BSCS

compiler construction I

504

6 - August - 2019

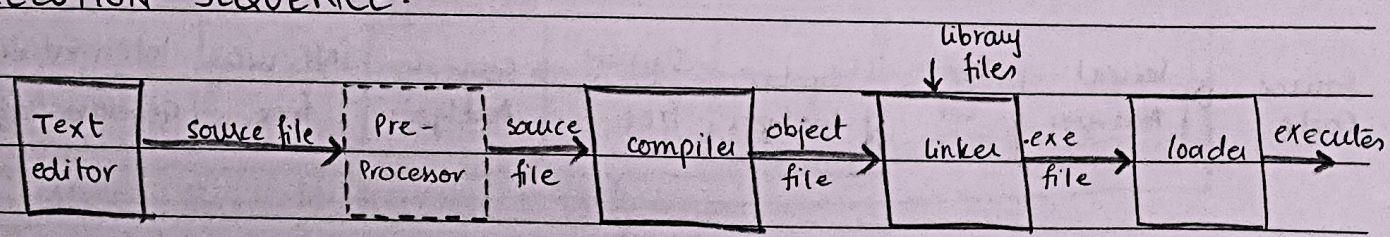
BOOK: Compiler construction – Louden.
Compiler – ULLMAN (2nd ed)

COMPILER: - Converts high level language to low level language.
- It identifies error (error detection) and reports error
- Compiler does NOT execute.
Execution is done by the processor

ERRORS: 1. Exceptions (Runtime error): detected during execution.
2. Compile time error: detected by compiler
3. logical errors : errors due to programming logic.

- Optimization is done by compiler:
1. Space optimization
2. Time optimization.
- Different compilers required for high level and low level language.
- When architecture or machine changes, compiler changes.
- There is no comparision between compiler and interpreter
- Interpreter takes line by line input, stops if error, else executes.
- Interpreter debugs, compiler does not. (assembler works like compiler)
- Compiled and interpreted languages (python, matlab) are different

EXECUTION SEQUENCE:

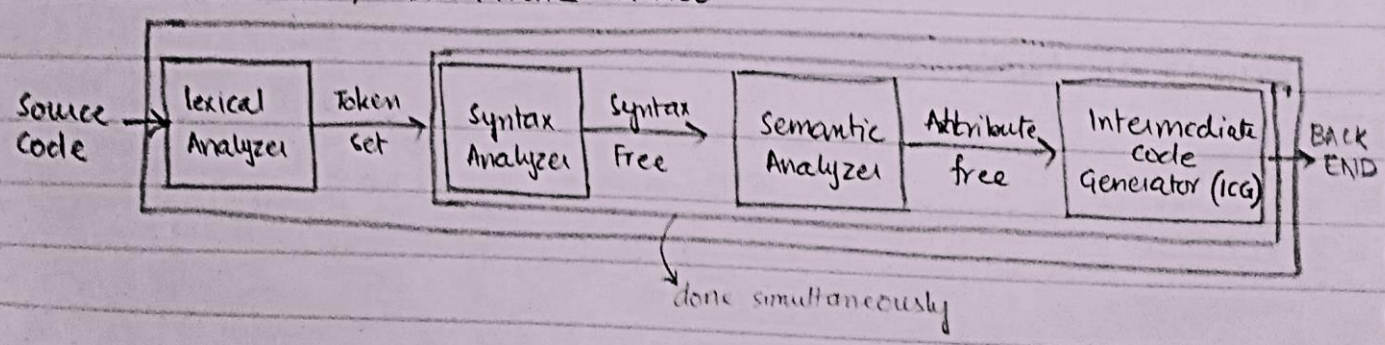


PERB 12.

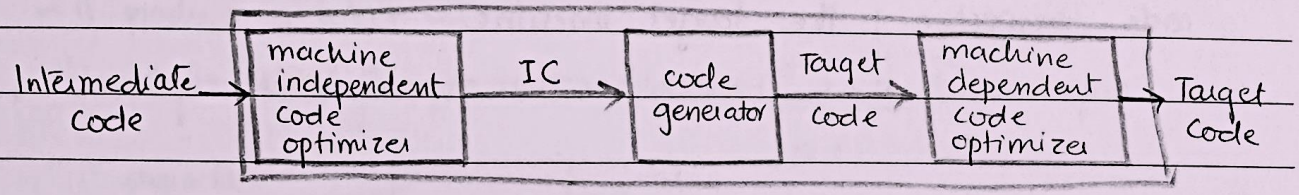
- Directives are pre-compiled directions (e.g: .stack, .data, .code, end main etc that can allocate space in memory) that gives directions on how the code will be interpreted.
- Pre-processor removes the directives and replaces them with corresponding values, converts into pure source file.
- object files are not executable as they don't have library files.
- linker adds library files which are precompiled and creates an executable file.
- loader is a module of OS which executes code : it loads file from secondary memory to primary memory (RAM) as processor only executes what is in primary memory.
- compiler assigns RELATIVE ADDRESS to all kinds of variables etc. loader changes the relative address to PHYSICAL ABSOLUTE ADDRESS.
- For any program, there are 4 kinds of memory available:
  1. code
  2. Static Data (created once, global/static variables)
  3. Heap (dynamic variables, objects)
  4. Stack (function's local data, local variables)

PHASES OF A COMPILER.

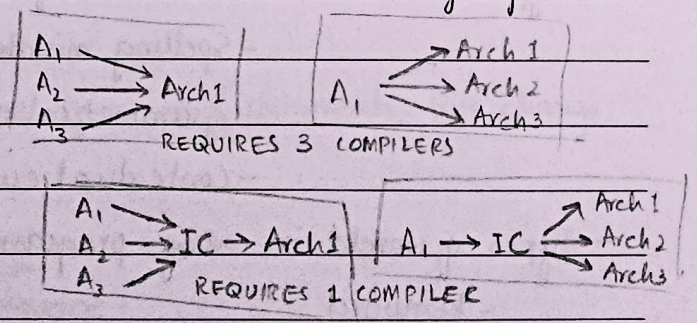
FRONT END / ANALYTICAL PHASE



BACKEND / SYNTHESIS PHASE



- Lexical analyzer reads the source file (a stream of characters) character by character. If no word is invalid, then it gives output in the form of TOKEN SET. (checks validity through RE's and FAs) (spelling mistake or any word which does not exist in dictionary is LEXICAL ERROR)
- Syntax analyzer checks grammar. Context free grammar is designed and checked. The input is a sequence of token set.
- Semantic analyzer checks grammar in ~~co~~ grammar free context. (e.g: int a; float a; same name for two variables is error)
- Multilanguage platforms use intermediate code before source file is ~~connected~~ converted to target code. Intermediate code allows us to acheive language independence.



Intermediate code:

1. Easy to translate
2. Avoid repetitions
3. Easier to generate
4. Achieve language and machine independence

MSIL: Microsoft Intermediate language.

- Machine independent code optimizer optimizes intermediate code irrespective of the target machine.

e.g: a = b + c; → identifies that this line is unuseful

b = c * d;

a = b + d;

- Code generator converts intermediate code to target code language. For this, the target architecture should be known.
- Machine dependent code optimizer manages space, registers and applies code rearrangement to produce optimized target code.

Correctness is most important for code and shouldn't be affected during optimization.

- Token is a combination of a $\boxed{\text{valid word + class name}}$

- Types of errors in Natural language vs. Artificial language:

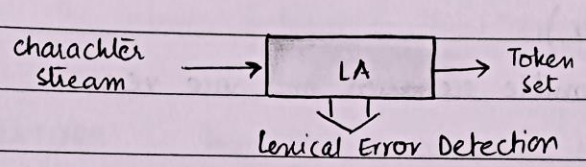
Natural language	Artificial language
- Spelling mistake	- lexical error
- grammatical error	- syntax error
- Contextual error	- semantic error.

- Types of words in any programming language:
  - keyword
  - identifiers
  - constants / literals
  - punctuators
  - operators.

20- August - 2019

LEXICAL ANALYZER

- It produces a token set
- Input is text / string



- reads character by character and breaks into words (when spaces, operator, punctuation occur; It is language specific)

What Lexical Analyzer does:

1. Scan / read file
2. Breaks into words
3. Validation of lexes
4. classify words

⇒ Token set

- There are certain substrings that are ignored (comments, line change, white spaces etc)

There are 5 ~~types of lexemes in a language~~ categories in a language:

- keywords
- Operators
- Punctuation
- Identifiers
- Constants

REB3/2.

- CLASSIFICATION OF LEXEMES:

RULE OF CLASSIFICATION: All those words that are syntactically replaceable belongs to same class. (i.e. replacement doesn't generate syntax error).

replaceable in all possible scenarios and vice versa.

- A function name can NEVER be a keyword.

Is int a = new int(); possible?

KEYWORS:

- DataTypes
- Void
- for
- foreach
- while
- do
- if
- else
- switch
- case
- default
- break
- continue
- return
- bool_const
- class
- struct
- interface
- access_modifiers
- static
- Virtual, Override
- Sealed (restricts further inheritance)
- abstract

QUERY: Can the word 'sealed' be used with functions in C#? if yes what concept.

How can we stop inheritance in C++?

PUNCTUATORS: (meaningful separators)

(punctuators are considered a word on its own)

, ; : :: { } ( ) [ ] . →

- all the above belong to their own separate class except last two

OPERATORS:

- syntactically replaceable
- Precedence and associativity

a+b+c left associative

a=b=c right associative

- unary operators (~ , ! || ++ , --) highest precedence

(Two classes: 'NOT', 'Inc/Dec' due to syntax of being prefix or postfix)

- MDM * / % (* is a unary operator while using C/C++)
- PM + -
- ROP < > <= >= != ==
- AND &&
- OR ||
- AND &
- OR |
- AOP = += -= *= /= %=

Select / create a language, design keywords, OOP inheritance, interface, struct, functions, features, methods of the language

PERB12.

27- August - 2019

Create a table for keywords and their classes, punctuators, operators

IDENTIFIERS:

$\Sigma = \{ alpha, digits, _ \}$

Created from the language above, usually.

e.g language:

- ID starts with either _ or alphabet
- First char can be followed by digit or alph.
- only _ is not a valid ID

R.E : $(_)(alpha + digit)^+ + (alpha)(alpha + digit)^*$

[rigix class library]

TASK: Explore and implement rigix.

CONSTANTS:

- Integer constants

$\Sigma = \{ digits, +, - \}$

$RE = ('+' + '-' + \Lambda)\ (digits)^+$

- Float constants

$\Sigma = \{ +, -, \cdot, digits \}$

$RE = ('+' + '-' + \Lambda)\ digits^* \cdot digits^+$

- Char constants

A = with or without \

n, \, t, b, 0, ..

B = with \
', \, "

C = without \
+, -, @, ...

RE = '[(C+A) + \(B+A)]'

- String Constant
  RE = " (anycharacter)* "
- every keyword is a valid identifier (in languages we students usually use)

SOURCE CODE

```
1  void main()
2  {
3     Int a=77+8abc +a_b_c;;
4     while (a<<<=b+++==c!=d)
5     if_else(a && bc.c=d*e)
6     char e="abc+=\\ab"abc+
7        =da+'\\'+'\n'+'a'+
8        'abc'+'+=6
```

Input = "void.... 6"

→ LA →

TOKEN SET

(void, -, 1)	(;, -, 3)
(main, -, 1)	(;, -, 3)
((, -, 1)	⋮
( ), -, 1)	(shift, «, 4)
({, -, 2)	(ROP, <=, 4)
(ID, Int, 3)	⋮
(ID, a, 3)	(string const, abc+=\\ab, 6)
(AOP, =, 3)	⋮
(int_const, 77, 3)	(char const, \\, 7)
(PM, +, 3)	⋮
(invalid lexeme, 8abc, 3)	(invalid lexeme, 'ab', 8)
(PM, +, 3)	(invalid lexeme, '+=', 8)
(ID, a_b_c, 3)	

Token has 3 parts (class , value, line #)

- if class / value is empty then its the same as class / value
- main is a special identifier
  (could become a keyword in your language, or could be semantically dealt with by having one main function, easier to deal syntactically)
- if line change character is used as terminator then a token for it is generated (e.g. python) otherwise it is ignored.
- LA also reports error, it firsts generates token
- For string and char the quotes are used for identification and not included in the value for the token.
- if there are no closing double quotes for string it stops when it gets a line change character, and then generates an error.
  (Invalid lexeme ," abc + = \\ abc + ', 6)
  double quotes are added to the string when there is error
- double quotes after a \ are treated as string
  any character after \ is treated as part of string.
- e.g: if there is a line change after \

LINE 6: char c" abc + = \" b + = c * d \

then it will be treated as a breaker.

- String has no specified length but character has, so break char according to length. (exception is the line change character)

String : a++='\=+'++5

Token set : (ID, a, 1)
(Increment, ++, 1)
(AOP, =, 1)
(invalid lexeme, '\=+', 1)
(char const, +, 1)
(increment, ++, 1)
(int const, 5, 1

COMMENTS:

- Single line comment //
- multi line comment /*

- Single line comment reads till it gets to the end of line.
- multi line ignores everything until it gets */ or gets to the end of code line change character would be ignored but would be used to increment line number.

99.97 ⟶ float

a.bc ⟶ a = ID, . = punct, bc = ID

9.9.abc ⟶ 9.9 = float . = punct, abc = ID

9.9bc ⟶ invalid word

9.abc95.bc.55.55 ⟶ 9 = int, . = punct, abc95 = ID,
. = punct = bc = ID, .55 = float
.55 = float

PERS 3/2.

- when . is read check if all previous are digits and the next character should be digit, then combine all next characters till a breaker is read.
- If the character after . is not digit then dot is a breaker.
- second . is always a breaker.
- flag can be used to check if temp has already has a .

- +/- ki logic khud karni hai :c

03 - September - 2019

SYNTAX ANALYZER:

- Only the class part is important and not the value part as all the values are syntactically replaceable
- To check syntax CONTEXT FREE GRAMMAR. (move according to sequence)
- stack / Recursion used for CFG.

CONTEXT FREE GRAMMAR:

- Terminals
- Non-Terminals <name_of_NT> ⟶ Production rule
- Production rules Terminal, non-terminal, both
- Starting Non-Terminal

- There is no counting technique in RE, so language e.g $a^n b^n$; $n>0$ can not be expressed as RE.
- Terminal is something that is already known / that we input (Tokens) CP
- Non-Terminal words etc that are unknown and we need to define it.
- Non-Terminals are defined through production rules.
- A starting point is required that would be non-terminal

e.g: int a;

int a, b

Non-Terminal as not defined in Token Set's class part.

<Dec> ⟶ DT, ID <LIST>

<LIST> ⟶ ;

<LIST> ⟶ , ID <LIST>

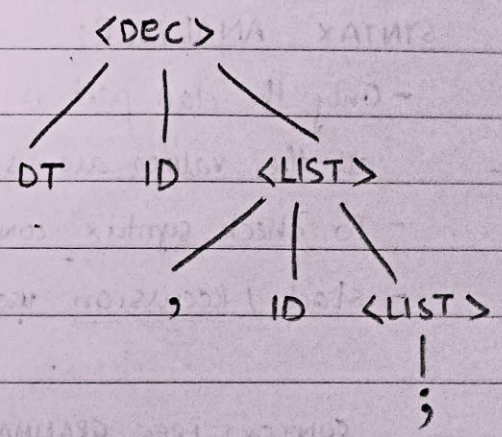
- Syntax / Pause Tree

  All terminals are leaf nodes

  All non-terminals ~~a~~ have branches

  - Match the sentence with production rules and if matched add it to the tree.

e.g DT ID , ID ;



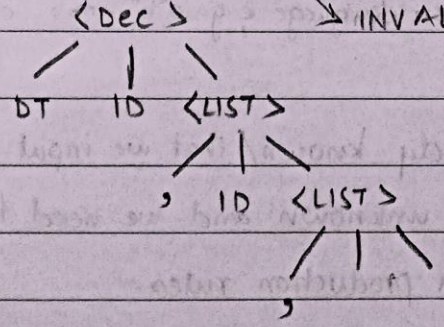
  - No branch should be unlabelled.

  - e.g: int a, b; b; ⟶ invalid as input isn't complete.

    For a valid input: • Tree should be complete

    • Input should be complete

  - e.g: int a, b, ⟶ INVALID as tree not complete



  - e.g: int a=5, b, c=b;

    int a=5=c;

    int a=b=c;

    int x, a=b=c=7;

<CONST> ⟶ int_const | float_const | char_const | string_const | bool_const

<Dec> → DT ID <LIST>

<LIST> ⟶ ;

<LIST> ⟶ , ID <LIST>

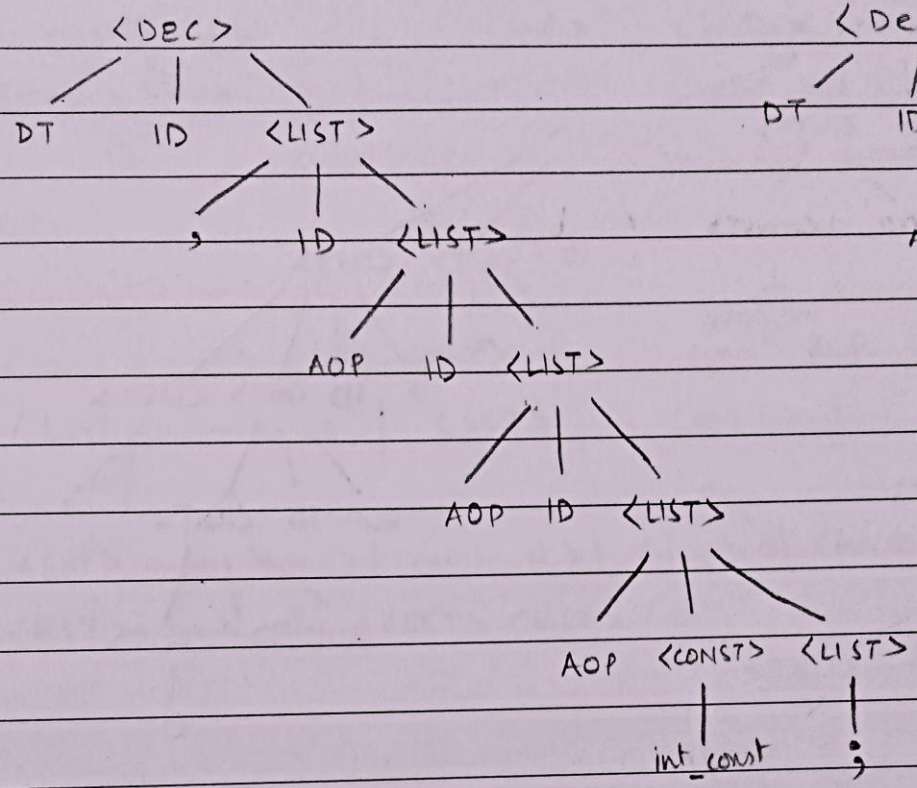
<LIST> ⟶ AOP ID <LIST>

<LIST> ⟶ AOP <CONST> <LT>

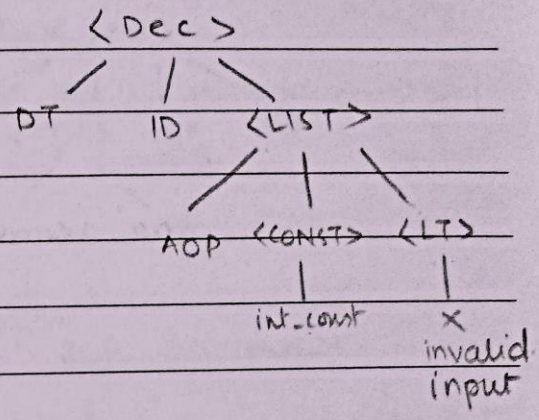
<LT> ⟶ ;

<LT> ⟶ , ID <LIST>

int x, a = b = c = 7;



int a = 5 = c;



DECLERATION:

<Dec> ⟶ DT ID <INIT> <LIST>

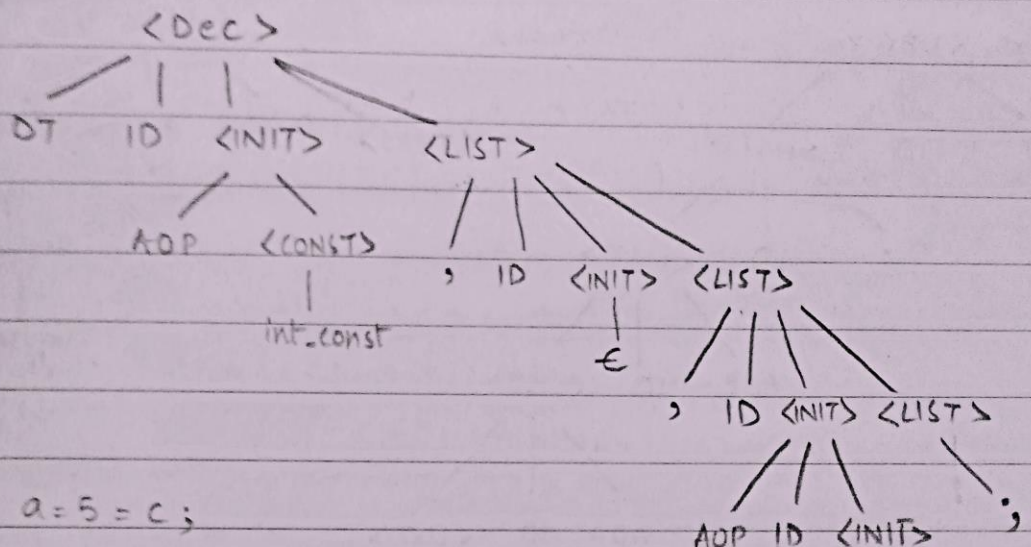
<INIT> ⟶ AOP ID <INIT>

<INIT> ⟶ AOP <CONST>

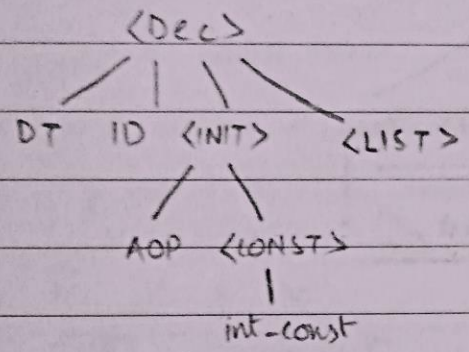
<INIT> ⟶ ∈ (null)

<LIST> ⟶ ; | , ID <INIT> <LIST>

e.g: int a = 5, b, c = b = d;



e.g: int a = 5 = c;



- while (condition) → ID, const, a < b, 9 < b, b < 9, 9 < 8
  - { multiple statements } (0 – many)
  - single statement
  - ;

<while_st> → while (<cond>) <body>

<cond> → ID | <const> | ID ROP ID | ID ROP <const> |
<const> ROP ID | <const> ROP <const>

OR:

<ID_const> → ID | <const>

<cond> → <ID_const> | <ID_const> ROP <ID_const>

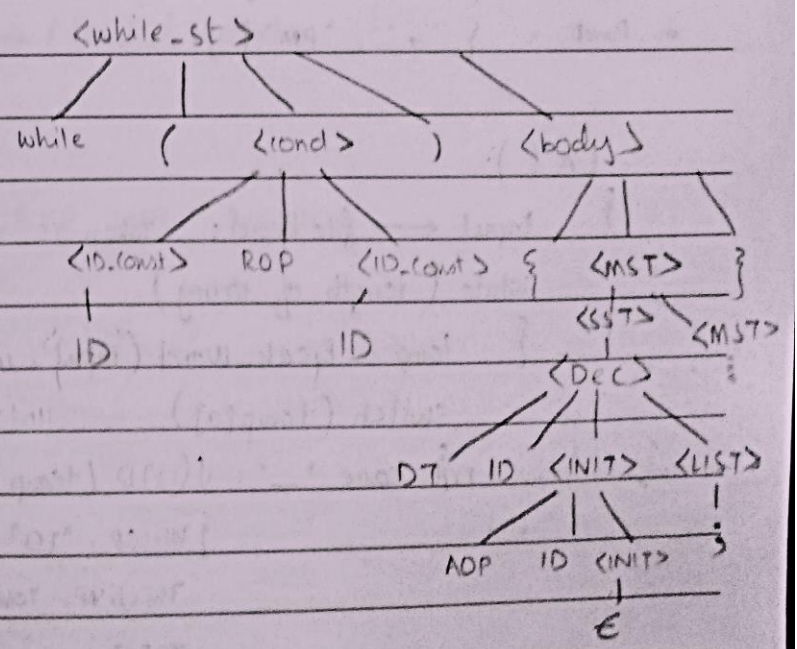
<body> → ; | <SST> | { <MST> }

<SST> → <Dec> | <while_st> | <if-else> | <for-st> | .....

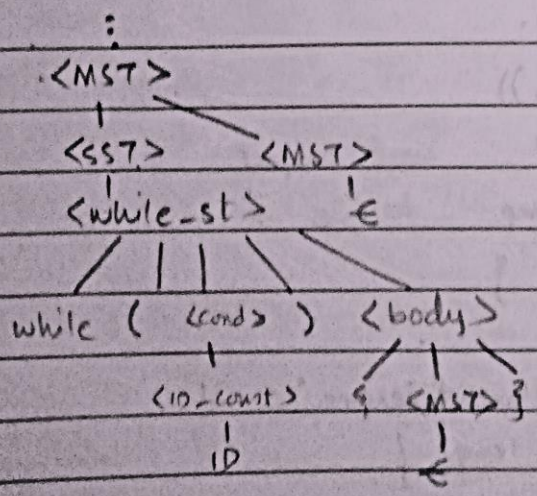
<MST> → ∈ | <SST> <MST>

e.g.:

```
while (a<b)
  { int a = b;
    while (b)
      {
      }
  }
```



⋮



IMPLEMENTATION OF LEXICAL ANALYZER: [File input & File output]

- String isKW (string word)
- isPunc
- isOpr [2D array]
- bool isStrConst
- bool isIntConst
- bool isFloatConst
- bool isChaConst
- bool isID

- KW = { "int", "DT" }, { "for", "for" } ....
- Opr = { "{", "{" }, ...
- Punct = { "+", "PM" }, ...

```
LA ()
{   Input ← file Read;    Token TS -> list
    While ( length of string )
    {   temp = Break_Word (Input, index)
        switch ( temp[0] )
        { case '_' : if(isID (temp))
                     {TS[i].CP = "ID"        and output
                      TS[i].VP = Temp        to file
                      TS[i].line = }
                     else
                     { TS[i].CP = "Invalid lexeme"
                       TS[i].VP = Temp }
```

IMPLEMENTATION OF LEXICAL ANALYZER: File input & File output

- String isKW (string word)
- isPunc
- isOpr [2D array]
- bool isStrConst
- bool isIntConst
- bool isFloatConst
- bool isChaConst
- bool isID

- KW = { "int", "DT" }, { "for", "for" }, ...
- Opr = { "{", "{" }, ...
- Punct = { "+", "PM" }, ...

```
LA ()
{   Input ← file Read;   Token TS → list
    While ( length of string )
    {   temp = Break_Word (Input, index)
        switch (temp[0])
        { case '_' : if(isID (temp))
                     {TS[i].CP = "ID"        and output
                      TS[i].VP = Temp        to file
                      TS[i].line = .  }
                     else
                     { TS[i].CP = "Invalid lexeme"
                       TS[i].VP = Temp  }
```

```
Case alph :   if ( isID (temp) )
              {CP = isKW (temp)
               if ( CP == " " )
               { TS[i]. CP = "ID"..}
               else
               { TS[i]. CP = CP...}}
              else
               {   invalid -> error   }
```

digit se start ⟶ integer / float

. ⟶ punctuator / float (length check)

+ - ⟶ integer / float / operator

SUBMISSION : 17th SEPTEMBER 2019 , Tuesday

24 - September - 2019

SYNTAX ANALYZER:

<do_while> → do {<MST>} while (<cond>);

<for_st> → for (<c1> ; <c2> ; <c3>) <body>

<c1> → <Dec> | ϵ | <Asgn_st>

<c2> → <cond> | ϵ

<c3> → ϵ | <Asgn_st> | <inc_dec>

<Asgn_st> → ID <x> AOP <EXP> ;

* a = 5	a = 5
* a[1] = 5	a = b
a[i] = 5	a = b * c
a[i].c = 5	a.b = 5
c.a[i] = 5	
c.b.x = 5	
c.b[i].x = 5	

<inc_dec> → ID <x> INC_DEC; |

INC_DEC ID <x>

These <asgn_st> and <inc_dec_st> would accept ; inside for which isn't correct.

For that:

① <Asgn_st> → ID <x> AOP <EXP>

<Ast> → <Asgn_st>;

② <inc_dec_st> → ID <x> INC_DEC

INC_DEC ID <x>

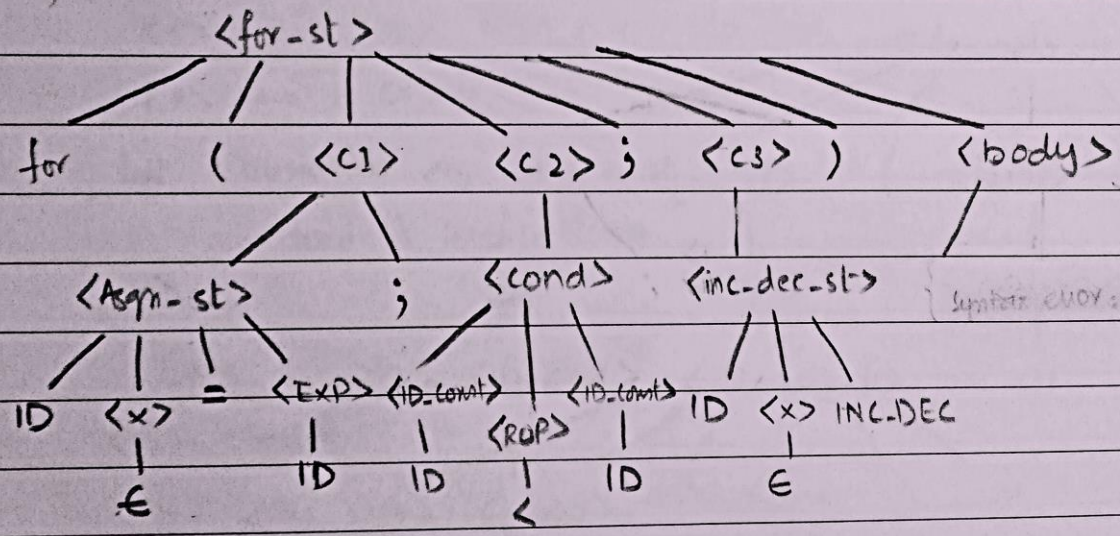
<in_st> → <inc_dec_st>;

There would be another ; added in Dec, so:

<for_st> → for (<c1> <c2> ; <c3>)

<c1> → <Dec> | ; | <Asgn_st> ;

```
for (a=b; a<b; a++)
{ while (true)
    {
    }
}
```



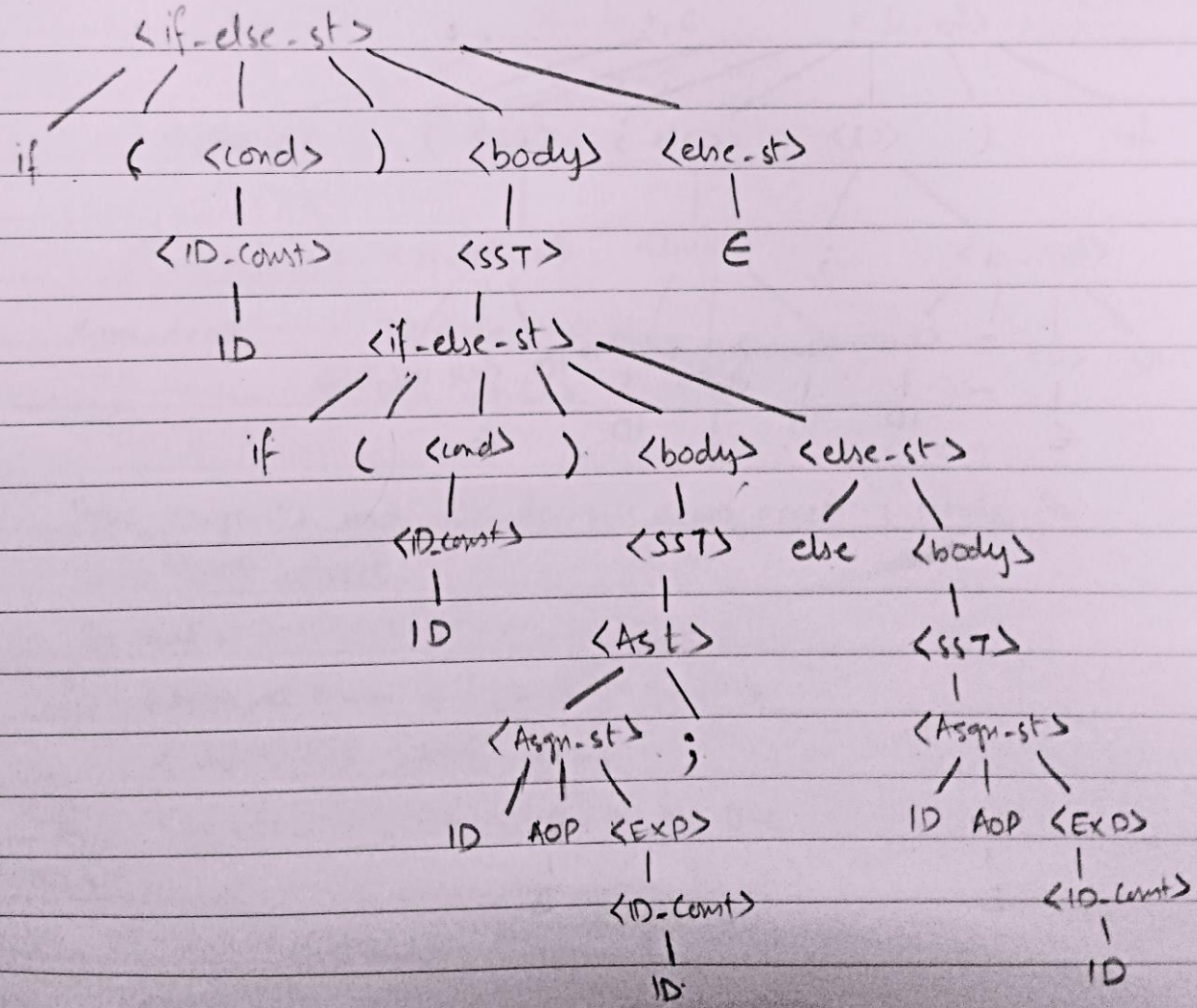
<if-else-st> → if (<cond>) <body> <else-st>

<else-st> → else <body> | ϵ

```
if (a)
if (b)
  a = b;
else
   b = a
```

dangling else

Ambigous grammar



PEB3/2.

Disambiguity Rule : else is always attached with last Open IF.

Characteristics of grammar:

1. Unambigous
2. Non-left Recursive
3. left factored

$\langle x \rangle \rightarrow \langle x \rangle a$: left recursive

$\langle x \rangle \rightarrow a \langle x \rangle$: right recursive

$\langle x \rangle \rightarrow a \langle x \rangle b$

left recursion is non-deterministic and it can lead to infinite loop

ba^*

$\langle x \rangle \rightarrow b \langle x1 \rangle$

$\langle x1 \rangle \rightarrow a \langle x1 \rangle \mid \epsilon$

for every left recursive grammar there exists a right recursive grammar.

$\langle A \rangle \rightarrow \langle A \rangle \alpha$

$\langle A \rangle \rightarrow \beta$

$\Downarrow$

$\langle A \rangle \rightarrow \beta (A')$

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

$\langle B \rangle \rightarrow \langle B \rangle \; a \langle x \rangle c \; \langle B \rangle d$

$\langle B \rangle \rightarrow y \langle c \rangle x$

$\Downarrow$

$\langle B \rangle \rightarrow y \langle c \rangle x \; \langle B' \rangle$

$\langle B' \rangle \rightarrow a \langle x \rangle c \langle B \rangle d \; \langle B' \rangle \mid \epsilon$

$\langle S \rangle \rightarrow a b \langle x \rangle c$

$\langle x \rangle \rightarrow \langle x \rangle \langle x \rangle bc$

$\langle x \rangle \rightarrow y c \langle B \rangle$

$\Downarrow$

$\langle S \rangle \rightarrow ab \langle x \rangle c$

$\langle x \rangle \rightarrow yc \langle B \rangle \; \langle x' \rangle$

$\langle x' \rangle \rightarrow \langle x \rangle bc \langle x' \rangle \mid \epsilon$

e.g.

$\langle x \rangle \longrightarrow \langle x \rangle \underbrace{\langle x \rangle \langle c \rangle d}_{\alpha}$

$\langle x \rangle \longrightarrow \underbrace{b \langle c \rangle y}_{\beta}$

$\Downarrow$

$\langle x \rangle \longrightarrow b \langle c \rangle y \; \langle x' \rangle$

$\langle x' \rangle \longrightarrow \langle x \rangle \langle c \rangle d \langle x' \rangle \mid \epsilon$

01- October - 2019

$\langle X \rangle \rightarrow \langle X \rangle \underbrace{a}_{\alpha}$

$\langle X \rangle \rightarrow \underbrace{b}_{\beta_1}$

$\langle X \rangle \rightarrow \underbrace{c}_{\beta_2}$

The above CFG can not be converted by the previous rule

So the new rule is:

$\langle A \rangle \rightarrow \beta_1 \langle A' \rangle \mid \beta_2 \langle A' \rangle$

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

If there are multiple β we would have to add multiple OR in $\langle A \rangle$

$\langle X \rangle \rightarrow b \langle X' \rangle \mid c \langle X' \rangle$

$\langle X' \rangle \rightarrow a \langle X' \rangle \mid \epsilon$

e.g: $\langle Y \rangle \rightarrow \langle Y \rangle bc$

$\langle Y \rangle \rightarrow x \langle Y \rangle b$

$\langle Y \rangle \rightarrow b \langle c \rangle y$

$\Downarrow$

$\langle Y \rangle \rightarrow x \langle Y \rangle b \langle Y' \rangle \mid b \langle c \rangle y \langle Y' \rangle$

$\langle Y' \rangle \rightarrow bc \langle Y' \rangle \mid \epsilon$

e.g: $\langle X \rangle \rightarrow \langle X \rangle \underbrace{a}_{\alpha_1}$

$\langle X \rangle \rightarrow \langle X \rangle \underbrace{b}_{\alpha_2}$

$\langle X \rangle \rightarrow \underbrace{c}_{\beta_1}$

$\langle X \rangle \rightarrow \underbrace{d}_{\beta_2}$

$\Downarrow$

$\langle X \rangle \rightarrow c \langle X' \rangle \mid d \langle X' \rangle$

$\langle X' \rangle \rightarrow a \langle X' \rangle \mid b \langle X' \rangle \mid \epsilon$

e.g:

$\langle S \rangle \rightarrow ab\langle X \rangle y$

$\langle X \rangle \rightarrow \langle X \rangle \underbrace{bc\langle Y \rangle}_{\alpha}$

$\langle X \rangle \rightarrow \underbrace{ab\langle X \rangle c}_{\beta_1}$

$\langle X \rangle \rightarrow \underbrace{y}_{\beta_2}$

$\langle Y \rangle \rightarrow \langle Y \rangle \underbrace{bx}_{\alpha_1}$

$\langle Y \rangle \rightarrow \langle Y \rangle \underbrace{\langle c \rangle bd}_{\alpha_2}$

$\langle Y \rangle \rightarrow \underbrace{c}_{\beta_1}$

$\langle Y \rangle \rightarrow \underbrace{y\langle Y \rangle b}_{\beta_2}$

$\Downarrow$

$\langle S \rangle \rightarrow ab\langle X \rangle y$

$\langle X \rangle \rightarrow y\langle X' \rangle \mid ab\langle X \rangle c\langle X' \rangle$

$\langle X' \rangle \rightarrow bc\langle Y \rangle\langle X' \rangle \mid \epsilon$

$\langle Y \rangle \rightarrow c\langle Y' \rangle \mid y\langle Y \rangle b\langle Y' \rangle$

$\langle Y' \rangle \rightarrow bx\langle Y' \rangle \mid \langle c \rangle bd\langle Y' \rangle \mid \epsilon$

$\langle A \rangle \rightarrow x\langle y \rangle$

$\langle A \rangle \rightarrow \langle B \rangle x\langle C \rangle$

$\langle B \rangle \rightarrow \langle A \rangle y\langle C \rangle$

$\langle B \rangle \rightarrow b$

This CFG is left recursive but in an indirect way

To remove left recursion we make it direct through substitution:

For now we are substituting <B> in <A>

<A> → <A> y <C> x <C>

<A> → b x <C> (substitute all values)

<A> → x <y> <B> (write right recursive rule as it is)

<B> → <A> y <C> (other rule as it is)

<B> → b

e.g: <x> → <B> <c> d

<x> → <c> d

<B> → <x> y

<B> → b

<C> → c <D>

<C> → y

<x> → b <c> d

<x> → <x> y <c> d

<x> → <c> d

<B> → <x> y

<B> → b

<C> → c <D>

<C> → y

⇓

<x> → <c> d <x'> | bcd <x'>

<x'> → y <c> d <x'> | ϵ

check?

e.g: $\langle X \rangle \rightarrow \langle B \rangle \langle C \rangle d$

$\langle X \rangle \rightarrow \langle C \rangle d$

$\langle B \rangle \rightarrow \langle X \rangle y$

$\langle B \rangle \rightarrow b$

$\langle C \rangle \rightarrow \langle X \rangle b \langle C \rangle$

$\langle C \rangle \rightarrow b \langle C \rangle \langle X \rangle$

$\langle X \rangle \rightarrow \langle X \rangle y \langle C \rangle d$

$\langle X \rangle \rightarrow \langle X \rangle b \langle C \rangle d$

$\langle X \rangle \rightarrow b \langle C \rangle d$

$\langle X \rangle \rightarrow b \langle C \rangle \langle X \rangle d$

$\langle C \rangle \rightarrow \langle X \rangle b \langle C \rangle$

$\langle C \rangle \rightarrow b \langle C \rangle \langle X \rangle$

$\langle B \rangle \rightarrow \langle X \rangle y$

$\langle B \rangle \rightarrow b$

$\langle X \rangle \rightarrow b \langle C \rangle d \langle X' \rangle \mid b \langle C \rangle \langle X \rangle d \langle X' \rangle$

$\langle X' \rangle \rightarrow y \langle C \rangle d \langle X' \rangle \mid b \langle C \rangle d \langle X' \rangle \mid \epsilon$

$\langle X \rangle \rightarrow \langle Y \rangle \langle X \rangle c \quad \Rightarrow \quad \langle X \rangle \rightarrow \langle B \rangle c \langle Y \rangle c$

$\langle X \rangle \rightarrow d \qquad \langle X \rangle \rightarrow \langle Y \rangle c$

$\langle Y \rangle \rightarrow \langle B \rangle c \langle Y \rangle \qquad \langle X \rangle \rightarrow d$

$\langle Y \rangle \rightarrow \epsilon$

PER 3/2

e.g: $\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle d$

$\langle B \rangle \rightarrow x \langle B \rangle c$

$\langle B \rangle \rightarrow \epsilon$

$\langle C \rangle \rightarrow \langle A \rangle c$

$\langle C \rangle \rightarrow x$

$\langle A \rangle \rightarrow x \langle B \rangle c \langle A \rangle c d$

$\langle A \rangle \rightarrow x \langle B \rangle c x d$

$\langle A \rangle \rightarrow \langle A \rangle c d$

$\langle A \rangle \rightarrow x d$

$\langle B \rangle \rightarrow x \langle B \rangle c$

$\langle B \rangle \rightarrow \epsilon$

$\langle C \rangle \rightarrow \langle A \rangle c$

$\langle C \rangle \rightarrow x$

$\langle A \rangle \rightarrow x \langle B \rangle c \langle A \rangle c d \langle A' \rangle \mid x \langle B \rangle c x d \langle A' \rangle \mid x d \langle A' \rangle$

$\langle A' \rangle \rightarrow c d \langle A' \rangle \mid \epsilon$

LEFT - FACTORED

<DEC> → DT ID <INIT> <LIST>

$\boxed{\text{<INIT> → =ID <INIT> | = <CONST> | } \epsilon}$

<INIT> → = <X> | ϵ

<X> → ID <INIT> | <CONST>

If there is any non-terminal with more than one rule having something similar on the left hand side then we break it so that the grammar does not have options while traversing (having options would require backtracking).

e.g: <S> → ab<X>b

<X> → ab<C>d

<X> → ab <X>d

<X> → y

<X> → ab <X1>

<X1> → <C>d | <X>d

<X> → y

e.g: <X> → a <X>b <C> ⇒ <X> → a <X1>

<X> → a <X> c <Y> <X1> → <X> <X2> | <C> d

<X> → a <C> d <X2> → b<C> | c<Y>

Indirect To direct

e.g: $\langle X \rangle \rightarrow \langle B \rangle c \langle X \rangle d$

$\langle X \rangle \rightarrow a \langle c \rangle d$

$\langle B \rangle \rightarrow a \langle Y \rangle b$

$\langle B \rangle \rightarrow b$

$\langle X \rangle \rightarrow b c \langle X \rangle d$

$\langle X \rangle \rightarrow a \langle Y \rangle b c \langle X \rangle d$

$\langle X \rangle \rightarrow a \langle c \rangle d$

$\langle B \rangle \rightarrow a \langle Y \rangle b$

$\langle B \rangle \rightarrow b$

left factoring

$\langle X \rangle \rightarrow a \langle X1 \rangle \mid b c \langle X \rangle d$

$\langle X1 \rangle \rightarrow \langle Y \rangle b c \langle X \rangle d \mid \langle c \rangle d$

$\langle B \rangle \rightarrow a \langle Y \rangle b$

$\langle B \rangle \rightarrow b$

Do again

e.g: $\langle X \rangle \rightarrow \langle B \rangle x \langle Y \rangle$

$\langle X \rangle \rightarrow x \langle A \rangle \langle C \rangle$

$\langle B \rangle \rightarrow \langle A \rangle c \langle X \rangle$

$\langle B \rangle \rightarrow \epsilon$

$\langle A \rangle \rightarrow y \langle A \rangle b$

$\langle A \rangle \rightarrow c$

$\langle B \rangle \rightarrow y \langle B \rangle$

$\langle X \rangle \rightarrow y \langle B \rangle x \langle Y \rangle$

$\langle X \rangle \rightarrow \langle A \rangle c \langle X \rangle x \langle Y \rangle$

$\langle X \rangle \rightarrow x \langle Y \rangle$

$\langle X \rangle \rightarrow x \langle A \rangle \langle C \rangle$

$\langle B \rangle \rightarrow \langle A \rangle c \langle X \rangle$

$\langle B \rangle \rightarrow \epsilon$

$\langle A \rangle \rightarrow y \langle A \rangle b$

$\langle A \rangle \rightarrow c$

$\downarrow$

$\langle X \rangle \rightarrow y \langle B \rangle x \langle Y \rangle$

$\langle X \rangle \rightarrow y \langle A \rangle b c \langle X \rangle x \langle Y \rangle$

$\langle X \rangle \rightarrow c c \langle X \rangle x \langle Y \rangle$

$\langle X \rangle \rightarrow x \langle Y \rangle$

$\langle X \rangle \rightarrow x \langle A \rangle \langle C \rangle$

<SST> → <DEC> | <if_else> | <while_st> | <for_st> |
<switch_st> | <do_while> | <fn_call> | <obj_dec> |
<Assgn_st> | <inc_dec_st> | ....

<fn_call> = a[i].fn()
a.fn()
fn()

a.fn()
a++
a = 10
a b= new a()

Show CFG Next Week

<SST> → <DEC> | <if_else> | <while_st> | <for_st> |
<switch_st> | <do_while> | <fn_call> | <obj_dec> |
<Assgn_st> | <inc_dec_st> | ....

<fn_call> = a[i].fn()
a.fn()
fn()

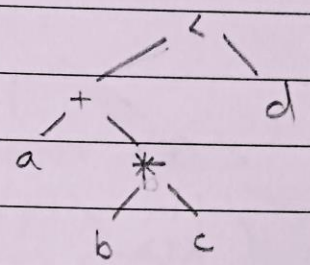
a.	fn()
a	++
a	= 10
a	b = new a()

Show CFG Next Week

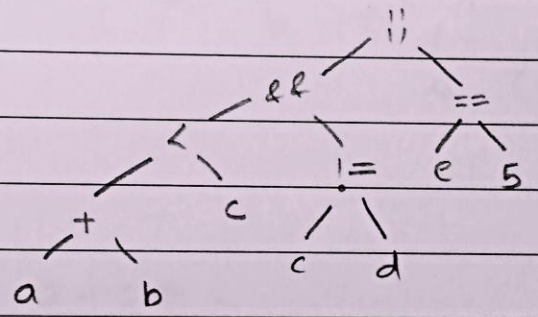
8 - October - 2019

EXPRESSION :

expression tree : a + b * c < d



a + b < c && c != d || e == 5



- when precedence same, tree grows towards left
- leaf nodes are the operands / Terminals.
- To go to a lower degree in a tree we need to have a non-terminal at the same level.

a + b * c

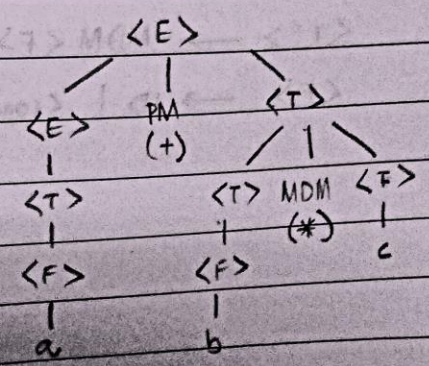
<E> → <E> PM <T>

<E> → <T>

<T> → <T> MDM <F>

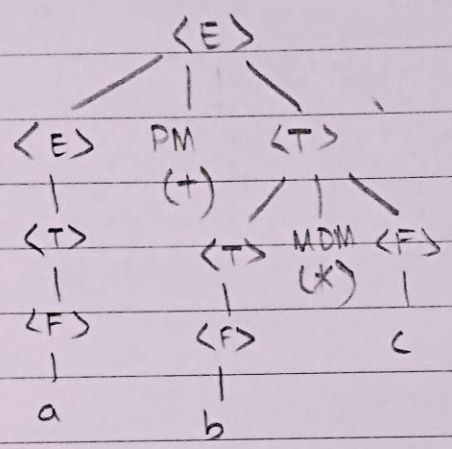
<T> → <F>

<F> → ID | <const> | <fn_call>

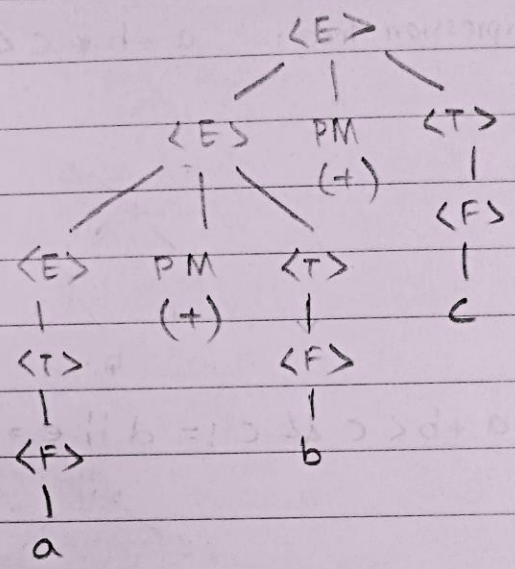


RE 3/12.

a + b * c



a + b + c



$\langle E \rangle \rightarrow \langle E \rangle \underbrace{PM \langle T \rangle}_{\alpha}$

$\langle E \rangle \rightarrow \underbrace{\langle T \rangle}_{\beta}$

$\langle T \rangle \rightarrow \langle T \rangle \underbrace{MDM \langle F \rangle}_{\alpha}$

$\langle T \rangle \rightarrow \underbrace{\langle F \rangle}_{\beta}$

$\langle E \rangle \rightarrow \langle T \rangle \langle E' \rangle$

$\langle E' \rangle \rightarrow PM \langle T \rangle \langle E' \rangle \mid \epsilon$

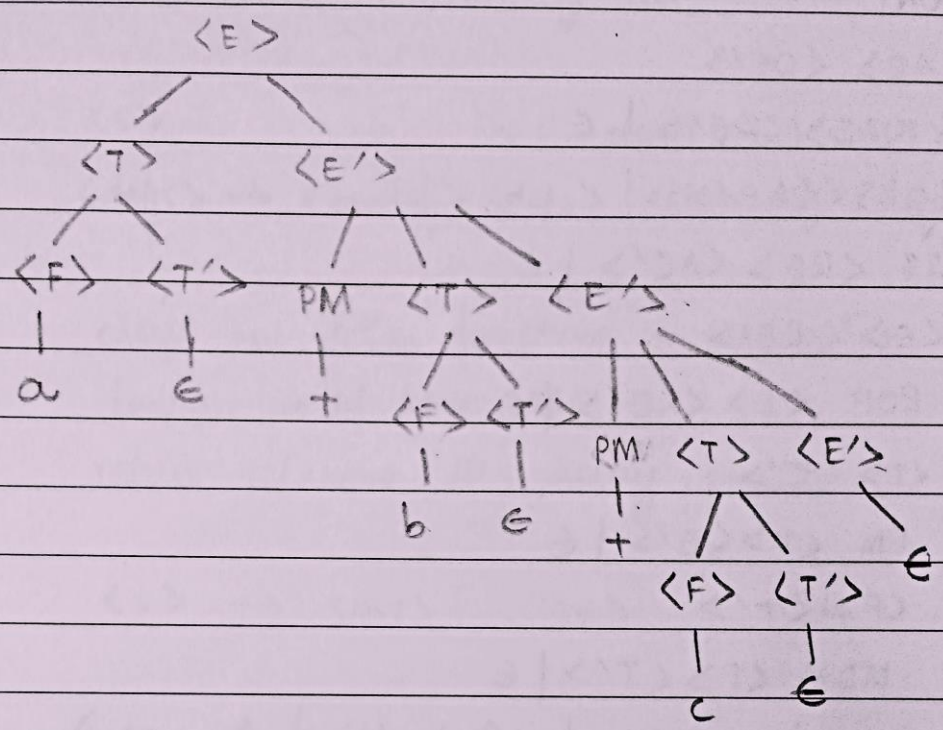
$\langle T \rangle \rightarrow \langle F \rangle \langle T' \rangle$

$\langle T' \rangle \rightarrow MDM \langle F \rangle \langle T' \rangle \mid \epsilon$

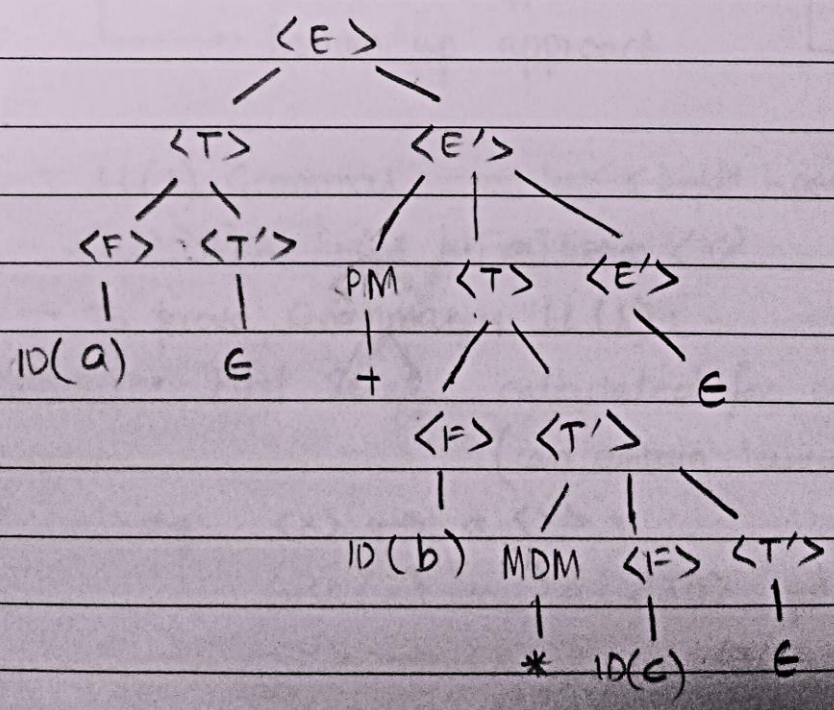
$\langle F \rangle \rightarrow ID \mid \langle const \rangle \mid \langle fn_call \rangle$

MDM
PM
ROP
&&
||

a + b + c


<E>
<T>
<E'>
<F>
<T'>
PM
<T>
<E'>
a
ϵ
+
<F>
<T'>
PM
<T>
<E'>
b
ϵ
+
ϵ
<F>
<T'>
c
ϵ

a + b * c


<E>
<T>
<E'>
<F>
<T'>
PM
<T>
<E'>
ID(a)
ϵ
+
ϵ
<F>
<T'>
ID(b)
MDM
<F>
<T'>
*
ID(c)
ϵ

CFG for EXPRESSION:

<OE> → <AE> <OE'>

<OE'> → || <AE> <OE'> | ∈

<AE> → <RE> <AE'>

<AE'> → && <RE> <AE'> | ∈

<RE> → <E> <RE'>

<RE'> → ROP <E> <RE'> | ∈

<E> → <T> <E'>

<E'> → PM <T> <E'> | ∈

<T> → <F> <T'>

<T'> → MDM <F> <T'> | ∈

<F> → ID | <const> | <fn_call> | (<OE>)
| ! <F> | <INC_DEC>

!a [5] → ID <Y>

a [5] * a [5] <Y> → [Ar] | <.> | .. <func_call>

a.b

a.fn()

a.fn().c

<cond> replaced by <OE>

Assignment mein <OE>

Parameter list <PL> → <OE>, <OE> ... etc

ar [<OE>]

There should be one starting non-terminal <S>

<S> void main () { <MST> }

<defs> → <fn-def> <defs> | <class-def> <defs> | .... | ϵ

class has either functions or attributes not direct MST

functions would have MST

return returns ID, array, pointer

<S> → <MST> | <class> ... etc

- Syntax Analyzer / Parse Tree
  - → Top down approach → back-Tracking
  - → Bottom up approach → Predictive — LL(K)

- LL(1) Grammar — Left to right input scanning, Left
  - → one input at a time
- To prove Grammar is LL(1):
  - First Set : calculated for every non-terminal (can contain terminals, ϵ)

e.g: <X> → a <Y> z

first (x) = {a}

<Y> → <X> bc

1- first (Y) = {<X>} → {a} = first (x) →

$\langle Y \rangle \rightarrow x \langle Z \rangle c$

2- $first(Y) = \{x\}$

$\langle Z \rangle \rightarrow \langle Y \rangle \langle X \rangle x$

$first(Z) = first(Y)$

$= \{a, x\}$

$first(Y) = \{a\} \cup \{x\}$

$= \{a, x\}$

$\langle X \rangle \rightarrow a \langle X \rangle c$

$\langle X \rangle \rightarrow \epsilon$

1- $first(X) = \{a\}$

2- $first(X) = \{\epsilon\}$

First set can never be empty, it can have ϵ.

$first(X) = \{a, \epsilon\}$

$\langle X \rangle \rightarrow \langle Y \rangle z$

$\langle Y \rangle \rightarrow a \langle B \rangle x$

$\langle Y \rangle \rightarrow \epsilon$

$first(X) = first(Y)$

$= \{a, \epsilon\} - \{\epsilon\} \cup \{z\}$

$= \{a, z\}$

$\langle X \rangle \rightarrow \langle Y \rangle \langle Z \rangle$

$\langle Y \rangle \rightarrow a \langle B \rangle x$

$\langle Y \rangle \rightarrow \epsilon$

$\langle Z \rangle \rightarrow c \langle Z \rangle x$

$\langle Z \rangle \rightarrow b \langle Y \rangle$

$\langle Z \rangle \rightarrow \epsilon$

$first(Y) = \{a, \epsilon\}$

$first(Z) = \{c, b, \epsilon\}$

$first(X) = first(Y)$

$= \{a, \epsilon\}$

$= \{a, \epsilon\} - \{\epsilon\} \cup first(Z)$

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

eg:

$\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle \langle D \rangle$

$\langle B \rangle \rightarrow b \langle B \rangle x$

$\langle B \rangle \rightarrow \epsilon$

$\langle C \rangle \rightarrow \underline{\langle A \rangle} x$

$\langle C \rangle \rightarrow \epsilon$

$\langle A \rangle \rightarrow a \langle Y \rangle$

$\langle A \rangle \rightarrow \epsilon$

$\langle D \rangle \rightarrow d \langle B \rangle c$

$\langle D \rangle \rightarrow \epsilon$

$first(D) = \{d, \epsilon\}$

$first(B) = \{b, \epsilon\}$

$\underline{first(A)} = \{a, \epsilon\} \cup first(B)$

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

$= \{a, b, \epsilon\} - \{\epsilon\} \cup \{\underline{first(C)}\}$

left recursion occurs!

$\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle \langle D \rangle$

$\langle B \rangle \rightarrow b \langle B \rangle x$

$\langle B \rangle \rightarrow \epsilon$

$\langle C \rangle \rightarrow \langle X \rangle \underline{x}$

$\langle X \rangle \rightarrow a \langle Y \rangle$

$\langle X \rangle \rightarrow \epsilon$

$\langle D \rangle \rightarrow d \langle B \rangle c$

$\langle D \rangle \rightarrow \epsilon$

$first(B) = \{b, \epsilon\}$

$first(X) = \{a, \epsilon\}$

$first(D) = \{d, \epsilon\}$

$first(C) = first(X)$

$= \{a, \epsilon\} - \{\epsilon\} \cup \{x\}$

$= \{a, x\} \cup \{\epsilon\}$

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

$first(A) = first(B)$

$= \{b, \epsilon\} - \{\epsilon\} \cup first(C)$

$= \{b, a, x, \epsilon\} - \{\epsilon\} \cup first(D)$

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

15- October- 2019

FOLLOW SET:

for each NT and can't have ϵ, can contain terminals, end marker. - Tree should be complete

- Input completely parsed.

$\langle X \rangle \rightarrow a \langle Y \rangle b$ follow $(Y) = \{b\}$

$\langle X \rangle \rightarrow a \langle B \rangle \langle C \rangle x$ follow $(C) = \{x\}$

follow(B) = first(C)

follow (S) = {End Marker} \$

- In Lexical Analyzer add one token at the end with class part = '\$' that is the end marker.
- For follow set, first check the follow set of the starting non-terminal which should be an end marker. (when $\langle S \rangle$ is not being called anywhere else)

e.g: $\langle S \rangle \rightarrow a \langle X \rangle c \langle B \rangle d$

$\langle X \rangle \rightarrow b \langle X \rangle \underline{\langle Y \rangle c}$

$\langle B \rangle \rightarrow x \langle A \rangle y$

$\langle Y \rangle \rightarrow y \langle X \rangle c \mid z \langle B \rangle a$

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

follow $(B) = \{a\} \cup \{d\}$

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

follow $(X) = \{c\} \cup \{c\} \cup$ follow$(Y) = \{c\} \cup \{y, z\}$

$= \{c, y, z\}$

e.g: $\langle S \rangle \rightarrow \langle X \rangle \langle Y \rangle \langle Z \rangle c$

$\langle Y \rangle \rightarrow a \langle Y \rangle b \mid b \langle C \rangle c$

$\langle Z \rangle \rightarrow z \langle A \rangle x \mid \epsilon$

follow (S) = { \$ }

follow (A) = { x }

follow (C) = { c }

follow (Z) = { c }

follow (Y) = { b } ∪ first (Z)

= { b } ∪ { z, ε }

= ~~{ b, z, ε }~~ − { ε } ∪ { c }

= { b, z, c }

follow (X) = first (Y)

= { a, b }

e.g: $\langle S \rangle \rightarrow \langle X \rangle \langle Y \rangle \langle Z \rangle c$

$\langle Y \rangle \rightarrow a \langle Y \rangle b \mid \epsilon$

$\langle Z \rangle \rightarrow z \langle A \rangle x \mid \epsilon$

follow (S) = { \$ }

follow (X) = first (Y)

= { a, ε } − { ε } ∪ first (Z)

= { a } ∪ { z, ε }

= { a, z, ε } − { ε } ∪ { c }

= { a, z, c }

follow (Y) = { b, z, c }

follow (A) = { x }

follow (Z) = { c }

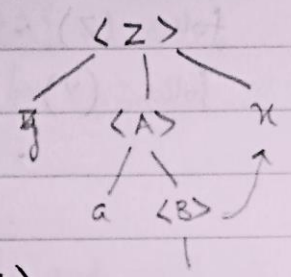
e.g. $\langle Y \rangle \rightarrow a \langle Y \rangle b \mid x$

$\langle Z \rangle \rightarrow z \langle A \rangle x \mid b$

$\langle A \rangle \rightarrow a \langle B \rangle$

for follow (B) there are no terminals or non-terminals after $\langle B \rangle$ so apparently nothing follows it. But, when $\langle B \rangle$ is called and ends, $\langle A \rangle$ also ends so the ~~non~~ terminal that follows $\langle A \rangle$ would also become the follows for $\langle B \rangle$.

This could also be understood by substituting the definition of $\langle A \rangle$ in $\langle Z \rangle \mapsto z \; a \langle B \rangle \; x$



$\therefore$ in <u>this</u> case follow (B) = follow (A)

- every non-terminal has a follow, if not, it is useless.

e.g. $\langle S \rangle \rightarrow \langle X \rangle \langle Y \rangle \langle Z \rangle$.

$\langle X \rangle \rightarrow a \langle B \rangle x \mid \epsilon$

$\langle Y \rangle \rightarrow y \langle C \rangle b \mid \epsilon$

$\langle Z \rangle \rightarrow z \langle Z \rangle a \mid \epsilon$

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

follow (B) = $\{x\}$

follow (C) = $\{b\}$

follow (Z) = $\{a\} \cup$ follow (S)

$= \{a, \$\}$

follow (Y) = first (Z) $= \{z, \epsilon\} - \{\epsilon\} \cup$ follow (S) $= \{z, \$\}$

follow (X) = first (Y) $= \{y, \epsilon\} - \{\epsilon\} \cup$ first (Z)

$= \{y, z, \epsilon\} - \{\epsilon\} \cup$ follow (S) $= \{y, z, \$\}$

e.g: $\langle A \rangle \rightarrow b \langle x \rangle e$

$\langle x \rangle \rightarrow B \langle Y \rangle \langle x \rangle$

$\langle x \rangle \rightarrow y \langle c \rangle x$

follow (x) = {e} ∪ follow (x)

when such situations happen we ignore the recursive call and take union of the remaining terminals.

If there is no other terminal then the grammar has fault and the follow set can not be empty.

e.g: $\langle S \rangle \rightarrow a \langle B \rangle \langle A \rangle y$

$\langle A \rangle \rightarrow a \langle B \rangle \langle C \rangle$

$\langle A \rangle \rightarrow b$

$\langle C \rangle \rightarrow x \langle A \rangle$

$\langle C \rangle \rightarrow z$

1- follow (A) = {y}

2- follow (A) = follow (C)

follow (C) = follow (A)

This an indirect recursive situation.

follow (A) = {y} ∪ follow (C)

follow (A) = {y} ∪

e.g: $\langle E \rangle \rightarrow \langle T \rangle \langle E' \rangle$

$\langle E' \rangle \rightarrow PM \langle T \rangle \langle E' \rangle \mid \epsilon$

$\langle T \rangle \rightarrow \langle F \rangle \langle T' \rangle$

$\langle T' \rangle \rightarrow MDM \langle F \rangle \langle T' \rangle \mid \epsilon$

$\langle F \rangle \rightarrow ID \mid \langle const \rangle \mid (\langle E \rangle)$

	first	follow
E	first (T) = first (F)	)
E'	PM , ϵ	follow (E) =)
T	first (F) = ID, const, (	first (E') = PM ,)
T'	MDM , ϵ	follow (T) = first (E') = PM,)
F	ID, const , (	first (T') = MDM , PM,)

e.g: $\langle S \rangle \rightarrow a \underline{\langle x \rangle} b \underline{\langle c \rangle} d$

$\langle c \rangle \rightarrow a \underline{\langle x \rangle} y \underline{\langle x \rangle}$

$\langle c \rangle \rightarrow c$

$\langle x \rangle \rightarrow b \underline{\langle c \rangle}$

$\langle x \rangle \rightarrow y$

follow $(x) = \{b\}$

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

follow (x) = follow (c)

follow (c) = follow (x)

follow $(c) = \{d\}$

follow $(x) = \{b, y\} \cup$ follow (c)

follow $(c) = \{d\} \cup$ follow (x)

Any of these follow can be substituted to remove recursion:

follow $(x) = \{b, y\} \cup \{d\} \cup$ follow (x)

$= \{b, y, d\}$

LL(1) Grammar — Proof

CONDITION # 01.

If a NT has more than 1 rules:

1. NT → .... first (NT)
2. NT → .... first (NT)

⋮ ⋮

n. NT → .... first (NT)

All the first sets should be disjoint. If it is not disjoint then the grammar would not be left factored.

CONDITION # 02

If first(NT) has ϵ then,

first (NT) $\cap$ follow (NT) = ϕ

e.g. <A> → <B> <X> c first (B) = {b, ϵ}

<B> → b <C> | ϵ follow(B) = first (X)

<X> → b <Y> | z = {b, z}

Both the first and follow sets are not disjoint

- Only find follow if the first set has ϵ

22-October-2019

SEMANTIC ANALYZER

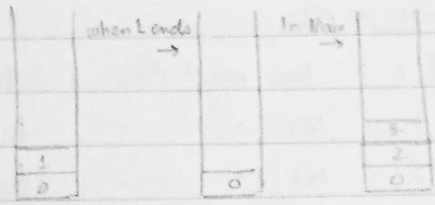
0 Global

```
float b = 9.5;
int a = 90;
void fn()
1 { int x;
    a += 2; }
void main()
2 { int a=7, b=9;
    while (a < b)
3   { char b = 'a';
      int a = 78;
      a++;
    }
    a += b*fn();
  }

class A
{ private : int a, x;
  public:
    void fn()
4   { - - - - - }
    int fn (int a, float b)
5   { char y;
      this.a = a
  }
}

Class C
{ void fn()
6 { A a = New A(); }
  A O2 = New B();
  O2.fn(); }
}
```

STACK



To get parent we can peak at the previous number in the stack

SYMBOL TABLE

1

Name	Type	Scope
b	float	0
a	int	0
fn	void→void	0
x	int	1
main	void→void	0
a	int	2
b	int	2
b	char	3
a	int	3
a	int	5
b	float	5
y	char	5
O2	A	6

2

Name	Type	Parent	Ref
A	class	—	
B	class	A	

3

Name	Type	A.M	T.M
a	int	private	–
x	int	private	–
fn	void→void	public	–
fn	int, float → int	public	–

```
Class B : A
{ public : void fn()
  {          }
}
```

PER3/2.

- redeclaration ~~is a~~ error is only within scope.
- Symbol Table is a table that keeps information of all
- for 'while (a < b)' the symantic analyzer will check if a is declared, if b is declared, if a and b are compareable.
- for 'a += b * fn()', the analyzer would check if a is declared, if b is declared, if function with that parameterlist is declared, if the multiply operation can be performed.
- for every declaration, it will also check if the variable/ID is already declared or not.
- Every scope can be identified by a unique number and its hierarchy is important.
- To maintain hierarchy use STACK. The scope follows a LIFO approach.

① Implement Symbol Table

② Use stack for scope.

- for 'A a = new A();', A is checked if a class of that name exists, and if A() and A can refer to each other, Are they compatible?
- Virtual and override keywords should be considered.
- SYMBOL TABLE:

  for variables, functions name, type and scope are important.

  - abstract, static written together?
  - abstract, virtual written together?, overide
  - Is there a by default parent class?
  - static, virtual written together?
  - procedure (doesn't return) function (returns 1 value)
- for functions reads Type, breaks on the arrow and saves both.

PES3/2.

- for all identifiers and declerations Table ① would be created
- for all class declerations Table ② would be created
- for each class a separate Table ③ would be created and its reference would be stored in Table ② Ref

22-October-2019

IMPLEMENTATION OF SYNTAX ANALYZER

CFG	FIRST SET	FOLLOW SET
<E> → <T><E'>	ID, this, Const, (, !	
<E'> → PM <T><E'> | ϵ	PM ϵ ∪)	)
<T> → <F><T'>	ID, this, Const, (, !	
<T'> → MDM <F><T'>	MDM	
<T'> → ϵ	ϵ ∪ PM,)	PM,)
<F> → <this> | ID <x>	this, ID	
→ <const>	Const	
→ (<E>)	(	
→ ! <F>	!	
<MST> → <SST> <MST>	for, while, if, switch, ID	
→ ϵ	ϵ ∪ }	}
<A> → <B><x>	a, b, x, ϵ ∪ follow(A)	

- When we remove ϵ from first set the resulting is a SELECTION SET.

Selection set is calculated for <u>every</u> rule

```
main ()
{   Token[] TS = LA (input)
    // requirement: return on the first syntax error found.
                = SA (TS)

SA ( )
{   i=0
    if (S()) // Tree is complete
    {  if (TS[i].CP == "$") // code is complete
                "valid Syntax" }
}
```

REB3/2.

- If there is any error :

  (" Invalid Syntax at line %d ", TS[i].lineNum);

- Implementation is rule by rule.
- every non-terminal would become a function.

  <T'> → MDM <F><T'>

  function declaration

  function calling

- every function would return true or false (tree is either complete or not)

```
bool E()
{ if (TS[i].CP == "ID" || "this" || "int_const" || "(" || "!" ) //check selection set
    { if (T())
        { if (E'())
            return True; }
    }
  else
    return False;
}

bool E'()
{ if (TS[i].CP == "PM")
    { i++;
      if (T())
      { if (E'())
          return True; }
    }
  else if (TS[i].CP == ")") // don't increment at ε
      return True;
  else
      return False;
}
```

PE8312.

```
bool F( )
{   if ( TS[i].CP == "this" || "ID" )
    {   if ( this( ) )
        {   if ( TS[i].CP == "ID" )
            {   if (x( ))
                    return True; }
        }
    }
    else if ( TS[i].CP == "const" ) // check all constants ie. int, string, float, char
    {   if ( const( ) )
            return True; }
    else if ( TS[i].CP == "(" )
    {   i++;
        if ( E( ) )
        {   if ( TS[i].CP == ")" )
                i++;
                return True; }
        }
    else if ( TS[i].CP == "!" )
    {   if ( F( ) )
            return True; }
    else
        return False;
```

SUBMISSION : 5th Nov 2019

PE83/2.

29- October - 2019

Symbol Tables:

1. ~~Class~~ Defs Table: N , T , Parent, Reference
2. ClassData Table: N , T , AM , TM
3. Scope Table: N , T , Scope

FUNCTIONS:

bool - Insert ~~CT~~ DT (N, T, Parent, & ref)

bool - Insert CDT (N, T, AM, TM), ref / CurrClassName)

bool - Insert ST (N, T, S)

Type - lookup ~~CT~~ DT (N)

Type - lookupCDT (N, T, ClassName/ref, & AM, & TM)

Curr Scope - Create Scope () { scope++; pushScopeOnStack () }

Curr Scope - Delete Scope () { pop(); }

Type - lookupST (N, scopestack, currClass)

Ret Type - lookupFn (N, parameteList, ClassName, bool Globalx, & AM, & TM)

- For every declaration checks if it is already declared or not
- Symantics mostly checks if the variables with operators are compatible with each other
- a.b.c = 5
  1. check if the word before . should not be a primitive type but an identifier.
  2. look up the variable in its class' data table.
- When looking up a variable its scope and hierarchy is important first searched in current scope, then in parent, parent, ..., lastly in global, if not in scope than current ClassData Table
- When looking up in CDT access modifier is important.

PER 3/2.

- fn() it can be global or in a class. First lookup through class name (as higher priority), if no class then lookup in global scope. Function's parameter list is important in lookup.
  Globalx is to tell if there is a . before fn calling i.e. should global scope be checked or not.
  If there is . it should be checked in class. If there is no class it should be checked in global only, if class and no . check both.

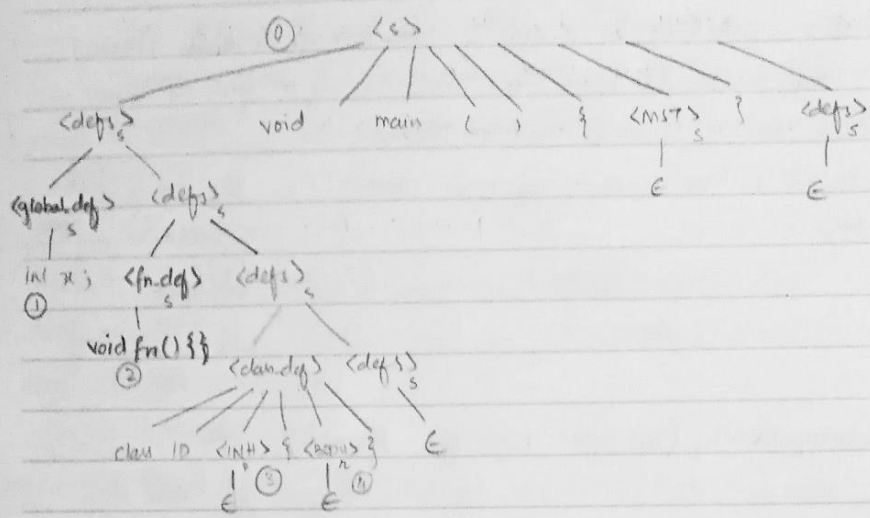
- Result Type Compatibility (left Type, Right Type, operator) RTC
  if null is returned, not compatible. Otherwise return resultant type
- Result Type Compatibility (Operand Type, Operator)

<S> → <defs> void main() { <MST> } <defs>
<defs> → <class_def> <defs> | ∈ | <fn_def> <defs> | <global_dec> <defs>
<class_def> → class ID <INH> { <body> }
<body> → AM : <decl> <body> | <ded> <body> | ∈
<decl> → <atti_dec> <decl> | <fn_dec> <decl> | ∈

- first step for semantic analyzer is to create the global scope.

```
int x;
void fn() { }
class A { }.
void main() { }
```

PERB12.



at 0 before <S> s = createScope // global scope created

at 1 insert ST (x , int , s)

* if it returns False : "redeclaration error" else valid

at 2 insertDT (fn , void → void , null , NULL) *

at 3 whenever in Syntax we get ID , save the value part in a variable first before incrementing the token set.

N = TS[i].valuePart

<INH> would tell if there is parent or not , for parent list return by reference

InsertDT (N , "class" , P , &r) *

at 4 class body would take r with it

at {} for braces that dont belong to class : { invokes create scope and } invokes deleteScope.

- Syntax Tree with semantic information is called ATTIBUTED TREE / ANNOTATED TREE

PE8 ∃ 12.

- Attributed Tree is for a specific input. uses CFG
- whenever there is declaration, there is insertion, whenever there is insertion, there is a redeclaration check
- having global variables is not a good approach.
- There are two kinds of attributes:
  - synthesize (bottom up approach) passed by reference
  - Inherited (Top down approach) passed by value

PES3/2.

5- NOVEMBER - 2019

$\langle class_def \rangle \rightarrow class\ \underset{N = TS[i].VP}{ID}\ \underset{p}{\langle INH \rangle}\ \{\ \underset{ref}{\langle class_body \rangle}\ \}$

$\langle class_body \rangle_{ref} \rightarrow \langle var_decl \rangle_{ref}\ \langle class_body \rangle_{ref} \mid \langle func_decl \rangle_{ref} \langle class_body \rangle_{ref} \mid \epsilon$

$\langle var_decl \rangle \rightarrow \langle AM \rangle \langle TM \rangle \langle DT \rangle\ ID\ \langle INIT \rangle\ \underset{rA,\ rT,\ rTM,\ ref}{\langle LIST \rangle}$

$\langle INIT \rangle \rightarrow AOP\ \langle OE \rangle \mid \epsilon$

$\langle LIST \rangle \rightarrow ,\ \underset{N}{ID}^{(i)} \langle INIT \rangle\ \langle LIST \rangle \mid ;$

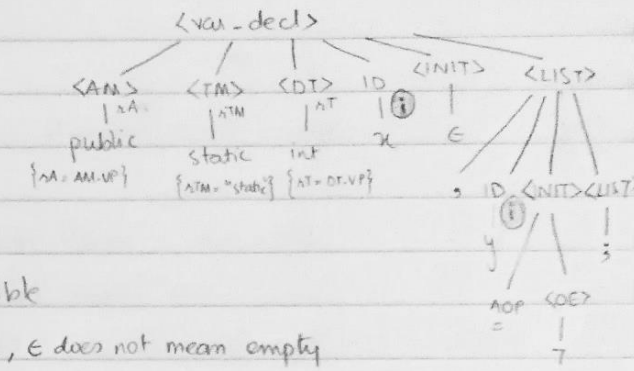
$\langle MST \rangle \rightarrow \langle SST \rangle \langle MST \rangle \mid \epsilon$

$\langle SST \rangle \rightarrow \langle DEC \rangle$

$\langle DEC \rangle \rightarrow \underset{rT}{\langle DT \rangle}\ \underset{N = ID.VP}{ID}^{(i)} \langle INIT \rangle \langle LIST \rangle$

r = by reference

public static int x, y = 7;



checks:

1. if x is not redeclared
2. if y is not redeclared
3. if y and 7 are compatible
4. set a default access modifier, ε does not mean empty

(i) insert CDT (N, rT, rA, rTM, ref)

- if insertion fails, redeclaration error.

(i) if(! insertCDT (N, rT, rA, rTM, ref)) { redeclaration error }

$\underset{rA}{\langle AM \rangle} \rightarrow AM\ \{rA = AM.VP\} \mid \epsilon\ \{rA = \text{"Private"}\}$

$\underset{rTM}{\langle TM \rangle} \rightarrow static\ \{rTM = \text{"static"}\} \mid \epsilon$

$ID\ \{N = ID.VP\}$

- Order of the parameters matter

PERSIA.

InsertFT (N, T, S)

N = { ID.VP } , $_\wedge$T = { DT.VP } , Scope

function parameter list <LIST1> → , ID { (i) if (!insertFT (N,T,S)) redeclaration } <INIT> <LIST1> | ;

~~<INH> → : DT~~ <DECL> → <DT> ID { (ii) } <INIT> <LIST>

class A : B { }

<INH>$_{\wedge P}$ → : ID$_{P = ID.VP}$ { (iii) } // single inheritance

(iii) { T = lookupDT (N)

if (T == null) { "undeclared" } else {

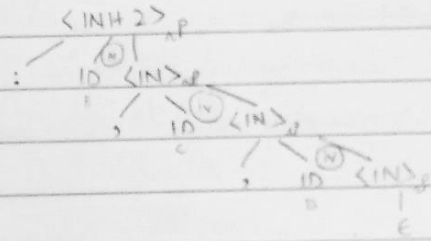
if (T != "class") { "wrong Type for inheritence" } }

$_\wedge$P = N }

<INH2>$_{\wedge P}$ → : ID <IN>$_{\wedge P}$ | ∈ // multiple inheritance

<IN>$_{\wedge P}$ → , ID <IN>$_{\wedge P}$ | ∈

class A : B, C, D



<INH2> → : ID (iii) <IN> | ∈

<IN> → , ID (iv) <IN> | ∈

(iv) { (iii)

$_\wedge$P += ',' + N }

Public Static A obj ; OR A obj ;

$_\wedge$T = A { T = lookupDT (N)

if (T == NULL) { undeclared } else {

if (T != "class") { wrong Type } }

- Whenever there is ID, lookup!

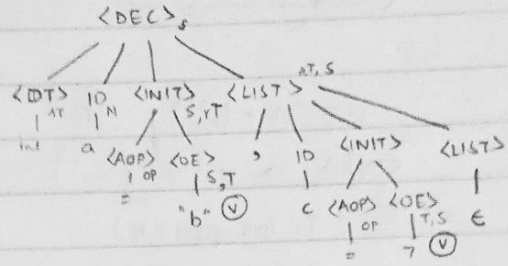
PES3/2.

<INIT> → AOP <OE> | ∈

e.g. int a = "abc";

int a = fn() * B
(int, float)

int a = "b", c = 7;



V { RT = RTC (rT, T, OP = {AOP.VP})
if (RT == NULL) { Type Mismatch } }

PERSIZ

19-November-2019

① <fn_def> → <RT> ID (<PL>) { <MST> }

<PL> → <T> ID <PL2> | ϵ | void

<PL2> → , <T> ID <PL2> | ϵ

<fn_call> → ID (<AL>)

<AL> → <OE> <AL2> | ϵ

<AL2> → , <OE> <AL2> | ϵ

insertDefT (N, T, Parent) → could be eliminated in case of functions.

- check whether a function is being declared or defined as both of them should be treated differently

```
int fn(int, float)          // prototype
int fn(int x, float y){}    // definition
int fn(int a, float b){}    // redeclaration of definition
```

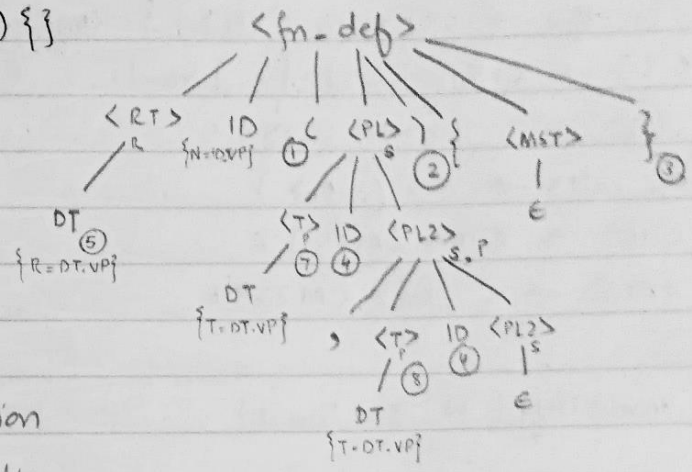
after definition, prototype can't be defined, it'd be redeclaration

insert_CDT (N, T, AM, TM, ref)

Type would be: int, float → int

PER3/2.

int fn (int x , float y) { }



① create scope for function

② insert function in table

③ destroy scope for function

④ insert variable in table. and check for redeclaration.

⑤ check the value part and save it to match with the return statement.
if <RT> gives ID then function returns an object, check if the class is declared.

if lookupDT (R) == NULL: {undeclared class}

⑥ Scope would be passed on to all functions inside function

⑦ Create a parameter list for a function.
{P = T}

⑧ Add parameters in the parameter list
{ P += ',' + T }

⑨ For Type to be saved in DefsT :
P + "→" + R

⑩ lastly: if (! insertDT (N, P + "→" + R))
"redeclaration error"

(i) $\langle fn\text{-}def \rangle \rightarrow \underset{R}{\langle RT \rangle}$ { if lookupDT(R) == NULL undeclared class } ID { N = TS[i].VP if !lookupDT(N) == NULL redeclaration }

({ S = createScope } $\underset{S,P}{\langle PL \rangle}$ { T = TS[i].VP if !lookupDT(T) == NULL redeclaration } $\underset{S}{\{ \langle MST \rangle \}}$)

{ if !insertDT(N, P+"→"+R) redeclaration error }

(ii) $\underset{S,P}{\langle PL \rangle} \rightarrow \langle T \rangle$ { T = TS[i].VP P = T } ID { N = TS[i].VP if !insertDT(N, T, S) redeclaration } $\underset{S,P}{\langle PL2 \rangle}$ | ϵ

(iii) $\underset{S,P}{\langle PL2 \rangle} \rightarrow ,\ \langle T \rangle$ { T = TS[i].VP P += ","+T } ID { N = TS[i].VP if !insertDT(N, T, S) redeclaration } $\underset{S,P}{\langle PL2 \rangle}$ | ϵ

lookupFunc(N, PL, CName, bool Global)

<SST> → ID <X> ;
<X> → (<AD>
<F> → ID <Y>
<Y> → (<AL>)

fn (a, b+c, fn());

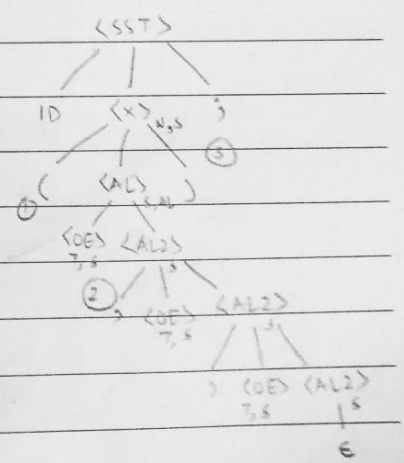
lookup (fn, T_1 + ",") + T_2 + ",") + T_3

for T_2 check type of b and type of c, then check if they are compatible and then find the final type, T_2.

① we get to know its a function

② OE returns type which would be concatenated with the Argument list

③ RT = lookupFunc (N, AL, __, __)
if (RT == NULL) { undeclared }



<OE> → <AE> <OE'>

<OE'> → || <AE> <OE'> | ∈

<AE> → <RE> <AE'>

<AE'> → && <RE> <AE'> | ∈

<RE> → <E> <RE'>

<RE'> → ROP <E> <RE'> | ∈

<E> → <T> <E'>

<E'> → PM <T> <E'> | ∈

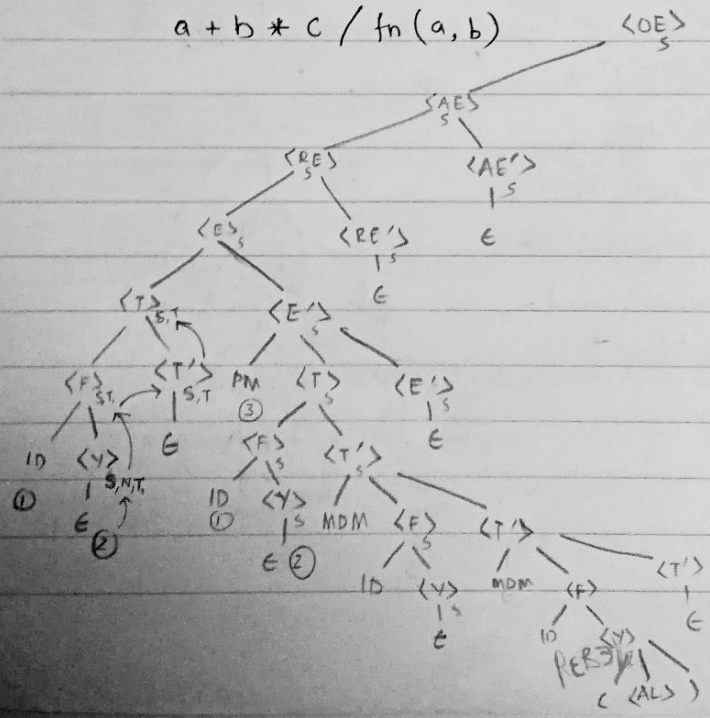
<T> → <F> <T'>

<T'> → MDM <F> <T'> | ∈

<F> → ID <Y> | (<OE>) | !<F> | <const>
| INC_DEC ID

<Y> → ∈ | (<AL>) | INC_DEC

a + b * c / fn(a, b)



S can be treated as global

① { N = ID.VP }

② { T = lookup (N, S)
if (T == NULL) redeclaration }

③ { O = PM.VP }

for compatibility: create a hardcoded table which has types and operator.

return	left	right	operator
int	int	int	+
float	int	float	*

$$a \overset{O_1}{+} b \overset{O_2}{*} c \overset{O_3}{/} \underbrace{fn(a,b)}_{T_5}$$

T_1 T_2 T_3 T_4

T_4 = Compatibility $(T_2, T_3, *)$. if $(T_4 == NULL)$ {Type Mismatch}

T_6 = Compatibility $(T_4, T_5, /)$ T_6 "

T_7 = Compatibility $(T_1, T_6, +)$ T_7 "

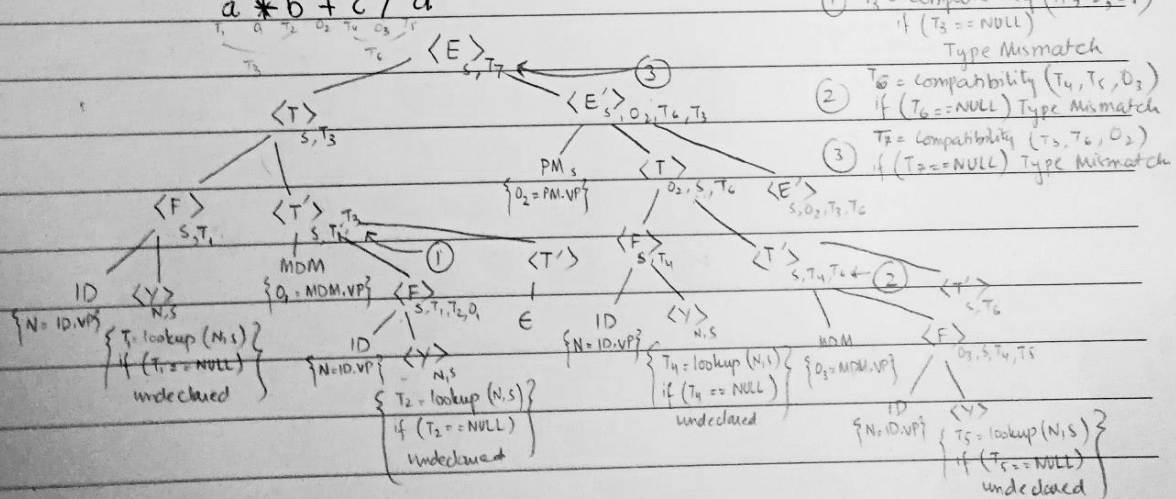
- first information is passed on to the sibling and then to the parent
- <F> builds the type of operand and passes the type to sibling
  if there is operator before <F>, compatibility check
- NULL means no change.

$a * b + c / d$

① T_3 = Compatibility (T_1, T_2, O_1)
if $(T_3 == NULL)$
Type Mismatch

② T_6 = Compatibility (T_4, T_5, O_3)
if $(T_6 == NULL)$ Type Mismatch

③ T_7 = Compatibility (T_3, T_6, O_2)
if $(T_7 == NULL)$ Type Mismatch



PE53/2.

$$\langle E \rangle_{T_2} \longrightarrow \langle T \rangle_{T_1} \quad \langle E' \rangle_{T_1, T_2}$$

$$\langle E' \rangle_{T_L, T} \longrightarrow PM \left\{ O = PM.VP \right\} \langle T \rangle_{T_R} \left\{ \begin{array}{l} T_A = Comp(T_L, T_R, O) \\ \text{if } (T_A == NULL) \text{ Mismatch} \end{array} \right\} \langle E' \rangle_{T_A, T}$$

$$| \ \epsilon \ \{ T = T_L \}$$

$$\langle T \rangle_{T_L} \longrightarrow \langle F \rangle_{T_1} \quad \langle T' \rangle_{T_1, T_2}$$

$$\langle T' \rangle_{T_L, T} \longrightarrow MDM \left\{ O = MDM.VP \right\} \langle F \rangle_{T_R} \left\{ \begin{array}{l} T_A = Comp(T_L, T_R, O) \\ \text{if } (T_A == NULL) \text{ Mismatch} \end{array} \right\} \langle T' \rangle_{T_A, T}$$

$$| \ \epsilon \ \{ T = T_L \}$$

Compatibility check

$$\langle F \rangle_{T} \longrightarrow ID_{N} \ \langle Y \rangle_{N,T} \ | \ (\langle OE \rangle) \ | \ ! \langle F \rangle_{T} \ (*) \ | \ \langle const \rangle_{T} \ | \ \text{Inc-Dec} \ ID_{N}$$

$$\langle Y \rangle_{N,T} \longrightarrow \epsilon \ | \ (\langle AL \rangle) \ | \ \text{Inc_Dec}$$

Array and pointers not included.

22- November - 2019

INTERMEDIATE CODE GENERATOR — Three Address Code

```
class A
{   int x;
    float fn( int a, float b)
    { int y = a+b*c;
      while (a <= 5)
      { a++; }
      b += 7        }
      int fn ( int x)
      { - - - - -  }
}
```

- low level languages are architecture/ machine dependent
- Intermediate code can vary
- Three Address code is not the only type
- In three address code every instruction can have atmost 3 addresses.
- A code can either have directives or statements
- Compiler stores the size of variables in its symbol table and also stores relative addresses.
- Declarations are not instructions, for declarations symbol table is populated but there is no code generated for it.
- When there is no constructor written, compiler adds its own code for null constructor.
- When a function is called, parameters and index is pushed on stack.
- Code generation usually starts from functions.

The code for class A:

fn_int_float fn Proc

$t_1 = b * c$

$t_2 = a + t_1$

$y = t_2$

fn EndP

$a = b * c / d \;\&\&\; c != e + 55 \;||\; c <= 5$

$t_1 = b * c$

$t_2 = t_1 / d$

$t_3 = e + 55$

$t_4 = c != t_3$

$t_5 = t_2 \;\&\&\; t_4$

$t_6 = c <= 5$

$t_7 = t_5 \;||\; t_6$

$a = t_7$

- The names for functions should be unique and because overloading is possible, we add the parameter type with the name, e.g: fn_int
  It is also possible for two classes to have function with the same name, so we can add: A_fn_int, (the class name) for global functions we can have g_fn_int.
  Return type does not make a function unique so we don't add it to the name of the function.
- All loops and conditionals can be coded through conditional and unconditional jumps.

Code for while loop in A_fn_int_float:

```
L1: t3 = a <= 5
    if (t3 == false) jmp L2
        a = a + 1
        jmp L1
L2: b = b + 7
```

```
a = b * c
if (b != d)
{ a++ }
else
{ b*=d }
b--
```

Intermediate code:

```
t1 = b*c
a = t1
t2 = b != d
if (t2 == false) jmp L1
    a = a+1
    jmp L2
L1:  b = b*d
L2:  b = b-1
```

```
if (a==b)
{ a++
  while (b<c)
  { a+=5 }
  b--
}
b+=2
```

```
t1 = a==b
if (t1 == false) jmp L1
    a = a+1
L3: t2 = b<c
    if (t2 == false) jmp L3
        a = a+5
        jmp L2
L3: b = b-1
L1: b = b+2
```

```
for ( int i=0; i<=b+c/d ; i++)
{ a = b+c/d-e
  b+=2
}
```

IC:

```
    i = 0
L1: t1 = c/d
    t2 = b+t1
    t3 = i <=t2
    if (t3 == false) jmp L2
      t4 = c/d
      t5 = b+t4
      t6 = t5 - e
      a = t6
      b = b+2
      i = i+1
      jmp L1
```

```
do { a = b*c;
   } while ( a < b);
```

```
L1: t1 = b*c
    a = t1
    t2 = a < b
    if (t2 == true) jmp L1
```

- Do switch yourself, it is a combination of if-else
- For break statements we unconditionally jump to the ending label and the continue statements jump unconditionally to to the starting label.

```
while (a<b)
{ b = a+c;
  if (a>5)
  { a++;
    do {
        b+=5;
      } while(a >=b);
  else
  { b=5;
    if (a==7)
        a++;
  }
  b *= 2
}
```

```
L1: t1 = a<b
    if (t1 == false) jmp L5
    t2 = a+c
    b = t2
    t3 = a>5
    if (t3 == false) jmp L3
    a = a+1
L2: b = b+5
    t4 = a>=b
    if (t4 == true) jmp L2
    jmp L4
L3: b=5
    t5 = a==7
    if (t5 == false) jmp L4
    a = a+1
L4: b = b*2
    jmp L1
L5:
```

- It is better to have individual labels instead of reusing labels (like we reused L4 in the above code)
- To push in stack is an architectural dependent task.
- For function's parameters, calling convention is important
  Pascal calling convention: right to left
  C calling convention: left to right

$a = b*c / fn(a - b*c, 5, b) - d$

$t_1 = b*c$

$t_2 = b*c$

$t_3 = a - t_1$

$t_4 = 5$

$t_5 = b$

Param t_5

Param t_4

Param t_3

$t_6 = call\ CN_fn_T_1_T_2_T_3, 3$

$t_7 = t_2 / t_6$

$t_8 = t_7 - d$

$a = t_8$

$b = fn(b + c * sum(a, b*c - d), b + fn(), 75) / e - d$

$t_1 = b*c$

$t_2 = t_1 - d$

$t_3 = a$

* $t_4 = call\ sum_T_1_T_2, 2$

$t_5 = c * t_4$

$t_6 = b + t_5$

$t_7 = call\ fn, 0$

$t_8 = b + t_7$

$t_9 = 75$

(*) $t_{10} = call\ fn_T_1_T_2_T_3, 3$

$t_{11} = t_{10} / e$

$t_{12} = t_{11} - d$

$b = t_{12}$

* param t_2

param t_3

(*) param t_9

param t_8

param t_6

PERS 3/2.

- expressions can include array

a = fn() / e [i+j] [i×j+k]

t_1 = call fn., 0

t_2 = i+j

t_3 = i*j

t_4 = t_3 + k

t_5 = t_1 / e [t_2] [t_4]

a = t_5

- <WHILE> ⟶ while (<OE>) <BODY>

{ if (T != bool) } { create Scope() } { destroyScope() }

for while's intermediate code:

① create label

② code expression returns t

③ if (t == false) jmp L2

④ code body

⑤ jmp L1

⑥ create label outside while

L1 :

[condition]

if (___ == false) jmp L2

[body code]

jmp L1

L2 :

IMPLEMENTATION OF INTERMEDIATE CODE

label CreateLabel ( )

Temp CreateTemp ( )

Output (string)

tindex = 1

Lindex = 1

CreateTemp() { return "t" + tindex++ }

Create Label() { return "L" + Lindex++ }

$$\langle \text{while_st} \rangle \longrightarrow \text{while} \begin{bmatrix} L1 = \text{Create label}() \\ \text{output}(L1 + ":") \end{bmatrix} (\underset{\text{Type, Tname}}{\langle OE \rangle}) \begin{bmatrix} L2 = \text{create label}() \\ \text{Output}(\text{"if("} + \text{Tname} + \text{"==false"} + \text{")jmp"} + L2) \end{bmatrix}$$

$$\langle \text{BODY} \rangle \begin{bmatrix} \text{output}(\text{"jmp"} + L1) \\ \text{output}(L2 + ":") \end{bmatrix}$$

$$\langle \text{do_while} \rangle \longrightarrow \text{do} \{ \langle \text{MST} \rangle \} \text{ while } (\langle OE \rangle);$$

$$\langle \text{do_while} \rangle \longrightarrow \text{do} \begin{bmatrix} L1 = \text{create label}() \\ \text{Output}(L1 + ":") \end{bmatrix} \{ \langle \text{MST} \rangle \} \text{ while } (\underset{\text{Type, Tname}}{\langle OE \rangle}); \begin{bmatrix} \text{Output}(\text{"if("} + \text{Tname} + \text{"==true)jmp"} + L1) \end{bmatrix}$$

$$\langle \text{if_else} \rangle \rightarrow \text{if } (\langle OE \rangle) \langle \text{BODY} \rangle \langle \text{OElse} \rangle$$

$$\langle \text{OElse} \rangle \longrightarrow \text{else } \langle \text{BODY} \rangle \mid \epsilon$$

$$\langle \text{if_else} \rangle \longrightarrow \text{if } (\langle OE \rangle) \begin{bmatrix} L1 = \text{Create Label}() \\ \text{Output}(\text{"if("} + \text{Tname} + \text{"==false)jmp"} + L1) \end{bmatrix} \langle \text{BODY} \rangle \underset{L1}{\langle \text{OElse} \rangle}$$

$$\underset{L1}{\langle \text{OElse} \rangle} \longrightarrow \text{else} \begin{bmatrix} L2 = \text{create label}() \\ \text{Output}(\text{"jmp"} + L2) \\ \text{Output}(L1 + ":") \end{bmatrix} \langle \text{BODY} \rangle \begin{bmatrix} \text{Output}(L2 + ":") \end{bmatrix}$$

condition

if (__ == false) jmp L1

if body

jmp L2

L1 :

else body

L2 :

$\langle fn\text{-}def \rangle \longrightarrow \langle RT \rangle_{R} \; ID_{N} \; (\langle PL \rangle_{P}) \; \{ \langle MST \rangle \} \; [\text{output}(\text{"EndP"})]$

$[\text{output}(CN + \text{"-"} + N + \text{"-"} + P + \text{"Proc"})]$

$a + b * c / d - e$

$t_1 = b * c$	$T_1 = Comp(T_b, T_c, *)$
$t_2 = t_1 / d$	$T_2 = Comp(T_1, T_d, /)$
$t_3 = a + t_2$	$T_3 = Comp(a, T_2, +)$
$t_4 = t_3 - e$	$T_4 = Comp(T_3, e, -)$

- copy a.b. as it is in the intermediate code