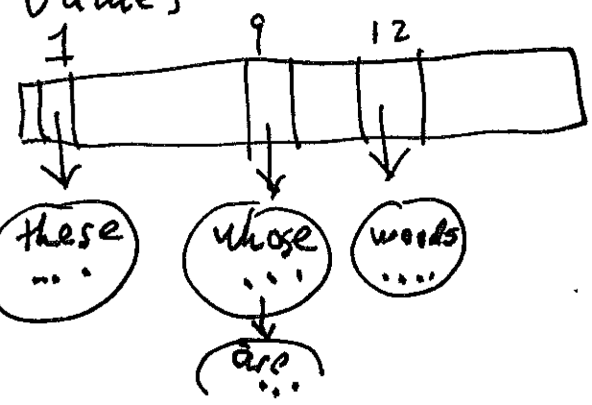
Build an array indexed by those indices containing the "values"

h("whose") → 9

h("woods") → 12

h("these") → 1

h("are") → 9



implements

Difference between Map and HashMap?

Interface   class

~~Map<String, Symb> hm~~

```
Map<String, Symb> hm =
      new HashMap<String, Symb>;
                  <>
List<int> li =
      new LinkedList<int>;
```

RE $\longrightarrow$ automata

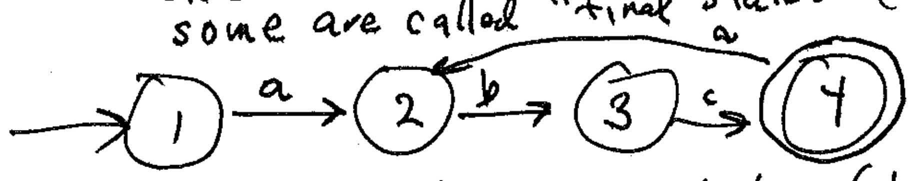
FSA : finite-state automaton

model of a simple computer.

Set of states (depicted by circles)

One is called "start state" (arrow from nowhere)

some are called "final states" (double circle)



transitions between states (labelled arrow)

member of Σ

abc | abcabc | . . .

(abc)$^+$

Equivalent concepts

1. Regular expression (RE)
   Regular language
3. Finite-state automaton (deterministic)
2. Non-deterministic finite-state automaton
4. table: rows are states
   columns are inputs
   cells are state numbers

	a	b	c	
∅ 1	2	5	5	
2	5	3	5	
3	5	5	4	
4	2	5	5	final
5	5	5	5	

---

generate : produce sentences in lg.
recognize : accept only sentences in lg.
transduce : take sentences in lg, create new sentences in other lg.

FSA is deterministic if:

1) no conflicting transitions

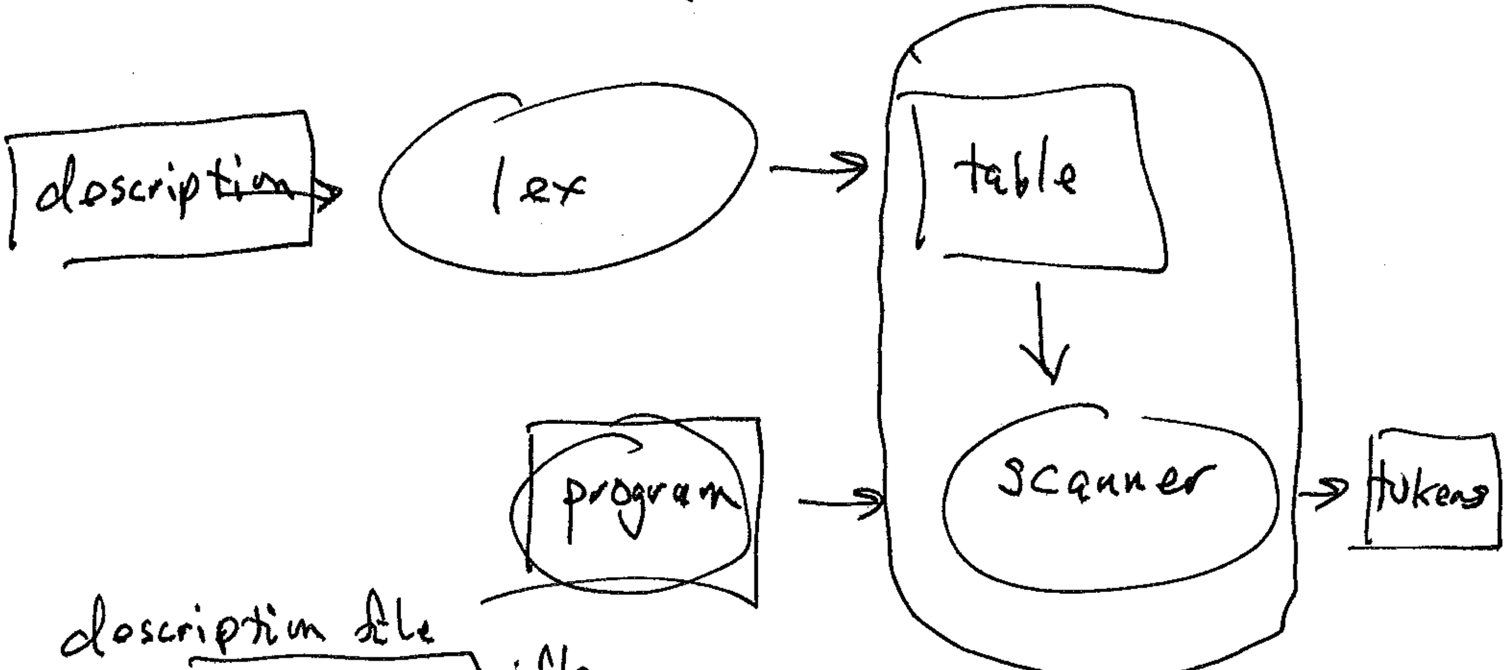
and

2) no λ transitions

else: non-deterministic

---

lex flex jflex . . . .



description file

jflex

Java code

%%

shorthands

%%

rules.

%ignorecase

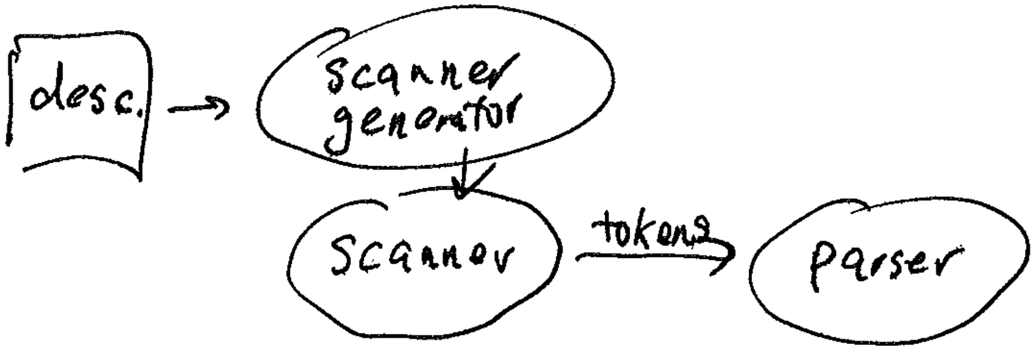
character classes

D = [0-9]

L = [a-z] or [a-zA-Z]

Letter = [a-e] | [g-h] | ~~[j-n]~~ [j-o] | [q-z]

~~F~~(f|F)(o|O)(r|R)



shared token codes : # small integers

usually : table built by parser generator

yacc: y.tab.h

javaCup: sym.java

Regular expressions for lex

bracket syntax for a range of values

[a-i] "character classes"

escape syntax for metacharacter

\[\+ \\

"[" "+"

no need to quote alphanumeric characters

case is significant

[pP][rR][iI][nN][tT]

%ignorecase

metacharacters : * + | ( ) . ?

^ : start of an input line

$: end of an input line

processing code: in braces; opening brace
on same line as its RE.

Where is the text that was matched?

lex: string variable: yytext

jflex: method: yytext()

evanescent!

If two REs overlap:

longest match wins.

matching same string: earlier rule wins.

Catch-all RE:

- error rule to match any character (at end)

End-of-input: token with code 0.

~~Wie~~

Weird situations

Pascal 31..42

Back up to last accepting state

Ada foo'bar

Right context: [0-9]+/".."

Alternatives to lex (flex, jflex, ...)

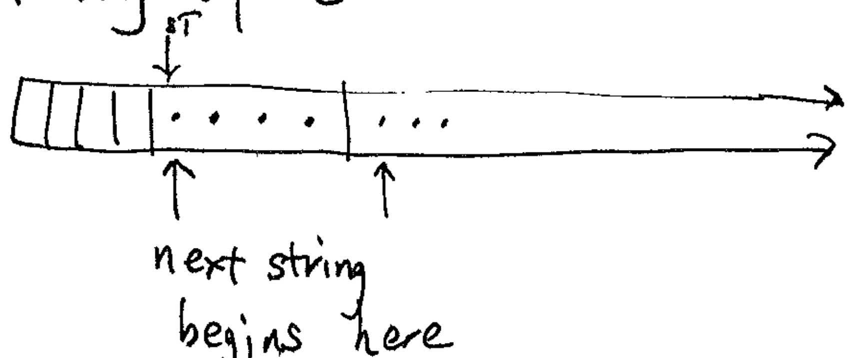
GLA, re2c: generate direct code

Practical considerations

Identifiers : put in ST or not?

If lg not block structured, yes.

String space



If case is significant:

don't convert case.

If case not significant

convert all identifiers to lower.

Literals

Integers : convert to internal (binary) representation.

out of range?

String : expand escaped characters

also store original form.

Reserved words (for, while, ...)

1) each one associated with a different token type.

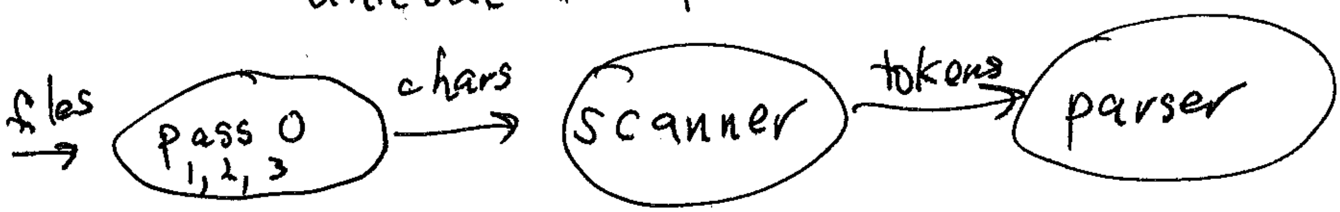
2) treat like identifiers, but look them up in a special table.

Compiler directives

1) file inclusion: stack of open files (recursion)

2) conditional compilation:
(for debugging, to customize code for specific environments)
previous step before the scanner (pass 0)

3) Unicode escapes



End of file: generate an EOF token.

Multi-character lookahead

when scanner enters an error state,
back up, returning characters
until it enters an accepting state.
If none, announce "error", skip one character.

Speed: use scanner generator
buffer reads
use a profiling tool

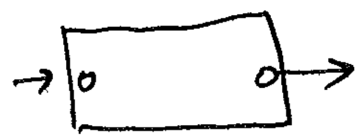
Error recovery

generate an "error" token.

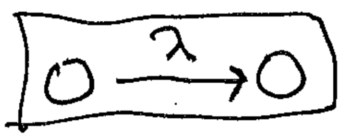
runaway strings and comments.

↓ might disallow crossing line boundary.

$RE \overset{①}{\Longrightarrow} NDFA \overset{②}{\Longrightarrow} FSA$

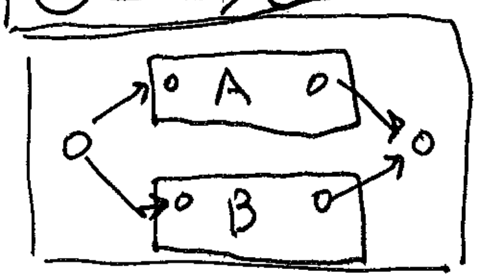


λ

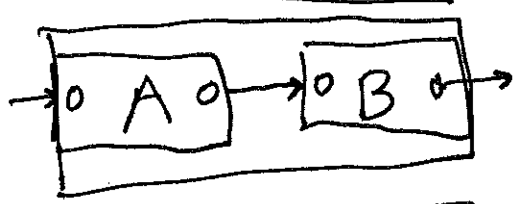


a

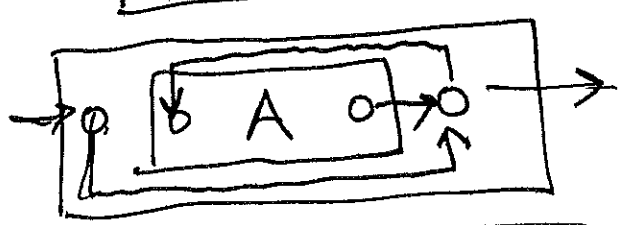
A|B



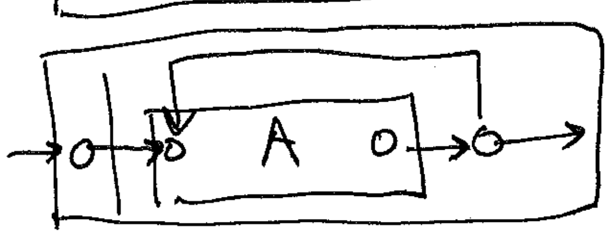
AB



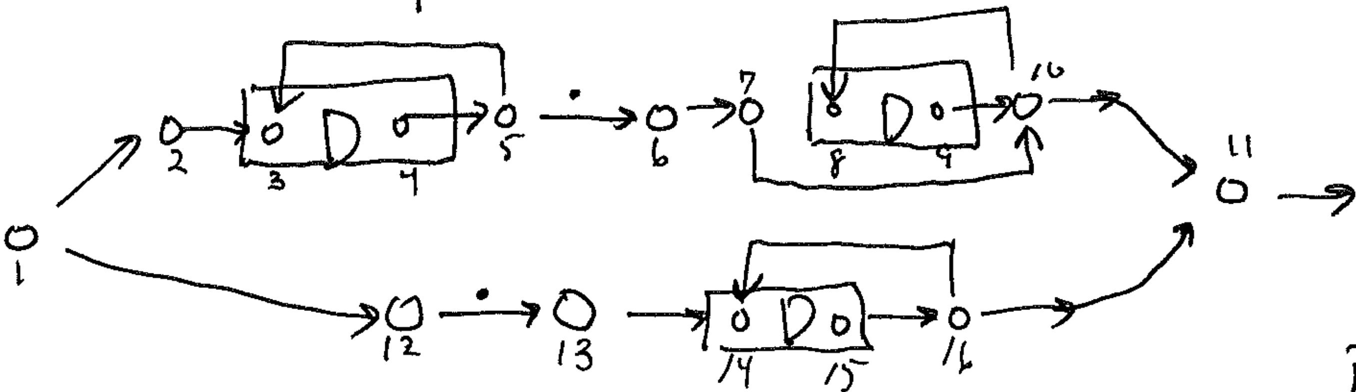
A^*



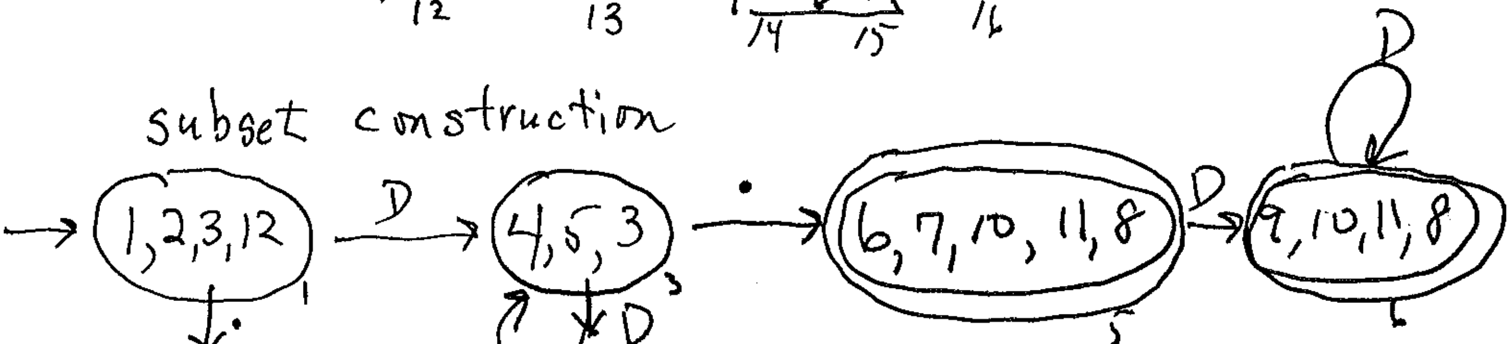
A^+

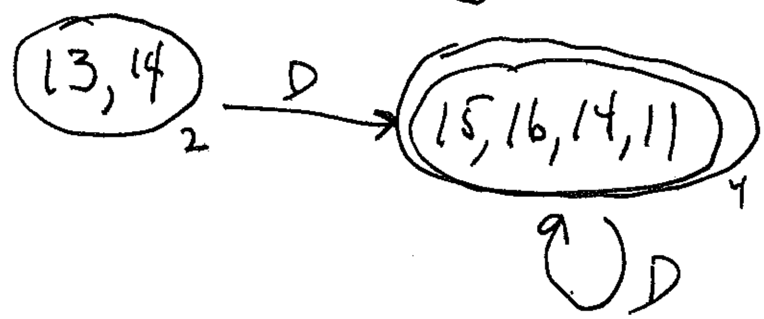


$D^+ . D^* \mid . D^+$

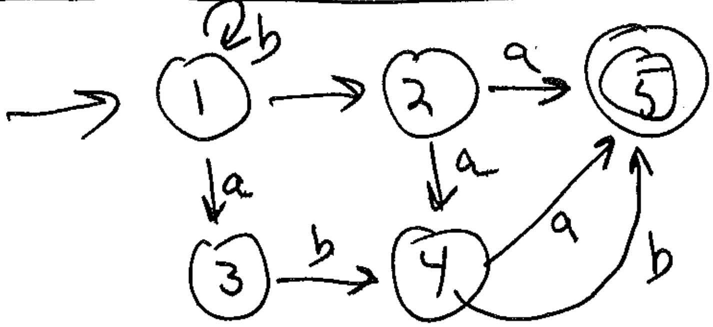


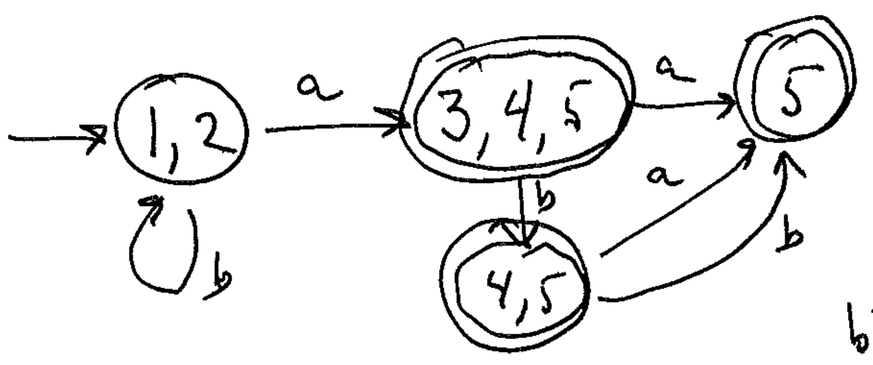
subset construction





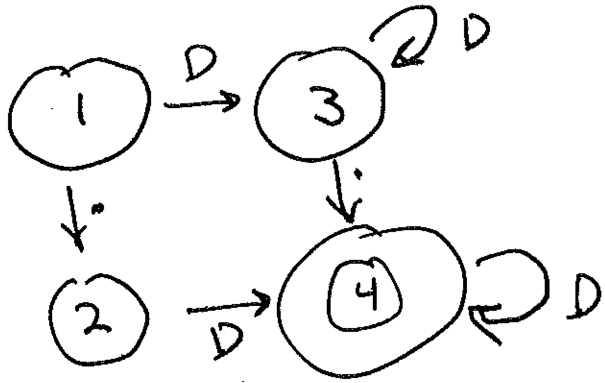
	D	.
1	3	2
2	4	
3	3	5
* 4	4	
* 5	6	
* 6	6	





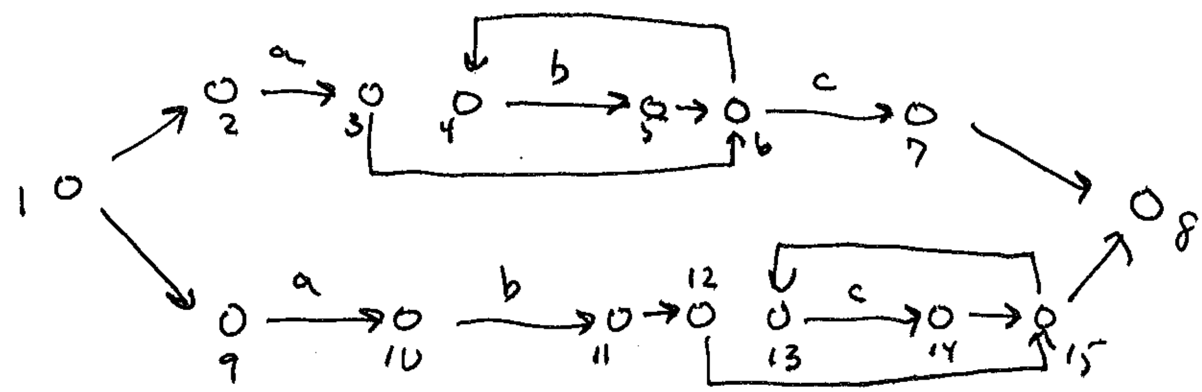
~~b* a (a|b~~

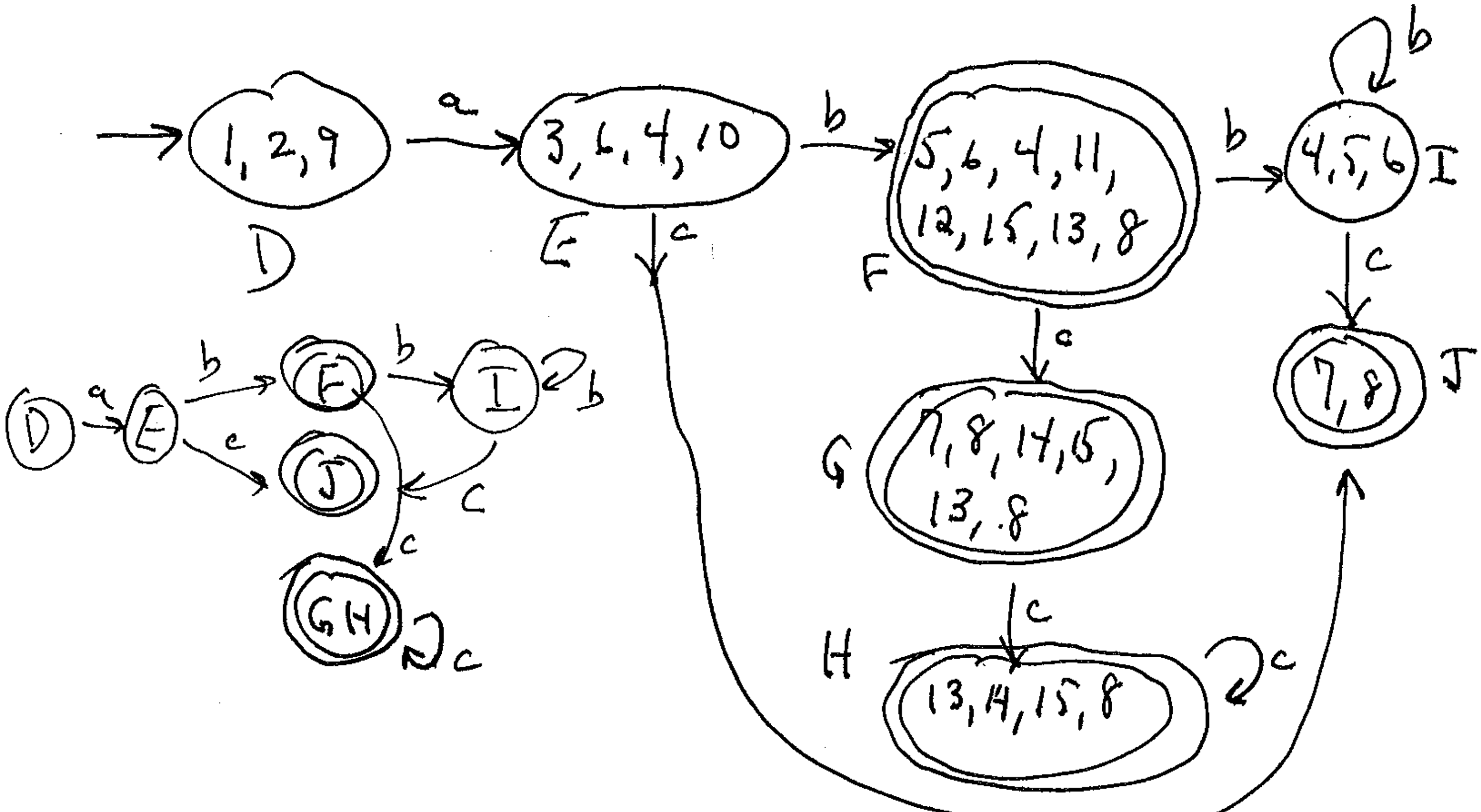
$b^* a(\lambda \mid a \mid b(\lambda \mid a \mid b))$

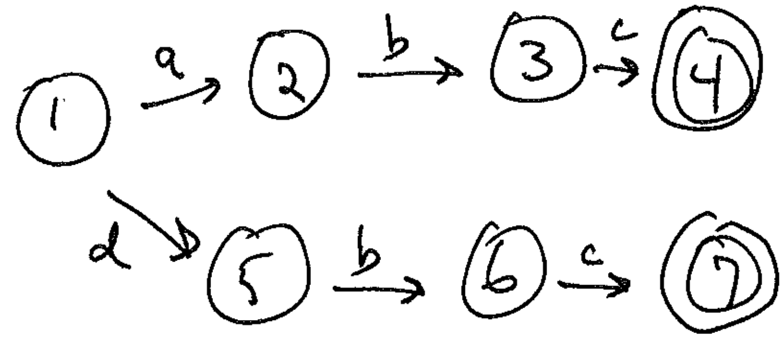


	D	.
1	3	2
2	4	
3	3	4
*4	4	

RE: $(ab^*c) \mid (abc^*)$

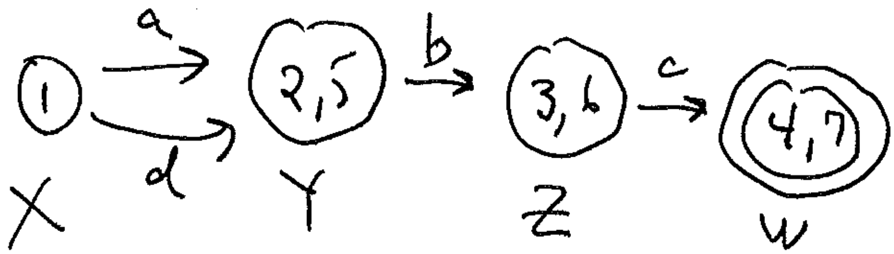






p. 98

	a	b	c	d
1	2	·	·	5
2	·	3	·	·
3	·	·	4	·
*4	·	·	·	·
5	·	6	·	·
6	·	·	7	·
*7	·	·	·	·

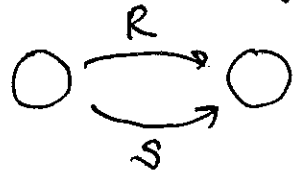


	a	b	c	d
X	Y	·	·	Y
Y	·	Z	·	·
Z	·	·	W	·
*W	·	·	·	·

RE → NDFA → DFA → RE

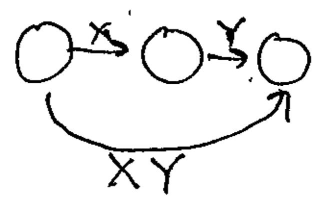
by transforming the DFA

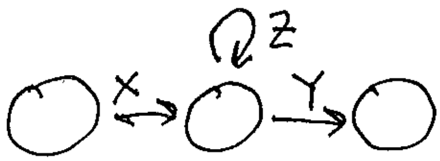
1) introduce a single accepting state

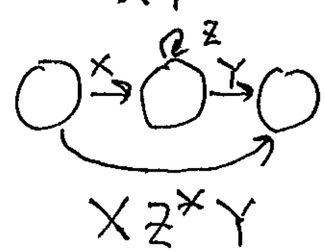
2) 

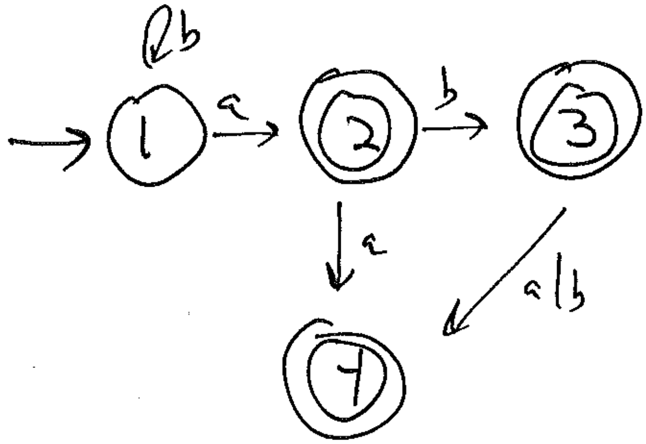
replace by

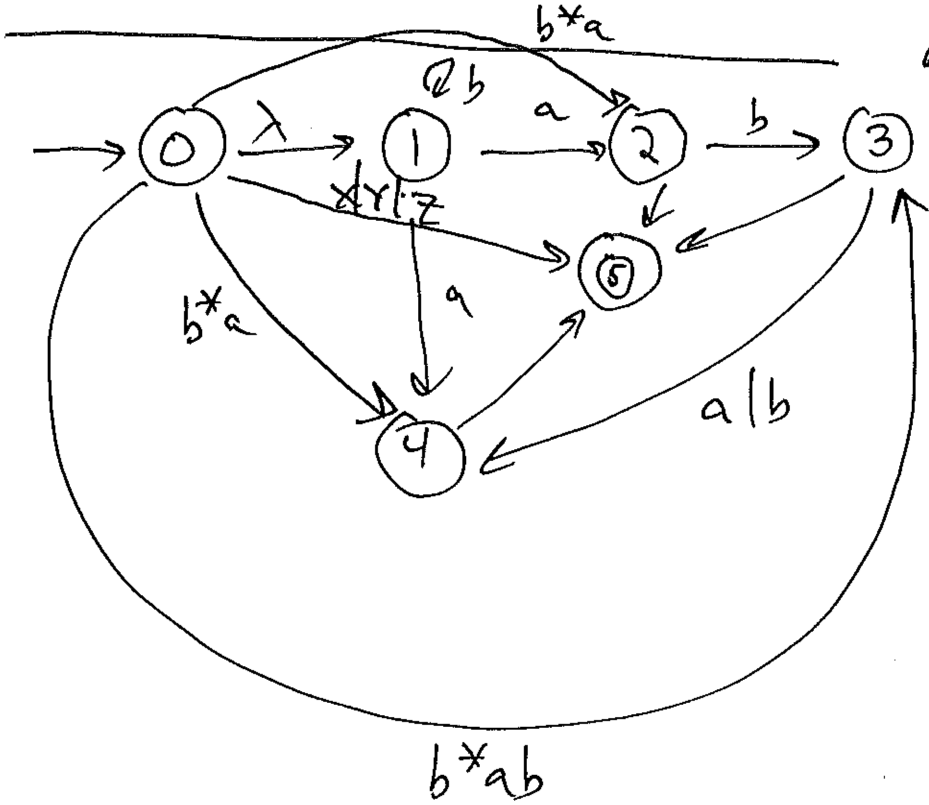

3) 

replace by


4) 

replace by






This figure is mis-copied!

X b^*a

Y b^*ab

Z b^*a

W $b^*ab(a|b)$

Midterm: delayed by 1 week!

Chapter 4.

Context-free [grammars CFGs
languages — set of strings

Strict superset of regular languages.

instead of a DFA, need a push-down automaton. (PDA)

We won't use PDAs

we will use CFGs

alphabet of terminal symbols. (tokens)

set of non-terminal symbols

convention: CamelCaps

start symbol (non-terminal)

usually P

rules (productions)

$$A \rightarrow X_1 \ \ldots \ X_m$$

nonterminal

terminal or nonterminal

$$A \rightarrow \varepsilon$$

epsilon (empty RHS)

language of rules: BNF

Backus Naur Form

$$\underbrace{\overbrace{\overbrace{P \Rightarrow A\ B}\Rightarrow a\ a\ B}^{\text{sentential form}}}_{\text{derivation}}$$

Conventions:

leftmost derivation:

~~all~~ always expand the leftmost non-terminal

rightmost derivation

always expand the rightmost non-terminal

read off the derivation backwards

Parse tree

describes a derivation

start symbol is the root.

interior nodes are nonterminals

leaves are terminals.

Regular grammar: like CFG, but RHS only: a single element of Σ followed optionally by a single non-terminal.

CFG: represented by BNF, can be parsed in $\Theta(n^3)$ time.

Useful CFGs can be parsed in $\Theta(n)$ time.

Context-sensitive grammar
like CFG, but LHS can ~~be~~ have extra "context" around the non-terminal.
can cover the concepts of "declared".
but not worth it.

Type-0 grammar
rules let you rewrite arbitrary patterns.

---

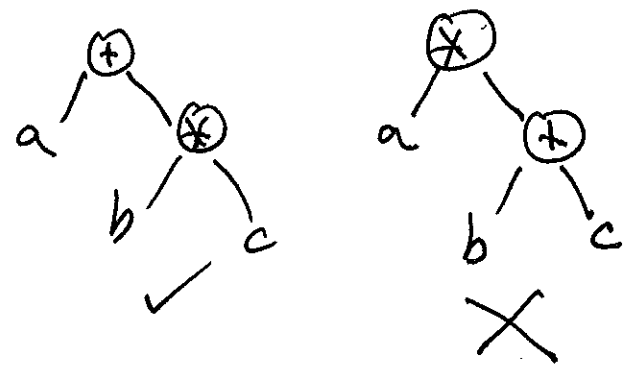
what kind of CFG?

Not ambiguous. (undecidable)

1	$S \to Aa$	$S \overset{1}{\Rightarrow} Aa \overset{3}{\Rightarrow} aa$
2	$\quad\to aA$	$S \overset{2}{\Rightarrow} aA \overset{3}{\Rightarrow} aa$
3	$A \to a$	
4	$B \to a$	

Not unreduced
~~not~~ nonterminals that cannot be reached.

Not wrong language.
$a+b*c$



Extended BNF

to have optional part of RHS

[optional part]

to have 0 or more repetitions

{ this stuff }

$A \to a\ [b\ c]\ d$

rewrite as

$A \to a\ N\ d$

$N \to b\ c$

$\to \lambda$

$A \to a\ \{b\ c\}\ d$

rewrite as

$A \to a\ N\ d$

$N \to \lambda$

$\to b\ c\ N$

Notation

recognizer : string → bool

parser : string → parse tree

top-down parser : left most derivation LL(1)

bottom-up parser : right most derivation LR(1)

1 symbol lookahead

leftmost

examine L → R

How to represent a BNF grammar.

~~Rules~~ BNF: Hash table mapping
Non-terminals → List of Rules.

Rule: List of RHS elements

Element: either a non-terminal or
a terminal

Java:

~~List~~ ~~RHS~~

decl
```
List <Element> RHS = . . .
```
~~for (e:E~~

code
```
for (Element e: RHS) {
    // use e
}
```

advice: alternative is to use
```
i = RHS.iterator()
i.hasNext()
i.next()
```
if navigating the structure requires modifying it.

LL(1) : top down
recursive descent

magic to be explained: how to choose among
competing rules.

Algorithms.

1) RuleDerivesEmpty (rule) → bool
2) SymbolDerivesEmpty (nonterminal) → bool
3) First(nonterminal) = set of terminal
4) Follow(nonterminal) = set of terminal
5) Predict(rule) = set of terminal

1, 2) List L all nonterminals that directly
yield λ.

for each $N \in L$, find all rules where
N in the RHS. Pretend to remove it.
(by subtracting from count of RHS
symbols)

If any RHS count becomes 0,
add its LHS to L.

$$\text{First}(\alpha) = \{ b \in \overset{\text{Terminals}}{\Sigma} \mid \alpha \overset{*}{\Longrightarrow} b\beta \}$$

String of symbols

Algorithm

Hint: if BNF is written top-down, then compute First() from the end to the beginning.

Consider first symbol of α.

Easy: α is empty. ~~anser~~ answer = { }

first symbol is terminal
answer = {that symbol}

Hard: first symbol is a nonterminal, N.

for each rule with LHS = N,
compute First(RHS) [avoid computing First(N)]
answer = Union of all those computations.

if N can derive λ, then union in ~~the~~ First(rest of α)

Follow(N) = { terminals that can come after N }

$$= \{ b \in \Sigma \mid \underset{\substack{\uparrow \\ \text{start symbol}}}{S} \overset{*}{\Longrightarrow} \alpha N b \beta \}$$

Algorithm

Hint: if BNF is top-down, start at top and work down.

for each place that N appears in a RHS of some ~~pro~~ rule r,

union in First($\underbrace{\text{symbols following N}}_{\text{"tail"}}$)

if tail can derive λ (or is empty), union in Follow(LHS(~~p~~ r))

Predict (r) = First (RHS (r))

<u>if</u> r derives λ <u>then</u> Union Follow(LHS(r))

Recursive-descent parser $(LL(1))$

top-down

1 token lookahead

read from start of program

One procedure for each non-terminal.

A case for each rule, determined by the predict set.

"Absorb" the RHS by match (terminals)

recursive call (for non-terminals)

```
proc P() {
    switch (peek()) {
        case f, case i, case id, case p, case $:
            Ds();
            Ss();
            match($);
            break;
        default:
            error()
    } //switch
} // proc P()

proc Ds() {
    switch (peek()) {
        case f, case i : D(); Ds(); break;
        case id, case p, case $: break;
        default: error();
    } //switch
} // proc Ds()
```

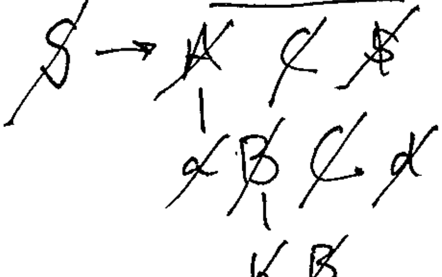
p. 148 example

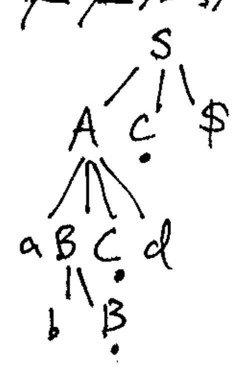
39

	First(RHS)	Follow(T)	Predict
S → A C $	{a, b, q, c, $}	∅	{a, b, q, c, $}
* C → c	{c}		{c}
→ λ	∅	{$, d}	{$, d}
* A → a B C d	{a}	{c, $}	{a}
→ B Q	{b, q}		{b, q, c, $}
* B → b B	{b}	{c, d, q, $}	{b}
→ λ	∅		{c, d, q, $}
* Q → q	{q}	{c, $}	{q}
→ λ	∅		{c, $}

```
proc A() {
    switch (peek()) {
        case a : match(a); procB(); procC(); match(d);
                 break;
        case {b, q, c, $}: procB(); procQ(); break;
        default: error( )
    }//switch
}//procA
```

Sentence : ~~a B d~~ ~~a~~ ~~b~~ ~~d~~ ~~$~~





	Predict
$S \to a\ b$	$\{a\}$
$\to a\ c$	$\{a\}$

conflict in predict sets $\Rightarrow$ grammar not LL(1)

fix: (factor out common prefix)

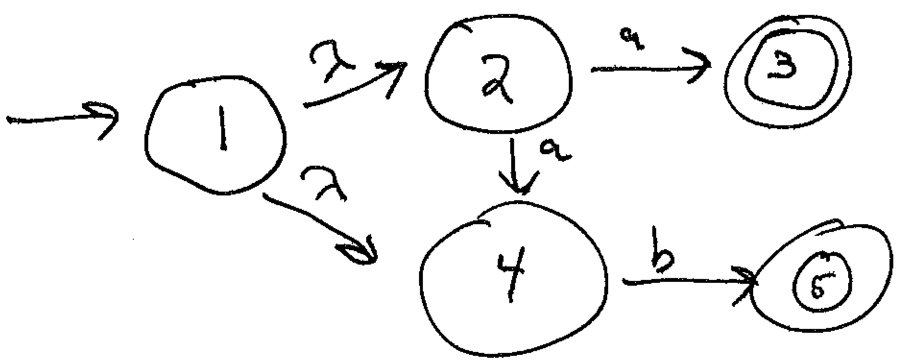
~~$S \to A\ b$~~

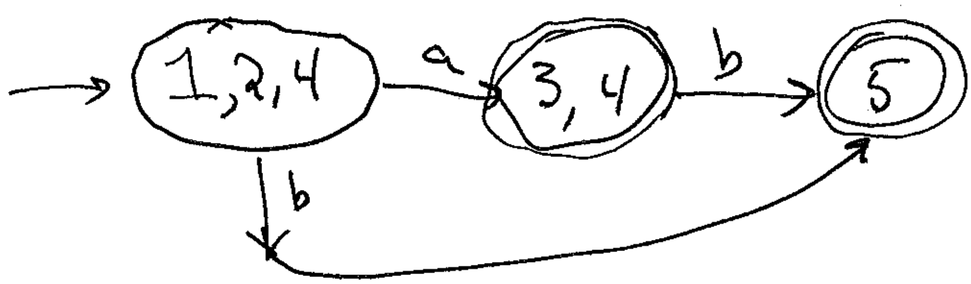
$S \to a\ B$

$B \to b$

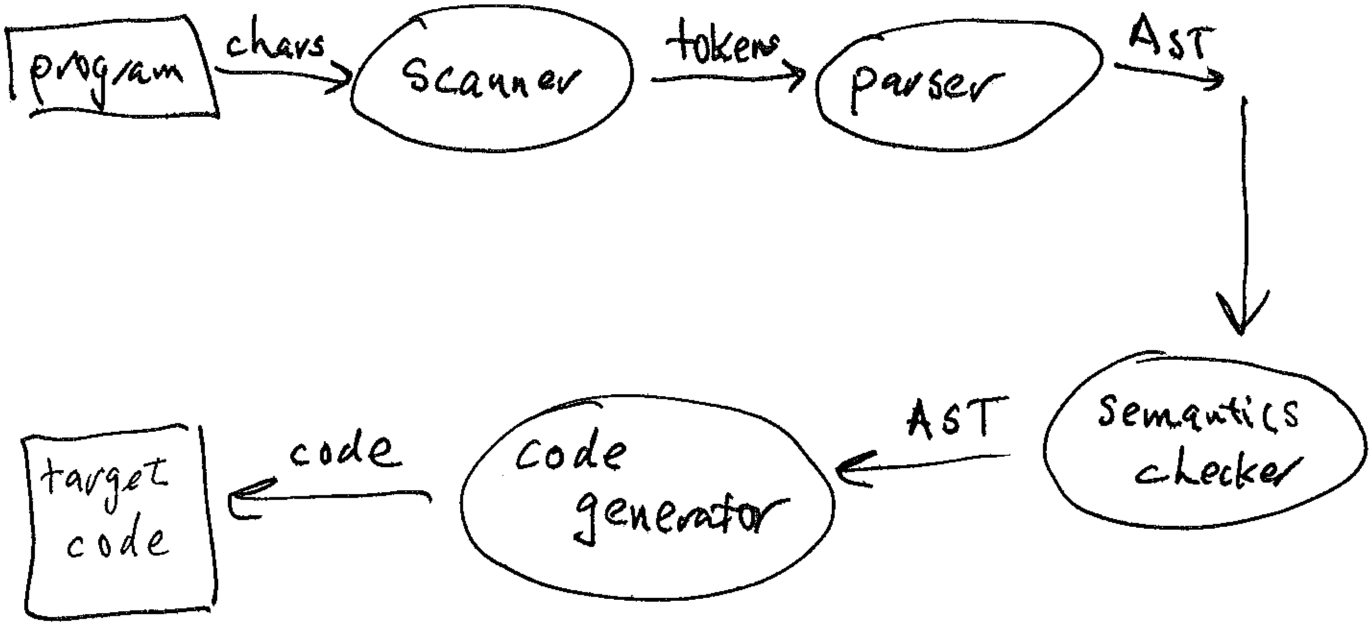
$\quad \to c$

Subset Construction





1a)



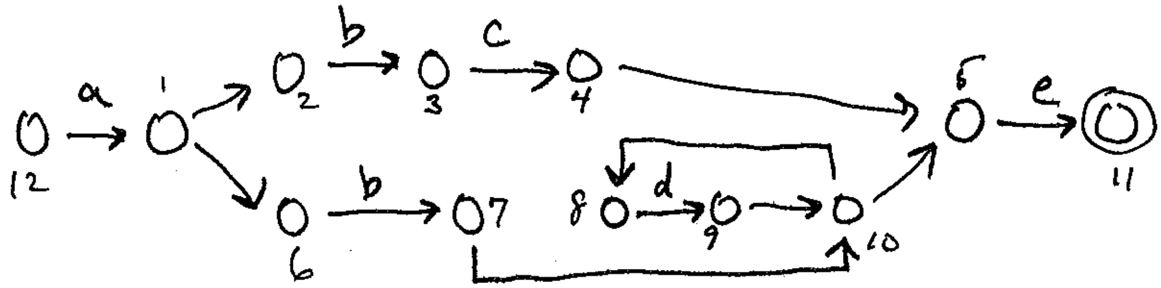
1b) jflex

1c) no

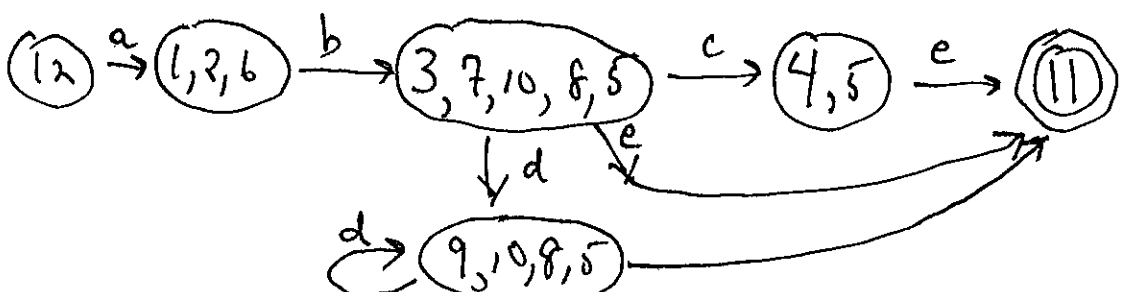
1d) $\Theta(1)$

$$a(bc \mid bd^*)e$$

2a)



2b)
2c)



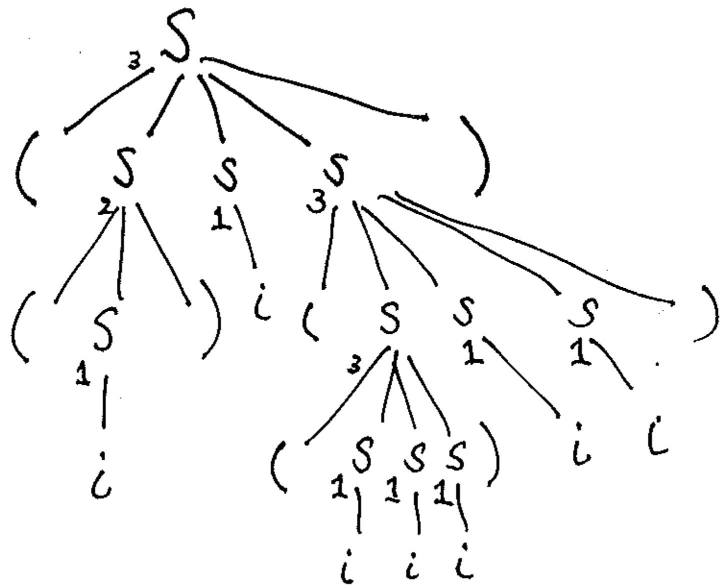
2d) abdde

1 $S \rightarrow i$

2 $S \rightarrow (S)$

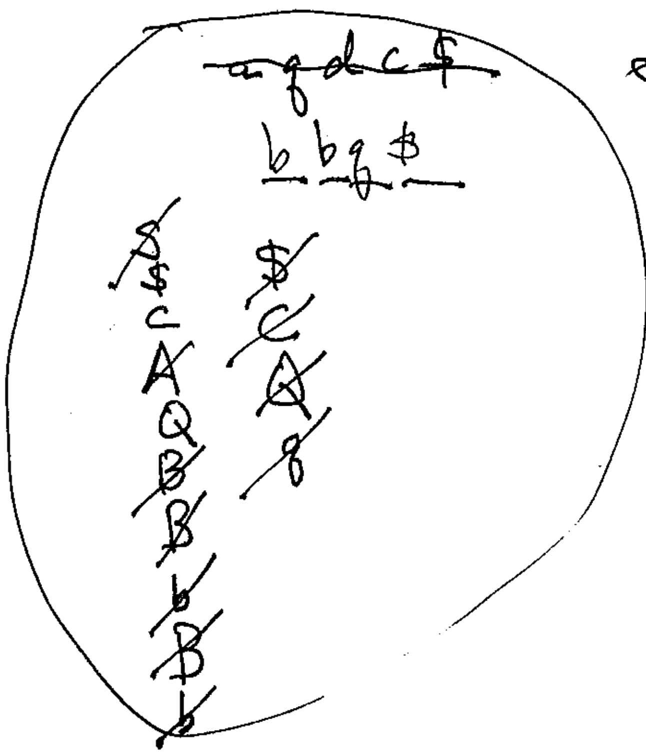
3 $S \rightarrow (S\ S\ S)$

$((i)\ i\ ((i\ i\ i)\ i\ i))$

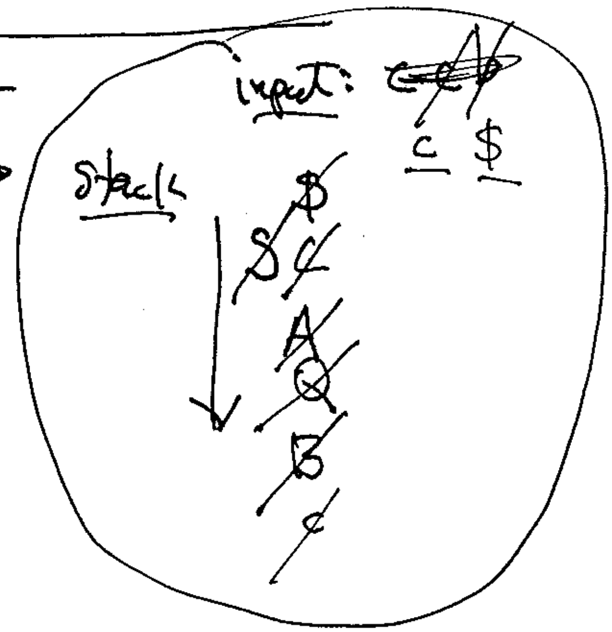


Not LL(1)

Table-driven LL(1) parser



examples



Advantage:
"generic" parser; the parser generator just builds a table.

Problem: The parse table can be sparse.

Solution: 1) Live with it.

2) Compress the table.

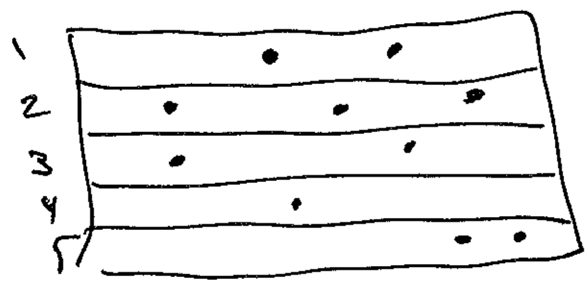
How?

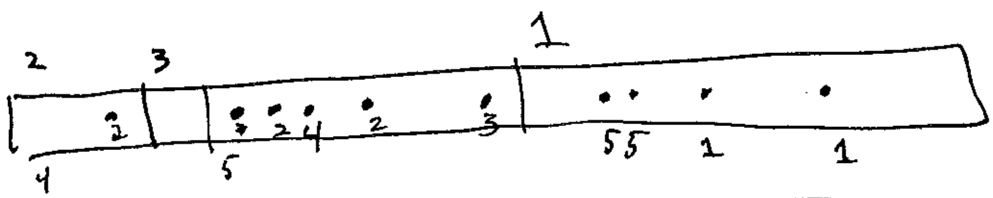
a) default value for each row.

b) hash the table:

$T[i,j]$: search at $h(i,j)$

c) double-offset indexing





To make a grammar LL(1)

1 $S \to id$	1 $S \to id$
2 $S \to (S\ )$	2 $S \to (S\ R$
3 $S \to (S\ S\ S\ )$	3 $R \to\ )$
	4 $\quad \to S\ S\ )$

<u>factoring</u>

$S \to$ if E then S_s end
$\quad \to$ if E then S_s else S_s end

factor:

$S \to$ if E then S_s R

$R \to$ end
$\quad\;\;$ else S_s end

---

fixing left recursion

$$S_s \to S_s \underbrace{\boxed{;\ S}}_{\alpha}$$
$$\quad \to S$$

$S_s \to X\ Y$

$X \to S$

$Y \to ;\ S\ Y$

$Y \to \lambda$

$E \to$ var $+ E$
$\quad \to$ var

$E \to$ var R

$R \to + E$
$\quad \to \cancel{\text{var}}\ \lambda$

General case:

$A \to A\alpha$

change to:

$A \to X\ Y$

new nonterminals

for every other production

$A \to \beta$

change to: $X \to \beta$

add production

$Y \to \alpha Y$

add production

$Y \to \lambda$

Dangling "else" problem

if B_1
    if B_2
        S_1
    else
        S_2 ;

2	S →	if E then S V	
3	→	other	
4	V →	else S	
5	→	λ	

table

	if	~~B~~E	then	else	other
S	2				3
V				4,5	

Properties of LL(K) parsers

1) compute correct leftmost parse
2) grammar is unambiguous
3) table driven parser uses $\Theta(n)$ time, $\Theta(n)$ space. (n = # of tokens)

Recovering from syntax errors.

1) print line, col, indicate what terminal was expected (predict set or match() param), what was seen.
2) try to enter a state you can continue from.
   - bad recovery causes a cascade of errors.
   - "panic mode": skip input until find a frequent delimiter, like semicolon.
   - Wirth: pass extra parameter to match(): set of possible lookaheads.

Repairing syntax errors

modify input to obtain acceptable parse.

situation

program

$$\underline{\alpha}\; x \;\beta$$

$\uparrow$

erroneous

1) modify α: prefer not, because α is fine as it is.
2) insert δ to get $\alpha\,\underline{\delta}\,x\,\beta$. "insert-correctable language"
3) delete x and keep going. makes progress.

Chapter 6: Bottom-up parsing.

good news: handles largest class of grammars that can be parsed deterministically.

bad news: less intuitive than top-down.

general idea: start with leaves of the parse tree: terminals.

occasionally reduce a string of ~~terminals~~ symbols to a non-terminal.

parsers are called <u>shift-reduce</u> parsers.

LR(K) — number of tokens of lookahead

L: read from start to end of program

R: rightmost derivation

1	$S \to E\ \$$	
2	$E \to +\ E\ E$	
3	$\to n$	

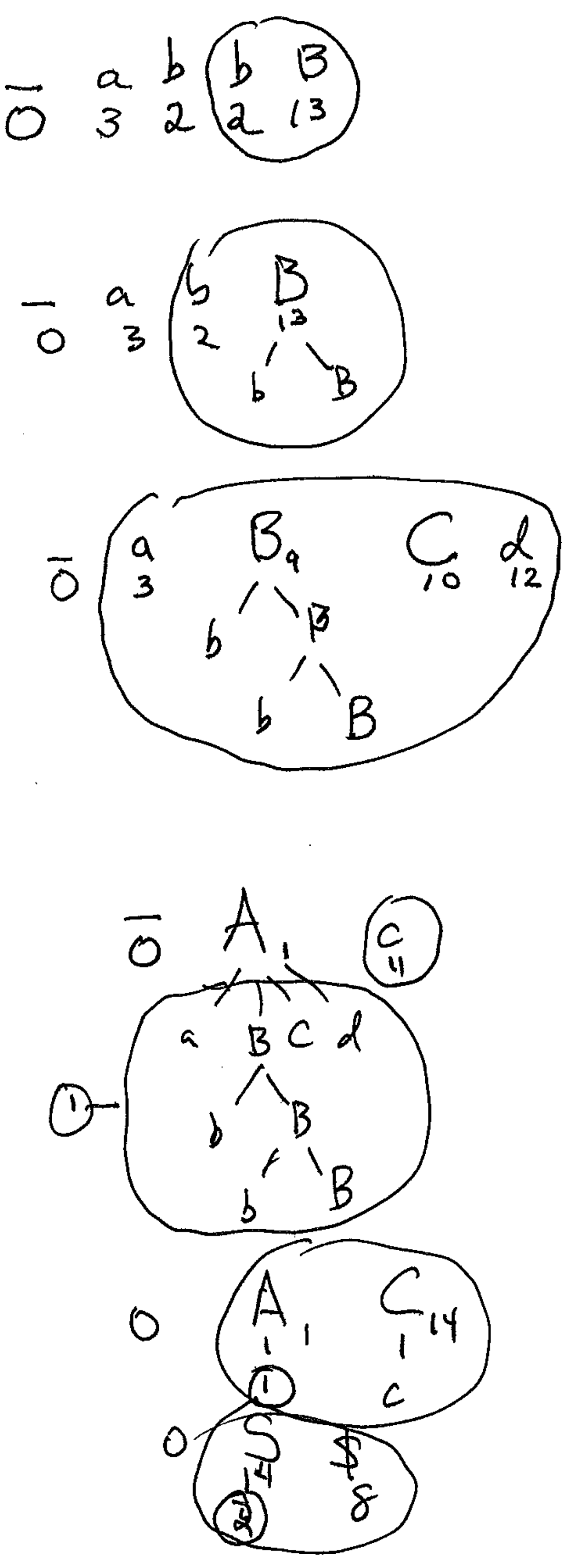
parse: $+\ n\ n\ \$$

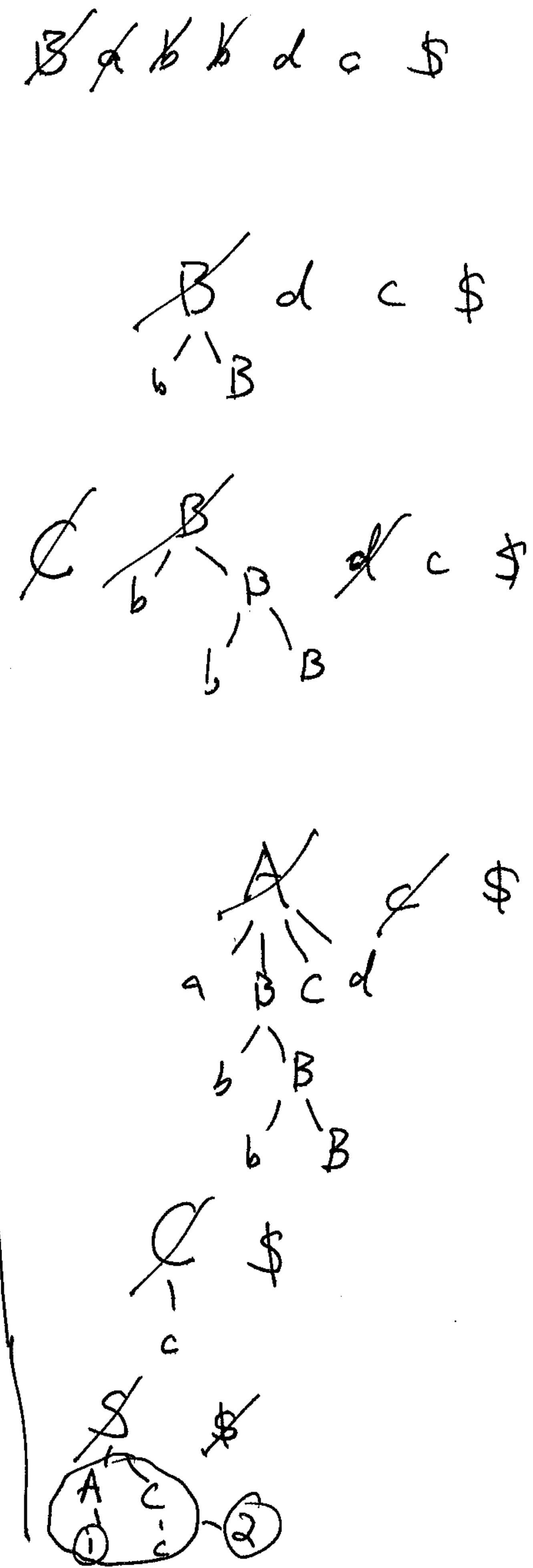
$$\overset{S}{\cancel{\$}\ \cancel{+}\ \cancel{n}\ \cancel{E}\ \cancel{n}\ \cancel{E}\ \cancel{E}\ \cancel{\$}}$$

$$\cancel{\$}\ \cancel{+}\ \cancel{n}\ \cancel{n}\ \cancel{\$}$$

$$\cancel{E}\ \cancel{E}\ \cancel{E}$$

$$S$$



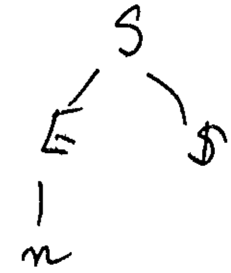


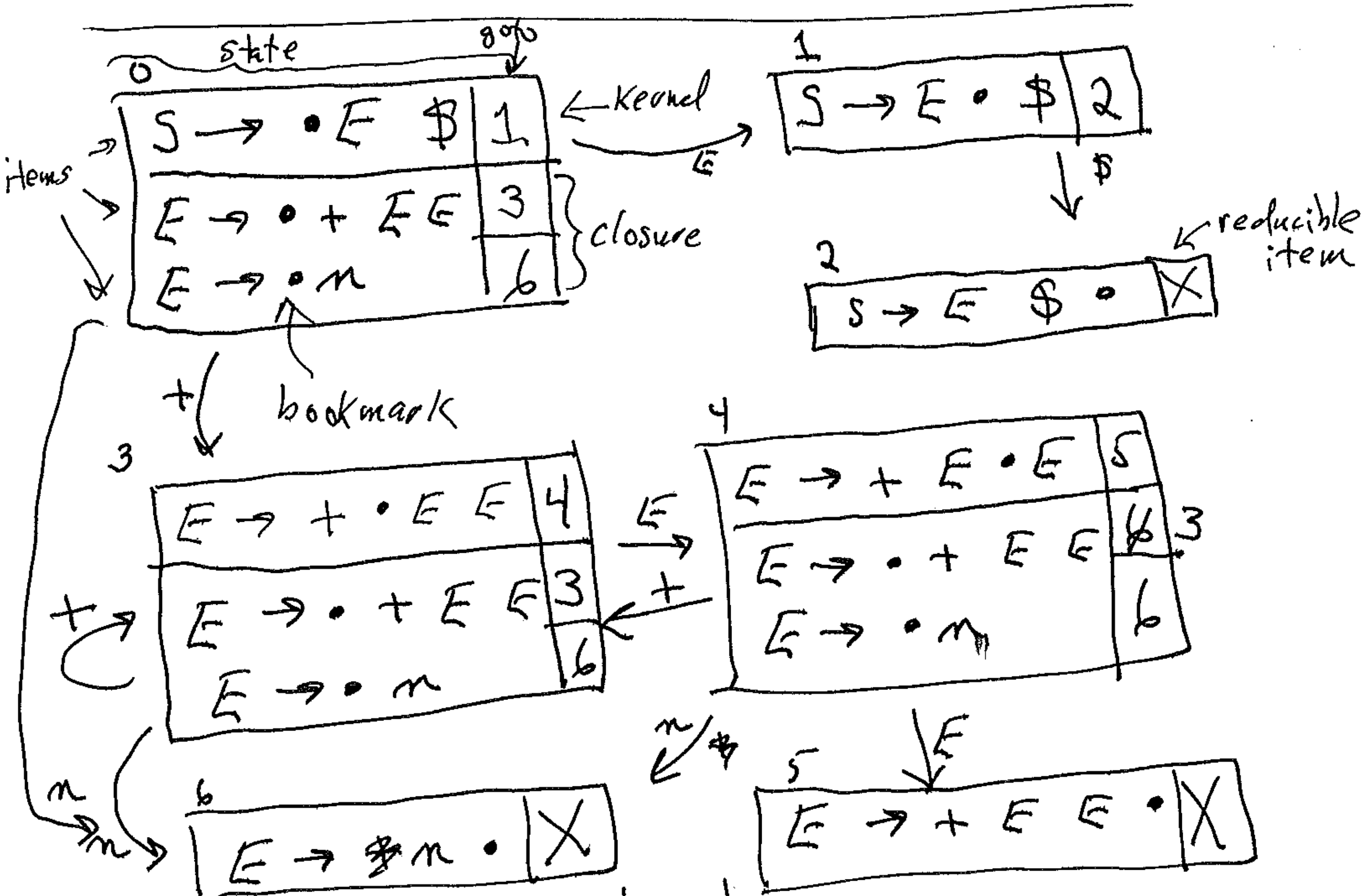
grammar (6.2 p 184)

1 $S \to E \; \$$

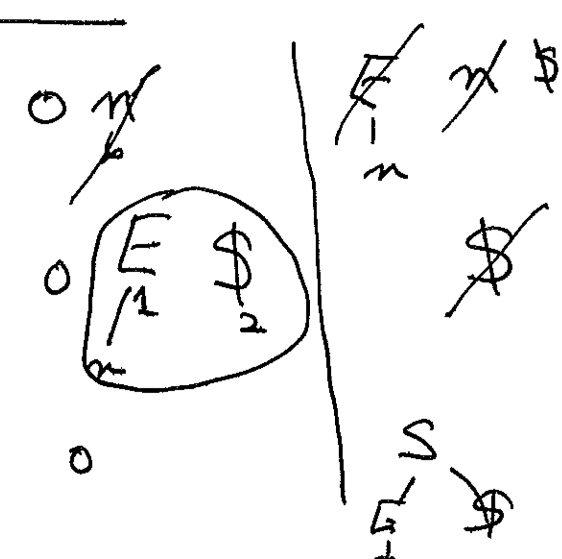
2 $E \to + \; E \; E$

3 $E \to n$





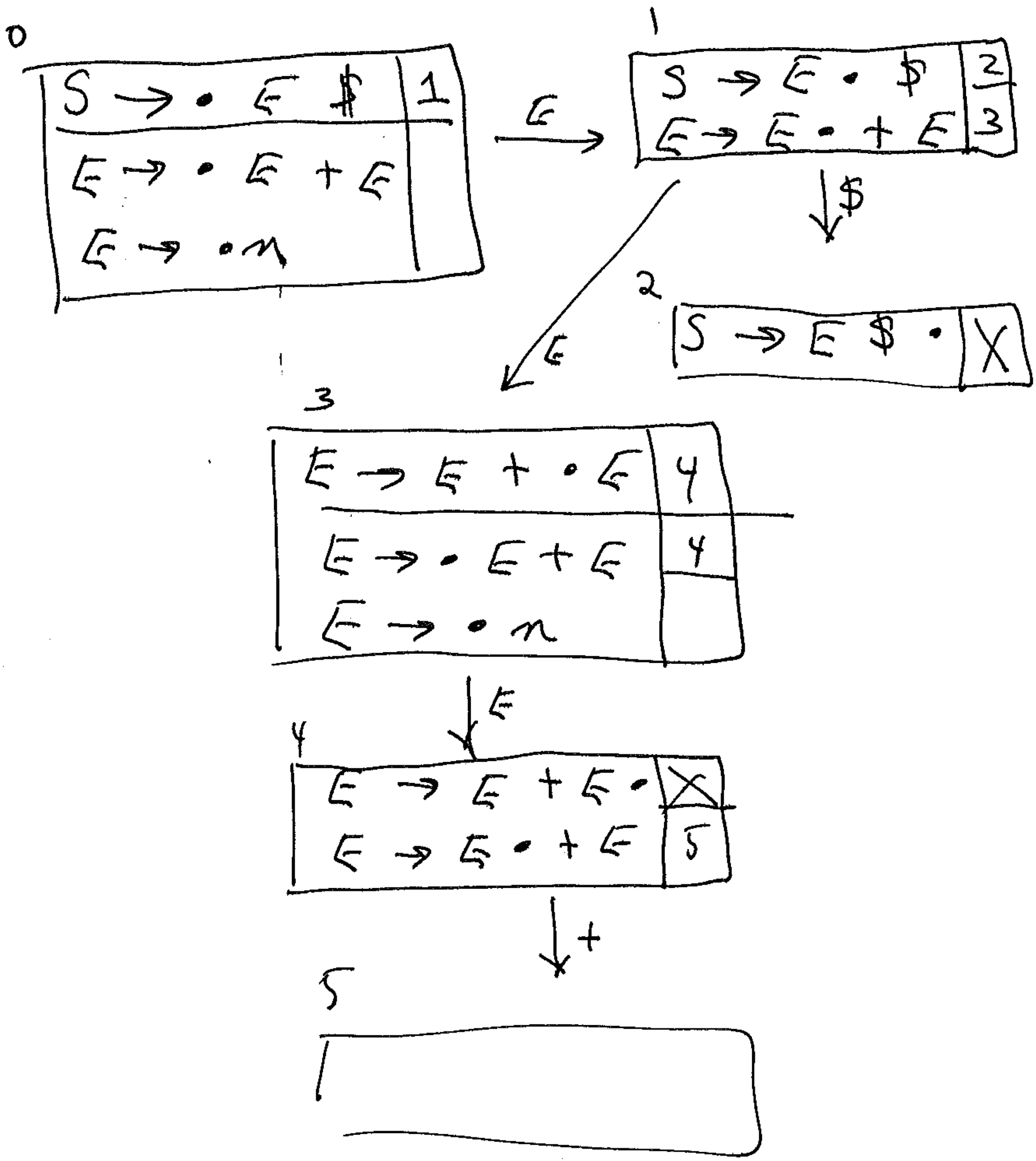
	+	n	$	S	E
0	$\boxed{3}$	$\boxed{6}$		accept	$\boxed{1}$
1	~~2~~	~~2~~	$\boxed{2}$	~~2~~	~~2~~
2	1	1	1	1	1
3	$\boxed{3}$	$\boxed{6}$			$\boxed{4}$
4	$\boxed{3}$	$\boxed{6}$			$\boxed{5}$
5	2	2	2	2	2
6	3	3	3	3	3



$S \to E\ \$$

$E \to E + E$

$E \to n$



Example 10g p. 227

51

1 St → S $
2 S → i = A ;
3 A → i = A
4 A → E
5 E → E + P
6 E → P
7 P → i
8 P → '(' i ; i ')'
9 P → "(" A ")"

0

St → • S $	1
S → • i = A ;	3

1

St → S • $	2

2

St → S $ •	X

$

3

S → i • = A ;	4

4

S → i = • A ;	5
A → • i = A	7
A → • E	10
E → • E + P	10
E → • P	
P → • (i ; i)	
P → • (A)	
P → • i	7

5

S → i = A • ;	6

6

S → i = A ; •	X

$

7

A → i • = A	8
P → i •	+) ;

8

A → i = • A	9
A → • i = A	7
A → • E	10
E → • E + P	10
E → • P	
P → • i	7
P → • (A)	
P → • (i ; i)	

9

A → i = A •	X

;)

10

A → E •	X
E → E • + P	11

;) +

11

E → E + • P	12
P → • i	
P → • (A)	
P → • (i ; i)	

12

E → E + P •	X

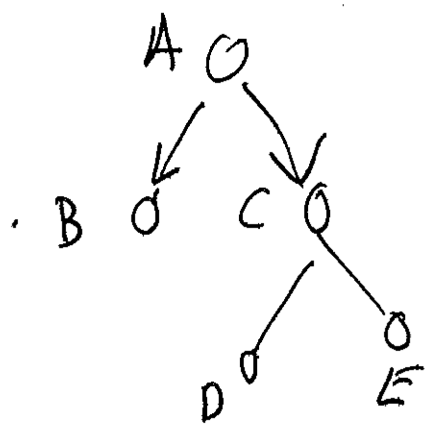
+ ;)

Syntax-directed translation

taking actions during parsing.

↑ syntactic actions

these actions may manipulate <u>semantic values</u> of symbols.



parse tree

information flowing downward: <u>inherited</u>.

flowing upward: <u>synthesized</u>.

Top-down parsing can pass information both ways.

Bottom-up parsing: synthesized only.

Example 1: decimal digits

$S_e \rightarrow Ds_f \ \$$

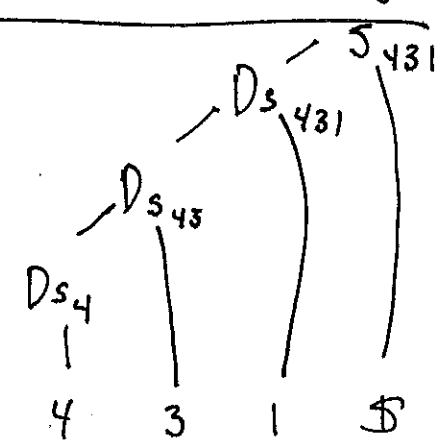
$\{e = f\}$

$Ds_a \rightarrow Ds_b \ d_c$

$\{a = b \cdot 10 + c\}$

$Ds_a \rightarrow d_c$

$\{a = c\}$



Example 2: decimal and octal digits

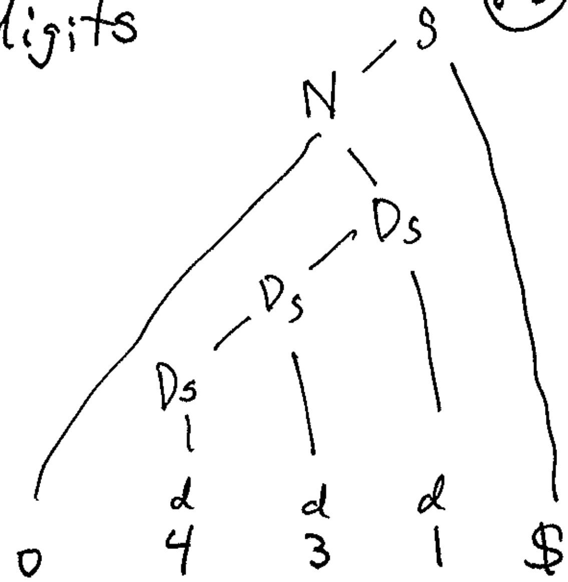
$S \rightarrow N \quad \$$

$N \rightarrow 0 \quad Ds$

$N \rightarrow Ds$

$Ds \rightarrow Ds \quad d$

$Ds \rightarrow d$



problem: cannot inform the semantic actions whether octal or decimal.

Example 3

rule cloning

$S_a \rightarrow N_b \quad \$ \qquad a = b$

$N_a \rightarrow 0 \quad Os_b \qquad a = b$

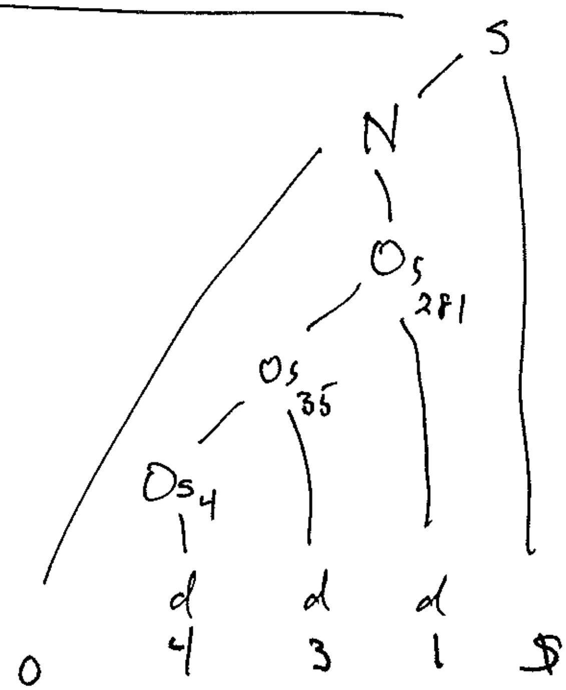
$N_a \rightarrow Ds_b \qquad a = b$

$Ds_a \rightarrow Ds_b \quad d_c \qquad a = b \cdot 10 + c$

$Ds_a \rightarrow d_c \qquad a = c$

$Os_a \rightarrow Os_b \quad d_c \qquad a = b \cdot 8 + c$

$Os_a \rightarrow d_c \qquad a = c$



Example 4: forcing semantic actions

$S_a \to N_b$
$a = b$

$N_a \to SigD \quad Ds_b$
$a = b$

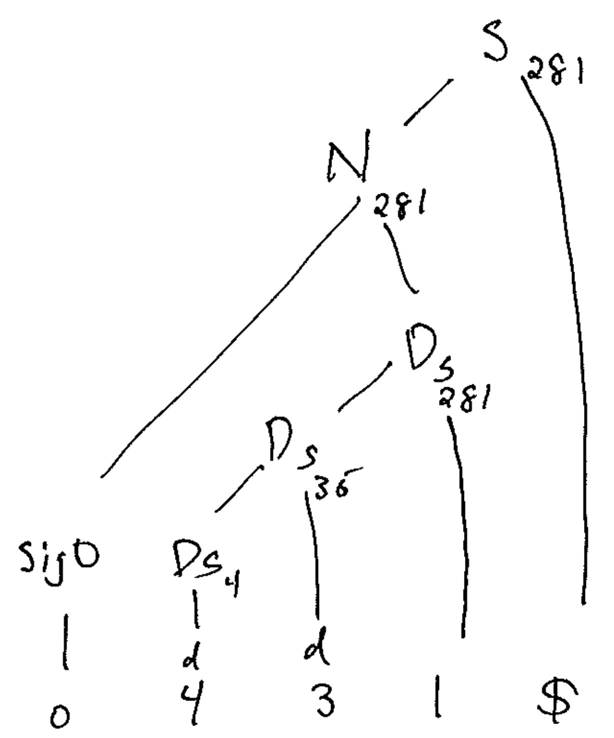
$N_a \to SigO \quad Ds_b$
$a = b$

$SigD \to \lambda$
$base = 10$

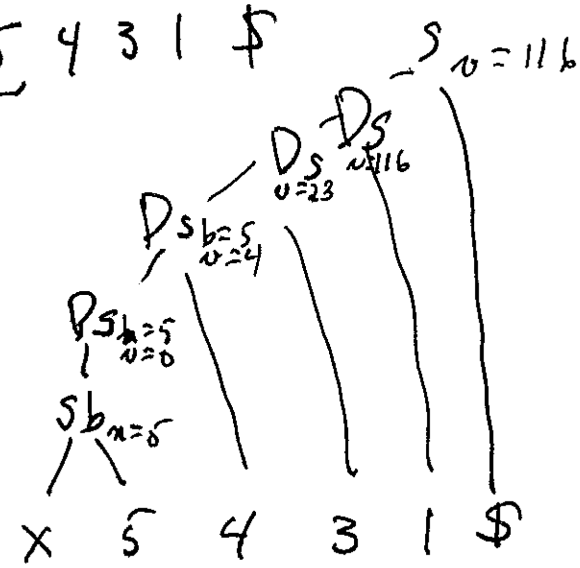
$SigO \to 0$
$base = 8$

$Ds_a \to Ds_b \quad d_c$
$a = b \cdot base + c$

$Ds_a \to d_c$
$a = c$



Generalize the lg: $\underline{x5}$ 4 3 1 \$



$S_{v'} \to Ds_{v,b} \quad \$$
$v' = v$

$Ds_{v',b'} \to Ds_{v,b} \quad d_c$
$b' = b$
$v' = v * b + c$

$Ds_{v,b} \to SetBase_n$
$v \leftarrow 0$
$b \leftarrow n$

$SetBase_n \to \lambda$
$n \leftarrow 10$

$\to x \quad d_c$
$\cancel{n \leftarrow c}$
$n \leftarrow c$

Enhance the lg to have any base

$x\,[\,3\;7\,]\;4\;2\;1\;\$$

also allow:

$x\,[\,x\,[\,8\,]\,2\;4\,]\;4\;2\;1\;\$$

left as an exercise.

Top-down syntax-directed translation

Expression grammar (like Lisp)

$S_{b'} \rightarrow V_b \; \$ \qquad b' = b$

$V_{b'} \rightarrow n \qquad b' = n$

$\cancel{\rightarrow (+ V \; V}$

$\rightarrow (\; E_b \;) \qquad b' = b$

$E_{b'} \rightarrow + \; V_a \; V_b \qquad b' = a + b$

$\rightarrow * \; Vs_b \qquad b' = b$

$Vs_{b'} \rightarrow V_a \; Vs_b \qquad b' = a * b$

$\rightarrow \lambda \qquad b' = 1$

Method resolution in Java

dynamic (at run time): based on value of the object

static (at compile time): based on class of the object

foo.m()

A: m(), f()
|
B: m(), g()

A foo = new B();

~~A.m()~~

foo.m()

need dynamic resolution

foo.f()

foo.g() $\boxed{\text{error}}$

A bar = new A();

~~A.m()~~

bar.m()

bar.g() $\boxed{\text{error}}$

X
|
Y

A: m(X), m(Y)
|
B: $\boxed{m(X)}$, m(Y)

X x = new Y();

A a = new B();

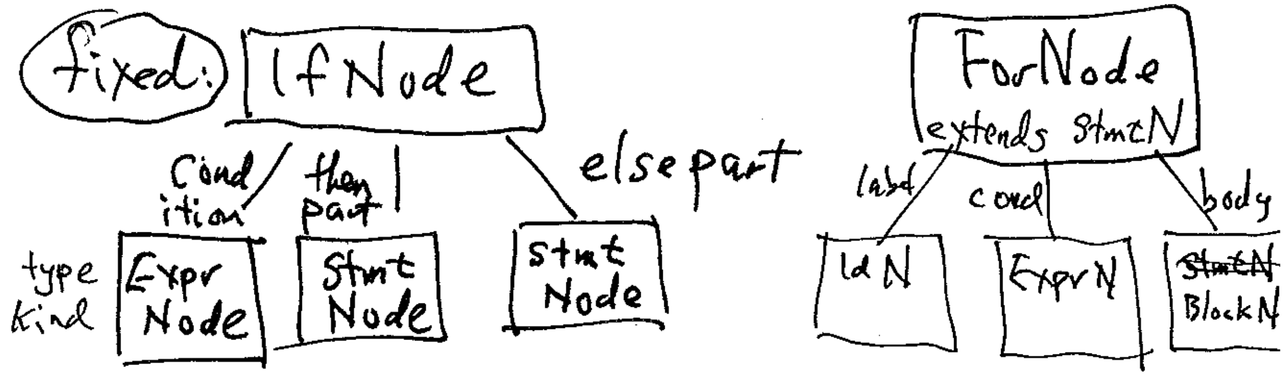
a.m(x); //prints B-X

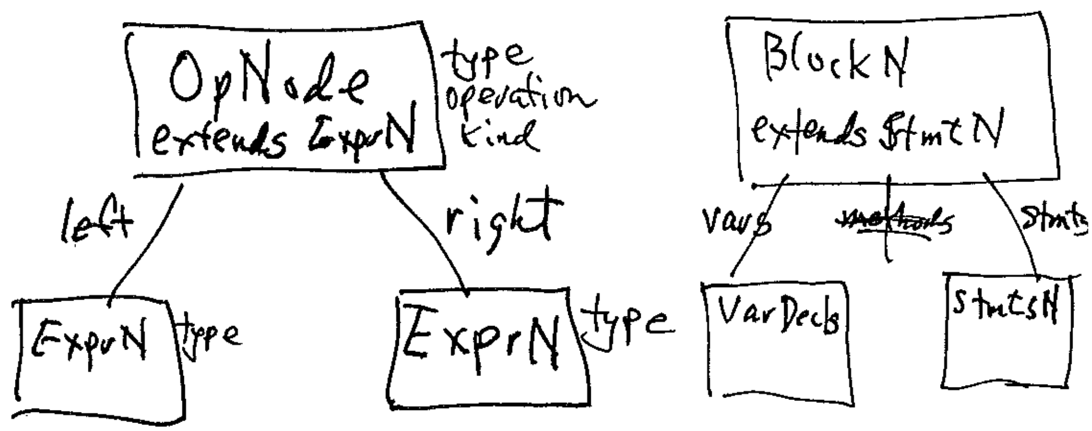
ASTs

distinguish: parse trees from ASTs.
concrete syntax trees

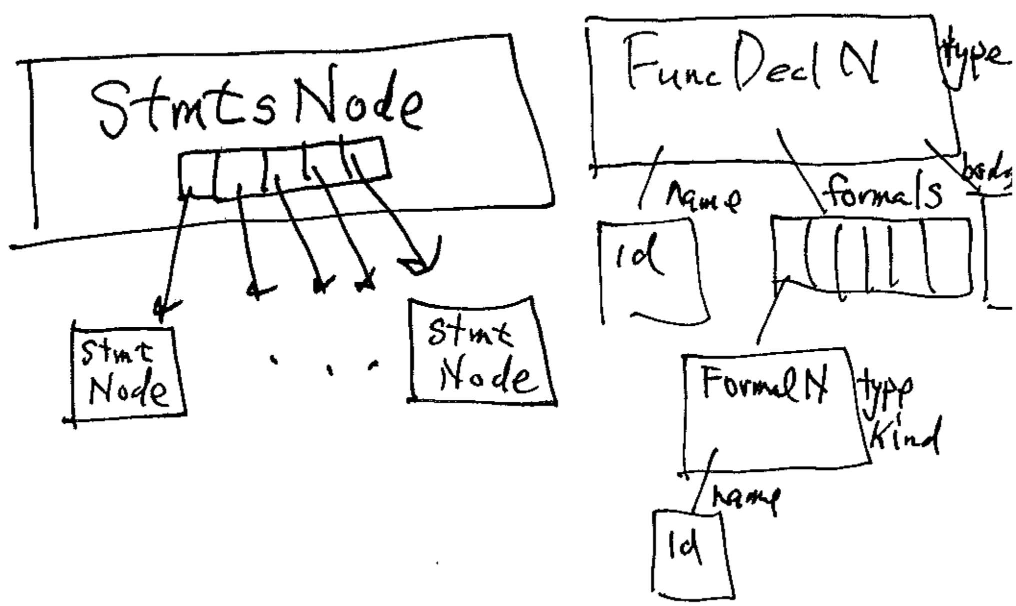
Number of children

fixed:





variable

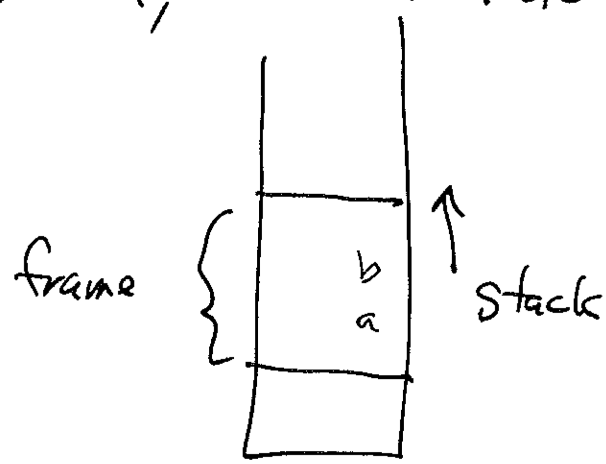


$a := b + 5;$

L-value
(location where variable is stored at runtime)

R-value (current arithmetic value)

example: a, b local ints



Facts:

constants (literals) have only R value.
<u>this</u> has only R value
in C, get an L value from an R value
with & (referencing)
in C, given an L ~~R~~ value, use
$*$ (dereferencing) to get its R value.

The Java Virtual Machine (JVM)

executes bytecode packaged in class files ("test.class")

bytecode designed to be compact.
    0-address: implicit stack

bytecode is safe: a verifier checks for type errors and stack errors.

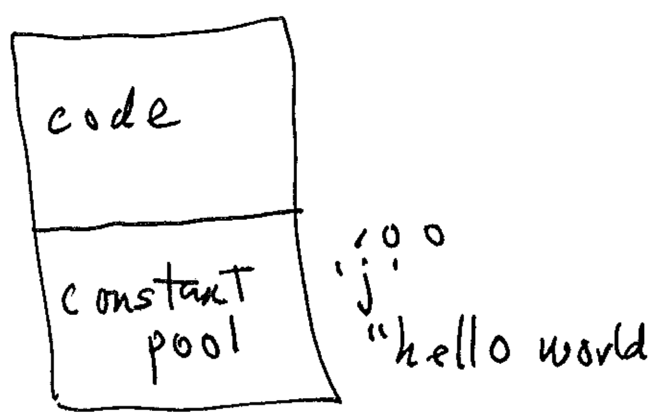
you can disassemble a class file.

    javap -c test.class

Types are described by short strings. p 400

Constants referred to in class files by an 16-bit index into "constant pool". Entries are typed.

Class file



Operations

Instructions in bytecode

arithmetic: pops 2 operands, does arithmetic, ~~places~~ pushes result.

iadd (integer addition)
fadd (float addition)
dmul (double multiplication)

Registers to hold local variables and parameters.

Numbered: 0, 1, 2 . . . for parameters, then local variables.

parameter 0 in dynamic methods: this.

registers are untyped.

Stack operations

register → stack

iload 39
fload
aload 14
reg 14 holds a reference. place the reference on stack.
shorthands
iload_3

stack → register
istore_4
↑ type ↑ register #

stack elements, registers are 32-bit values

long, double use 2 registers, 2 stack elements

$\downarrow$ even, odd

bool, char, byte, short: use 32 bits.

ldc : load a constant literal.

ldc ~~#~~ "hello"
ldc 'x'
ldc false
ldc 14

} translated by Jasmin to reference elements in constant pool.

type conversion

i2f

get/put a field of a class (~~inst~~ class → stack)

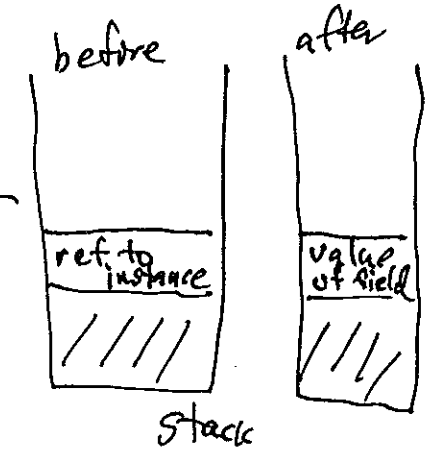
getstatic name type ~~class~~ → Ljava/io/PrintStream

$\downarrow$

java/lang/System/out

putstatic name type
(stack → class)

getfield
putfield



Calling a method

invokestatic name → java/lang/Math/pow(DD)D

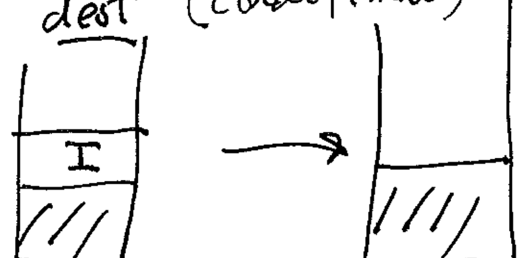
parameters on stack:
left-most deepest

invokevirtual name
push ref to instance
push parameters
call

invokespecial name (for constructors)
push ref to uninitialized object
(created by new)
push parameters

Branching (Jumping)

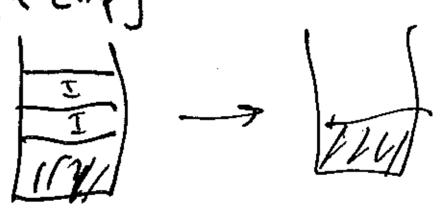
ifgt dest (conditional)



code:
inst
inst
inst
L: inst
inst

goto dest (unconditional)

if_ifcmpgt dest (conditional)



Stack operations

pop

swap : interchanges top 2 stack elements.

dup_x1 : duplicates the element at top of stack to a position 3 below the top.

<u>while</u> Bool {

body

}

```
L2: code for Bool
    conditional branch if false to L1
    code for Body
    go to L2
L1:
```

better code

```
    jump L3
L4: code for Body
L3: code for Bool
    conditional branch if true to ~~L3~~ L4
```

```
if B {              |  code for Bool
    S1              |  conditional branch to L1 (if false)
} else {            |  code for S1
    S2              |  unconditional branch to L2
}                   |  L1: code for S2
                    |  L2:
```

```
func foo(){
    var a;  reg 0
    var b;  reg 1
    if B {
        var c;  reg 2
    } else {
        var d;  reg 3
    }
}
```

Keep track of reg.
Sequence # as ~~an~~ a hidden ST entry.

Chapter 8: Symbol tables

First used in semantics checker.

Can check for reserved words using ST during scanning.

C: typedef.

Handling nested scopes.

Stack of hash tables

push : open a scope
pop : close a scope

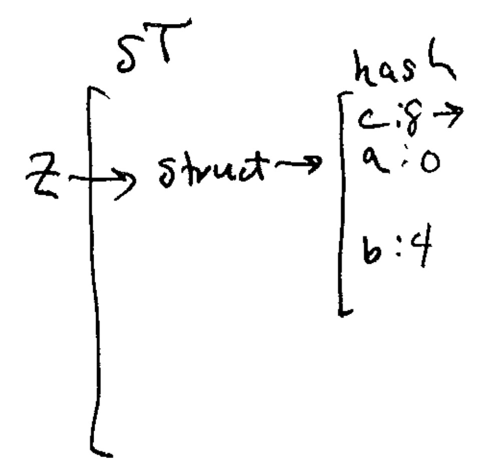
semantics checker
code generator

possible to build a single unified hash table.

How to treat structs.

represent a struct type in ST as a hash table.

```
struct {
    a: int;
    b: float;
    c: struct {
        d: int;
    }
} z
```



Overloading

- field (instance variable in class)
  no overloading, only hiding.
- methods
  foo (a: int)
  foo (b: bool)
  1) could store in ST : Key = "foo, int"
  "foo, bool"
  but to search at the invocation point is very expensive.
  2) store with key "foo"
  data: list of all methods with that name.

  resolution
  to search at invocation:
  get the list of possibilities from the closest scope where the list is not empty.
  Select the "best" of those possibilities.
- operators
  put definitions in ST.
- enumeration constants (WED) : list in ST

```
var
   a,b: record
          x,y: int
        end;
begin
   .
   a.x := 7;
   with a do
      x := 1;
      y := y+1;
   end;
```

Pascal with
for efficiency
and clarity

To handle:
Open a scope when
encounter with.

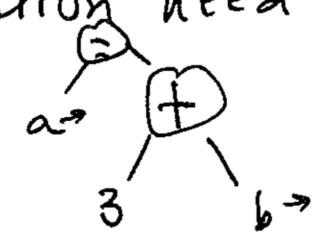
Processing declarations (semantics checker)

Semantics checker: ST should point the identifier's entry to
a type descriptor: type (int / char) kind (array?) reg#

Every access in the AST to that identifier
points to that same descriptor.

Therefore, code generation need not use ST.

a = 3 + b;



Particular AST nodes.

IfNode
ForNode
Labelled node (For Node): sem.check: unique, put in ST
code gen: save 2 labels in

Jasmin

Labelled for loop.

Semantics checking

1) ensure label is not in ST globally.
2) place label in ST. (avoid dup)
3) open new scope, put label in new scope, too. (to store code locations)
4) every break, continue: check ST for existence of label. point the AST to the label information.

Code generation

1) generate Jasmin Labels (JLs) for the loop (2 or 3)
2) Store "start", "end" labels in the label information.
3) generate code
4) every break: code goto JL
   end
5) every continue:
   code goto JL
   start

Booleans and generating code for JVM

Represent as 0 (false) 1 (true)

```
ldc 1
iconst_1
```

Handle $a < b$

```
code to evaluate a (maybe simple: iload's)
code to evaluate b
if_icmplt   L1
```

integer

<

```
     iconst_0
     goto  L2
L1:  iconst_1
L2:
```

Handle $b1 = b2$

```
code for b1
code for b2
if_icmpeq
```

similar

Handle b1 and b2

```
code for b1
code for b2
iand
```

short-circuit?

```
     code for b1
     ifeq L1
     code for b2
     goto L2
L1:  iconst_0
L2:
```

```
class foo {
    public static int a, b;

    static void bar() {
        a = b
    }
}
```

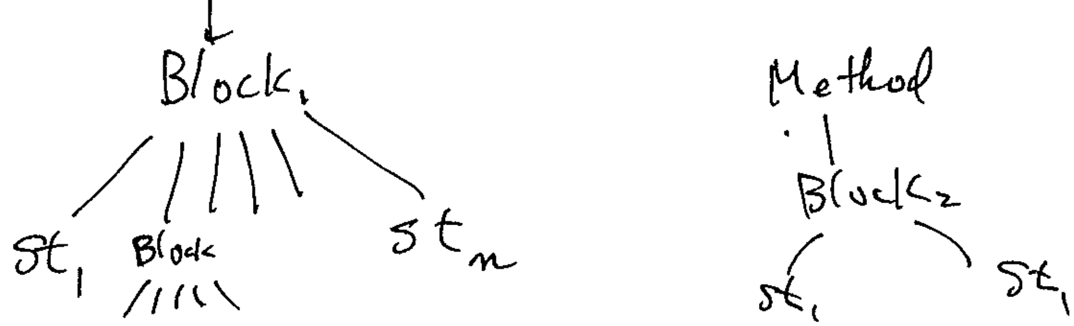
```
.class foo
.field static a I
.field static b I
.method static bar()V
.local
.limit locals 1
.limit stack 10
  getstatic foo/b I
  putstatic foo/a I
.end method

.method static main
    ([Ljava/lang/String;)V
  putstatic for each const

  invokestatic foo/main()V
  return
.end method
```

Other semantic checks

1. type checking
2. reachability checking
3. exception checking

---

(2) invalid to have an unreachable statement.
each statement node in AST has
bool { isReachable
terminatesNormally (not break, continue, return, throw)



st$_1$ inherits isReachable, Block$_1$ synthesizes terminates from st$_n$.
Block$_2$ is marked isReachable.
declarations terminate normally.

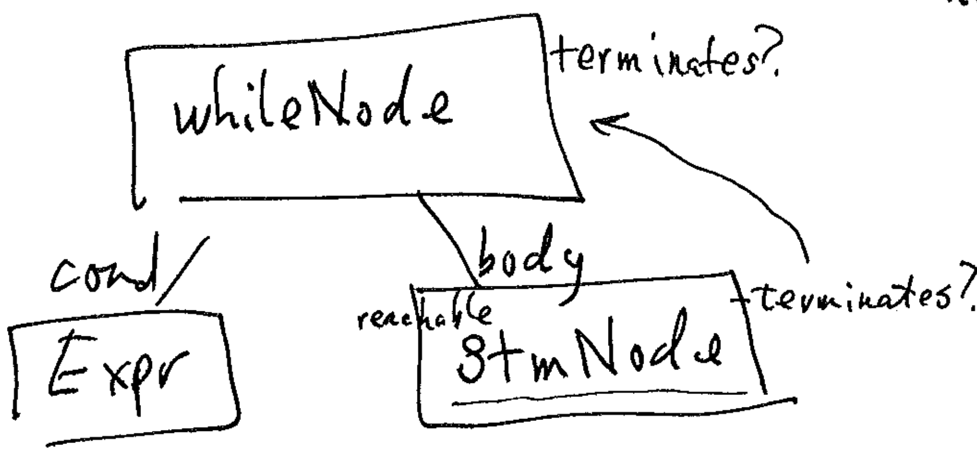
B's isReachable is derived from A's terminates

return, break, continue, throw do not terminate normally

if: cond / then — else
maybe start then, else part's is Reachable.
resulting tN is OR of tN of the children.

<u>while</u> B <u>do</u> S <u>end</u>

assume S is reachable
if S terminates normally, so does <u>while</u>
otherwise <u>while</u> does not terminate normally.



---

switch: tN is the <u>OR</u> of the tNs of all cases.

---

(3) Semantic check in Java for checked exceptions.

rules:

any method that can exit with exception must say so.

a <u>catch</u> may only refer to an exception that the <u>try</u> can throw.

multiple <u>catch</u> clauses on a <u>try</u> each may only catch an exception still possible.

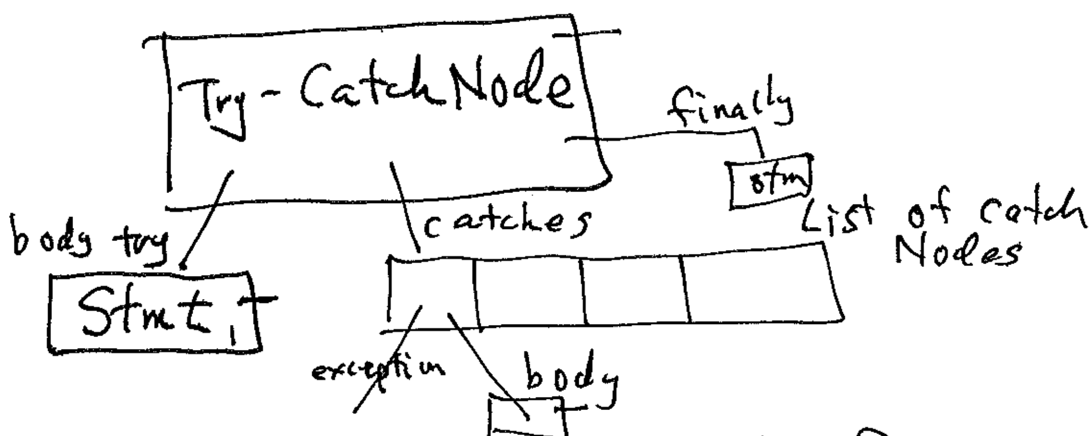
To implement 3, checker for Stmts must generate a list of exceptions the Stm can throw.
set

Block Node can throw any exception that its internal statements can throw.

IfNode: inherits all exceptions that condition, then-part, else-part can throw.

FcnCall Node: can throw all exceptions listed in the FcnDecl Node.

Try-catch Node



Keep track of possible Exception Set $= \rho$

initially, ρ = Stmt$_1$.exceptionSet;

for each catch Node

  verify the exception is in ρ

  remove that exception from ρ

  add into ρ all exceptions from body

add in the exceptionSet of the finally clause

place result as the exceptionSet of TryCatchNo

DeclList → DeclList ; Decl
→ Decl

before

DeclList → X Y
X → Decl
Y → ; Decl Y
Y → λ

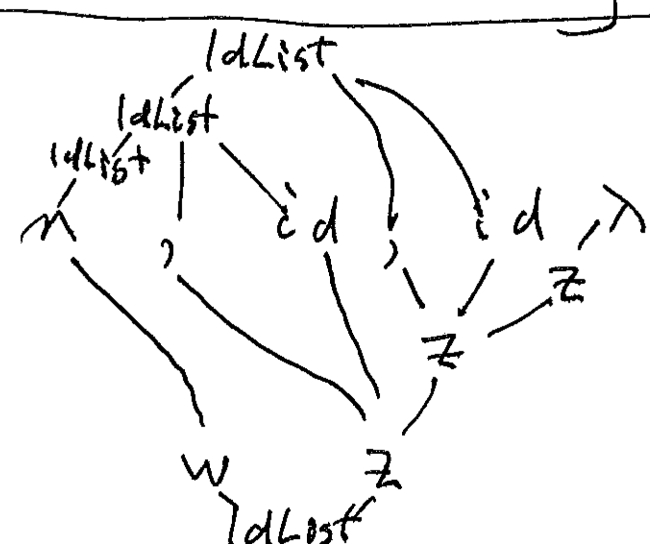
after

IdList → IdList , id
→ id
→ n

before

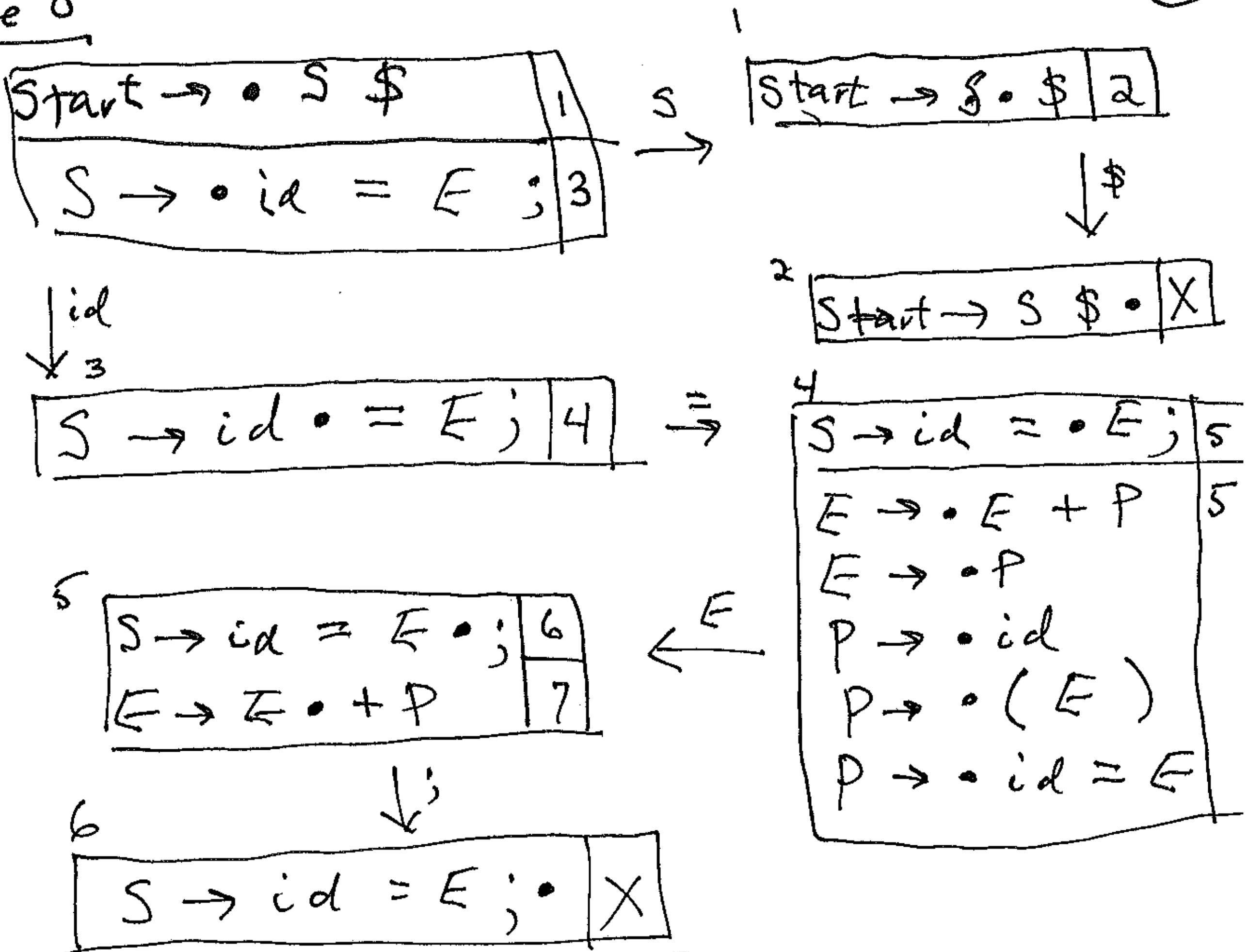
IdList → W Z
W → id
W → n
Z → , id Z
Z → λ

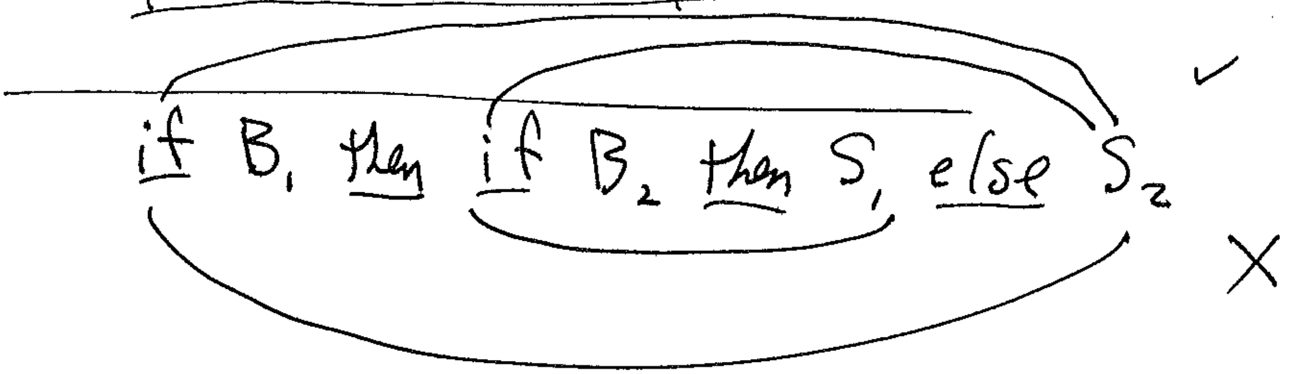
after



before

State 0


Start → • S $ | 1
S → • id = E ; | 3
S
1
Start → S • $ | 2
$
2
Start → S $ • | X
id
3
S → id • = E ; | 4
=
4
S → id = • E ; | 5
E → • E + P | 5
E → • P
P → • id
P → • (E)
P → • id = E
E
5
S → id = E • ; | 6
E → E • + P | 7
;
6
S → id = E ; • | X


if B_1 then if B_2 then S_1 else S_2
✓
X