COSMOS

Team 10 Members

1) Avula Dinesh : CS20BTECH11005
2) Ayush Jha : CS20BTECH11006
3) Yuvraj Singh Shekhawat : CS20BTECH11057
4) Rohan Atkurkar : CS20BTECH11041
5) Shashank Shanbag : CS20BTECH11061

Roles

1) Avula Dinesh : Tester
2) Ayush Jha : System Integrator
3) Yuvraj Singh Shekhawat : Language Guru
4) Rohan Atkurkar : System Architect
5) Shashank Shanbag : Project Manager

Language Features Implemented

COSMOS provides:

- Handles large numbers upto 10^^{65000} by using scientific notification
- All basic gravitational formulaes mentioned in whitepaper.
  - Gravitational Force between two objects
  - Kepler’s law
  - Acceleration due to gravity
  - Gravity below the surface of a homogeneous planet
  - Gravity above the surface of a planet
  - Gravitational Potential Energy

Language Features Implemented

  - Escape velocity
  - Orbital velocity
  - Time period for satellite
  - Binding Energy
  - Satellite orbit height
- Black-Body Radiation formulae:-
  - Stefan's Law
  - Wien's Displacement Law
  - Rayleigh-Jeans Law

Language Features Implemented

- Relativity Formulas
  - Lorentz factor
  - Energy-Mass Equivalence
- Light Formulas
  - Energy of lightwave
  - Redshift
  - Parallax
- Star Magnitude Formulas and many more......

COSMOS?

- COSMOS is predominantly built for space scientists and learning students. It is a statically typed programming language, and it serves as a scientific calculator for astronomical problems.
- It aims at assisting people with no background in programming. Various familiar data types and functions implementing a variety of formulas make COSMOS a user-friendly language.

Phases of Transpiler

$$\begin{tikzcd}
\text{Input: .cos file} \arrow[r] & \text{Lexical Analysis} \arrow[d] \\
 & \text{Syntactic Analysis} \arrow[d] \\
 & \text{Semantic Analysis} \arrow[d] \\
\text{Output: .cpp file} & \text{Code generation} \arrow[l]
\end{tikzcd}$$

What is Lexical Analysis?

1) It converts the input program into a sequence of tokens.

2) A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

3) Lexer does these 4 tasks : Tokenization, Removing whitespace characters, remove comments, generates error messages by providing row and column numbers.

What does our Lexer do?

1) lexer.l takes an input cosmos program(.cos) and returns stream of tokens.
2) Our lexical analyzer uses flex.
3) Generated stream of tokens will be used by the parser for further processing.

How to run Lexer?

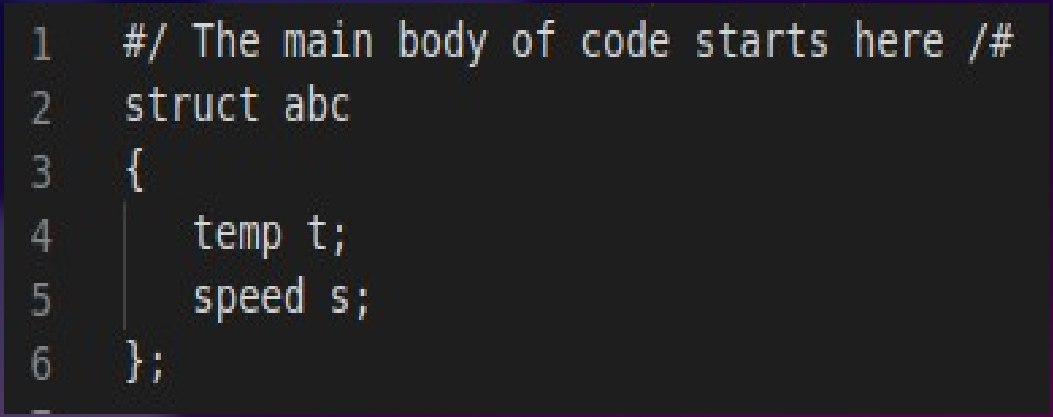
1) To run lexer using terminal commands :

- lex lexer.l
- gcc -o lexer_out lex.yy.c
- ./lexer_out <input_file_name> output_file_name

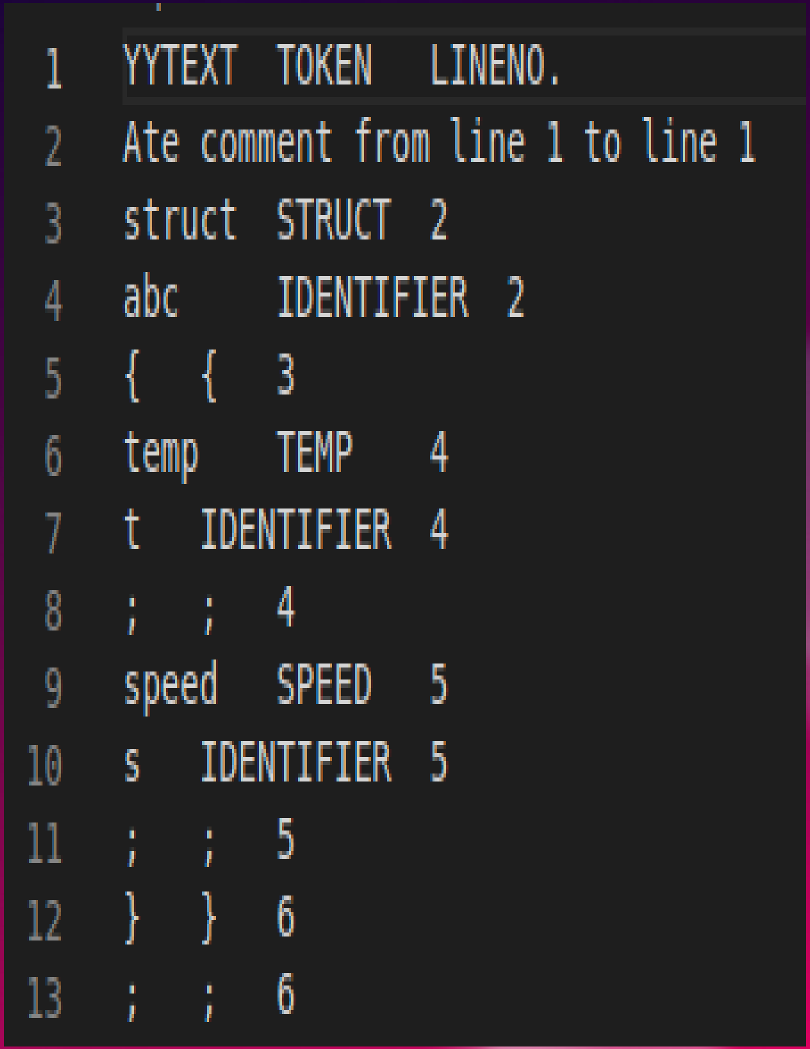
2. To simply run our lexer on all test cases using makefile, the command is: make

3. To remove all the output files and intermediate generated code files, the command is : make clean

Image of working lexer



```
#/ The main body of code starts here /#
struct abc
{
    temp t;
    speed s;
};
```



```
YYTEXT  TOKEN   LINENO.
Ate comment from line 1 to line 1
struct  STRUCT  2
abc     IDENTIFIER  2
{   {   3
temp    TEMP    4
t   IDENTIFIER  4
;   ;   4
speed   SPEED   5
s   IDENTIFIER  5
;   ;   5
}   }   6
;   ;   6
```

What is Syntactic Analysis(Parser)?

- The parser is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as *Syntax Analyzer*.
- We have implemented Bottom-up Parser as we have used ANSI C grammar as our reference.
- Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.

What does our Parser do?

- lexer.l takes an input cosmos program(.cos) and returns stream of tokens . Our lexer generates lex.yy.c
- parser.y takes stream of tokens returned by lexer and checks whether it follows the defined grammar. If it does not follow, then it gives syntax error with line no. mentioned.
- Our Parser generates files y.tab.h and y.tab.c files.
- We are using Yacc.

How to use Parser?

To manually run the parser

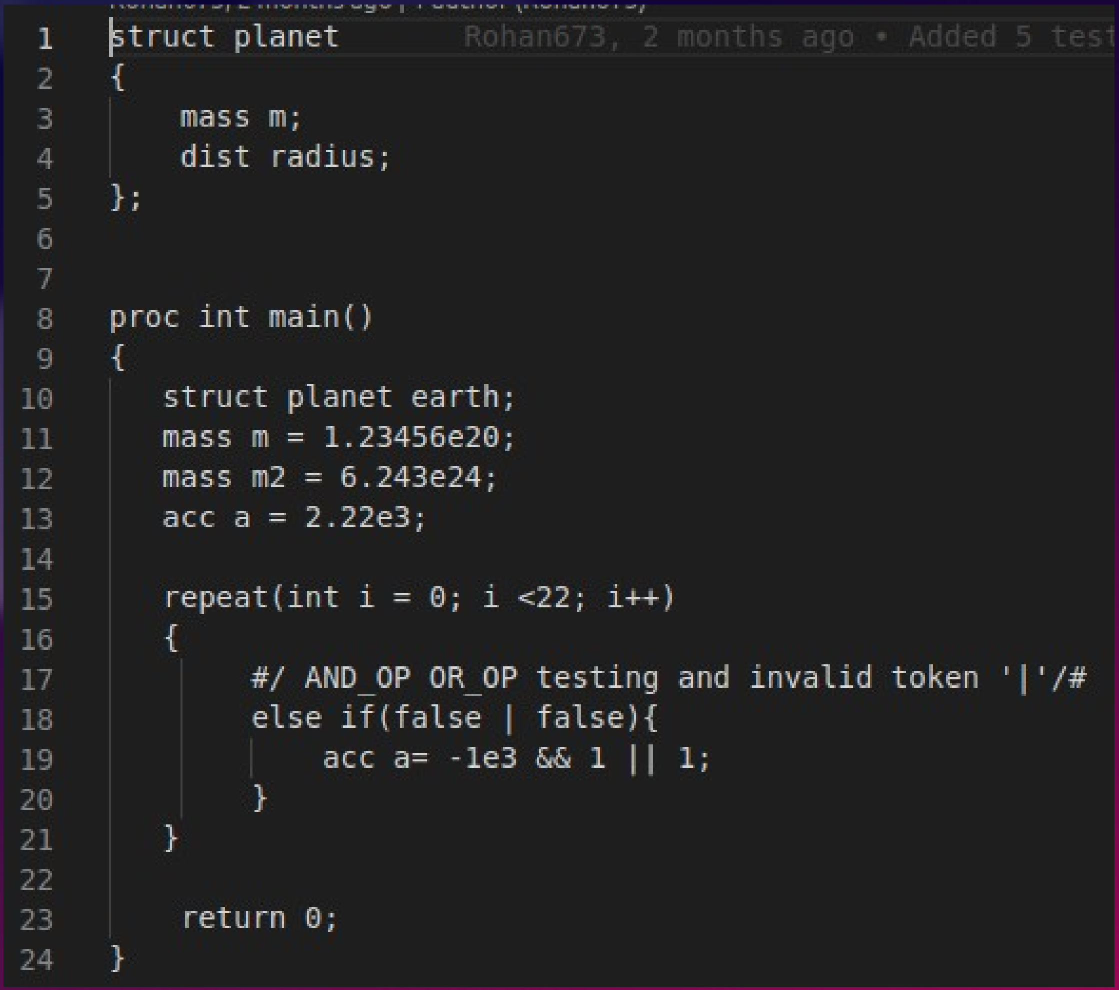
1) flex lexer.l
2) yacc -d parser.y
3) gcc -o testing_parser lex.yy.c y.tab.c -lfl
4) ./testing_parser < input_filepath

To run the parser using Makefile

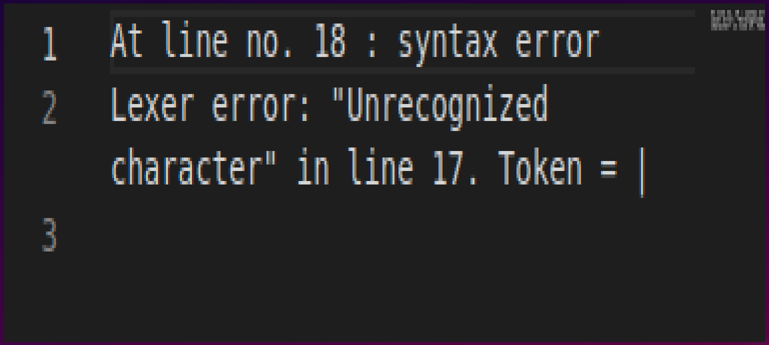
- **Run a test case** make test
- **Run all test cases** make all
- **Error message format(if any)**

  At line no. l, syntax error

Image of working Parser



```
 1  struct planet         Rohan673, 2 months ago • Added 5 test
 2  {
 3      mass m;
 4      dist radius;
 5  };
 6
 7
 8  proc int main()
 9  {
10     struct planet earth;
11     mass m = 1.23456e20;
12     mass m2 = 6.243e24;
13     acc a = 2.22e3;
14
15     repeat(int i = 0; i <22; i++)
16     {
17          #/ AND_OP OR_OP testing and invalid token '|'/#
18          else if(false | false){
19              acc a= -1e3 && 1 || 1;
20          }
21     }
22
23      return 0;
24  }
```



```
At line no. 18 : syntax error
Lexer error: "Unrecognized
character" in line 17. Token = |
```

What is Semantic Analysis?

- Semantic Analysis makes sure that declarations and statements of program are semantically correct.
- It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition.
- Type checking is an important part of semantic analysis which makes sure that each operator has matching operands.

What is Symbol Table?

- Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.
- It is built-in lexical and syntax analysis phases.
- The information is collected by the analysis phases of the compiler and is used by the synthesis phases of the compiler to generate code.
- It is used by the compiler to achieve compile-time efficiency.

How is our Symbol Table implemented?

- We have used STL Map Data Structure.
- There are 4 types of labels:
  - IDENT(for identifiers),
  - FUNCTION (for functions),
  - STRUCT_IDENT (for structs),
  - UNDEF (Other than the above 3)

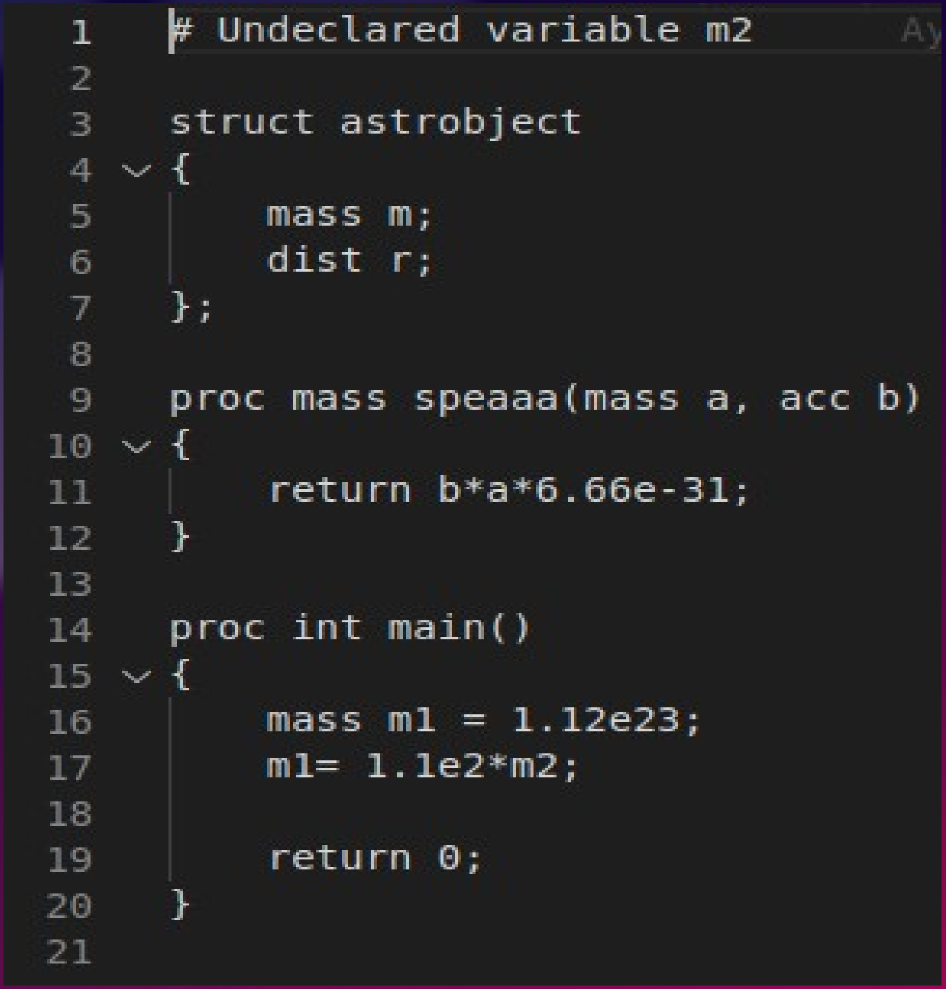
How to run Semantic Analyzer?

- To run the semantic analyzer using Makefile on test cases, use command: make
- To run the semantic analyzer manually,
  - flex lexer.l
  - bison -d -t parser.y
  - g++ -o testing_parser.out lex.yy.c parser.tab.c -lfl
  - ./testing_parser.out testFile.cos

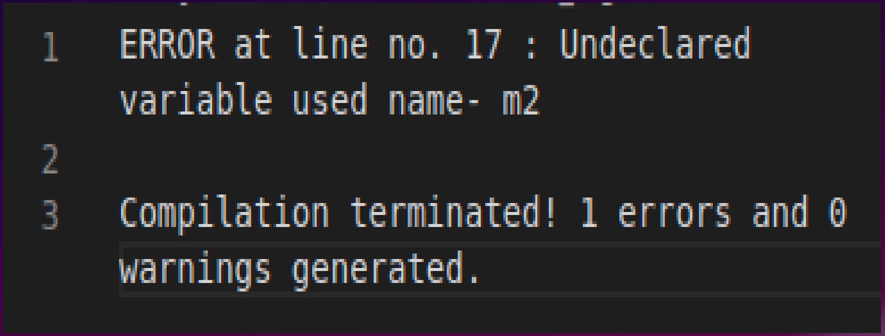
What does our Semantic Analyzer do?

- Checks for any undeclared variables.
- Checks for any undeclared functions in use.
- Checks for variables which are declared but unused.
- Checks if any operator has invalid operands.
- Type checking.

Image of working Semantic Analyzer



```
# Undeclared variable m2

struct astrobject
{
    mass m;
    dist r;
};

proc mass speaaa(mass a, acc b)
{
    return b*a*6.66e-31;
}

proc int main()
{
    mass m1 = 1.12e23;
    m1= 1.1e2*m2;

    return 0;
}
```



```
ERROR at line no. 17 : Undeclared
variable used name- m2

Compilation terminated! 1 errors and 0
warnings generated.
```

What is Code generation?

- For Transpiler, it involves converting/translating the semantically correct input code into another high-level programming language with appropriate headers defined for the new datatypes and new operators involved in the code.
- In our case, we convert:

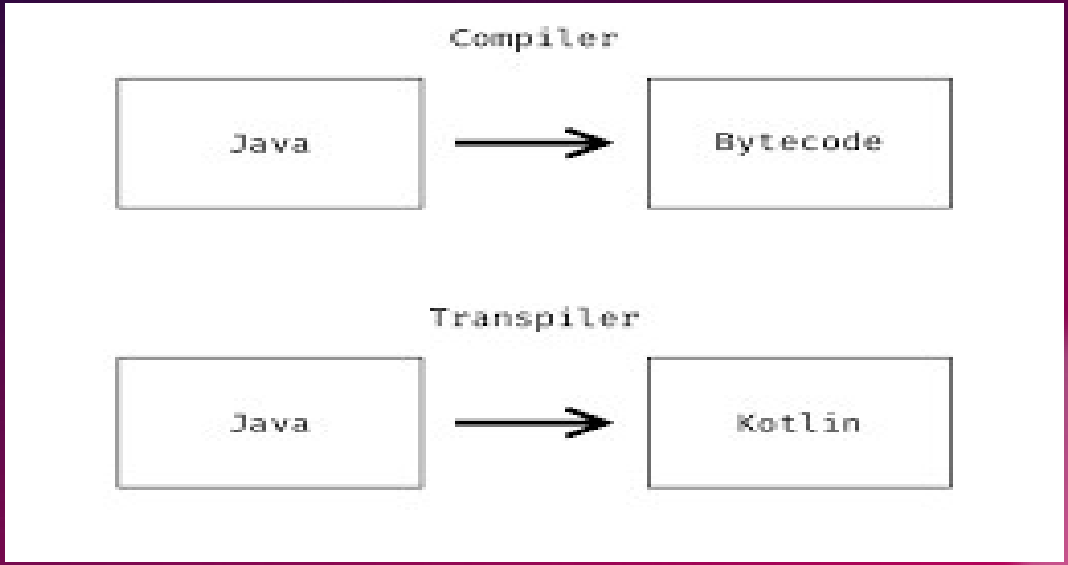
$$\text{.cos} \longrightarrow \text{.cpp}$$

What is Transpiler?

- Also known as source-to-source translator, source-to-source compiler (S2S compiler), transcompiler.
- It takes the source code of a program written in a programming language as its input and produces an equivalent source code in the same or a different programming language.
- Example: JSweet transpiles from Java to TypeScript or JS.

How it is different from Compiler?

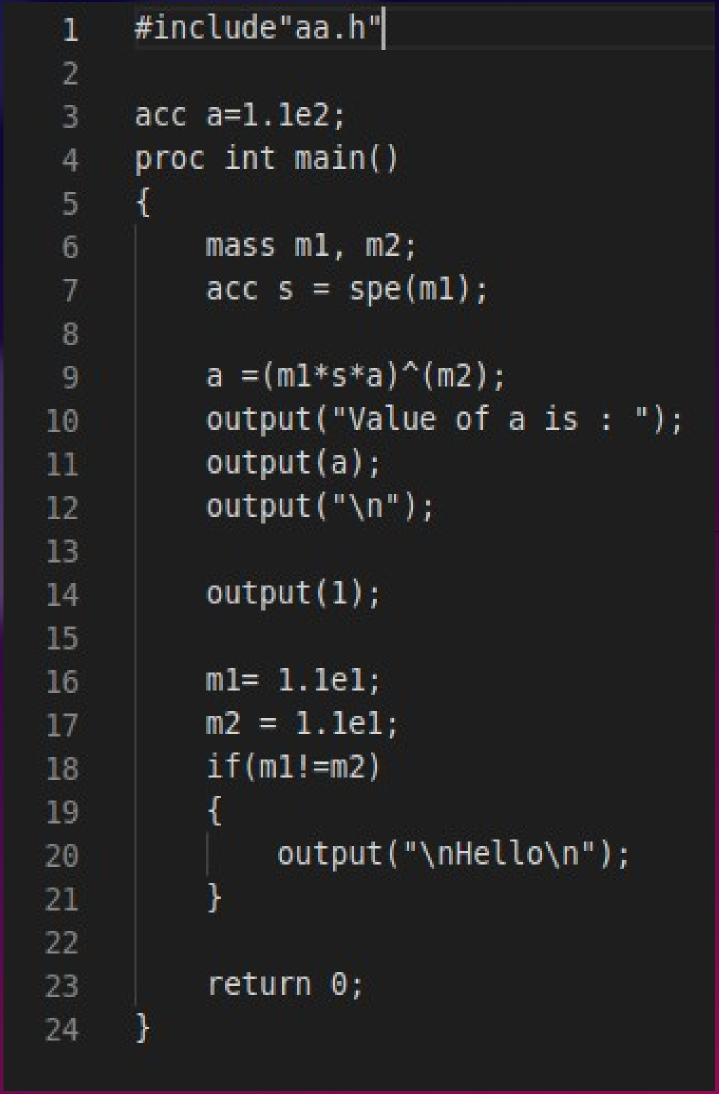
- The biggest difference is that Compiler converts the input source code written in a high-level language to an output code in a low-level language whereas Transpiler converts the input source code written in a high-level language to another high-level language.



How to run Code generation phase?

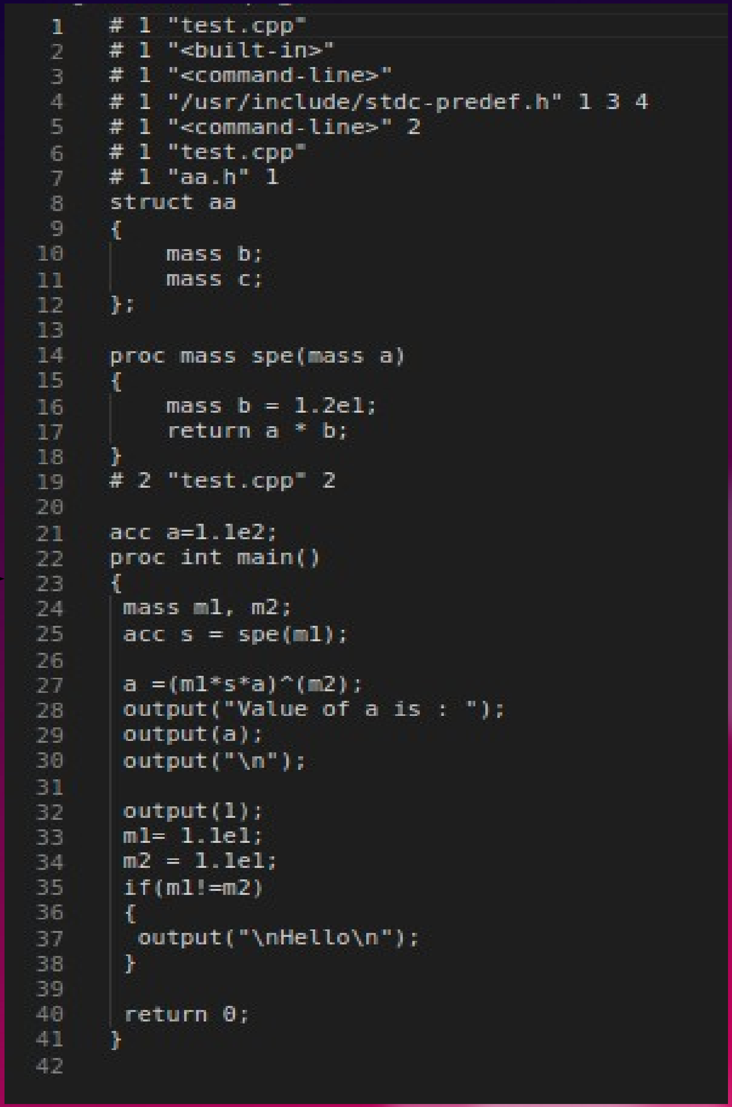
- To run the code generation phase using Makefile on test cases, use command: make
- To run the code generation phase manually,
  - flex lexer.l
  - bison -d -t parser.y
  - g++ -o testing_parser.out lex.yy.c parser.tab.c -lfl
  - cp testFile.cos testFile.cpp
  - g++ testFile.cpp -o PREPROCESSED_testFile.cpp.cos -E
  - rm testFile.cpp
  - ./testing_parser.out PREPROCESSED_testFile.cpp.cos

Image of working Code generation



```
#include"aa.h"

acc a=1.1e2;
proc int main()
{
    mass m1, m2;
    acc s = spe(m1);

    a =(m1*s*a)^(m2);
    output("Value of a is : ");
    output(a);
    output("\n");

    output(1);

    m1= 1.1e1;
    m2 = 1.1e1;
    if(m1!=m2)
    {
        output("\nHello\n");
    }

    return 0;
}
```

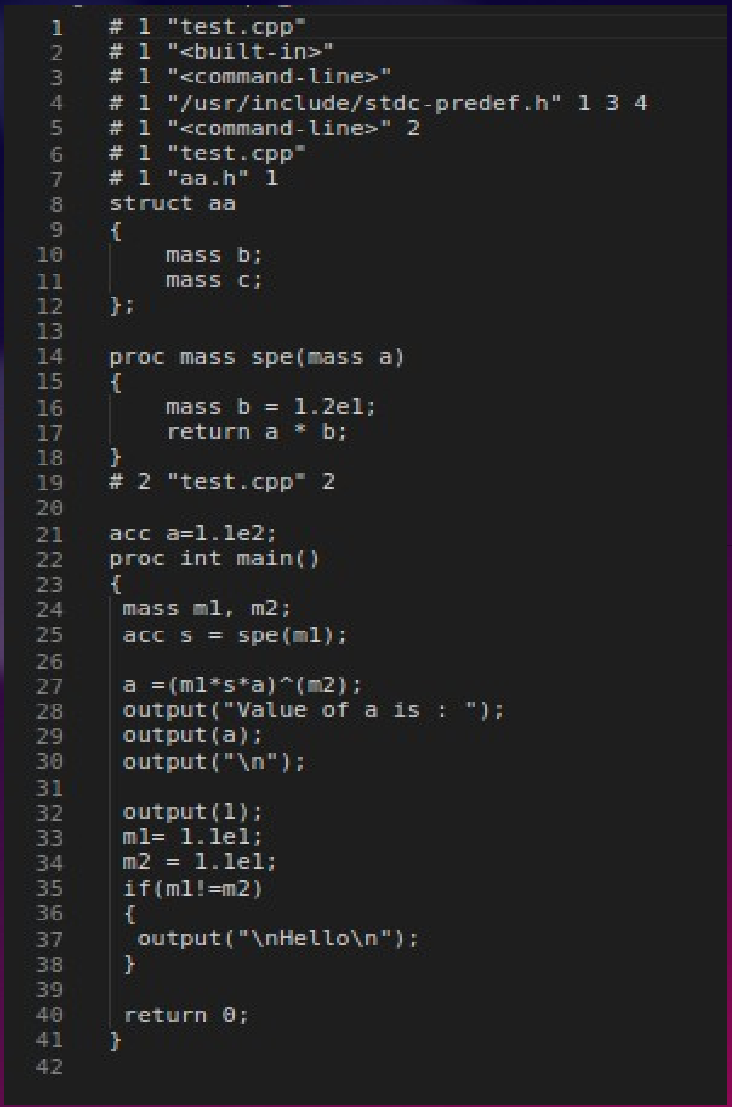
Source Code



```
# 1 "test.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "test.cpp"
# 1 "aa.h" 1
struct aa
{
    mass b;
    mass c;
};

proc mass spe(mass a)
{
    mass b = 1.2e1;
    return a * b;
}
# 2 "test.cpp" 2

acc a=1.1e2;
proc int main()
{
 mass m1, m2;
 acc s = spe(m1);

 a =(m1*s*a)^(m2);
 output("Value of a is : ");
 output(a);
 output("\n");

 output(1);
 m1= 1.1e1;
 m2 = 1.1e1;
 if(m1!=m2)
 {
  output("\nHello\n");
 }

 return 0;
}

```

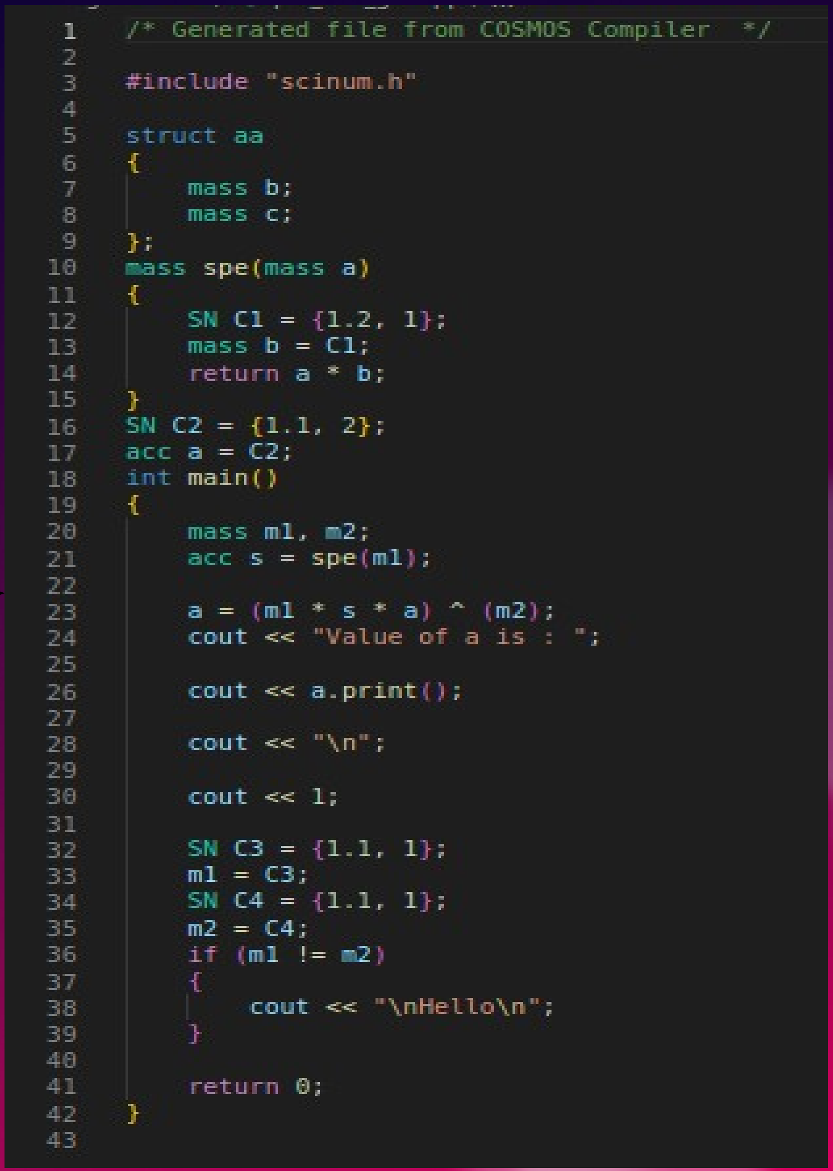
Preprocessed Code
(After Semantic, Before Codegen)

Image of working Code generation



```
# 1 "test.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "test.cpp"
# 1 "aa.h" 1
struct aa
{
    mass b;
    mass c;
};

proc mass spe(mass a)
{
    mass b = 1.2e1;
    return a * b;
}
# 2 "test.cpp" 2

acc a=1.1e2;
proc int main()
{
 mass m1, m2;
 acc s = spe(m1);

 a =(m1*s*a)^(m2);
 output("Value of a is : ");
 output(a);
 output("\n");

 output(1);
 m1= 1.1e1;
 m2 = 1.1e1;
 if(m1!=m2)
 {
  output("\nHello\n");
 }

 return 0;
}
```

Preprocessed Code
(After Semantic, Before Codegen)



```
/* Generated file from COSMOS Compiler  */

#include "scinum.h"

struct aa
{
    mass b;
    mass c;
};
mass spe(mass a)
{
    SN C1 = {1.2, 1};
    mass b = C1;
    return a * b;
}
SN C2 = {1.1, 2};
acc a = C2;
int main()
{
    mass m1, m2;
    acc s = spe(m1);

    a = (m1 * s * a) ^ (m2);
    cout << "Value of a is : ";

    cout << a.print();

    cout << "\n";

    cout << 1;

    SN C3 = {1.1, 1};
    m1 = C3;
    SN C4 = {1.1, 1};
    m2 = C4;
    if (m1 != m2)
    {
        cout << "\nHello\n";
    }

    return 0;
}
```

After Codegen(Transpilation)

Thank You