Developer documentation- BlueCompiler

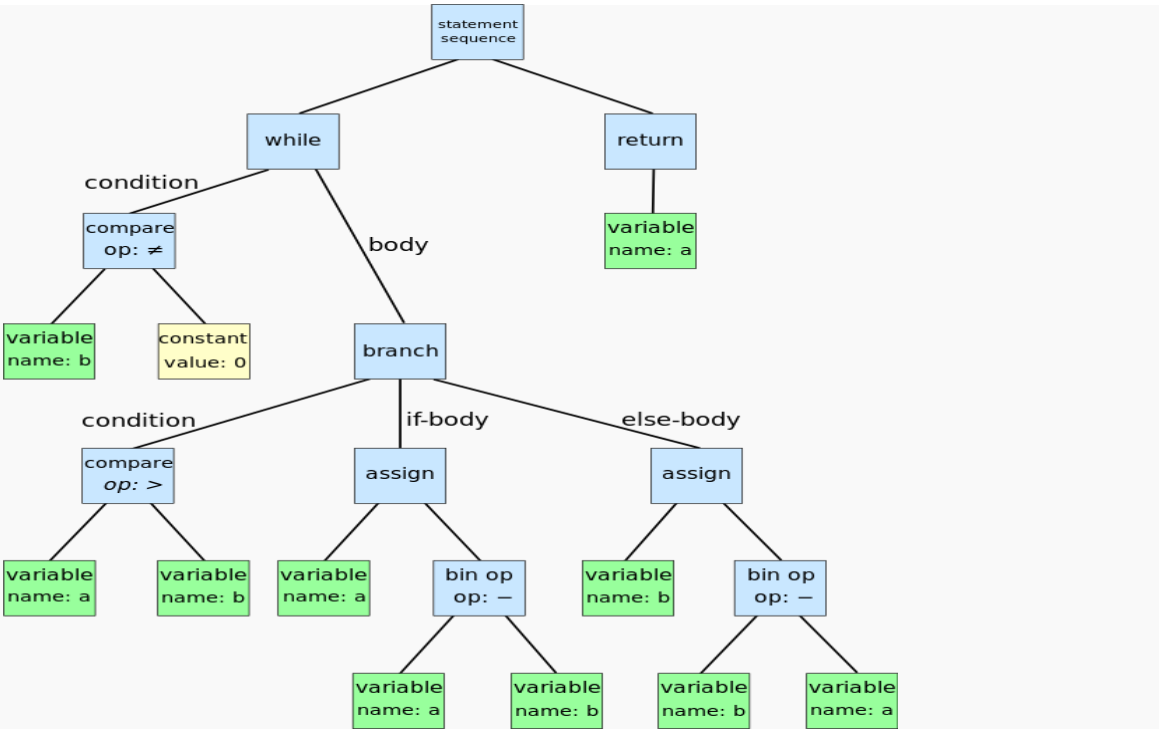
1 INTRODUCTION TO FLEX AND BISON AND COMPILER CONSTRUCTION

Writing a compiler from scratch requires us to decide the grammar for the language. Let us go through the steps that a standard compiler goes through for any language:

1. **Lexical Analysis with *Flex***: Split input data into a set of tokens (identifiers, keywords, numbers, brackets, braces, etc.)
2. **Semantic Parsing with *Bison***: Generate an AST while parsing the tokens. Bison will do most of the legwork here, we just need to define our AST.
3. **Execution with C++**: This is where we walk over our AST and execute the AST using an Interpreter designed in C++. An alternative here would be to generate the machine code and achieve platform independence.

To get a basic understanding of AST:
An abstract syntax tree for the following code for the following algorithm:

```
while b ≠ 0
if a > b
a := a − b
else
b := b − a
return a
```



2 LEXER GENERATOR OR LEX.L

The first task would be to break down the input data stream into some meaningful tokens. The conversion of unclassified raw data to the tokens is done by the lexer. A lexer generator file tells the Flex tool a set of rules that Flex would use to generate a lexer. The rules here are defined using rules of simple regular expressions. A good concept of regular expression is required to understand the following code. Explanation of regular expression is beyond the scope of this document.

A simple flex file has the following symantic format:

definitions

%%

rules

%%

user code

The flex input file consists of three sections, separated by a line with just `%%' in it

Let us look at our flex file now:

```
%option noyywrap
```

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

%option noyywrap

%%

"while" return TOKEN_WHILE;
"!="    {yylval.op = _strdup(yytext); return TOKEN_OPERATOR;}
"<="    {yylval.op = _strdup(yytext); return TOKEN_OPERATOR;}
">="    {yylval.op = _strdup(yytext); return TOKEN_OPERATOR;}
"&&"    {yylval.op = _strdup(yytext); return TOKEN_OPERATOR;}
"||"    {yylval.op = _strdup(yytext); return TOKEN_OPERATOR;}
"{"   return TOKEN_BEGIN;
"}"   return TOKEN_END;
"["   return BOX_OPEN;
"]"   return BOX_CLOSE;
"do"  return TOKEN_DO;
"if"  return TOKEN_IF;
"else" return TOKEN_ELSE;
"vector" return TOKEN_VECTOR;
"vector2d" return TOKEN_VECTOR2d;
"return" return TOKEN_RETURN;
"pow"  return TOKEN_POW;
"factorial" return TOKEN_FACTORIAL;
"acos" return TOKEN_ACOS;
"sqrt" return TOKEN_SQRT;
"rotatez" return TOKEN_ROTATEZ;
"magnitude_squared" return TOKEN_MAGNITUDESQR;
"transform" return TOKEN_TRANSFORM;
"min" return TOKEN_MIN;
"dot" return TOKEN_DOT;
"cross" return TOKEN_CROSS;
","    return TOKEN_COMMA;
"=="    {yylval.op = _strdup(yytext); return TOKEN_OPERATOR;}
[a-zA-Z_][a-zA-Z0-9_]* {yylval.name = _strdup(yytext); return TOKEN_ID;}
[-]?[0-9]*[.]?[0-9]+    {yylval.val = atof(yytext); return TOKEN_NUMBER;}
[()=;]  {return *yytext;}
[*/+-<>%] {yylval.op = _strdup(yytext); return TOKEN_OPERATOR;}
[ \t\n] {/* suppress the output of the whitespaces from the input file to stdout */}
#.* {/* one-line comment */}

%%
```

The first portion of the file specifies the imports required by the tool to parse the rest of the logic in the file. You can add your custom imports here and use functionality imported. For example if I have a complex header that defines a function to scan numbers with more precision, than offered by Flex, we simply import the header and use the functionality in the lexer. An importand thing to note with respect to the project: if we need higher precision in parsing, it would be a good idea to define a new scanning

method and import it in the lexer generator file. The problem with the approach would be though the lexer has the potential to scan high precision numbers the parser does not understand this number. The result could be wrong value beyond a certain point (concept of wrap around of data types). The fix is to also look into the parser and custom built it to understand this precision.

In the lexer generator file, we notice that we have defined the tokens that we would like to scan and the word patterns corresponding to the tokens. This can be thought of as a machine that knows the word pattern to look for and when a particular pattern is found; it stores the scan as per the rule defined for it. As an example consider:

```
"pow"  return TOKEN_POW;
```

This would scan for the group of letters as {‘p’,’o’,’w’} and if a occurrence is found where all three letters are arranged consecutively in the specified order, it replaces the internal representation of the scan with a simple TOKEN_POW.

In some cases we need to store the actual scan rather than a named conversion for it. For example:

```
[-]?[0-9]*[.]?[0-9]+    {yylval.val = atof(yytext); return TOKEN_NUMBER;}
```

This would look for numbers and store them. The function ‘atof(yytext)’ is used to scan floating point numbers. Note: this is a limitation of our scanner, we cannot scan numbers with higher precision than float.

With our understanding of the lexer generator and the lexer, we should now focus on the language grammar. The lexer returns tokens which are used by the parser in the next stage to create an AST.

3 LANGUAGE GRAMMAR

Let us take a look at the grammar used in the language construction. The language semantics were designed keeping a similarity with Python programming language. The syntax of the resulting language is similar to Python.

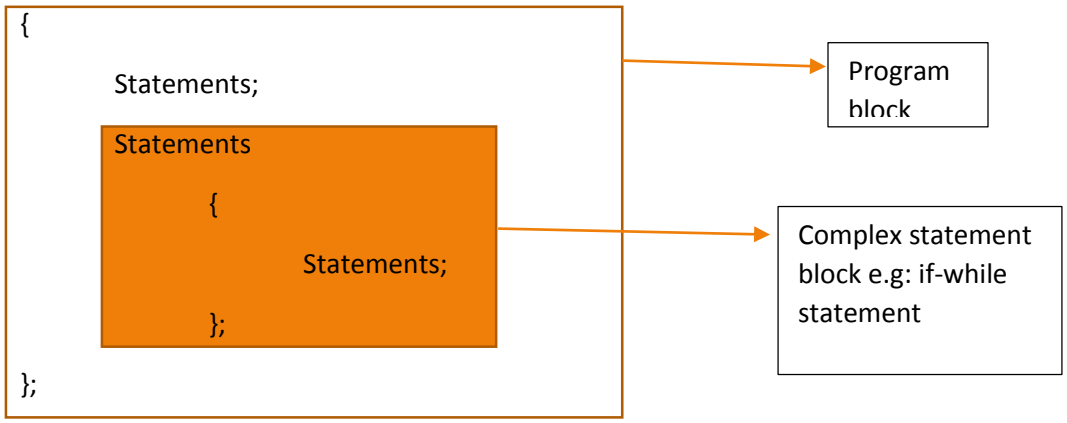
```
program: statement';' { astDest = $1; };

block: TOKEN_BEGIN statements TOKEN_END{ $$ = $2; };

statements: {$$=0;}
    | statements statement ';' {$$=makeStatement($1, $2);}
    | statements block';' {$$=makeStatement($1, $2);};
```

This defines that each complete sentence in the program would be terminated by a ‘;’. In other words a program consists of a ‘statement’ and is terminated by a ‘;’. This makes us to jump into the definition of a ‘statement’. As we can see a statement definition is recursive on itself. This allows us to squeeze in multiple statements in the program. Notice the rule: `statements block';'` This means that a

statement can also consist of a block. But when we think of blocks we should also notice that a block is terminated by a ';' so essentially a program is constructed as a block in the form:



We get the idea of scope resolution from the block structure. Note that at present the language understands blocks but scope resolution is not defined in it. This is because the Execution environment (defined later) just keeps one simple stack internally which is not refreshed on scope resolution (global stack of pointers). This is done to keep the program structure simple and the execution fast. As the initial requirements did not require scope resolution, more importance was given to precise and fast computation.

Here $1, $2 represents the value of the token that would be returned by the flex lexer, $1 is the value of the first token as read by the lexer. For example for a rule:

```
statements block';'{$$=makeStatement($1, $2);};
```

Here $1 represents the value of the token read as 'statements' and $2 refers to the value of the token read as 'block'. These are now passed as an argument to a function 'makeStatement' which is responsible to creating the appropriate structure in the AST. These functions can be found in the file 'astgen.cpp'

As we have got the basic understanding of the components of the rule, let us look at the rest of the grammar:

```
statement:
      whileStmt {$$=$1;}
       | ifStmt{$$=$1;}
       | block {$$=$1;}
       | call {$$=$1;}
       | func {$$=$1;}
       | vector {$$=$1;}
       | vector2d {$$=$1;}
       | return {$$=$1;}
       | assignment {$$=$1;}
```

This rule just defines the type of statements that are supported by the language. Let us look at these statement types now.

```
whileStmt: TOKEN_WHILE '(' expression ')' TOKEN_DO statement{$$=makeWhile($3, $6);};


ifStmt: TOKEN_IF  '(' expression ')' TOKEN_DO statement{$$=makeIf($3, $6);};
      | TOKEN_IF  '(' expression ')' TOKEN_DO statement TOKEN_ELSE TOKEN_DO
      statement{$$=makeIf($3, $6, $9);};
      | TOKEN_IF  '(' expression ')' TOKEN_DO statement TOKEN_ELSE  TOKEN_IF '('
      expression ')' TOKEN_DO statement TOKEN_ELSE TOKEN_DO statement{$$=makeElseIf($3,
      $6, $10, $13, $16);};
```

We would be defining expression as:

```
expression: TOKEN_ID {$$=makeExpByName($1);}
       | TOKEN_NUMBER {$$=makeExpByNum($1);}
       | TOKEN_OPERATOR expression {$$=makeExp(NULL,$2,$1);}
       | expression TOKEN_OPERATOR expression {$$=makeExp($1, $3, $2);}
       | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE {$$=makeVec1delement($1, $3);};
       | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE BOX_OPEN TOKEN_NUMBER BOX_CLOSE
{$$=makeVec2delement($1, $3, $6);};
```

The last two lines of expression is added for supporting vectors and matrices in the language. We would soon look into the implementation for them but let us look at function definition first.

```
func:  TOKEN_ID TOKEN_ID '('  ')' statement {$$=makeFunc($2, $5);};
     | TOKEN_ID TOKEN_ID '(' signatures ')' statement {$$=makeFunc($2, $4, $6);};
```

Let us define signatures now:

```
signatures: {$$=0;}
          |signature signatures{$$=makeSignatures($2,$1);}
signature:
         TOKEN_COMMA TOKEN_ID assignment{$$=makeSignature($2,$3);}
         |TOKEN_ID assignment{$$=makeSignature($1,$2);}
```

Now that we know about signatures let us go one level back and look at grammar for vector and vector2d:

```
vector: TOKEN_VECTOR TOKEN_ID '=' '(' array ')' {$$=makeVector($2,$5);}
      | TOKEN_VECTOR TOKEN_ID '=' '(' ')' {$$=makeNullVector($2);}
      | TOKEN_VECTOR TOKEN_ID '=' arrExpression {$$=makeAssignment($2, $4);}


vector2d: TOKEN_VECTOR2d TOKEN_ID '=' BOX_OPEN vectors BOX_CLOSE
{$$=makeVector2d($2,$5);}
        | TOKEN_VECTOR2d TOKEN_ID '=' arrExpression {$$=makeAssignment($2, $4);}

vectors: {$$=0;}
       | vectors TOKEN_COMMA '(' array ')' {$$=makeVectors($1,$4);}
```

```
       | vectors '(' array ')' {$$=makeVectors($1,$3);}
       | vectors '('  ')' {$$=makeNullVectors();}
```

Now array is defined as:

```
array: {$$=0;}
     | array TOKEN_COMMA arrExpression {$$=makeArray($1,$3);}
     | array arrExpression  {$$=makeArray($1,$2);}
```

So now we dive deeper into arrExpression grammar:

```
arrExpression: TOKEN_ID {$$=makeExpByName($1);}
    | TOKEN_NUMBER {$$=makeExpByNum($1);}
    | arrExpression TOKEN_OPERATOR arrExpression {$$=makeExp($1, $3, $2);}
```

As we see that this grammar is built on top of the grammar for 'expression' so that we can impose more restrictions on the grammar for vectors and matrices.

Now let us look at the assignment grammar rule:

```
assignment: TOKEN_ID '=' expression {$$=makeAssignment($1, $3);}
          | TOKEN_ID {$$=makeAssignment($1);};
          | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE '=' expression
{$$=makeVecAssignment($1, $3, $6);}
          | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE BOX_OPEN TOKEN_NUMBER BOX_CLOSE '='
expression {$$=makeVec2dAssignment($1, $3, $6, $9);}
          | TOKEN_ID '=' call{$$=makeFuncAssignment($1, $3);}
```

The last two types of statements are the return and the call statements, the grammar for them is simple:

```
call: TOKEN_ID '(' array ')' {$$=makeCall($1, $3);};

return: TOKEN_RETURN expression{$$=makeReturnByExp($2);}
```

This concludes the major part of our grammar, but we also need to support various inbuilt functions. The ideal place to add rules for supporting inbuilt functions is at 'expression'. So, our new 'expression' grammar becomes:

```
expression: TOKEN_ID {$$=makeExpByName($1);}
       | TOKEN_NUMBER {$$=makeExpByNum($1);}
       | TOKEN_OPERATOR expression {$$=makeExp(NULL,$2,$1);}
       | expression TOKEN_OPERATOR expression {$$=makeExp($1, $3, $2);}
       | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE {$$=makeVec1delement($1, $3);};
       | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE BOX_OPEN TOKEN_NUMBER BOX_CLOSE
{$$=makeVec2delement($1, $3, $6);};
       | TOKEN_POW '(' expression TOKEN_COMMA expression ')'{$$=makePow($3, $5);};
       | TOKEN_FACTORIAL '(' expression ')'{$$=makeFact($3);};
       | TOKEN_ACOS '(' expression ')'{$$=makeAcos($3);};
       | TOKEN_SQRT '(' expression ')'{$$=makeSqrt($3);};
       | TOKEN_ROTATEZ '(' expression TOKEN_COMMA expression ')'{$$=makeRotatez($3,$5);};
```

```
       | TOKEN_MAGNITUDESQR '(' expression ')'{$$=makeMagnitudesqr($3);};
       | TOKEN_MIN '(' expression TOKEN_COMMA expression TOKEN_COMMA expression
')'{$$=makeMin($3,$5,$7);};
       | TOKEN_DOT '(' expression TOKEN_COMMA expression ')'{$$=makeDot($3,$5);};
       | TOKEN_CROSS '(' expression TOKEN_COMMA expression ')'{$$=makeCross($3,$5);};
       | TOKEN_TRANSFORM '(' expression ')'{$$=makeTransform($3);};
       | '(' expression TOKEN_OPERATOR expression ')' {$$=makeExp($2, $4, $3);}
```

All tokens used in the grammer should have a formal description:

```
%token TOKEN_BEGIN TOKEN_END TOKEN_WHILE TOKEN_DO BOX_OPEN BOX_CLOSE
```

To understand the concept of tokens we would need to refer ourself to the portion on lexing. The lex tokens are passed to the parser so, these are forward declarations for the parser generator (parser.y).

4 PARSER GENERATOR OR THE PARSER.Y

As we have an understanding of the grammar for the language, we would now understand the remaining parts of the parser generator file.

All tokens used in the grammer should have a formal description:

```
%token TOKEN_BEGIN TOKEN_END TOKEN_WHILE TOKEN_DO BOX_OPEN BOX_CLOSE
```

To understand the concept of tokens we would need to refer ourself to the portion on lexing. The lex tokens are passed to the parser so, these are forward declarations for the parser generator (parser.y).

We also have the possibility of forward declaring named tokens:

```
%token<op> TOKEN_OPERATOR
```

This is more from the point of view to make the code more legible. So the named token can be used in rules and would be less confusing for the reader to read.

The tokens that are used in the parser generator file for the first time should also be defined in the following form:

```
%type<ast> signature signatures array vector vector2d vectors return
```

Now that we are done with the forward declarations, we take a look at the data structure that is used to hold the AST generated by the parser:

```
struct AstElement* astDest;

union {
    double val;
    char* op;
    char* name;
    struct AstElement* ast;
}
```

We also have a function that would tell us if during parsing there are parse errors. This function returns the line number where the error/inconsistency with parse grammar is. The line number can be used to debug if there is a syntactical error in the program.

```
void yyerror(const char* const message)
{
    fprintf(stderr, "Parse error at :%s\n", message);

}
```

This function could be placed at any level of the compiler (the lexer, the parser or even the execution environment). Keeping this with the parser code gives us possibility to design more robust and informative debug/stack trace messages.

With the AST generated by the parser the second phase of compilation begins.

5 "HOUSTON, WE'VE HAD A PROBLEM HERE!"

We have an AST generated so far from the parser. We now can easily convert this to machine code and forget about the platform dependence. The execution would be done low level by assembly language and would be fast. This step looks tough but with the help of the tool called LLVM this can be a breeze.

This is practically the easiest step in the whole of the programming if we just need to convert it to machine code. Implementing an Interpreter from scratch is not an easy task and is not recommended. Our approach here is a forced decision as at the time of implementation LLVM was mostly supported in an UNIX based environment. We tried to tackle this by using CygWin on windows but due to a bug in its implementation, we could not get it to incorporate using CMake and Visual Studio seems to also have a bug with LLVM integration.

To take a look at the ongoing discussions that established the problems with LLVM, please check the following links:

llvm-3-4-linker-errors-on-vs-2012

llvm-compilation-errors-on-vs-2012

And finally the unproductive discussion on:

alternative-to-llvm-for-c-bytecode-generation

This forces us to implement an Interpreter. To keep the implementation flexible we have made it modular.

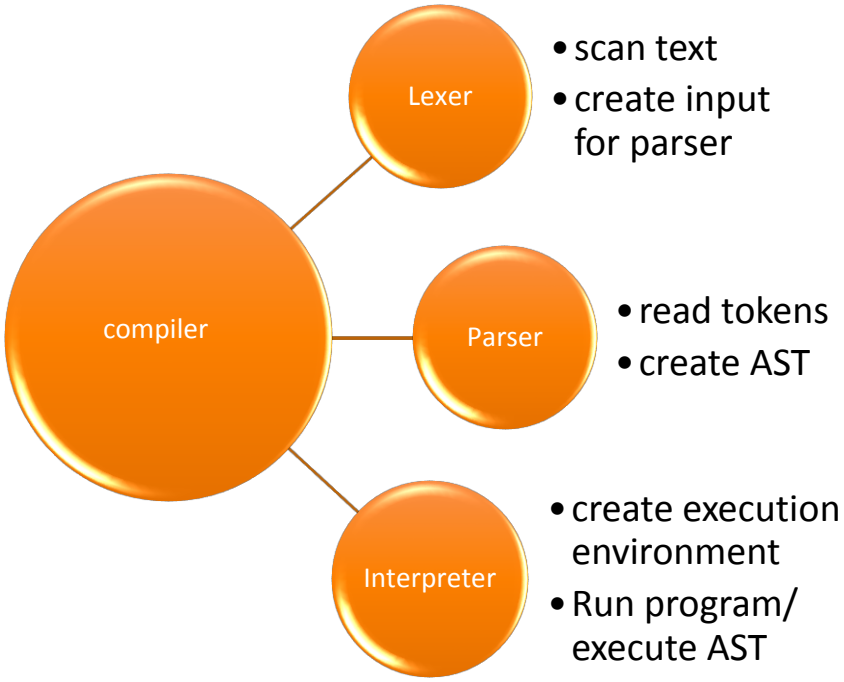
6 PROJECT STRUCTURE: MODULARITY

At this point we would direct our attention to the project structure. The project is kept modular. The modular nature of the project would make it flexible enough so that if we need an LLVM implementation in near future (when the bugs are fixed and the support is promising); it would be possible.

. Note if you are using winflexbison in Visual studio parser.y would generate header file “parser.tab.h” and the parser tool “parser.tab.cpp” as opposed to header and cpp file generation using flex and bison in a pure Unix environment. Please make appropriate imports in you generator files keeping in mind the environment they would be compiled in. Once the files are generated you would not need to re-generate them again, so from that point they become environment independent.

The compiler so far we have seen has two major modules: the lexer and the Parser. The final output from the parser is an AST. This AST is more inclined towards Bison’s generated parser. At this point we could gear up with LLVM and just walk the AST from the root node and start emitting machine code. This is relatively a simple job (given that we have a fix for the above issues). The other approach is write our own interpreter from scratch. To do this we need to convert this AST to our own AST that we design for the language. This step is also common for interpreter with LLVM so we are not wasting time and resource doing this anyway!

The first two modular components are the lexer and the parser and then we come across the third component which is the AST generator divided in files “astgen.h” and “astgen.cpp”. The fourth and the last component is the Interpreter whose functionality is divided into files “astrexec.h” and “astexec.cpp”. The following diagram would help us to grip our concepts:



7 AST GENERATION

At this point the data structure that we use to generate our AST, is defined in “astgen.h”. This file has an enum that is used in the Interpreter to lookup the type of node that it is currently processing and takes the appropriate step.

```
enum {ekId, ekNumber, ekBinExpression, ekAssignment, ekWhile, ekFunc, ekSignatures,
ekSignature, ekCall, ekStatements, ekIf, ekElseIf, ekArray, ekVector, ekVectors,
ekVector2d, ekVec1delement, ekVec2delement, ekVecAssignment, ekVec2dAssignment,
ekRtrnByExp, ekFuncAssign, ekPow, ekFact, ekAcos, ekSqrt, ekRotatez, ekMagnitudesqr,
ekTransform, ekMin, ekDot, ekCross, ekLastElement} kind;
```

The second aspect is a struct which holds all the different datatype that would be stored as node sin our AST. In the code struct AstElement represents a single node. It has a enum to set the type of the node, it can have a double value (not array of values), and a char* name and another struct element which is a complicated struct. Let us first look at example code from our project.

```
struct AstElement
{
       //this enum provides reference lookup for array search in astexec.cpp
    enum {ekId, ekNumber} kind;
    struct
    {
        double val;
        char* name;
        struct
        {
            struct AstElement *left, *right;
            char* op;
        }expression;

    }
```

Now we can extend AstElement to hold any type of node (like while, if-else, arrays, functions,etc). So the implementation can be found in “astgen.h”. Also keep in mind that we would need the enum to lookup node type and make processing decisions in our interpreter or in “astexec.cpp”

To get a detailed look of the data structure please refer to the “astgen.h” file. To add new data structure members or to create new node elements we simply have to add their representation as a struct. Let us look at an example. Suppose our AST can only create nodes for expressions so far, as shown in the code above. Now let us create a new node for statements:

```
struct AstElement
{
       //this enum provides reference lookup for array search in astexec.cpp
    enum {ekId, ekNumber, ekStatements} kind;
    struct
    {
        double val;
        char* name;
        struct
        {
            struct AstElement *left, *right;
            char* op;
```

```
        }expression;
             //create a left node here to hold data type as int or double


        struct
        {
            int count;
            //struct AstElement** statements;
                    std::vector<struct AstElement*>statements;
        }statements;

    }
```

Also note that we have made an appropriate entry in our enum to allow for the interpretor to lookup.

With this concept now, we can move on to create the meat of our AST. This is packed into a single file called the “astgen.cpp”. The functions defined here are called by our parser and uses our data structure to create the nodes internally. When a rule is matched in the parser it fires the appropriate task associated with it. For example:

```
whileStmt: TOKEN_WHILE '(' expression ')' TOKEN_DO statement{$$=makeWhile($3, $6);};
```

In the parser would call the function makeWhile in astgen.cpp:

```
struct AstElement* makeWhile(struct AstElement* cond, struct AstElement* exec)
{

    struct AstElement* result = new AstElement();
    result->kind = AstElement::ekWhile;
    result->data.whileStmt.cond = cond;
    result->data.whileStmt.statements = exec;

    return result;
}
```

So, our AST node for while is created.

At this point it is worth discussing that, if we can find a work around we would use our generated AST and convert to machine code. The conversion would occur here at “astgen.cpp” and we would write our LLVM code in the makeWhile function. That would be the end of our worries and LLVM would take care of creating an execution environment for us. As for us, we need to continue with our interpreter implementation as a workaround.

8 INTERPRETER CONSTRUCTION

The Interpreter houses the logic to walk node to node in the AST, which is other words is the execution of the program. The Interpreter houses the logic for creation of the execution environment. The execution environment is responsible to hold the current state of the AST walk (program execution) and it can be cleaned up (destroyed) after the execution.

The execution environment is kept clean and minimal to allow fast program execution. Our entire language is handled by this small structure:

```
struct ExecEnviron
{
	/*  The stack for storing Numeric values from AST execution.  */
	std::map<std::string,double>var;

	/* The stack for storing function mappings for calls later. */
	std::map<std::string,AstElement*>func;

	/* The stack for storing numeric arrays.reused as multi signature pass in function
calls and dynamic print */
	std::vector<double>arr;

	/* reference to starting name */
	std::string varName;

	/* Simple map to store PropertySet mapings to provide as check to interface above.
*/
	std::map<std::string,double>propertySet;
};
```

The stacks in the execution environment are commented to indicate the purpose they serve. Property set stack is to initiate the values passed to the program by the blue framework.

To set the propertySet stack in the execution environment we expose the following interfaces, to layers above:

```
void setPropertySet(struct ExecEnviron* e,char* varName, double val)
{
	e->var[varName] = val;
	e->propertySet[varName] = val;
}

void setPropertySet(struct ExecEnviron* e,char* varName, double val0, double val1)
{
	e->var[varName+std::to_string(0)] = val0;
	e->propertySet[varName+std::to_string(0)] = val0;

	e->var[varName+std::to_string(1)] = val1;
	e->propertySet[varName+std::to_string(1)] = val1;
}
```

The second interface is used to set the value of vectors, more specifically only a 1x2 vector in RC configuration. An example would be:

```
setPropertySet(e,"end",930.775243, 503.483954);
```

And the first interface is pretty simple to use:

```
setPropertySet(e,"endCurvature", 1.0/367);
```

To get values from the propertySet, we expose the following interface to layers above:

```
double getPropertySet(struct ExecEnviron* e, char* varName)
{
	return e->var[varName];
}
```

To use this we just

```
std::map<std::string,double> propertySet = getPropertySet(e);

std::for_each(propertySet.begin(), propertySet.end(), printPair);

void printPair(const std::pair<std::string, double > &p)
	{
		std::cout << "Name: " << p.first << "\t";
		std::cout<<"Value: "<<p.second<<std::endl;
	}
```

We also have interface to set constants for the execution environment. We use this to set the value of π. This can be extended in future to set any value of constant just after the execution environment is created. This way we just keep our constants in the execution environment and the user or the programmer (user of the compiler) does not need to worry about setting these constants. The second advantage is that it lives and dies with the execution environment, so it takes up zero static space:

```
static void initConstants(struct ExecEnviron* e, struct AstElement* a)
{
	e->var["PI"] = 3.14159265;
	e->propertySet["PI"] = 3.14159265;
}
```

We also have interfaces to create and free the execution environment:

```
struct ExecEnviron* createEnv()
{
	return new ExecEnviron();
}

void freeEnv(struct ExecEnviron* e)
{
    free(e);
}
```

This gives us the flexibility in the above layers to create and destroy execution environments dynamically:

```
struct ExecEnviron* e = createEnv();

execAst(e, astDest);
```

So from the upper layers we have the flexibility of creating and running multiple execution environments each with different settings.

And the single function that starts the magic of AST walk is:

```
void execAst(struct ExecEnviron* e, struct AstElement* a)
{
		initConstants(e,a);
	dispatchStatement(e, a);

}
```

This then calls the function:

```
/* Dispatches any AST statement execution expression */
static void dispatchStatement(struct ExecEnviron* e, struct AstElement* a)
{
    assert(a);
    assert(runExecs[a->kind]);

    runExecs[a->kind](e, a);
}
```

And we already have an idea about the lookup array from our discussions in the “astgen.h” enum

```
/* lookup array for non-value AST elements */
static void(*runExecs[])(struct ExecEnviron* e, struct AstElement* a) =
{
       NULL, /* ID and numbers are canonical and */
       NULL, /* don't need to be executed */
       NULL, /* a binary expression is not executed */
       execAssign,
       execWhile,
       execFunc,
       NULL, //reserved for execSignatures
       NULL,
       execCall,
       execStmt,
       execIf,
       execElseIf,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       execVecAssign,
       execVec2dAssign,
       execRtrnByExp,
       execFuncAssign,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL
};
```

So this looks up for the appropriate type of node and calls the associated function to handle it. The functions for handling each node is complicated but to take a look at a simple one let us look at:

```
//responsible for storing the function address in the map.. execution should be done in a
separate map
static void execFunc(struct ExecEnviron* e, struct AstElement* a)
{
       assert(a);
```

```
        assert(AstElement::ekFunc == a->kind);
        std::stringstream ss;
        ss << (a->data.func.signatures);
        char* fname = (a->data.func.name);

        e->func[fname]= a;

}
```

The function is stored in a stack along with its name and signature. This gives us the possibility to do polymorphism in our language. To get a clear understanding of the entire program just look at few other node functions.

The most important node creating process is:

```
static int execBinExp(struct ExecEnviron* e, struct AstElement* a)
```

The function is commented at various levels make sure to go through them and have a clear understanding of how this function works before moving to other functions.

The Interpreter has three different types of array lookups, we have seen the first type the rest two are:

```
/* Lookup Array for AST elements which yields values */
static int(*valExecs[])(struct ExecEnviron* e, struct AstElement* a) =
{
        execTermExpression,
        execTermExpression,
        execBinExp,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        execVec1dEl,
        execVec2dEl,
        NULL,
        NULL,
        NULL,
        NULL,
        execPow,
        execFact,
        execAcos,
        execSqrt,
        execRotatez,
        execMagnitudesqr,
        execTransform,
```

```
        execMin,
        execDot,
        execCross
};

static int (*arrExecs[])(struct ExecEnviron* e, struct AstElement* a) =
{
        NULL, /* ID and numbers are canonical and */
        NULL, /* don't need to be executed */
        NULL, /* a binary expression is not executed */
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        execArray,
        execVector,
        execVectors,
        execVector2d,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
};
```

Though the arrExecs looks like we could have merged it with runExecs lookup array, we decided to keep this structure separate if in future we decide to support multidimensional matrices.

The Interpreter also handles the inbuilt functions. The inbuilt functions are treated as one single node by the AST and the interpreter executes this node as an inbuilt function. Let us look at the inbuilt function to get arc cosine:

```
static int execAcos(struct ExecEnviron* e, struct AstElement* a)
{
        int length = dispatchExpression(e,a->data.Acos.expr);
        double val;
        std::string temp = "Temp";

        if(length == 1)
        {
```

```
                val = e->var[e->varName];
        }
        else
        {
                std::cout<<"execAcos: Unsupported Math operation on vectors!"<<std::cout;
        }


        if(length == 1)
        {
                e->var[temp] = ( std::acos(val));
                e->varName = temp;
                return 1;
        }
        else
        {
                std::cout<<"execAcos: Unsupported Math operation on vectors!"<<std::endl;
        }

        return NULL;
}
```

This brings us to the end of our discussion about the interpreter. For the reader if it still does not make sense about how the entire project works right from the moment when we pass on our program as a text file, the next section is specifically aimed at that!

9 SEWING IT ALL TOGETHER

So far we have seen the bare-bones for our compiler. We would now look at an example and understand how to extend/add features to our compiler. Our compiler, as discussed in the previous sections has three main logical components:

1. The lexer generator (or the lexer.l file). The lexer is in charge of parsing tokens from the program we provide as input. The lexer identifies these tokens based on rules set in it and does appropriate transformations to the scanned tokens in memory and passes as input for the parser
2. The parser generator (or the parser.y file). The parser reads the tokens provided by the lexer and based on the grammar rules set in it, takes appropriate actions or creates the AST for the interpreter.
3. The interpreter (the astexec.cpp file) walks the AST from the root node or in other words executes the program.

The astgen files are the AST generators which help the parser to convert scanned to tokens to system defined AST based on the grammar rules. The astgen.h file holds the data structure for the AST generation.

Suppose we would like to make a change and add support for a new inbuilt function which returns the dot product of vectors.
We would start with the lexer as it is the first in line to scan the input text. We need to reserve a keyword for the scanner/lexer so that it recognizes a dot product operation when it sees one. So in the lexer.l file add the following rule:

```
"dot" return TOKEN_DOT;
```

This would tell the lexer to replace the keyword dot with TOKEN_DOT in memory whenever it is scanned in text. Next we need to add a rule in the parser. As this is going to be an inbuilt function a good approach would be to add it as an expression rule:

```
expression: TOKEN_DOT '(' expression TOKEN_COMMA expression ')'{$$=makeDot($3,$5);};
```

The parser would come across a co match the rule and make call to makeDot. Now we need our makeDot to create that AST element for us. We first create the data structure for holding necessary:

```
struct
		{
			struct AstElement* left;
			struct AstElement* right;
		}dot;
```

Now astgen.cpp is called and we generate the AST element as per the following code:

```
struct AstElement* makeDot( struct AstElement* left,struct AstElement* right)
{
	struct AstElement* result = new AstElement();
	result->kind = AstElement::ekDot;
	result->data.dot.left = left;
	result->data.dot.right = right;
	return result;
   }
```

Now with the AST element in place we have to define appropriate functions for the interpreter to calculate the dot product. We add the following to the astexec.cpp:

```
static int execDot(struct ExecEnviron* e, struct AstElement* a)
{

	int lengthRight = dispatchExpression(e,a->data.dot.right);
	double b[2];
	double c[2];
	double sum = 0;
	std::string varNameRight = e->varName;
	int lengthLeft = dispatchExpression(e,a->data.dot.left);
	std::string varNameLeft = e->varName;
```

```
	std::string temp = "Temp";
	if(lengthRight==1 && lengthLeft==1)
	{
		std::cout<<"execDot: Dot product operation undefined for non array type
elements"<<std::endl;
	}
	else if( e->var.find(varNameRight+std::to_string(0)) != e->var.end() )
	{
		//found 1d array. Names with Sample0 Sample1. Copy array to temp array
		for(int i=0;i<lengthRight;i++)
		{
			b[i] = e->var[(varNameRight+std::to_string(i))]  ;
			c[i] = e->var[(varNameLeft+std::to_string(i))]  ;

			sum += (b[i]*c[i]);
		}

		//e->var[temp] = std::inner_product(b, b + sizeof(b) / sizeof(b[0]), c, 0);
		e->var[temp]=sum;
		e->varName = temp;

		return 1;
	}
	else
	{
		std::cout<<"execDot: Dot product operation undefined for 2D array type
elements"<<std::endl;
	}
	return NULL;
}
```

As we add the execution logic for our inbuilt function in out C++ interpreter, note that we have to also make an appropriate entry at the lookup tables for the interpreter. Notice that our execution pushes its computed value to the stack after successful execution. This means we need to add this to the lookup table which supports int type returns.

```
/* Lookup Array for AST elements which yields values */
static int(*valExecs[])(struct ExecEnviron* e, struct AstElement* a) =
{
	execDot,

};
```

The entry must be made at the right place of the lookup array. Please refer to the section above for a complete representation of the lookup arrays.

This lookup array would be referred by our interpreter as it walks from one AST element to the other. At each step of the AST walk it refers to this array and finds the appropriate method to execute for that particular AST element.

Now we have an understanding about all the parts and how they work. The last two sections provide a different representation of the grammar and lists down all supported tokens in the language.

10 COMPLETE LIST OF TOKENS SUPPORTED

Token (in Program)	Internal Representation
while	TOKEN_WHILE
!= , <= , >= , && , || , ==	TOKEN_OPERATOR
{	TOKEN_BEGIN
}	TOKEN_END
[	BOX_OPEN
]	BOX_CLOSE
func	BEGIN_FUNC
do	TOKEN_DO
if	TOKEN_IF
else	TOKEN_ELSE
vector	TOKEN_VECTOR
vector2d	TOKEN_VECTOR2d
return	TOKEN_RETURN
pow	TOKEN_POW
factorial	TOKEN_FACTORIAL
acos	TOKEN_ACOS
sqrt	TOKEN_SQRT
rotatez	TOKEN_ROTATEZ
magnitude_squared	TOKEN_MAGNITUDESQR
transform	TOKEN_TRANSFORM
min	TOKEN_MIN
dot	TOKEN_DOT
cross	TOKEN_CROSS
[a-zA-Z_][a-zA-Z0-9_]*	TOKEN_ID
[-]?[0-9]*[.]?[0-9]+	TOKEN_NUMBER
* , /, +, -, <, >, %	TOKEN_OPERATOR

11 THE GRAMMER IN CLASSIC BNF NOTATION

```
<Program> ::= <statement> “;”

<Statements>  ::=  <statements> <statement> “;”
              | <statements> <block> “;”

<Block> ::= <TOKEN_BEGIN> <statements> <TOKEN_END>


<statement>   ::= <whileStmt>
              | <ifStmt>
              | <block>
              | <call>
```

```
              | <func>
              | <vector>
              | <vector2d>
              | <return>
              | <assignment>


<Assignment>  ::= <TOKEN_ID> “=” <expression>
              | <TOKEN_ID>
              | <TOKEN_ID> <BOX_OPEN> <TOKEN_NUMBER> <BOX_CLOSE> “=” <expression>
              | <TOKEN_ID> <BOX_OPEN> <TOKEN_NUMBER> <BOX_CLOSE> <BOX_OPEN>
              <TOKEN_NUMBER>  <BOX_CLOSE> “=” <expression>
              | <TOKEN_ID> “=” <call>


<Expression>  ::= <TOKEN_ID>
              | <TOKEN_NUMBER>
              | <TOKEN_OPERATOR> <expression>
              | <expression> <TOKEN_OPERATOR> <expression>
              | <TOKEN_ID> <BOX_OPEN> <TOKEN_NUMBER> <BOX_CLOSE>
              | <TOKEN_ID> <BOX_OPEN> <TOKEN_NUMBER> <BOX_CLOSE> <BOX_OPEN> <TOKEN_NUMBER>
              <BOX_CLOSE>
              | <TOKEN_POW> “(“ <expression> <TOKEN_COMMA> <expression> “)”
              | <TOKEN_FACTORIAL> “(“ <expression> “)”
              | <TOKEN_ACOS> “(“ <expression> “)”
              | <TOKEN_SQRT> “(“ <expression> “)”
              | <TOKEN_ROTATEZ> “(“ <expression> <TOKEN_COMMA> <expression> “)”
              | <TOKEN_MAGNITUDESQR> “(“ <expression> “)”
              | <TOKEN_MIN> “(“ <expression> <TOKEN_COMMA> <expression> <TOKEN_COMMA> <
              expression> “)”
              | <TOKEN_DOT> “(“ <expression> <TOKEN_COMMA> <expression> “)”
              | <TOKEN_CROSS> “(“ <expression> <TOKEN_COMMA> <expression> “)”
              | <TOKEN_TRANSFORM> “(“ <expression> “)”
              | “(“ <expression> <TOKEN_OPERATOR> <expression> “)”


<arrExpression> ::= <TOKEN_ID>
                 | <TOKEN_NUMBER>
                 | <arrExpression> <TOKEN_OPERATOR> <arrExpression>


<array> ::= <array> <TOKEN_COMMA> <arrExpression>
        | <array> <arrExpression>


<vector2d> ::= <TOKEN_VECTOR2d> <TOKEN_ID> “=” <BOX_OPEN> <vectors> <BOX_CLOSE>
           | <TOKEN_VECTOR2d> <TOKEN_ID> “=” <arrExpression>



<vectors> ::= <vectors TOKEN_COMMA '(' array ')'
          | vectors '(' array ')'
          | vectors '('  ')'

<whileStmt> ::= <TOKEN_WHILE> “(“ <expression> “)” <TOKEN_DO> <statement>
```

```
<ifStmt> ::= <TOKEN_IF>  “(“ <expression> “)” <TOKEN_DO> <statement>
         | <TOKEN_IF>  “(“ <expression> “)” <TOKEN_DO> <statement> <TOKEN_ELSE>
         <TOKEN_DO> <statement>
         | <TOKEN_IF>  “(“ <expression> “)” <TOKEN_DO> <statement> <TOKEN_ELSE>
         <TOKEN_IF> “(“ <expression> “)” <TOKEN_DO> <statement> <TOKEN_ELSE> <TOKEN_DO>
         <statement>


<func> ::= <BEGIN_FUNC> <TOKEN_ID> “(“  “)” <statement>
       | <BEGIN_FUNC> <TOKEN_ID> “(“ <signatures> “)” <statement>


<Signatures> ::= <signature> <signatures>


<Signature> ::= <TOKEN_COMMA> <TOKEN_ID> <assignment>
         |<TOKEN_ID> <assignment>

<call> ::= <TOKEN_ID> “(“ <array> “)”

<Return> ::= <TOKEN_RETURN> <expression>
```