Module-I

High level language (HLL)

↓

[Preprocessor]

↓ pure HLL

[Compiler]

↓ assembly language

[assembler]

↓ machine code

#include stdio.h
- Preprocessor

HLC
↓
preprocesso
↓ pure
compiler
↓ ac
assembler
↓
mc

Phases of a Compiler

↓ pure HLL

Lexical analysis

↓ tokens

Syntax analysis

↓ (Passes)

Semantic analysis

↓ (valid parse tree)

Intermediate Code generation

↓ (3 address code)

Code optimization

↓

Code generation

[Symbol table]

[Error handling]

TRACE KTU

Grammar / CFG

$S \rightarrow id = E;$

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id$

```
main ()
{
x = a+b*c;
}
```

Parse tree

$x = a+b*c$

$\therefore$ It is valid.

If the expression is $x = a+(b*c)$, the above parse is invalid. Because parenthesis is absent.

If the syntax is able to deive a parsetree, then the syntax is valid. This is the way how the syntax analysis check whether the syntax is correct / wrong.

Logical errors are called semantic errors.

2 types of errors :-

Syntax errors.

Semantic errors.

(line number is absent)

TRACE KTU

Parse tree is meaningful or not = Semantic analysis

eg:
```
void main ()
{
int x[10], a=3, b=4, c=5;

x= a+ b*c;
printf ("%d", x);
getch ();
}
```

Logically incorrect, because here x is array, a, b and c are integer variables. Therefore the output should be an integer after performing the operation. Here semantic analyzer declares the presence of error.

TRACE KTU

Intermediate code generation

o/p will be a 3-address code (only 3 variables)

$x = a+b*c$ 4 address code

Converting the 4 address code in to 3 address code

$t_1 = b*c$

$t_2 = a+t_1$

$x = t_2$.

Code optimization

without changing the logic, change / decrease the number of steps is called optimization. (cycles)

Reducing time and space complexity.

$t_1 = b*c$ $x = a+t_1$

Code generation

Converts the optimized code into assembly language.

$t_1 = b * c$; $x = a + t_1$

$x = a + b * c$

ie; MOV R_1, R_2 $R_0 \leftarrow a$

ADD R_2, R_0 $R_1 \leftarrow b$

MOV, R_0, X $R_2 \leftarrow c$

Translator

A translator is a program that takes as i/p a prog written in one programming language and produces as o/ program in another language. If the source language is a high level language and object language as a LLL a assembly language, such a translator is called Comp Source prgm must be first combined. ie; it is trans in to object prgm, then the resulting object prgm is loa in to m/y and executed.

Interpreter

Interpreter is a translator that transforms a pr language in to simplified programing language called i which can be directly executed using an interpreter. Interpreters are smaller than compilers and facilitates implementation of complex programing language constr

TRACE KTU

However, the main disadv. of interpreter is that execution time of an interpreter is usually slower than a compiled object program.

Structure of a Compiler

A compiler takes i/p as source program & produces equivalence sequence of machine instructions. This process is very complex in terms of logical point of view and implementation point of view. That is why the compilation process is partitioned in to different subprocess called phases.

The first phase is called Lexical analysis or it is also called scanner. Lexical analyzer separates characters of the source language in to stream of tokens. The usual tokens are keywords, operators, identifiers etc. This set of tokens is passed to the next phase. Lexical analyzer uses regular expression to identify tokens.

2nd phase is called Syntax analysis. The syntax analyzer groups tokens together in to syntatic structures. for eg: 3 tokens representing a+b might be grouped in to a syntatic structure called an expression. Syntatic structures can be regarded as a tree called parse tree whose leaves are tokens. Rules of any programming language is entirely represented using grammar/ context free grammar. Syntax analyzer checks whether the i/p given is according to grammar/ not.

TRACE KTU

A parser has two fns. It checks whether the o/p of lexical analysis is permitted by specification of source language. It also imposes a tree like structure called parse tree that is used by subsequent phases of the Compiler. A parse tree is a diagram which exhibits syntax structure of expressions. Rules of any programming language using context free lang, it consists of some no. of prodn. Syntax analyzer has to check whether the i/p given is according to grammar / not.

Parse tree is given to semantic analyzer, it checks whether the it is meaningful or not.

Intermediate code generator uses the structure produced by syntax analyzer to produce a stream of simple instructions. Many styles of intermediate codes are possible, one common ~~add~~ style is 3 address code.

Code optimization is an optimal phase, it in Compiler design to improve the intermediate code so that the ultimate object program runs faster and takes less space.

The final phase code generation produces the object code while deciding m/y location for data and selecting registers in which each computations to be done. Design a code generator that produces efficient object program is one of the most difficult part in compiler design both practically and theoretically.

TRACE KTU

Table management / book keeping keeps track of the names used by the pgm and records essential information. Data structure used to record these information is called a symbol table. For eg: a Compiler needs to know whether a variable represent an integer or real number, size of an array, num of argument fn etc are stored in symbol table. inputs

The error handler is ~~involved~~ invoked whether an error in source program is detected, it must warn the programmer by issuing a diagonostic information. One of the most important fn of a Compiler is the detection and reporting of errors. Eg:

* Lexical analyzer may be unable to proceed because the next token in the source program is misspelled.
* Syntax analyzer may be unable to infer a structure ~~with~~ because of a missing parenthesis.
* Intermediate code generator detect an operator whose operands are of ~~con~~ incompatible types.

Whenever any phases of the Compiler discovers an error, it must report the error to the error handler which issues an appropriate diagonostic message.

TRACE KTU

1) Write the o/p of lexical analyzer and syntax analyzer for the following statement for train language

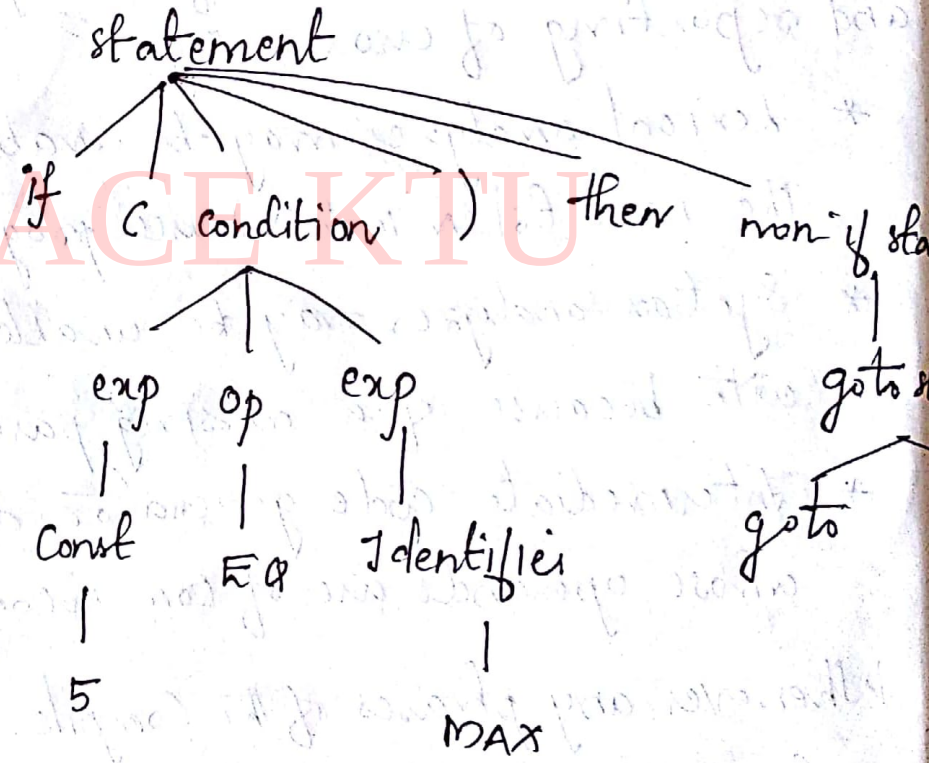
IF (5 EQ MAX) then go to 100

5 = Constant, MAX = identifier, 100 = label

o/p ⇒ if ([constant, 100] EQ [id, 102]) then goto [label, 104]

100	5
102	MAX
104	100

Lexical analyzer reads source program one character at time, it treats a token as a pair consisting of 2, one is token type and other is token value.



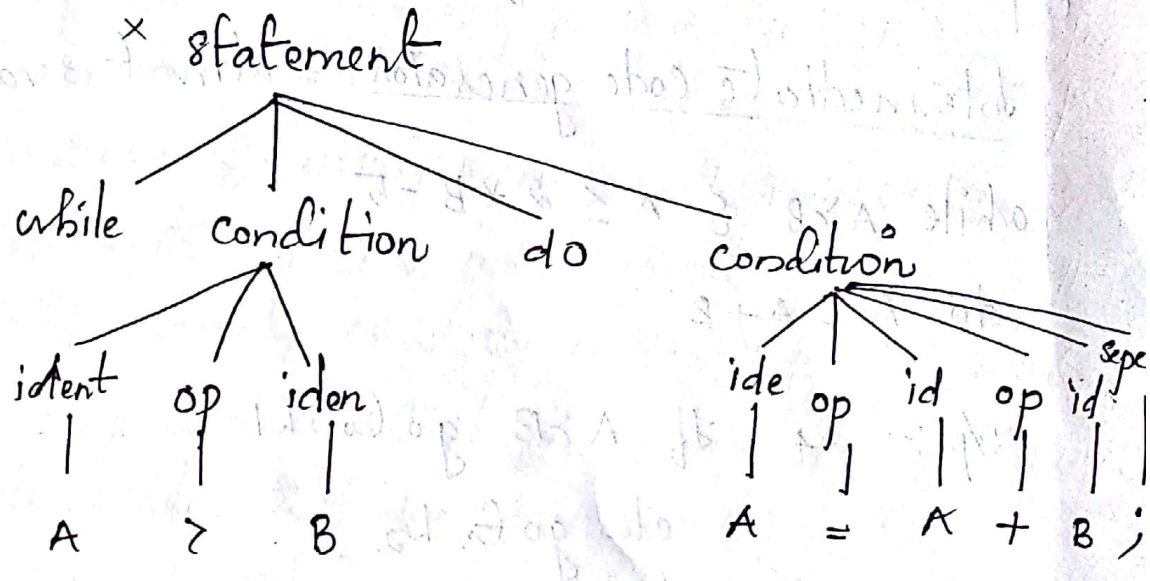
TRACE KTU

2) Write the o/p of lexical and syntax analyzer for the following fragment and intermediate c generator

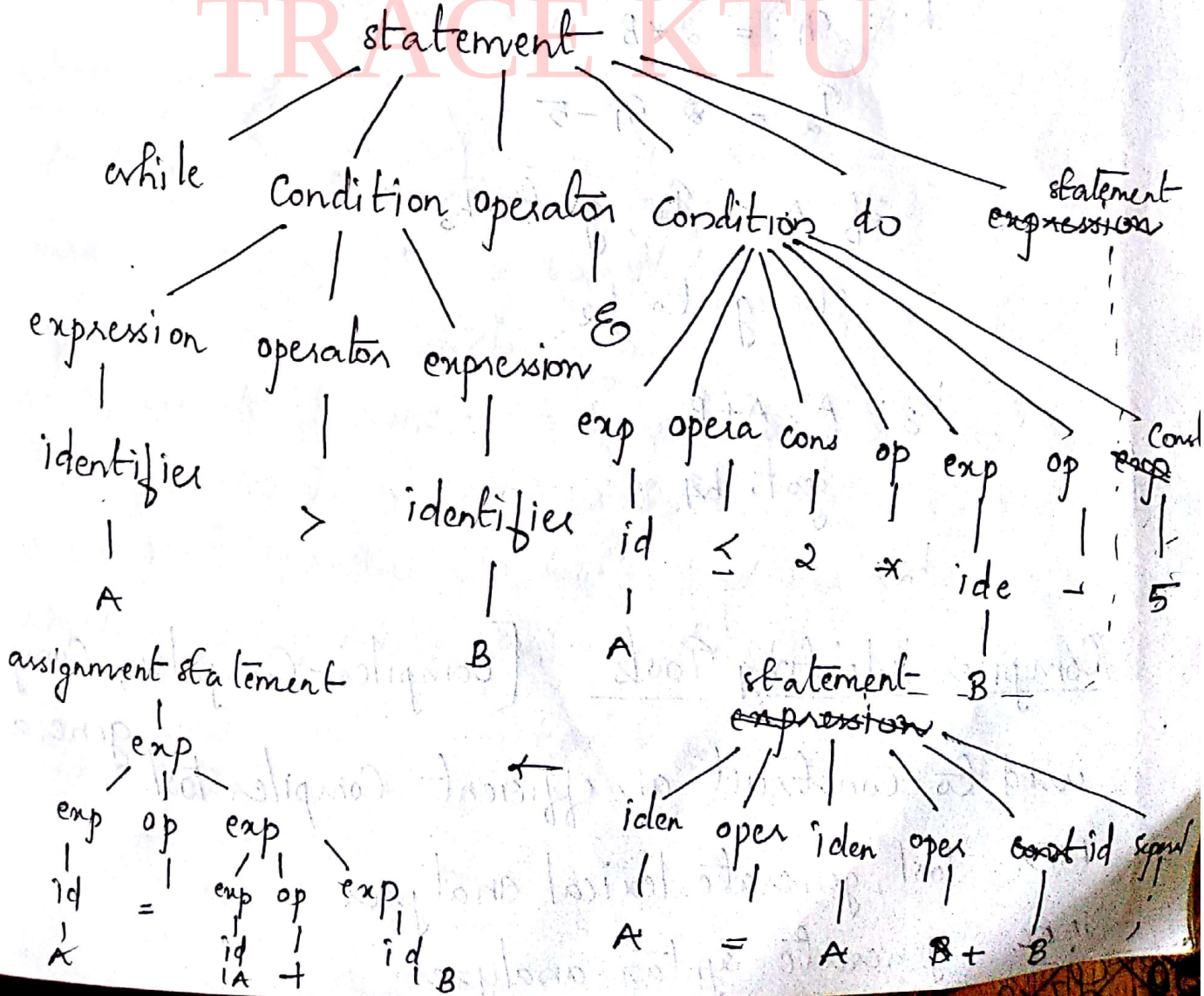
while A > B && A ≤ 2 * B - 5

do A = A + B;

while [identifier, 50] > [identifier, 52] and
[identifier, 50] ≤ [constant *] [identifier, 52]
− [constant]
do [identifier, 50] = [identifier, 50] + [identifier, 52]



TRACE KTU



while [identifier, 100], [operator, 102] [identifier, 104]
[operator, 106] [identifier, 100] [operator, 108] [Constant,
[operator, 112] [identifier, 104] [operator, 114] [Constant,
do [identifier, 100] [operator, 118] [identifier, 100]
[operator, 110] [identifier, 104]

<u>Intermediate code generator</u> = atmost 3 variables

while $A > B$ & $A \leq 2 * B - 5$
do $A = A + B$

o/p :- L_4 : If $A > B$ go to L_1
else go to L_2

L_1 : $T_1 = 2 * B$
$T_2 = T_1 - 5$

If $A \leq T_2$, go to L_3
else go to L_2

L_3 : $A = A + B$
go to L_4

L_2 :

<u>Compiler Writing Tools</u> [Compiler-Compiler, Compi
used to construct an efficient Compiler tool. general

LEX = tool, generate lexical analyzer
YACC = generate syntax analyzer

TRACE KTU

A no of tools have been developed to generate Compiler. This tools are called Compiler writing tool / Compiler-Compiler / compiler generator. They produce a Compiler from some form of specification of a language. These specification include lexical and syntatic structure of source language, description of what language generation, description of target machine etc. o/p is

for eg: instead of writing a prgm to perform syntax analysis, user writes a CFG and Compiler-Compiler automatically converts CFG to a prgm for syntax analysis. LEX is a Compiler writer tool for lexical analysis, YACC is a Compiler writer tool for syntax analysis (yet another Compiler Compiler)

TRACE KTU

Bootstrapping

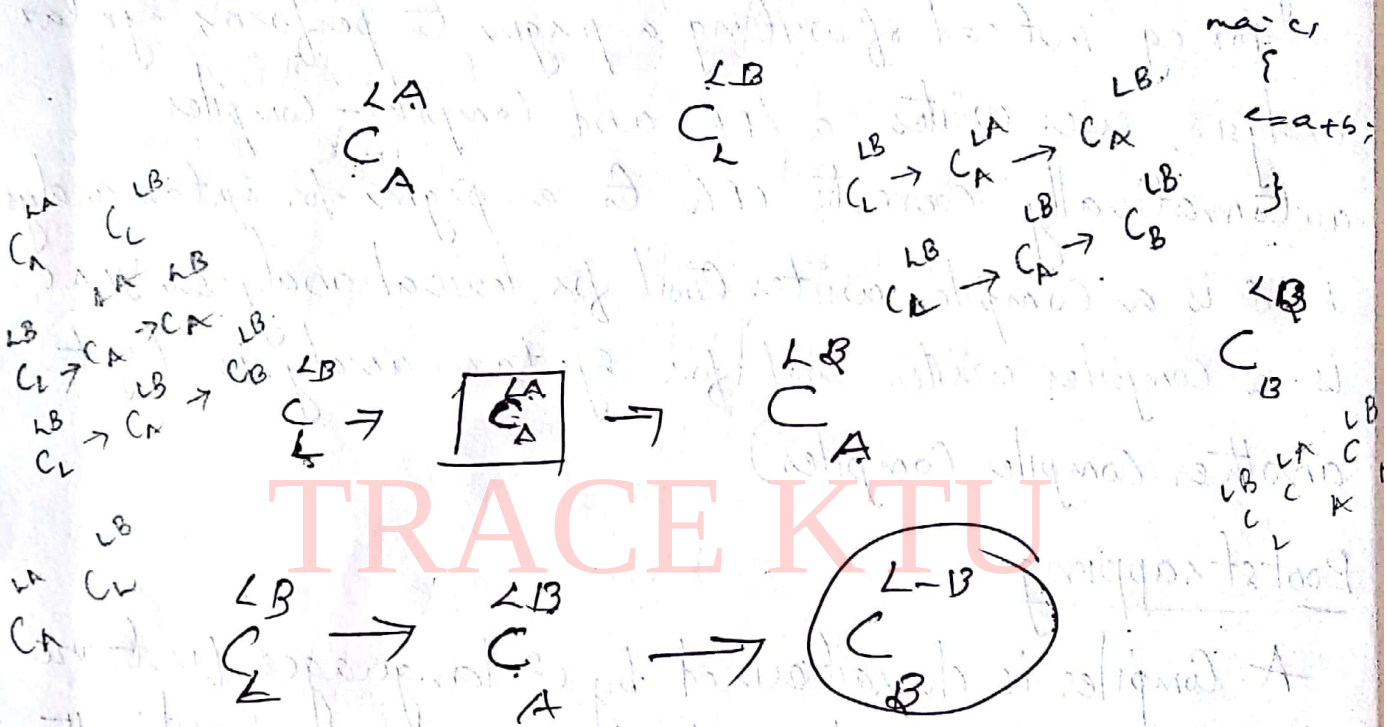
A Compiler is characterized by 3 languages. First one source language, 2nd is object language and 3rd is the language written in which compiler is written.

A Compiler that runs on one machine, produces object code for another machine is called Cross-Compiler. Suppose we want to produce a Compiler for language L to run on machine B and produce ~~source~~ code for B. We have Compilers C_A^{LA} and C_L^{LB} are need to produce C_B^{LB}

$$C_L^{LB} \rightarrow \boxed{C_A^{LA}} \rightarrow C_A^{LB}$$

$$C_L^{LB} \rightarrow \boxed{C_A^{LB}} \rightarrow C_B^{LB}$$

This is a 2 step process, we first run C_L^{LB} through C_A^{LA} to produce C_A^{LB}, a cross compiler for L which runs on machine A and produces code for B. Then we run C_L^{LB} through the cross-compiler to produce the desired compiler for L that runs on machine B and produce object code for B.



Need for lexical analysis

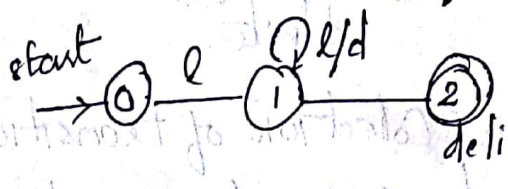
* we can greatly simplify the design of syntax analyzer
* Keeping track of line composed numbers
* Removing whitespaces.
* Deleting Comments

⇒ Regular expression is a notation that describes tokens

TRACE KTU

Input Buffering

identifier regular expression = $l(l/d)^*$ (letter followed by combination of letter / digits)

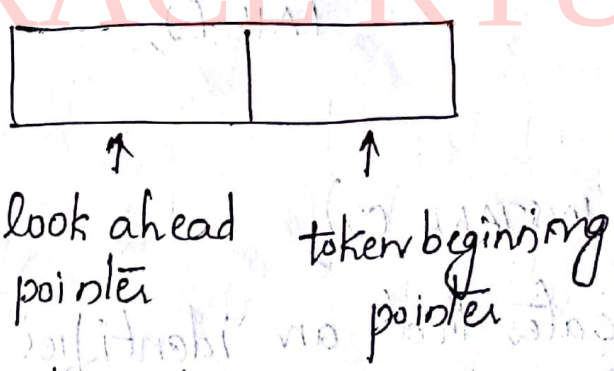


Input Buffering

* lexical analyzer scans the character of the source program one at a time to discover tokens.
* lexical analyzer reads its i/p from an i/p buffer
* There are 2 pointers token beginning pointer and look ahead pointer.
* first pointer marks the beginning of the tokens being discover.

look ahead pointer scans ahead until the token is discovered.

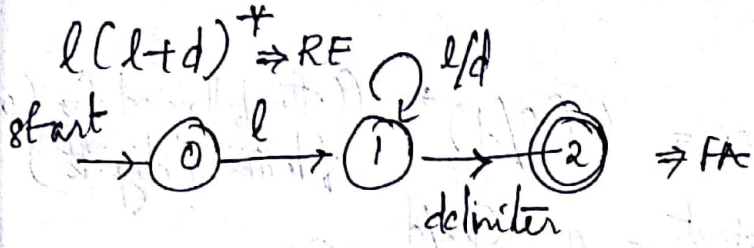
TRACE KTU



Design of lexical analyzer

One way to design any program is to describe the prgm by means of a flowchart. A specialized kind of flowchart is called transition diagram. Here the circle represents states and states are connected by edges. following is a transition diagram for identifier.

$l(l+d)^* \Rightarrow RE$



$\Rightarrow FA$

To turn a collection of transition diagrams into a prog we construct segment of code for each state.

state 0

```
c = getchar ();
IF ( LETTER (c));
  go to state1;
else
  FAIL ();
```

state 1

```
c = get char ();
if (letter (c) or
    digit (c))
  go to state 1;
else if (delimiter (c))
  go to state 2;
else
  FAIL ();
```

state 2

```
RETRACT ();
return (id, INSTALL ());
```

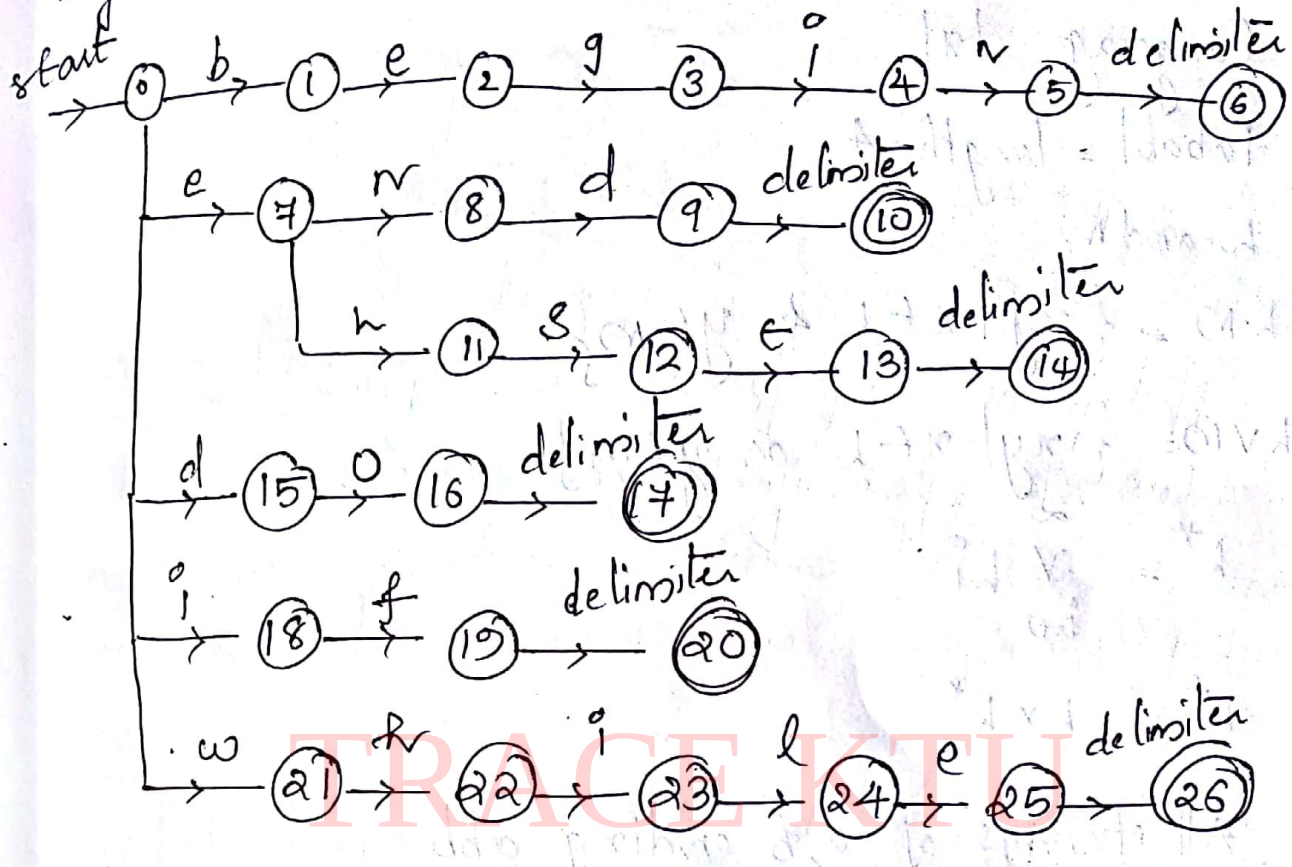
state 2 indicates that an identifier has been found since the delimiter is not a part of identifier, we must retract the look ahead pointer.
we must also install the newly found identifier in the symbol table if it is not already there.

TRACE KTU

Recognition of Tokens

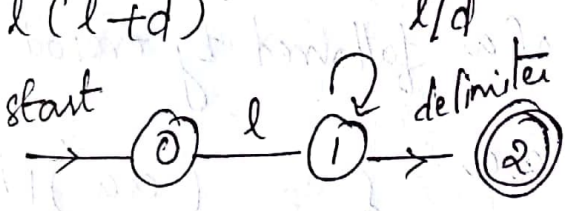
Keywords

begin, end, do, while, if, else ....

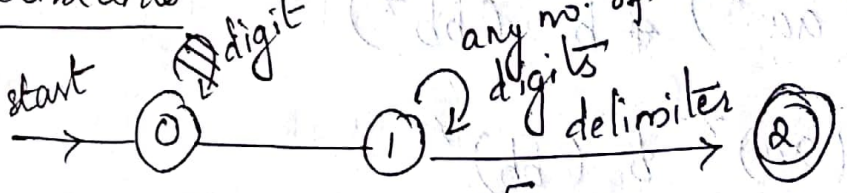


Identifier

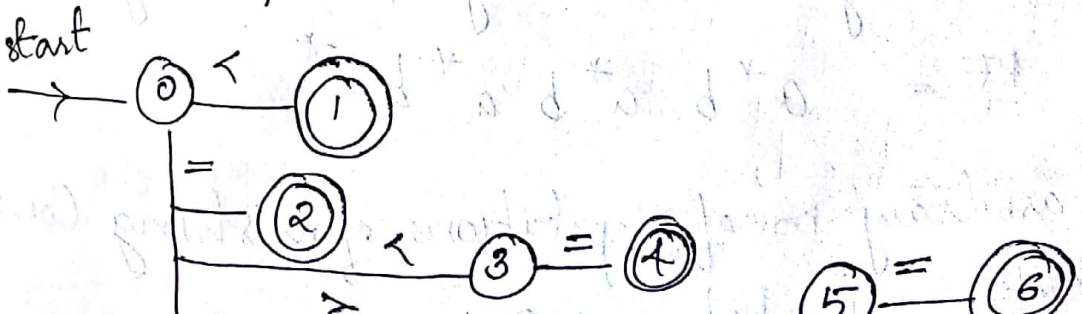
$l(l+d)^*$



Constants



Relational operators [<, =, <=, >=, <>]



TRACE KTU

Specification of Tokens

string → a, b, abc....
→ ε

language |x|

|ababl| = length = 4

L and M

$L \cdot M = \{xy \mid x \in L \ \& \ y \in M\}$

$L \vee M = \{xy \mid x \in L \text{ or } y \in M\}$

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

$L^+ = L \times L^*$

① All strings of a, b ending abb

$(a+b)^* abb$ or $(a/b)^* abb$

② Set of strings of even no. of a followed by ~~even~~ odd

$L = \{baa, aab, aa...\}$ $(aa^*)+b$

$RE = (aa^*) b (bb^*)$

$= (aa)^* b (bb)^*$ $(0^*, 0^*, 0^*)^*$

③ Set of strings with exactly 3 b's belongs to

$RE = a^* b a^* b a^* b a^*$

④ An arbitrary no. of repetitions of a string Consi of exactly 2 1's belongs 0, 1

TRACE KTU

$(0^*10^*10^*)^*$

⑤ Write a regular expression that generates string which doesnot contain regular substring 11.

$0^* + (11)^* 0^*$

$RE = (0^+10^+10^+)^*$

$(10+0)^*$ 11 ×

$(10+1)^*$ 00 ×

Finite Automata

It is a language recognizer that takes some i/p and answers 'yes' if that is a sentence of the language and 'no' otherwise.

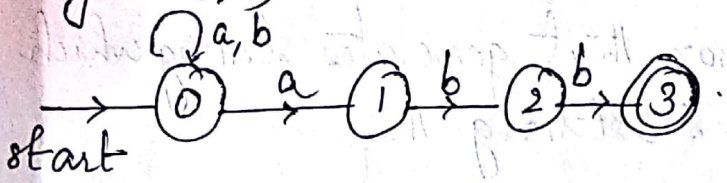
NFA

A better way to convert a regular expression to a recognizer is to construct a generalize transition diagram is called NFA. An NFA has no restrictions on the label of their edges that is a symbol can label several edges out of the same state. 'ε' string is also a possible label. An NFA consists of

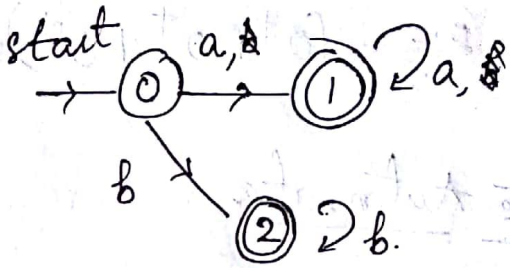
① finite set of states
② a set of i/p symbols
③ a transition function
④ start state
⑤ final state

TRACE KTU

eg: $(a/b)^* abb$



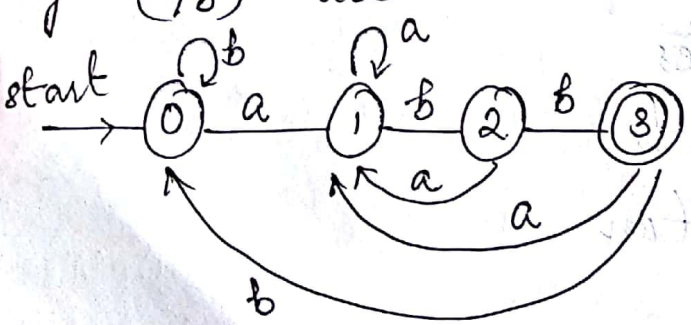
$(aa^*)\,bb^*)$ ← {aa, bb,



DFA

DFA is a special case of NFA such that there are no ε-moves, for each state s and an i/p symbol 'a' there is exactly one edge out of s label a. DFA is a simple and concrete algorithm for recognizing strings. An NFA has more states so it is hard to simulate whereas in a DFA they have atmost one path from the start state labelled by any string.

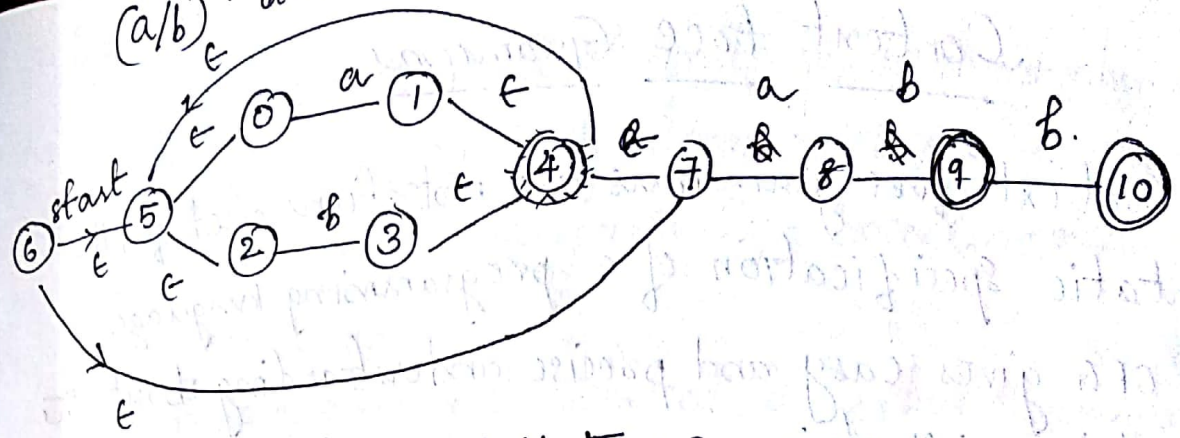
eg: $(a/b)^* abb$



* reg. exp to NFA
ε- NFA to DFA } refer in TOC

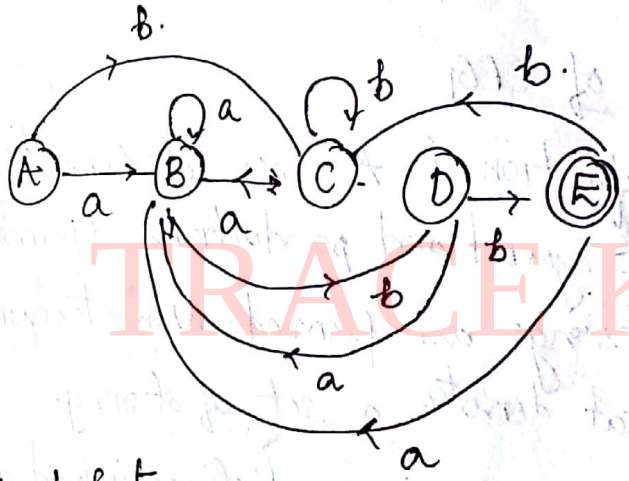
TRACE KTU

$(a/b)^* abb.$



ε-closure of 6 ⇒ 6, 7, 5, 0, 2.

DFA minimization



non-final state

(A, B, C, D) (E) final state

(A, B, C) (D), (E)

(A, C) (B) (D) (E)

A, B, D, E

TRACE KTU

Module - 2

Context free Grammar

Context free grammars are notations used for the syntatic specification of a programming language.

CFG gives easy and precise understanding about the syntatic specification

An efficient parser can construct from a properly define grammar

formal definition of CFG

The formal definition is $A \rightarrow \infty$. It consists of terminals non-terminals, start symbol and productions. Terminals are symbols in which strings are formed. Non-terminals are syntatic variable that denotes a set of string.

In a grammar one non-terminal is distingcished as a start symbol.

Productions specify the manner in which terminals and non terminals can be combined to form strings.

eg: $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

id+id *id

$E \rightarrow E+T$

$E+T*F$

$E+T*f$

$T+T*f$

$f+f*f \rightarrow id+id*id$

TRACE KTU

Derivations

Beginning with the start symbol each step replaces the non-terminal by the body of one of its productions. This derivational view corresponds top-down construction of a parse-tree (left

Bottom of passing is related to a class of derivations is called rightmost derivation in which each right most non-terminal is rewritten at each step.

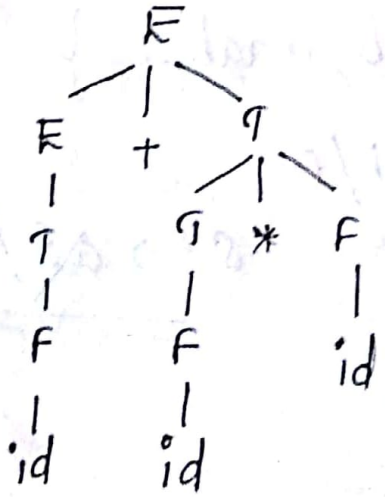
$E \rightarrow E+T$
$\rightarrow T+T$
$\rightarrow F+T$
$\rightarrow id+T$
$\rightarrow id+T*F$
$\rightarrow id+F*F$
$\rightarrow id+id*F$
$\rightarrow id+id*id$

TRACE KTU

Parse tree

It is a graphical representation of derivation that filters out orders in which productions are applied to replace non-terminals.

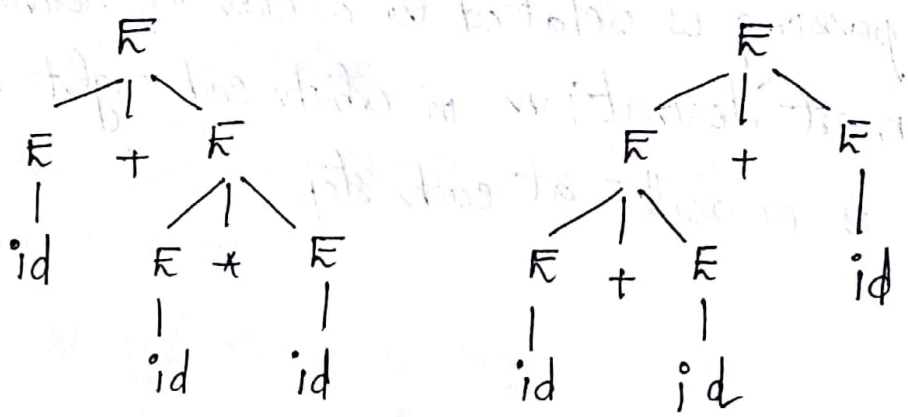
A



Ambiguity

A grammar that produces more than one parse tree said to be ambiguous. An ambiguous grammar produces than one left most or right most derivations.

$E \rightarrow E+E \,/\, E*E \,/\, (E/E) \,/\, id$



① $a^* = \{\epsilon, a, aa, aaa \ldots\}$

$S \rightarrow aS / \epsilon$

$S \rightarrow aS$

$\rightarrow aaS$

$\rightarrow aaaS$

$\rightarrow aaaaaS$

$\rightarrow aaaa\epsilon$

$\rightarrow a^4$

② $a^*b = \{\epsilon b, ab, aab \ldots$

1) $S \rightarrow as / b / \epsilon$ OR

1) $S \rightarrow Ab$

$A \rightarrow aA / \epsilon$

③ $a^+b = \{ab, aab, aaab \ldots\}$

$S \rightarrow aA / b / \epsilon$

$A \rightarrow b$

$S \rightarrow aA / b / \epsilon$

$S \rightarrow aS / b$

$S \rightarrow aS / b$

a

a

ab

as

aas

aab

TRACE KTU

$S \to aAb$

$A \to aA / \epsilon$

$S \to aAb$
$aaAb$ ab
aaA
a
aaAb

④ $a(b/c^*) = \{ab, a, ac, acc \ldots\}$

$S \to aA$

$A \to b/\epsilon/cA$

$S \to aA$
$A \to b|\epsilon|cA$
a

⑤ $a^n b^n \mid n \geq 1$

Context free grammar, Context free languages.

$L = \{ab, aabb, aaabbb \ldots\}$

$S \to \cancel{aSB}\ asb \mid ab$

1) $a(ab)^*(a/b)$

$L = \{aaba, aababb, ab, \ldots\}$

$S \to aAB$

$A \to abA \mid \epsilon$

$B \to a/b$

$S \to aAB$
$A \to abA \mid \cancel{ab}\ \epsilon$
ab aba
ababa
$C \to a|b$
$aabA|\epsilon$
aab

2) $a^n b^m c^m d^n \mid m \geq 1, n \geq 1$

$L = \{abccd, abcd, aabcd \ldots\}$

$S \to abcd \quad S \to aAd \mid aSd$

$A \to bAc \mid bc$

$S \to aSbc$
$S \to abcd$
$S \to aSd$
aaAdd
aabcdd

3) $a^n b^n c^m d^m \mid m, n \geq 1$

$a^4 b^4 c^3 d^3$

$S \to aAb \mid aSb$

$A \to cAd \mid cd$

$S \to aAb$
acAdb
acdb

TRACE KTU

$S \to aAb \mid asb$

$A \to cdA \mid cd$

$a^4b^4c^3d^3$.

$S \to asb$.

$aaAbb$.

$aacdAbb$.

$aacd$

$S \to AB$

$A \to aAb \mid ab$

$B \to cBd \mid cd$

$S \to asb$

$aasbb$.

Parser

It is a program that takes i/p as string and produces o/p either a parse tree or an error message. There are two types of parser

(i) Top-down parser

(ii) Bottom-up parser

Top-down parser

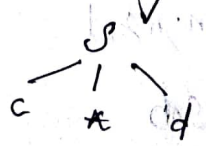
Top down parser starts with the root and work down to leaves. This can be viewed as an attempt to find leftmost derivation for an i/p string. Consider the following productions (grammar)

$S \to cAd$

$A \to ab \mid a$

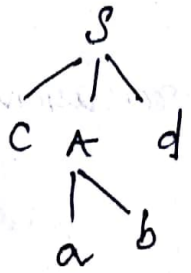
Let the i/p string be $w = cad$

To construct a parse tree for this sentence, the top-down parser creates a tree consisting of single node 'S' then it expands 'S' as

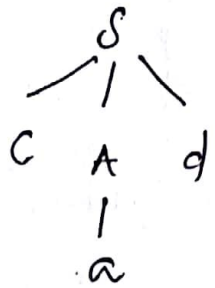


TRACE KTU

Then expand A as



When checking for a match, it finds 'b' instead of d. Now the parser will backtracks and go back to A to see whether there is an another alternative for A.



Now the i/p and parse tree matches and parser announces successful completion.

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

$A \rightarrow \beta A'$

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

TRACE KTU

Difficulties with top-down parsing

1) Left Recursion

A grammar is said to be left recursive if there is a non-terminal A such that there is a derivation $A \rightarrow A\alpha$ for some α. A left recursive grammar can cause a top down parser to go in to an infinite loop

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

after removing left recursion

$$A \rightarrow \beta A'$$

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

① Eg: $E \to E + T / T$ } left recursion
$T \to T * F / F$
$F \to (E) / id$

$A \to A\alpha / \beta$

after removing left recursion

$A \to \beta A'$
$A' \to \alpha A' / \epsilon$

$E \to TE'$
$E' \to +TE' / \epsilon$
$T \to FT'$
$T' \to *FT' / \epsilon$
$F \to (E) / id$

$A \to A\alpha / \beta$
$A \to \beta A'$
$A' \to \alpha A' / \epsilon$

② Second problem concerns backtracking
If we make a sequence of expansions and subsequently discover a mismatch, then we have to undo the semantic effects. undoing the semantic actions requires substantial overhead

③ Third problem with top down parsing is that the order in which alternative line can affect the language acceptance.

Sananya Chakraborty

Recursive descent parser

A parser that uses a set of recursive procedures to recognize its i/ps is called recursive descent parser. for each non-terminal there is a procedure & and execution begins with the procedure of start symbol

TRACE KTU

Eg:- $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)/id$

```
procedure E ()
   begin
     procedure T();
     procedure E'();
   end
procedure E'()
   begin
   if input symbol='+'
          ADVANCE ();
     procedure T();
     procedure E'();
   end
procedure T()
   begin
     procedure F();
     procedure T'();
   end
procedure T'()
   begin
   if input symbol = '*' then
          ADVANCE();
     procedure F().
     procedure T'().
   end
```

```
procedure F()
   begin
   if i/p symbol = '(' then
          ADVANCE();
     procedure E();
   if i/p symbol = ')' then
          ADVANCE();
   ~~proceed~~ else
     error message
   end
```

TRACE KTU

General form

$A \rightarrow x_1 / x_2 \ldots / x_k$

```
void A ()
{
for (i=1 to k)
    {
 if (xi is a non terminal)
            call procedure x();
 else if (xi is a terminal) then
            Advance the input pointer
 else
    error
    }
}
```

TRACE KTU

Predictive Parser

First and follow

Construction of both top-down and bottom up parser aided by two fns. first and follow associated with a grammar

FIRST

If α is any string of grammar symbol, then first (α) be the set of terminals that begin string derived from α

1. If x is a terminal, $First(x) = \{x\}$

2. If x is a non-terminal and $x \to a\alpha$ is a production then $first(x) = \{a\}$, if $x \to \epsilon$ is also a production then add ϵ to $first(x)$; $first(x) \to \{a, \epsilon\}$

3. If $x \to y_1, y_2 \ldots y_n$ then $first(x) = first(y_1)$ if y_1 doesnot have an ϵ-production. Else add $first(y_2)$ to $first(x)$ and so on.

Eg:
$$E \to TE'$$
$$E' \to +TE' / \epsilon$$
$$T \to FT'$$
$$T' \to *FT' / \epsilon$$
$$F \to (E) / id$$

$A \to \alpha B \beta$

TRACE KTU

E, E', T, T' and F

$First(E) = First(T) = First(F) = \{c, id\}$

$First(E') = \{+, \epsilon\}$

$First(T') = \{*, \epsilon\}$

<u>FOLLOW</u>

Follow of a non-terminal A to be B is the set of terminals that appear immediately to the right of A

1. place \$ in follow of s if s is the start symbol.

2. If there is a production $A \to \alpha B \beta$, and $\beta \neq \epsilon$ then follow(B) will contain everything in $first(\beta)$

3. If there is a production $A \to \alpha B \beta$ and first (β) Co
ϵ, then follow (B) Contains first (β) and follow (A)

Eg: $E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$

first(E) = First (T) = First (F) = $\{ (, id \}$

first (E') = $\{ +, \epsilon \}$

First (T') = $\{ *, \epsilon \}$

$\therefore$ follow (E) = $\{ \$,) \}$

follow (T) = $\{ +, \$,) \}$

follow (E') = $\{ \$,) \}$

follow (T') = $\{ +, \$,) \}$ ϵ = follow

follow (F) = $\{ *, +, \$,) \}$

1. find the first and follow for the following gram

$S \to a / \wedge / (T)$

$T \to T, S / S$

TRACE KTU

first (S) = {a}

first (T) =

<u>Eliminate left recursion</u>

$T \to T, S \;/\; S$

$\alpha \qquad \beta$

$A \to A\alpha \,/\, \beta$

$A \to \beta A'$

$A' \to \alpha A' \,/\, \epsilon$

first (S) = { a, ^, (}

$T \to S T'$

$T' \to , S T' \,/\, \epsilon$

→ first (S) = { a, ^, (}

first (T) = { a, ^, (}

first (T') = { , , ε }

follow (T') =

follow (S) = { $, , ε

= { $, first (T'), follow (T) }

= { $,), follow (T) }

= { $,),) }

follow (T) = {) }

follow (T') = follow (t) = {) }.

$A \to A\alpha \,/\, \beta$

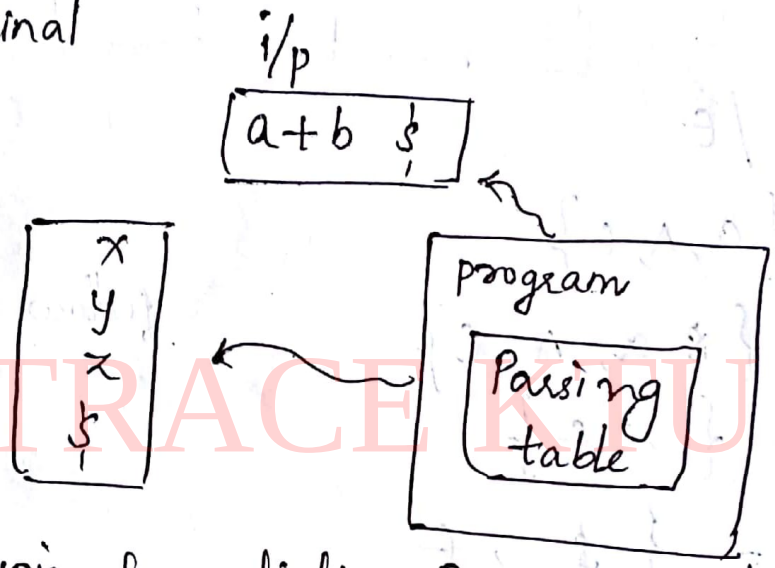
$A \to \beta A'$

$A' \to \alpha A' \,/\, \epsilon$

TRACE KTU

Predictive Parser

It is an efficient way of representing recursive descent parser. It has an i/p, stack, parsing table and an o/p. The i/p contain string to be parsed followed by $. stack contains sequence of grammar symbols. Initially stack contain sequence of grammar symbol. Parsing table is a 2D array like M[A, a] A is non-terminal and a is terminal



TRACE KTU

Construction of predictive Parsing table

i/p : Context free grammar

o/p : Parsing table

method

1st step : for each production $A \to \alpha$ do steps 2 & 3.

2nd step : for each terminal 'a' in first of α, add $A \to \alpha$ to M[A, a]

3rd step : If ϵ is in first (α), then add $A \to \alpha$ to M[A, b] for each 'b' in follow (A)

If ϵ is in first(α) and $\$$ is in follow(A) then add $A \to \alpha$ to $M[a, \$]$.

4th step: Make each undefined entry as error.

1. Consider the following set of productions.

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$

(a) Construct a predictive parser table

(b) Pass the following string using predictive parsing table.

$id+id*id\$$

→ check whether the grammar is LL(1) or not?

$E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$

TRACE KTU

F(E)

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}

FIRST (E') = {+, ϵ} FIRST (T') = {*, ϵ}

Follow (E) = {), $}

follow (E') = {), $}

Follow (T) = follow (T') = {+,), $}

Follow (F) = {*, +,), $}

ⓐ Parsing table

	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

$E \to TE'$
$E'T$

ⓑ

stack	i/p	o/p
E	id+id*id$	
$E'T$	id+id*id$	$E \to TE'$
$E'T'F$	id+id*id$	$T \to FT'$
$E'T'id$	id+id*id$	$F \to id$
$E'T'$	+id*id$	
E'	+id*id$	$T' \to \epsilon$
$E'T+$	+id*id$	$E' \to +TE'$
$E'T$	id*id$	
$E'T'F$	id*id$	$T \to FT'$
$E'T'id$	id*id$	$F \to id$
$E'T'$	*id$	
$E'T'F*$	*id$	$T' \to *FT'$

TRACE KTU

$\$ E'T'F$	$id\ \$$	
$\$ E'T'id$	$id\ \$$	$F \to id$
$\$ E'T'$	$\$$	
$\$ E'\epsilon$	$\$$	$T' \to \epsilon$
$\$ E'$	$\$$	
$\$$	$\$$	$E' \to \epsilon$

$\therefore$ the string is accepted

LL(1) grammar (predictive parser)

A grammar whose parsing table has not multiple entries is said to be LL(1) where the first L indicates left to right and second L indicates left most derivation.

1) Check the following grammar is LL(1) or not

$S \to ict\ SS' \mid a$

$S' \to eS \mid \epsilon$

$C \to b$

FIRST (S) = {i, a}

FIRST (S') = {e, ε}

FIRST (C) = {b}.

FOLLOW (S) = {$, e, ε}

FOLLOW (S') = {$, e, ε}

FOLLOW (C) = {t}.

TRACE KTU

	i	t	a	e	b.	$\$_1$
S	$S \to ictss'$		$S \to a$.			
S'				$S' \to es$		$S' \to \epsilon$
c		$c \to b$			$c \to b$	

TRACE KTU

Module-3

Bottom-Up Parsers

Bottom up parsing

1) shift reduce parser
2) Operator precedence parser

shift reduce parser

It attempts to construct a parse tree for an i/p string beginning at leaves and working up towards the root. Consider the following grammar

$S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$

TRACE KTU

for an i/p string abbcde we want to reduce this to a start symbol

abbcde

aAbcde

aAcde

aAcBe

S

Each replacement of the right side of production by the left side is called reduction.

Handle :- It is a substring that matches the body of production, it is a substring which is the right side of

production such that replacement of substring by produ
of left side leads to a reduction to the start symbol.

Handle Pruning :-

Right most derivation in reverse is obtained by ha
pruning. We start with string of terminals 'w' whic
we wish to parse to reconstruct the derivation in rev
we locate handle and replace by the left side of the
production, repeating this process until we locate start

Stack implementation of shift reduce parser

A convenient way to implement shift reduce pars
to use a stack and an i/p buffer. We use $ to m
bottom of stack and i/p.

TRACE KTU

Possible actions in a shift reduce Parser

① Shift : next symbol is shifted to top of the stack.

② Reduce : Handle that appears on the top is replaced b
appropriate non-terminal.

③ Accept : Here parser announces successful completion

④ Error : It is a situation in which the parser ca
either shift / reduce.

1. Consider the following grammar

$E \rightarrow E+E$ $\quad$ $E \rightarrow (E)$

$E \rightarrow E*E$ $\quad$ $E \rightarrow id$

show the steps of a shift reduce parser corresponding to the i/p $id_1 + id_2 * id_3$ \$

stack	i/p buffer	action
\$	$id_1 + id_2 * id_3$ \$	shift
\$ id_1	$+ id_2 * id_3$ \$	Reduce $E \to id$
\$ E	$+ id_2 * id_3$ \$	shift
\$ E +	$id_2 * id_3$ \$	shift
\$ E + id_2	$* id_3$ \$	Reduce $E \to id$
\$ E + E	$* id_3$ \$	shift
\$ E + E *	id_3 \$	shift
\$ E + E * id_3	\$	Reduce $E \to id$
\$ E + E * E	\$	Reduce $E \to E * E$
\$ E + E	\$	Reduce $E \to E + E$
\$ E	\$	accept

TRACE KTU

2) $S \to a / \wedge / (T)$

$T \to T, S / S$

Consider the following grammar show the steps of shift reduce parser corresponding to following i/p

(a) (a, (a,a))

(b) (((a,a), ∧, (a)), a)

stack	i/p buffer	action
$\$_1$	$(a,(a,a))\$_1$	shift
$\$_1($	$a,(a,a))\$_1$	shift
$\$_1(a$	$,(a,a))\$_1$	Reduce $S \to a$
$\$_1(S$	$,(a,a))\$_1$	shift
$\$_1(S,$	$(a,a))\$_1$	shift
$\$_1(S,($	$a,a))\$_1$	shift
$\$_1(S,(a$	$,a))\$_1$	shift
$\$_1(S,(a,$	$a))\$_1$	shift
$\$_1(S,(a,a$	$))\$_1$	shift
$\$_1(S,(a,a)$	$)\$_1$	shift
$\$_1(S,(a,a))$	$\$_1$	

stack	i/p buffer	action
$\$_1$	$(a,(a,a))\$_1$	shift
$\$_1($	$a,(a,a))\$_1$	shift
$\$_1(a$	$,(a,a))\$_1$	Reduce $S \to a$.
$\$_1(S$	$,(a,a))\$_1$	~~shift~~ Reduce $T \to S$
$\$_1(T$	$,(a,a))\$_1$	shift
$\$_1(T,$	$(a,a))\$_1$	shift
$\$_1(T,($	$a,a))\$_1$	shift

TRACE KTU

$\$_1$ (T, (a	, a)) $\$_1$	Reduce $S \to a$
$\$_1$ (T, (S	, a)) $\$_1$	shift
$\$_1$ (T, (S,	a)) $\$_1$	shift
$\$_1$ (T, (S, a	)) $\$_1$	Reduce $S \to a$
$\$_1$ (T, (S,		
$\$_1$ (T, (S	, a)) $\$_1$	Reduce $T \to S$
$\$_1$ (T, (T	, a)) $\$_1$	shift
$\$_1$ (T, (T,	a))	shift
$\$_1$ (T, (T, a	))	Reduce $S \to a$
$\$_1$ (T, (T, S	))	shift
$\$_1$ (T, (T, S)	)	shift
$\$_1$ (T, (T, S))		

TRACE KTU

2. Conflicts

1) shift reduce Conflict = unable to say whether to shift / reduce.

2) Reduce-reduce Conflict

b) (((a,a), ∧, (a)), a)

stack	i/p buffer	action
$	(((a,a), ∧, (a)), a)	shift
$ (	((a,a), ∧, (a)), a)	shift
$ ((	(a,a), ∧ (a)), a)	shift
$ (((	a,a), ∧, (a)), a)	shift
$ (((a	, a), ∧, (a)), a)	Reduce S→a
$ ((S	, a), ∧, (a)), a)	Reduce T→S
$ (((T	, a), ∧, (a)), a)	shift
$ (((T,	a), ∧, (a)), a)	shift
$ (((T, a	), ∧, (a)), a)	Reduce S→a
$ (((T, S	), ∧, (a)), a)	shift Reduce S→T
$ (((T, T)	), ∧, (a)), a)	shift
$ (((T, T)	, ∧, (a)), a)	shift
$ (((T, T),	∧	
$ (((T, S	), ∧, (a)), a)	Reduce T→T, S
$ (((T	), ∧, (a)), a)	shift
$ (((T)	, ∧, (a)), a)	shift Reduce S→(T)
$ ((S	, ∧, (a)), a)	Reduce T→S.
$ ((T	, ∧, (a)), a)	shift

TRACE KTU

$ ((T,	∧, (a)), a)	shift
$ ((T,∧	, (a)), a)	Reduce S → ∧
$ ((T,S	, (a)), a)	Reduce ~~S → T~~ T → T,S
$ ((T,T	, ~~(a))~~, a)	~~shift~~
$ ((T,T		
$ ((T	, (a)), a)	shift
$ ((T,	(a)), a)	shift
$ ((T,(	a)), a)	shift
$ ((T,(a	)), a)	Reduce S → a
$ ((T,(S	)), a)	Reduce T → S
$ ((T,(T	)), a)	shift
$ ((T,(T)	), a)	~~s~~ reduce S → (T)
$ ((T,S	), a)	~~shi~~ reduce T → T,S
$ ((T	), a)	shift
$ ((T)	, a)	Reduce S → (T)
$ (S	, a)	Reduce T → S
$ (T	, a)	shift
$ (T,	a)	shift
$ (T, a	)	Reduce S → a
$ (T,S	)	~~s~~ reduce T → T,S
$ (T	)	shift
$ (T)	$	Reduce S → (T)

TRACE KTU

$\$_1 S$ $\$_1$ accept

(a) $S \to a \mid \wedge \mid (T)$ $\Rightarrow (a, (a,a))$

$T \to T, S \mid S$

stack	i/p buffer	action
$\$_1$	$(a, (a,a)) \$_1$	shift
$\$_1 ($	$a, (a,a)) \$_1$	shift
$\$_1 (a$	$, (a,a)) \$_1$	Reduce $S \to a$
$\$_1 (S$	$, (a,a)) \$_1$	Reduce $T \to S$
$\$_1 (T$	$, (a,a)) \$_1$	shift
$\$_1 (T,$	$(a,a)) \$_1$	shift
$\$_1 (T,($	$a,a)) \$_1$	shift
$\$_1 (T,(a$	$, a)) \$_1$	Reduce $S \to a$
$\$_1 (T,(S$	$, a)) \$_1$	Reduce $T \to S$
$\$_1 (T,(T$	$, a)) \$_1$	shift
$\$_1 (T,(T,$	$a)) \$_1$	shift
$\$_1 (T,(T,a$	$)) \$_1$	reduce $S \to a$
$\$_1 (T,(T,S$	$)) \$_1$	reduce $T \to T,$
$\$_1 (T,(T$	$)) \$_1$	shift
$\$_1 (T,(T)$	$) \$_1$	~~shift~~ Reduce S
$\$_1 (T,S$	$) \$_1$	Reduce $T \to T$
$\$_1 (T$	$) \$_1$	shift
$\$_1 (T)$	$\$$	Reduce $S \to (T$

TRACE KTU

S_1 S_1 accept.

Operatorprecedence parser

Operator grammar is a grammar which has a property that no production right side is epsilon' or has two adjacent non-terminals.

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

It is an operator grammar

$E \rightarrow EAE \mid (E) \mid id$ is not an operator grammar.

In operator precedence parsing we use 3 disjoint precedence relations ($<, >, =$) b/w certain pairs of terminals. These precedence relation guides the selection of handles.

If $A < B$ we say B take precedence over A

$A = B$ then A and B have same precedence.

$A > B$ means A takes precedence over B.

eg: $E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

	+	*	id	S_1
+	=	<	<	>
*	>	=	<	>
id	>	>	=	>
S_1	<	<	<	=

⇒ 'id' is having high precedence

'S_1' is having always low precedence

$\$_1 < id + id * id \$_1$

$\$_1 < id > + < id > * < id > \$_1$

$\$_1 \; E + E * E \; \$_1$

$\$_1 \; + \quad * \quad \$_1$

$\$_1 < + < * > \$_1$

$\$_1 < E + < E * E > \$_1$

$\$_1 < E + E \; \$_1$

$\$_1 < \; + > \$_1$

$\$_1 < + > \$_1$

$\$_1 < E + E > \$_1 \Rightarrow \$_1 E \$_1 \Rightarrow \$_1 E \$_1$

TRACE KTU

Efficient Parsers

This area shows how to construct efficient bordes of parsers for a large class of CFG.

They are also called LR parsers, because they scan i/p from left to right and construct a right most derivation in reverse.

LR parsers are attractive for variety of ~~cases~~ reasons They are

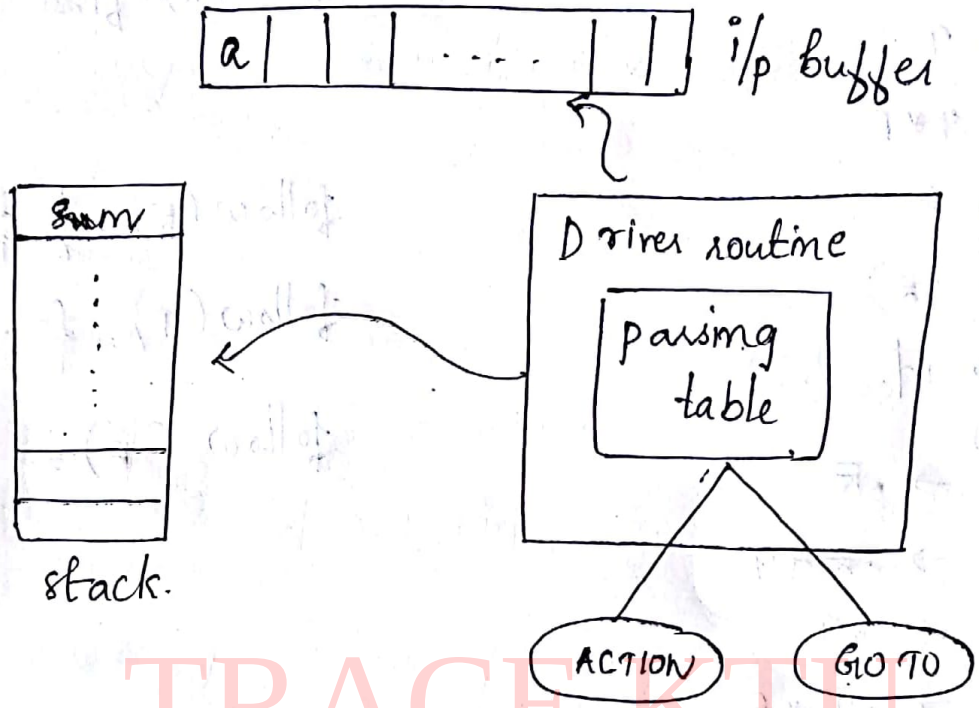
① They can be constructed to recognize all programing languages constructs

② LR parsing is more general than other techniques

③ LR parsers can detect syntatic errors as soon as it is possible to do so.

Drawbacks

Too much work is required to implement LR parsers.



TRACE KTU

1) SLR parser (simple)
2) CLR parser (canonical)
3) LALR parser. (Look ahead)

① Construct SLR grammar for following grammar.

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

1) SLR parsing table
2) Stack parsing.

Bottom-up

no need to eliminate left recursion.

Top down

follow (E) = {), $ }

follow (T) = { +,), $ }

follow (F) = { +, *,), $ }

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id.

I_0 : E' → .E (augmentation)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

→ E, T, F, (, id

I_1 : GOTO (I_0, E)
E' → E.
E → E.+T

I_2 : GOTO (I_0, T)
E → T.
T → T.*F

I_3 : GOTO (I_0, F)
T → F.

first (E) = first (T) = first (F) = { (, id }

follow (E) = { $, +,) }

follow (T) = { $, +,), * }

follow (F) = { $, +,), * }

TRACE KTU

I_4 : GoTo (I_0, ( )

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

I_5 : GoTo (I_0, id)

$F \to id \cdot$

$\Rightarrow$ +

I_6 : GoTo (I_1, +)

$E \to E + \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$\Rightarrow$ *

I_7 : GoTo (I_2, *)

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$\Rightarrow$ E, T, F, (, id

I_8 : GoTo (I_4, E)

$F \to (E \cdot)$

$E \to E \cdot + T$

I_2 : GoTo (I_4, T)

$E \to T \cdot$

$T \to T \cdot * F$

I_3 : GoTo (I_4, F)

$T \to F \cdot$

I_4 : GoTo (I_4, ( )

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

I_5 : GoTo (I_4, id)

$F \to id \cdot$

$\Rightarrow$ T, F, (, id

I_9 : GoTo (I_6, T)

$E \to E + T \cdot$

$T \to T \cdot * F$

I_3 : GoTo (I_6, F)

$T \to F \cdot$

I_4 : GoTo (I_6, ( )

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

TRACE KTU

I_5 : GOTO (I_6, id)

$F \to id.$

$\Rightarrow$ F, (, id

I_{10} : GOTO(I_7, F)

$T \to T * F.$

I_4 : GOTO (I_7, ( )

$F \to (.E)$

$E \to .T$

$E \to .E + T$

$T \to .T * F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_5 : GOTO (I_7, id)

$F \to id.$

$\Rightarrow$), +

I_{11} : GOTO(I_8,))

$F \to (E).$

I_6 : GOTO (I_8, +)

$E \to E + .T$

$T \to .T * F$

$T \to .F$

$F \to .(E)$

$F \to .id$

$\Rightarrow$ *

I_7 : GOTO(I_9, *)

$T \to T * .F$

$F \to .id$

$F \to .(E)$

$A \to .\alpha . a\beta$

I_0 : $F \to .(E)$

GOTO (I_0, ( ) = I_7

Action [0, ( ] = S_4

GOTO (I_0, id) = I_5

Action [0, id] = S_5

I_1 : $E \to E . + T$

GOTO (I_1, +) = I_6

action (1, +) = S_6

I_2 : $T \to T . * F$

GOTO (I_2, *) = I_7

action (2, *) = S_7

I_3 :

I_4 : $F \to .(E)$

GOTO (I_4, ( ) = I_4

action (4, ( ) = S_4

$F \to .id$

action (4, id) = S_5

TRACE KTU

	id	+	*	(	)	$	E	T	F
	action						goto		
0	S_5			S_4			1	2	3
1		S_6				accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

TRACE KTU

I_5 :

I_6 :

I_7 :

I_8 : $F \to (E \cdot)$ $E \to E \cdot + T$ $\cdot id$

$(8,))$ $(8, +)$

I_9 :

I_2

$E \to T \cdot$

follow $(E) = \{), +, \$ \}$

action $[2,)] = r_2$

action $[2, \$] = r_2$

action $[2, +] = r_2$

action $[3, \$] = r_4$ ← action $[3,)] = r_4$

I_1 (Special case)

$E' \to E \cdot$ (augmentation case)

action $[1, \$]$ = accept

I_3

action $[3, *] = r_4$

follow $(T) = \{ +,), \$, * \}$

action $[3, +] = r_4$

I_9 $E \to E+T.$

follow(E) = { $, +,) }

action[9, $] = r_1

action[9, +] = r_1

action[9,)] = r_1

I_5 $F \to id.$

follow

I_{10}. $T \to T*F.$

follow(T) = { +,), $, * }.

action [10, +] = r_3

action [10,)] = r_3

action [10, $] = r_3

action [10, *] = r_3.

Algorithm

i/p : C_1 the canonical collection of sets of items for an augmented grammar G.

o/p : If possible, an LR parsing table consisting of a parsing action function ACTION and goto fn. GOTO

Method : Let $C = \{I_0, I_1, \dots I_n\}$ The states of the parser are 0, 1 ... n state i being constructed from I_i.

The parsing actions for state i are determined as follows

If $[A \to \alpha . a\beta]$ is in I_i and GOTO $(I_i, a) = I_j$, then set ACTION [i, a] to shift j. Here a is a terminal.

If $[A \to \alpha .]$ is in I_i then set ACTION [i, a] to "reduce $A \to \alpha$" for all a in FOLLOW(A)

If $[S' \to S.]$ is in I_i, then set ACTION [i, $] to "accept"

TRACE KTU

4) If GOTO $(I_i, A) = I_j$, then GOTO $[i, A] = j$

5) All entries not defined by rule (1) through (4) are made "error"

6) The initial state of the parser is the one constructed from the set of items containing $[S' \to S]$.

CLR parsing

check whether the grammar is SLR(1) or not ? is same as SLR parsing table.

string parsing in SLR table

stack	i/p
0	id*id+id $
0 id 5	*id+id $
0 F 3	*id+id $
0 T 2	*id+id $
0 T 2 * 7	id+id $
0 T 2 * 7 id 5	+id $
0 T 2 * 7 F 10	+id $
0 T 2	+id $
0 E 1	+id $
0 E 1 + 6	id $
0 E 1 + 6 id 5	$
0 E 1 + 6 F 3	$
0 E 1 + 6 T 9	$
0 E 1	$ accept

TRACE KTU

CLR parsing (No. of states is high)

$S \to CC$

$C \to cC / d$

1) $S \to CC$
2) $C \to cC$
3) $C \to d$

first(S) = first(C) = { c, d }

follow(S) = { $ }

follow(C) = { $, c, d }.

I_0 : $S' \to \cdot S, \$$

$S \to \cdot CC, \$$

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

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

$\Rightarrow S, C, c, d$

I_1 : GOTO (I_0, S)

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

I_2 : GOTO (I_0, C)

$S \to C \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

I_3 : GOTO (I_0, c)

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

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

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

I_4 : GOTO (I_0, d)

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

$\Rightarrow C, c, d$

I_5 : GOTO (I_2, C)

$S \to CC \cdot, \$$

I_6 : GOTO (I_2, c)

$C \to c \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

I_7 : GOTO (I_2, d)

$C \to d \cdot, \$$

TRACE KTU

I_8 : $\Rightarrow$ c, c, d

I_8 : GOTO (I_3, C)

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

I_3 : GOTO (I_3, C)

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

$C \to \cdot CC$, c/d

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

I_4 : GOTO (I_3, d)

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

$\Rightarrow$ C, c, d

I_9 : GOTO (I_6, C)

$C \to cC\cdot$, $

I_6 : GOTO (I_6, c)

$C \to c\cdot C$, $

$C \to \cdot cC$, $

$C \to \cdot d$, $

I_7 : GOTO (I_6, d)

$C \to d\cdot$, $

*$\Rightarrow$ $E \to \cdot C$ (+T, a)

$C \to d$, +

B. (first(+T, a) = +)

TRACE KTU

	ACTION			GOTO	
	c	d	$	S	C
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					

I_4 : $C \to d \cdot, c/d$

follow (C) = {$, c, d}

r_3

I_5 : $S' \to C C \cdot, \$$

S

I_7 : $C \to d \cdot, \cancel{\{\$, c, d\}}$

$\$ \Rightarrow r_3$.

3.

	ACTION			GO TO	
	c (small)	d	$	S	C (capital)
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

TRACE KTU

Algorithm for Construction of CLR parsing table.

i/p : A grammar G augmented by production $S' \rightarrow S$

o/p : If possible the canonical LR parsing action function ACTION and go to function GOTO.

Method

1) Construct $C = \{I_0, I_1 \ldots I_n\}$, the collection of sets of LR(1) items for G.

2) state i of the parser is constructed from I_i. The parsing actions for state i are determined as follows

a) If $[A \rightarrow \alpha . a\beta, b]$ is in I_i, and GOTO $[I_i, a] = j$, then set ACTION $[i, a]$ to shift "j"

b) If $[A \rightarrow \alpha ., a]$ is in I_i, then set ACTION $[i, a]$ to "reduce $A \rightarrow \alpha$".

c) If $[S' \rightarrow S., \$]$ is in I_i then set ACTION $[i, \$]$ to "accept".

If a conflict results from the above rules, the grammar is not said to be LR(1) and the algorithm is said to fail.

3) The go to transitions for state i are determined as follows. If GOTO $(I_i, A) = I_j$, the GOTO $[i, A] = j$

4) All entries not defined by rules (2) through (3) are made "error".

TRACE KTU

5) The initial state of the parser is the one constructed from the set containing item $[S' \to \cdot S, \$]$

LALR

① Construct an LALR parsing table for following grammar

$S \to CC$
$C \to cC$
$C \to d$.

I_0 : $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

$\Rightarrow S, C, c, d$

I_1 : GOTO (I_0, S)
$S' \to S \cdot, \$$

I_2 : GOTO (I_0, C)
$S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

I_3 : GOTO (I_0, c)
$C \to c \cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

I_4 : GOTO (I_0, d)
$C \to d \cdot, c/d$

$\Rightarrow C, c, d$

I_5 : GOTO (I_2, C)
$S \to CC \cdot, \$$

I_6 : GOTO (I_2, c)
$C \to c \cdot C, \$$

I_{36} : $C \to c \cdot C, c/d/\$$
$C \to \cdot cC, c/d/\$$
$C \to \cdot d, c/d/\$$

I_{47} : $C \to d \cdot, c/d/\$$

I_{89} : $C \to cC \cdot, c/d/\$$

TRACE KTU

	c(small)	d	$	S	C
0	s_{36}	s_{47}		1	2
1			accept		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Construction of LALR parsing table

i/p : A grammar G augmented by production $S' \to S$

o/p : The LALR parsing tables ACTION and GOTO

Method

1. Construct $C = \{I_0, I_1, \ldots I_n\}$, the Collection of sets of LR(1) items.

TRACE KTU

2. For each case present among the sets of LR(1) items, find all set having that case, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots J_m\}$ be the resulting sets of LR(1) items. The parsing action for state i are constructed from J_i in the same manner. If there is a parsing action conflicting, the algorithm fails to produce a parser, and the grammar is said not to be LALR (1).

4. The GOTO table is constructed as follows. If J is the

union of one or more.

Sets of LR (1) items is $i.e.$ $I_0, I_1 \cup I_2 \cup \ldots \cup I_n$, then the cases of GOTO (I_1, x), GOTO (I_2, x) ... GOTO (I_n, x) are the same, since $I_1, I_2, \ldots I_n$ all have the same case. Let k be the union of all sets of items having the same case or GOTO (I_n, x). Then GOTO $(I, x) = k$.

The table problems by this algorithm is called the LALR parsing table for G. If there are no parsing action conflicts, then the grammar is said to be an LALR (1) grammar.

TRACE KTU

Module 5

3 Address Code

The sequence of statements, general form is A = B op C where A, B, C are programmable device names/operands op stands for any operation. Reason for the name 3 address code that each statement contains 2 for operands and 1 for result.

Expressions like $x+y*z$ will be reformulated as

$t_1 = y*z$

$t_2 = x+t_1$

where t_1, t_2 are compiler generated temporary names.

TRACE KTU

Additional 3 address statements

1) Assignment of the form A = B op C
2) A = op C

where op is a unary minus or logical negation or a shift operation.

3) Unconditional jump and go to : The meaning of these instructions is to execute a 3 address statement
4) Indexed assignments of the form A = B[i], A[i] = B
5) Pointer assignments of the form A = address[B], A = *B

etc.

3 address codes can be implemented in of the following way

(i) Quadruple

(ii) Triple

Quadruple

In quadruple form of 3 add. code representation, we may use a record structure of 4 fields, arg1, arg2, operator and result

A = −B *(C + D) would be translated to 3 add. code using quadruple as follows

$T_1 = -B$

$T_2 = C + D$

$T_3 = T_1 * T_2$

$A = T_3$

(pointers)	operator	Arg1	Arg2	Result
(0)	unary minus	B	−	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	=	T_3	−	A

Triple

	Op	Arg1	Arg 2
(0)	uni	B	—
(1)	+	C	D
(2)	*	(0)	(0), (1)
(3)	=	A	(2)

TRACE KTU

To avoid entering temporary names in to symbol table one can ~~assign~~ allow the statement computing temporary value to represent that value. Now a 3 address statement is represented by a structure with only 3 fields, operator arg1, arg2. Since 3 fields are used this intermediate code format is called triple.

Indirect triple

Another implementation of 3 add. code which has been considered that listing pointers to triples rather than listing triples themselves. It is called indirect triple.

	statement
(6)	(0)
(7)	(1)
(8)	(2)
(9)	(3)

TRACE KTU

Assignment

① Construct an SLR parsing table for the following CFG

$S \to L=R$

$S \to R$

$L \to *R$

$L \to id$

$R \to L$

② Construct predictive parsing table for the following CFG. check whether it is LL(1) or not?

$E \to TE'$ $\quad T \to FT'$ $\quad F \to PF'$

$E' \to +E / \epsilon$ $\quad T' \to T / \epsilon$ $\quad F' \to *F / \epsilon$

$P \to (E) / a / b / \epsilon$

③ Check whether the following grammar is in LL(1) or not?

$S \rightarrow aAc / bB$

$A \rightarrow Abc / Abd / c$

$B \rightarrow f / g$

$C \rightarrow h / i$

④ Construct LR(1) items (LR(0) + lookahead) and then construct CLR parsing table for the following grammar.

$S \rightarrow Ba / bBC / dc / bda$

$B \rightarrow d$

⑤ Construct LR(0) items (no lookahead) and then check whether the grammar is SLR(1) or not for the following CFG

$S \rightarrow SA / A$

$A \rightarrow a$

⑥ Construct a LALR parsing table for the following grammar

$S \rightarrow aAd / bBd / aBc / bAc$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

⑦ Construct CLR parsing table for the following CFG. where / is a terminal

$R \rightarrow RR$

$R \rightarrow R/R$

$R \rightarrow R*$

$R \rightarrow \epsilon$

$R \rightarrow (R)$

$R \rightarrow \cdot$

⑧ Explain various source language issues, storage allocation strategies and storage organization

TRACE KTU

(1) $E \to E+T \,/\, T$

$T \to T*F / F$

$F \to (E) \,/\, id$

1. $E \to E+T$
2. $E \to T$
3. $T \to T*F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

first(E) = first(T) = first(F) = {(, id}

follow(E) = { $, +,) }

follow(T) = { $, +,), * }

follow(F) = { $, +,), * }

I_0 :

$E' \to \cdot E, \$$

$E \to \cdot E+T, \$, +$

$E \to \cdot T, +, \$$

$T \to \cdot T*F, *, \$, +$

$T \to \cdot F, *, \$, +$

$F \to \cdot (E), \cancel{*}\ *\ \ \$, +, *$

$F \to \cdot id, \cancel{*}\ \ \$, +, *$

TRACE KTU

(7) $R' \to \cdot R, \$$

$R \to \cdot RR, \$ (/*$

$R \to \cdot R|R, \$ (/*$

$R \to \cdot R*, \$ (/*$

$R \to \cdot, \$ (/*$

$R \to \cdot (R), \$ (/*$

Syntax Directed Translation

This is an extension of CFG. It is a notational framework for intermediate Code generation. It is obtained by attaching semantic actions to the production of CFG. This is useful because it enables the Compiler designer to express the generation of intermediate Code directly in terms of Syntatic structure of source language. Semantic actions are enclosed in brackets { } and it appears after the productions.

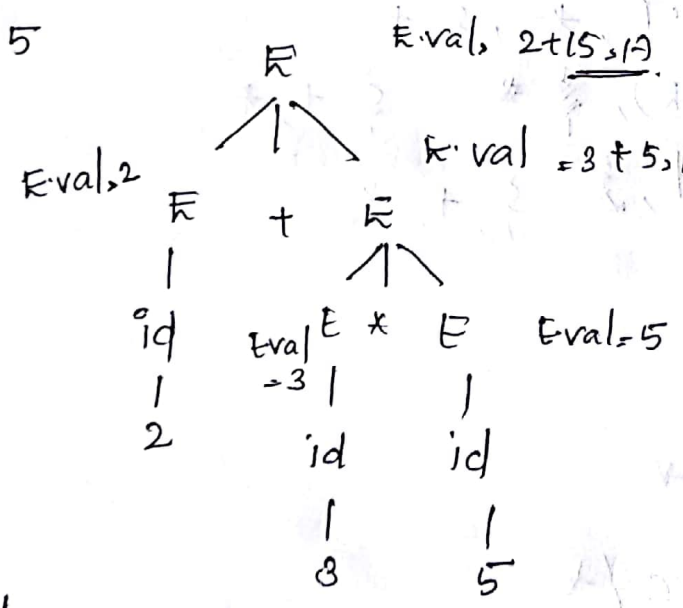
Eg: $E \rightarrow E^1 + E^2 \quad \{E.val = E^1.val + E^2.val\}$

$E \rightarrow E^1 * E^2 \quad \{E.val = E^1.val * E^2.val\}$

$E \rightarrow id \quad \{E.val = id.val\}$

TRACE KTU

2 + 3 * 5



Translation of Assignment statements

The following grammar describes the form of assignment statements

$A \rightarrow id = E$

$E \rightarrow E + E \;/\; E * E \;/\; id$

where A stands for an assignment statement, translation of E be a structure with two fields E.place, E.code (E.value, E.address)

where E.place holds the value of expression and E.code is a 3 address code evaluating the expression.

1. $A \rightarrow id = E$
2. $E \rightarrow E^1 + E^2$
3. $E \rightarrow E^1 * E^2$
4. $E \rightarrow id$

Semantic Actions

1. { A.code = E.code || id.place || = || E.place }

   T = New Temp()

2. { E.place = T

   E.code = $E^{(1)}$.code || E^2.code || E^1.place || + || E^2.place }

   T = new Temp()

3. { E.place = T

   E.code = E^1.code || E^2.code || E^1.place || * || E^2.place }.

4. { E.place = id.place

   E.code = null

   }.

TRACE KTU

1) $A \to \underbrace{Abc}_{A\ \alpha} \mid \underbrace{Abd}_{A\ \alpha} \mid \underbrace{c}_{\beta}$

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$A \to cA'$

$A' \to bcA' \mid \epsilon \mid bdA'$

$A \to cA'$

$A' \to bdA'$

2) Find first and follow

$E \to TE'$

$E' \to +E \mid \epsilon$

$T \to FT'$

$T' \to T \mid \epsilon$

$F \to PF'$

$F' \to *F' \mid \epsilon$

$P \to (E) \mid a \mid b \mid \epsilon$

TRACE KTU

FIRST(E) = FIRST(T) = FIRST(F) = FIRST(P) = ~~FIRST(F)~~ = {c, a, b, ε, first

FIRST(E') = {+, ε}

FIRST(T') = {c, a, b, ε, *}

FIRST(F') = {*, ε}

FOLLOW(E) = {$,)}

FOLLOW(T) = {+, $,)}

FOLLOW(E') = {$,)}

FOLLOW(T') = {+, $,)}

FOLLOW(F) = {c, a

FOLLOW(F') = {c, a,

FOLLOW(P) = {*

~~FOLLOW(P')~~

* $S \rightarrow AB$

$A \rightarrow a/\epsilon$

$B \rightarrow b$

$first(S) = \{ first(A), first(B) \}$

$= \{ a, \epsilon, b \}$

* $S \rightarrow ABC$

$A \rightarrow a/\epsilon$

$B \rightarrow b/\epsilon$

$C \rightarrow c/\epsilon$

$First(S) = \{ first(A), first(B), first(C) \}$

$= \{ a, \epsilon, b, c \}$

TRACE KTU

1) $A \rightarrow CxA/\epsilon$

$C \rightarrow xBx/z$

$z \rightarrow xCy$.

$first(A) = first(C) = \{ x \}, \epsilon \quad = \{ x, \epsilon \}$.

$first(z) = \{ x \}$.

$follow(A) = \{ \$ \}$

$follow(C) = \{ x, y \}$. $\quad follow(B) = \{ x \}$.

$follow(z) = \{ x, y \}$

	x	y	$\$$
A	$A \to xA$		$A \to \epsilon$
B			
C	$C \to x$ $C \to xBx$	$C \to z$	
Z	$Z \to xCy$		

Translation of Boolean Expression

Boolean expressions are composed of Boolean operators such as AND, OR etc.

production	Semantic Rule
$B \to B_1 \,\|\|\, B_2$	$B_1.True = B.True$ $B_1.False = newLabel()$ $B_2.True = B.True$ $B_2.False = B.False$ $B.code = B_1.code \,\|\|\, B_2.code \,\|\|\, label(B_1.False)$
$B \to B_1 \,\&\&\, B_2$	$B_1.True = newLabel()$ $B_1.False = B.False$ $B_2.True = B.True$ $B_2.False = B.False$ $B.code = B_1.code \,\|\|\, B_2.code \,\|\|\, label(B_1.True)$

TRACE KTU

$B \rightarrow !B_1$

$B_1.True = B.False$

$B_1.False = B.True$

$B.code = B_1.code$

Module-6

Code optimization & code generation

Code optimization

Improving code for increasing the efficiency of target language. Finding certain pattern and replace it with some efficient constructs.

Registers - fastest m/y elements.

Richest source of optimization is efficient use of registers.

TRACE KTU

Optimization refers to techniques a compiler can employ to produce a better object language prgm.

Principle sources of opti^n

Code optim^n are applied after syntax analysis. Technique consists of detecting patterns in the prgm and replacing it by efficient pgm constructs. The richest source of opti^n is the efficient & utilization of registers and inst^n set machine.

Inner loops

90-10^ rule states that 90% of time is spend in 10% of

code, thus the most heavily travelled part of program, inner loops are an obvious target for optimization.

Further optimizations

Another imp. source of optimization is common sub expression elimination

```
main ()
{
  ==========
  for (i=1; i<=100; i++)
  {
    ==========
    ==========
  }
  ==========
}
```

① $A = (B+C)+D$

$B = E+(B+C)$

② $A[i+1] = B[i+1]$

$j = i+1$

$A[j] = B[j]$

So computation get reduced and we reuse the value of $(i+1)$.

Loop optimization

```
prod = 0;
i = 1;
while (i ≤ 20)
{
prod = prod + A[i] * B[i];
i = i+1;
}
```

optimized 3 address code is ⇑

1) prod = 0
2) i = 1
3) $T_1 = 4*i$ address(A)
4) $T_2 = add(A) - 4$
5) $T_3 = T_2[T_1]$
6) $T_4 = add(B) - 4$
7) $T_5 = T_4[T_1]$
8) $T_6 = T_3 * T_5$
9) Prod = prod + T_6
10) $I = I+1$
11) if $I \le 20$ go to (3)

TRACE KTU

eg:- $i = 1$ A = 1000

$T_1 = 4$ B = 2000

$T_2 = 996$

$T_3 = 1000$ — (1)

$T_4 = 1996$

$T_5 = 2000$ — (1)

$T_6 = 1000 \times 2000$

$i = 2$

$T_1 = 8$

$T_2 = 996$

$T_3 = 996 + 8 = 1004$

32 bits

0	0th	1st	2nd	3rd
4	4	5	6	7
8	8	9	10	11

* Basic Block

TRACE KTU

Sequence of instructions where jump and goto are absent in b/w those instruction.

Starting instruction of basic block is called leader.

B_1 [prod = 0, i = 1] → leader

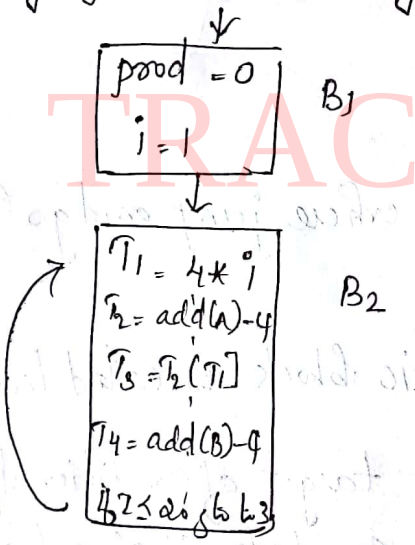
target of conditional/unconditional can be treat as a leader.

B_2 [$T_1 = 4 * i$ ⋮ if $T \le 20$] → leader

It is the sequence of consecutive statements which may be entered on only at begining when entered or exit without hold/jump except at the end. Each basic block has a leader. Generally first statement is a leader, any statement that is the target of conditional, unconditional jump/goto is also a leader.

Flow graph

Basic blocks and its successor relations are shown by means of graph called flow graph.



Directed acyclic graph (DAG)

DAG is a data structure ~~after~~ automatically analyzing basic blocks. This shows how the value computed by each statement in a basic block is used in subsequent statements in the block.

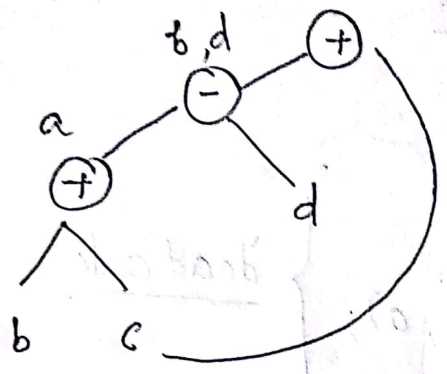
TRACE KTU

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$



Optimization of Basic Blocks

1) DAG representation of basic block
2) Finding Common Subexpressions
3) Dead Code Elimination
4) The use of algebraic identities
5) Representation of array reference.

TRACE KTU

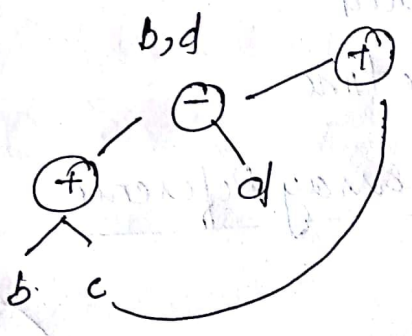
Finding Common Subexpressions

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$



$\Rightarrow$ $a = b + c$

$d = a - d$

$c = d + c$

Here replacing 'b' with d. Because d is latest label.

Dead Code Elimination

Eliminate dead code in building blocks for directed acyclic graphs.

Dead code :-

```
a=0;
if (a=-1)
{
printf("%d", a);
}.
```

} dead code

Use of algebraic Identities

Algebraic identities

$x+0 = 0+x = x$

$x \times 1 = 1 \times x = x$

$x-0 = x$

$x/1 = x$

reduction in strength

expensive	cheaper
x^2	$x \times x$
$2 \times x$	$x+x$
$x/2$	$x \times (0.5)$

TRACE KTU

Constant folding

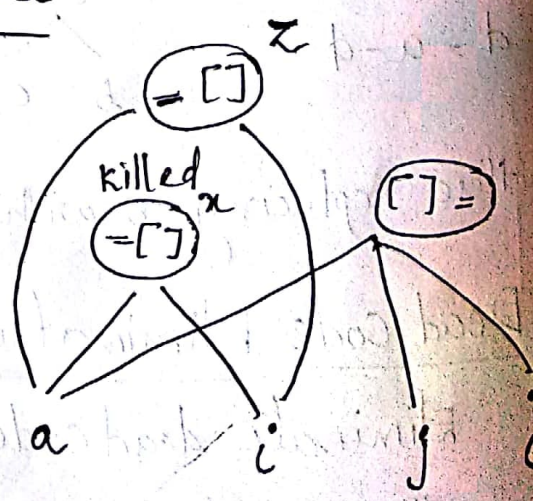
2×3.14 will be replaced by 6.28 at compile time

Representation of array Reference

eg: 1 $x = a[i]$

$a[j] = y$

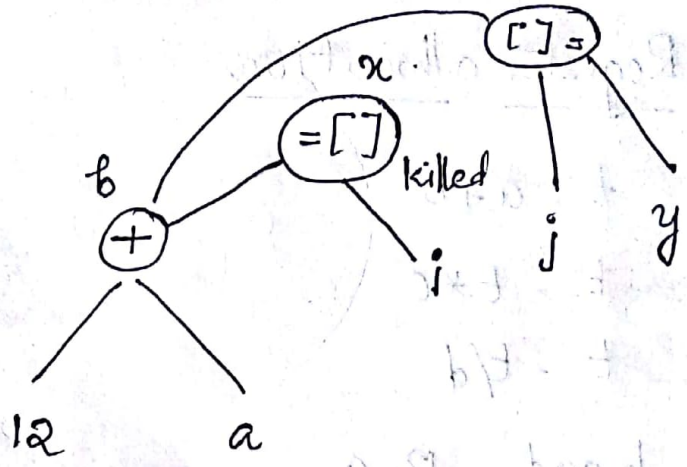
$z = a[i]$



eg: 2 $b = 12 + a$

$x = b[i]$

$b[j] = y$



Code generation

1) Instruction selection
2) Register allocation
3) Instruction ordering

Instruction selection

eg: $x = y + z$

Load R0, y

Add R0, R0, z

store x, R0

TRACE KTU

Issues in the design of code generator

1) Input to the code generator

3 address code is the input. Can be triple, quadruple etc.

2) Target program - Can be RISC or CISC

3) Instruction selection

a = b + 2	mov R0, b
	Inc R0
	Inc R0
	mov a, R0

	mov R0, b
	Add R0, #2
	mov a, R0
	mov c, # 0c

Register allocation

t = a+b

t = t*c

t = t/d

Load R1, a

ADD R1, b

MUL R0, c

DIV R1, d

STORE R1, t

Instruction ordering (NP Complete problem)

Load R1, a

Load R2, b

Add R1, R2

Load R3, c

Add R3, R2

Load R4, b b is calling home

Add R4, R2

MOV R4, d

ADD R4, R2

MOV R1, a

MOV R2, b

ADD R2, R1 (a+b)

MOV R3, d

ADD R2, R3 (b+d)

ADD R3, R1

MOV R3, c

ADD R3, R1

Here order the instruct for reducin the registe m/

* Simple Code Generator

4 uses of register

1) all operands / some should be in registers. c=a+

TRACE KTU

2) Make good temporaries $c = a+b+d \Rightarrow$ $t_1 = a+b$, $t_2 = t_1+d$, $t_3 = t_2$ → temporary registers

3) used to hold values

4) run time storage management.
   stack pointer is a register

① Register and ~~address~~ Descriptions

② Code generation algorithm

③ design of get Reg ()

① Register and ~~address~~ Desciptors.
↓
x is a variable → R_1 register
↓
having add. descriptor [1000] pointing

x [5] 1000

$R_1 \leftarrow x$
↓
[1000]

② Code generation algorithm

① machine instruction for operation

② managing register and ~~address~~ descriptors

① GetReg() is a function supplying necessary registers for a program.

c = a+b;

MOV Ro, a
ADD Ro, b
store c, Ro

getReg (c = a+b)

Ra, Rb, Rc
load Ra, a
Load Rb, b
ADD Ro, Rb
STORE Rc, Ra

TRACE KTU

(2)

(3) Design of getreg()

$x = y + z$

(i) if y is a reg

TRACE KTU

Module 4

Syntax directed definitions (SDD)

syntax directed translation (refer)

Synthesized attribute and inherited attribute

$E \rightarrow A + S$

E is dependend on both A and S, so they are called synthesized attribute

$E \rightarrow AS$

$A \rightarrow aE$

$S \rightarrow aA$

A is calculated by the value of parent. A and S are both inherited attribute. A is the sibling of S.

If a variable is dependent on parent and sibling, then it is inherited.

$A \rightarrow BCD$

$[x_i]$ is inherited only when it have dependency $[x_{i-1}]$

Synthesized attributes only then it is 'S' attributed definition

Inherited attributed only, then it is (L) attributed definition

DFS from left to right

Bottom up evaluation of S attributed definitions

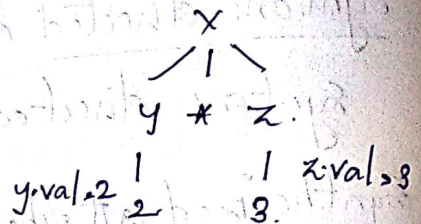
Implementing using a stack:

TRACE KTU

stack	value
y	2
z	3
x	6

$x \to y * z$

$\{ x.val = y.val * z.val \}$



eg: $L \to En$ {print val [top]}

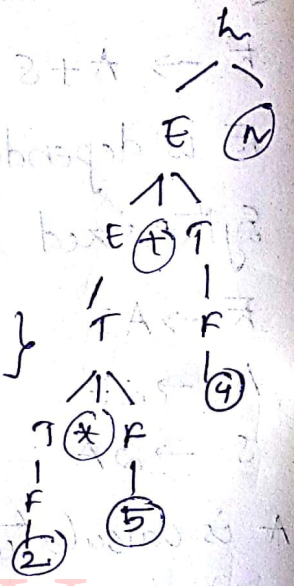
$E \to E+T$ $\{ E.val = E.val + T.val \}$

$E \to T$ $\{ E.val = T.val \}$

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

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

$F \to digit$



TRACE KTU

Input	stack	val	production
3*5+4n	–	–	–
*5+4n	3	3	–
*5+4n	F	3	F→digit
*5+4n	T	3	T→F
5+4n	T*	3_	–
+4n	T*5	3_5	–
+4n	T*F	3_5	F→id
+4n	T	15	
+4n	E	15	T→T*F E→T

4N	E+	15	T → T*F
N	E+4	15	—
N	E+F	15	—
N	E+T	15	—
N	E	15	E → E+T

L-attributed Definition

A → B C D

B → —

A → LM { L.i = l(A.i)
M.i = m(L.s)
A.s = f(M.s) }

A → QR { R.i = r(A.i)
$\boxed{Q.i = q(R.s)}$ not possible
A.s = f(Q.s) } .

no here inheritence is not valid.
∴ not an L-attributed

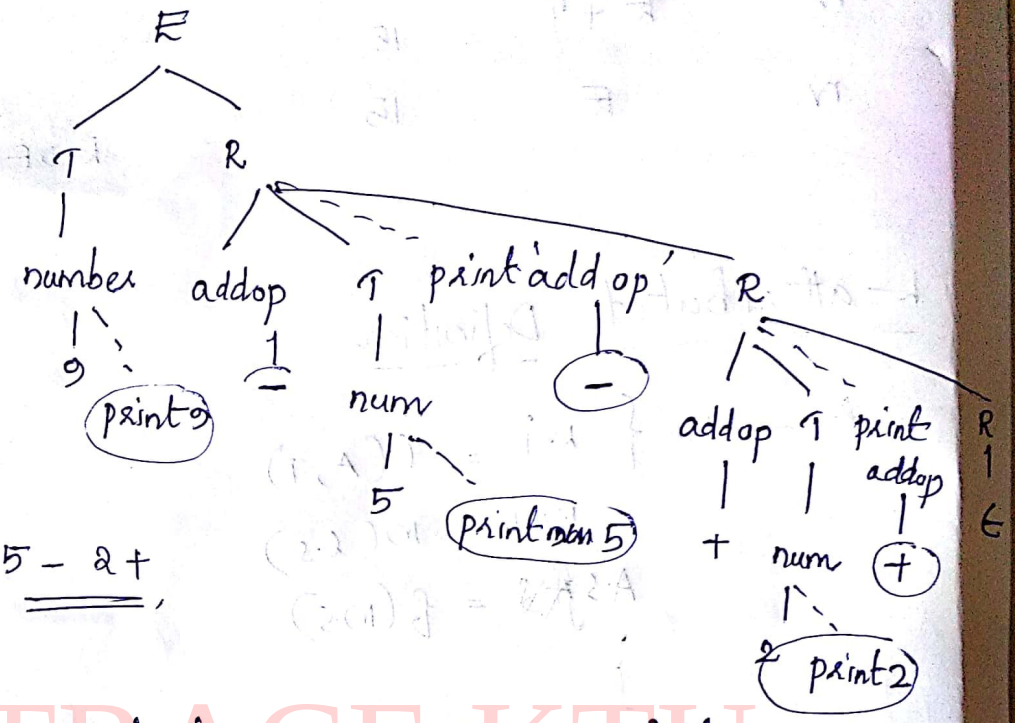
A simple translation scheme that maps infix expression with addition and subtraction into corresponding post fix expressions.

eg: E → TR { }
R → add op T { print 'add op' } R/ε
T → num { print (num.val) }

TRACE KTU

infix to postfix

string $(9-5+2) = (95-2+)$



O/p : 95 - 2 +

TRACE KTU

Bottom up evaluation of inherited attributes

$E \rightarrow TR$

$R \rightarrow +T\{print '+'\}R / -T\{print '-'\}R / \epsilon$

$T \rightarrow num\{print 'num.val'\}$

$E \rightarrow TR$ markup non-terminal

$R \rightarrow +TMR / -TNR / \epsilon$

$T \rightarrow num\{print 'num.val'\}$

$M \rightarrow \epsilon\{print '+'\}$

$N \rightarrow \epsilon\{print '-'\}$

Eg: $D \rightarrow T\ L\ \{L.in = T.type\}$

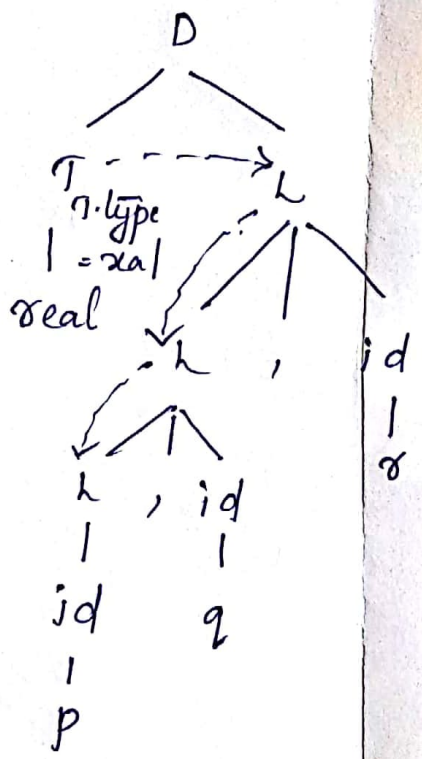
$T \rightarrow int\ \{T.type = integer\}$

$T \rightarrow real\ \{T.type = real\}$

$L \rightarrow L, id\ \{L_1.in = L.in\}$

$L \rightarrow id$

i/p $\rightarrow$ real p, q, r



i/p	state	production
real p, q, r	—	—
p, q, r	real	
p, q, r	T	~~T→rea~~ T→real
, q, r	T p → id	
, q, r	T L	L→id
q, r	T L,	—
, r	T L, q	
, r	T L, q	
, r	T L	L→L, id
r	T L,	—
—	T L, r	—

TRACE KTU

—	Th	$h \to h, id$
—	D	$D \to Th$

TRACE KTU