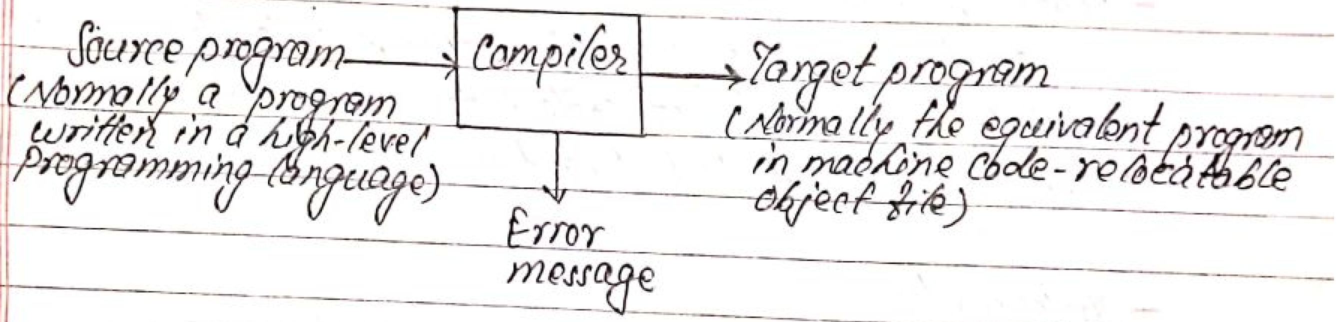
Compiler:

A compiler is a program that takes a program written in a source language and translates it into an equivalent program in a target language.



- Compiler – Scans the entire program and translates it as a whole into machine code.
- Interpreter – It translates one line of source code at a time into machine code and the executes it.

Compiler vs Interpreter

Compiler	Interpreter
- Overall execution time is faster	- Overall execution time is Comparitively slower.
- The resulting executable is some form of machine-specific binary code.	- The resulting code is some sort of intermediate code.
- Memory requirement is more.	- Memory requirement is less.
- Takes large amount of time to analyze source code.	- Takes less amount of time to analyze the source code.
- E.g. C, C++.	- E.g. Python, Ruby.

Cousins of the compiler

- Pre-processor:
  Translates source code into simpler or slightly lower level source code, for compilation by another compiler.

- Loader & linkers:
  If the target program is machine code, loaders are used to load the target code into memory for execution. Linkers are used to link target program with the libraries.

- Interpreter:
  Interpreter performs compilation, loading and execution in lock-steps.

- JIT compiler (Just in time):
  JIT compilers perform complete compilation immediately followed by linking & loading.

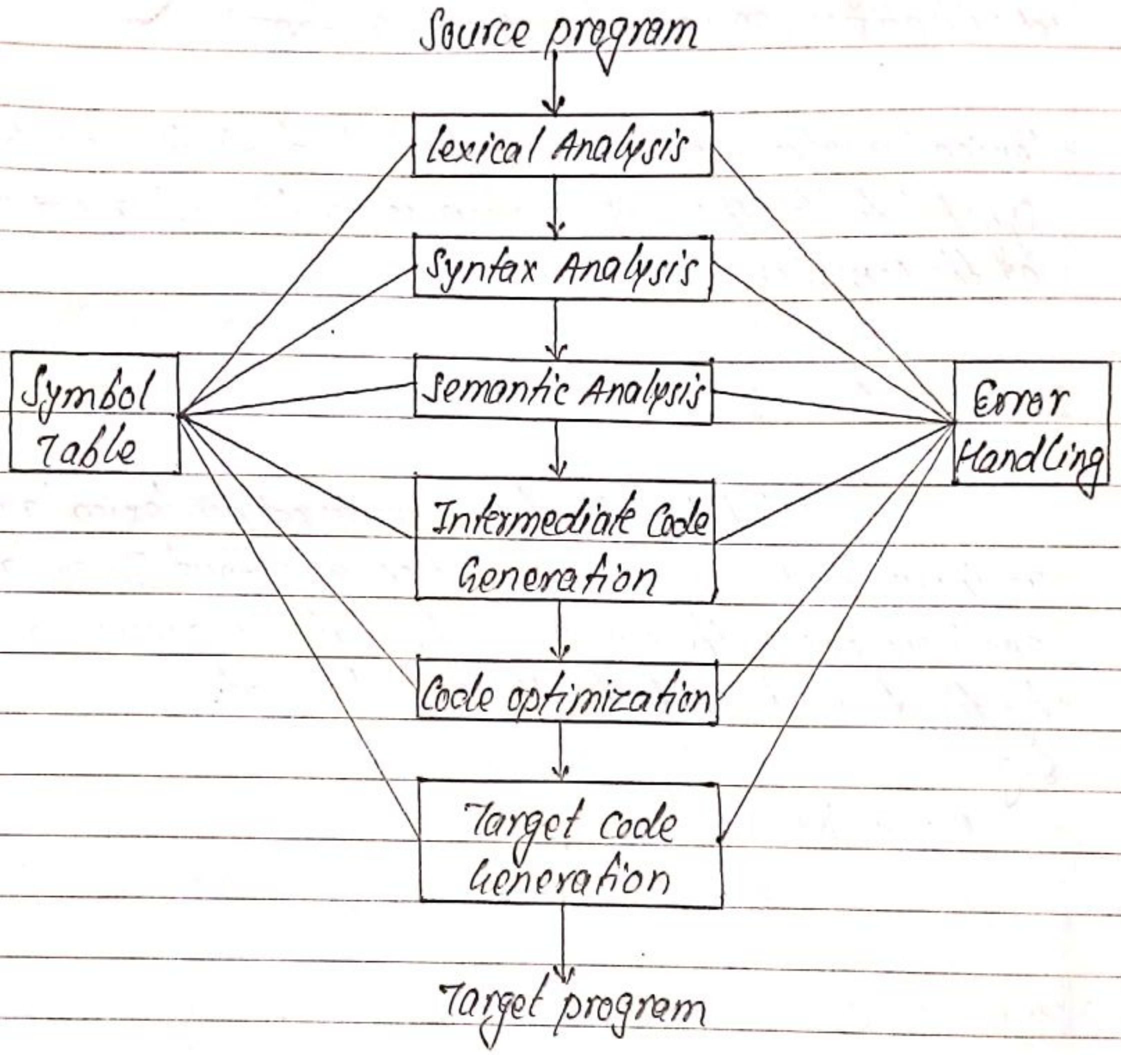
Phases of compiler

There are two major parts of compiler : Analysis and Synthesis.

- Analysis phase breaks up the source program into constituent pieces and creates an intermediate representation of the source program. Analysis of source program includes : lexical analysis, syntax analysis & semantic analysis.

- Synthesis phase constructs the desired target program from the intermediate representation. The synthesis part of compiler

Consists of the following phases: Intermediate code generation, code optimization and Target code generation.

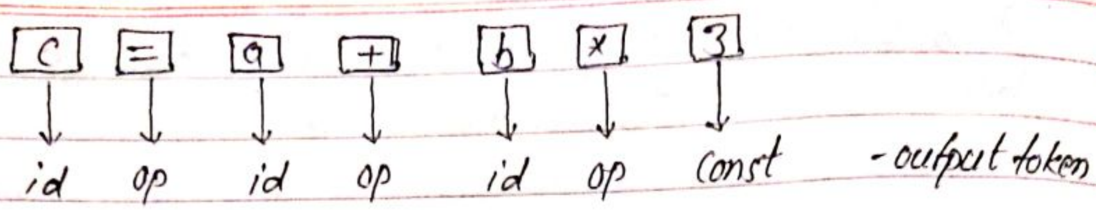


1. Lexical Analysis:

It is the first phase of compiler. In this phase, lexical analyzer reads the source program and returns the tokens of the source program. Token is a sequence of characters that can be treated as a single logical entity. (such as identifiers, operators, keywords, constants etc.)

For. e.g.

$c = a+b*3$ - Input string



id : identifier , op : operator, const : constant

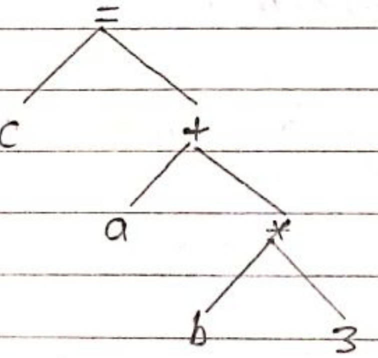
- lexical analyzer puts information about identifiers into the symbol table. This information is used in subsequent passes of the compiler.

2. Syntax Analysis:

It takes the token produced by lexical analysis as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the token is syntactically correct or not.

E.g.

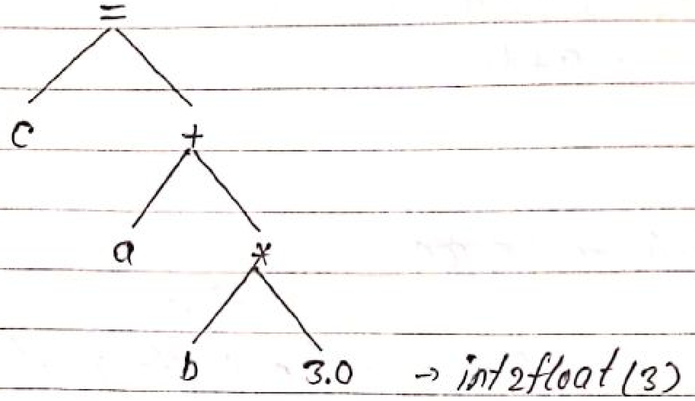
c = a+b*3



3. Semantic Analysis:

Semantic analysis checks whether the parse tree constructed follows the rules of language. semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

E.g.



4. Intermediate Code Generation :

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that it ~~makes~~ can easily translated into the the target machine code.

E.g.

$t_1 = 3.0$; int2float(3)

$t_2 = b * t_1$;

$t_3 = a + t_2$;

$c = t_3$;

5. Code optimization :

It is used to improve the intermediate code so that the output of the program could run faster and takes less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution without wasting the resources.

E.g.

$t_2 = b * 3.0$;

$c = a + t_2$;

6. Code Generation:

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language.

E.g.

```
MOV  R1, b
MUL  R1, 3.0
MOV  R2, a
ADD  R1, R2
MOV  c, R1
```

Symbol Table

A symbol table stores information about keyword & tokens found during the lexical analysis. The symbol table is consulted in almost all phases of compiler.

Error Handling

It is responsible for handling of the errors which can occur in any phase of the compilation.

E.g.

- Handling missing symbols during the lexical analysis by inserting symbol.
- Automatic type conversion during the semantic analysis.

※

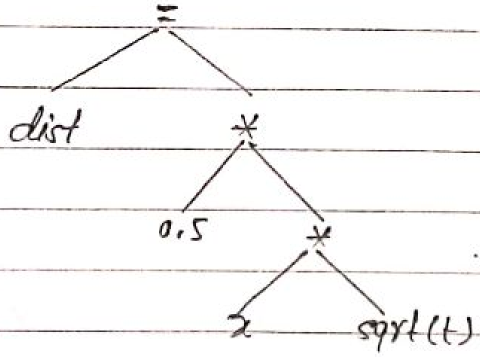
dist = 0.5 * x * sqrt (t) ;   Input string

lexical analysis

$\boxed{dist}$ $\boxed{=}$ $\boxed{0.5}$ $\boxed{*}$ $\boxed{x}$ $\boxed{*}$ $\boxed{sqrt(t)}$ ⟶ symbol table
id op const op id op fncall

token – identifier
lexeme – dist

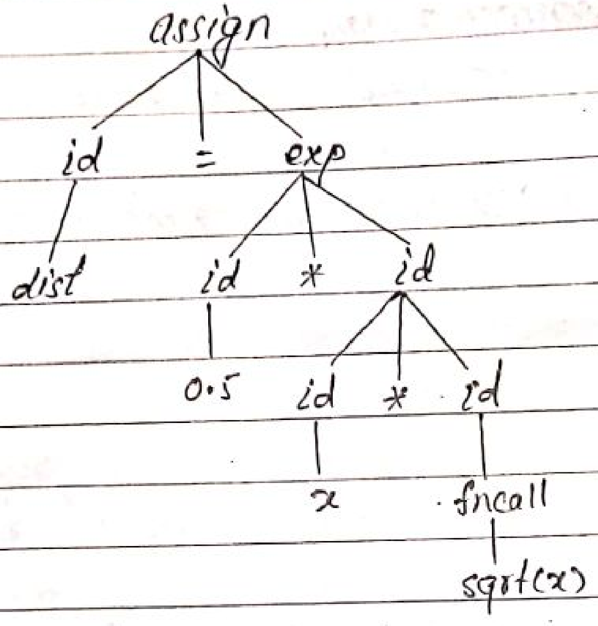
syntax analysis



- If

G =

assign = id = exp
id → . . . .
exp = id+id / id-id / id * id / id



Intermediate code

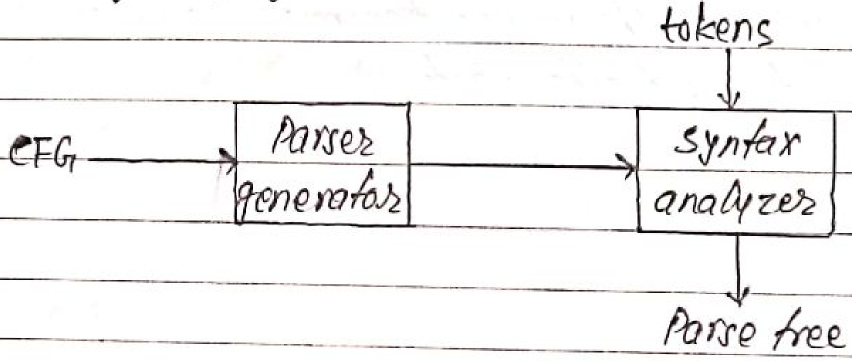
```
t1 = int2float (sqr(t)):
t2 = x * t1;
t3 = 0.5 * t2;
dist = t3;
```

Compiler Construction Tools

Some commonly used compiler-construction tools include;

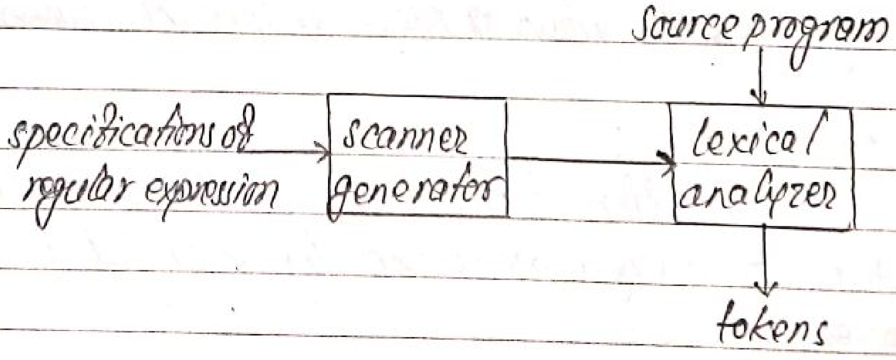
1. Parser Generators:

Parser generator takes the grammatical description of a programming language and produces a syntax analyzers (parsers).



2. Scanner Generators:

Scanner generator generates lexical analyzer from the input that consist of regular expression description based on tokens of a language.



3. Syntax-directed translation Engine :

Syntax-directed translation engine produce collections of routines that walk a parse tree and generates intermediate code.

4. Automatic code generators :

Automatic Code generators are used to generate the machine language for the target machine by translating each operation of the intermediate language using a collection of rules that define those translation.

5. Data flow Engines:

It is used in code optimization. Data flow analysis is a key part of the code optimization that gathers the information, i.e. the values transmitted from one part of a program to each other parts.

Phases and Passes

→ Each individual unique step in compilation process is called a phase such as lexical analysis, syntax analysis & so on.

→ Different phases may be combined into one or more than one group, this group of phases makes the concept of passes.

- One-pass compiler:

If different phases of compilation process are grouped into a single group then this is called as one pass compiler

- Multipass compiler:

Compilation process grouped into different phases.

- Two-pass compiler:

All phases of compiler are grouped into two phases: Analysis & synthesis.

One-pass vs Multipass Compiler

One-Pass Compiler	Multi-pass Compiler
1. In a single pass compiler all the phases of compiler are grouped into one pass.	1. In multi-pas compiler the different phases of a compiler are grauped into different plases.
2. A one-pass compiler is a compiler that passes through the source code of each compilation unit only once.	2. A multi-pass compiler processes the source code of a program several times.
3. It does not look back at code it previously processed.	3. Each pass takes the result of the previous pass as the input and creates an intermediate output.
4. It is faster than multi-pass compiler.	4. It is slow comporative to one-pass compiler.
5. Has a limited scope.	5. Has a great scope.
6. There is no intermediate code generation.	6. There is intermediate code generation.
7. Memory consumption is lower	7. Memory consumption is higher
8. Eg: Pascal's compiler	8. E.g: C++ Compiler

→ To describe the one pass compiler, the key points are:

- Grammar
- Derivation
- Parse tree
- Ambiguity
- Associativity of operators (left to right)
- Syntax directed translation

→ A language is described by syntax definition
- Grammar (CFG) or Backus Naur Form (BNF)
- Syntax Directed Translation to get intermediate code.
- Translation scheme.

→ CFG have four basic components:
1. A set of terminal symbols that represent tokens.
2. A set of non-terminal symbol
3. A set of production
4. A start symbol.

Let us take example for converting an infix expression containing digits and + & - operators into postfix / prefix notation.

Grammar

list → list+list | list-list | digit
digit → 0|1|2|3|4|5|6|7|8|9

{list, digit → non-terminal, other termi

~~Infix ex~~ Input infix exp^n : 9-5+2

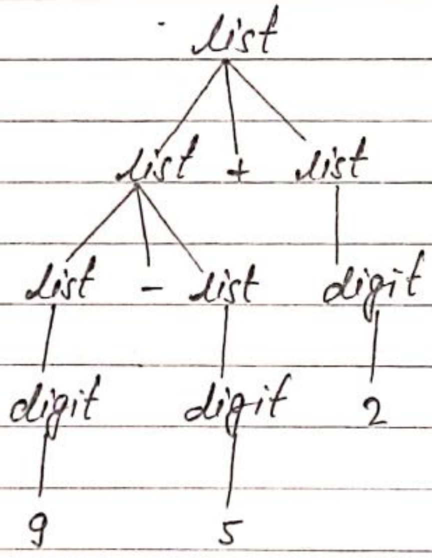
Derivation

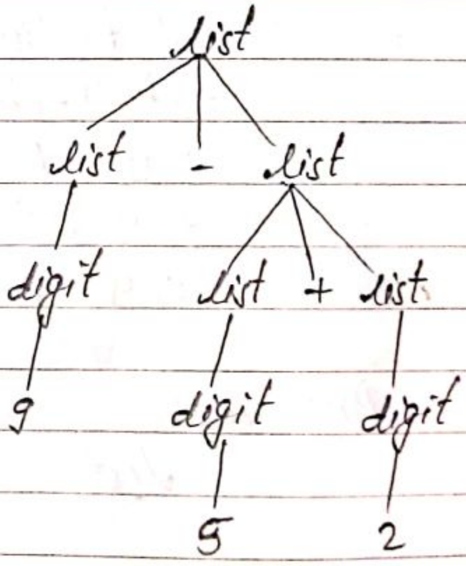
Start with start symbol and replace non-terminal repeatedly by the right side of production for that terminal.

list ⇒ list + list	list ⇒ list - list
⇒ list - list + list	⇒ digit - list
⇒ digit - list + list	⇒ 9 - list
⇒ 9 - list + list	⇒ 9 - list + list
⇒ 9 - digit + list	⇒ 9 - digit + list
⇒ 9 - 5 + list	⇒ 9 - 5 + list
⇒ 9 - 5 + digit	⇒ 9 - 5 + digit
⇒ 9 - 5 + 2	⇒ 9 - 5 + 2

Parse tree:

Parse tree is the pictorial representation of the derivation of a string from the start symbol of grammar.





Ambiguity

If there are more than one same type of derivation for a string i.e. more than one distinct parse tree can be formed.

Associativity of operator

If there are operators to the left as well as right of any operand, then associativity of operator plays the role i.e. which operator to be taken first for that operand.

$9-5+2 \Rightarrow (9-5)+2$

$9+2-5 \Rightarrow (9+2)-5$

$9+5*4 \Rightarrow 9+(5*4)$

The unambiguous grammar for infix expⁿ can be written as

list → list + digit | list - digit | digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

string : 9 - 5 + 2

list ⇒ list + digit

⇒ list - digit + digit

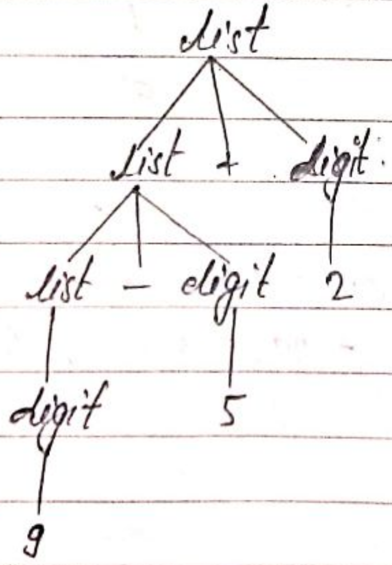
⇒ digit - digit + digit

⇒ 9 - digit + digit

⇒ 9 - 5 + digit

⇒ 9 - 5 + 2

Parse tree :

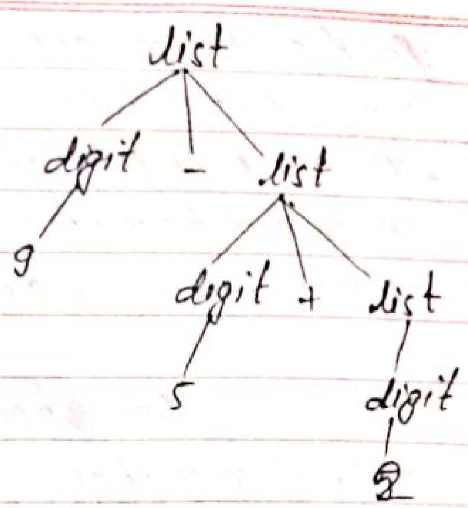


Annotated / Decorated.

If grammar

list → digit + list | digit - list | digit

list ⇒ digit - list
⇒ 9 - list
⇒ 9 - digit + list
⇒ 9 - 5 + list
⇒ 9 - 5 + digit
⇒ 9 - 5 + 2



Recursive Descent Parsing

Based on the semantic rules, we have to define the semantic actions associated with each production of the grammar.

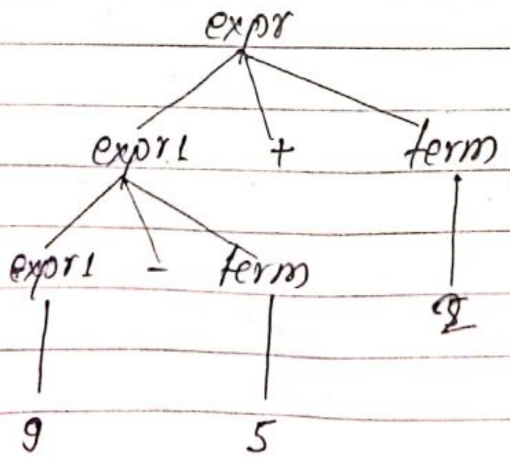
- The CFG with semantic action embaded with each production is called translation schemes. Here each semantic action are placed within { } at the position of execution.

Translation scheme of infix to prefix based on the defined semantics

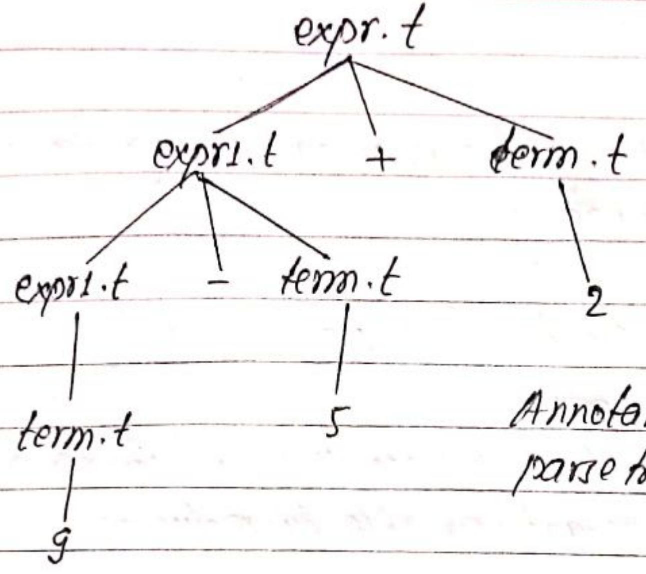
expr → expr1 + term {print '+'}
expr → expr1 - term {print '-'}
expr → term
term → 0 {print '0'}
term → 1 {print '1'}
term → 2 {print '2'}
⋮
term → 9 {print '9'}.

for e.g.
A string 9-5+2

1. Construct parse tree

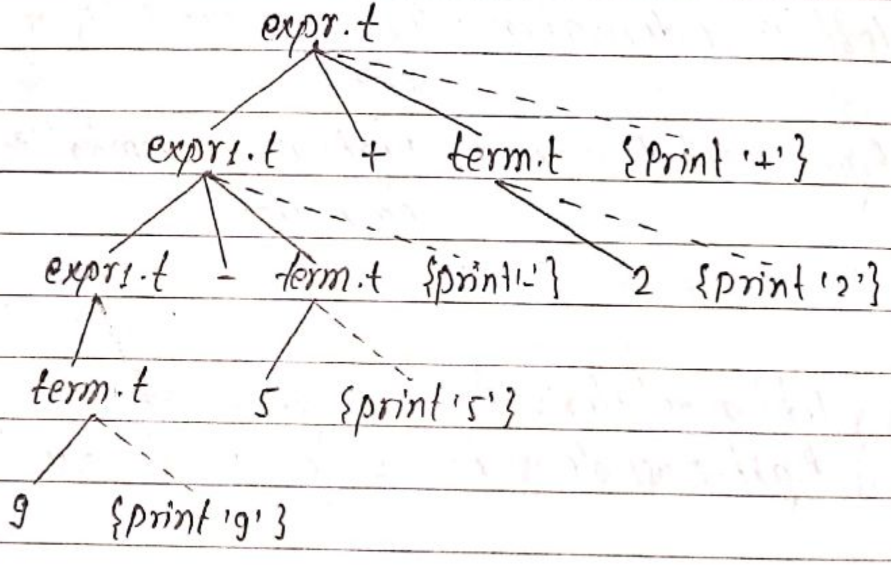


2. Assign the attributes like x.a where x is a non terminated and 'a' is attribute.



Annotated parse tree or decorated parse tree.

3. Use semantic action to the annotated tree



4. DFT (Depth first Traverse) the annotated tree to execute semantic action to find the postfix string.

First action - print '9'
2nd action - print '5'
3rd action - print '-'
4th action - print '2'
5th action - print '+'

⇒ 95-2+ postfix of 9-5+2

Parsing

- process of deriving the terminal string from the defined grammar (Derivation).

Types:

Top-down : starting from start variable to get final terminal string.

Bottom-up : starting with terminal string and get the start variable by reduction.

On derivation, which symbol is replaced first that defines

- left-most derivation : left most symbol is replaced at each step.
- Right-most derivation : Right most symbol is replaced at each step.

{ left-most derivation ⇒ right most reduction }
{ Right-most derivation ⇒ left most reduction }

- Lookahead : The current symbol scanned from the input string.

Consider a grammar

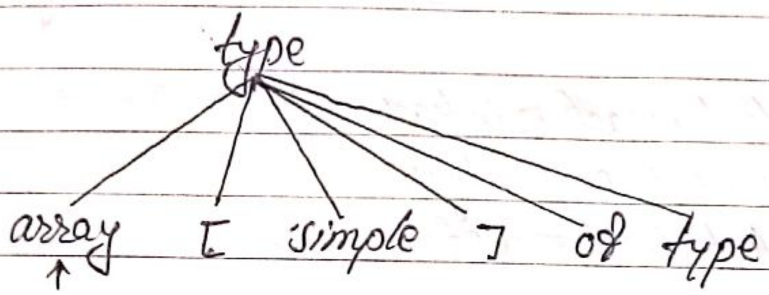
type → simple | ↑id | array [simple] of type.
simple → integer | char | num dotdot num

Here type, simple are non-terminal and others are terminal.
This grammar defines the syntax of simple type expression in

a language like pascal.

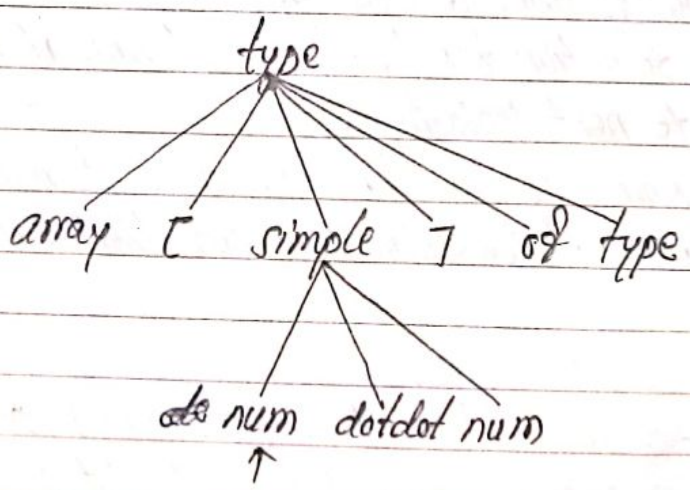
Input: array [num dotdot num] of integer

The parse tree for this string start with type (start symbol
- current lookahead symbol: array

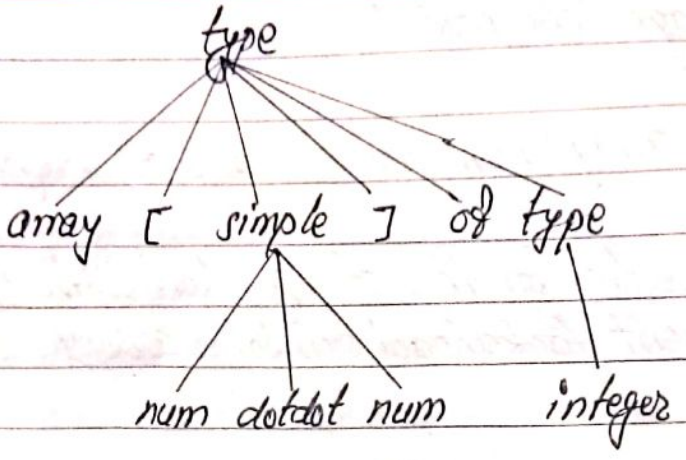


lookahead = next symbol = [

From parse tree, the current symbol = [, lookahead = nextsymbo
current lookahead = num, symbol at parse tree = simple
so replace simple with num dotdot num
parse tree



Here, num dotdot] of matches in parse tree with lookahead.
Now lookahead integer, symbol at parse tree type.
Replace type with integer its production.



Here, lookahead → integer
From parse tree → integer
next lookahead = eof

So, parsing complete.

To design the top down parser for the grammar.
- Define procedure (function) for each non-terminal symbol.
- Scan input symbol left to right and if it is non-terminal, call the function for it, if it is terminal match the terminal and go to next lookahead.
- At any point if the terminal does not matches - parse error
- At the end if lookahead is eof, then parsing complete.

So for above grammar,
type → simple | ↑id | array[simple] of type
simple → integer | char | num dotdot num

1. function to check matching lookahead

```
match ( Token t )
{
    if ( lookahead == t )
```

```
        lookahead = next_token ;
    else
        Parse_Error ;
    }
```

2. function for type

```
    type ( )
    {
        if (lookahead is in (integer, char, num))
            simple ( ) ;
        else if (lookahead = '↑')
        {
            match ('↑');
            match ('id');
        }
        else if (lookahead = 'array')
        {
            match ('array');
            match ( '[' );
            simple ( );
            match (']');
            match ('of');
            type ( );
        }
        else
            Parse_error;
    }
```

3. Function simple ( )

```
simple()
{
  if (lookahead = 'integer')
     match('integer');
  else if (lookahead = 'char')
     match('char');
  else if (lookahead = 'num')
  {
     match('num');
     match('dotdot');
     match('num');
  }
  else
     parse_error;
}
```

Q.

Grammar

expr → expr + term
expr → expr - term
expr → term
term → 0/1/2/3/4/5/6/7/8/9

- Remove the left-recursion from the grammar.
- Define the semantic rules, semantic action (translation schemes) for postfix.
- Construct the parse tree with semantic action to convert into postfix.

Sol^n

- Removing left recursion

$expr \rightarrow term\ rest$
$rest \rightarrow +\ term\ rest$
$rest \rightarrow -\ term\ rest$
$rest \rightarrow \epsilon$
$term \rightarrow 0|1|2|3|4|5|6|7|8|9$

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta$

$$\left[\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \epsilon \end{array}\right]$$

⇓ Removing ε

$$\left[\begin{array}{l} A \rightarrow \beta A' \mid \beta \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_1 \mid \alpha_2 \end{array}\right]$$

- Semantic rule for translating infix to postfix

Production	sem. Rules
$expr \rightarrow term\ rest$	$expr.t = term.t \,\|\, rest.t$
$rest \rightarrow +\ term\ rest$	$rest.t = term.t \,\|\, '+' \,\|\, rest.t$
$rest \rightarrow -\ term\ rest$	$rest.t = term.t \,\|\, '-' \,\|\, rest.t$
$rest \rightarrow \epsilon$	$rest.t = \epsilon$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
⋮	⋮
$term \rightarrow 9$	$term.t = '9'$

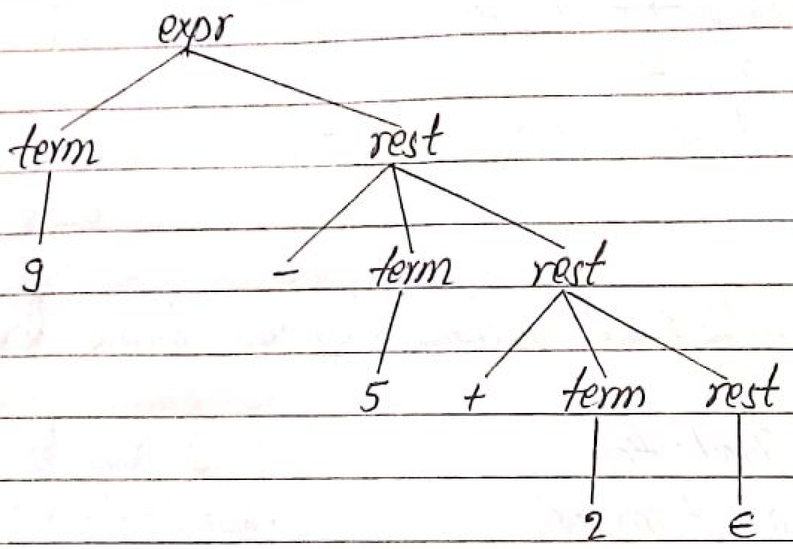
syntax directed defn

Translation schemes

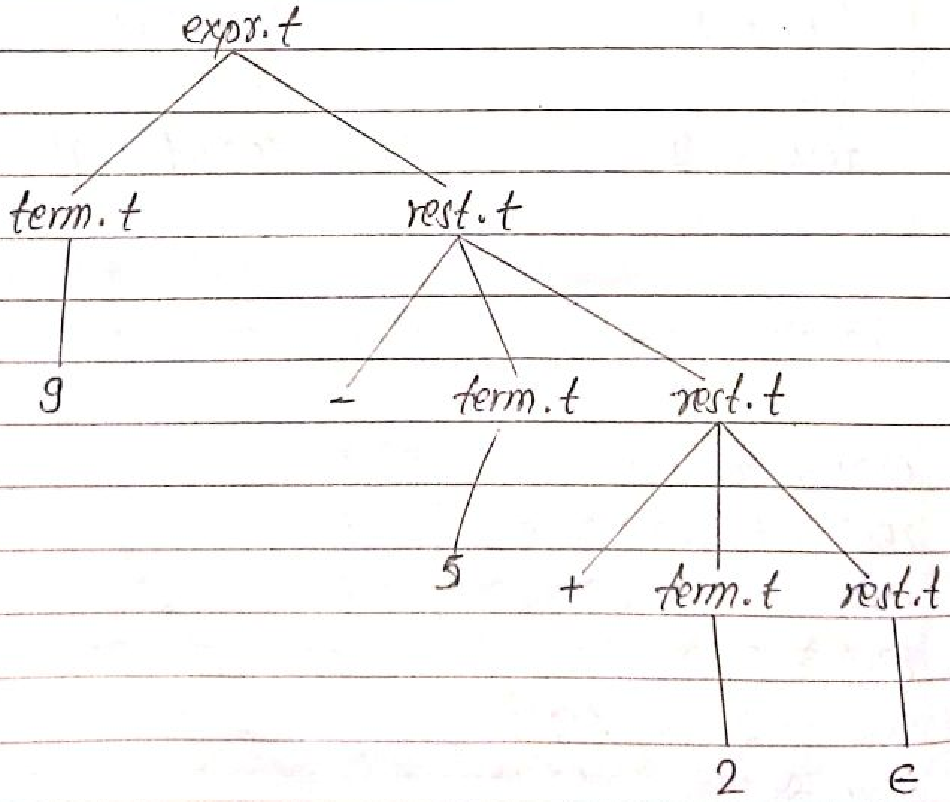
$expr \rightarrow term\ rest$
$rest \rightarrow +\ term\ \{print\ '+'\}\ rest$
$rest \rightarrow -\ term\ \{print\ '-'\}\ rest$
$rest \rightarrow \epsilon$
$term \rightarrow 0\ \{print\ '0'\}$
$term \rightarrow 1\ \{print\ '1'\}$
⋮
$term \rightarrow 9\ \{print\ '9'\}$

9 - 5 + 2

Parse tree:

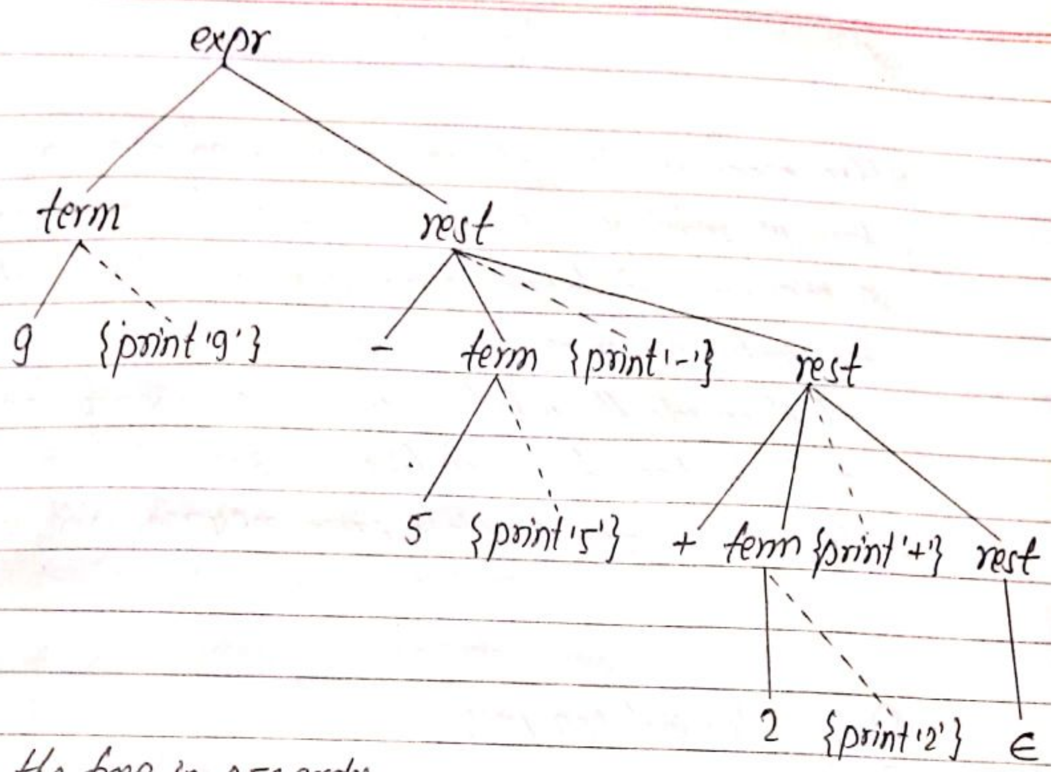


$\Downarrow$ Annotated parse tree



⇒ Parse tree with semantic actions

Attributes → 2 types
→ synthesized (child to parent)
→ Inherited (parent to child or sibling to sibling)



Traverse the tree in DFS order
95-2+ → postfix

* Infix to postfix translator function

```
match()

expr()
{
    term();
    rest();
}
term()

rest()
{
  if lookahead('+')
      match('+');
  else
      lookahead('-')
      match('-')
      term();
      rest();
  }
  error()
```

← Rough sketch मात्र हो। Complete गर्नुपर्छ।