భారతీయ సాంకేతిక విజ్ఞాన సంస్థ హైదరాబాద్
भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

fns

finculate-not-speculate

Project Report

Members:

DIYA GOYAL - CS20BTECH11014
DONTHA AARTHI - CS20BTECH11015
NYALAPOGULA MANASWINI - CS20BTECH11035
SUSHMA - CS20BTECH11051
KHARADI MONIKA - CS20BTECH11026
NAMITA KUMARI - CS20BTECH11034
MD ADIL SALFI - CS20BTECH11031

Written by: Manaswini, Diya, Aarthi

Project Roles

Diya Goyal :	Project Manager
Nyalapogula Manaswini :	System Architect
Dontha Aarthi :	System Architect
Sushma :	System integrator
Namita Kumari :	Language Guru
Kharadi Monika :	Tester

Contents

1. Introduction

FNS - A procedural programming language to ease financial calculations. It has inbuilt functions to calculate simple interest, compound interest, SIP and step-up SIP maturity returns. The goal is to do complicated calculations in the backend so that the user has an exact value for the amount of money involved in the picture.

All calculations involving bank loans, returns and investments have the formula of compound interest at its core. Doing mental calculations is non-intuitive because compound interest has an exponential graph, hence making estimates is very difficult. Our language can be a useful tool to automate calculations for the end-user so that they can make better decisions that affect their financial health.

Functions like:
Simple Interest: Calculates the simple interest, given principal, interest rate and time.

$$\mathbf{S.I = P \times T \times R / 100}$$

where,
P = principal
T = time
R = interest rate

Compound Interest: Calculates compound interest amount, given principal, interest rate, number of times interest is compounded per year, time in years.

$$\mathbf{A = P \times (1 + r/n)^{nxt}}$$

where,
A = final amount
P = principal
r = interest rate
n = number of times interest is compounded per year
t = time in years

SIP Maturity: Gives idea about the returns on their investments made through SIP.

$$\mathbf{M = P \times ((1 + r)^{n} - 1) \times (r + 1)/r}$$

where,

M = SIP maturity
P = monthly investment
n = number of months
r = monthly interest rate

SIP maturity details are:

Investment = P x n
Interest = M - Investment
Return percentage = Interest x 100 / Investment

2. Language Tutorial

2.1 Structure of program

Note: All statements end with a semicolon in our language.

- **execute function**

Our program consists of the declaration of an execute function which is similar to the main function in C/C++ language. It takes no input and no output parameters. The main code of the program is contained in this. Other user-defined functions can be defined outside the execute function.

Below is an example of the structure of execute function:

```
execute: ( ) -> ( )
{
        display("Hello World!");
}
```

- **display function**

display function is an in-built function which is used to display the statements, variable values, etc.

Syntax for the display function:

```
display("Hello World!");
display("a = %d", a);
```

- **accept function**

accept function is an in-built function which is used to accept the input from the user.

Syntax for the accept function:

```
accept("%s", "hello world!");
```

2.2 Variable declaration

Here in the declaration, we have a data type followed by a variable name. Variable names should not be a keyword.

Below shown is the syntax of the variable declaration:

int number;
double decimal;
string s;
date d;
month m1, m2;

2.3 Variable Initialisation

Below is the syntax for variable initialisation:

int number = 10;
double deci = 0.456;
string s = "Hello";
date d = 01_10_2022;
month m1 = 11_2021, m2 = 12_2022;

2.4 Function declaration

Here in the declaration, we have function name followed by colon (:) , followed by input parameters along with respective data types closed in parentheses, followed by arrow (->) and output parameters along with respective data types in parentheses.
And the body of the function is enclosed in curly brackets.

Below is an example of function declaration:

```
foo : (int a, int b) -> (int c)
{
        c = a + b;
}
```

3. Language Reference Manual

3.1 Lexical Analysis

- **comments**

Both single and multiline comments are supported.
All tokens after **$$** are treated as comments and are ignored by the compiler.
Multiline comments start with **$/** and end with **/$**, anything between these 2 characters will be ignored.

Example:
```
$$ This is a single line comment
$/ This is a
    multi-line comment /$
```

- **identifiers**

Identifiers must start with a letter, which can be followed by a sequence of letters, digits, and underscores. Other special characters cannot be used in the identifier. This language is case sensitive, so App, app and aPp are all considered different.

- **key words**

Following are reserved keywords in our language, they cannot be used as identifiers.

int	double	date	month	if	else
continue	loop	fbreak	break	string	

- **constants**
  - ☐ **Integer constants (int) :** consists of numbers both positive and negative.
  - ☐ **Decimal constants (double) :** consists of floating point numbers both positive and negative.
  - ☐ **Strings (string) :** contains sequence of letters enclosed in “ ”.

- ☐ **Date (date) :** contains date, month and the year in dd_mm_yyyy format.
- ☐ **Month (month) :** contains month along with the year in mm_yyyy.

- **Operators**

Operator	**Description**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

- **Relational operators**

>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
==	Equal
!=	Not Equal

- **Logical operators**

&	Logical And
|	Logical Or

!	Logical Not

- **Assignment operators**

=	Assignment
+=	Addition and assignment
-=	Subtraction and assignment

- **Precedence and Associativity**

Decreasing order of precedence:

( )	Left to right
!	Left to right
^	Left to right
* / %	Left to right
+ -	Left to right
> < >= <= == !=	Left to right
& |	Left to right
=	Right to left

3.2 Data Types

- **int :** consists of numbers both positive and negative.
  Example: int a = 3, b = -5;
- **double :** consists of floating point numbers both positive and negative.
  Example: double a = 9.8, b = -0.9;
- **string :** contains sequence of letters enclosed in “ ”.
  Example: string s = “hello”;
- **date :** contains date, month and the year in dd_mm_yyyy format.

Example: date = 01_04_2022;

- **month :** is month along with the year in mm_yyyy.
  Example: month = 08_2022;

3.2 Expressions

- **Expressions and operators**

Expressions are combinations of identifiers and operators.
The operators include “+”, “-” , “*” , “/” , “%” , etc.

```
expr: expr op expr
    | “(” expr “)”
    ;
```

- **Statements**

Statements are selection statements, jump statements, variable declarations, iteration statements, function calls, print statements, input statements.
All statements end with a semicolon in our language.

- **fbreak, continue, break**

fbreak is used to break and come out of the function.
continue is used to bring the program control to the beginning of the loop.
break is used to get out of a loop.

3.3 Project Execution

1. make all
2. make t1 (running test cases)
3. make e1 (running error test cases)

This will output a token stream, syntax errors (if any), ast, symbol table and semantic errors (if any).

4.Project Plan

1. Discussed the idea and implementation of the language. Decided on the roles. Filled the google form regarding the details of tools, and a few broad specifications.
2. Discussed the details of the inbuilt functions supported by the language. Came up with the syntax of the language and wrote the white paper and submitted the assignment 1.
3. We have started writing code for the lexer using Lex. Also writing test cases for testing the lexical analyser.
4. Finished the lexical analyser part, recorded demo videos and made presentations and submitted the lexical analyser. We were also finding and exploring resources to start with the parser.
5. Started working on the parser implementation - started working on the grammar rules and functions that are required. We used yacc/bison to implement the parser.
6. Completed writing the grammar (which was very large). We were working on reducing the conflicts and made some changes in the lexer so that the lexer and parser work hand in hand.
7. Made some minor changes to the parser in order to solve the conflicts. Recorded the demo videos and made the presentation slides and submitted the parser and updated lexer.
8. We now started to read about semantic analysis and how to implement it.
9. We were not able to find a proper resource and were still trying ways to start the next phase.
10. Now when we got some idea about the next step, that is, implementing the symbol table, we realized that our grammar was very large and it had a lot of rules that were similar and are not required at all.
11. Then we started working on the grammar and made it short and brief and started to work on the structure of the symbol table. Wrote a few basic functions like adding an entry to the symbol table, printing the table.
12. We wrote the necessary actions, made some minor changes in accordance with the symbol table and wrote functions for searching in the symbol table.
13. Started exploring the ast implementation. Also started working on the ast, added a structure for it, implemented the nodes and made the tree structure.

14. Added actions, functions for the ast tree traversal and for printing the ast tree.
15. Completed the ast tree implementation and started with the semantic analysis.
16. Exploring the semantic analysis, the semantic errors, handling and all.
17. Started with semantics, wrote the required functions, like checking the operand types, function for checking usage of undeclared variables and a function for checking re-declaration of a variable.
18. Then started adding these functions in the actions of the appropriate grammar rule.
19. And then we realized that on updating the value of a variable, it's not getting updated in the symbol table. So, we worked on this issue and finally made the changes which were updating the changes made to variables to the symbol table.
20. There was no time, so we couldn't proceed to code generation as we were close to the final submission deadline. So we worked on making the final report, final presentation slides, demo video and presentation video.

5. Language evolution

Our project plan was simple, it started with an idea to ease financial calculations. The original idea of our project was to implement a few financial functions like Simple Interest, Compound interest, SIP maturity, etc. We didn't change our plan much but did not succeed in completing the project.

First we thought of including arrays in our language but left that idea as it became complicated.We completed our lexer and parser but later on the semantic phase we realized that we need to change our grammar for semantic implementation. So, we completely changed our grammar. We got stuck in the semantic phase in finding proper resources for AST, symbol table.

After we found a proper resource, we started working on the symbol table and AST. We completed the symbol table, AST and started working on semantics. We completed semantics. In semantics we implemented type checking, Multiple declarations, Undeclared variables.

We created functions.c and functions.h which contain the inbuilt functions (like Simple Interest, Compound Interest, SIP Maturity) implementations as described in white paper of our language.

We couldn't proceed to code generation as we were close to the final submission deadline. So we worked on making the final report, final presentation slides, demo video and presentation video.

The reasons for not completing the project are lack of proper resources. Moreover, only 50% of the team was actively contributing to the project. So, it was a burden for those who were working on the project.

6.Compiler Architecture

The components of the compiler are:

1. **Lexer**

Lexer converts a sequence of characters into a sequence of tokens.errors like invalid characters,incomplete multi line comments are handled with white spaces getting ignored.

2. **Parser**

Parser takes the tokens produced by lexer and matches with grammar rules.

- **Symbol table :** Symbol table contains identifier name, data type and corresponding value. It is implemented using a struct array.
- **AST (Abstract Syntax Tree) :** AST is a syntax tree. It is a tree representation of the syntactic structure of our program. It is implemented using struct (node) which has left and right nodes.

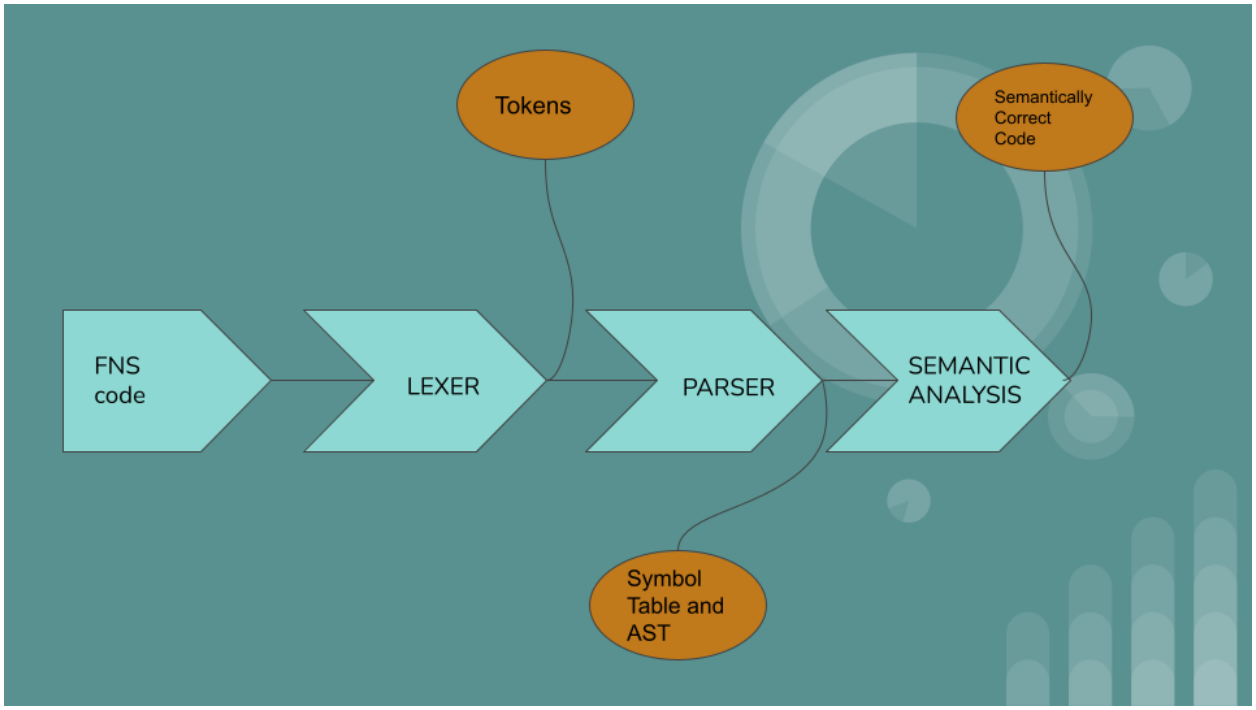
Syntax errors will be shown here.

3. **Semantic**

The semantic check uses AST, symbol table and semantically checks it. Semantic Analysis is the process of drawing meaning from a text, ensuring the declarations and statements of a program are done in this process. Functions of Semantic Analysis are :

- **Type Checking:** Makes sure that each operator has matching operands or in other words ensures that data types are used in a way consistent with their definition.
- **Multiple declarations:** Checks if an identifier is declared more than once.
- **Undeclared variable usage:** Checks if an undeclared variable is being used in the program.

If any of the above semantic errors are found in the program, then they are printed.


Tokens
Semantically Correct Code
FNS code
LEXER
PARSER
SEMANTIC ANALYSIS
Symbol Table and AST

7. Development Environment

7.1 GNU MAKE

Make file is very useful for compiling when there are many files which require a different variety of commands.

• It determines the correct order for updating files.

• If we change a few source files and then run make all, it does not recompile all of your program. It compiles only those files that depend directly or indirectly on the files that we changed.

• It can also regenerate, use, and then delete files which are not needed using -rm clean commands.

7.2 Github

We used Git for pushing our work whenever a part is done so that every other team member can access it and can commit if there were any modifications. Github is a code hosting platform for version control and collaboration which was useful for us to work together on the project .

7.3 VS Code

We used the VSCode platform for editing and compiling the code. VS code features like extensions and the easy to use terminal were very helpful.

8. Test plan and Test suites

Test case 1 (Lexer output):

```
SI : (int principal, int time, int rate) -> (int amount)
{
      amount = principal * time * rate / 100;
}

execute: () -> ()
{
      int p = 10000;
      int t = 2;
      int r = 5;
      int amt;

      SI (p, t, r) -> (amt);
      display("Amount : %d", amt);
}
```

Output

```
Token Type=IDENTIFIER, Line=1, Length=2, Text='SI'
Token Type=COLON_OP, Line=1, Length=1, Text=':'
Token Type=L_PAREN, Line=1, Length=1, Text='('
Token Type=INT, Line=1, Length=3, Text='int'
Token Type=IDENTIFIER, Line=1, Length=9, Text='principal'
Token Type=COMMA_OP, Line=1, Length=1, Text=','
Token Type=INT, Line=1, Length=3, Text='int'
Token Type=IDENTIFIER, Line=1, Length=4, Text='time'
Token Type=COMMA_OP, Line=1, Length=1, Text=','
Token Type=INT, Line=1, Length=3, Text='int'
Token Type=IDENTIFIER, Line=1, Length=4, Text='rate'
Token Type=R_PAREN, Line=1, Length=1, Text=')'
Token Type=ARROW, Line=1, Length=2, Text='->'
Token Type=L_PAREN, Line=1, Length=1, Text='('
Token Type=INT, Line=1, Length=3, Text='int'
Token Type=IDENTIFIER, Line=1, Length=6, Text='amount'
Token Type=R_PAREN, Line=1, Length=1, Text=')'
Token Type=L_BRACE, Line=2, Length=1, Text='{'
Token Type=IDENTIFIER, Line=3, Length=6, Text='amount'
Token Type=ASSIGN, Line=3, Length=1, Text='='
Token Type=IDENTIFIER, Line=3, Length=9, Text='principal'
Token Type=MULTI_OP, Line=3, Length=1, Text='*'
Token Type=IDENTIFIER, Line=3, Length=4, Text='time'
Token Type=MULTI_OP, Line=3, Length=1, Text='*'
Token Type=IDENTIFIER, Line=3, Length=4, Text='rate'
```

```
Token Type=DIV_OP, Line=3, Length=1, Text='/'
Token Type=NUMBER, Line=3, Length=3, Text='100'
Token Type=STM_DELIM, Line=3, Length=1, Text=';'
Token Type=R_BRACE, Line=4, Length=1, Text='}'
Token Type=IDENTIFIER, Line=6, Length=7, Text='execute'
Token Type=COLON_OP, Line=6, Length=1, Text=':'
Token Type=L_PAREN, Line=6, Length=1, Text='('
Token Type=R_PAREN, Line=6, Length=1, Text=')'
Token Type=ARROW, Line=6, Length=2, Text='->'
Token Type=L_PAREN, Line=6, Length=1, Text='('
Token Type=R_PAREN, Line=6, Length=1, Text=')'
Token Type=L_BRACE, Line=7, Length=1, Text='{'
Token Type=INT, Line=8, Length=3, Text='int'
Token Type=IDENTIFIER, Line=8, Length=1, Text='p'
Token Type=ASSIGN, Line=8, Length=1, Text='='
Token Type=NUMBER, Line=8, Length=5, Text='10000'
Token Type=STM_DELIM, Line=8, Length=1, Text=';'
Token Type=INT, Line=9, Length=3, Text='int'
Token Type=IDENTIFIER, Line=9, Length=1, Text='t'
Token Type=ASSIGN, Line=9, Length=1, Text='='
Token Type=NUMBER, Line=9, Length=1, Text='2'
Token Type=STM_DELIM, Line=9, Length=1, Text=';'
Token Type=INT, Line=10, Length=3, Text='int'
Token Type=IDENTIFIER, Line=10, Length=1, Text='r'
Token Type=ASSIGN, Line=10, Length=1, Text='='
Token Type=NUMBER, Line=10, Length=1, Text='5'
Token Type=STM_DELIM, Line=10, Length=1, Text=';'
Token Type=INT, Line=11, Length=3, Text='int'
Token Type=IDENTIFIER, Line=11, Length=3, Text='amt'
Token Type=STM_DELIM, Line=11, Length=1, Text=';'
Token Type=IDENTIFIER, Line=13, Length=2, Text='SI'
Token Type=L_PAREN, Line=13, Length=1, Text='('
Token Type=IDENTIFIER, Line=13, Length=1, Text='p'
Token Type=COMMA_OP, Line=13, Length=1, Text=','
Token Type=IDENTIFIER, Line=13, Length=1, Text='t'
Token Type=COMMA_OP, Line=13, Length=1, Text=','
Token Type=IDENTIFIER, Line=13, Length=1, Text='r'
Token Type=R_PAREN, Line=13, Length=1, Text=')'
Token Type=ARROW, Line=13, Length=2, Text='->'
Token Type=L_PAREN, Line=13, Length=1, Text='('
Token Type=IDENTIFIER, Line=13, Length=3, Text='amt'
Token Type=R_PAREN, Line=13, Length=1, Text=')'
Token Type=STM_DELIM, Line=13, Length=1, Text=';'
Token Type=DISPLAY, Line=14, Length=7, Text='display'
Token Type=L_PAREN, Line=14, Length=1, Text='('
Token Type=STRING_LITERAL, Line=14, Length=13, Text='"Amount : %d"'
Token Type=COMMA_OP, Line=14, Length=1, Text=','
Token Type=IDENTIFIER, Line=14, Length=3, Text='amt'
Token Type=R_PAREN, Line=14, Length=1, Text=')'
```

```
Token Type=STM_DELIM, Line=14, Length=1, Text=';'
Token Type=R_BRACE, Line=15, Length=1, Text='}'
```

Test case 2 (Symbol table output):

```
execute: ()->()
{
	$$This is single line COMMENT

	$/This a
	multiline
	comment /$

	int a = 5;
	date x = 01_12_2022;
	double z = 6.74;
	month w = 01_2022;

	a = 0.1 + 2;
	display("%i\n" , a);

	loop(a < 7)
	{
	if(a == 5)
	{
		a++;
	}
	else {
		break;
	}
	}
	int h;
	int e = h;

}
```

Output

```
SYMBOL      DATATYPE     TYPE        LINENO     INP_PARAM   OP_PARAM       VALUE
-------------------------------------------------------------------------------------
execute     null        function       2          0           0           (null)
-------------------------------------------------------------------------------------
a           int         variable      10          0           0           a+1
-------------------------------------------------------------------------------------
x           date        variable      11          0           0           01_12_2022
-------------------------------------------------------------------------------------
z           double      variable      12          0           0           6.74
-------------------------------------------------------------------------------------
```

```
w           month       variable      13          0           0           01_2022
-------------------------------------------------------------------------------------
display     null        function      16          0           0           (null)
-------------------------------------------------------------------------------------
loop        N/A         keyword       18          0           0           (null)
-------------------------------------------------------------------------------------
if          N/A         keyword       20          0           0           (null)
-------------------------------------------------------------------------------------
else        N/A         keyword       24          0           0           (null)
-------------------------------------------------------------------------------------
break       N/A         keyword       25          0           0           (null)
-------------------------------------------------------------------------------------
h           int         variable      28          0           0           (null)
-------------------------------------------------------------------------------------
e           int         variable      29          0           0           h
-------------------------------------------------------------------------------------
```

Error test case 1 (Syntax error):

```
compound_interest : (double principal, double rateY, double timeY, int n) ->
(double amt)
{
      rateY = rateY/100;
      amt = principal*(1+rateY/n)^(n*timeY);
}

execute : () -> ()
{
      double amount
      compound_interest(10000.0, 2.5, 10.5, 2) -> (amount);
      display("Amount = %d", amount);
}
```

Output:

```
Token Type=IDENTIFIER, Line=1, Length=17, Text='compound_interest'
Token Type=COLON_OP, Line=1, Length=1, Text=':'
Token Type=L_PAREN, Line=1, Length=1, Text='('
Token Type=DOUBLE, Line=1, Length=6, Text='double'
Token Type=IDENTIFIER, Line=1, Length=9, Text='principal'
Token Type=COMMA_OP, Line=1, Length=1, Text=','
Token Type=DOUBLE, Line=1, Length=6, Text='double'
Token Type=IDENTIFIER, Line=1, Length=5, Text='rateY'
Token Type=COMMA_OP, Line=1, Length=1, Text=','
Token Type=DOUBLE, Line=1, Length=6, Text='double'
Token Type=IDENTIFIER, Line=1, Length=5, Text='timeY'
Token Type=COMMA_OP, Line=1, Length=1, Text=','
Token Type=INT, Line=1, Length=3, Text='int'
Token Type=IDENTIFIER, Line=1, Length=1, Text='n'
```

```
Token Type=R_PAREN, Line=1, Length=1, Text=')'
Token Type=ARROW, Line=1, Length=2, Text='->'
Token Type=L_PAREN, Line=1, Length=1, Text='('
Token Type=DOUBLE, Line=1, Length=6, Text='double'
Token Type=IDENTIFIER, Line=1, Length=3, Text='amt'
Token Type=R_PAREN, Line=1, Length=1, Text=')'
Token Type=L_BRACE, Line=2, Length=1, Text='{'
Token Type=IDENTIFIER, Line=3, Length=5, Text='rateY'
Token Type=ASSIGN, Line=3, Length=1, Text='='
Token Type=IDENTIFIER, Line=3, Length=5, Text='rateY'
Token Type=DIV_OP, Line=3, Length=1, Text='/'
Token Type=NUMBER, Line=3, Length=3, Text='100'
Token Type=STM_DELIM, Line=3, Length=1, Text=';'
Token Type=IDENTIFIER, Line=4, Length=3, Text='amt'
Token Type=ASSIGN, Line=4, Length=1, Text='='
Token Type=IDENTIFIER, Line=4, Length=9, Text='principal'
Token Type=MULTI_OP, Line=4, Length=1, Text='*'
Token Type=L_PAREN, Line=4, Length=1, Text='('
Token Type=NUMBER, Line=4, Length=1, Text='1'
Token Type=ADDITION_OP, Line=4, Length=1, Text='+'
Token Type=IDENTIFIER, Line=4, Length=5, Text='rateY'
Token Type=DIV_OP, Line=4, Length=1, Text='/'
Token Type=IDENTIFIER, Line=4, Length=1, Text='n'
Token Type=R_PAREN, Line=4, Length=1, Text=')'
Token Type=POWER, Line=4, Length=1, Text='^'
Token Type=L_PAREN, Line=4, Length=1, Text='('
Token Type=IDENTIFIER, Line=4, Length=1, Text='n'
Token Type=MULTI_OP, Line=4, Length=1, Text='*'
Token Type=IDENTIFIER, Line=4, Length=5, Text='timeY'
Token Type=R_PAREN, Line=4, Length=1, Text=')'
Token Type=STM_DELIM, Line=4, Length=1, Text=';'
Token Type=R_BRACE, Line=5, Length=1, Text='}'
Token Type=IDENTIFIER, Line=7, Length=7, Text='execute'
Token Type=COLON_OP, Line=7, Length=1, Text=':'
Token Type=L_PAREN, Line=7, Length=1, Text='('
Token Type=R_PAREN, Line=7, Length=1, Text=')'
Token Type=ARROW, Line=7, Length=2, Text='->'
Token Type=L_PAREN, Line=7, Length=1, Text='('
Token Type=R_PAREN, Line=7, Length=1, Text=')'
Token Type=L_BRACE, Line=8, Length=1, Text='{'
Token Type=DOUBLE, Line=9, Length=6, Text='double'
Token Type=IDENTIFIER, Line=9, Length=6, Text='amount'
Token Type=IDENTIFIER, Line=10, Length=17, Text='compound_interest'
Parsing failed here.
Syntax Error!
```

Error test case 2 (Multiple declarations and undeclared variables):

```
$$ Multiple Declarations and un-declared variables

execute : () -> ()
{
	int a1 = 10;
	double b1 = 4.56;
	string s1 = "string";
	date d1 = 11_11_2011;
	month m1 = 09_2022;

	loop(a1--)
	{
	$$ Error here (Multiple declarations of b1)
	double b1 = 0.5^2;
	}

	$$ Error here (Using undeclared variable)
	e1 = c1++;

}
```

Output:

```
4 errors found in semantic analysis.
Errors are:
	Line 14: Multiple declarations of "b1"
	Line 18: Variable "e1" Using undeclared variable...
	Line 18: Variable "c1" Using undeclared variable...
	Line 18: Operands of different types (null, int)
```

Error test case 3 (Type checking):

```
$$ Type checking

execute : () -> ()
{
	int a = 5;
	date d = 01_11_2022;

	$$ Error here (addition of int and date datatype is wrong)
	int b = a + d;
	display("b = %d", b);

	double c = 6.7;

	$$ No error here (multiplication of int and double is possible)
```

```
	c = c * a;

	string s = "hello";
	$$ Error here (increment of string is not possible)
	s++;

	$$ No error here (decrement of double and increment of int is possible)
	c--;
	++a;

	month m = 11_2022;

	$$ Error here (comparision of int and month is wrong)
	if(a > m)
	{
	display("Incompatible comparision");
	}

	loop (a & b)
	{
	$$ Error here (power of a string is not possible)
	string s1 = s ^ a;
	}

	$$ No error here (multiplication, division, and addition of int and
double is possible)
	int e = ((3*6) + 0.5)/5;

	double f = -c;
	double g = -s;
}
```

Output:

```
6 errors found in semantic analysis.
Errors are:
	Line 9: Operands of different types (int, date)
	Line 19: Operands of different types (string, int)
	Line 28: Operands of different types (int, month)
	Line 36: Operands of different types (string, int)
	Line 43: Operands of different types (string, int)
	Line 43: Operands of different types (double, string)
```

Error test case 4 (Reserved keyword):

```
$$ Reserved Keywords Usage
```

```
simpleIntrst : (int a) -> (int b)
{
      accept("%d", a);
}

execute : () -> ()
{
      $$ Error here (using a keyword as variable name)
      double SIPmaturity, compoundIntrst;
}
```

Output:

```
3 errors found in semantic analysis.
Errors are:
      Line 3: Function used here "simpleIntrst" is a keyword.
      Line 11: Variable used here "SIPmaturity" is a keyword.
      Line 11: Variable used here "compoundIntrst" is a keyword.
```

Error test case 5 (Function Semantics):

```
$$ Noumber of arguments in function is not matching

SI : (int principal, int time, int rate) -> (int amount)
{
      amount = principal * time * rate / 100;
}

execute: () -> ()
{
      int p = 10000;
      int t = 2;
      int r = 5;
      int amt;

      SI (p, t) -> ();
      display("Amount : %d", amt);
}
```

Output:

```
2 errors found in semantic analysis.
Errors are:
      Line 15: Number of input parameters for "SI" do not match. Expected: 3,
Given: 2
      Line 15: Number of output parameters for "SI" do not match. Expected: 1,
Given: 0
```

9. Conclusions and lessons learnt

We got a good idea on the in depth view of working of compilers. We understood how the lexer, parser and semantic checking of a compiler work hand-in-hand. We learnt how to use git and github which played a very important role as it is a group project. It helped in collaborating with the team members.

We understood the importance of testing the written code with the help of test cases. It helped us in identifying and rectifying the mistakes made in our code. We understood the importance of makefile. Without makefile, we had to run multiple commands to compile and test which was cumbersome. With make file, it was just a single command to compile our code.

10. Appendix

10.1 Lexer code (lexer.l)

```
%{
      #include <stdio.h>
      #include <stdlib.h>
      #include "parser.tab.h"

      void print_tokens(char* c);
      static void mult_line_check(void);

      int line_count = 1;

%}

D                       [0-9]
L                       [a-zA-Z_]
H                       [a-fA-F0-9]
E                       [Ee][+-]?{D}+
%%


"break"                 { strcpy(yylval.ast_node.name,(yytext));
print_tokens("BREAK")           ; return (BREAK);}
"continue"              { strcpy(yylval.ast_node.name,(yytext));
print_tokens("CONTINUE")        ; return (CONTINUE);}
"accept"                        { strcpy(yylval.ast_node.name,(yytext));
print_tokens("ACCEPT")          ; return (ACCEPT);}
"double"                { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DOUBLE")          ; return (DOUBLE);}
"string"                        { strcpy(yylval.ast_node.name,(yytext));
print_tokens("STRING")          ; return (STRING);}
"else"                  { strcpy(yylval.ast_node.name,(yytext));
print_tokens("ELSE")            ; return (ELSE);}
"if"                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("IF")              ; return (IF);}
"loop"                   { strcpy(yylval.ast_node.name,(yytext));
print_tokens("LOOP")            ; return (LOOP);}
"fbreak"                        { strcpy(yylval.ast_node.name,(yytext));
print_tokens("FUNCTION_BREAK") ; return (FUNCTION_BREAK);}
"date"                          { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DATE")                  ; return (DATE);}
"month"                               { strcpy(yylval.ast_node.name,(yytext));
print_tokens("MONTH")                 ; return (MONTH);}
"int"                   { strcpy(yylval.ast_node.name,(yytext));
print_tokens("INT")             ; return (INT);}
```

```
"+="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("ADD_ASSIGN")     ; return (ADD_ASSIGN);}
"-="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("SUB_ASSIGN")     ; return (SUB_ASSIGN);}
"*="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("MUL_ASSIGN")     ; return (MUL_ASSIGN);}
"/="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DIV_ASSIGN")     ; return (DIV_ASSIGN);}
"%="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("MOD_ASSIGN")     ; return (MOD_ASSIGN);}
"++"                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("INC_OP")         ; return (INC_OP);}
"--"                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DEC_OP")         ; return (DEC_OP);}
"&"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("AND_OP")         ; return (AND_OP);}
"|"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("OR_OP")          ; return (OR_OP);}
"<="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("LE_OP")          ; return (LE_OP);}
">="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("GE_OP")          ; return (GE_OP);}
"=="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("EQ_OP")          ; return (EQ_OP);}
"!="                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("NE_OP")          ; return (NE_OP);}
";"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("STM_DELIM")      ; return (STM_DELIM);}
"{"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("L_BRACE")        ; return (L_BRACE);}
"}"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("R_BRACE")        ; return (R_BRACE);}
":"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("COLON_OP")       ; return (COLON_OP);}
"="                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("ASSIGN")         ; return (ASSIGN);}
"("                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("L_PAREN")        ; return (L_PAREN);}
")"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("R_PAREN")        ; return (R_PAREN);}
"["                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("L_BRAKET")       ; return (L_BRAKET);}
"]"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("R_BRAKET")       ; return (R_BRAKET);}
","                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("COMMA_OP")       ; return (COMMA_OP);}
"!"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("NOT_OP")         ; return (NOT_OP);}
```

```
"-"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("SUBTRACT_OP")    ; return (SUBTRACT_OP);}
"+"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("ADDITION_OP")    ; return (ADDITION_OP);}
"*"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("MULTI_OP")       ; return (MULTI_OP);}
"/"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DIV_OP")         ; return (DIV_OP);}
"%"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("MOD_OP")         ; return (MOD_OP);}
"<"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("LESSER_OP")      ; return (LESSER_OP);}
">"                     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("GREATER_OP")     ; return (GREATER_OP);}
"^"                             { strcpy(yylval.ast_node.name,(yytext));
print_tokens("POWER")           ; return (POWER);}
"->"                           { strcpy(yylval.ast_node.name,(yytext));
print_tokens("ARROW")           ; return (ARROW);}
"display"                      { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DISPLAY")        ; return (DISPLAY);}
{D}{D}"_"{D}{D}"_"{D}{D}{D}{D}       { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DATE")    ; return (DATE);}
{D}{D}"_"{D}{D}{D}{D}          { strcpy(yylval.ast_node.name,(yytext));
print_tokens("MONTH")   ; return (MONTH);}

"$$".*            { print_tokens("COMMENT"); }

"$/"                           { /*printf("Multi-line comment starts
here\n");*/ mult_line_check();}

L?\"(\\.|[^\\"])*\"     { strcpy(yylval.ast_node.name,(yytext));
print_tokens("STRING_LITERAL"); return (STRING_LITERAL);} // yylval.str =
strdup(yytext);
{L}({L}|{D}|"_")*          { strcpy(yylval.ast_node.name,(yytext));
print_tokens("IDENTIFIER"); return (IDENTIFIER);}
{D}+                    { strcpy(yylval.ast_node.name,(yytext));
print_tokens("NUMBER"); return (NUMBER);}  // sscanf(yytext, "%d",
&yylval.ival);
{D}*"."{D}+({E})?       { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DOUBLE_CONST"); return (DOUBLE_CONST);}
{D}+"."{D}*({E})?       { strcpy(yylval.ast_node.name,(yytext));
print_tokens("DOUBLE_CONST"); return (DOUBLE_CONST);}

[ \t\v]                 { /*eat_up_white_spaces();*/ }
[\n]                    { line_count++;}
.                  { printf("ERROR: Invalid character %s at line number
%d\n",yytext,line_count); return (ERROR); }

%%
```

```
int column = 0;

static void mult_line_check(void)
{
	int c;
	int line = line_count+1;

	while(1)
	{
		int loop = 0;
		switch(input())
		{
			case '\0':
				printf("ERROR: Unterminated comment at line %d\n",
line);
				exit(-1);
				loop = -1;
				break;

			case '/':
				if((c = input()) == '$')
				{
					loop = -1;
					printf("Multi-line comments terminated.\n");
					break;
				}
				unput(c);
				break;

			case '\n':
				line_count++;

			default:
				break;
		}

		if(loop == -1)
		{
			break;
		}
	}
}

void print_tokens(char* c)
{
	printf("Token Type=%s, Line=%d, Length=%d, Text='%s'\n", c, line_count,
yyleng, yytext);
}
```

```
int yywrap() {
      return 1;
}
```

10.2 Parser code (parser.y)

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <ctype.h>
    #include "lex.yy.c"
    #include "symbolTable.h"
    #include "ast.h"
    #include "semantics.h"
    #include "symbolTable.c"
    #include "ast.c"
    #include "semantics.c"


    void yyerror();
    int yylex();

    extern int line_count;
%}


%token
  ASSIGN  '='
  SUBTRACT_OP   '-'
  ADDITION_OP    '+'
  MULTI_OP    '*'
  DIV_OP   '/'
  AND_OP '&'
  L_PAREN  '('
  R_PAREN  ')'
  MOD_OP '%'
  OR_OP '|'
  COMMA_OP ','
  STM_DELIM ';'
  COLON_OP ':'
  L_BRACE '{'
  R_BRACE '}'
  L_BRAKET '['
  R_BRAKET ']'
  INC_OP "++"
```

```
  DEC_OP "--"
  ADD_ASSIGN "+="
  SUB_ASSIGN "-="
  MUL_ASSIGN "*="
  DIV_ASSIGN "/="
  MOD_ASSIGN "%="
  LESSER_OP "<"
  GREATER_OP ">"
  NE_OP "!="
  EQ_OP "=="
  GE_OP ">="
  LE_OP "<="
  ARROW "->"
  BREAK "break"
  CONTINUE "continue"
  FUNCTION_BREAK "fbreak"
  IF "if"
  ELSE "else"
  LOOP "loop"
;

%token
  INT "int"
  DOUBLE "double"
  CHAR "char"
  STRING "string"
  DATE "date"
  MONTH "month"
  ARRAY "array"
;

%token
  IDENTIFIER
  NUMBER
  DOUBLE_CONST
  STRING_LITERAL
  NOT_OP
  POWER
  ERROR
  DISPLAY
  ACCEPT
;


/* Operator precedence */
%left  COMMA_OP
%right '?' COLON_OP ASSIGN ADD_ASSIGN SUB_ASSIGN
%left  OR_OP
%left  AND_OP
```

```
%left  EQ_OP NE_OP
%left  LESSER_OP GREATER_OP GE_OP LE_OP
%left  ADDITION_OP SUBTRACT_OP
%left  MULTI_OP DIV_OP MOD_OP POWER
%right INC_OP DEC_OP
%left  L_PAREN L_BRAKET

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%union {
    struct var1 {
        char name[100];
        struct node* nd;
    } ast_node;

    struct var2 {
        char name[100];
        struct node* nd;
        char type[30];
    } sem_node;
}


%type<ast_node> ASSIGN SUBTRACT_OP ADDITION_OP MULTI_OP DIV_OP AND_OP L_PAREN
R_PAREN
MOD_OP OR_OP COMMA_OP STM_DELIM COLON_OP L_BRACE R_BRACE L_BRAKET R_BRAKET
INC_OP
DEC_OP ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN LESSER_OP
GREATER_OP
NE_OP EQ_OP GE_OP LE_OP ARROW BREAK CONTINUE FUNCTION_BREAK IF ELSE LOOP INT
DOUBLE
STRING DATE MONTH IDENTIFIER NUMBER DOUBLE_CONST STRING_LITERAL NOT_OP POWER
ERROR
DISPLAY ACCEPT

%type<ast_node> declarations declaration function vardec_stmt paramdecls
compound_stmt
paramdecl typename print_list print_stmt stmt selection_stmt jump_stmt
expression_stmt
empty_stmt iteration_stmt program else_stmt inp_stmt inp_list function_call
par_list

%type <sem_node> expr exprs initializer initializer_list

%%
%start program;

program: declarations {
```

```
    $$.nd = make_node($1.nd, NULL, "program");
    head = $$.nd;
};

declarations: declarations declaration { $$.nd = make_node($1.nd, $2.nd,
"decls"); }
|             %empty { $$.nd = make_node(NULL, NULL, "null"); }
;

declaration: function { $$.nd = make_node($1.nd, NULL, "decl"); }
|            vardec_stmt STM_DELIM   {$$.nd = make_node($1.nd, NULL, "decl"); }
;

function: IDENTIFIER {add('F');} COLON_OP L_PAREN paramdecls {
symbol_table[search_id($1.name)].param_cnt_ip = param_ip;
    param_ip = 0;} R_PAREN "->" L_PAREN paramdecls {
symbol_table[search_id($1.name)].param_cnt_op = param_ip; param_ip = 0;}
R_PAREN compound_stmt R_BRACE
{
    $5.nd->token = "inp_param";
    $10.nd->token = "out_param";
    node* temp = make_node($5.nd, $10.nd, "params");
    $$.nd = make_node(temp,$13.nd , $1.name);
}
;

paramdecls: paramdecl  {$$.nd = make_node($1.nd, NULL, "par_decls"); }
|           %empty { $$.nd = make_node(NULL, NULL, "null"); }
;
paramdecl:  paramdecl COMMA_OP typename IDENTIFIER {
    add('V');
    param_ip++;
    $4.nd = make_node(NULL, NULL, yytext);
    node* temp = make_node($3.nd, $4.nd, "param");
    $$.nd = make_node($1.nd, temp, "par_decl");
}
|           typename IDENTIFIER {
    add('V');
    param_ip++;
    $2.nd = make_node(NULL, NULL, yytext);
    $$.nd = make_node($1.nd, $2.nd, "par_decl");
}
;

typename: INT   { insert_type("int");}   {$$.nd = make_node(NULL, NULL,
"int");}
|         DOUBLE  { insert_type("double");}   {$$.nd = make_node(NULL, NULL,
"double");}
```

```
|         STRING  { insert_type("string");}   {$$.nd = make_node(NULL, NULL,
"string");}
|         DATE  { insert_type("date");}   {$$.nd = make_node(NULL, NULL,
"date");}
|         MONTH   { insert_type("month");}   {$$.nd = make_node(NULL, NULL,
"month");}
;

print_list: %empty  { $$.nd = make_node(NULL, NULL, "null"); }
|           print_list COMMA_OP exprs  {$$.nd = make_node($1.nd, $3.nd,
"print_ls"); }
;

print_stmt: DISPLAY {add('F');} L_PAREN STRING_LITERAL print_list R_PAREN
STM_DELIM {
    $4.nd = make_node(NULL, NULL, "str");
    $$.nd = make_node($4.nd, $5.nd, "print");
}
;

inp_list: %empty  { $$.nd = make_node(NULL, NULL, "null"); }
|           inp_list COMMA_OP exprs  {$$.nd = make_node($1.nd, $3.nd,
"inp_ls"); }
;

inp_stmt: ACCEPT {add('F');} L_PAREN STRING_LITERAL inp_list R_PAREN STM_DELIM
{
    $4.nd = make_node(NULL, NULL, "str");
    $$.nd = make_node($4.nd, $5.nd, "input");
}
;

function_call: IDENTIFIER {check_decls($1.name);} L_PAREN par_list {
check_function($1.name, param_ip, 0); param_ip = 0; } R_PAREN "->" L_PAREN
par_list { check_function($1.name, param_ip, 1); param_ip = 0; } R_PAREN {
    $$.nd = make_node($4.nd, $9.nd, $1.name);
}
;

par_list: %empty  { $$.nd = make_node(NULL, NULL, "null"); }
|           par_list COMMA_OP expr  { $$.nd = make_node($1.nd, $3.nd,
"par_ls"); param_ip++; }
|           expr       { $$.nd = make_node($1.nd, NULL, "par_ls"); param_ip++;
}
;

stmt: compound_stmt R_BRACE { $$.nd = make_node($1.nd, NULL, "comp_stmt"); }
|     selection_stmt { $$.nd = make_node($1.nd, NULL, "sel_stmt"); }
|     jump_stmt  { $$.nd = make_node($1.nd, NULL, "j_stmt"); }
```

```
|     expression_stmt  { $$.nd = make_node($1.nd, NULL, "exp_stmt"); }
|     empty_stmt   {$$.nd = make_node($1.nd, NULL, "emp_stmt"); }
|     vardec_stmt STM_DELIM { $$.nd = make_node($1.nd, NULL, "vdec_stmt"); }
|     iteration_stmt  { $$.nd = make_node($1.nd, NULL, "it_stmt"); }
|     print_stmt  { $$.nd = make_node($1.nd, NULL, "pr_stmt"); }
|     inp_stmt  { $$.nd = make_node($1.nd, NULL, "inp_stmt"); }
|     function_call { $$.nd = make_node($1.nd, NULL, "func_call"); }
;

expression_stmt: exprs STM_DELIM { $$.nd = make_node($1.nd, NULL,"expr_stm"); }
;

jump_stmt: CONTINUE {add('K');} STM_DELIM  { $$.nd = make_node(NULL, NULL,
"continue"); }
|          BREAK {add('K');} STM_DELIM     { $$.nd = make_node(NULL, NULL,
"break"); }
|          FUNCTION_BREAK {add('K');} STM_DELIM      { $$.nd = make_node(NULL,
NULL, "f_break"); }
;

empty_stmt: STM_DELIM {$$.nd = make_node(NULL,NULL, "NULL");}
;


vardec_stmt: typename IDENTIFIER { yytext = $2.name; add('V'); $2.nd =
make_node(NULL, NULL, $2.name);} ASSIGN initializer {
    node* temp = make_node($1.nd, $2.nd, "type_id");
    $$.nd = make_node(temp, $5.nd, "=");
    check_type($1.name, $5.type);
    int x=search_id($2.name);
    if(x!=-1)
    {
        add_value(x,$5.name);
    }
}
|            typename IDENTIFIER { yytext = $2.name; add('V'); $2.nd =
make_node(NULL, NULL, $2.name);
    $$.nd = make_node($1.nd, $2.nd, "vardecl");
}
|            vardec_stmt COMMA_OP IDENTIFIER { yytext = $3.name; add('V');
$3.nd = make_node(NULL, NULL, $3.name);} ASSIGN initializer {
    node* temp = make_node($3.nd, $6.nd, "=");
    $$.nd = make_node($1.nd, temp, "vardecl_st");
    check_type(search_type($3.name), $6.type);
     int x=search_id($3.name);
    if(x!=-1)
    {
        add_value(x,$6.name);
    }
```

```
}
|            vardec_stmt COMMA_OP IDENTIFIER {
    yytext = $3.name;
    add('V');
    $3.nd = make_node(NULL, NULL, $3.name);
    $$.nd = make_node($1.nd, $3.nd, "vardecl_st");
}
;

initializer: expr   { $$.nd = make_node($1.nd, NULL, "init"); }
|            L_BRACE initializer_list R_BRACE   { $$.nd = make_node($2.nd,
NULL, "init"); }
;

initializer_list: initializer   { $$.nd = make_node($1.nd, NULL, "init_list");
}
|                 initializer_list COMMA_OP initializer { $$.nd =
make_node($1.nd, $3.nd, "init_list"); }
;

compound_stmt:  L_BRACE
|               compound_stmt stmt  { $$.nd = make_node($1.nd, $2.nd, "stmts");
}
;

else_stmt: ELSE {add('K');} stmt   { $$.nd = make_node($3.nd, NULL, "else"); }
|          %empty               { $$.nd = make_node(NULL, NULL, "null"); }
;

selection_stmt : IF {add('K');} L_PAREN expr R_PAREN stmt else_stmt  {
    node* temp = make_node($4.nd, $6.nd, "if");
    $$.nd = make_node(temp, $7.nd, "select_stm");
}
;

iteration_stmt: LOOP {add('K');} L_PAREN expr R_PAREN stmt       {
    $$.nd = make_node($4.nd, $6.nd, "loop");
}
;


exprs: expr     { $$.nd = make_node($1.nd, NULL, "expr" );
strcpy($$.type,$1.type); strcpy($$.name,$1.name);}
|      exprs COMMA_OP expr      { $$.nd = make_node($1.nd, $3.nd, "expr"); }
;
```

```
expr: IDENTIFIER              { strcpy(value, yytext); char* id_type =
search_type($1.name); strcpy($$.type, id_type); check_decls($1.name); $$.nd =
make_node(NULL, NULL, yytext); }
|     NUMBER                  { strcpy(value, yytext); $$.nd = make_node(NULL,
NULL, yytext); strcpy($$.type, "int"); }
|     DOUBLE_CONST            { strcpy(value, yytext); $$.nd = make_node(NULL,
NULL, yytext); strcpy($$.type, "double");}
|     STRING_LITERAL          { strcpy(value, yytext); $$.nd = make_node(NULL,
NULL, yytext); strcpy($$.type , "string"); }
|     DATE                    { strcpy(value, yytext); $$.nd = make_node(NULL,
NULL, yytext); strcpy($$.type, "date"); }
|     MONTH                   { strcpy(value, yytext); $$.nd = make_node(NULL,
NULL, yytext); strcpy($$.type, "month"); }
|     L_PAREN exprs R_PAREN        { $$.nd = make_node($2.nd, NULL, "exprs");
strcpy($$.type,$2.type);char* temp=strcat($1.name,$2.name);
strcpy($$.name,strcat(temp,$3.name));}
|     expr ASSIGN expr             { $$.nd = make_node($1.nd, $3.nd, "=");
check_type($1.type, $3.type); int x=search_id($1.name);  if(x!=-1){
add_value(x,$3.name); }}
|     expr ADDITION_OP expr        { $$.nd = make_node($1.nd, $3.nd, "+");
check_type($1.type, $3.type);char* temp=strcat($1.name,"+");
strcpy($$.name,strcat(temp,$3.name));}
|     expr SUBTRACT_OP expr %prec ADDITION_OP   { $$.nd = make_node($1.nd,
$3.nd, "-"); check_type($1.type, $3.type); char* temp=strcat($1.name,"-");
strcpy($$.name,strcat(temp,$3.name)); }
|     expr MULTI_OP expr                { $$.nd = make_node($1.nd, $3.nd, "*");
check_type($1.type, $3.type); char* temp=strcat($1.name,"*");
strcpy($$.name,strcat(temp,$3.name));}
|     expr DIV_OP expr %prec MULTI_OP   { $$.nd = make_node($1.nd, $3.nd, "/");
check_type($1.type, $3.type); char* temp=strcat($1.name,"/");
strcpy($$.name,strcat(temp,$3.name));}
|     expr MOD_OP expr          { $$.nd = make_node($1.nd, $3.nd, "%");
check_type($1.type, $3.type); char* temp=strcat($1.name,"%");
strcpy($$.name,strcat(temp,$3.name));}
|     expr "+=" expr            { $$.nd = make_node($1.nd, $3.nd, "+=");
check_type($1.type, $3.type);int x=search_id($1.name); char*
temp=strcat($1.name,"+"); strcpy($$.name,strcat(temp,$3.name)); if(x!=-1){
add_value(x,$$.name); }}
|     expr "-=" expr            { $$.nd = make_node($1.nd, $3.nd, "-=");
check_type($1.type, $3.type);int x=search_id($1.name); char*
temp=strcat($1.name,"-"); strcpy($$.name,strcat(temp,$3.name)); if(x!=-1){
add_value(x,$$.name); }}
|     "++" expr                 { $$.nd = make_node( NULL,$2.nd, "++");
check_type($2.type, "int") ; int x=search_id($2.name); char*
temp=strcat($2.name,"+"); strcpy($$.name,strcat(temp,"1")); if(x!=-1){
add_value(x,$$.name); }}
|     "--" expr %prec INC_OP    { $$.nd = make_node(NULL, $2.nd, "--");
check_type($2.type, "int") ; int x=search_id($2.name);   char*
```

```
temp=strcat($2.name,"-"); strcpy($$.name,strcat(temp,"1"));if(x!=-1){
add_value(x,$$.name); }}
|     expr "++"                 { $$.nd = make_node($1.nd, NULL, "++");
check_type($1.type, "int") ; int x=search_id($1.name); char*
temp=strcat($1.name,"+"); strcpy($$.name,strcat(temp,"1")); if(x!=-1){
add_value(x,$$.name);} }
|     expr "--" %prec INC_OP    { $$.nd = make_node($1.nd, NULL, "--");
check_type($1.type, "int") ; int x=search_id($1.name); char*
temp=strcat($1.name,"-"); strcpy($$.name,strcat(temp,"1"));  if(x!=-1){
add_value(x,$$.name); }}
|     expr LESSER_OP expr       { $$.nd = make_node($1.nd, $3.nd, "<");
check_type($1.type, $3.type); }
|     expr GREATER_OP expr      { $$.nd = make_node($1.nd, $3.nd, ">");
check_type($1.type, $3.type); }
|     expr GE_OP expr           { $$.nd = make_node($1.nd, $3.nd, ">=");
check_type($1.type, $3.type);}
|     expr LE_OP expr           { $$.nd = make_node($1.nd, $3.nd, "<=");
check_type($1.type, $3.type);}
|     expr OR_OP expr           { $$.nd = make_node($1.nd, $3.nd, "|");
check_type($1.type, $3.type); }
|     expr AND_OP expr          { $$.nd = make_node($1.nd, $3.nd, "&");
check_type($1.type, $3.type); }
|     expr EQ_OP expr           { $$.nd = make_node($1.nd, $3.nd, "==");
check_type($1.type, $3.type); }
|     expr NE_OP expr %prec EQ_OP    { $$.nd = make_node($1.nd, $3.nd, "!=");
check_type($1.type, $3.type); }
|     expr POWER expr                { $$.nd = make_node($1.nd, $3.nd, "^");
check_type($1.type, $3.type); char* temp=strcat($1.name,"^");
strcpy($$.name,strcat(temp,$3.name));}
|     SUBTRACT_OP expr               { $$.nd = make_node(NULL, $2.nd, "neg");
strcpy($$.type, $2.type); printf("%s\n", $2.type); check_type($2.type, "int")
;}
|     NOT_OP expr                    { $$.nd = make_node(NULL, $2.nd, "not"); }
;

%%


int main() {
      yyparse();
    printf("\n\n");
    printf("\nSYMBOL          DATATYPE         TYPE                  LINENO
INP_PARAM       OP_PARAM           VALUE \n");

printf("--------------------------------------------------------------------------
-------------------------------------------\n");

    for(int i = 0; i < count; i++)
    {
```

```
        printf("%s\t\t%s\t\t%s\t\t%d\t\t%d\t\t%d\t\t%s\t\n",
symbol_table[i].id_name, symbol_table[i].data_type, symbol_table[i].type,
symbol_table[i].line_no, symbol_table[i].param_cnt_ip,
symbol_table[i].param_cnt_op, symbol_table[i].value);

printf("--------------------------------------------------------------------------
-----------------------------------------\n");
    }

    for(int i = 0; i < count; i++)
    {
        free(symbol_table[i].id_name);
        free(symbol_table[i].type);
    }

    printf("\n\n\n\n");
    levelTree(head, 0);
    printf("\n\n\n\n");

    if(semantic_err > 0)
    {
        printf("%d errors found in semantic analysis.\nErrors are:\n",
semantic_err);
        for(int i = 0; i < semantic_err; i++)
        {
            printf("\t %s", error_list[i]);
        }
    }
    else{
        printf("No errors found in semantic analysis.\n");
    }
}


void yyerror () {
      fprintf(stderr, "Parsing failed here.\nSyntax Error!\n");
}
```

10.3 functions.h

```
void simpleIntrst (double principle, double rateY, double timeY, double
*amount);
void compoundIntrst (double principle, double rateY, double timeY, int n,
double *amount);
void SIPmaturity (double mnthlyInv, double growthRateY, int months, double
*maturity);
void SIPmaturityDeets (double mnthlyInv, double growthRateY, int months, double
*maturity, double *inv, double *intrst, double *returnPerc);
```

10.4 function.c

```
#include "functions.h"
#include <stdlib.h>
#include <math.h>

void simpleIntrst (double principle, double rateY, double timeY, double
*amount)
{
    *amount = principle * rateY * timeY / 100;
}

void compoundIntrst (double principle, double rateY, double timeY, int n,
double *amount)
{
    rateY = rateY/100;
    *amount = principle * pow((1 + rateY/n),(n*timeY));
}

void SIPmaturity (double mnthlyInv, double growthRateY, int months, double
*maturity)
{
    double i = growthRateY/12/100;
    *maturity = mnthlyInv * (pow(1+i, months)-1) * (1+i)/i;
}

void SIPmaturityDeets (double mnthlyInv, double growthRateY, int months, double
*maturity, double *inv, double *intrst, double *returnPerc)
{
    SIPmaturity(mnthlyInv, growthRateY, months, maturity);
    *inv = mnthlyInv * months;
    *intrst = *maturity - *inv;
    *returnPerc = *intrst/ (*inv) * 100;
}
```

10.5 symbolTable.h

```
struct dataType {
    char * id_name;
    char * data_type;
    char * type;
    char * value ;
    int line_no;
    int param_cnt_ip;
    int param_cnt_op;
} symbol_table[200];

int count = 0;
int q;
char type[30];
char value[70];
int param_ip = 0;
int param_op = 0;


void insert_type(char*);
void add_value(int , char*);
void add(char);
int search(char*);
int search_id(char*);
```

10.6 symbolTable.c

```
void add(char c)
{
    if(c == 'V' || c == 'F')
    {
        for(int i = 0; i < 17; i++)
        {
            if( !strcmp(reserved[i], strdup(yytext)))
            {   if(c == 'F')
                {
                    if(!strcmp(yytext, "display") || !strcmp(yytext, "execute")
|| !strcmp(yytext, "accept"))
                    {
                        break;
                    }
                }
                if(c == 'V')
                    sprintf(error_list[semantic_err], "Line %d: Variable used
here \"%s\" is a keyword.\n", line_count, yytext);
                else {
                    sprintf(error_list[semantic_err], "Line %d: Function used
here \"%s\" is a keyword.\n", line_count, yytext);
```

```
                }
                semantic_err++;
                return;
            }
        }
    }

    q = search(yytext);
    if(!q)
    {
        if(c == 'K')
        {
            symbol_table[count].id_name = strdup(yytext);
            symbol_table[count].data_type = strdup("N/A");
            symbol_table[count].line_no = line_count;
            symbol_table[count].type = strdup("keyword\t");
            symbol_table[count].value = 0;
            count++;
        }
        else if(c == 'V')
        {
            symbol_table[count].id_name = strdup(yytext);
            symbol_table[count].data_type = strdup(type);
            symbol_table[count].line_no = line_count;
            symbol_table[count].type = strdup("variable");
            symbol_table[count].value = 0;
            count++;
        }
        else if(c == 'F')
        {
            symbol_table[count].id_name = strdup(yytext);
            symbol_table[count].data_type = strdup("null");
            symbol_table[count].line_no = line_count;
            symbol_table[count].type = strdup("function");
            symbol_table[count].value = 0;
            count++;
        }
    }
    else if((c == 'V' || c == 'F') && q)
    {
        sprintf(error_list[semantic_err], "Line %d: Multiple declarations of
\"%s\" \n", line_count, yytext);
        semantic_err++;
    }
}

int search(char *temp)
{
    int i;
```

```
    for(i = count-1; i >= 0; i--)
    {
        if(strcmp(symbol_table[i].id_name, temp) == 0)
        {
            return -1;
        }
    }
    return 0;
}

void insert_type(char *s)
{
    strcpy(type, s);
}

void add_value(int count1, char *value)
{
    symbol_table[count1].value = strdup(value);
}

int search_id(char* name)
{
    for(int i = 0; i < count; i++)
    {
        if(strcmp(symbol_table[i].id_name, name) == 0)
        {
            return i;
        }
    }
    return -1;
}
```

10.7 ast.h

```
typedef struct node {
    struct node *left;
    struct node* right;
    char *token;
}node;

node* head;

void printInorder(node *);
void levelTree(node*, int);
node* make_node(node *left, node *right, char *token);
```

10.8 ast.c

```
node* make_node(node* left, node* right, char *token)
{
    node *newNode = (node*) malloc (sizeof(node));
    char* newToken = (char*) malloc(strlen(token)+1);
    strcpy(newToken, token);
    newNode->left = left;
    newNode->right = right;
    newNode->token = newToken;

    return newNode;
}

void printInorder(node *tree) {
      int i;
      if (tree->left) {
            printInorder(tree->left);
      }
      printf("%s, ", tree->token);
      if (tree->right) {
            printInorder(tree->right);
      }
}

void levelTree(node* root, int space)
{
    if(root == NULL){
        return;
    }
    space += 10;
    levelTree(root->right, space);
    for(int i = 10; i < space; i++)
    {
```

```
        printf(" ");
    }
    printf("%s\n", root->token);
    levelTree(root->left, space);

}
```

10.9 semantics.h

```
int label = 0;
char buff[100];

char error_list[30][100];
char reserved[17][30] = {"int", "double", "char", "if", "else", "loop",
"execute", "break", "fbreak", "continue",
    "date", "month", "display", "SIPmaturity", "simpleIntrst",
"compoundIntrst", "SIPmaturityDeets"};
int semantic_err = 0;

void check_decls(char*);
int check_type(char*, char*);
char* search_type(char*);
void check_function(char*, int, int);
```

10.10 semantics.c

```
// checks if undeclared variable is being used in the program
void check_decls(char* c)
{
    q = search(c);
    if(q == 0)
    {
        sprintf(error_list[semantic_err], "Line %d: Variable \"%s\" Using
undeclared variable...\n", line_count, c);
        semantic_err++;
    }
}
```

```
// checks whether the data type of operands match or not
int check_type(char* a, char* b)
{
    int result;
    // if datatypes not compatible result = -2

    if( !strcmp(a, b)) result = 0;
    else if( !strcmp(a, "int") && !strcmp(b, "double"))
    {
        result = 1;
    }
    else if( !strcmp(a, "double") && !strcmp(b, "int"))
    {
        result = 2;
    }
    else
    {
        result = -2;
        sprintf(error_list[semantic_err], "Line %d: Operands of different types
(%s, %s)\n", line_count, a, b);
        semantic_err++;
    }
    return result;
}

//  checks if a function call's number of input and output parameters match
with the function decl or not.
void check_function(char* name, int num, int type)
{
    if(type == 0) //input
    {
        int c = symbol_table[search_id(name)].param_cnt_ip;
        if(c != num)
        {
            sprintf(error_list[semantic_err], "Line %d: Number of input
parameters for \"%s\" do not match. Expected: %d, Given: %d\n", line_count,
name, c, num);
            semantic_err++;
        }
    }
    else { //output
        int c = symbol_table[search_id(name)].param_cnt_op;
        if(c != num)
        {
            sprintf(error_list[semantic_err], "Line %d: Number of output
parameters for \"%s\" do not match. Expected: %d, Given: %d\n", line_count,
name, c, num);
            semantic_err++;
        }
```

```
    }
}

char* search_type(char* name)
{
    for(int i = 0; i < count; i++)
    {
        if(strcmp(symbol_table[i].id_name, name) == 0)
        {
            return symbol_table[i].data_type;
        }
    }
    return "null";
}
```