COMPILERS

maTricks OVERVIEW

Language Features

- Primitive types
- Operators
- Control flow
- Functions
- User defined types

Primitive Types

- matrixInt: denoted by keyword
- mInt matrixFloat: denoted by keyword
- mFloat matrixString: denoted by keyword
- mString Scalar: denoted by keyword scalar (scalar represents a floating point number.)
- Index: denoted by keyword index (index is especially used for indices in iterators. It takes only positive integers

Operators

=	assignment
+	element wise addition in matrices and scalar addition
-	element wise subtraction in matrices and scalar subtraction
*	matrix multiplication & scalar, scalar multiplication
**	scalar multiplication with matrix
$	Acumulator

- / division with scalar
- # inverse of matrix
- ‘ Transpose
- => Function return type delimiter
- Relational operators
- ==, >= ,<= ,> ,< ,!=
- Logical operators
- ||, ,&&

Bitwise Operators:

~	(xor)
!	(not)
~~	(xor with scalar)
|	(or)
&	(and)

Control Flow

- If-else statements :

```
if (index)
  { statements }
else if  (index)
  { statements }
else  { statements }
```

- Loops :

```
for(index i=0;i<INDEXtype;i++)
 {
    statements
 }
```

Functions

```
Def SolveEquations(mInt A(9,9),mString B(9,9))=> mFloat(9,9)
{
    mFloat X(1,1)=A*(#B);


    return X;
}
```

User Defined Types

```
mStruct Hello
{
   mInt avgsdf(5,3);
    mInt saA(4,5);
};
```

PHASES OF COMPILER

- Lexical Analysis
- Parsing
- Semantic Analysis
- Code Generation

Lexical Analysis

Lexers

- Specification:-
  - Identifiers - {ALPHA}{ALPHANUM}*
  - Keywords - Def, return, ONES, ZEROES, ROW, COLUMN, IDENTITY, lambda, mInt, mFloat, mString, mStruct, scalar, if, else, for, break, pass
    - Operators - =, +, -, *, **, $, / , #, ‘ , ==, >= ,<= ,> ,< ,!= ,|| ,| ,& ,&& ,~ , ! ,~~ , =>
    - Delimiters - ; , ({ [ ] }) Other tokens – number constants-{DIGIT}+ - {DIGIT}+"."{DIGIT}*
    - comments-

      /* MULTILINE COMMENTS */

      // SINGLE LINE COMMENTS

Operator Specifications:

=	assignment
+	element wise addition in matrices and scalar addition
-	element wise subtraction in matrices and scalar subtraction
*	matrix multiplication & scalar, scalar multiplication
**	scalar multiplication with matrix
$	Acumulator

/	division with scalar
#	inverse of matrix
‘	Transpose
=>	Function return type delimiter

Relational operators

==, >= ,<= ,> ,< ,!=

Logical operators

||, ,&&

Bitwise Operators:

~	(xor)
!	(not)
~~	(xor with scalar)
|	(or)
&	(and)

Implementation

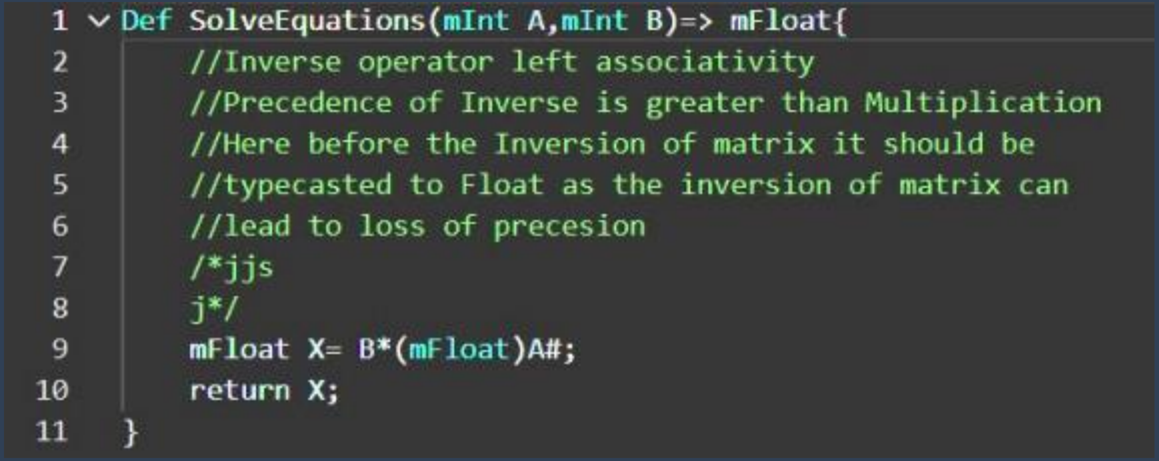
- We used a lexical analyzer generator tool named FLEX which is an improved version of lex. The tool consists of 3 main parts. The syntax of the parts is described as follows.
  - Definition:

    ```
    %{
        //definitions
    %}
    ```
  - Rules:

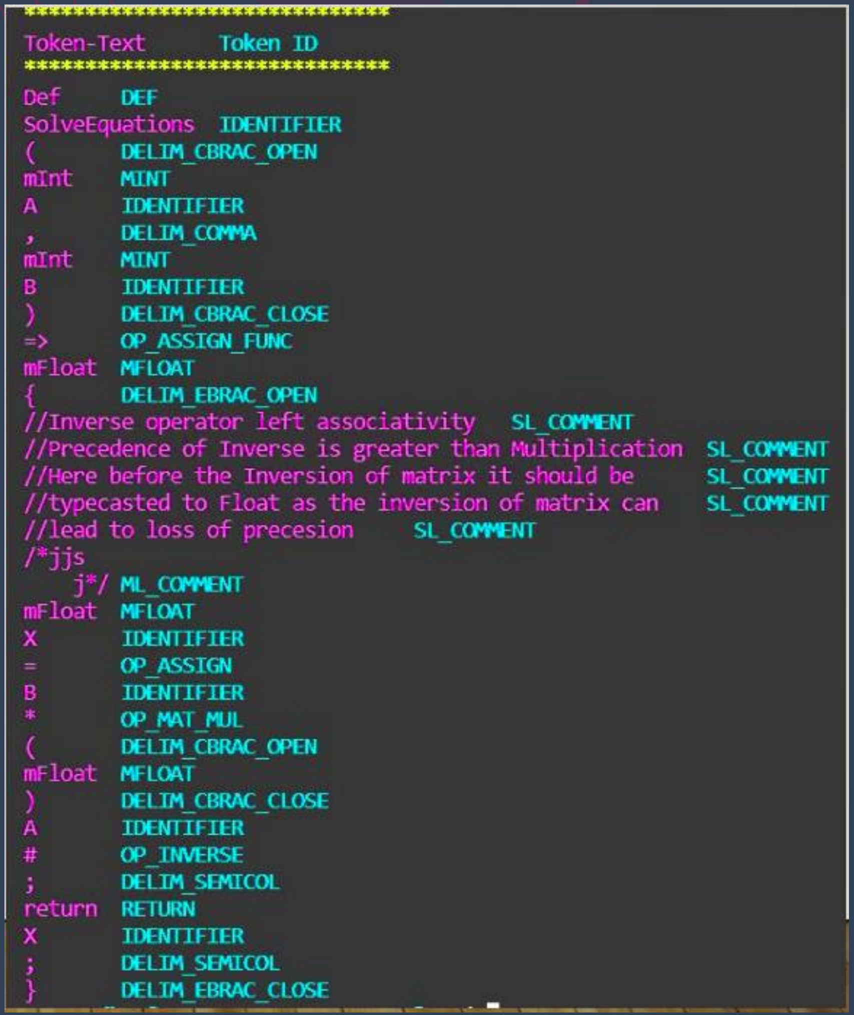
    ```
    %%
        //rules
    %%
    ```
  - User code section:

    ```
    //enter your code
    ```

Examples



```
Def SolveEquations(mInt A,mInt B)=> mFloat{
    //Inverse operator left associativity
    //Precedence of Inverse is greater than Multiplication
    //Here before the Inversion of matrix it should be
    //typecasted to Float as the inversion of matrix can
    //lead to loss of precesion
    /*jjs
    j*/
    mFloat X= B*(mFloat)A#;
    return X;
}
```



```
******************************
Token-Text      Token ID
******************************
Def       DEF
SolveEquations  IDENTIFIER
(         DELIM_CBRAC_OPEN
mInt      MINT
A         IDENTIFIER
,         DELIM_COMMA
mInt      MINT
B         IDENTIFIER
)         DELIM_CBRAC_CLOSE
=>        OP_ASSIGN_FUNC
mFloat    MFLOAT
{         DELIM_EBRAC_OPEN
//Inverse operator left associativity   SL_COMMENT
//Precedence of Inverse is greater than Multiplication  SL_COMMENT
//Here before the Inversion of matrix it should be      SL_COMMENT
//typecasted to Float as the inversion of matrix can    SL_COMMENT
//lead to loss of precesion     SL_COMMENT
/*jjs
    j*/   ML_COMMENT
mFloat    MFLOAT
X         IDENTIFIER
=         OP_ASSIGN
B         IDENTIFIER
*         OP_MAT_MUL
(         DELIM_CBRAC_OPEN
mFloat    MFLOAT
)         DELIM_CBRAC_CLOSE
A         IDENTIFIER
#         OP_INVERSE
;         DELIM_SEMICOL
return    RETURN
X         IDENTIFIER
;         DELIM_SEMICOL
}         DELIM_EBRAC_CLOSE
```

Parsers

Parser -- Grammar Design

- It is a generated parser using BISON(YACC) ● Grammar is of type LALR
- Made sure that there are no shift/Reduce and Reduce - reduce conflicts
- Program is assumed to be made of any number of statements. Where each statement can be of type
  - Declarations statement
  - Expression statement
  - Compound statement
  - Iteration / selection statement
  - Function definitions etc….
- Order of precedence and associativity of operators is defined before hand in YACC

Error Handling

- Token ‘error’ is added at productions where there is potential chance for a error.
- Here this error token is added at various levels ,like from inside of particular statement like (at location where size is declared in variable declaration) to overall statement level
  - Ex:
    - Identifier -> IDENTIFIER error INITIALISATION
    - statement -> error ‘;’
  - Here if the error cannot be handled in 1st production , it is at least caught in the overall statement (may be skipping some statements if ‘;’ is not present) to prevent further errors

- The location of error is retrieved from the lexer (by maintaining the lineNo. (incremented on every new line) and columnNo. (incremented by length of the token returned to the parser)
- As the action of appropriate error production, corresponding error message is printed along with the location information.

SEMANTIC PHASE

Specification

- The actions to be performed for each syntactic rule (for each expression in the grammar) is written in the curly braces in the bison tool.
- Corresponding to each grammar rule, actions are written with the functions are invoked which are written in other module named "semantics.c"

Implementation

- We had two parts to implementation
  - Symbol table:

    which has been done like hash table with adjacency list like implementation.
  - semantics.c Module:

    which contains the functions for type checking and error handling. All the functions required for semantic checks are written in this module. All the entries in the symbol table are filled by this module.

    Example:- type checking is done by a function called check_type

Interface with tool

- We have used modular structure of files by invoking headers to interface between tool and files
- semantic.h header file is included in the tool so that the functions written in the braces can be called.
- In semantics.h file the interface with symbol table sym_tab.h is included.

CodeGen

IR

- Directly generated IR from the Semantic actions of grammar
- Explored LLVM IR and not able to find a way to represent our data types (matrices primitively)
- So, decided to make our own simple IR to suit our needs later in machine code generation
- It is a Quadruple type IR
- Action res sz op1 sz OP op2 sz
  - Assign t1 2,2 a 2,2 + b 2,2
  - I.e t1 is assigned the sum of two 2X2 matrices

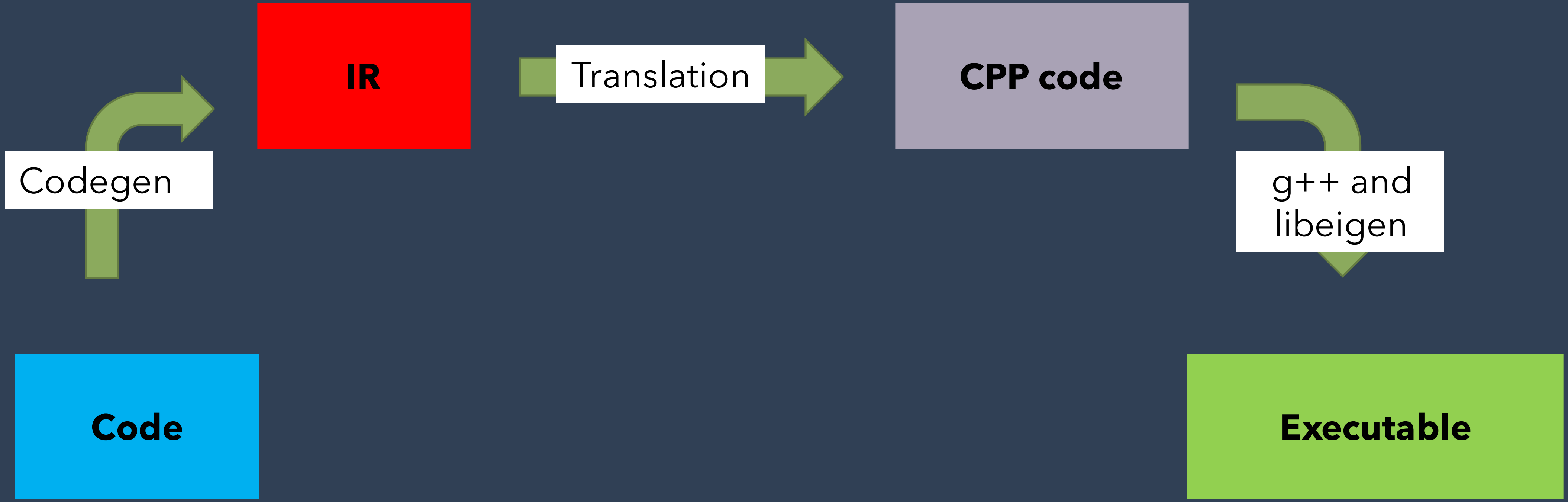
Syntax

- Declaration
  - define type var sz : define MINT a 2,2 (MINT is encoded as int)
- Assignment
  - assign var sz a sz OP b sz
  - val-assign a ele b ele
  - call-assign var sz funcname
- Functions
  - func-begin funcname
  - arg-define type var sz
  - func-end

Translations

- As we choose our custom IR, to generate the machine code, we are left with only choice to translate to some known form
- We choose to translate the IR to CPP code
- As for our design of the language, we should be performant, so instead of redoing the work we decided to use library
- We are using libeigen for this work. It is built for this purpose of efficient matrix computations

WORK FLOW


IR
Translation
CPP code
Codegen
g++ and libeigen
Code
Executable

Team

NAME	ROLE
P Srivardhan	Project Manager
V Nikhilesh	System Architect
Mahidhar Alam	System Architect
K Rithvik	Language Guru
Prashanth Routhu	System Integrator
M Shathanand Sai	System Tester
T Eshwar	System Tester

THANK YOU