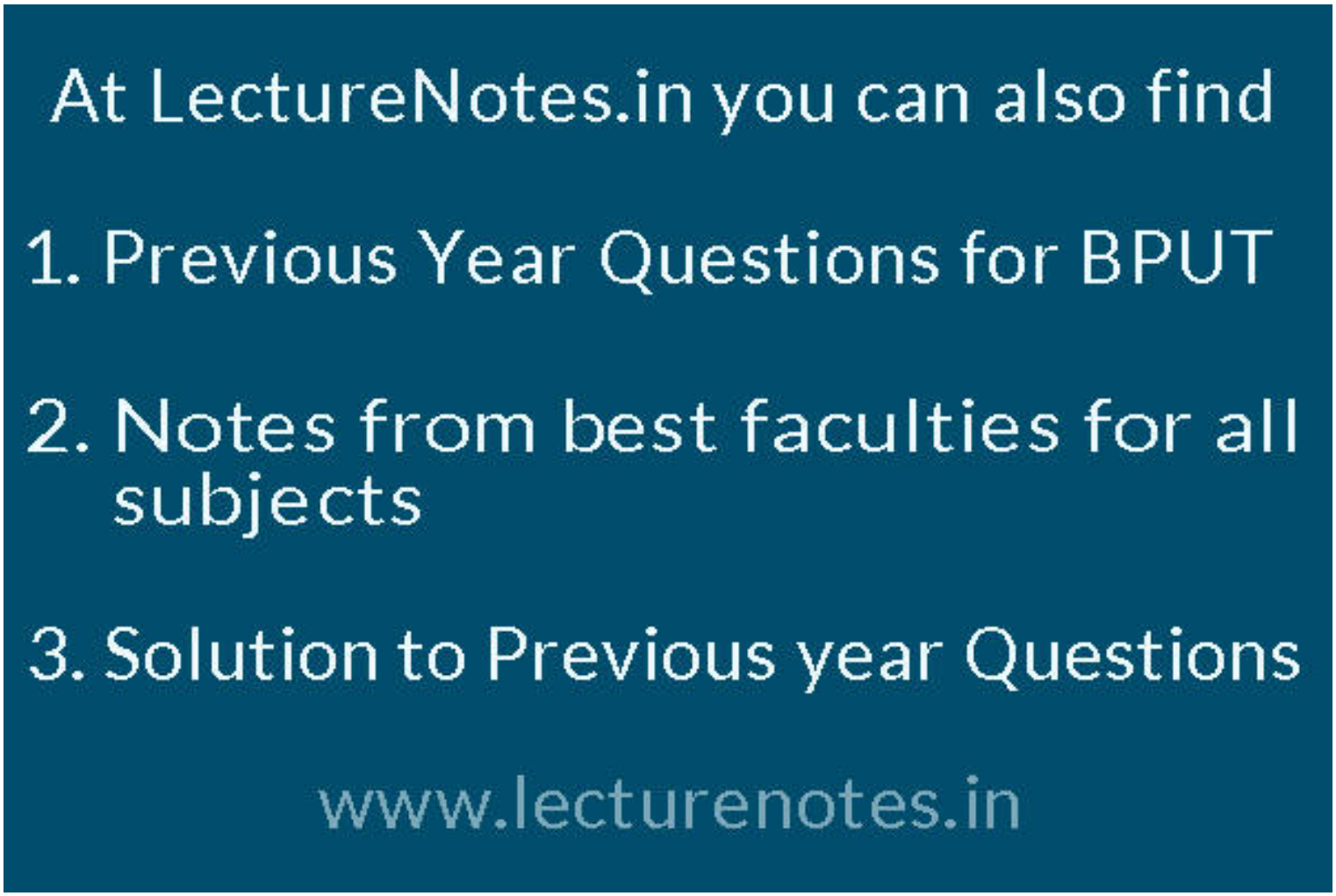
Compiler Design

This download is Sponsored by:

Contributed By:
Sibananda Achari
Trident Academy of Technology

Disclaimer



Compiler Design

Topic:

Introduction To Compiler Design

Contributed By:

Sibananda Achari

Trident Academy of Technology

→ Introduction to compilers 9/12/13

Introduction to Compilers

→ Need of compiler

→ Structure of compiler

→ Tools for compiler writing

Types of compiler —

→ one-pass compiler:

- It passes through the source code of each compilation unit (phase) only once

→ Multi-pass compiler:

- Processes the source code of a program several times.
- Multipass compilers are sometimes called wide compilers

→ Load and go compiler:

- On compiling technique in which there is no stop between loading and execution.

→ Optimising compiler:

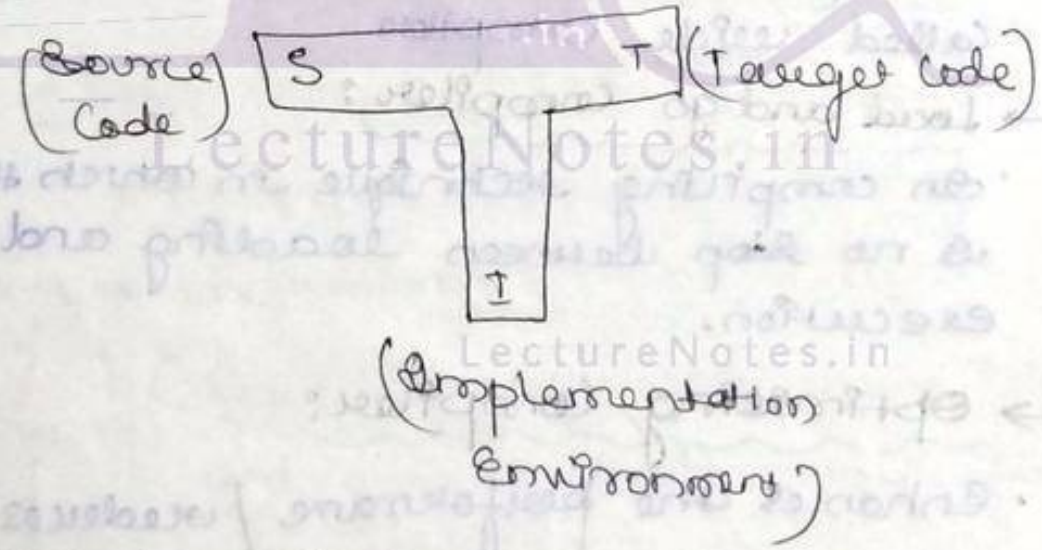
- Enhances the performance / reduces the size of the resulting machine program. (optimizes the program code).
- They require multiple passes in order to analyse the entire program and maximise the use of code throughout.

- Can be achieved by using simple instructions that can be executed fast.

→ Native Compiler -

- Compiles programs for the same architecture of OS that it is running on.
- Platform is a combination of CPU architecture and OS.
- Compiles programs that can be executed on same platform.

T-diagram -



→ Cross Compiler -

- Executes in an environment and generates code for another.

Eg - Microsoft Visual Studio includes a native compiler.

→ Embedded devices such as mobile phones, washing machines uses cross compiler. (11)

→ Semantics Analysis

→ This phase checks the program for semantics error and gathers type information for the successive phases.

Eg - Using a variable that is not declared is checked during semantics analysis.

• Type Checking -

→ Check types of operands (possibly imposing type conversions); No real no. for index out of bounds array etc.

→ CFG cannot be used to specify semantics rules. At the end of semantics analysis, syntax directed output is obtained.

→ Normally, semantic information cannot be represented by CFL used in syntax analysis. CFG used in Syntax Analysis are integrated with attributes (semantic rules). The result is, syntax directed translation and attribute grammar.

• Synthesis Phase -

→ Starts with intermediate code generation.

→ Intermediate Code Generation

→ Intermediate code is machine independent.

→ An intermediate code is generated as a program for an abstract machine.

(V)

→ The intermediate code should be easy to translate into the target program.

→ As intermediate code we consider the three address code, similar to assembly sequence of instructions with atmost three operands such that:

→ There is atmost one operator, in addition to the assignment. Thus, we make explicit the operators precedence.

→ Temporary names must be generated to compute intermediate operations.

Eg-

```
if (a<=b)
{
    a=a-c;
}
c=b*c;
```

3-Address Code

```
-t1 = a>b
if -t1 goto L0
-t2 = a-c;
a = -t2
L0: -t3 = b*c;
    c = -t3;
```

→ Code optimisation

- optimising the space consumed by variables and program.

→ Different compilers have different optimisation techniques.

→ Code Generation

→ Generates the target code consisting of assembly code.

- Memory locations are selected for each variable.

- Instructions are translated into a sequence of assembly instructions.
- Variables and intermediate results are assigned to memory registers.

Error Handling

→ lexical analyser reports invalid characters.
→ Syntax analyser reports invalid token sequences
→ Semantics analyser reports invalid scope errors.

Compiler Construction Tools

These are :

→ Scanner generators
→ Parser generators
→ Syntax directed translation engines
→ Code generator generator
→ Data flow analysis engines

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Lexical Analysis, Finite Automata

Contributed By:

Sibananda Achari

Trident Academy of Technology

Lexical Analysis

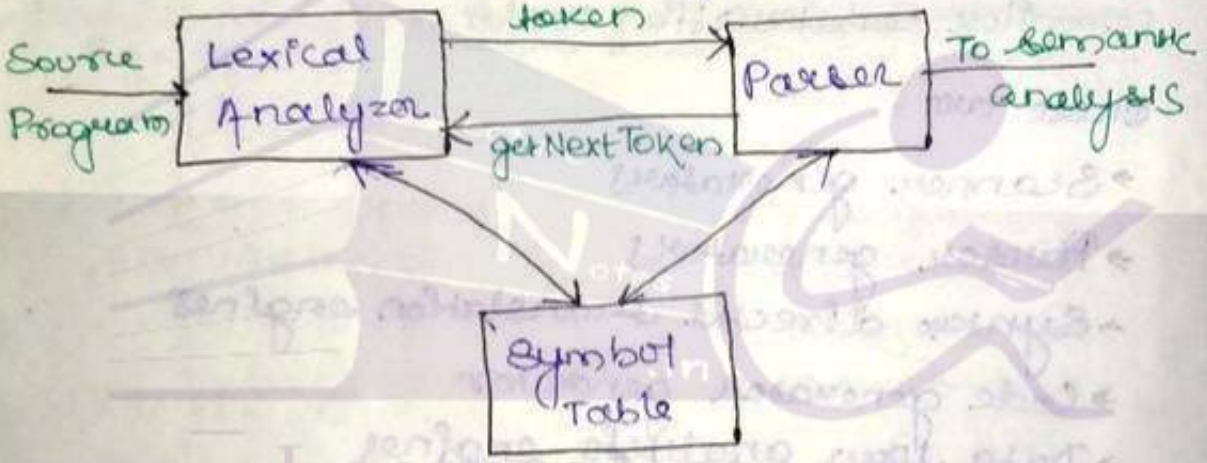
The role of lexical analyser

→ The first phase of the compiler.

→ Reads the input characters and produces as output a sequence of tokens that the parser uses for syntax analysis.

→ Strips out source program comments & white space in form Stripping out from the of blank, tab & line.

→ Correlates error messages from the compiler with the source program.

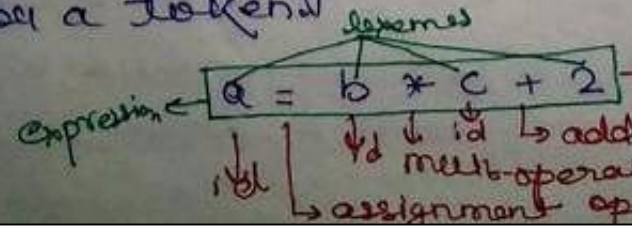


Tokens, Patterns and Lexemes

→ A token is a pair a token name and an optional token value.

→ A pattern is a description of the form that the lexemes of a token may take.

→ A lexeme is a sequence of characters in the source program that matches the pattern for a token.



Symbol Table -

→ Matrix like structure having rows and columns.

→ Scanning the entire strings from left to right.

→ Here,

a

<id, 1>

→ 1 denotes pointer to the symbol table entry for a.

→

1	a	- - -
2	b	- - -
3	c	- - -

→ For operators we are not maintaing a pointer to the symbol table.

→ For identifiers we are maintaining a pointer to the symbol table

→ let if (expr)
---
else
...

So, the regular expression for above is -

0 → token

[S → (if) expr statement | (if) (expr) statement (else) stmt. | ∈]

If S → if expr stmt | ∈ , then compiler will generate error as rule is not specified in the regular expression.

→ Literal means a sequence of characters enclosed in double quotes.

Example

Token	Pattern	Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
Comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letter and digits	pi, score, D2
number	Any numeric Constant	3.1415, 0, 6.02e23
literal	Anything but "surrounded by"	"core dumped"

Attributes for tokens

→ Apart from token itself lexical analyzer passes other information regarding token, which is called attributes.

→ It helps in compilation when more than one pattern matches a lexeme

→ Eg: E = M * C * * 2

<id, pointer to ST entry for E>

<assign-op>

<id, pointer to ST for M>
<mult-op>
<id, pointer to ST entry for C>
<exp-op>
<number, integer value 2>

(ch-3 Pg-110-129)

Input Buffering

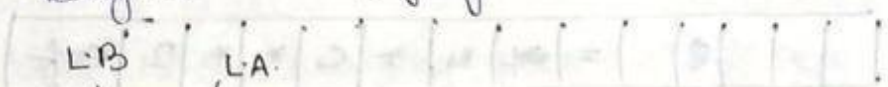
→ It is desirable for the lexical analyzer to read its input from an input buffer because many characters beyond the next token may have to be examined before the next token itself can be determined.

→ 2 pointers are used in the I/P buffer.

* lexeme Beginning is a pointer marks the beginning of the token being discovered.

* look ahead pointer scans ahead of the beginning point, until the token is discovered.

lexeme Beginning
look ahead pointer
Begin L = L+1 ; J = J+1 ...



Begin L = L+1 ; J = J+1 ...

Begin L = L+1 ; J = J+1 ...

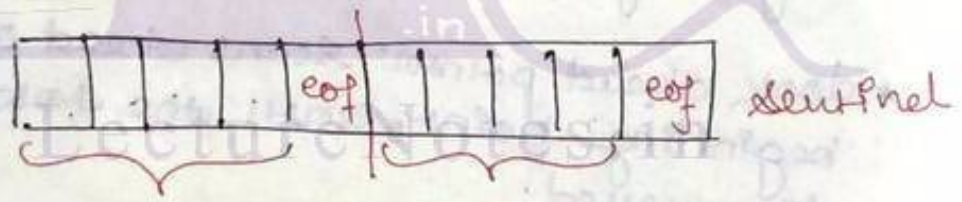
There are different buffering scheme which are used commonly compilers.

→ One buffer scheme -

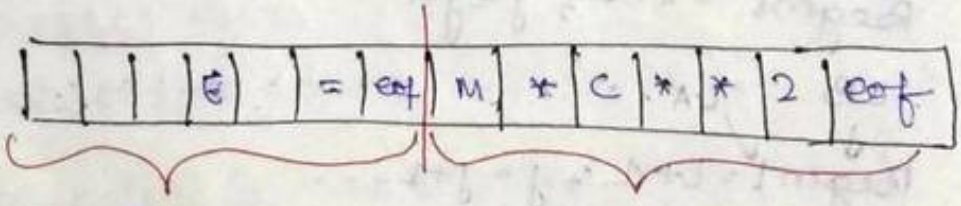
There are problems if a lexeme crosses the buffer boundary. To scan the rest of lexeme, the buffer has to be refilled thereby overwriting the first part of the lexeme.

→ Two buffer scheme -

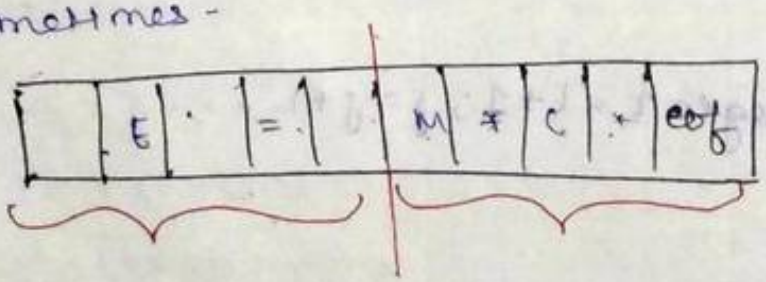
Here, buffers 1 and 2 are scanned alternatively. When the end of the current buffer is encountered the other buffer is filled. Hence, the problem encountered in previous scheme is solved.



Let $E = M * C * * 2$



Sometimes -



Lexical Error

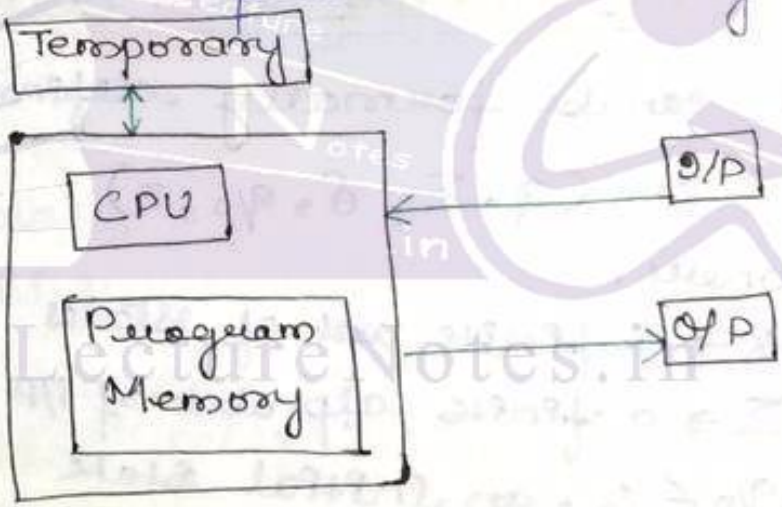
Some errors are out of power of lexical analyzer to recognize.

Lexical Analysis

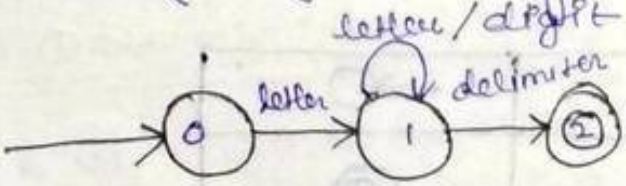
- Finite State Machine
- Regular Expression
- Lexical Analyzer

→ Automaton - It is a machine responsible for processing the input and generating output.

It represents CPU + Program memory



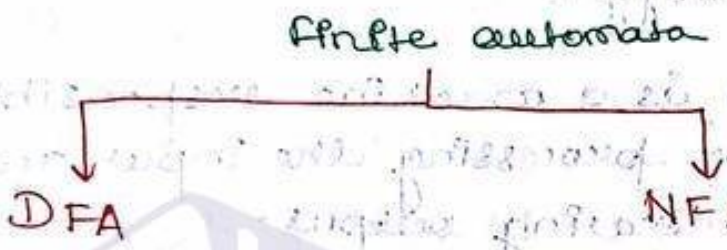
→ In finite state automaton, temporary memory is absent. Then the structure will look like Turing machine & PDA.



→ An identifier can be defined as a string of letters and digits that begins with a letter.

→ State Automaton: An automatic transition from one state to another state or set of state on an input.

→ Finite State Automation: Due to finite no. of state



① Deterministic Finite State automata

→ A DFA can be formally defined as

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

where,

i) Q → a finite set of states

ii) Σ → a finite alphabet of I/P symbols

iii) $q_0 \in Q$, an initial state

iv) $F \subseteq Q$ → a set of final states

v) $\delta : (Q \times \Sigma) \rightarrow Q$ → a transition function.

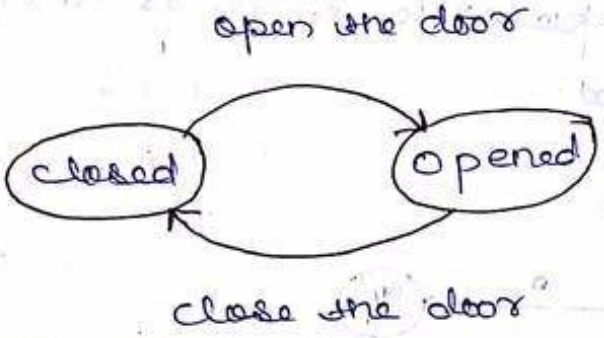
→

Initial State	→○
Final State	◎
Other State	○
Transition / move	→

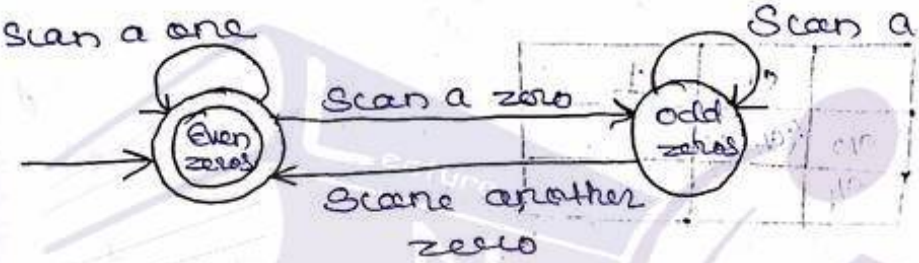
Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

→ the movement will be from left to right. (2)
No temporary m/m but program memory having I/P tape.

→ Eg:



→ Eg: Check whether a binary no. has even no. of zeros or not.



→ Generally DFA this is called a machine M which accept a language.

→ The language is the set of string the machine accept. which is denoted as L(M).

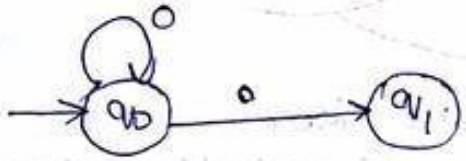
→ L(M) = { Binary string having even no. of zeros }

Non-Deterministic Finite State Automata

→ NFA is a finite state machine where from each state and a given input symbol the automaton may jump into several possible next states.

→ This distinguishes it from the DFA, where the next possible state is uniquely determined.

→ Eg-



→ Here, $\delta = Q \times \Sigma$

→ State Table

	0	1
q_0	$\{q_0, q_1\}$	
q_1		

→ A NFA is represented formally by a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$ consisting of where,

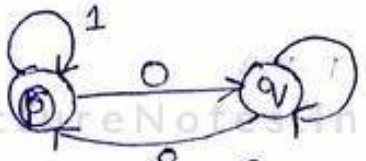
- a finite set of states Q
- a finite set of i/p symbols Σ
- a transition relation

$$\Delta : Q \times \Sigma \to P(Q)$$

- an initial (or start) state $q_0 \in Q$
- a set of states F distinguished as accepting or final states $P \subseteq Q$.

→ Transition function takes 2 input.
- an input tape
- current state

→ Eg- $\delta(P,0) = \{q\}$, $\Sigma = \{0,1\}$



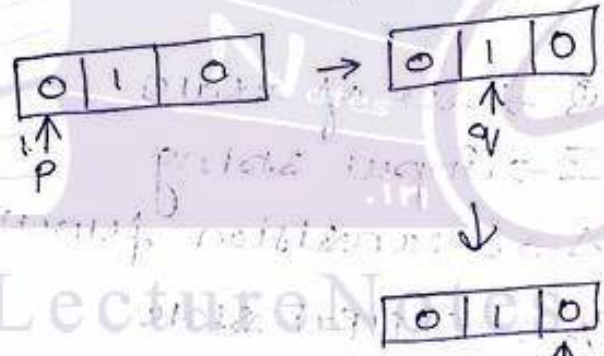
$\delta(P,1) = \{P\}$

$\delta(q,0) = \{P\}$

$\delta(q,1) = \{q\}$

To generate a language for the automata

def



A language is a set of string accepted by the finite automata.

∈-NFA (∈-Transition of a NFA)

→ allows us to make a transition on the empty string.

→ Spontaneously make a transition without receiving an i/p symbol

→ allows a transformation to a new state without consuming any input symbols. (23)

DFA	∈ - NFA
$\delta(q_i, \in) = \{q_i\}$	$\delta(q_i, \in) = \{q_j\}$ $q_i, q_j \in Q$ $q_i \neq q_j$

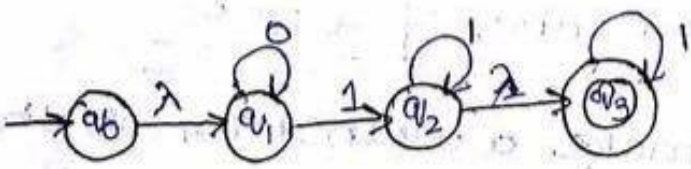
$$\in\text{-NFA} \rightarrow \text{NFA} \rightarrow \text{DFA}$$

→ An NFA-∈ is represented formally by a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$ consisting of

where,

$Q \rightarrow$ set of states
$\Sigma \rightarrow$ input string
$\Delta \rightarrow$ transition function
$q_0 \rightarrow$ initial state
$F \rightarrow$ final state

Conversion of ∈-NFA to NFA



Here, $M = \{(q_0, q_1, q_2, q_3), \{0, 1, \in\}, \delta, q_0, \{q_3\}\}$

M' = NFA without ∧ moves or ∈-transitions

Step I : To find out the ∈ closure of the given automata-

∈closure $(q_0) = \{q_0\} \cup \{q_1\} = \{q_0, q_1\}$

∈-closure $(q_1) = \{q_1\}$

∈closure $(q_2) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$

∈closure $(q_3) = \{q_3\}$

Here, $M' = \{\{\{q_0, q_1\}, \{q_1\}, \{q_2, q_3\}, \{q_3\}\}, \{0, 1\}, \delta', \{q_0, q_1\}, \{\{q_2, q_3\}, \{q_3\}\}\}$

∈-NFA or NFA to DFA Conversion

For each NFA we can find an equivalent DFA which accepts the same language.

Subset construction algorithm

initially, ∈closure (q_0) is the only state in Dstates, and it is unmarked;
while (there is an unmarked state T in Dstates)
{
  mark T;
  for (each i/p symbol a)
  {
    U = e-closure (move (T, a));
    if (U is not in Dstates)
      add U as an unmarked state to Dstates;
    Dtran [T, a] = U;
  }
}

Push all states to stack.

Conversion of NFA to DFA

$T = \epsilon\text{-closure}(q_0) = \{q_0, q_1\}$.

Dstates -

T = not marked

for each input 0 and 1

$= \epsilon\text{-clo}(\text{move}(\{q_0, q_1\}, 0))$

$= \epsilon\text{-clo}(\text{move}(\{q_0\}, 0) \cup \text{move}(\{q_1\}, 0))$

$= \epsilon\text{-clo}(\phi \cup q_1)$

$= \epsilon\text{-clo}(q_1)$

$= \{q_1\}$

let $\{q_1\} = B$

Now, $T = \{q_0, q_1\}$

$B = \{q_1\}$

The next input is -

$\epsilon\text{-clo}(\delta(\{q_0, q_1\}, 1))$

$= \epsilon\text{-clo}(\delta(\{q_0\}, 1) \cup \delta(\{q_1\}, 1))$

$= \epsilon\text{-clo}(\phi \cup q_2)$

$= \epsilon\text{-clo}(q_2)$

$= \{q_2, q_3\}$

let $\{q_2, q_3\}$ is labelled as C.

Now, (T) = $\{q_0, q_1\}$

B = $\{q_1\}$

C = $\{q_2, q_3\}$

The next step is check for B –

ϵ-clo($\delta(\{q_1\}, 0)$)

= ϵ-clo(q_1)

= $\{q_1\}$ = B (already exists)

Next,

ϵ-clo($\delta(\{q_1\}, 1)$)

= ϵ-clo(q_2)

= $\{q_2, q_3\}$ = C (already exists)

Now, (T) = $\{q_0, q_1\}$

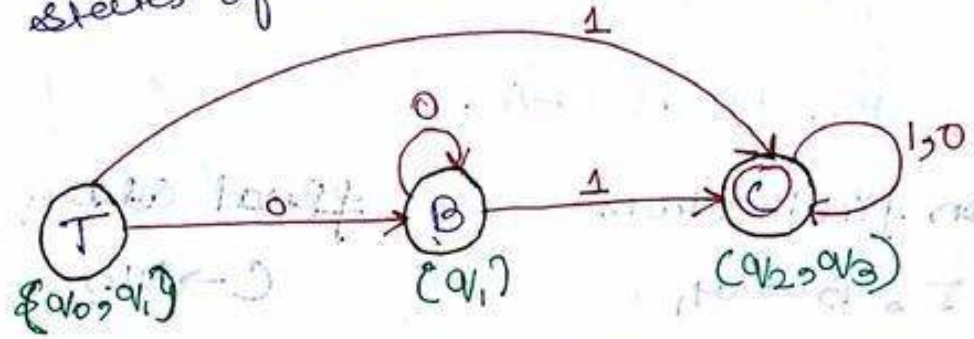
(B) = $\{q_1\}$

(C) = $\{q_2, q_3\}$

ϵ-clo($\delta(C, 0)$) = $\{\phi\}$ (dstates)

ϵ-clo($\delta(C, 1)$) = $\{q_2, q_3\}$ = C

∴ 3 states are discovered which are the states of the DFA.



DTran [T, a] = ∪ DTran [B, 1] = C

DTran [T, 0] = B/C DTran [C, 1] = C

DTran [T, 1] = C

Minimization of DFA

Algorithm:

→ Partition the set of states into 2 groups:

- G_1 : set of accepting states
- G_2 : set of non accepting states

→ For each new group G_1

- partition G_1 into subgroups such that
  - *states S_1 & S_2 are in the same group
- iff for all I/P symbols a_i
  - *states S_1 & S_2 have transitions to states in the same group

→ Start state of the minimized DFA is the group containing the start state of the original DFA.

→ Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

From the final DFA,

non final States	final States
I. B → G_1	C → G_2

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

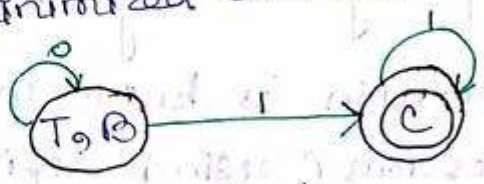
$\delta(T_0, 0) = B$ S_1

$\delta(B, 0) = B$ S_2

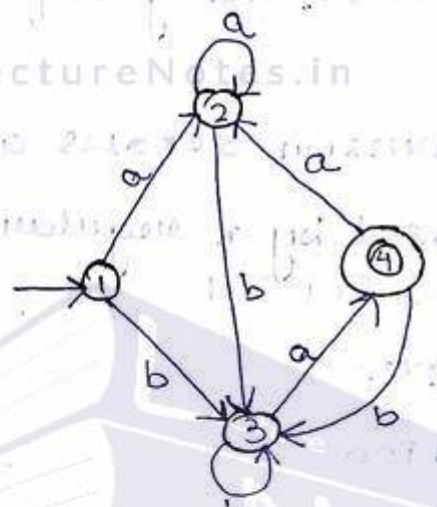
$\delta(T_0, 1) = C$ S_1

$\delta(B, 1) = C$ S_2

So minimized DFA is,



Q

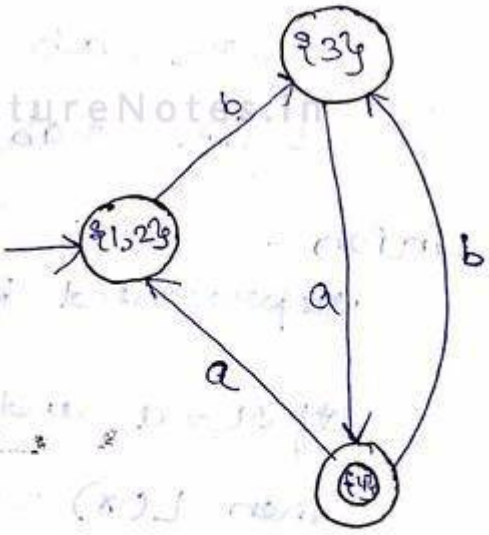


Groups : {1, 2, 3} {4}

{1, 2} {3}

(No more partitioning)

a	b
1 → 2	1 → 3
2 → 2	2 → 3
3 → 4	3 → 3



Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Lexical Analysis Regular Expressions

Contributed By:

Sibananda Achari

Trident Academy of Technology

Regular Expressions

→ We use regular expression to describe tokens of a programming language.

→ A regular expression is built up of simpler regular expressions (using defining rules).

→ Each regular expression denotes a language.

→ A language denoted by a regular expression is called as a regular set.

→ The operations are -

- Concatenation
- union / alternation
- Kleen closure

→ Concatenation -

If $r_1 = a$, $r_2 = b$

$r_1 . r_2 = ab$

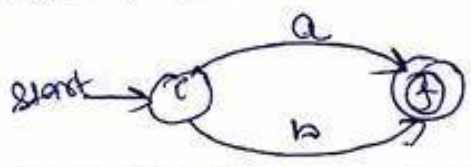
$L(r) = \{ab\}$

→ Union -

represented by + or | → pipeline

If $r_1 = a$ and $r_2 = b$.

then $L(r) = \{a, b\}$



A regular language is a formal language that is accepted by a DFA, NFA and read-only TM.

A regular language can be described by a regular expression built on simple regular grammar.

Operations on languages

→ Concatenation –

$$L_1 L_2 = \{S_1 S_2 \mid S_1 \in L_1 \text{ \& } S_2 \in L_2\}$$

→ union –

$$L_1 \cup L_2 = \{S \mid S \in L_1 \text{ or } S \in L_2\}$$

→ exponentiation –

$L^0 = \{\in\}$ $L^1 = L$ $L^2 = LL$

$\omega = ab$, $\omega^0 = \in$, $\omega^2 = \omega \cdot \omega = abab$

→ Kleene closure : Concatenate L with itself any no. of time

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

(It includes empty string with other string generated)

$L = \{a, b\} \cup \{\in\}$

→ Positive closure : Concatenate L with itself at least one or more

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

(It excludes empty string)

$L^+ = L^* - \{\in\}$

$(a,b)^* = \{\in, ab, aa, bb, ba, \ldots\}$

$(a,b)^+ = \{a, b, aa, ab, ba, bb, \ldots\}$

Converting a regular expression into a NFA (Thomson's Construction)

Conversion of expression → NFA.

imp: 23/12/13

Regular Expression to DFA directly (175-180)

Augment the regular expression r with a special end symbol # to make accepting states important the new expression is r#

r# → augmenting

functions -

r = (a|b)*a.

r# → (a|b)*a# → leave node

↓

Syntax tree

To construct a syntax tree we need precedence (* . |)

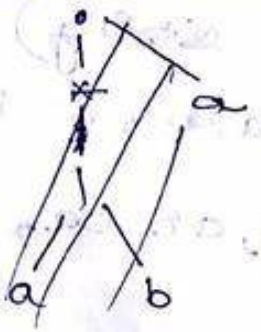
* → 1 (asterisk)

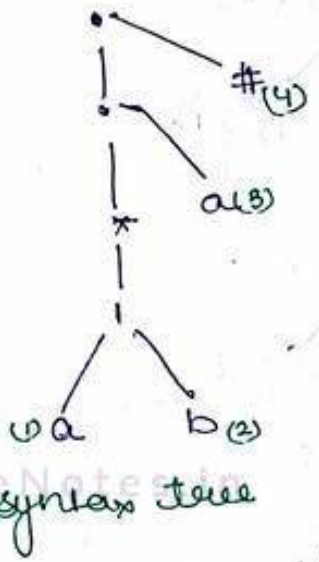
. → 2 (concatenation)

1 → 3 (alteration)

For expression, we have ( ) that's the highest precedence.

a+b





$r = (a|b)^* a$

Syntax tree

We numbered the leaf nodes.

Now, the final statement is to calculate the nodes.

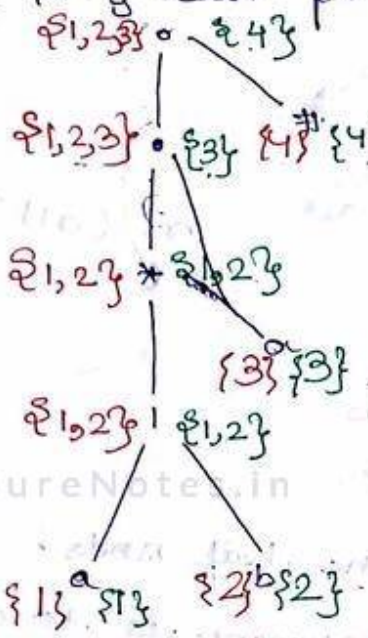
nullable: The sub tree at node n generates language including the empty string.

Firstposition: The set of positions that can match the first symbol of a string generated by the subtree at node n.

Lastposition: The set of position that can match that last symbol of a string generated by the subtree at node n.

followposition: The set of position that can follow position i in the tree.

I Calculate 1st & last pos^n of each node. (34)



(no itself)
last pos^n
first pos^n (no itself)

How to calculate firstpos & lastpos?

n	nullable(n)	firstpos(n)	lastpos(n)
leaf labelled ∈	true	∅	∅
leaf labelled with pos^n i	false	{i}	{i}
 c_1 c_2	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
 c_1 c_2	nullable(c_1) & nullable(c_2)	if (nullable(c_1)) 1st pos(c_1) ∪ 1st pos(c_2) else 1st pos(c_1)	if (nullable(c_2)) last pos(c_1) ∪ last pos(c_2) else last pos(c_2)
 c_1	true	firstpos(c_1)	lastpos(c_1)

(34)

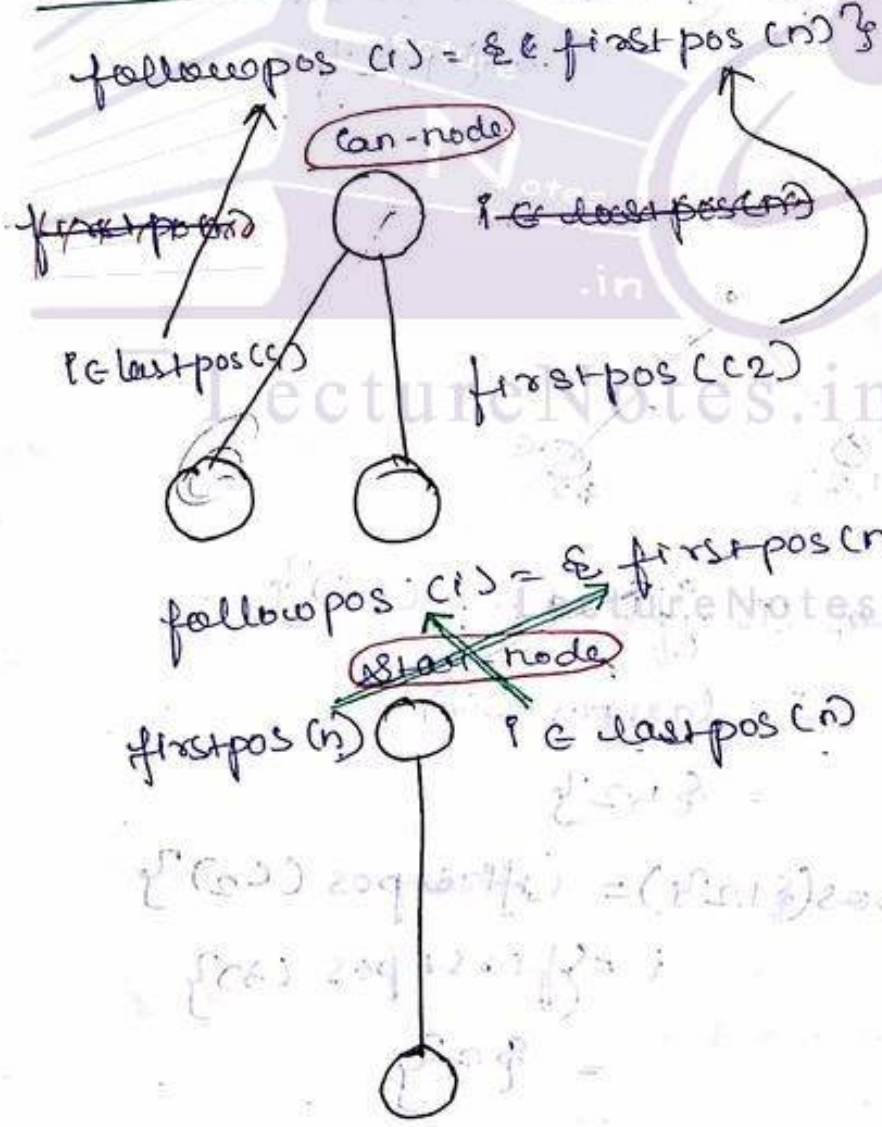
How to evaluate followpos?

2 rules define the function followpos:

→ If n is concatenation-node with left child c_1 and right child c_2, & i is a position in lastpos(c_1), then all positions in firstpos(c_2) are in followpos(i).

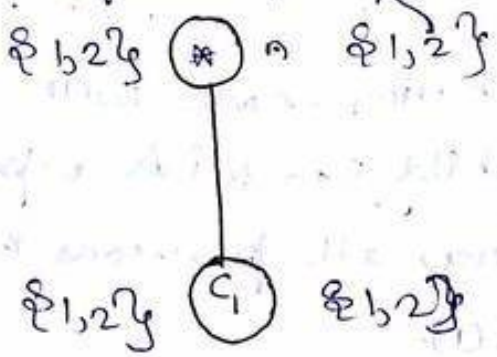
→ If n is a star-node, and i is a position in lastpos(n) then all position in firstpos(n) are in followpos(i)

VISUALISATION



Star node:

followpos (i)



i ∈ lastpos (n) → * (value = lastpos {1,2})

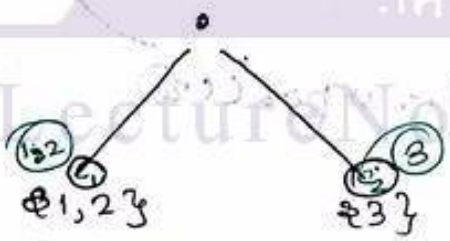
$s = \{1,2\}$

followpos (n) = { firstpos (n) }

followpos ({1,2}) = { firstpos (*) }

= {1,2}

Concatenation node:



followpos (n) = { firstpos (c2) }

$s = lastpos(c_1)$

= {1,2}

followpos ({1,2}) = { firstpos (c_2) }

i = { firstpos (3) }

= {3}

followpos

Then we define the function followpos for the positions leafnode and not for the nonleaf node.

26/12/18

$(a|\epsilon)bc^*\#$

Find out - nullable
- firstpos
- lastpos
- followpos
- create DFA using the algorithm

Lexical errors

→ Some errors are out of power of lexical analyzer to recognize.

```
fi (a == f(x)) ...
```

→ Here fi is taken as an identifier

* The advantage of two buffer is it overcomes overflow.

Error Recovery

→ Panic mode: When lexical analyzer is not able to continue with the process of compilation, it resorts to panic mode of recovery.

→ In this mode the following actions are performed to identify a token.
- Successive characters are ignored until we reach to a well formed token.

- delete one character from the remaining input.
- insert a missing character into the remaining input
- Replace a character by another character.
- Transpose two adjacent characters.

LEX

→ lex is a scanner generator

- input is description of patterns & actions.
- output is a C program which contains a function yylex() which, when called, matches patterns and performs actions per input.
- Typically, the generated scanner performs lexical analysis and produces tokens for the (YACC-generated) parser.

(YACC – Yet Another C compiler)

The Lex and Flex Scanner Generators

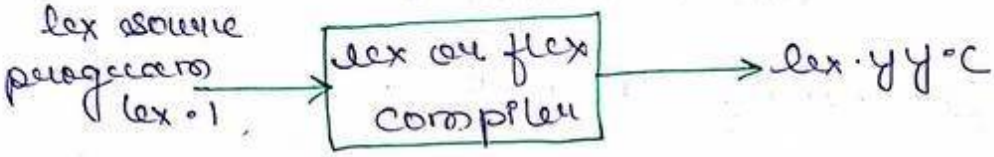
→ lex & its newer cousin flex are scanner generators.

→ Systematically translate regular definitions into C source code for efficient scanning

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

→ Generated code is easy to integrate in C applications. (39)

Creating a lexical Analyzer with Lex & FLEX



Lex Specification (Pg 143)

- A lex specification consists of 3 parts:

```
1. {
     regular definitions, C declarations
   %}
   %%
   translation rules
   %%
   user-defined auxiliary procedures
```

- The translation rules are of the form -

```
P1  {action1}
P2  {action2}
 :
Pn  {actionn}
```

(39)

$\delta(q_i, a) = q_j$

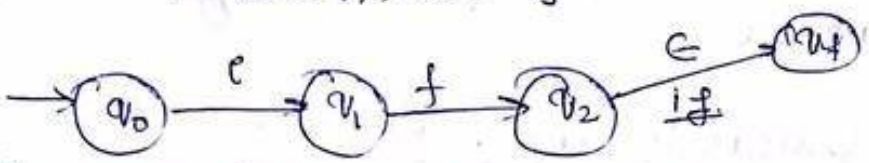


Fig. transition diagram to generate an identifier if

Lex Source Program

→ lex source is a table of

- regular expressions &
- corresponding program fragments

```
%{
  digit [0-9]
  letter [a-zA-Z]
%}
%%
{letter} ({letter}|{digit})*    (contains)
%%
main()
{ yylex(); }
```

← Invokes the lexical analyzer

contains the matching lexeme

```
{ printf ("id : %s\n", yytext);
  printf ("new line\n"); }
```

eg

The table is translated to a C pgm (lex.yy.c)

Lex ~~fragment~~ C program

→ Consider the Lex fragment:

- a {printf ("read 'a' \n"); }
- b {printf ("read 'b' \n"); }

→ After compiling we obtain a binary executable which when executed on the o/p.

The code produces -
asfghjkl read 'a'
ghjread 'b'
fghkjread 'b'

Example - A lex program

```
%{
  int abc_count, xyz_count;
%}
%%
ab[cC] {abC_count++;}
xyz {xyz_count++;}
\n {;}
. {;}
%%
main()
{ abC_count = xyz_count = 0;
  yylex();
  printf ("%d occurences of abc or abC \n",
                    abc_count);
  printf ("%d occurences of xyz \n", xyzl_
                    count);
}
```

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Syntax Analysis

Contributed By:

Sibananda Achari

Trident Academy of Technology

SYNTAX ANALYSIS

Token → [Syntax Analyser] → : Parse tree / Syntax tree

→ tool used for syntax and (syntactic construct) yacc is YACC.

→ SA = parser ← interacts with → symbol table

→ CFG is type-2 grammar, recognized by PDA.

→ DPDA → has odd palindrome recognition

→ NDPDA → has even palindrome recognition

→ CFG consists of → production rules
- set of NT
- set of T
- starting variable, S

→ $A \rightarrow \alpha$ (A ↓ NT; α → string of grammar symbols (T & NT))

→ $\alpha\beta\gamma$ → denotes string of grammar symbols.

* The symbols that are used as terminals are specified in the following form -
  - → lower case letters early in the alphabet as a, b, c - - - .
  - → operators as +, *, /, etc.
  - → parenthesis, comma
  - → digits like 0 to 9 & bold face identifiers as id, if

* The symbols for NT are -
  - → upper case letters early in the alphabet
  - → start symbol
  - → lower case italics as expr, stmt.
  - → upper case letters as X, Y, Z represent grammar symbol i.e. either T or NT.
  - → α, β, γ represent string of grammar symbol

Difference between SA & LA

→ Separating the syntactic structure of a language into lexical and non lexical parts provides a convenient way of modularising front end of compiler.

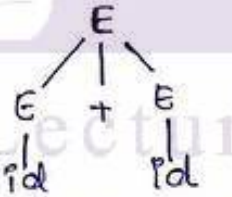
→ RE are concise & easy to understand notations for tokens.

→ By using RE, constructs like identifiers, keywords, constants and whitespaces can be described in a proper manner. However, CFG are used for describing nested string as if then else stat; balanced parenthesis etc.

Ambiguity.

→ Derivation of parse tree -
It follows 2 ways -
- leftmost derivation, lm
- rightmost derivation, rm

Eg -



$E \Rightarrow E+E \overset{rm}{\Rightarrow} E+id \overset{rm}{\Rightarrow} id+id$

$\overset{lm}{\Rightarrow} id+E$

$\overset{lm}{\Rightarrow} id+id$

If there exists 2 or more parse tree for a given CFG, then grammar is said to be ambiguous, needs to be resolved.

Imp Eg - $E \to E+E \mid E*E \mid E/E \mid (E) \mid id$

→ This is an ambiguous grammar. To resolve:
- precedence
- associativity need to be considered

$\therefore E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

T = term

F = factor

Elimination of left recursion:

A grammar is said left recursive if it has a NT A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$.

Rule to overcome:

For $A \to A\alpha \mid \beta$ including $\in$ then reduce left recursion by $A \to \beta A'$,

$A' \to \alpha A' \mid \in$

$A \rightarrow A\alpha \mid \beta$

⇓ Eliminate left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

To resolve ambiguity, we need to rewrite it using CFG -

So, from the last example,

i.e.

$E \rightarrow E+E \mid E*E \mid (E) \mid id$

⇓

$E \rightarrow \underbrace{E}_{A} + \underbrace{T}_{\alpha} \mid \underbrace{T}_{\beta}$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$A \rightarrow A\alpha \mid \beta$

Ans → $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT \mid \epsilon$

$F \rightarrow (E) \mid id$

Q $S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

→ $S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

$A \rightarrow Ac$

$A \Rightarrow Sd$

$\Rightarrow Aad$

The algorithm says that-

→ Arrange the non terminals in some sequence.

S, A or A, S
A_1 A_2 A_1 A_2

→ Two loops are used -

```
for (i=1 to n)
{
    for (j = 1 to i-1)
    {
        if (Ai → Aj γ)
        {
            if (Aj → δ1 | δ2 | δ3)
                replace Ai → δ1γ | δ2γ | δ3γ
        }
    }
}
```

A → Sd replaced by A → Aad

→ Eliminate left recursion.

A_i → non terminals

i = 2 $A_i = A_2 = A$

j = 1 $A_2 \rightarrow S\gamma$

$S \rightarrow Aa \mid b$

Now we need to replace it -

$A_2 \rightarrow Sd$

$A \rightarrow Aad$

$A \rightarrow bd$

Now, we are getting the production (17)
rule as -

$S \rightarrow Aa \mid b$

$A \rightarrow Ac$

$A \rightarrow Aad$

$A \rightarrow bd$

now, we need to eliminate left recursion for the above grammar.

$A \rightarrow \underset{\alpha}{A}\, c \mid \underset{\alpha}{A}\, ad \mid \underset{\beta}{bd} \mid \underset{\beta}{\epsilon}$

$S \rightarrow Aa \mid b$

$A \rightarrow bd\, A' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \epsilon$

Left factoring:

→ It is a grammar transformation that is useful for producing a grammar suitable for predictive parsing or top-down parsing.

→ When the choice between two alternative productions of A ($A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$) is not clear then, the decision of using the production rules to obtain right choice.

→ On such a situation, we have to back track & check for other choice.

→ Backtracking introduces overhead & this overhead needs to be eliminated or left factored.

(17)

Let a CFG be these -

$S \to cAd$

$A \to ab \mid a$

$\omega = cad$

$S \overset{lm}{\Rightarrow} cAd$ } * sentential (string of terminal & non terminal symbols)

$\overset{lm}{\Rightarrow} cabd$ } sentence (string of terminal symbols)

* $S \overset{lm}{\Rightarrow} cAd$

$\overset{lm}{\Rightarrow} cad$

On left factoring,

$A \to \alpha\beta_1 \mid \alpha\beta_2$

eg: $A \to ab \mid a$

⇓

left factored

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$

$A \to a\,A'$ ($\because \alpha \to a$)

$A' \to b \mid \epsilon$

A parse tree always starts from S symbol.

Dangling if - else problem

Let the CFG be -

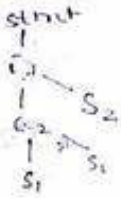
Syntax: Stmt → if expr then Stmt

if expr then Stmt else Stmt

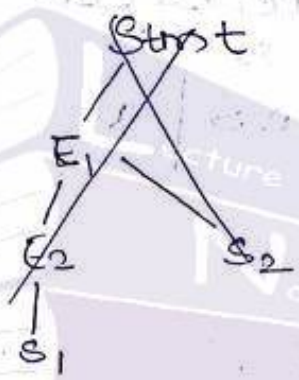
other stmt → terminal

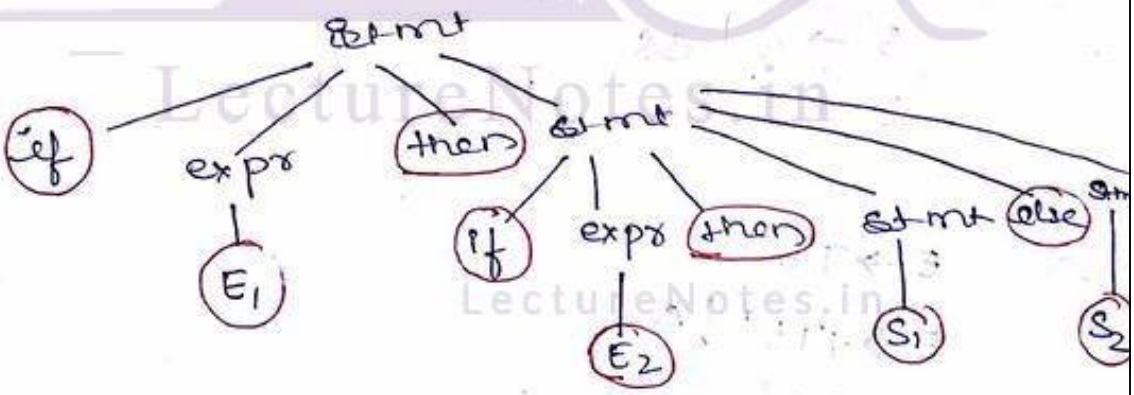
Eg if E_1 then if E_2 then S_1 else S_2

Draw a parse tree of the given problem by applying CFG.

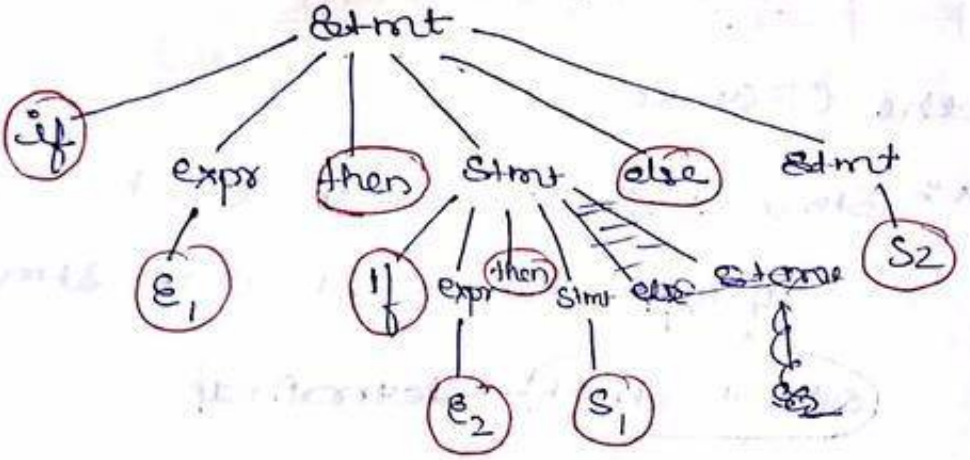


⇒





Parse tree contains every token generated by the grammar



∴ The above given grammar is ambiguous.

To eliminate ambiguity the grammar need to be left factored.

$S \rightarrow ietSe \mid ietSeS \mid a$

$E \rightarrow b$

$S \rightarrow ietSS' \mid a$

$S' \rightarrow \epsilon \mid eS$

$E \rightarrow b$

Q $E \rightarrow TE'$

$E \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

For the above CFG, [w = id + id * id], generate a parse tree using leftmost derivation.

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Top Down Parsing

Contributed By:

Sibananda Achari

Trident Academy of Technology

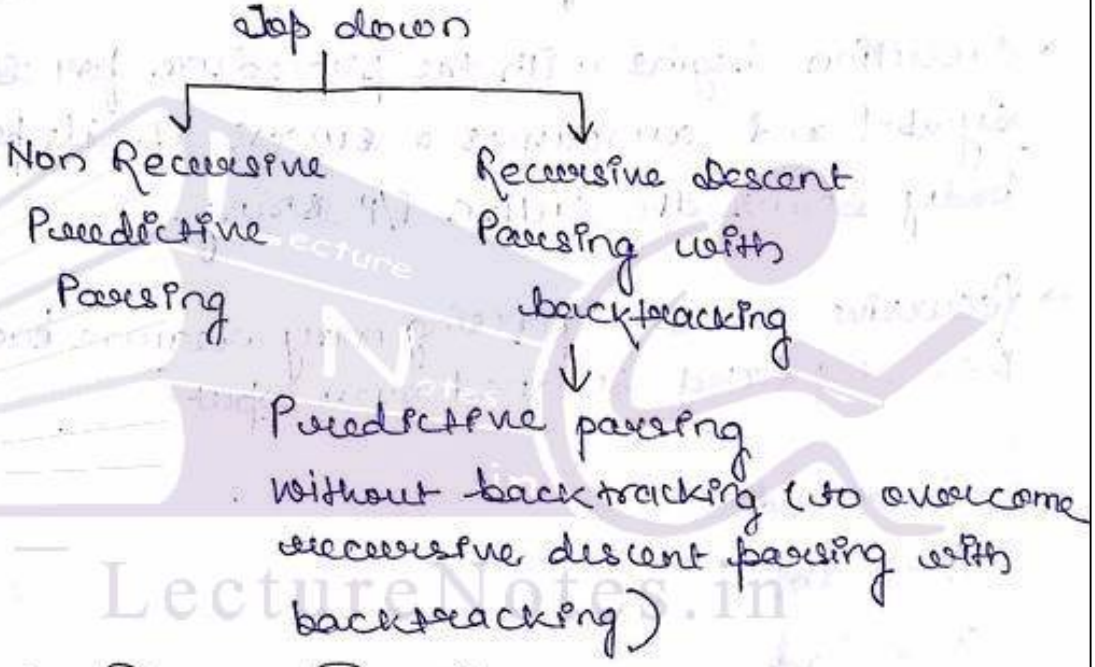
Types of Parsing Techniques

→ Top-down (leftmost derivation)

→ Bottom-up (rightmost derivation)

Top down parsing:

- It generates parse tree starting from the root and creating nodes of the tree in preorder.
- It can be viewed as finding leftmost derivation of an input string.



Recursive Descent Parsing

For every non-terminal, a procedure is present and it needs to be called.

Algorithm

```
void A()
{
  Choose a A-production A → X1 X2 .... Xk
  for (i=1 to k)
  {
    if (Xi is a non terminal)
      call procedure Xi();
```

```
else if (Xi == current i/p symbol)
    advance to the next i/p symbol
else
    /* report error */
}
}
```

→ A recursive descent parsing consists of a set of procedures (one for each non terminal).

→ Execution begins with the procedure for start symbol and announces a success if its procedure body scans the entire i/p string.

→ Recursive descent parsing may require backtracking i.e. repeated scanned over input.

Q $S \rightarrow cAd$

$A \rightarrow ab \mid a$

$\omega = cad$

→ Algorithm matches the production $S \rightarrow cAd$ with $\omega = cad$ (given string).

x_1 x_2 x_3

When $x_i = A$ is encountered, it uses another algorithm to make use of the production $A \rightarrow ab \mid a$.

When a is encountered, the loop is entered again to encounter b.

But 'ab' doesn't match with 'd' in w = cad, therefore, backtracking occurs.

Then the next production that is, A → a is advanced to. It matches, then i's value is incremented and d is also matched.

→ Based on the concept of DFS (Depth First Search)

Recursive Predictive Parsing

FIRST (calculated for α)

FOLLOW (calculated for NT (non-terminal)

Eg= If A → α, then we need to calculate FOLLOW (A) and FIRST (α).

→ FIRST (α)

In this set of terminals that are derived from α. Gives the first terminal -

Eg - S → cAd

FIRST (cAd) = {c}

FIRST (S) = {c}

For finding out the FOLLOW of a non terminal, RHS of production rule needs to be checked. In the above example, FOLLOW (A) = {d}

FOLLOW (A)

Gives the terminal that immediately follows the given non-terminal.

NOTE: (Pg -217 -221)

FIRST(α), where α is any string of grammar symbols is defined as the set of terminals that begin strings derived from α.

→ If $\alpha \overset{*}{\Rightarrow} \in$ (i.e. α produces $\in$ in one or more steps)

then,

$$\boxed{\text{FIRST}(\alpha) = \{\in\}}$$

→ If $A \Rightarrow c\gamma$

then

$$\text{FIRST}(c\gamma) = \{c\}$$

$$\text{FIRST}(A) = \text{FIRST}(c\gamma)$$

$$\boxed{\text{FIRST}(A) = \{c\}}$$

→ Three steps to be used to computes FIRST(X) (where, X is a grammar symbol), applying the following rules until no more terminals or $\in$ can be added to any first set are:

Step-I

If X is a terminal, FIRST(X) = {X}

Step-II

If X is a NT and for $X \to Y_1 \dots Y_n$

If $Y_1 \to \in$, then FIRST(X) = FIRST($Y_2, Y_3 \dots Y_n$)

Step-III

If $\forall i = 1$ to n are $\in$, then FIRST(X) = {$\in$}

FIRST() and FOLLOW()

$A \to \alpha \cap \epsilon$

FIRST $(\alpha) = \{\epsilon$

FIRST $(\epsilon) = \{\epsilon$

Q $F \to (E) \mid id$. Find FIRST $(E))$ &
FIRST (id).

→ FIRST $(E)) = \{($

FIRST $(id) = \{id\}$

FIRST $(F) = \{(, id\}$

Q $A \to cCd$

FIRST $(cCd) = \{c\}$

FIRST (A) = FIRST $(\alpha) = \{c\}$

Rule: If a long string is there —

$A \to Y_1 Y_2 Y_3 \ldots Y_n$

then, FIRST $(Y_1 Y_2 Y_3 \ldots Y_n) = \{Y_1\}$

- If Y_1 is ϵ, then in R.E

$\epsilon Y_2 Y_3 \ldots Y_n = Y_2 Y_3 \ldots Y_n$

FIRST $(Y_1 Y_2 Y_3 \ldots Y_n) = \{Y_2\}$

- If $Y_1 \ldots Y_n$ is ϵ,

then

FIRST $(Y_1 \ldots Y_n) = \{\epsilon\}$

(56) FOLLOW()

→ Follow is calculated for a non terminal only.

→ It is the set of terminals, a, that appear immediately to the right of

→ A in some sentential form.

→ Let us assume, there exists a derivation,

⇒ If $S \overset{*}{\Rightarrow} \alpha A a \beta$, then

FOLLOW(A) = a

⇒ If $S \rightarrow \alpha A$, then

FOLLOW(A) = { $ }

⇒ If Exp → Exp * Exp | Exp + Exp | id

let Exp be the start symbol S

then,

FOLLOW(S) = { $ }

⇒ If $A \rightarrow \alpha B \beta$, then everything in FIRST(β) − ϵ is in FOLLOW(B).

i.e. **FOLLOW(B) = FIRST(β) − ϵ**

⇒ If $A \to \alpha B$ or $A \to \alpha B \beta$ and FIRST(β) contains $\in$ then everything in FOLLOW(A) is in FOLLOW(B).

$$\text{FOLLOW}(B) = \text{FOLLOW}(A)$$

→ To find the FIRST and FOLLOW of the production rule the grammar should be non ambiguous then left recursion should be eliminated if exist and the grammar should be left factored.

Q $E \to E * E \mid E + E \mid (E) \mid id$

It is ambiguous as we cannot define whether we should go for * or + for derivation.

So we need to resolve it.

∵ + has lowest precedence

So we will go with -

$$E \to \underset{A}{E} \underset{\alpha}{+ T} \mid \underset{\beta}{T}$$

$$T \to T * F \mid F$$

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

Now, in the above new production we will check for left recursion.

∵ loop is created as the first symbol is repeated again and again as $E \rightarrow E+T \mid T$.

∴ The left recursion rule is applied -

$$E \rightarrow T\,E'$$

$$E' \rightarrow +TE' \mid \epsilon$$

~~From the above production -~~

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

Now, the above new production should be left factored.

The rule is -

$$\boxed{A \rightarrow \alpha\beta_1 \mid \alpha\beta_2}$$

In the above grammar, no production is as above mentioned rule so no left factor is required.

$ should be always at the end.

	FIRST ( )	FOLLOW ( )
E	(, id	), $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	*, +,), $

(59)

(V) F → (, id

FIRST ((E) = { (}

FIRST (id) = { id }

FIRST (F) = { (, id }

(IV) T' → * , ∈

(III) T

FIRST (FT') = FIRST (F)

= { (, id }

(II) E' FIRST (E') = + , ∈

(I) E → (, id

(I) We will check for (E) as (id) do not satisfy the condition - A → αBβ

E → T E' (α = ∈, B = T, β → ∈)

If FIRST (β) is ∈, then FOLLOW (E') = FOLLOW (E)

E → T E' (α = T, B = E', β = ∈)

If FIRST (E') contains ∈ then everything except ∈ is in FOLLOW (T).

(II) E' → +TE' | ∈ (α = +, B = T, β = E')

~~E' → +TE' | ∈~~

(III) FOLLOW (T) = FOLLOW (E')

T → FT' (α = ∈, A = F, α = T', β = ∈)

T → FT' | F (α = ∈, B = T', β = ∈ | α = ∈, B = F, β = ∈)

FOLLOW (T') = FOLLOW (T)

FOLLOW (F) = FIRST (T') − ∈

(IV) T' → * F T' | ∈

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

LL(1) Grammar

Scanning the input from left to right

Stands for leftmost derivation

1 input symbol of look ahead at each step to make parsing action

description.

→ 2 types of parsing are there -

- left &
- right

The grammar should be non ambiguous, left factored and do not have left recursion is called LL(1) grammar.

Parsing table is a 2D array where rows are defined by non terminals and columns are defined by terminals

Pg - (224 - 228)

Construction of Recursive Predictive PT (61)

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ∈	E' → ∈
T	T → FT'			T → FT'		
T'		T' → ∈	T' → *FT'		T' → ∈	T' → ∈
F	F → (id)			F → (E)		

Steps to built the Parsing Table

→ For each production $A \to \alpha$ of the grammar, do the following -

- If ∈ is in FIRST(α), then for each terminal in FOLLOW(A) add production rule $A \to \alpha$ to the parsing table (M[ ] for the non-terminal A & terminal B),

  M[A, b].

- If ∈ is in FIRST(α) and $ is in FOLLOW(A), then add the production rule $A \to \alpha$ to the parsing table

  M[A, $].

- For each terminal a, in FIRST(α) add $A \to \alpha$ to $M[A, a]$.

Q $S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

Find the FIRST() & FOLLOW() and then construct the predictive parsing table.

→ FIRST(S) = i, a

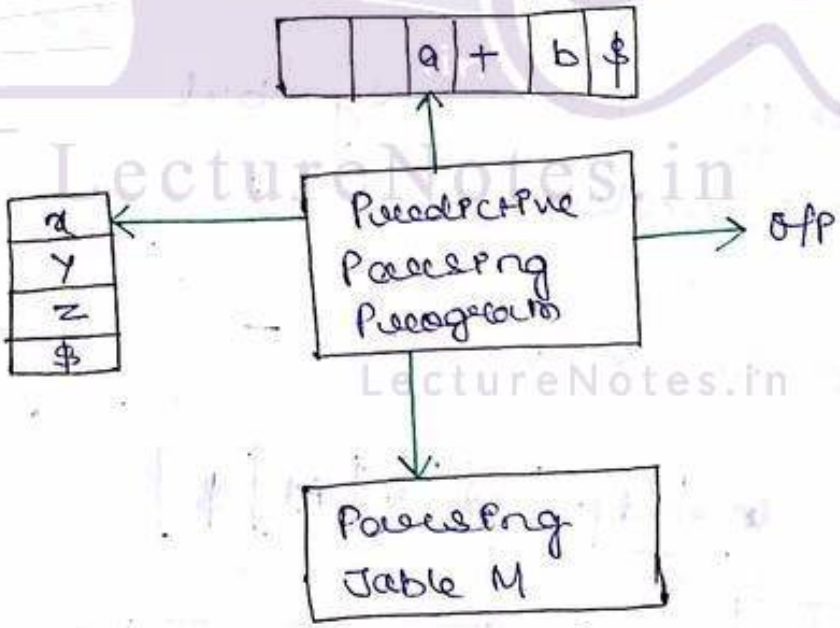
	FIRST()	FOLLOW()
S	i, a	e, $
S'	e, ε	e, $
E	b	t

	i	e	t	a	b	$
S	S→iEtSS'	S→eiEtSS'		S→a		
S'		S'→eS S'→ε				S'→ε
E			E→b		t→b	

It can be built by maintaining a stack explicitly. The parser may mix / induces a left most derivation if w is the input that has been matched so far then the stack holds a sequence of grammar symbols α such that

$$S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$$

The parser ~~remfx~~ is table driven. It uses predictive parsing table M. The table driven parser has an I/P buffer, a stack containing a sequence of grammar symbols, a parsing table and an output string.



I - Set the input point to the first symbol of w,

where w → I/P string provided

II- Set X to the TOS

III- while (X != $)

IV- If (X is a) pop the stack and advance the input ip.

V - else if (X is a terminal) error();

VI - else if (M[X,a] = X → Y1 Y2 ... Yk)

VII - {

o/p the production X → Y1 Y2 ... Yk

VIII - Push Yk, Yk-1 ..... Y1 onto the

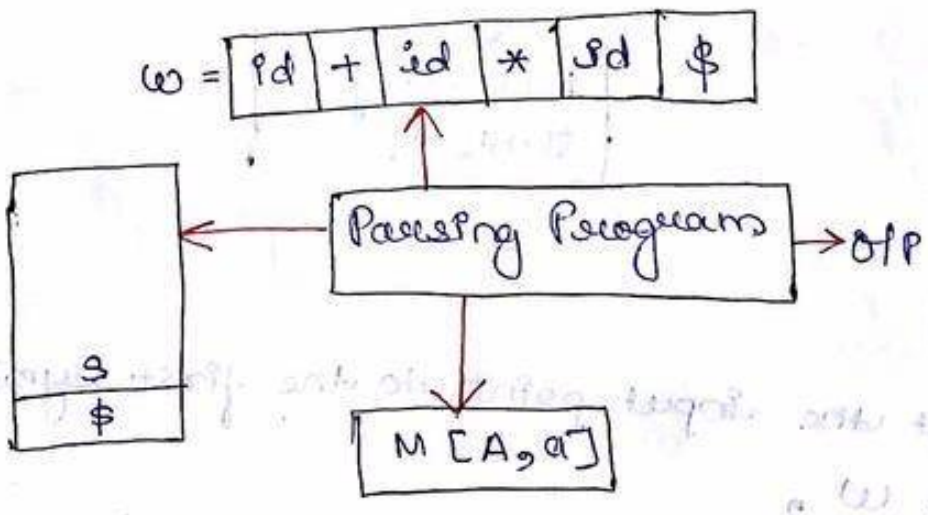
IX - stack such that Y1 is on the top

}

X- Set X to the TOS symbol

XI - ends while //end while

Q $w = id + id * id$



Matched	Stack	I/P	O/P / Action
	E $	id + id * id $	
	T E' $	id + id * id $	E → TE'
	F T' E' $	id + id * id $	T → FT'
id	id T' E' $	id + id * id $	F → id
	T' E' $	+ id * id $	matched
	E' $	+ id * id $	T' → ∈
id +	+ T E' $	+ id * id $	E' → +TE'
	T E' $	id * id $	matched
	F T' E' $	id * id $	T → FT'

The CFG for E is –

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

M [E, id] =

$E \rightarrow TE'$

id + id	id T' E' $	id * id $	F → id
	T' E' $	* id $	matched
id + id *	* F T' E' $	* id $	T' → *FT'
	F T' E' $	id $	~~E' → +TE'~~ matched
id + id * id	id T' E' $	id $	F → id
	T' E' $	$	matched
	E' $	$	T → ∈
	$	$	E' → ∈

(66) # Types Of Errors

Common programming errors can occur at different levels of compilation are.

→ **Lexical Errors :**

Eg- misspelling of identifiers, keywords / operators and missing codes around the text string.

→ **Syntactic Errors :**

Eg- missing braces / extra braces, misplaced semicolon.

→ **Semantic Errors :**

Eg- It includes mismatch between operators and operands.

→ **Logical Errors :**

Eg- It can be anything from incorrect reasoning on the part of the programmer.

→ The error handler in a parser has to report the presence of errors clearly and accurately, recover from each error quickly to detect subsequent errors and at minimal overhead to process the programs.

(67)

→ The different error recovery strategies are -

- Panic Mode Recovery
- Phrase level Recovery

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Bottom Up Parsing

Contributed By:

Sibananda Achari

Trident Academy of Technology

→ Bottom-Up Parsing

- Operator Precedence Parsing etc.
- LR Parsers

@15-01-14

→ Bottom up parsing corresponds to construction of a parse tree for an input string.

→ Beginning at the leaves and working up towards the root.

→ Bottom-up parser are also known as shift-reduce parsers.

→ Shift-reduce parsers are of 2 types -

- Operator precedence parser: It is applied to small class of grammar.
- LR Parsers: It scans the i/p string from left to right and generates a rightmost derivation tree.

Q Why shift - reduce?

→ Bottom up parsing can be thought as a process of reducing a string w to the start symbol S of the grammar. At each reduction step, a specific substring matching the body of the production is replaced by the non terminal at the head of the production.

There are 4 types of actions performed by Shift - reduce parsers:

→ Shift: It shifts the next I/P symbol on the top of the stack.

→ Reduce: The right end of the string to be reduced must be present at the TOS. Locate the left end of the string within the stack and replace it by non terminal.

→ Accept: After successful completion of parsing, the parser announces accept.

→ Error: In case of syntax error, calls an error recovery routine.

Handle Pruning

→ Handle pruning is related to the replacement of a handle within (it is a substring which matches the body of the production) by a non terminal present at the head of the production.

→ It can be obtain from rightmost derivation in reverse. Formally, if S produces $\alpha A w$ i.e.

$$S \Rightarrow \alpha A w$$
$$\Rightarrow \alpha \beta w$$
(rm)

→ 

→ Reverse of rightmost derivation → handle pruning

→
$$\alpha \beta w$$
$$\Downarrow$$
$$\alpha A w$$
$$\Downarrow$$
$$S$$

→ It identifies a position following α where the handle of the string $\alpha\beta w$ can be identified and β is replaced by the non terminal A.

Q

Input	Handle
id * id	

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Right most derivation of id * id

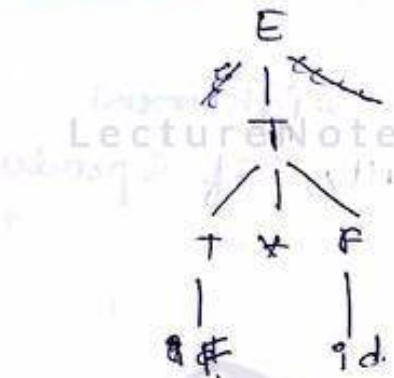
$E \Rightarrow E+T$

$\Rightarrow E+T*F$

$\Rightarrow E+T*(E)$

$\Rightarrow E+T*$

$E \quad \varepsilon$

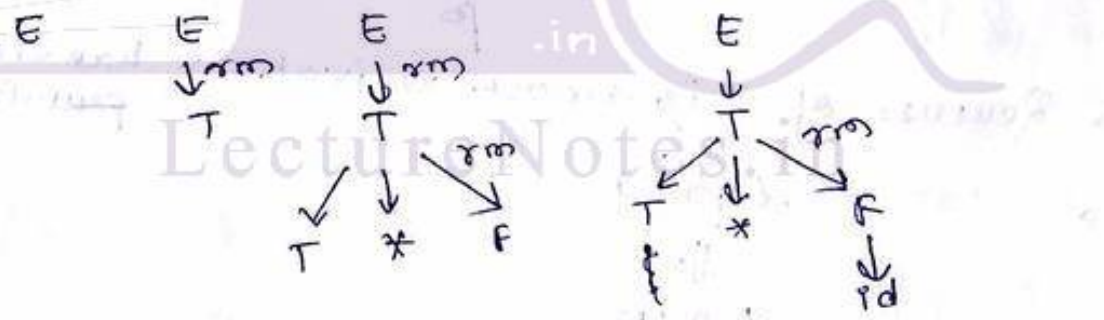


$E \Rightarrow T$

$\Rightarrow T*F$

$\Rightarrow T*id$

$\Rightarrow id*id$



Input	Handle	Replacing production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$
E		

Q Using the input string $id * id \$$, define the steps of action performed by shift-reduce parser. Using a stack, where the stack initially has end marker $\$$.

→

Stack	Input	Action
$\$$	$id_1 * id_2 \$$	
$\$ id_1$	$id_1 * id_2 \$$	$F \to id$
$\$ F$	$* id_2 \$$	$T \to F$
$\$ T$	$* id_2 \$$	$F \to id$
$\$ T *$	$id_2 \$$	$T \to T * F$
$\$ T * id_2$	$\$$	$E \to T$
$\$ T * F$	$\$$	
$\$ T$	$\$$	
$\$ E$	$\$$	

Intermediate Code Generation

20/01/14

→ In compiler, we generate Intermediate code & target code.

→ Intermediate code generates the target code.

Benefits of Intermediate code generation

→ It is machine Independent. As it is independent, a compiler for a different machine can be used by attaching it for the new machine.

→ A machine Independent code optimiser can be applied to the Intermediate code representation.

3 address Code :

→ There are different forms of Intermediate code representation. One of them is three address code.

→ It is represented as for the following format-

$$x := y \; op \; z$$

→ There will be always one operator & two operands on right hand side. And it permits one operand at LHS.

→ Here, x y z can be constant, temporary variables, memory addresses or variable reads.

→ Here, temporary variables are generated by the compiler.

Statements	Three address code format
→ Assignment Statements having binary operators.	$x := y \; op \; z$
→ Assignment Statements having unary operator	$x := op \; z$
→ Copy Statement / Assignment Statement	$x := y$
→ Unconditional jump statements	goto L (where L is the label)
→ Conditional jump statements	if x relop y goto L ↳ relational operator
→ procedure call / function call statement	param x_1 → parameters param x_2 ⋮ param x_n call P, n → no. of arguments ↓ Procedure name
→ Indexed statement Eg: $x = arr[i]$ $arr[i] = x$	$x := y[i]$ $x[i] := y$
→ Pointer / Address statement	$x := \&y$ $*x := y$ $x := *y$

Eg: M = a * y * z

Find its three address code.

⇒ The three Address code is

$t_1 := a * y$

$t_2 := t_1 * z$

$M := t_2$

Implementation of Three Address Code

→ Three address statement is an abstract form of intermediate code.

→ On compiler, this statements can be implemented as records with fields for the operator and the operands.

→ 3 such representation are there -

- Quadruples
- Triples
- Indirect Triples

Eg Find the three address code (TAC) of the following statement:

$$a = \underbrace{b * -c}_{(1)} + \underbrace{b * -c}_{(2)}$$

⇒ The TAC for the above is -

0	$t_1 = -c$
1	$t_2 = b * t_1$
	~~$a = t_1 + t_2$~~
2	$t_3 = -c$
3	$t_4 = b * t_3$
4	$t_5 = t_2 + t_4$
5	$a = t_5$

Quadruple representation:

Index	operator	arg 1	arg 2	result
0	u-	c		t_1
1	*	b	t_1	t_2
2	u-	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	assign	t_5		a

Triple Representation:

Index	operator	arg 1	arg 2
0	u-	c	
1	*	b	(0)
2	u-	c	
3	*	b	(2)
4	+	(1)	(3)
5	assign	a	(4)

Indirect Triple Representation

Table-I.

	op	arg 1	arg 2
21	u-	c	
22	*	b	(21)
23	u-	c	
24	*	b	(23)
25	+	(22)	(24)
26	assign	a	(25)

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Table - II

	Statement
0	21
1	22
2	23
3	24
4	25
5	26

Q Find the triple representation of the following statements:

i) $x[i] = y$

ii) $x = y[i]$

⇒ The TAC for the above is –

index	op	arg1	arg2
0	[]=	x	i
1	assign	(0)	arg y
2			

index	op	arg1	arg2
0	[]=	y	i
1	assign	x	(0)

Operator Precedence Parsing

23/01/14 (77)

→ Precedence Relations

→ Bottom-up parsers for a large class of CFG can be easily developed using operator grammars.

→ operator grammars have the property that no production right side is empty or has two adjacent nonterminals.

→ It should not produce ∈-production.

→ This property enables the implementation of efficient operator-precedence parsers. These parsers rely on the following 3 precedence relation:

Relation	Meaning
a <· b	a yields precedence to b
a =· b	a has same " as b
a ·> b	a takes precedence over b

→ These operator precedence relations allow to delimit the handles in the right sentential forms : <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

24/01/14

Q What are the disadvantages of operator Precedence Parsing?

→ It is hard to handle tokens like - Sign which has two different precedence

→ It is applied only to small class of

Operator grammar.

→ Relationship between a grammar for the language to be parsed and the operator precedence parser is very tedious.

Thus

Ways to determine Precedence Relations

→ There are 2 common ways -

- using operator precedence relations
- Construct unambiguous grammar for the language which reflects both precedence and associativity

→ **using operator precedence relation**

→ The idea behind it is to delimit the handle of a right sentential form with $<\cdot$ marking left end and $=\cdot$ appearing interior to the handle if any and $\cdot>$ marking the right end.

Eg - $\$ id_1 + id_2 * id_3 \$$

It is not in the sentential form as no non-terminal is there.

So we need to delimit the handle:

Id has higher precedence over +.

$$\overset{a_i}{12} \;\cdot\!> \overset{a_{i+1}}{+} \;<\!\cdot\; \overset{a_{i+2}}{13}$$

$$E \quad + \quad E$$

$$E$$

→ Operator precedence parser belongs to the shift reduce parser.

→ $ has less precedence as compared to any terminal.

$$\therefore \cdot > \$ \;\; id_1 \;+\; id_2 \;*\; id_3 \cdot > \$$$

relation ⇒ $\boxed{\$ \lessdot id_1 \doteq > + \lessdot id_2 \doteq > * \lessdot id_3 \doteq > \$}$

Step-I: Scan the string from the left end until the leftmost $\cdot>$ is encountered

Step-II: Then scan backward over any $\doteq$ until $<\cdot$ is encountered.

Step-III: Then handle contains everything to the left of the first $\cdot>$ and to the right of $<\cdot$ including any surrounding non terminal.

↓ meaning

To check whether two adjacent non terminals appears in a right sentential form.

→ $\$ \lessdot \cdot \; id_1 \doteq > + \lessdot id_2 \doteq > * \lessdot \cdot \; id_3 \doteq > \$$

→ $\$ \; E + <\cdot id_2 \cdot> * <\cdot id_3 \cdot> \$$

→ $\$ \; E + E * <\cdot id_3 \cdot> \$$

→ $\boxed{\$ \; E + E * E \; \$}$ ⇐ Handle

→ $ + * $

Now, we need to establish a relationship for the above –

→ $ <· + <· * ·> $

1st ·> is scanned so move back

$ ∴ E * E should be reduce first

→ $ E + E $

→ $ <· + ·> $

1st ·> is scanned so move back

→ $ E $

The terminals identified are id

+ * $

Now, we will construct a table where no. of rows = no. of columns.

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Q id + - * / ↑ $

→ Operator Precedence Parsing Algorithm

Initialize: Set ip to point to the first symbol of w$.

Repeat: let a be the top stack symbol, and b the symbol pointed to by ip.

if S is on the top of the stack (accept) and ip points to S then return

else

let a be the top terminal on the stack, and b the symbol pointed to by ip.

if a <· b or a =· b then

push b onto the stack

advance ip to the next input symbol.

elseif a ·> b then

repeat

pop the stack

until the top stack terminal is related by <·

by the terminal most recently popped

else error ()

end

27/1/14

Steps of Derivation	Stack	Input String	Action
	$	$id_1 + id_2 * id_3$ $	Input to the algorithm is string w$ and precedence table
			~~accept~~ shift
$E \to id_1$	id_1 $	$+id_2 * id_3$ $	reduce
	$	$+id_2 * id_3$ $	$a = id \; .> \; b = +$ shift
	$+ $	$id_2 * id_3$ $	shift
$E \to id$	$id_2 + $	$* id_3$ $	reduce
	$+ $	$* id_3$ $	shift

The production rule is –

$E \to E+E \mid E*E \mid id$

– popped
– action

sp → pointer pointing to an input string

$a = \$ \qquad b = id$

$a <\cdot b$

larger prec – pop
less prec – push

	$* + $	id_3 $	shift
$E \to id$	$id_3 * + $	$	reduce
$E \to E*E$	$* + $	$	reduce
	$+ $	$	reduce
$E \to E+E$	$	$	

NOTE:

Let Θ_1 and Θ_2 be two operators.

If the relationship between the two operators has same precedence then we can write it as

$$\Theta_1 \cdot> \Theta_2 \quad \text{or} \quad \Theta_2 \cdot> \Theta_1$$

→ If the operators have same precedence i.e: $\Theta_1 = \Theta_2$ then we can write it as $\Theta_1 \cdot> \Theta_2$ or $\Theta_2 \cdot> \Theta_1$ if they are left associative.

Eg - + -
* /

→ If the operators have same precedence but right associative then

$$\Theta_1 <\cdot \Theta_2 \text{ and } \Theta_2 <\cdot \Theta_1$$

Eg - exponentiation (↑)

→ If $\Theta_1 \cdot> \Theta_2$ and they are left associative then the precedence relation can be written as -

$$\Theta_1 \cdot> \Theta_2 \text{ and } \Theta_2 <\cdot \Theta_1$$

→ If $\Theta_1 <\cdot \Theta_2$ and the relationship is in combination of left and right associativity:

$$\Theta_1 <\cdot \Theta_2 \text{ and } \Theta_2 \cdot> \Theta_1$$

Eg - * and +

→ Make Θ_1 is having less precedence than id

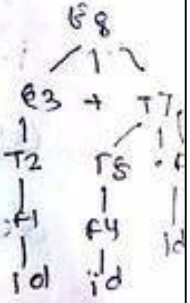
i.e. $\Theta_1 <\cdot id$ and $id \cdot> \Theta_1$

$\$ <\cdot id$ and $id \cdot> \$$

$(<\cdot id$ and $id \cdot>)$

$(<\cdot ($ and $) \cdot>)$

$(\doteq)$



Q. $(id_1 * (id_2 \uparrow id_3) - id_4 / id_5$

(Pg-158-173)

→ Operator Precedence Relation from associativity and Precedence.

Algorithm for Constructing Precedence function

→ Create function f_a for each grammar terminal a and for the end of string symbol;

→ Partition the symbols into groups so that f_a and g_b are in the same group if $a = b$ (there can be symbols in the same group even if they are not connected by this relation);

→ Create a directed graph whose nodes are in the groups next for each symbols a & b.

do : place an edge from the group of

② g_b to the group of f_a, if $a < b$,

otherwise if $a > b$ place an edge from the group of f_a to that of g_b

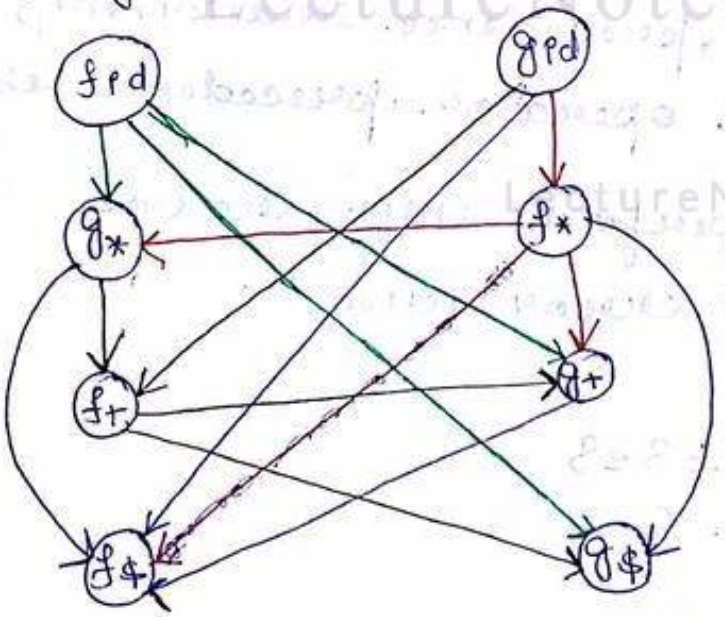
→ If the constructed graph has a cycle then no precedence functions exist. when there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

30/1/14

Eg- $E \to E+E \mid E*E \mid id$

I/P string - id + id * id $

Every a should be mapped to b -



After $ no outgoing way is there, therefore we calculate the path from the starting point to $.

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

(86)

The precedence table -

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Computation Of First Terminal, Last Terminal, Leading and Trailing of a non-terminal for a given non-production rule

Step-I: Compute the first terminal & last terminal.

Step-II: Compute leading and trailing.

Step-III: From the leading and the trailing, establish precedence relationship and build operator precedence table.

Step-IV: Finally perform shift, reduce accept or error action.

Q $S \rightarrow iCtSeS$

$S \rightarrow iCtS$

$S \rightarrow a$

$C \rightarrow b$

→ The grammar is ambiguous and this grammar defines precedence relationship between a pair of terminals that can have two precedence relationship.

The First Terminal can be calculated for a given non terminal were the terminals that can be first in a string derive from that non terminal.

Similarly, for each non terminal those terminals that are the last terminal in a string derived from that non terminal.

There are 2 rules to calculate first terminals and the last terminals for a given non terminal.

First Terminal	Last Terminal
Rule 1: $A \overset{*}{\Rightarrow} \gamma a \delta$ where, γ is either $\in$ or a single non terminal then add a to the ~~first~~ last	

Rule 1:
First terminal A is equal to a such that

$$a \mid A \overset{+}{\Rightarrow} \gamma a \delta$$

where, $\gamma \rightarrow$ either $\in$ / non terminal.

Add a to the first terminal of A = a such that $A \Rightarrow \gamma a \delta$ and γ is $\in$ / single non terminal.

Rule 2:
Last terminal of A = a such that

$$A \Rightarrow \gamma a \delta$$

and, δ is $\in$ / single non terminal

	First Terminal	Last Terminal
S	i, a	e, t, a
C	b	b

- $C \rightarrow b$; $\gamma = \in$, $a = b$, $\delta = \in$
- $S \rightarrow a$; $\gamma = \in$, $a = a$, $\delta = \in$
- $S \rightarrow iCtS$; $\gamma = \in$, $a = i$, $\delta = CtS$
- $S \rightarrow iCtS$; $\gamma = i$, $a = C$, $\delta = tS$
- $S \rightarrow iCtSeS$; $\gamma = \in$, $a = i$, $\delta = CtSeS$
- $S \rightarrow iCtSeS$; $\gamma = i$, $a = C$, $\delta = tSeS$
- $S \rightarrow iCtSeS$; $\gamma = iCt$, $a = S$, $\delta = eS$

Algorithm to Construct Operator Precedence Relation

Rule 1: leading (A) = $\{a \mid A \stackrel{+}{\Rightarrow} \gamma a \delta\}$

where, $\gamma \rightarrow$ either ϵ / a single nonterminal

Rule 2: Trailing (A) = $\{a \mid A \stackrel{+}{\Rightarrow} \gamma a \delta\}$

where, $\delta \rightarrow$ either ϵ / a single nonterminal.

Rule 3: If a is in the leading of A and there is a production

$B \rightarrow A\alpha$

then a is in the leading of B.

Rule 4: Compute the leading & trailing from the first & last terminal.

Rule 5: The leading is use to establish $<\cdot$ relationship and the trailing helps to calculate $\cdot>$ relationship.

$a <\cdot b$ – leading

$a \cdot> b$ – trailing

Rule 6:
- Set $\$$ is less than a i.e. $\$ <\cdot a$ $\forall a$ in leading (S).
- Set $b \cdot> \$$ $\forall b$ in trailing (S).

where S → start symbol in production rule.

Rule 7: Choose a production.

Let $A \rightarrow X_1 X_2 \ldots X_n$

For $i = 1, 2, \ldots, n-1$

→ if (X_i and X_{i+1} == terminal)

then set $X_i = X_{i+1}$

→ if ($i <= n-2$ && X_i and X_{i+2} == terminal and X_{i+1} == non terminal)

then set $X_i = X_{i+2}$

→ if (X_i == terminal and X_{i+1} == non terminal)

then set $a \lessdot X_{i+1}$ ∀ a in leading (X_{i+1})

→ if (X_i == non terminal and X_{i+1} == terminal)

then set $a \gtrdot X_{i+1}$ ∀ a in the trailing (X_i).

Q $S \rightarrow i C t S e S$

$S \rightarrow i C t S$

$S \rightarrow a$

$C \rightarrow b$

→

	Leading	Trailing
First Terminal		Last Terminal
S	i, a	t, e, a
C	b	b

Once table satisfies for a particular relationship then use another value of i.

Now, we need to establish relationship between $ and S. So for that —

(i) leading (S) —

{i, a} $ <· i $ <· a

	i	b	t	a	e	$
i			=	⋖		
b			·>			
t	<·		⋖	⋖	=	·>
a	⋖		⋖		·>	·>
e			⋖	<·	·>	·>
$	<·			⋖		

(ii) Trailing (S) = {t, e, a}

t ·> $ e ·> $ a ·> $

~~(iii) leading (C) = {b}~~

$ <· b

Let the production be —

S → iCtSeS
A → x1 x2 x3 x4 x5 x6

i = 1 to 5 (∵ n = 6 ∴ n-1 = 5)

→ i = 1 x1 = i x2 = C
1 ≤ 4 x1 = i x3 = t x2 = C
→ i = 2 x2 = C x3 = b x4 = S
→ i = 3 x3 = t x4 = S x5 = e
→ i = 4 x4 = S x5 = e x6 = S
→ i = 5 x5 = e x6 = S

$S \rightarrow i\ C\ t\ S$
$A \quad x_1\ x_2\ x_3\ x_4$

for $i = 1$ to 3

$i=1 \quad x_1 = i \quad x_2 = C \quad x_3 = t$
$i \doteq t$

$i=2 \quad x_2 = C \quad x_3 = t$
~~t, ab~~ $\rightarrow t$

$i=3 \quad x_3 = t \quad x_4 = S$
$* <\cdot\ i, a$

$S \rightarrow a$
$A \rightarrow x_1$

Stack	Relation	Input	Action
$	$<\cdot$	ibtibtaea$	shift
i $	$<\cdot$	btibtaea$	shift
b i $	$\cdot>$	tibtaea$	reduce
i $	$\doteq$	tibtaea$	shift
t i $	$<\cdot$	ibtaea$	shift
i t i $	$<\cdot$	btaea$	shift
b i t i $	$\cdot>$	taea$	reduce.
i t i $	$\doteq$	taea$	shift
t i t i $	$<\cdot$	aea$	shift
a b i t i $	$\cdot>$	ea$	reduce
t i t i $	$\doteq$	ea$	shift
e t i t i $	$<\cdot$	a$	shift
a e t i t i $	$\cdot>$	$	reduce
e t i t i $	$\cdot>$	$	reduce
t i t i $	$\cdot>$	$	reduce
i t i $	$\cdot>$	$	reduce

id $	.>	$	reduce.
i $	.>	$	reduce
$		$	accept

Assignment

Resolve the ambiguity in the given grammar. Build operator precedence parsing table using leading and trailing and then perform shift reduce parsing for input - ibtibtaea$ using the non ambiguous production rule.

Shift reducing Parsing = Bottom up Parsing

Conflicts During Shift - Reduce Parsing

→ There are CFG for which shift - reduce parsers cannot be used.

→ Stack contents and the next input symbol may not decide action;

- Shift / reduce conflict : Whether make a shift operation or a reduction.
- reduce / reduce conflict : The parser cannot decide which of several reductions to make.

→ If a shift - reduce parser cannot be used for a grammar, that grammar is called as non - LR(k) grammar.

- L → left - right scanning
- R → rightmost derivation
- k → k look ahead.

→ Any ambiguous grammar cannot be LR grammar

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

LR Parsing Algorithm

Contributed By:

Sibananda Achari

Trident Academy of Technology

3/2/2013

2. LR Parsers

→ Three types of LR parsing are there -

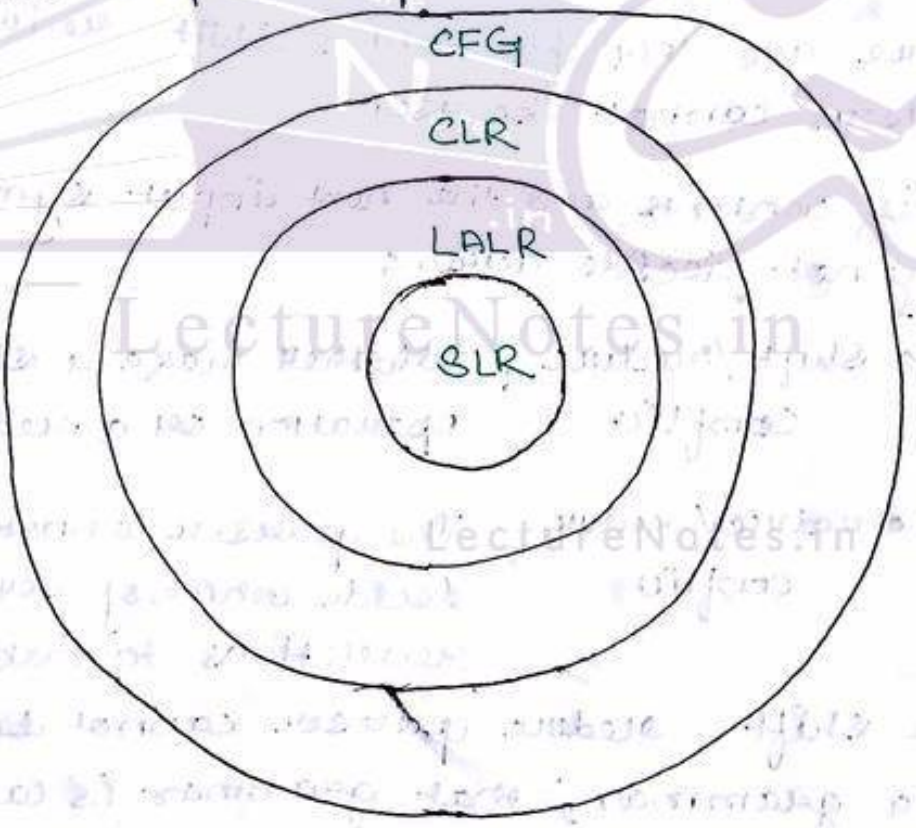
- → SLR
- → CLR
- → LALR

→ CLR : Canonical LR parser

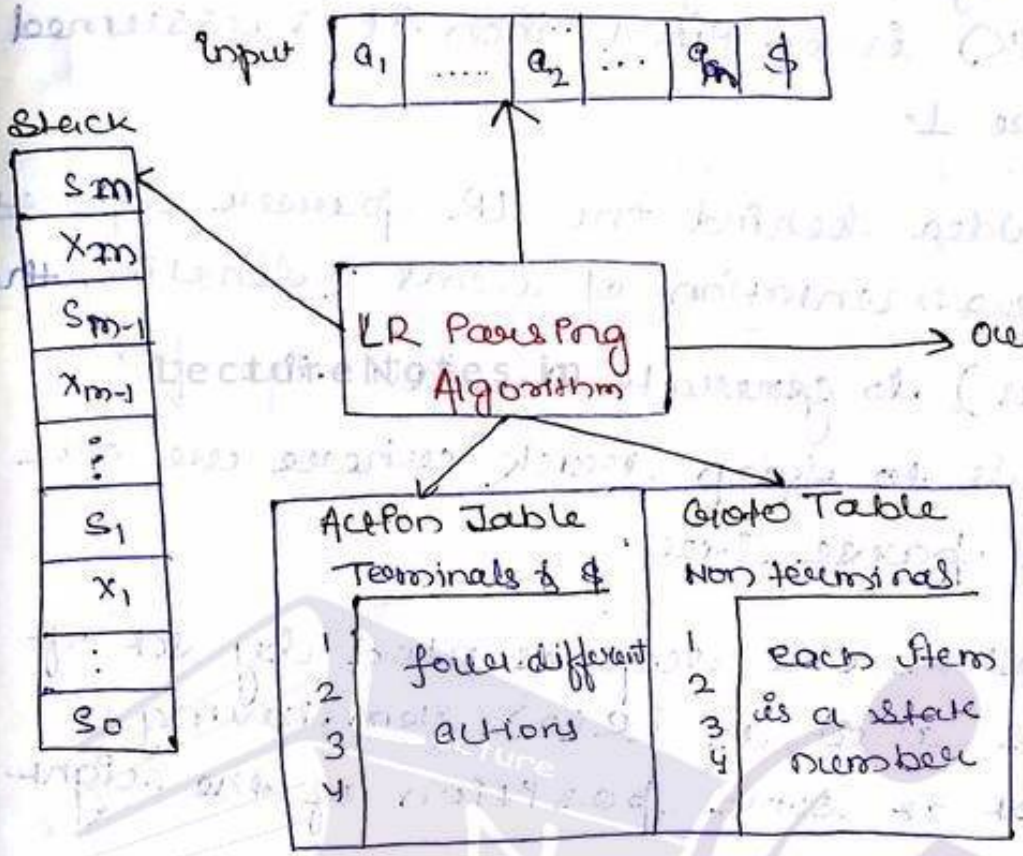
most commonly used LR parser

→ LALR : look ahead LR parser

→ SLR : Simple LR parser



LR Parsing Algorithm



→ The LR parsers are table driven parsers very much like non recursive LL parsers

→ The concept of LR(k) parsing defines scanning the input from left to right and constructing a rightmost derivation in reverse.

→ And K denotes the no. of input symbols of look ahead that are used in making parsing decisions.

→ The general case of K is taken as, K=0 or K=1.

→ Generally the case considered for making parsing decision is $K \leq 1$. Of (K) is omitted then it is assumed to be 1.

→ The idea behind the LR parser depends on representation of items (denoting the states) to generate a state the first step is to keep track wherever we are in a parse tree.

→ The states are represented by set of items called as LR(0) item having a dot at some position of the right side.

→ Eg: $A \rightarrow aBb$

Possible LR(0) items:

$A \rightarrow \cdot aBb$

$A \rightarrow a \cdot Bb$

$A \rightarrow aB \cdot b$

$A \rightarrow aBb \cdot$

→ They are also known as item in short.

If there is a production $A \rightarrow \epsilon$ then it generates only one item, set of items/ LR(0) items i.e. $A \rightarrow \cdot$

→ This representation indicates how much of a production we have seen at a given point in the parsing process.

→ To construct collection of sets of LR(0) items also known as canonical LR(0) collection is the basis for constructing SLR parsers.

We define an augmented grammar.

G' is G with a new production rule

$$S' \longrightarrow S$$

where S' → new start symbol.

NOTE: The new starting symbol indicates the parser when it should stop parsing and generates a message accept of the input.

Acceptance occurs only when the parser is about to reduce

$$S' \longrightarrow S.$$

→ We take the help of 2 types of functions.

- CLOSURE
- GOTO

(98) → CLOSURE

If I is a set of items of a grammar G then closure of I is the set of items constructed from I by using two rules:

Rule 1:

Initially add every item in I to closure (I).

Rule 2:

If $A \rightarrow \alpha . B\beta$ and there is a production rule $B \rightarrow \gamma$ then add item $B \rightarrow .\gamma$ to the closure (I) if not already there.

→ GOTO (I, X)

GOTO (I, X) is define to be the closure of a set of all items $[A \rightarrow \alpha X . \beta]$ such that $[A \rightarrow \alpha . X\beta]$ is in I.

which says that X can be non terminal or terminal but B is always a non terminal.

Actions of a LR parser

→ shift : Shifts the next input symbol and the state s onto the stack:

$(s_2 X_1 s_1 \dots X_m s_m, a_i, a_{i-1} \dots a_n s) \rightarrow$

$(s_2 X_1 s_1 \dots X_m s_m a_i s, a_{i-1} \dots a_n s)$

→ reduce : $A \rightarrow \beta$ (or an where r is a production number)

→ pop a β (= r) items from the stack.

→ then push A and s where s = goto [s_{m-r}, A]

$(s_2 X_1 s_1 \dots X_m s_m, a_i, a_{i-1} \dots a_n s) \rightarrow$

$(s_0 X_1 s_1 \dots X_{m-r} s_{m-r} A s, a_i \dots a_n s)$

→ output is the reducing production reduce $A \rightarrow \beta$

→ accept : Parsing successfully completed

→ Error : Parser detected an error (an empty entry in the action table)

(SLR) Parsing tables for Expression Grammar

(Pg - 240

$E \rightarrow E-T$

$E \rightarrow T$

T

$G_1 : E \rightarrow E+E \mid E*E \mid (E) \mid id$

⇓

ambiguity need to be removed

(1) $E' \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

Now we need to have augmented grammar G'.

$$E' \rightarrow E \quad = I$$

$$E \rightarrow E+T \quad = I$$

Next step is to build collection of a set of LR(0) items.

Closure (I) = closure ($\{E' \rightarrow .E\}$)

I_0 : { $E' \rightarrow .E$
$E \rightarrow .E-T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$ }

Initial state of DFA (I_0)

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

I_1: goto (I_0, E) = Closure $(\{E' \to .E\})$

$= \{E' \to E., E \to E. + T\}$

prefixes Vpable prefix

Vpable prefixes are set of right sentential forms that can appear on the stack of a shift reduce parser. It is always possible to add terminals to the end of Vpable prefixes to obtain a right sentential form.

I_2: goto (I_0, T) = closure $(\{E \to .T\})$

$= \{E \to T., T \to T. * F\}$

I_3: goto (I_0, F) = closure $(\{T \to .F\})$

$= \{T \to F.\}$

I_4: goto $(I_0, C) = \{F \to C. E),$

$E \to .E + T,$

$E \to .T,$

$T \to .T * F,$

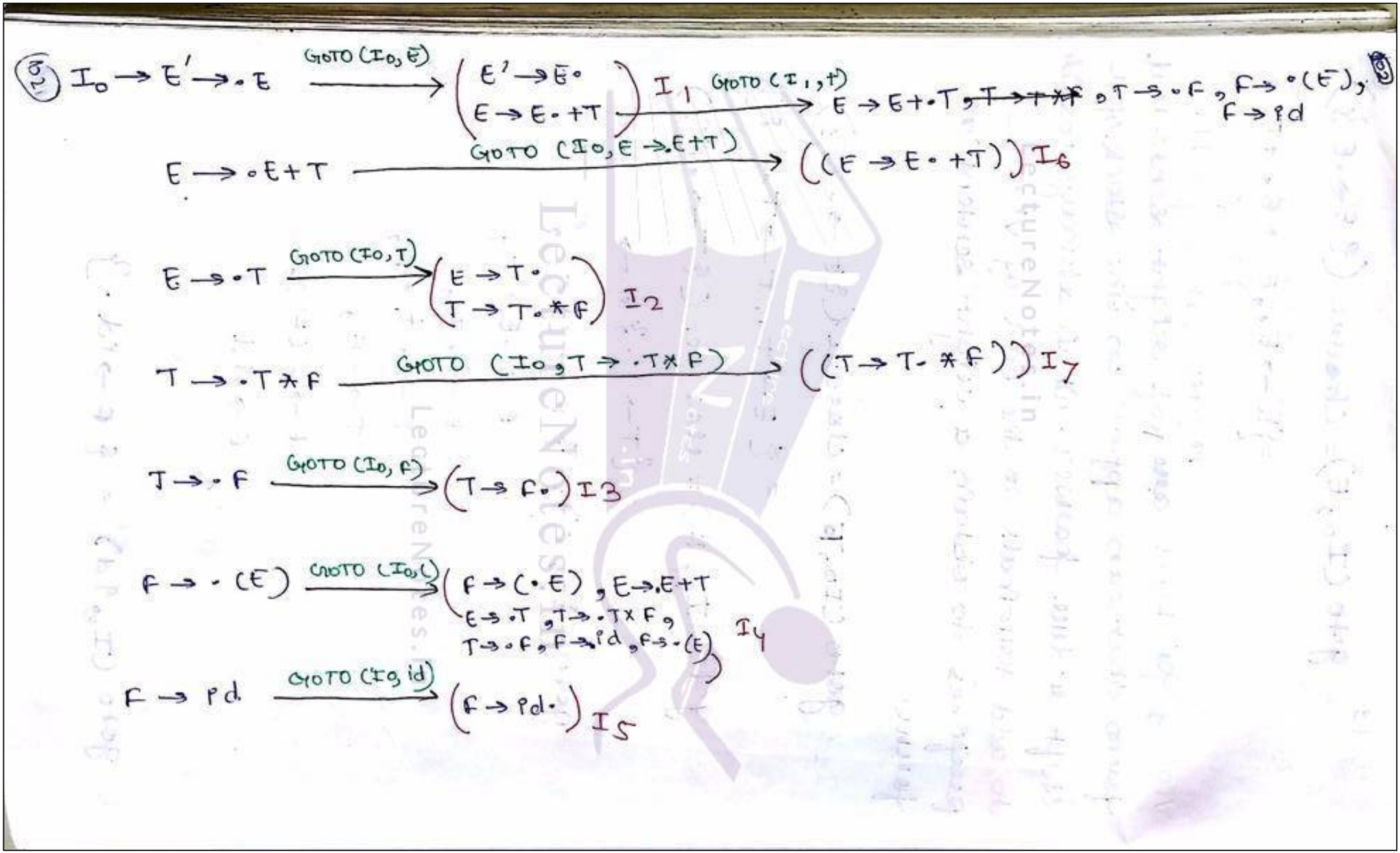
$T \to .F,$

$F \to .(E),$

$F \to .id$

$\}$

I_5: goto $(I_0, id) = \{F \to id.\}$



Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

SLR Parser

Contributed By:

Sibananda Achari

Trident Academy of Technology

SLR

Construction of the Canonical LR(0) collection.

→ C is { closure (S' → · S)) }

repeat the following until no more set of LR(0) items can be added to C

for each I in C and each grammar symbol X.

if goto(I, X) is not empty and not in C

add goto(I, X) to C

→ goto function is a DFA on the sets in C.

Augmented

C(E' → · E) } Kernel items

= E' → · E

I_0:

E → · E + T
E → · T
T → · T * F
T → · F
F → · id
F → · (E) } non-Kernel items

goto (I_0, E)

I_1: E' → E · } Kernel items
E → E · + T

goto (I_0, T)

I_2:

$E \to T\cdot$

$T \to T\cdot * F$ } Kernel Items

goto (I_0, F)

I_3:

$T \to F\cdot$ − KI

goto (I_0, id)

I_4:

$F \to id\cdot$ − KI

goto $(I_0, ()$

I_5:

$F \to (\cdot E)$ − KI

goto $(I_1, +)$

I_6:

$E \to E+\cdot T$ − KI

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot id$

$F \to \cdot (E)$ } non kernel Item

$A \to \cdot B\,B$

$B \to \gamma$

$B \to \cdot\gamma$

goto $(I_2, *)$

I_7:

$T \to T*\cdot F$ − KI

$F \to \cdot id$

$F \to \cdot (E)$ } non KI

goto (I_5, E)

I_8:

$F \rightarrow (E \cdot) - KI$

goto (I_6, T)

I_9:

$E \rightarrow E + T \cdot \rightarrow KI$

$T \rightarrow T \cdot * F, \rightarrow$ npn KI

goto (I_7, F)

I_{10}:

$T \rightarrow T * F \cdot - KI$

goto $(I_8,)$

I_{11}:

$F \rightarrow (E) \cdot \rightarrow KI$

Constructing SLR Parsing Table

→ Construct the canonical collection of sets of LR(0) items for G'. $C \leftarrow \{ I_0 \dots I_n \}$.

→ Create the parsing action table as follows.

- If a is a terminal $A \rightarrow \alpha \cdot a \beta$ in I_i and goto $(I_i, a) = I_j$, then action (i, a) is shift j.
- If $A \rightarrow \alpha \cdot$ is in I_i, then action $[i, a]$ is reduce $A \rightarrow \alpha$ for all a in FOLLOW(A), where $A \neq S'$.
- If $S' \rightarrow S \cdot$ is in I_i then action $[i, \$]$ is accept.

- If any conflicting actions are generated by the rules, the grammar is not SLR(1).

Parsing Table.

	action						goto		
State	id	+	*	(	)	$	E	T	F
0	S4	S6		S5			1	2	3
1		S6				accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S4			S5			8	2	3
5		r6	r6		r6	r6			
6	S4			S5				9	3
7	S4			S5					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

I Action [I0, $] = accept

A → α
A → α · aβ

II Action [0, id] = S4
I0 and goto(I0, id) = I4

III I_0 and goto $(I_0, ()) = I_5$

Action $[0, (] = S5$

Reduction

$A \to \alpha.$

$A \neq E'$

I $E \to T.$

I_2 ~~and Action $[2, I] =$~~

Follow $(E) = (), \$)$

Action $[2,)] = r2$

Action $[2, \$] = r2$

II $T \to F.$

Follow $(T) = (), +, \$)$

Action $[3, +] = r4$

Action $[3,)] = r4$

Action $[3, \$] = r4$

SLR (1) Grammar

→ An LR parser using SLR (1) parsing tables for a grammar G is called as the SLR (1) parser for G.

→ If a grammar G has an SLR (1) parsing table, it is called SLR (1) grammar (or SLR grammar in short).

→ Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar because it may generate shift-reduce conflict or reduce-reduce conflict.

Pg-255
Eg-4.48.

Shift/reduce and reduce/reduce conflict

→ If a state doesn't know whether it will make a shift operation or reduction for a terminal, we say that there is a shift/reduce conflict.

→ If a state doesn't know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a reduce/reduce conflict.

→ If the SLR parsing table of a grammar G has a conflict we say that grammar is not SLR grammar.

Canonical LR Parser

→ A LR(1) item is

$$A \rightarrow \alpha \cdot \beta\, a$$

where a is the look-head of the LR(1) item.

(a is a terminal or end marker)

→ To avoid some of invalid reductions, the states need to carry more information.

→ Extra information is put into a state by including a terminal symbol as a second component in an item.

→ Such an object is called LR(1) item.

- 1 refers to the length of the second component.
- The lookahead has no effect in an item of the form [A → α . β, a], where β is not ∈.
- But an item of the form [A → α., a] calls for a reduction in A → α only if the next input symbol is a.
- The set of such a's will be a subset of FOLLOW(A), but it could be a proper subset.

→ closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I).
- If A → α . Bβ, a in closure(I) and B → γ is a production rule of G, then B → .γ, b will be in the closure(I) for each terminal b in FIRST(βa).

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Goto operation

If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I, X) is defined as follows:

If $A \to \alpha . X \beta, a$ in I
then every item in closure

Construction of the Canonical LR(1) collection

Algorithm

C is { closure ({S' → S. $, $}) }
repeat the following until no more set of LR(1) items can be added to C
for each I in C and each grammar symbol X
if goto(I, X) is not empty.

Calculate the FIRST(C), FOLLOW(S), FIRST(S), FOLLOW(C).

I_0 : $S' \to S, \$$ ⇒ $S' \to S, \$$ is of the form $A \to \alpha \cdot X\beta, a$ where β is ϵ & a, FIRST($\epsilon \$$) = FIRST($\$$) = $\$$

$S \to CC, \$$

$C \to cC, \$$

$C \to d$

LR(1) item (contd.) 13/02/14

→ When βa in the LR(1) item $A \to \alpha \cdot \beta, a)$ is not empty, the look-ahead does not have any effect.

→ When β is empty ($A \to \alpha \cdot, a$), we do the operation reduction by $A \to \alpha$ only if the next input symbol is a (not for any terminal in FOLLOW(A)).

Construction of the canonical LR(1) collection

Algorithm

C is {closure ({$S' \to \cdot S, \$$})}

repeat the followings until no more set of LR(1) items can be added to C.

for each I in C and each grammar symbol X

if goto(I, X) is not empty and not in C.

add goto(I, X) to C

goto function is a DFA on the sets in C.

Q also $S' \to S$

$S \to CC$

$C \to cC$

$C \to d$

$\Rightarrow$ I_0: closure ({ ($S' \to .S, \$$) }) =

$(S' \to .S, \$)$

$(S \to .CC, \$)$

$(C \to .cC, c/d)$

$(C \to .d, c/d)$

I_1: goto (I_0, S) =

$(S' \to S., \$)$

I_2: goto (I_0, C) =

$(S \to C.C, \$)$

$(C \to .cC, \$c/d)$

$(C \to .d, \$c/d)$

I_3: goto (I_0, C)

$C \to c.C, c/d$

$C \to .cC, c/d$

$C \to .d, c/d$

I_4: goto (I_0, d)

$C \longrightarrow d., c/d$

I_5: goto (I_2, C)

$S \longrightarrow CC., \$$

I_6 : goto (I_2, c)

$C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

I_7 : goto (I_2, d)

$C \rightarrow d \cdot, \$$

I_8 : goto (I_3, C)

$C \rightarrow cC \cdot, c/d$

I_9 : goto (I_6, C)

$C \rightarrow cC \cdot \$$

14/02/14

Look ahead LR Parser

LALR parser can be created from CLR parsing techniques based on the idea of combining the core states.

$I_3 \rightarrow$

$C \rightarrow c \cdot C, c/d$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

$I_6 \rightarrow$

$C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

$I_{36} \rightarrow C \rightarrow c \cdot C, c/d / \$$

$C \rightarrow \cdot cC, c/d/\$$

$C \rightarrow \cdot d, c/d/\$$

Core States

→ for each core (the 1st component) present among the set of LR(1) items, find all the sets having same core and replace the sets by there union.

I_4 -

$C \rightarrow d\cdot, c/d$

I_7 -

$C \rightarrow d\cdot, \$$

⇒ I_{47} -

$C \rightarrow d\cdot, c/d, \$$

I_8 -

$C \rightarrow cC\cdot, c/d$

I_9 -

$C \rightarrow cC\cdot, \$$

⇒ I_{89} -

$C \rightarrow cC\cdot, c/d, \$$

I → core state
w → look ahead symbol

In total we have 7 sets of states after LALR parsing which was before 10 states due to CLR parsing.

States	ACTION			GOTO	
	c	d	$	S	C
0	S36	S47		1	2
1			accept		
2	S36	S47			5
3,6	S36	S47		89	
47	r3	r3	r3		
89	r2	r2	r2		
			r1		

Error Recovery

* ACTION / GOTO

	c	d	$	S	C
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Error Recovery In Predictive Parsing

Contributed By:

Sibananda Achari

Trident Academy of Technology

Error Recovery in Predictive Parsing

Error Recovery first do the stack of a table driven predictive parser. Since, it makes explicit the terminals and non terminals that the parser hopes to match with the remainder of the input. An error is detected during predictive parsing when the terminal on the top of the stack does not match the next input symbol or when the non-terminal A is on the top of stack, a terminal 'a' is the next input symbol and in the parsing table M[A, a] is empty.

LL Parsing Technique

⇒ To handle error detected, the symbols in the first and follow are used as the synchronizing tokens and for the empty cells the tokens are represented as synch (synchronizing tokens).

If the parser looks up entry M[A, a] and finds that it is blank then the input symbol 'a' is skipped.

→ If the entry is synch then the non terminal on the TOS is popped to resume parsing.

→ If a token on the TOS does not match the input symbol then we pop the token from the stack.

	FIRST()	FOLLOW()
E	(, id	), synch, $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	*, +,), $

	id	+	*	(	)	$
E	E → TE'			E → TE'	synch	synch
E'		E' → +TE'			E' → ∈	E' → ∈
T	T → FT'	synch		T → FT'	synch	synch
T'		T' → ∈	T' → * FT'		T' → ∈	T' → ∈
F	F → (id)	synch	synch	F → (E)	synch	synch

Stack	Input	Remark
E$	)id * + id $	M[E,)] = empty, skip
E$	id * + id $	
TE'$	id * + id $	
FT'E'$	id * + id $	F → id
T'E'$	* + id $	
*FT'E'$	* + id $	
FT'E'$	+ id $	
		M[F, +] = synch
T'E'$		F popped
E'$	+ id $	
+TE'$	+ id $	
TE'$	+ id $	
FT'E'$	id $	
T'E'$	id $	F → id
E'$	$	
$	$	

17/02/14

LR Error Recovery Techniques

→ An LR Parser will detect error when it consults the parsing action table and finds an error entry.

→ All empty entries in the action table are error entries.

→ Errors are never detected by consulting the goto table.

→ An LR parser will announce error as soon as there is no valid continuation for the scanned input portion.

→ A CLR parser will reduce after announcing an error.

→ However, SLR and LALR will make several reductions before announcing an error.

→ All the LR parsers will never shift an erroneous input symbol ~~however~~ onto the stack.

→ There are 2 modes of recovery -

- Panic mode recovery
- Phrase level recovery

Panic mode recovery:

→ It scan down the stack until a state S with goto on a particular non-terminal is found.

goto [S, A] = action

→ Discard 0 or more input symbols until a symbol 'b' is found in the FOLLOW(A).

→ The parser stack non-terminal A and the state goto [s, A] and it resumes normal parsing.

Phrase Level Recovery:

→ Each entry in the action table is marked with specific error routine.

→ An error routine inserts or deletes symbols onto the stack or input for the following cases -

* Missing Parenthesis - missing left / right parenthesis
* Missing operators
* Missing operands.

Error Recovery in Operator Precedence Pars

→ There are 2 points in parsing process at which an operator precedence parser can discover syntactic errors.

- If no precedence relation holds between the terminal on the top of the stack and between the current input.
- If a handle is been found but there is no production with the handle as a right side.

Handling Error Reductions:

→ The error checker for reductions need to check the following:

- If +, -, *, / is reduced, it checks that non terminals appear on both the sides. If not, it issues a message missing operand.
- If id is reduced, it checks that there is no non terminal to the right or left. If there is, it generates missing operator.
- If parenthesis is reduced, it checks that there is a non terminal between parenthesis. If not, it can raise no expression inside parenthesis.

YACC (Yet Another Compiler Compiler)

→ It is a LALR parser generator.

→ It takes ambiguous grammar and resolves conflict.

Reduce - Reduce Conflict

→ By choosing conflicting productions, listed top in the YACC specification.

Shift / Reduce Conflict

→ It is resolved in favour of shift.

→ YACC resolves shift/reduce conflict by attaching precedence and associativity to each production involved in a conflict as well as to each terminal involved in conflict.

→ It must choose between shifting each input symbol and reducing by production $A \longrightarrow \alpha$.

→ YACC reduces in 2 cases -

Case-I

If the precedence of the production is greater than that of the input symbol a.

Case-II

If the precedences are same and the associativity is left. Otherwise, there is a shift.

→ Normally, the precedence of the production is taken to be same as that of its right most terminal.

Eg $E \rightarrow E+E \mid E*E \mid (E) \mid id$

To construct the parsing table we need to augment the grammar.

So the augmented grammar is -

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

$E' \rightarrow E$

$E \rightarrow E+E \mid E*E \mid (E) \mid id$

I_0 :

$E' \rightarrow .E$

$E \rightarrow .E+E$

$E \rightarrow .E*E$

$E \rightarrow .(E)$

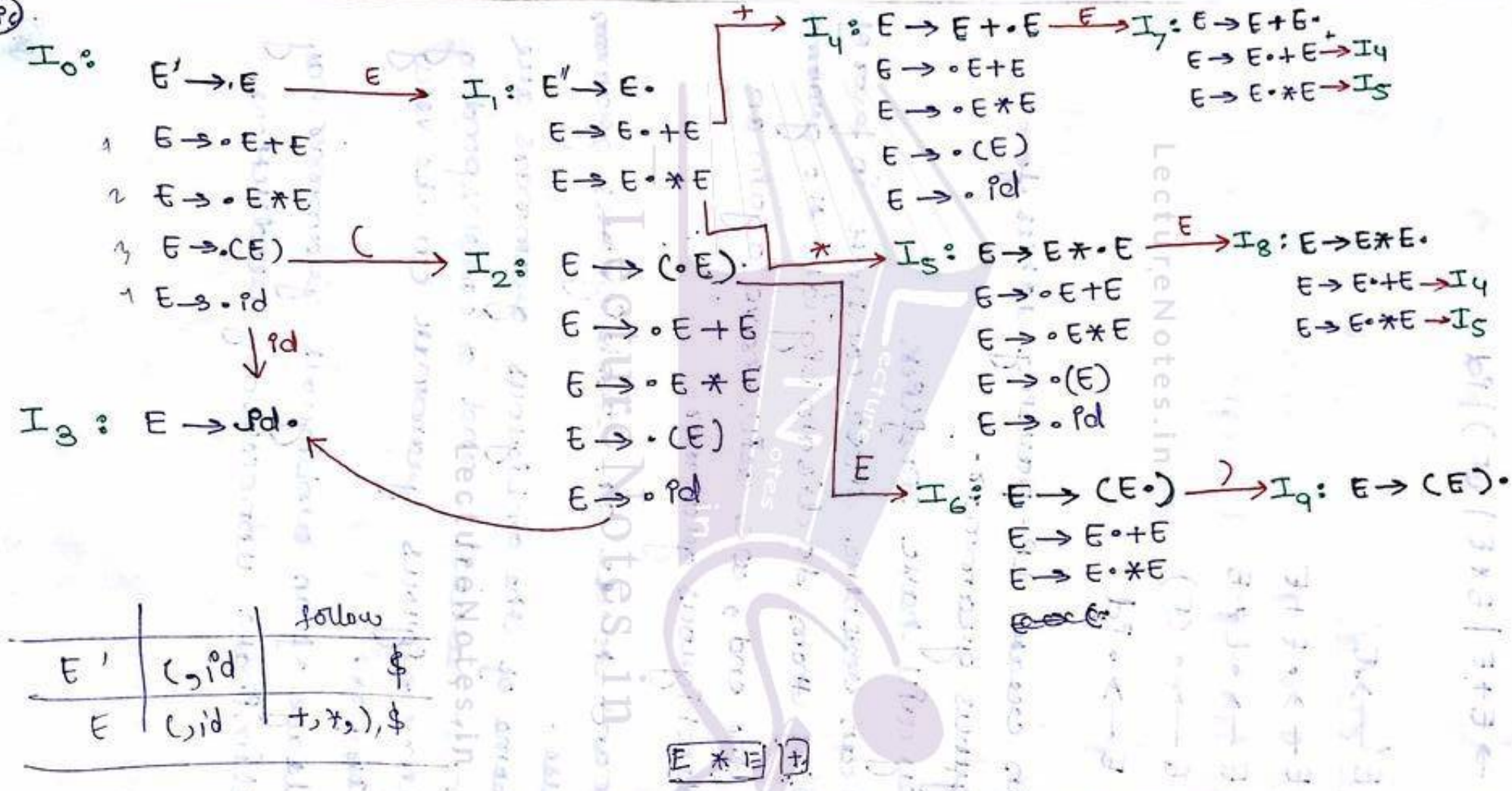
$E \rightarrow .id$

→ We can create LR-parsing tables for ambiguous grammar.

- They will have conflicts
- We can resolve these conflicts in favor of one of them to disambiguate the grammar.
- At the end, we will have again an unambiguous grammar.

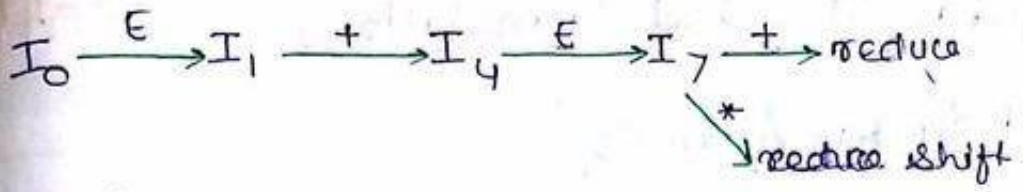
→ We want to use an ambiguous grammar because -

- Some of the ambiguous grammars are much natural, and a corresponding unambiguous grammar can be very complex.
- Usage of an ambiguous grammar may eliminate unnecessary reductions.



		follow
E'	(, id	$
E	(, id	+, *,), $

E * (



→ when current token is +

- shift → + is right-associative
- reduce → + is left-associative

→ when current token is *

- shift → * has higher precedence than +
- reduce → + has higher precedence than *

SLR Parsing Table for Ambiguous Grammar

	Action						GOTO
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			acc	
2	S3			S2			6
3		r4	r4		r4	r4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5		S9		
7		r1	S5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Syntax Directed Translation And Symbol Table

Contributed By:

Sibananda Achari

Trident Academy of Technology

Ch- Syntax Directed Translation

18/02/14

Beyond Syntax Analysis

→ Semantic analysis is the phase where we collect information about the type of expressi and check for type related errors.

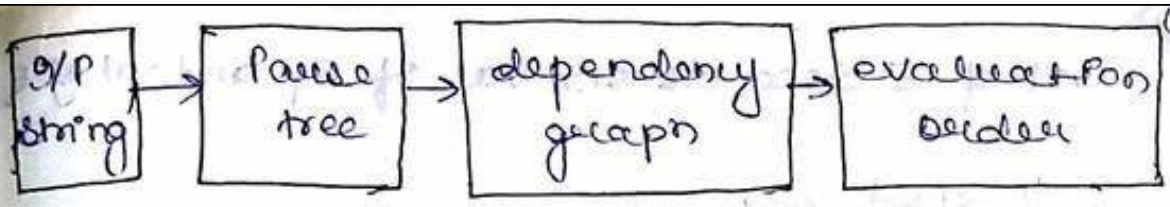
→ The more information we collect at comp time, the less overhead we had at runn time.

→ Collecting type information may involve "computations".

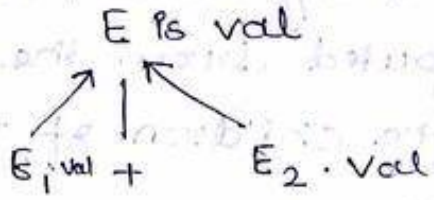
- what is the type of x+y given the types of x and y?

→ Tool: attribute grammars

- CFG
- Each grammar symbol has associated attributes.
- The grammar is augmented by rules (semantic actions) that specify how the values of attributes are computed from other attributes, which is called semantic rule.
- The process of using semantic actions to evaluate attributes is called syntax directed translation.



Eg: $E = E_1 + E_2$



E value is dependent on E_1 and E_2.

Syntax directed definitions-

→ It is a generalization of a CFG in which each grammar symbol has an associated set of attributes.

- Synthesized attribute
- Inherited attribute

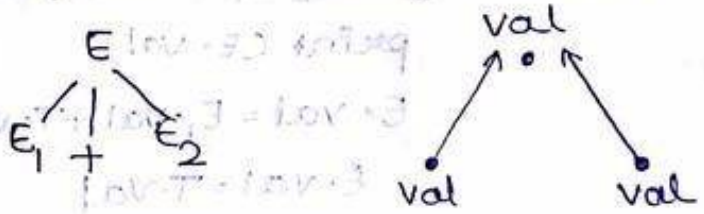
→ An attribute can represent a string, a no., a type, a memory location or content.

→ The value of an attribute at a parse tree node is defined by a semantic rule.

Eg $E \to E+E$

$E \to E_1+E_2$

Parse tree - Dependency graph-



val of E depends on val of E_1 and val of E_2

Synthesized attributes

→ The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

→ A syntax directed definition that uses synthesized attributes is said to be an S-attributed definition.

→ A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root.

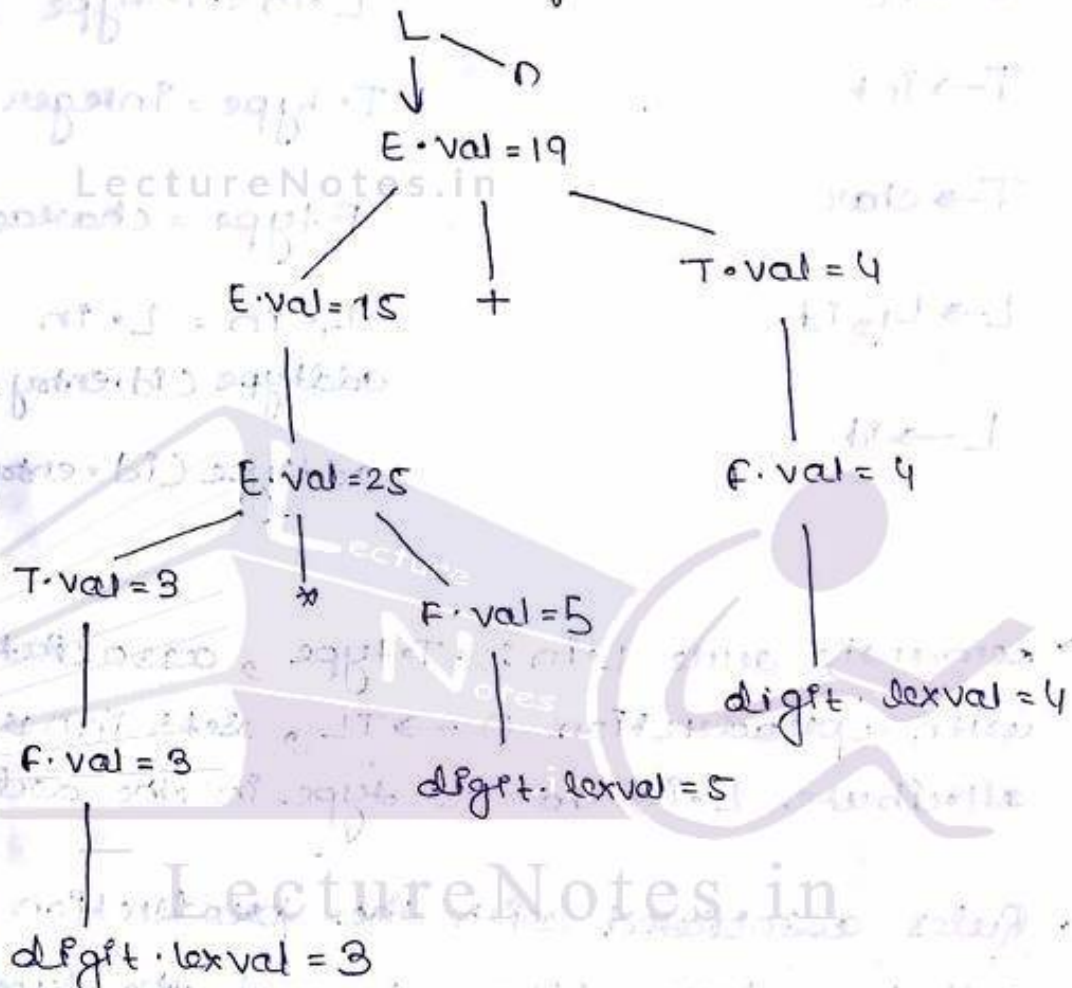
→ A parse tree showing the value (s) of its attribute (s) is called an annotated parse tree.

Syntax directed definition of a simple desk calculator

Production	Semantic rule
$L \to En$	print (E.val)
$E \to E_1 + T$	E.val = E_1.val + T.val
$E \to T$	E.val = T.val
$T \to T_1 * F$	T.val = T_1.val * F.val

$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

Annotated parse tree for 3*5+4n



Inherited attributes

→ The value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node.

→ An inherited value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of that node.

→ It is used to express the dependence of a programming language construct.

Syntax-directed definition of a grammar of declarations

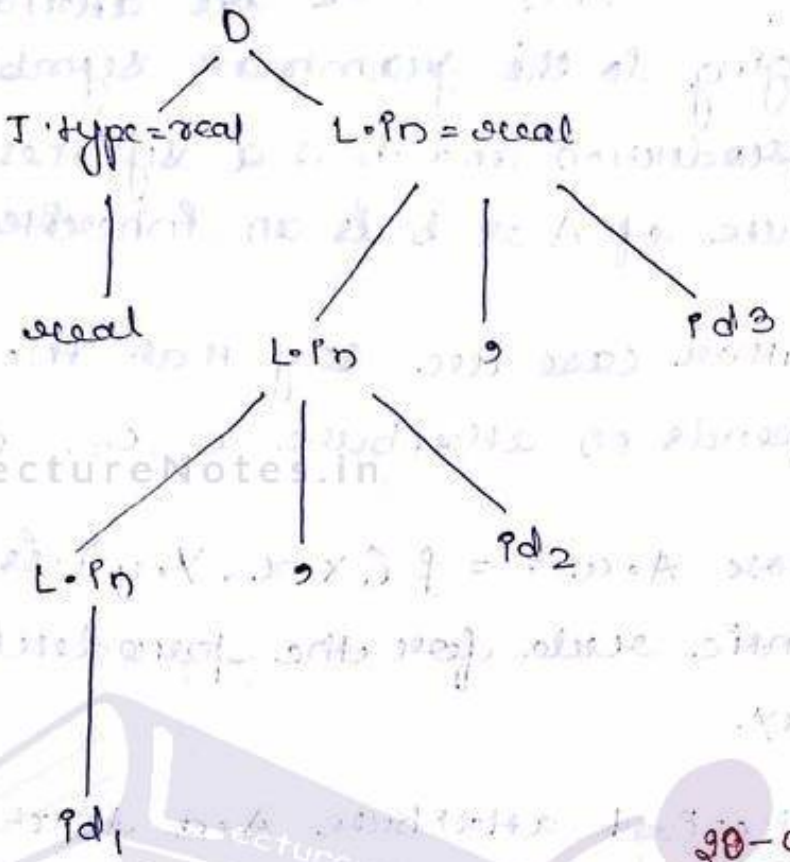
Production	Semantic rule
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow char$	$T.type = character$
$L \rightarrow L_1, id$	$L_1.in = L.in$ addtype (id.entry, L.in)
$L \rightarrow id$	addtype (id.entry, L.in)

→ Semantic rule $L.in := T.type$, associated with production $D \rightarrow TL$, sets inherited attribute $L.in$ to the type in the declaration.

→ Rules associated with the production for L call procedure addtype to add the type of each identifier to its entry in the symbol table.

→ The rules then pass this type down the parse tree using the inherited attribute $L.in$.

Annotated parse tree for real id1, id2, id3



20-02-14

① Dependency Graphs

→ The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.

→ If an attribute b at a node in a parse tree depends on a attribute c_1 then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.

→ If a production $A \to \alpha$ has associated with it a set of semantic rules of the form -

$$b := f(c_1, c_2 \cdots c_k)$$

20/02/14

where $c_1, c_2 \ldots c_k$ are attributes belonging to the grammar symbols of the production and b is a synthesized attribute of A or b is an inherited attribute.

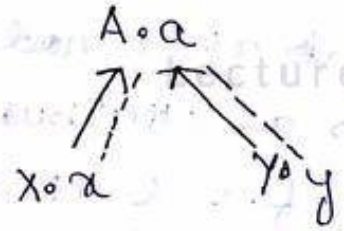
→ In either case, we say that the attribute b depends on attribute $c_1, c_2 \ldots c_k$;

→ Suppose $A.a := f(X.x, Y.y)$ is a semantic rule for the production $A \rightarrow XY$.

→ Synthesized attribute A.a that depends on the attributes X.x and Y.y.

→ The dependancy among node is the solid lines.

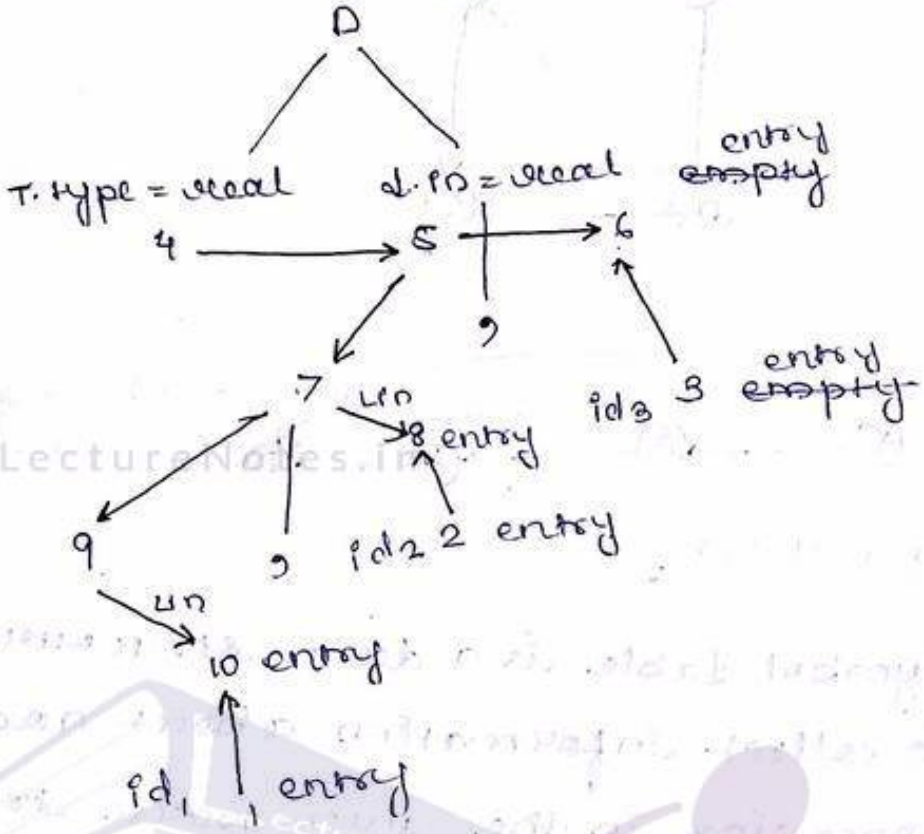
→ The parse tree is represented by dotted lines.



Applications of SDT –

- Disambiguate overloaded operators
- Type checking
- Uniqueness checking
- Name checking
- Type coercion

Dependency Graphs for real id1, id2, id3



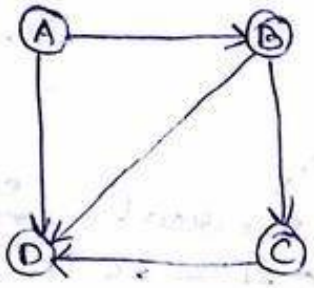
Evaluation Order:

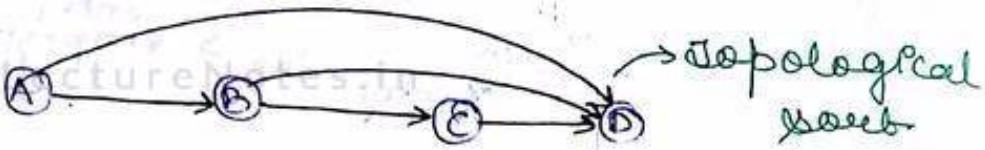
→ Evaluation order of semantic rules is obtained by a topological sort of a directed acyclic graph.

→ In the topological sort, the dependent attributes $c_1, c_2 \ldots c_k$ in a semantic rule $b := f(c_1, c_2 \ldots c_k)$ are available at a node before f is evaluated.

→ Evaluation of the semantic rules in this order yields the translation of the input string.

Eg-





Symbol Table

→ Symbol Table is a data structure used to collect information about nodes appearing in the given parse tree or information about names appearing in the source program.

→ It keeps the track about the scope or binding information about names.

→ Each entry in the symbol table has a pair of information.

name	description type
id	integer

→ Information consists of attributes such as datatype, location etc.

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

→ Whenever a name is encountered it is checked in the symbol table to see if edges already present of information is not there, a new entry is created.

int id1
char c

name	description type
if	keyword if
id1	Integer
c	character

→ In some cases, the symbol table record is created by the lexical analyser as soon as the name is encountered in the input and the attributes of the name are entered.

When the declarations are processed.

→ operations on a symbol table

- Determines whether a given name is in the table.
- Add a new name to the table.
- Delete a name from the table.
- Access information associated to a given name.
- Add new information for a given name.

(36)

→ There are various approaches to symbol table organisations.

- Linear list
- Search Tree
- Hash table

(136)

Linear List

→ It is the simplest approach in the symbol table organisation.

→ The new names are added to the table in order they arrive.

→ A name is searched for its existence linearly.

→ The average no. of comparisons required are proportional to the no. of entries in the symbol table.

→ It takes less space but more access time.

Search Tree

→ More efficient than linear trees.

→ It provides two links. left and right which point to the record in the search tree.

→ A new name is added at a proper position such that it can be access alphabetically.

→ The time of searching is proportional to $\log_2 n$.

Hash Table

→ It is a table of K pointers from 0 to K-1. that points to the symbol table to search a value ~~define~~ we find the hash value of the name ~~use~~ by applying suitable hash function.

→ If a record is non existent then the record for the name is created and added to the list.

The Scope of information in the symbol table

→ Each name posses a region of validity within the source program called the scope of the name.

→ The rules governing the scope of a name in a block structure language are as follows:

- a name declared within a block B is valid only within the block B.
- if block B_1 is nested under block B_2, then any name that is valid for B_2 is also valid for B_1.

- The rules become more complicated when a complex program is analysed.
- To handle such a situation multiple symbol table for each active block is maintained with the following information :

  * Each table is list of names and there associated attributes and the tables are organised on the stack.
  * Whenever a new block is entered, a new table is pushed onto the stack.
  * When a declaration is compiled the table on the stack is searched for the name.
  * If name is not found, it is inserted.
  * When a reference is translated, it is searched in all tables starting from the top

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Intermediate Code Generation

Contributed By:

Sibananda Achari

Trident Academy of Technology

Intermediate Code Generation

→ ICGs are m/c. machine independent codes but are close to machine instructions.

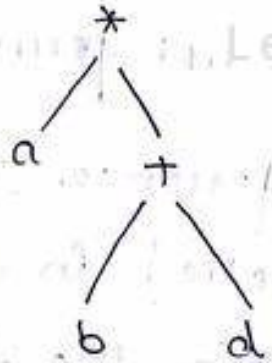
→ The given source program is converted to an equivalent program in an intermediate language by the [ICG].

The intermediate language can be syntax tree, postfix notation, 3-address code.

Syntax Tree

Syntax tree is a variant of parse tree where each leaf node represents an operand and each internal node represents an operator.

Consider a sentence. a * (b + d)



Semantic rules for the syntax tree-

$E \rightarrow E_1 \text{ op } E_2$
$E \rightarrow (E1)$
$E \rightarrow id$
} production rule

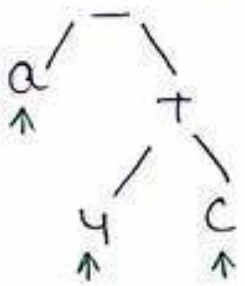
node (op, left, right)
leaf (id, entry)
} 2 common functions

$E.val := E_1.val \text{ op } E_2.val$
$E.val := E_1.val$
$E.val := \text{LEAF (id, entry)}$
} semantic rules

Q $a - 4 + c$

Define all the semantic rules / action / translation.

⇒



(P1) LEAF (id, entry − a)
(P2) LEAF (id, entry − c)
(P3) LEAF (number, val 4)

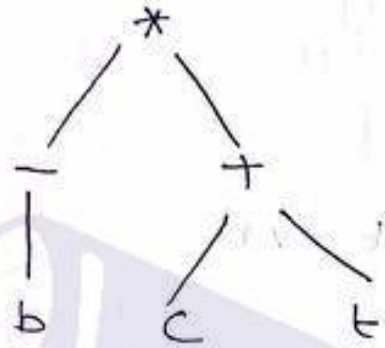
(P4) NODE (+, P3, P2)

(P5) NODE (−, P1, P4)

Q $-b*(c+t)$

Define translations.

⇒



(P1) UNARY (−, P2)

(P2) LEAF (id, entry-b)

(P3) LEAF (id, entry-c)

(P4) LEAF (id, entry-t)

(P5) NODE (+, P3, P4)

(P6) NODE (*, P1, P5)

Postfix Notation

Eg: 4 * 5 + 3

Convert the expression from infix to postfix.

⇒ ((4 * 5) + 3)

= ((45) * + 3)

= (45) * 3 +

= 45 * 3 +

Q $E \rightarrow E_1 \text{ op } E_2$

Postfix Notation - $E := E_1\ E_2 \circ op$

$E \circ val := E_1 \circ val \mid E_2 \circ val \mid op$

→ It is another useful form of intermediate code if the language is mostly expression.

→ $E \rightarrow E_1 \text{ op } E_2$

$E \rightarrow (E_1)$

$E \rightarrow id$

$E \circ val := E_1 \circ val \mid E_2 \circ val \mid op$

$E \circ val := E_1 \circ val$

$E \circ val := id$

3- Address Code

Usually contains three addresses, two for operands, one for result.

$x := y \text{ op } z$

x, y, z are names, constants or compiler-generated temporaries,

op stands for an operator such as fixed or floating point arithmetic operator.

Representation of ICG

There are 4 forms of representation:

- Quadruple
- Triple
- Indirect Triple
- Static Single Assignment (SSA)

Q (i) $x + y * z$

(ii) $A = -B * (C+D)$

$\Rightarrow$ $A = -B * (C+D)$

T1 := C + D

T2 := − B

T3 := T1 * T2

A := T3

$\Rightarrow$ $x + y * z$

T1: y * z

T2: x + T1

Types of Three - Address Statements

→ Assignment statements of the form: $x := y \text{ op } z$
where op is a binary arithmetic or logical operation;

→ Assignment statements of the form: $x := \text{op } y$
where op is a unary operation.
Essential unary operations include unary minus, logical negation, and shift operators;

→ Copy statements of the form: $x := y$
where the value of y is assigned to x;

→ The unconditional jump goto L. The 3-address statement with label L is the next to be executed.

→ Conditional jumps such as:

if x relop y goto L.

This instruction applies a relational operator (<, =, >, <=, etc.) to x and y, and executes the statements with label L next if x stands in relation relop to y;

→ Procedure Calls: call p, n and return values from functions:

(146) return y. Their typical use is the following:

```
param x1
param x2
..
param xn
call p, n
```

generated as part of a call of the function: P(x1, x2 ... xn). The integer n indicates the number of actual parameters.

24/02/14

→ Indexed assignments of the form:

x := y[i]

x[i] := y

The first one sets x to the value in the location i memory units beyond y. The statement x[i] := y sets the content of the location i units beyond x to the value of y;

→ Address and pointer assignments:

x := &y

x := *y

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Implementation of TAC Expression

Eg $a = -b * (c+d)$

0 $t1 := c+d$
1 $t2 := -b$
2 $t3 := t1 * t2$
3 $a := t3$

Quad representation:

Index	Op	Arg1	Arg2	Res
(0)	+	c	d	t_1
(1)	uminus	b		t_2
(2)	*	t1	t2	t_3
(3)	=	$a t_3$	~~t3~~	a

Triple representation:

Index	Op	Arg 1	Arg 2
(0)	+	c	d
(1)	uminus	b	
(2)	*	(0)	(1)
(3)	=	(2)	a

It avoids putting temporary names in symbol table.

Indirect triple representation:

It consists of 1 array of pointers to array of triples

(0)	16
(1)	17
(2)	18
(3)	19

	op	Arg1	Arg2
(16)	+	c	d
(17)	Uminus	b	
(18)	*	(0)	(1)
(19)	=	(2)	a

Q Generate TAC for the given expression

$$(X) = (-a * b + -a * b) \rightarrow E$$

(X) → id

$\Rightarrow$

$t_1 := -a$

$t_2 := t_1 * b$

$t_3 := \text{Uminus } a$

$t_4 := t_3 * b$

$t_5 := t_2 + t_4$

$x := t_5$

Converting the above into postfix form

$S \rightarrow (id = E) \rightarrow$ starting root

(S) $X = E$

id.place = E.place
value value

E.code
TA statements

S -> id = E → Production

Semantic rules -
S.code := E.code || gen (id.place ':=' E.place)

→ E.place → the name that will hold the value of E

→ E.code → the sequence of three-address statements evaluating E.

→ gen → generates the three address code. Expression appearing instead of variables when passed to gen and quoted operands like '+'.

→ || → symbol is used for concatenation, rules are added to show the relationship bet^n LHS & RHS. Such relationship is called Semantic rules

Production	Semantic Rules
$E \rightarrow E_1 + E_2$	E.place := newtemp; E.code := E1.code || E2.code || gen (E.place ':=' 'E_1.place + E2.place')

→ newtemp → generates temporary names.

$E \rightarrow E_1 * E_2$	E.place := newtemp; E.code = E1.code || E2.code || gen (E.place ':=' E1.place '*' E2.place)
$E \rightarrow -E_1$	E.place := newtemp; E.code = E1.code | gen(E.place ':=' 'uminus' E1.place)
$E \rightarrow (E1)$	E.place := E1.place; E.code = E1.code
$E \rightarrow id$	E.place := id.place; E.code := ''

→ To distinguish between expression nodes on RHS & LHS the expression nodes on RHS is represented with E1.

TAC for Boolean Expressions

→ Boolean expressions are composed of the Boolean operators (and, or & not)

→ Boolean expressions have 2 primary purposes

- to compute logical values
- to be used as conditional expressions

→ Relational expressions are of the form:

A relop B where A & B are arithmetic expressions. relop can be of the following types: <, >, <=, >=, ==, != etc.

→ Two methods are there for translating boolean functions -

- encode true & false numerically & evaluate a boolean expression.
- Implement boolean expression by a flow of control.

→ Consider a boolean expression generated by the following grammar.

E → E or E | E and E | not E | (E) | id relop id | true | false

→ The precedence of operator is as follows not as highest, then and, then or as lowest.

→ Boolean expression have different translations depending on the context.

* Compute logical values

Eg- a or b and not c

a OR b AND NOT c

JA start :-

t1 := NOT c

t2 := b AND t1

t3 := a OR t2

Semantic Rules / Translation Rules

production rule	Semantic rule
E → E1 or E2	{ E.place := newtemp; emit (E.place ':=' E1.place 'or' E2.place) }
E → E1 and E2	{ E.place := newtemp; emit (E.place ':=' E1.place 'and' E2.place) }
E → not E1	{ E.place = newtemp; emit (E.place ':=' 'not' E1.place) }
E → (E1)	{ E.place := E1.place }

$E \rightarrow id_2$ relop id_2	{E.place = newtemp; emit ("if" id_2.place relop id_2.place "goto" nextstat +3}; emit (E.place ":=" "0"); emit ("goto" nextstat +2); emit (E.place ":=" "1")}
E → true	{E.place := newtemp; emit (E.place ":=" "1")}.
E → false	{E.place := newtemp; emit (E.place ":=" "0")}

eg: ~~a ← b~~

For the above given eg:- a OR b AND NOT c

t1 = NOT c

if b AND t1
goto L true

L1 : if a OR t1
goto L true

L false : — —

L true : — —

Eg : a < b or c < d and e < f

```
        if a < b goto Ltrue
        goto L1
L1 :    if c < d goto L2true
        goto Lfalse
L2 :    if e < f goto Ltrue
        goto Lfalse
Lfalse :
Ltrue :
```

short circuit code

TAC for Boolean Expression

```
100: if a < b goto 103
101: t1 := 0
102: goto 104
103: t1 := 1
104: if c < d goto 107
105: t2 := 0
106: goto 108
107: t2 := 1
108: if e < f goto 111
109: t3 := 0
110: goto 112
111: t3 := 1
112: t4 := t2 and t3
```

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Short - Circuit :

In short - circuit or jumping code, the boolean operators AND, OR, NOT translates into jumps. The operators themselves do not appear in the code. Instead the value of the expression is represented by the position.

Eg - if (X < 100 || X > 200 && X != Y)
X = 0;

Short - circuit code :

```
iffalse (X < 100) goto L2
iffalse  X > 200  goto L1
L1:
L2:
```

Find the lines replaced and missed in the above short circuit code.

⇒
```
    if X < 100 goto Ltrue
    goto L1
L1: if X > 200 goto L2
    goto Lfalse
L2: if X != Y goto Ltrue
    goto Lfalse
Lfalse :
Ltrue :
```

SSA (Static Single Assignment)

25/02/14

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

All the assignments have distinct names

SSA –

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

→ Static Single Assignment is an intermediate representation that facilitates certain code optimization to distinct aspects. Distinguish SSA from 3-address code.

→ The first is that all assignments in SSA are two variables with distinct names. Hence the term static single assignment.

→ The same variable may be defined in two different control flow paths in a program.

eg: If (flag)
x = -1;
else
x = 1;
y = x * a;

⇒ if (flag)
$x_1 = -1$;
else
$x_2 = 1$;
$y = \phi(x_1, x_2) * a$;

→ The ϕ function returns the value of its argument that corresponds to the control flow path that was taken to get to the assignment.

Type Expressions

→ A basic type is a type expression.

→ A typename is a type expression.

→ A type expression can be formed by applying the array type constructor to a number and a type expression.

eg array (number, type expression)
int a[2];

$\Rightarrow$ a (2, Integer)

→ The type expression can be formed by using the type constructor for function types.

→ Type expression may contain variables whose values are type expression.

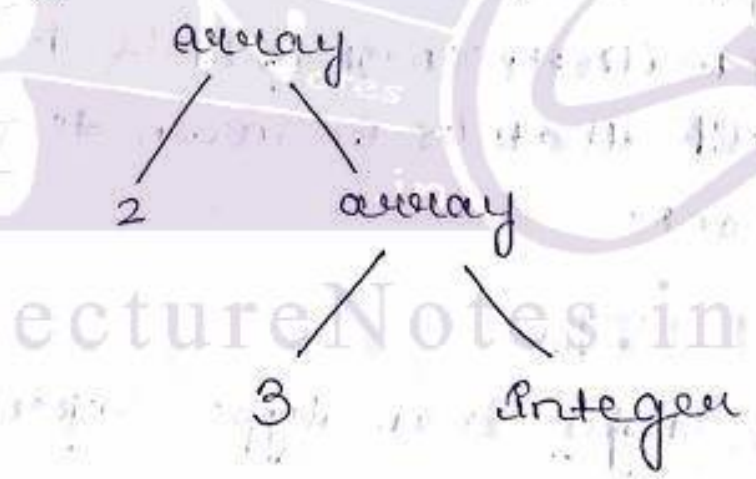
Eg: Int [2] [3];

$\Rightarrow$ array (2, type expression);

array (2, array (3, Integer));

↓

Syntax tree



Production rule:

$D \rightarrow T\ id\ ; \mid D \mid \epsilon$

name of the record

$B \rightarrow Int \mid float$

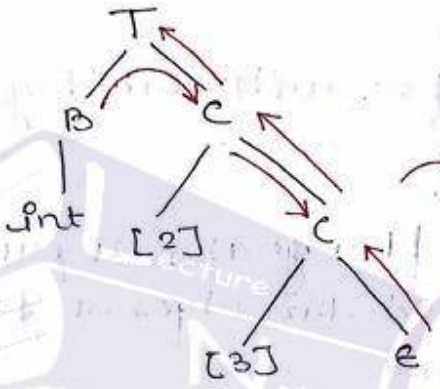
$T \rightarrow B\ C$

$C \rightarrow \epsilon \mid [num]\ C$

Eg: int [2][3];

D ⇒ T id
⇒ B C id
⇒ int C id
⇒ int [num] C id
⇒ int [num] [num] id
⇒ int [2] [3] id ;

Eg: Syntax tree



→ Dependency graph

Type Expression

$\{ t = B.type, \; w = B.width \}$ T → BC

$\{ B.type = integer, \; B.width = 4 \}$ B → int

$\{ B.type = float, \; B.width = 8 \}$ B → float

$\{ C.type = t, \; C.width = w \}$ C → ϵ

$\{ array(number.value, C1.type), \; C.width = number.value * C1.width \}$ C → ϵ | [num] C

Eg: C2, integer) w = 2 * 4 = 8

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Compiler Design

Topic:

Code Generation And Code Optimization

Contributed By:

Sibananda Achari

Trident Academy of Technology

Basic Blocks And Flow Graphs

→ **Flow Graph:**

→ A graph representation of a 3-Address statements.

→ It is useful for understanding code generation algorithm.

→ Nodes in the flow graph represent computation and the edges represent the flow of control.

→ Flow graphs are useful to find inner loops where program is ~~acce~~ expected to spent most of its time.

→ The basic blocks can be generated from the flow graph in the following manner:

Step-I: Partition the Intermediate code into basic blocks.

Basic blocks are maximal sequence

3-address instructions with the following (161)
properties.

→ The flow of control can enter the basic block through the first instruction in the block. i.e. there are no jumps into the middle of the block.

→ Control will leave the block at the end. (last instruction) without halt or possibility of branching.

Step-II: The basic blocks become the nodes of the flow graph whose edges indicates which block can follow which other blocks.

Algorithm 1: Partition into basic blocks.

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

→ We first determine the set of leaders, the first statement of basic blocks.

The rules we use are as follows:

- The 1st statement is a leader.
- Any statement that is the target of a conditional / unconditional goto is a leader. (163)

Code Optimization

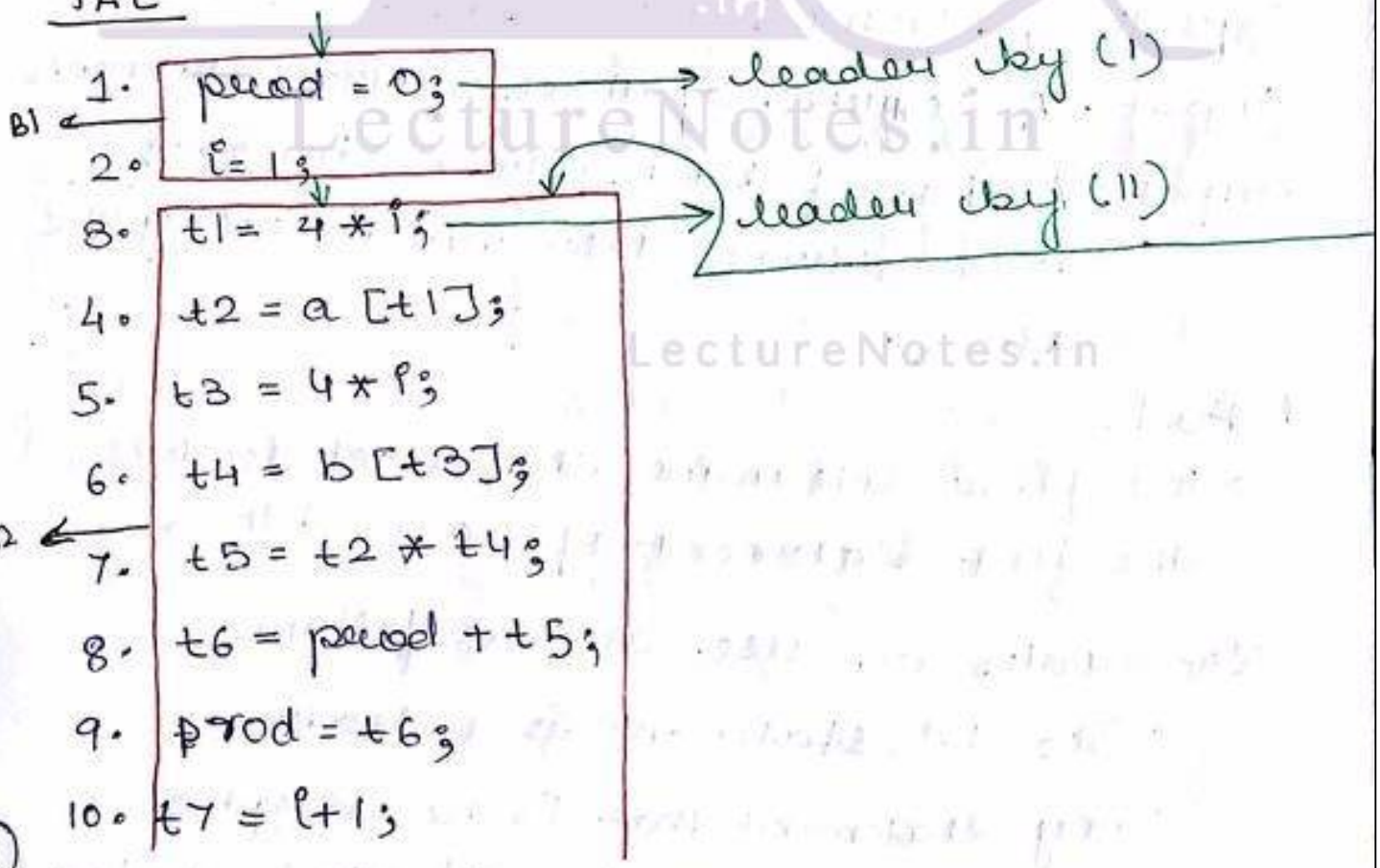
Q

```
begin
  prod = 0;
  i = 1;
  do begin
    prod = prod + a[i] * b[i];
    i = i+1;
  end
    while (i<=20)
end
```

If there is a statement involving do-while, while, for then the statement need to be change into if statement with conditional jump.

The 3-address code for the above given code is —

JAC



```
11. i = t7;
12. if (i<=20) goto 3
```

Now we need to identify the leaders —

preread = 0; leader by (1)

t1 = 4 * i; leader by (11)

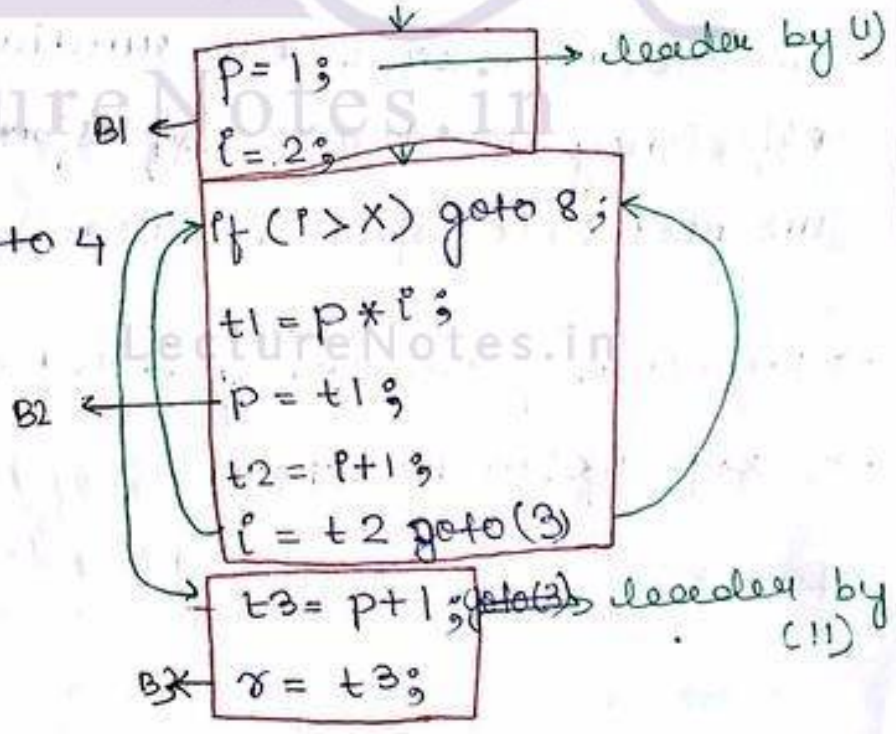
Q

```
exp(x)
{
    int p = 1;
    for (i = 2; i<=x; i++)
    {
        p = p * i;
    }
    r = p + 1;
}
```

Convert it into TAC.

⇒

1. exp(x)
2. p = 1
3. if(i<= x) goto 4
4.
5.



loop = { B2 }

TAC -

```
exp(x)
{
1.  p = 1;
2.  i = 2 goto 7
3.  t1 = p * i;
4.  p = t1;
5.  t2 = i + 1;
6.  i = t2;
7.  if (i <= x) goto 3
8.  t3 = p + 1;
9.  r = t3;
```

Principle Sources Of Optimisation

→ We want to save the runtime and to improve efficiency in terms of space and time so we need code optimization.

→ Function preserving transformation

→ loop optimization

Function preserving

→ Common subexpression elimination

→ Copy propagation

→ Dead code elimination

→ constant folding

Dead Code Elimination:

→ If a piece of code define at a particular program point is not used anywhere in the program then that code is called as dead code and it should be removed from the program.

Eg
$c = a * b$
$x = q$ → Dead code
:
$d = a * b$

⇒ $c = a * b$, $d = a * b$ → Common subexpression

⇒ $t1 = a * b$
$c = t1$
$d = t1$

Common Subexpression Elimination:

An expression need not be evaluated if it was previously computed and the values of the variables in this expression have not changed since earlier computation.

eg:- $a = b * c$; → common subexpression
:
$d = b * c + x - y$;

$\Rightarrow t1 = b * c$

$a = t1$

$d = a + x - y \rightarrow$ If a not written, then $a = t1$ or, $d = t1 + x - y$ would become a dead code

If in the question, dead code correction hasn't been mentioned then we could have written and statement.

28/02/14

Compile time evaluation

69

```
x = 12.4;
:
y = x/2.3; → execution time action
```

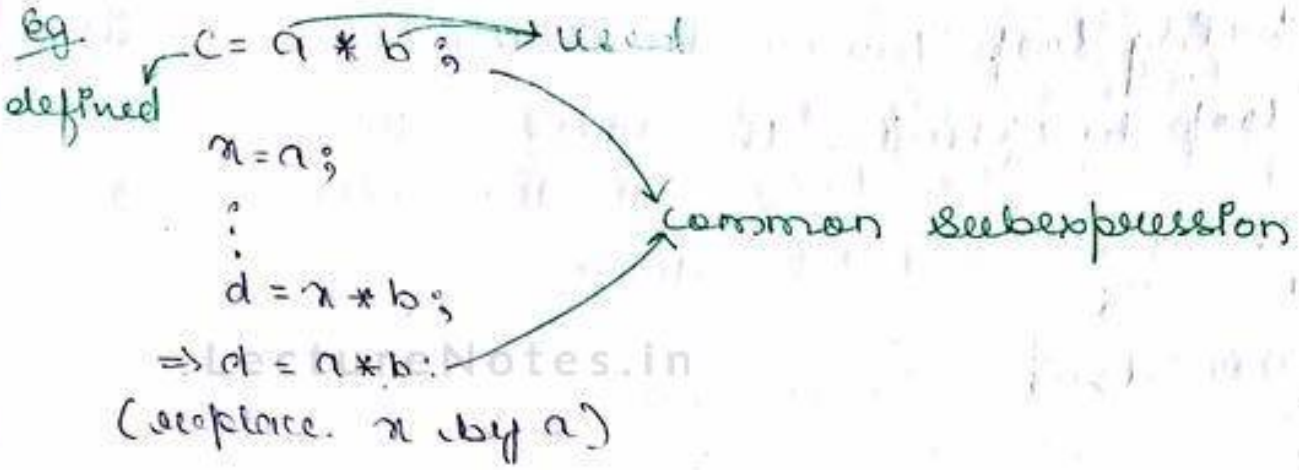
We can improve the efficiency of the program by shifting execution time actions to compile time action.

$y = 12.4/12.3 \rightarrow$ compile time action

We can evaluate an expression by a single value known as constant folding.

If a variable is assigned a constant value and used in an expression without changing its assigned value then some portion of the expression can be evaluated using constant value.

Variable Propagation



If a variable is assigned to another variable, we use one in place of another. This will be useful to carry out other optimizations that otherwise isn't possible.

If we replace x by a, then a*b and x*b will be identified as common subexpression and the common subexpressions need to be removed as follows:

```
t1 = a * b
c = t1
:
d = t1
```

Code Motion

We can improve the execution time of the program by reducing the evaluation of frequency of expressions.

Evaluation of expression is moved from one part of the program to another in such a way that it is evaluated less frequently.

loops are executed several times we can bring loop invariant statement out of the loop to optimize the code.

```
a = 200;
while (a>0)
{
   b=a+y;  → loop invariant statement
   if (a%b==0)
   printf ("%d", a)
}
```

⇓

```
a = 200;
b=a+y;
while (a>0)
{
   if (a%b==0)
   printf ("%d", a);
}
```

Induction Variable & Strength Reduction

An induction variable may be defined as an integer scalar variable which is used in loop for some assignment statements.

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Strength reduction means replacing high strength operators by a low strength operators.

Q
```
i=1
while (i<10)
{
  :
  y = i*4;
}
```

⇒
```
i=1;
t1=4;
while (t1<40)
{
  y=t1;
  t1=t1+4;
}
```

Use algebraic identities - certain computation that look different to the compiler and are not identified as common subexpression, they are actually same.

Eg b op c. could be denoted different from c op b.

Certain operations like multiplication & addition will produce same result and they can be treated as common subexpression.

Peephole Optimization

A statement by statement code generation strategy often produce target code that contains redundant instructions and sub-optimal constructs.

A simple but effective technique for improving the target code is peephole optimization.

It is a method for trying to improve the performance of the target program by examining a short sequence of target instructions called the peephole and replacing these instructions by a short or faster sequence of instruction whenever possible.

Peephole is a small moving window on the target program that need to be optimization.

Techniques of Peephole Optimization.

→ Redundant instruction elimination

→ Flow of control optimization

→ Redundant Instruction Elimination

Eg 1) MOV a, R0

2) MOV R0, a

We can delete instruction 2 because whenever instruction 2 is executed, instruction 1 will ensure that the value of register R0 is already there in a.

→ Flow of Control Optimization

- Unreachable Code

```
#define debug 0
if (debug)
{
    print debugging information
}
```

⇓

```
#define debug 0
if (debug == 1) goto L1
goto L2
L1 : Print debug information
L2 : ..
```

⇓ optimization

```
If (debug ≠ 1) goto L2
Print debug information
L2 : ..
```

The unnecessary jumps can be eliminated in the intermediate code by replacing the jump sequences.

Undecidability

Decidability: A problem with two answers yes or no is said to be decidable if the corresponding language is recursive. That language is also said to be decidable.

A problem or a language is said to be undecidable if it is not decidable.

⇒ The intersection of two regular language is decidable.

⇒ The intersection of two CFL is undecidable.

⇒ Let G be an unrestricted grammar. Then the problem of whether or not $L(G) = \phi$ is undecidable.

⇒ Let M be any TM, then the problem of whether or not L(M) is finite is undecidable.

⇒ Let M be any TM, then the problem of L(M) is regular is undecidable.

⇒ Let M_1 and M_2 be two TMs, then is $L(M_1) \subseteq L(M_2)$ is undecidable.

(173)

⇒ Let G_1 be any unrestricted grammar and G_2 be any regular grammar then the problem of whether or not $L(G_1) \cap L(G_2) = \phi$ is undecidable.

⇒ Let G_1 & G_2 be two CFGs, then $L(G_1) \cap L(G_2) \neq \phi$ and $L(G_1) \subseteq L(G_2)$ are undecidable.

⇒ The post correspondance problem PS undecidable.

⇒ The modified PCP is also undecidable.

⇒ The elimination of ambiguity from a CFG is undecidable.

Q Which of the following is/are undecidable?

1. G is a CFG. & $L(G) = \phi$?
2. G is a CFG. & $L(G) = \Sigma^*$?
3. M is a TM. & $L(M)$ regular?
4. A is a DFA and N is an NFA. $L(A) = L(N)$?

(a) 3 only
(b) 1 & 3 only
(c) 1, 2 & 3 only
(d) 2 & 3 only

(173)

P & NP Class

→ **P-Class**: A language L is said to be in P-class if there exist a deterministic TM M such that M is of the time complexity $P(n)$ for some polynomial P and M accepts L. Here n is the no. of moves taken by TM to accept the string.

→ Eg:

2-SAT
PATH
Relatively Prime

→ P-class problems can be solved efficiently.

→ **NP-Class**: A language L is said to be in NP-class if there exist a non-deterministic TM M such that M is of time complexity $P(n)$ for some polynomial P and M accepts L.

Eg:

SAT
PATH
composite
CLIQUE

→ Polynomial Time Reduction:

If we can transform instances of one problem in deterministic polynomial time into instances of a second problem that has the same answer yes/no. Then we will say the first problem is polynomial time reducible to second problem.

→ NP- Complete:

A language L is said to be in NP-complete if it satisfies the following -

(i) L must be in NP-class.

(ii) For every language L' in NP, if there is a polynomial time-reduction from L' to L.

→ Eg: 3-SAT

SAT

PATH

CLIQUE

→ NP-Hard :

A language L is said to be in NP-hard, if

(I) $\forall L' \in NP$. if $L' \leq_P L$

(II) L May or may not be in NP class

eg :- TSP

⇒ Every NP-complete is also NP-Hard but not vice-versa.

NOTE:

- If $L_1 \leq_P L_2$ and if L_1 is NP-complete then L_2 is in NP-Hard if we don't know anything about L_2 that it is in NP or not.
- If $L_1 \leq_P L_2$ and if L_1 is NP-complete and given L_2 in NP-class then L_2 is also in NP-complete.
- If $L_1 \not\leq_P L_2$ and iff L_1 is undecidable ⇒ L_2 is also undecidable.
- If $L_1 \leq_P L_2$ and iff L_2 is undecidable ⇒ L_1 is also decidable.

(177)

Q Let 'S' be an NP-complete, Q and R be two other problems not known to be in NP. Q $\leq_P$ S and S $\leq_P$ R. Then which of the following is true?

(a) Q is NP-complete.

(b) Q is NP-Hard

(c) R is NP-complete

(d) R is NP-Hard

Q Ram and Shyam have been asked that a certain problem π is NP-complete. Ram shows a polynomial time reduction from 3-SAT problem to π and Shyam shows a polynomial time reduction from π to 3-SAT. Which of the following can be inferred from these reductions?

3-SAT $\leq_P$ π ... NP-complete

$\pi \leq_P$ 3-SAT ... NP

(a) π is NP Hard

(b) π is in NP but is not NP-complete

(c) π is NP complete

(d) π is neither NP hard nor in NP.

Q Consider 3 decision problems P_1, P_2, P_3.

Given that - P_1 is decidable and P_2 is undecidable

Which one of the following is true?

(a) P_3 is decidable if $P_1 \leq_P P_3$.

(b) P_3 is undecidable if $P_3 \leq_P P_2$.

(c) P_3 is undecidable if $P_2 \leq_P P_3$.

(d) P_3 is decidable if $P_3 \leq_P \overline{P_2}$.

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.