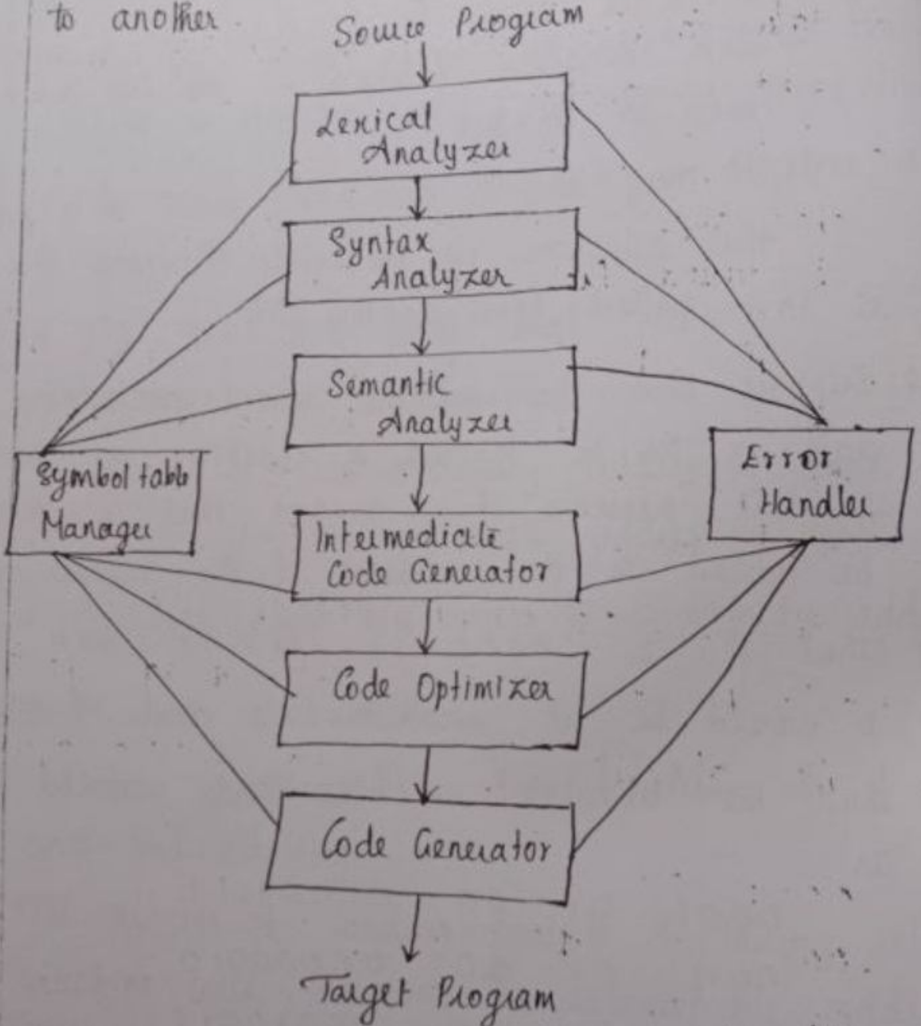
IV THE PHASES OF COMPILER

* A compiler operates in phases, each of which transforms the source program from one representation to another.



* Symbol table Management and error handling interact with the six phases of lexical analysis, syntax Analysis, Semantic Analysis, Intermediate code Generation, Code optimization and code Generation

Analysis of the Source Program.

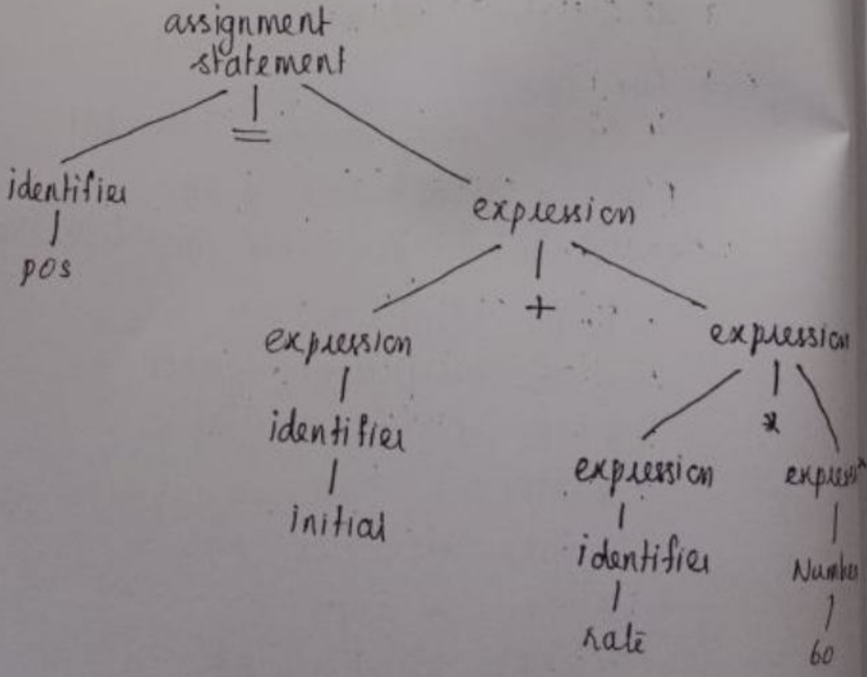
* In compiling analysis consists of three phases.
  1. Linear Analysis
  2. Hierarchial Analysis
  3. Semantic Analysis

1. Linear Analysis:

* Linear analysis is called as lexical analysis or scanning.
* Linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.
* For eg)
  In the assignment statement
  pos = initial + rate * 60
  would be grouped into the following tokens.
  1. The identifier pos
  2. The assignment symbol =
  3. The identifier initial
  4. The + sign
  5. The identifier rate
  6. The * sign
  7. The number 60
* The blanks separating the characters of these tokens would be eliminated during lexical analysis

2 <u>Hierarchial Analysis</u>:

* Hierarchial analysis is called as parsing or syntax analysis
* Hierarchial analysis in which characters or tokens are grouped hierarchially into nested collections with collective meaning.
* It groups the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
* The grammatical phrases of the source program are represented by a parse tree.



* The h
expressi

* The

1.
2.
3.

* Rul
defi
to
* The
1.
2.

ng or"

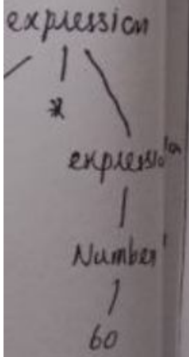
tokens

ons with

ram into

compiler

ogram



* The hierarchial structure of a program is expressed by recursive rules.

* The rules of the definitions of expressions
  1. Any identifier is an expression
  2. Any number is an expression
  3. If $expression_1$ and $expression_2$ are expressions then

     $expression_1 + expression_2$

     $expression_1 * expression_2$

     $(expression_1)$

     are expressions.

* Rules (1) and (2) are basic rules, Rule (3) defines expressions in terms of operators applied to other expressions.

* The rules of the definition of statement
  1. If $identifier_1$ is an identifier and $expression_2$ is an expression then

     $identifier_1 = expression_2$

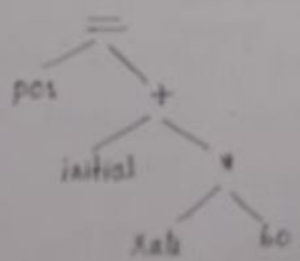
     is a statement
  2. If $expression_1$ is an expression and $statement_2$ is a statement then

     while ($expression_1$) do $statement_2$

     if ($expression_1$) then $statement_2$

     are statements

* The division of lexical and syntax analysis simplifies the overall task of the analysis
* The parse tree is a syntactic structure of the input
* A more common internal representation of this syntactic structure is given by the syntax tree
* A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the leaves are operands of an operator



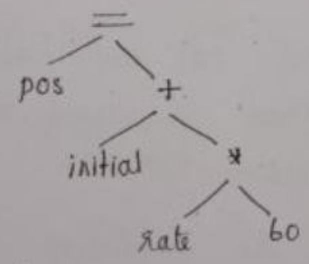
Semantic Analysis:

* This phase checks the source program for semantic errors and gathers type information for the code generation phase
* It uses the hierarchial structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements
* An important component of semantic analysis is type checking

* Here the
Operands
specification
* For eg)
require
a real
* For eg)
Type
a real
approach
* This is
for the
integer

Symbol
* An im
the id
collect
of each

* The division of lexical and syntax analysis simplifies the overall task of the analysis
* The parse tree is a syntactic structure of the input
* A more common internal representation of this syntactic structure is given by the syntax tree
* A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the leaves are operands of an operator



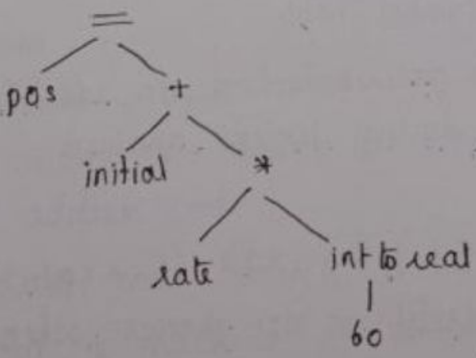
Semantic Analysis:

* This phase checks the source program for semantic errors and gathers type information for the code-generation phase
* It uses the hierarchial structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements
* An important component of semantic analysis is type-checking.

* Here the
Operands
specifical
* For eg)
require
a real
* For eg)
Typ
a rea
approo
* This
for t
integ

Symbo
* An
the
coll
of

* Here the compiler checks that each operator has operands that are permitted by the source language specification
* For eg) many programming language definitions require a compiler to report an error every time a real number is used to index an array.
* For eg)
  Type checking reveals that * is applied to a real rate and integer 60. The general approach is to convert the integer into a real
* This is achieved by creating an extra node for the operator inttoreal that converts the integer into a real.



Symbol Table Management:

* An important function of a compiler is to record the identifiers used in the source program and collect information about the various attributes of each identifier

* These attributes may provide information about the storage allocated for an identifier, its type, its scope. in the case of procedure names, things such as the number and type of its arguments, the method of passing each arguments and the type returned
* A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier.
* The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.
* When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table
* However, the attributes of an identifier cannot normally determined during lexical analysis.

For eg)

var pos, init, rate : real

the type real is not known when pos, init, rate are seen by the lexical analysis.

* The remaining phases enter information about identifier into the symbol table and use this information in various way.

Error Det

* Each
detect
error
further
detect

Intermedia

* After
compile
of the
* The in
proper
1.
2
* The
of f
and
* The
lang
can
* The
add

Error Detection and Reporting:

* Each phase can encounter errors. However after detecting an error a phase must deal with the error so that compilation can proceed. allowing further errors in the source program to be detected.

Intermediate Code Generation:

* After syntax and semantic analysis phase, some compilers generate an explicit intermediate representatio of the source program.
* The intermediate representation should have two properties
  1. It should be easy to produce
  2. It should be easy to translate into target program.
* The intermediate representation can have variety of forms such as, syntax tree, postfix notation. and three address code.
* Three address code which is like assembly language for a machine in which every location can act like a register.
* The source program might appear in three? address code as

$$
\begin{aligned}
temp1 &= inttoreal(60)\\
temp2 &= id_3 * temp_1\\
temp3 &= id_2 + temp2\\
id_1 &= temp_3.
\end{aligned}
$$

* This intermediate form has several properties.
  1. Each three address instruction has atmost one operator in addition to the assignment operator.
  2. The compiler must generate a temporary names to hold the value computed by each instruction
  3. Some 'three address" instruction have fewer than three operands. eg)
     first and last instructions.

Code Optimization:

* The code optimization phase attempts to improve the intermediate code so that faster running machine code will result.

$$
\begin{aligned}
temp1 &= id_3 * 60.0\\
id_1 &= id_2 + temp1
\end{aligned}
$$

* However, there are simple optimizations that significantly improve the running time of the target program without slowing compilation time too much.

Code C

* The gene reta

* The inte perf

* An regi

Code Generation

* The final phase of the compiler is the generation of target code, consisting of relocatable machine code or assembly code.
* The intermediate instructions are translated into a sequence of machine instructions that perform the same task.
* An aspect is the assignment of variables to registers.

```
MOVF  id3, R2
MULF  *60.0  R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id
```