RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Assignment

Course Code	CSC310A
Course Name	Compilers
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No.	17ETCS002159
Semester/Year	06/2020
Course Leader(s)	Ms. K S Suvidha

Declaration Sheet

Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	06/2020
Course Code	CSC310A		
Course Title	Compilers		
Course Date		to	
Course Leader	Ms. K S Suvidha		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

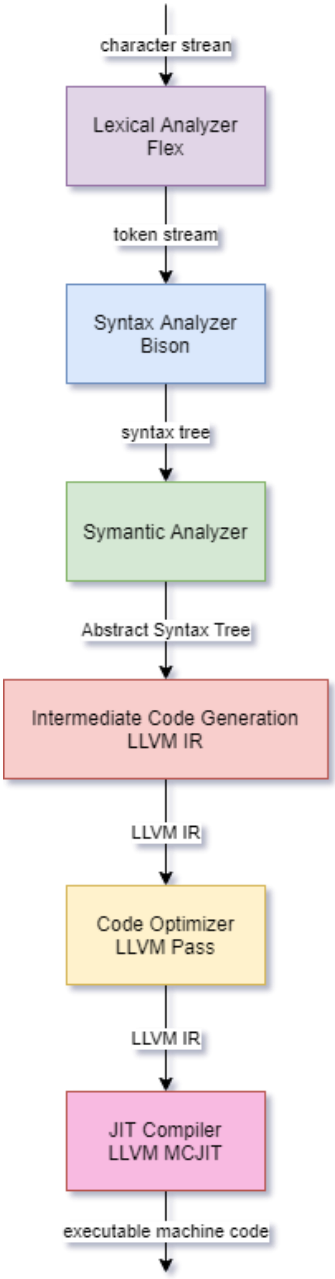
List Of Figures

1 Question A

Solution to Question A

1.1 Introduction

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers. [5]



The assignment is to build such a compiler, to do this we use several tools as shown below,

Figure 1-1 Compiler Recipe

Flex: Flex is a tool for generating scanners, programs which recognized lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate.

Bison: Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1), IELR(1) or canonical LR(1) parser tables.

LLVM: In an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR (usually, but not always, by building an AST and then converting the AST to LLVM IR). This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code, as shown in Figure 1-2 LLVM Optimizer. This is a very straightforward implementation of the three-phase design, but this simple description glosses over some of the power and flexibility that the LLVM architecture derives from LLVM IR. [1]

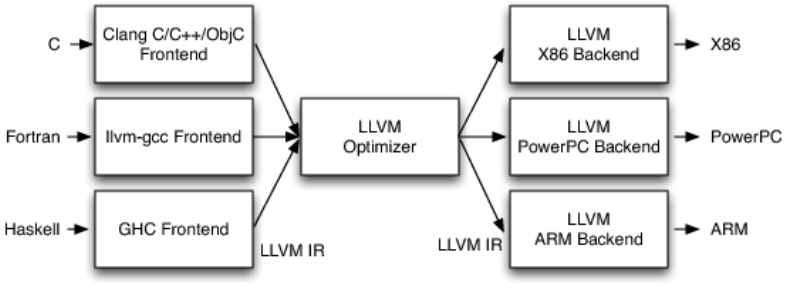


Figure 1-2 LLVM Optimizer

1.2 Identification and grouping of Tokens

1.2.1 Keywords

TOKEN	FOR	IN	RANGE	IF	ELSE
RE	for	in	range	if	else

1.2.2 Operators

Arithmetic Operators

TOKEN	PLUS	MINUS	MUL	DIV	ASSIGN
RE	+	-	*	/	=

Comparison Operators

TOKEN	GRT	GRTEQ	LES	LESEQ	NOTEQ	EQUAL
RE	>	>=	<	<=	!=	==

Boolean Operators

TOKEN	AND	OR	NOT
RE	and	or	not

1.2.3 Special Symbols

TOKEN	LPAREN	RPAREN	LBRACE	RBRACE	LBRACKET	RBRACKET	COMMA
RE	(	)	{	}	[	]	,

TOKEN	EOF
RE	<<eof>>

1.2.4 Literals

TOKEN	DECIMAL	FRACTION	STRING
RE	-?[0-9]+	-?[0-9]+\.[0-9]*	\'([^\\\"]|\\.)*\'

Note: The RE for STRING defined here does not include the newlines and other escape sequences, a DFA was created in lex for doing so, refer to implementation in lex.

1.2.5 Identifier

TOKEN	IDENTIFIER
RE	[a-zA-Z_][a-zA-Z_0-9]*

1.3 Implementation in Lex

tokens.l

```
%{
    #include <string>
    #include <cerrno>
    #include <climits>
    #include <cstdlib>
    #include <cstring> // strerror

    #include "driver/driver.hpp"
    #include "parser.hpp"
    #include "ast/ast_structures.hpp"

    // temporary for storing the string literal
    std::string g_str;

%}

%option noyywrap nounput noinput batch

%x str
%s normal

%{
    yy::parser::symbol_type make_DECIMAL(const std::string& s, const yy::parser::location_type
& loc);
    yy::parser::symbol_type make_FRACTION(const std::string& s, const yy::parser::location_typ
e& loc);
```

```
    yy::parser::symbol_type make_IDENT(const std::string& s, const yy::parser::location_type&
loc);
%}

ident       [a-zA-Z_][a-zA-Z_0-9]*
num         [0-9]
blank       [ \t\r]

%{
    // runs each time a pattern is matched
    #define YY_USER_ACTION  loc.columns(yyleng);
%}

%%

%{
    yy::location& loc = drv.location;
    loc.step();
%}
 //<- one leading blank space; comments should start one blank space after
 /* state automata for string literal */

'                   { g_str = ""; BEGIN(str); } /*eat '*/
<str>\'             { BEGIN(normal); return yy::parser::make_STRINGLIT(std::make_unique<string
lit>(g_str, loc), loc); }
<str>\\n            g_str += "\n";
<str>\\t            g_str += "\t";
<str>\\r            g_str += "\r";
<str>\\\'           g_str += "'";
<str>\\(.|\n)       g_str += yytext[1];
<str>[^\\']+        g_str += std::string(yytext);

{blank}+            { loc.step(); }
\n+                 { loc.lines(yyleng); loc.step(); }
"and"               { return yy::parser::make_AND(loc); }
"or"                { return yy::parser::make_OR(loc); }
"not"               { return yy::parser::make_NOT(loc); }
"if"                { return yy::parser::make_IF(loc); }
"else"              { return yy::parser::make_ELSE(loc); }
"for"               { return yy::parser::make_FOR(loc); }
"in"                { return yy::parser::make_IN(loc); }
"range"             { return yy::parser::make_RANGE(loc); }
"+"                 { return yy::parser::make_PLUS(loc); }
"-"                 { return yy::parser::make_MINUS(loc); }
"*"                 { return yy::parser::make_MUL(loc); }
"/"                 { return yy::parser::make_DIV(loc); }
"="                 { return yy::parser::make_ASSIGN(loc); }
">"                 { return yy::parser::make_GRT(loc); }
">="                { return yy::parser::make_GRTEQ(loc); }
```

```
"<"                 { return yy::parser::make_LES(loc); }
"<="                { return yy::parser::make_LESEQ(loc); }
"!="                { return yy::parser::make_NOTEQ(loc); }
"=="                { return yy::parser::make_EQUAL(loc); }
{num}+\.{num}*      { return make_FRACTION(yytext, loc); }
-?{num}+            { return make_DECIMAL(yytext, loc); }
{ident}             { return make_IDENT(yytext, loc); }
"("                 { return yy::parser::make_LPAREN(loc); }
")"                 { return yy::parser::make_RPAREN(loc); }
"{"                 { return yy::parser::make_LBRACE(loc); }
"}"                 { return yy::parser::make_RBRACE(loc); }
"["                 { return yy::parser::make_LBRACKET(loc); }
"]"                 { return yy::parser::make_RBRACKET(loc); }
","                 { return yy::parser::make_COMMA(loc); }
#.*                 /* eat everything; single line comment */
.                   { throw yy::parser::syntax_error
                      (loc, "invalid character: " + std::string(yytext));
                    }
<<EOF>>             { return yy::parser::make_END(loc); }

%%

yy::parser::symbol_type make_DECIMAL(const std::string& s, const yy::parser::location_type& l
oc) {
  std::unique_ptr<decimal> temp = std::make_unique<decimal>(std::strtoll(yytext, NULL, 10), lo
c);
  return yy::parser::make_DECIMAL(std::move(temp), loc);
}

yy::parser::symbol_type make_FRACTION(const std::string& s, const yy::parser::location_type&
loc) {
  std::unique_ptr<fraction> temp = std::make_unique<fraction>(std::strtold(yytext, NULL), loc)
;
  return yy::parser::make_FRACTION(std::move(temp), loc);
}

yy::parser::symbol_type make_IDENT(const std::string& s, const yy::parser::location_type& loc
) {
  std::unique_ptr<identifier> temp = std::make_unique<identifier>(s, loc);
  return yy::parser::make_IDENT(std::move(temp), loc);
}

// code from bison manual: https://www.gnu.org/software/bison/manual/html_node/Calc_002b_002b-
Scanner.html

void driver::scan_begin() {
  if (file.empty() || file == "stdin")
    yyin = stdin;
  else if (!(yyin = fopen(file.c_str(), "r"))) {
```

```
      std::cerr << "cannot open " << file << ": " << strerror(errno) << '\n';
      exit (EXIT_FAILURE);
    }
}

void driver::scan_end() {
  fclose(yyin);
}
```

driver.hpp

```
#pragma once

#include <map>
#include <string>
#include "parser.hpp"

// declare the YY_DECL as our custom parser driver
#define YY_DECL yy::parser::symbol_type yylex(driver& drv)

YY_DECL;

class driver {
   public:
    driver();

    std::map<std::string, int> variables;

    int result;

    // to run the parser on a given file
    int parse(const std::string& f);

    // name of the file being parsed
    std::string file;

    // handling the scanner
    // NOTE: defined in tokens.l
    void scan_begin();
    void scan_end();

    // token location
    yy::location location;
};
```

driver.cpp

```
#include "driver.hpp"

#include "parser.hpp"
```

```
driver::driver() { }

int driver::parse(const std::string& f) {
    file = f;
    location.initialize(&file);

    // scan_begin and scan_end are defined in tokens.l
    scan_begin();

    yy::parser parse(*this);

    // int res =
    parse();

    scan_end();

    // return res;
    return 0;
}
```

Note: For the header files and other sources, please refer to Appendix B

1.4 Design of Context Free Grammar

```
program     : stmts

stmts       : stmt
            | stmts stmt

stmt        : expr
            | var_decl
            | conditional
            | for_loop
            | for_range

for_loop    : "for" "(" expr "," expr "," expr ")" block

for_range   : "for" identifier "in" "range" "decimal" block

block       : "{" stmts "}"

conditional     : "if" expr block "else" block
                | "if" expr block

var_decl    : "identifier" "identifier"
```

```
            | "identifier" "identifier" "=" expr

literals    : "decimal"
            | "fraction"
            | "stringlit"

expr        : identifier "=" expr
            | identifier "(" call_args ")"
            | identifier
            | literals
            | binop_expr
            | unaryop_expr
            | compare_expr
            | array_access
            | "(" expr ")"

call_args   : /*blank*/
            | expr
            | call_args[arg] "," expr

array_access    : identifier "[" expr "]"
                | array_access "[" expr "]"

binop_expr  : expr "and" expr
            | expr "or" expr
            | expr "+" expr
            | expr "-" expr
            | expr "*" expr
            | expr "/" expr

compare_expr    :   expr ">" expr
                |   expr ">=" expr
                |   expr "<" expr
                |   expr "<=" expr
                |   expr "==" expr
                |   expr "!=" expr

unaryop_expr    : "not" expr
```

Minimum two data types:

- `decimal`
- `fraction`

Minimum two control statements:

- `if`

- `else`

Minimum two looping statements:

- `for`
- `for i in range`

Input-output functions:

- `display`
- `read`

Compound statements and two-dimensional Array:

- `{ block }`
- `array[idx]`
- `array[idx][jdx]`
- `array[idx][jdx][kdx]`

1.5 Implementation in Yacc

```
%skeleton "lalr1.cc"
%require "3.5"
%language "c++"

%defines

// variant will make sure we can use our non-trivial types
%define api.value.type variant
%define api.token.constructor
%define parse.assert

// this will be added to the parser.cpp file, cyclic-dependecy is resolved by using
// forward declaration of the driver class, this is added verbatim
// if you want to declare any variables do not do in this requires section
%code requires {
    #include <string>
    #include <memory>
    #include <typeinfo>

    class driver;

    #include "ast/ast_structures.hpp"
    // love you c++ gods, g++ gave me much help in debugging
    // <3
    #include "visitor/visitor.hpp"
    #include "visitor/visitor_pprint.hpp"
```

```
    #include "external/loguru.hpp"

    static int cnt = 0;
}

// parsing context
%param { driver& drv }

// for location tracking
%locations
%verbose

// because we'll be using the driver class methods
%code {
    #include "driver/driver.hpp"

    std::shared_ptr<block> program_block;

    visitor_pprint v_pprint;
}

// to make sure there are no conflicts prepend TOK_
%define api.token.prefix{TOK_}
%token
    END  0  "end of file"
    AND     "and"
    OR      "or"
    NOT     "not"
    FOR     "for"
    IN      "in"
    RANGE   "range"
    IF      "if"
    ELSE    "else"
    ASSIGN  "="
    PLUS    "+"
    MINUS   "-"
    MUL     "*"
    DIV     "/"
    LPAREN  "("
    RPAREN  ")"
    LBRACE  "{"
    RBRACE  "}"
    LBRACKET "["
    RBRACKET "]"
    COMMA   ","
    GRT     ">"
    GRTEQ   ">="
    LES     "<"
    LESEQ   "<="
```

```
    NOTEQ   "!="
    EQUAL   "=="

%token  <std::unique_ptr<identifier>>   IDENT       "identifier"
%token  <std::unique_ptr<decimal>>      DECIMAL     "decimal"
%token  <std::unique_ptr<fraction>>     FRACTION    "fraction"
%token  <std::unique_ptr<stringlit>>    STRINGLIT   "stringlit"
%nterm  <std::unique_ptr<identifier>>   identifier // add this for verbosity
%nterm  <std::unique_ptr<expression>>   expr
%nterm  <std::unique_ptr<expression>>   literals
%nterm  <std::unique_ptr<expression>>   binop_expr
%nterm  <std::unique_ptr<expression>>   unaryop_expr
%nterm  <std::unique_ptr<expression>>   compare_expr
%nterm  <std::unique_ptr<block>>        stmts
%nterm  <std::unique_ptr<block>>        program
%nterm  <std::unique_ptr<block>>        block
%nterm  <std::unique_ptr<statement>>    stmt
%nterm  <std::unique_ptr<statement>>    conditional
%nterm  <std::unique_ptr<statement>>    for_loop
%nterm  <std::unique_ptr<statement>>    for_range
%nterm  <std::unique_ptr<std::vector<std::unique_ptr<expression>>>> call_args
%nterm  <std::unique_ptr<variable_declaration>> var_decl
%nterm  <std::unique_ptr<array_access>> array_access

%printer { yyo << $$; } <*>;

%start program;

%code {
    #define DEBUG_PARSER
    #undef DEBUG_PARSER
}

%%

// left associativity

%left "+" "-";
%left "*" "/";

// program consists of statements

program     : stmts {

                program_block = std::move($1);
                program_block->accept(v_pprint);

                }
            ;
```

```
// statements can consist of single or multiple statements

stmts[block]       : stmt {

                $block = std::make_unique<block>();

                $block->statements.emplace_back(std::move($1));
                $$->accept(v_pprint);

                }
            | stmts[meow] stmt {

                $meow->statements.emplace_back(std::move($2));

                // i added this because i std::move everytime and this moves the $block also
                // so i std::move back $meow to block to retain the address of main block
                // it was becoming null before, added null check in main.cpp as well
                // - shadowleaf

                $block = std::move($meow);

                }
            ;

// statement can be an expression or an variable declaration

stmt        : expr {
                $$ = std::make_unique<expr_statement>(std::move($1));
                $$->accept(v_pprint);
                }
            | var_decl {
                $$ = std::move($1);
                }
            | conditional {
                $$ = std::move($1);
                 }
            | for_loop {
                $$ = std::move($1);
                $$->accept(v_pprint);
                }
            | for_range {
                $$ = std::move($1);
                $$->accept(v_pprint);
            }

            ;

// for loops
```

```
for_loop    : "for" "(" expr "," expr "," expr ")" block {
                $$ = std::make_unique<for_loop>(std::move($3), std::move($5), std::move($7), s
td::move($9));
            }
            ;

for_range   : "for" identifier "in" "range" "decimal" block {
                $$ = std::make_unique<for_range>(std::move($2), std::move($5), std::move($6));
            }
            ;

// a block

block       : "{" stmts "}" {
                    $$ = std::move($2);
                    $$->accept(v_pprint);
                    }
            ;

// conditional statement

conditional     : "if" expr block "else" block {
                    $$ = std::make_unique<conditional>(std::move($2), std::move($3), std::move
($5));
                    $$->accept(v_pprint);
                    }
                | "if" expr block {
                    $$ = std::make_unique<conditional>(std::move($2), std::move($3));
                    $$->accept(v_pprint);
                }
                ;

// variable declaration and/or assignment

var_decl    : "identifier" "identifier" {
                $$ = std::make_unique<variable_declaration>(std::move($1), std::move($2));
                $$->accept(v_pprint);
            }
            | "identifier" "identifier" "=" expr {
                $$ = std::make_unique<variable_declaration>(std::move($1), std::move($2), std:
:move($4));
                $$->accept(v_pprint);
            }
            ;

// all the literals, like integers, fractions and string literals

literals    : "decimal"  {
```

```
                $$ = std::move($1);
                // LOG_S(INFO) << "found decimal at " << @1.begin.line << "." << @1.begin.colu
mn;
                $$->accept(v_pprint);

                }
            | "fraction" {

                $$ = std::move($1);
                $$->accept(v_pprint);

                }
            | "stringlit" {

                $$ = std::move($1);
                $$->accept(v_pprint);

            }
            ;

// all the expression statements

expr        : identifier "=" expr {

                $$ = std::make_unique<assignment>(std::move($1), std::move($3));
                $$->accept(v_pprint);

                }

            | identifier "(" call_args ")" {
                // function call

                $$ = std::make_unique<function_call>(std::move($1), std::move($3));
                $$->accept(v_pprint);

                }
            | identifier {
                // just an identifier

                $$ = std::move($1);
                $$->accept(v_pprint);

                }

            | literals {

                // literal, either decimal or fractional
```

```
                $$ = std::move($1);

                }
            | binop_expr {

                // some binary operation (numeric, not boolean)

                $$ = std::move($1);
                $$->accept(v_pprint);
                }
            | unaryop_expr {
                // a and or not, unary boolean expression

                $$ = std::move($1);
                }
            | compare_expr {
                // a comparison expression

                $$ = std::move($1);
                }
            | array_access {
                // accessing an element of array

                $$ = std::move($1);
                }
            | "(" expr ")" {
                $$ = std::move($2);
                $$->accept(v_pprint);
                }
            ;

identifier  : "identifier" { $$ = std::move($1); $$->accept(v_pprint); }

// call arguments of a function
// can be blank

call_args[args_list]   : /*blank*/ {
                $args_list = std::make_unique<std::vector<std::unique_ptr<expression>>>();
                }
            | expr {
                $args_list = std::make_unique<std::vector<std::unique_ptr<expression>>>();
                $args_list->push_back(std::move($1));
                }
            | call_args[arg] "," expr {
                $arg->push_back(std::move($3));
                $args_list = std::move($arg);
                }
            ;
```

```
// array access for arr[0], arr[<some expr that evaluate to decimal>]
// or for the future can also be arr['string'] for maps
array_access    : identifier "[" expr "]" {
                    $$ = std::make_unique<array_access>(std::move($1), std::move($3));
                    $$->accept(v_pprint);
                    }
                | array_access "[" expr "]" {
                    $$ = std::make_unique<array_access>(std::move($1), std::move($3));
                    $$->accept(v_pprint);
                }
                ;

// binary operators

binop_expr  : expr "and" expr {

                $$ = std::make_unique<binary_operator>('&', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);

                }
            | expr "or" expr {

                $$ = std::make_unique<binary_operator>('|', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);

                }
            | expr "+" expr {
                $$ = std::make_unique<binary_operator>('+', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);

                }
            | expr "-" expr {
                $$ = std::make_unique<binary_operator>('-', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);
                }
            | expr "*" expr {
                $$ = std::make_unique<binary_operator>('*', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);
                }
            | expr "/" expr {
                $$ = std::make_unique<binary_operator>('/', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);
                }
            ;

// binary boolean comparison operators
compare_expr    :   expr ">" expr {
                        $$ = std::make_unique<comp_operator>(">", std::move($1), std::move($3)
);
```

```
                        $$->accept(v_pprint);
                    }
                |   expr ">=" expr {
                        $$ = std::make_unique<comp_operator>(">=", std::move($1), std::move($3
));
                        $$->accept(v_pprint);
                    }
                |   expr "<" expr {

                        $$ = std::make_unique<comp_operator>("<", std::move($1), std::move($3)
);
                        $$->accept(v_pprint);
                    }
                |   expr "<=" expr {

                        $$ = std::make_unique<comp_operator>("<=", std::move($1), std::move($3
));
                        $$->accept(v_pprint);
                    }
                |   expr "==" expr {

                        $$ = std::make_unique<comp_operator>("==", std::move($1), std::move($3
));
                        $$->accept(v_pprint);
                    }
                |   expr "!=" expr {

                        $$ = std::make_unique<comp_operator>("!=", std::move($1), std::move($3
));
                        $$->accept(v_pprint);
                }
                ;

// unary operations

unaryop_expr    : "not" expr {
                    $$ = std::make_unique<unary_operator>('!', std::move($2), @$);
                    $$->accept(v_pprint);
                    }
                ;

// // boolean expression

// boolean_expr    : expr "and" expr {

//                     }
//                 | expr "or" expr {

//                     }
```

```
//                 | expr "xor" expr {

//                     }

/* testing out a grammar */
/*
program     : expr { std::cout << "expr: " << cnt++ << "\n"; }
            ;

expr        : "decimal" { std::cout<< "decimal: " << cnt++ << "\n"; $$ = std::move($1); }
            | expr "+" expr { std::cout << "expr + expr: " << cnt++ << "\n"; $$ = std::make_un
ique<binary_operator>('+', std::move($1), std::move($3)); }
            ;
*/
%%

void yy::parser::error (const location_type& l, const std::string& m) {
  std::cerr << l << ": " << m << '\n';
}
```

1.6 Testing

To test the grammar various test cases were made, the program made for the compiler can also generate IR code and then use the LLVM MCJIT (Machine Code Just In Time) Compiler to execute the generated IR.

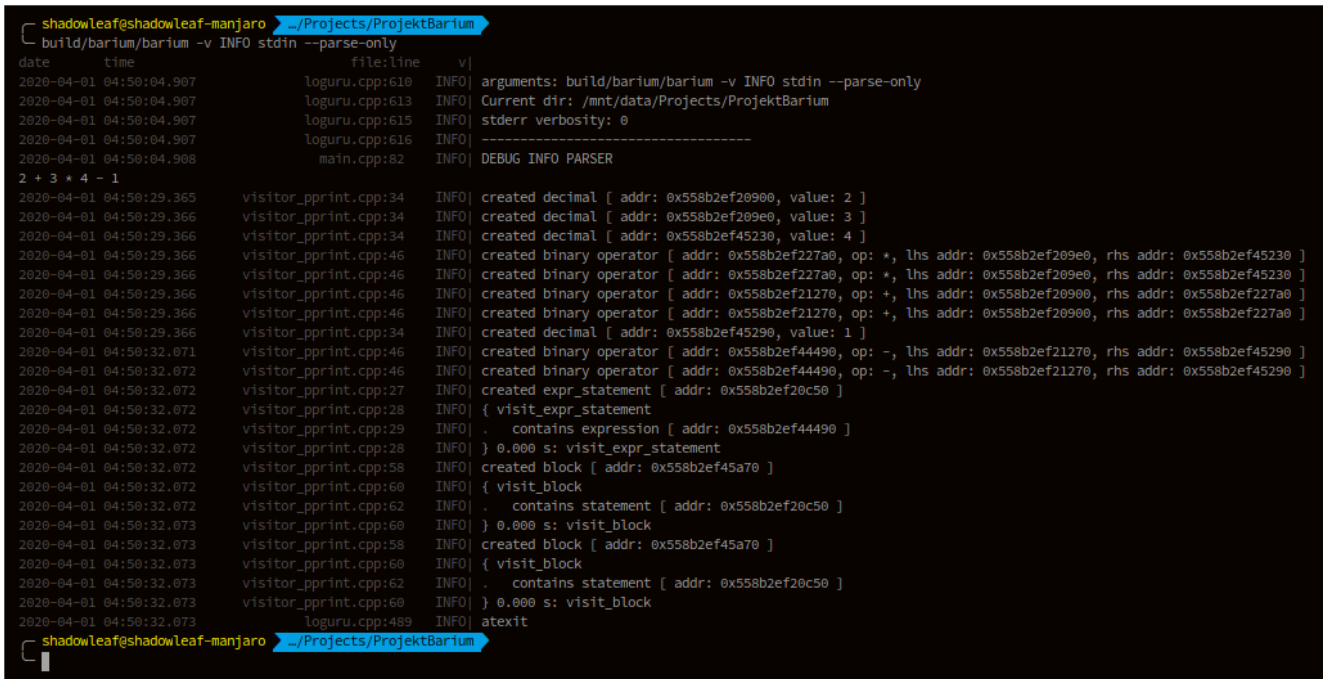


Figure 1-3 Parsing Simple Arithmetic Expression

Input `2 + 3 * 4 - 1` is given to the program, first the lexical analyzer converts this character stream to tokens, i.e. `DECIMAL, PLUS, DECIMAL, MUL, DECIMAL, MINUS, DECIMAL`, as defined by our tokens earlier.

These tokens are fed to the scanner, which uses the grammar rules that we provided to perform actions on matching the syntax of the tokens. For example, when 2 + 3 is found, expr + expr is matched and then into a binary operator. Similarly, it is done for the entire program.

The Abstract Syntax Tree is thus generated, while doing so we print the nodes of the tree, which can be seen in the terminal as form of LOG INFO, the address of the node and its contents are printed, we can use this information to create a visual syntax tree, so it'll be easier for us to comprehend. Refer to Figure 1-4 AST for 2 + 3 * 4 - 1, we can see how our expression is converted to a AST, the leaf nodes are the terminals, which are all decimals in our case, few reduce operations are omitted, for optimizations, like decimal is reduced to expr and then attached to binary_operator, but bison optimizes this and directly does the reduction operation.

From the figure, we can see that operator associativity and precedence is maintained as written in the grammar file.

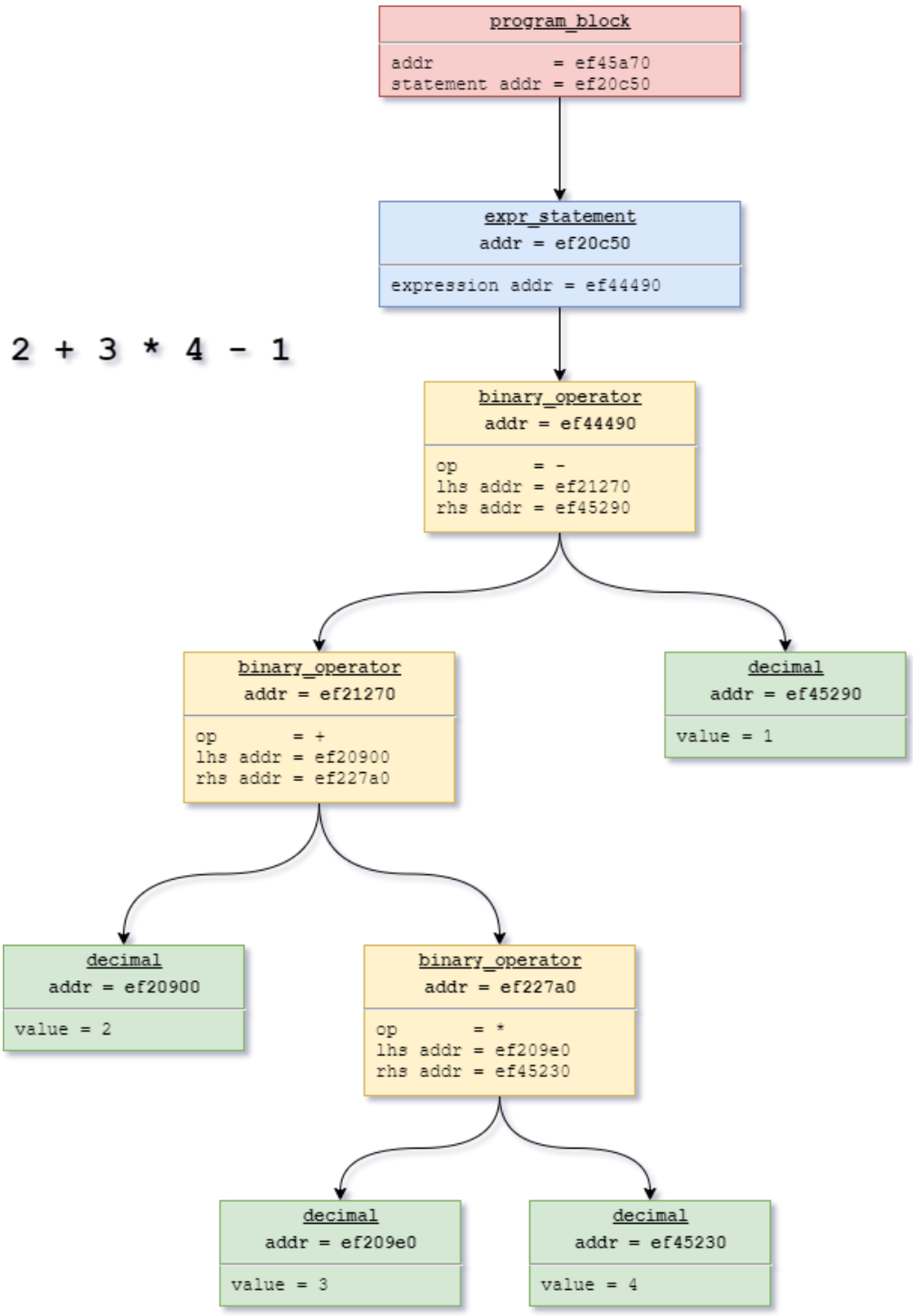


Figure 1-4 AST for 2 + 3 * 4 - 1

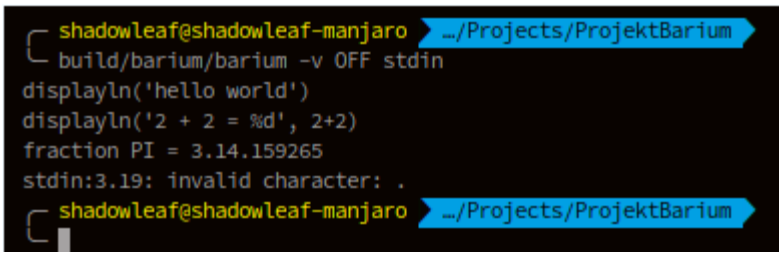


Figure 1-5 Fractional Number Syntax Error

The above figure demonstrates the syntax error caused due to wrong notation of the fraction number.

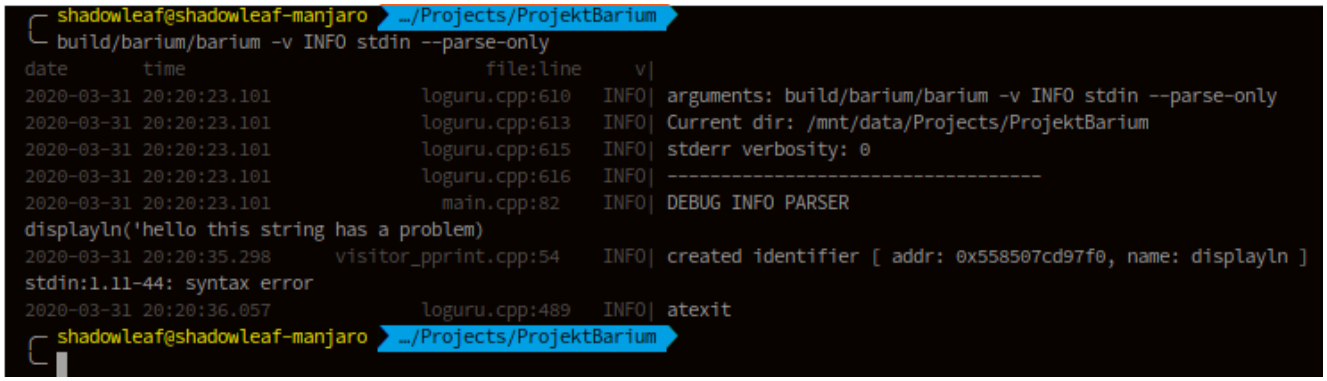


Figure 1-6 String Syntax Error

The above figure demonstrates the syntax error where a single terminating quote of the string is missing.

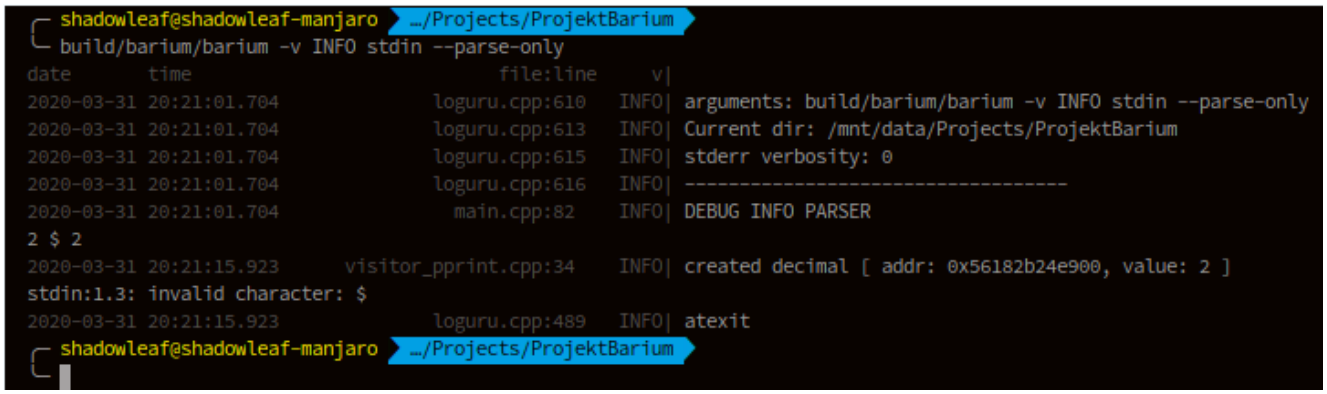


Figure 1-7 Parse Error, unrecognized character

The above figure demonstrates the parser throws a syntax error when an invalid character is given to the program.

Figure 1-8 Simple JIT Compile and Execute

The above figure shows a simple program, that is passed through all the stages of compilation as shown in Figure 1-1 Compiler Recipe, the output is displayed. *Comments are ignored in the source code.*

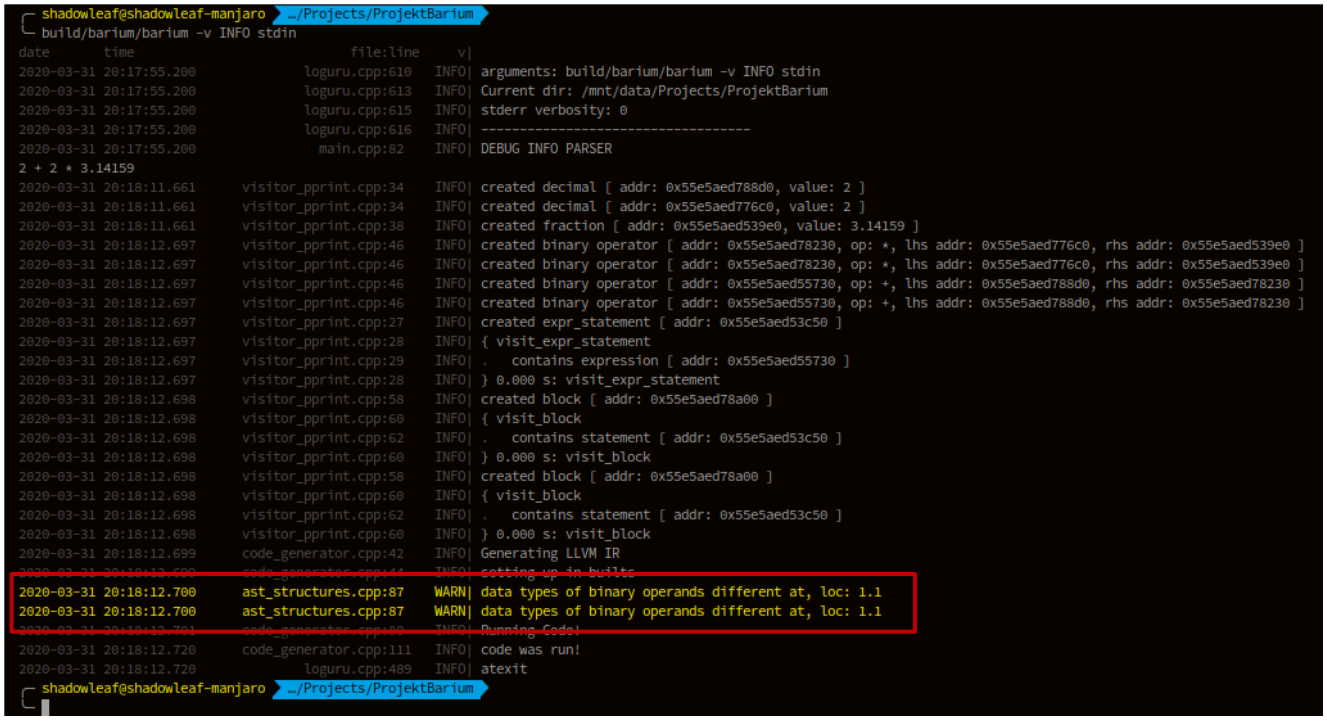


Figure 1-9 Semantic Analyzer data type cast warnings

The above program shows how the program shows a type casting warning when we've tried to do `2 + 2 + 3.14159`, i.e. addition of a decimal to a fraction.

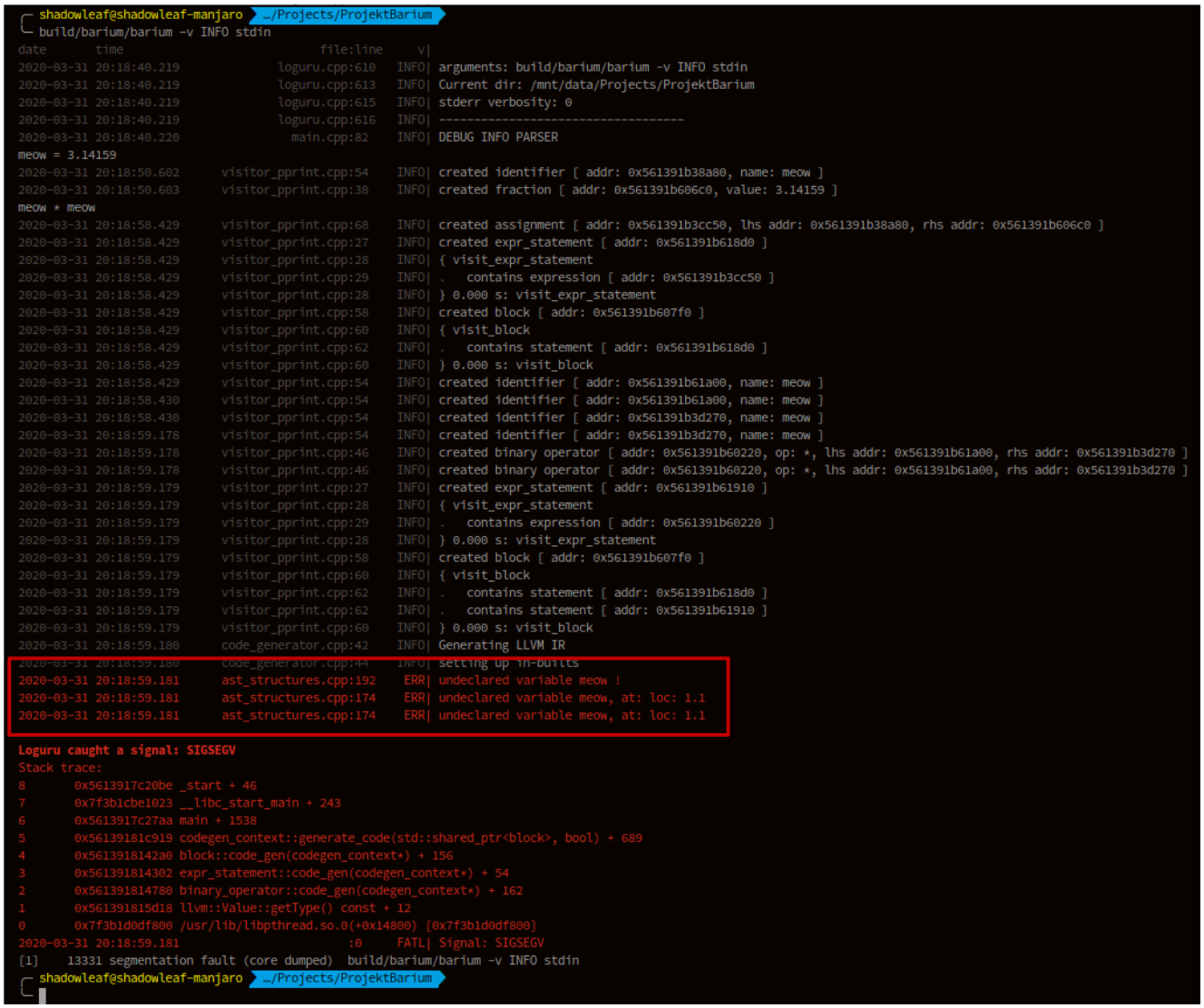


Figure 1-10 Undeclared Variable Error

The above figure shows how the program throws an error when we try to assign value to a variable which is undeclared, this happens during the Intermediate Code Generation stage.

Control Statements, Looping Statements, Arrays and IO Statements run are attached in Appendix A Logs, the program was tested and it works in parsing them.

1.7 Results and Comments

```
shadowleaf@shadowleaf-manjaro …/Projects/ProjektBarium
build/barium/barium -h -v OFF
OVERVIEW: Barium Compiler
Uses STDIN if <input file> is not specified
-v [OFF | INFO | ERROR] sets the verbosity level
USAGE: barium [options] <input file>

OPTIONS:

General options:

  --dump-ir     - Dump the Generated IR on output
  -o=<filename> - Specify output filename
  --parse-only  - Only Parse the source file

Generic Options:

  --help        - Display available options (--help-hidden for more)
  --help-list   - Display list of available options (--help-list-hidden for more)
  --version     - Display the version of this program
shadowleaf@shadowleaf-manjaro …/Projects/ProjektBarium
```

Figure 1-11 ProjektBarium help screen

So, this was ProjektBarium, a simple, tiny compiler using the LLVM Frontend to generate IR and execute the code using the built in MCJIT compiler.

Here is an Example Run

Source File

test_shape_area.bar

```
# calculate the area of a shapes
fraction area = 0.0
fraction PI = 3.14159265358979323846264338327950288419716 9

# 1. circle
fraction rad = 20.5
area = PI * rad * rad
displayln('area of circle with rad %.5f = %.20f sq units', rad, area)

# 2. surface area of cylinder use radius of above circle
fraction height = 69.6
area = 2.0 * PI * rad * height + 2.0 * PI * rad * rad
displayln('surface area of cylinder with height %.5f = %.20f sq units', height, area)
```

Generated LLVM IR

```
2020-04-01 04:53:18.324      code_generator.cpp:73    INFO| Generated IR
; ModuleID = 'barium-jit'
source_filename = "barium-jit"

@.str = private constant [46 x i8] c"area of circle with rad %.5f = %.20f sq units\00", align 1
@.str.1 = private constant [59 x i8] c"surface area of cylinder with height %.5f = %.20f sq units\00", align
1

declare void @display(i8*, ...)

declare void @displayln(i8*, ...)

define i64 @main() {
entry:
  %height = alloca double
  %rad = alloca double
  %PI = alloca double
  %area = alloca double
  store double 0.000000e+00, double* %area
  store double 0x400921FB54442D18, double* %PI
  store double 2.050000e+01, double* %rad
  %PI1 = load double, double* %PI
  %rad2 = load double, double* %rad
  %math_tmp = fmul double %PI1, %rad2
  %rad3 = load double, double* %rad
  %math_tmp4 = fmul double %math_tmp, %rad3
  store double %math_tmp4, double* %area
  %rad5 = load double, double* %rad
  %area6 = load double, double* %area
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([46 x i8], [46 x i8]* @.str, i64 0, i64 0),
double %rad5, double %area6)
  store double 6.960000e+01, double* %height
  %PI7 = load double, double* %PI
  %math_tmp8 = fmul double 2.000000e+00, %PI7
  %rad9 = load double, double* %rad
  %math_tmp10 = fmul double %math_tmp8, %rad9
  %height11 = load double, double* %height
  %math_tmp12 = fmul double %math_tmp10, %height11
  %PI13 = load double, double* %PI
  %math_tmp14 = fmul double 2.000000e+00, %PI13
  %rad15 = load double, double* %rad
  %math_tmp16 = fmul double %math_tmp14, %rad15
  %rad17 = load double, double* %rad
  %math_tmp18 = fmul double %math_tmp16, %rad17
  %math_tmp19 = fadd double %math_tmp12, %math_tmp18
  store double %math_tmp19, double* %area
  %height20 = load double, double* %height
  %area21 = load double, double* %area
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([59 x i8], [59 x i8]* @.str.1, i64 0, i64 0),
double %height20, double %area21)
  ret i64 1
}
```

OUTPUT

```
area of circle with rad 20.50000 = 1320.25431267111048327934 sq units
surface area of cylinder with height 69.60000 = 11605.35742162605311023071 sq units
```

Its amazing how you can write code that writes code so it can take code as input and execute it.

1.7.1 Limitations

The language is very limited and cannot be called a language since we didn't do a full implementation of functions and modules. LLVM has full support for these features and even more!

The language is missing recursion, which is very fundamental when it comes to writing some of our basic data structures like linked lists.

1.7.2 Further Improvements

The goal of project barium was to create a functional language, but things didn't turn out so well, i plan to do so at some later point in time, i.e. restructure the grammar to form a functional language. But why? Function languages are fundamentally simple and have a very simple syntax, but they are very powerful, everything is a function, numbers are encoded as Churchill Encodings, which are Lambdas, which is a function. This makes it easy to create a full-fledged language with very less code.

Bibliography

1. The Architecture of Open Source Applications http://www.aosabook.org/en/llvm.html
2. Loren Segal, Writing Your Own Toy Compiler Using Flex, Bison and LLVM, https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/
3. LLVM Kaleidoscope Tutorial, https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/
4. Satyajit Ghana, ProjektBarium, https://github.com/satyajitghana/ProjektBarium
5. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D, Compilers: Principles, Techniques, and Tools
6. Regents of the University of California, Flex, version 2.5 A fast scanner generator https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html


```
Total Lines of Code Written: 5.5K

Languages Used: 6 (C++ 65.4%)

License: MIT
```

Appendix A (Testing and Logs)

1.7.3 test_all_ir.bar

```
displayln('hellow fellow bariums')
decimal meow = 20
displayln('meow value is %f', meow / 3.0)
displayln('2+3*4+2+2*12 = %d', 2 + 3 * 4 + 2 + 2 * 12)
displayln('0 or 2 = %d', 0 or 2)
displayln('not 0 = %d', not 0)
displayln('1 and 1 = %d', 1 and 1)
```

LOG

```
date       time                          file:line     v|
2020-04-01 04:53:18.029              loguru.cpp:610   INFO| arguments: ./build/barium/barium test_files/test_all_ir.bar -v
INFO --dump-ir
2020-04-01 04:53:18.029              loguru.cpp:613   INFO| Current dir: /mnt/data/Projects/ProjektBarium
2020-04-01 04:53:18.029              loguru.cpp:615   INFO| stderr verbosity: 0
2020-04-01 04:53:18.029              loguru.cpp:616   INFO| ----------------------------------
2020-04-01 04:53:18.032      code_generator.cpp:42    INFO| Generating LLVM IR
2020-04-01 04:53:18.032      code_generator.cpp:44    INFO| setting up in-builts
2020-04-01 04:53:18.034      ast_structures.cpp:87    WARN| data types of binary operands different at, loc: 1.1
2020-04-01 04:53:18.034      ast_structures.cpp:139   WARN| Producing BITWISE operator at, loc: 1.1
2020-04-01 04:53:18.034      code_generator.cpp:73    INFO| Generated IR
; ModuleID = 'barium-jit'
source_filename = "barium-jit"

@.str = private constant [22 x i8] c"hellow fellow bariums\00", align 1
@.str.1 = private constant [17 x i8] c"meow value is %f\00", align 1
@.str.2 = private constant [18 x i8] c"2+3*4+2+2*12 = %d\00", align 1
@.str.3 = private constant [12 x i8] c"0 or 2 = %d\00", align 1
@.str.4 = private constant [11 x i8] c"not 0 = %d\00", align 1
@.str.5 = private constant [13 x i8] c"1 and 1 = %d\00", align 1

declare void @display(i8*, ...)

declare void @displayln(i8*, ...)

define i64 @main() {
entry:
  %meow = alloca i64
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([22 x i8], [22 x i8]* @.str, i64 0, i64 0))
  store i64 20, i64* %meow
  %meow1 = load i64, i64* %meow
  %cast_double = sitofp i64 %meow1 to double
  %math_tmp = fdiv double %cast_double, 3.000000e+00
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([17 x i8], [17 x i8]* @.str.1, i64 0, i64 0), double
%math_tmp)
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([18 x i8], [18 x i8]* @.str.2, i64 0, i64 0), i64 40)
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([12 x i8], [12 x i8]* @.str.3, i64 0, i64 0), i64 2)
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([11 x i8], [11 x i8]* @.str.4, i64 0, i64 0), i64 -1)
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @.str.5, i64 0, i64 0), i64 1)
  ret i64 1
}
2020-04-01 04:53:18.036      code_generator.cpp:80    INFO| Running Code!
2020-04-01 04:53:18.110      code_generator.cpp:111   INFO| code was run!
2020-04-01 04:53:18.112              loguru.cpp:489   INFO| atexit
hellow fellow bariums
meow value is 6.666667
2+3*4+2+2*12 = 40
0 or 2 = 2
```

```
not 0 = -1
1 and 1 = 1
```

1.7.4 test_array.bar

```
displayln('arr[0] = %d', arr[0])
displayln('arr[1][2] = %d', arr[1][2])
```

LOG

```
date       time                            file:line     v|
2020-04-01 04:53:18.451              loguru.cpp:610   INFO| arguments: ./build/barium/barium test_files/test_array.bar -v
INFO --parse-only
2020-04-01 04:53:18.451              loguru.cpp:613   INFO| Current dir: /mnt/data/Projects/ProjektBarium
2020-04-01 04:53:18.451              loguru.cpp:615   INFO| stderr verbosity: 0
2020-04-01 04:53:18.451              loguru.cpp:616   INFO| ----------------------------------
2020-04-01 04:53:18.451                main.cpp:82    INFO| DEBUG INFO PARSER
2020-04-01 04:53:18.451      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x55ff6f5aba80, name: displayln ]
2020-04-01 04:53:18.451      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x55ff6f5d37a0, value: "arr[0] = %d"
]
2020-04-01 04:53:18.451      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x55ff6f5d41e0, name: arr ]
2020-04-01 04:53:18.451      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x55ff6f5b1730, value: 0 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:106   INFO| created array_access [ addr: 0x55ff6f5d4880, var_name: arr,
index expr addr: 0x55ff6f5b1730 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x55ff6f5afc50, ident: displayln
]
2020-04-01 04:53:18.452      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.452      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x55ff6f5d37a0 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x55ff6f5d4880 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.452      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x55ff6f5b1790 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.452      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x55ff6f5afc50 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.452      visitor_pprint.cpp:58    INFO| created block [ addr: 0x55ff6f5d49b0 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.452      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x55ff6f5b1790 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.452      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x55ff6f5d31d0, name: displayln ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x55ff6f5d34b0, value: "arr[1][2] =
%d" ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x55ff6f5d42d0, name: arr ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x55ff6f5d43b0, value: 1 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:106   INFO| created array_access [ addr: 0x55ff6f5d3160, var_name: arr,
index expr addr: 0x55ff6f5d43b0 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x55ff6f5d3b10, value: 2 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:107   INFO| created multidim array_access [ addr: 0x55ff6f5d3520, next_dim
addr: 0x55ff6f5d3160, index: 0x55ff6f5d3b10 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x55ff6f5b17d0, ident: displayln
]
2020-04-01 04:53:18.452      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.452      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x55ff6f5d34b0 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x55ff6f5d3520 ]
2020-04-01 04:53:18.452      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.453      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x55ff6f5d3a30 ]
2020-04-01 04:53:18.453      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.453      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x55ff6f5b17d0 ]
2020-04-01 04:53:18.453      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.453      visitor_pprint.cpp:58    INFO| created block [ addr: 0x55ff6f5d49b0 ]
2020-04-01 04:53:18.453      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.453      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x55ff6f5b1790 ]
2020-04-01 04:53:18.453      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x55ff6f5d3a30 ]
2020-04-01 04:53:18.453      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.453              loguru.cpp:489   INFO| atexit
```

1.7.5 test_branch.bar

```
if (2 > 3) { displayln('2 > 3') } else { displayln('2 < 3') }
```

LOG

```
date       time                           file:line     v|
2020-04-01 04:53:18.199              loguru.cpp:610   INFO| arguments: ./build/barium/barium test_files/test_branch.bar -v
INFO --parse-only
2020-04-01 04:53:18.199              loguru.cpp:613   INFO| Current dir: /mnt/data/Projects/ProjektBarium
2020-04-01 04:53:18.199              loguru.cpp:615   INFO| stderr verbosity: 0
2020-04-01 04:53:18.199              loguru.cpp:616   INFO| ----------------------------------
2020-04-01 04:53:18.199                main.cpp:82    INFO| DEBUG INFO PARSER
2020-04-01 04:53:18.199      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x55a4ea3d5880, value: 2 ]
2020-04-01 04:53:18.199      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x55a4ea3d4670, value: 3 ]
2020-04-01 04:53:18.199      visitor_pprint.cpp:87    INFO| created comp_operator [ addr: 0x55a4ea3d59b0, op: >, lhs addr:
0x55a4ea3d5880, rhs addr: 0x55a4ea3d4670 ]
2020-04-01 04:53:18.199      visitor_pprint.cpp:87    INFO| created comp_operator [ addr: 0x55a4ea3d59b0, op: >, lhs addr:
0x55a4ea3d5880, rhs addr: 0x55a4ea3d4670 ]
2020-04-01 04:53:18.199      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x55a4ea3d51e0, name: displayln ]
2020-04-01 04:53:18.199      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x55a4ea3b2730, value: "2 > 3" ]
2020-04-01 04:53:18.199      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x55a4ea3b0c50, ident: displayln
]
2020-04-01 04:53:18.199      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.199      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x55a4ea3b2730 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.200      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x55a4ea3d47a0 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.200      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x55a4ea3b0c50 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.200      visitor_pprint.cpp:58    INFO| created block [ addr: 0x55a4ea3d5250 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.200      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x55a4ea3d47a0 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.200      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x55a4ea3d44b0, name: displayln ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x55a4ea3d52d0, value: "2 < 3" ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x55a4ea3aca80, ident: displayln
]
2020-04-01 04:53:18.200      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.200      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x55a4ea3d52d0 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.200      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x55a4ea3d42a0 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.200      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x55a4ea3aca80 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.200      visitor_pprint.cpp:58    INFO| created block [ addr: 0x55a4ea3b0a80 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.200      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x55a4ea3d42a0 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.200      visitor_pprint.cpp:91    INFO| created conditional [ addr: 0x55a4ea3d4440, comp_expr:
0x55a4ea3d59b0, then_expr: 0x55a4ea3d5250 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:93    INFO| { visit_conditional
2020-04-01 04:53:18.200      visitor_pprint.cpp:94    INFO| .   contains else_expr [ addr: 0x55a4ea3b0a80 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:93    INFO| } 0.000 s: visit_conditional
2020-04-01 04:53:18.200      visitor_pprint.cpp:58    INFO| created block [ addr: 0x55a4ea3d53b0 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.200      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x55a4ea3d4440 ]
2020-04-01 04:53:18.200      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.200              loguru.cpp:489   INFO| atexit
```

1.7.6 test_io.bar

```
displayln('hello')
decimal meow
read('%d', meow)
```

LOG

```
date       time                           file:line     v|
2020-04-01 04:53:18.251              loguru.cpp:610   INFO| arguments: ./build/barium/barium test_files/test_io.bar -v INFO
--parse-only
2020-04-01 04:53:18.251              loguru.cpp:613   INFO| Current dir: /mnt/data/Projects/ProjektBarium
2020-04-01 04:53:18.251              loguru.cpp:615   INFO| stderr verbosity: 0
2020-04-01 04:53:18.251              loguru.cpp:616   INFO| ----------------------------------
2020-04-01 04:53:18.251                main.cpp:82    INFO| DEBUG INFO PARSER
2020-04-01 04:53:18.258      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x559331c75a80, name: displayln ]
2020-04-01 04:53:18.258      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x559331c9d7a0, value: "hello" ]
2020-04-01 04:53:18.258      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x559331c79c50, ident: displayln
]
2020-04-01 04:53:18.258      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.258      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x559331c9d7a0 ]
2020-04-01 04:53:18.258      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.258      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x559331c9ea20 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.259      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x559331c79c50 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.259      visitor_pprint.cpp:58    INFO| created block [ addr: 0x559331c9e250 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.259      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x559331c9ea20 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.259      visitor_pprint.cpp:83    INFO| created variable_declaration [ addr: 0x559331c9e880, type:
decimal, ident: meow, assign_expr: 0 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x559331c9d1a0, name: read ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x559331c9d440, value: "%d" ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x559331c9d540, name: meow ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x559331c9d540, name: meow ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x559331c7b7a0, ident: read ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.259      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x559331c9d440 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x559331c9d540 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.259      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x559331c9e340 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.259      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x559331c7b7a0 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.259      visitor_pprint.cpp:58    INFO| created block [ addr: 0x559331c9e250 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.259      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x559331c9ea20 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x559331c9e880 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x559331c9e340 ]
2020-04-01 04:53:18.259      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.259              loguru.cpp:489   INFO| atexit
```

1.7.7 test_loop.bar

```
for i in range 2 { displayln('meow') }
for (i = 0, i < 10, i = i + 1) { displayln('meow') }
```

LOG

```
date       time                           file:line     v|
2020-04-01 04:53:18.394              loguru.cpp:610   INFO| arguments: ./build/barium/barium test_files/test_loop.bar -v
INFO --parse-only
2020-04-01 04:53:18.394              loguru.cpp:613   INFO| Current dir: /mnt/data/Projects/ProjektBarium
2020-04-01 04:53:18.394              loguru.cpp:615   INFO| stderr verbosity: 0
2020-04-01 04:53:18.394              loguru.cpp:616   INFO| ----------------------------------
2020-04-01 04:53:18.395                main.cpp:82    INFO| DEBUG INFO PARSER
2020-04-01 04:53:18.395      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f54a80, name: i ]
2020-04-01 04:53:18.395      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f7c7a0, name: displayln ]
2020-04-01 04:53:18.395      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x556637f7d1e0, value: "meow" ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x556637f58c50, ident: displayln
]
```

```
2020-04-01 04:53:18.396      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.396      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x556637f7d1e0 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.396      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x556637f7d880 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.396      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x556637f58c50 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.396      visitor_pprint.cpp:58    INFO| created block [ addr: 0x556637f58a80 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.396      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x556637f7d880 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.396      visitor_pprint.cpp:102   INFO| created for_range [ addr: 0x556637f7d250, ident: i, range_lim:
2, then_expr addr: 0x556637f58a80 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:58    INFO| created block [ addr: 0x556637f7d9b0 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.396      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x556637f7d250 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.396      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f5a7a0, name: i ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x556637f7c210, value: 0 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:68    INFO| created assignment [ addr: 0x556637f7d8c0, lhs addr:
0x556637f5a7a0, rhs addr: 0x556637f7c210 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f7c440, name: i ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f7c440, name: i ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x556637f7d2d0, value: 10 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:87    INFO| created comp_operator [ addr: 0x556637f7d3a0, op: <, lhs addr:
0x556637f7c440, rhs addr: 0x556637f7d2d0 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f7d400, name: i ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f7d5f0, name: i ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f7d5f0, name: i ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:34    INFO| created decimal [ addr: 0x556637f7d740, value: 1 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:46    INFO| created binary operator [ addr: 0x556637f7cf70, op: +, lhs
addr: 0x556637f7d5f0, rhs addr: 0x556637f7d740 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:46    INFO| created binary operator [ addr: 0x556637f7cf70, op: +, lhs
addr: 0x556637f7d5f0, rhs addr: 0x556637f7d740 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:68    INFO| created assignment [ addr: 0x556637f7cf00, lhs addr:
0x556637f7d400, rhs addr: 0x556637f7cf70 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:54    INFO| created identifier [ addr: 0x556637f7cfe0, name: displayln ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:42    INFO| created stringlit [ addr: 0x556637f7d0c0, value: "meow" ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:72    INFO| created function_call [ addr: 0x556637f7cbb0, ident: displayln
]
2020-04-01 04:53:18.396      visitor_pprint.cpp:73    INFO| { visit_function_call
2020-04-01 04:53:18.396      visitor_pprint.cpp:78    INFO| .   contains arg [ addr: 0x556637f7d0c0 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:73    INFO| } 0.000 s: visit_function_call
2020-04-01 04:53:18.396      visitor_pprint.cpp:27    INFO| created expr_statement [ addr: 0x556637f7d330 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:28    INFO| { visit_expr_statement
2020-04-01 04:53:18.396      visitor_pprint.cpp:29    INFO| .   contains expression [ addr: 0x556637f7cbb0 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:28    INFO| } 0.000 s: visit_expr_statement
2020-04-01 04:53:18.396      visitor_pprint.cpp:58    INFO| created block [ addr: 0x556637f5a730 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.396      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x556637f7d330 ]
2020-04-01 04:53:18.396      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.397      visitor_pprint.cpp:98    INFO| created for_loop [ addr: 0x556637f58ed0, init_expr addr:
0x556637f7d8c0, cond_expr addr: 0x556637f7d3a0, update_expr addr: 0x556637f7cf00, then_expr addr: 0x556637f5a730 ]
2020-04-01 04:53:18.397      visitor_pprint.cpp:58    INFO| created block [ addr: 0x556637f7d9b0 ]
2020-04-01 04:53:18.397      visitor_pprint.cpp:60    INFO| { visit_block
2020-04-01 04:53:18.397      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x556637f7d250 ]
2020-04-01 04:53:18.397      visitor_pprint.cpp:62    INFO| .   contains statement [ addr: 0x556637f58ed0 ]
2020-04-01 04:53:18.397      visitor_pprint.cpp:60    INFO| } 0.000 s: visit_block
2020-04-01 04:53:18.397              loguru.cpp:489   INFO| atexit
```

1.7.8 test_shape_area.bar

```
# calculate the area of a shapes
fraction area = 0.0
```

```
fraction PI = 3.14159265358979323846264338327950288419716939
```

```
# 1. circle
fraction rad = 20.5
area = PI * rad * rad
displayln('area of circle with rad %.5f = %.20f sq units', rad, area)

# 2. surface area of cylinder use radius of above circle
fraction height = 69.6
area = 2.0 * PI * rad * height + 2.0 * PI * rad * rad
displayln('surface area of cylinder with height %.5f = %.20f sq units', height, area)
```

LOG

```
date       time                            file:line     v|
2020-04-01 04:53:18.321              loguru.cpp:610   INFO| arguments: ./build/barium/barium test_files/test_shape_area.bar
-v INFO --dump-ir
2020-04-01 04:53:18.321              loguru.cpp:613   INFO| Current dir: /mnt/data/Projects/ProjektBarium
2020-04-01 04:53:18.321              loguru.cpp:615   INFO| stderr verbosity: 0
2020-04-01 04:53:18.321              loguru.cpp:616   INFO| -----------------------------------
2020-04-01 04:53:18.323      code_generator.cpp:42    INFO| Generating LLVM IR
2020-04-01 04:53:18.323      code_generator.cpp:44    INFO| setting up in-builts
2020-04-01 04:53:18.324      code_generator.cpp:73    INFO| Generated IR
; ModuleID = 'barium-jit'
source_filename = "barium-jit"

@.str = private constant [46 x i8] c"area of circle with rad %.5f = %.20f sq units\00", align 1
@.str.1 = private constant [59 x i8] c"surface area of cylinder with height %.5f = %.20f sq units\00", align 1

declare void @display(i8*, ...)

declare void @displayln(i8*, ...)

define i64 @main() {
entry:
  %height = alloca double
  %rad = alloca double
  %PI = alloca double
  %area = alloca double
  store double 0.000000e+00, double* %area
  store double 0x400921FB54442D18, double* %PI
  store double 2.050000e+01, double* %rad
  %PI1 = load double, double* %PI
  %rad2 = load double, double* %rad
  %math_tmp = fmul double %PI1, %rad2
  %rad3 = load double, double* %rad
  %math_tmp4 = fmul double %math_tmp, %rad3
  store double %math_tmp4, double* %area
  %rad5 = load double, double* %rad
  %area6 = load double, double* %area
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([46 x i8], [46 x i8]* @.str, i64 0, i64 0), double %rad5,
double %area6)
  store double 6.960000e+01, double* %height
  %PI7 = load double, double* %PI
  %math_tmp8 = fmul double 2.000000e+00, %PI7
  %rad9 = load double, double* %rad
  %math_tmp10 = fmul double %math_tmp8, %rad9
  %height11 = load double, double* %height
  %math_tmp12 = fmul double %math_tmp10, %height11
  %PI13 = load double, double* %PI
  %math_tmp14 = fmul double 2.000000e+00, %PI13
  %rad15 = load double, double* %rad
  %math_tmp16 = fmul double %math_tmp14, %rad15
  %rad17 = load double, double* %rad
  %math_tmp18 = fmul double %math_tmp16, %rad17
```

```
  %math_tmp19 = fadd double %math_tmp12, %math_tmp18
  store double %math_tmp19, double* %area
  %height20 = load double, double* %height
  %area21 = load double, double* %area
  call void (i8*, ...) @displayln(i8* getelementptr inbounds ([59 x i8], [59 x i8]* @.str.1, i64 0, i64 0), double
%height20, double %area21)
  ret i64 1
}
2020-04-01 04:53:18.325      code_generator.cpp:80    INFO| Running Code!
2020-04-01 04:53:18.333      code_generator.cpp:111   INFO| code was run!
2020-04-01 04:53:18.333              loguru.cpp:489   INFO| atexit

area of circle with rad 20.50000 = 1320.25431267111048327934 sq units
surface area of cylinder with height 69.60000 = 11605.35742162605311023071 sq units
```

Appendix B (Source Code)

For Complete reference and instructions on building and testing this compiler refer:

https://github.com/satyajitghana/ProjektBarium

Source Code

main.cpp

```cpp
#include <cstdlib>
#include <iostream>
#include <memory>
#include <sstream>

#include "ast/ast_structures.hpp"
#include "codegen/code_generator.hpp"
#include "driver/driver.hpp"

#include "external/loguru.hpp"

#include "llvm/ADT/APFloat.h"
#include "llvm/ADT/ArrayRef.h"
#include "llvm/ADT/Optional.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/ExecutionEngine/Orc/CompileUtils.h"
#include "llvm/ExecutionEngine/SectionMemoryManager.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Verifier.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Support/Host.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetOptions.h"

// #include "lld/Common/LLVM.h"
// #include "lld/Common/Driver.h"

extern std::shared_ptr<block> program_block;

int main(int argc, char* argv[]) {
    using namespace llvm;

    // hide all the preset command line options that llvm takes
```

```
    llvm::cl::OptionCategory CompilerCategory(
        "Compiler Options", "Options for controlling the compilation process.");
    llvm::cl::HideUnrelatedOptions(CompilerCategory);

    cl::opt<std::string> InputFilename(cl::Positional, cl::desc("<input file>"),
                                       cl::value_desc("filename"),
                                       cl::init("stdin"));

    // output file name of the executable and the object file
    cl::opt<std::string> OutputFilename(
        "o", cl::desc("Specify output filename"), cl::value_desc("filename"));

    // only parse, dont run
    cl::opt<bool> ParseOnly("parse-only", cl::desc("Only Parse the source file"));

    // dump ir code
    cl::opt<bool> DumpIR("dump-ir", cl::desc("Dump the Generated IR on output"));

    loguru::g_preamble_thread = false;
    loguru::g_preamble_uptime = false;

    // parses the arguments and removes the -v option
    loguru::init(argc, argv);

    // parse the command line options
    llvm::cl::ParseCommandLineOptions(
        argc, argv,
        "Barium Compiler\nUses STDIN if <input file> is not specified\n"
        "-v [OFF | INFO | ERROR] sets the verbosity level");

    // initiate the driver
    driver drv;

    // if verbosity is set then log the parser debug info
    LOG_S(INFO) << "DEBUG INFO PARSER";
    if (InputFilename.empty()) {
        // parse stdin
        drv.parse("stdin");
    } else {
        drv.parse(InputFilename);
    }

    if (not ParseOnly) {
        std::string output_file =
            OutputFilename.empty() ? "output" : OutputFilename.getValue();

        codegen_context ctx;

        if (program_block == nullptr) {
            LOG_S(ERROR) << "error converting ast to ir! program_block was NULL!";
            exit(0);
        }

        // generate the LLVM IR
        ctx.generate_code(program_block, DumpIR);

        // run the code
        ctx.run_code();
    }

    return 0;
}
```

ast_structures.hpp

```
#pragma once

#include <llvm/IR/Value.h>
#include <iostream>
#include <vector>

#include "../visitor/visitor.hpp"
#include "location.hh"

// this produces cyclic import problem
// #include "codegen/code_generator.hpp"
// so forward declare codegen_context
class codegen_context;

// enum to keep track of which node is which
enum class node_type {
    decimal,
    fraction
};

/// every class has to have a default constructor, since parser.y
/// needs to create these objects without arguments first

// namespace barium {

class node {
   public:
   yy::location loc;
    virtual ~node() {}
    virtual llvm::Value* code_gen(codegen_context* ctx) {
        std::cerr << "ERROR code_gen not implemented" << '\n';
        return nullptr;
    }

    virtual std::string to_str() {
        std::cerr << "ERROR to_str not implemented" << '\n';
        return "";
    }

    virtual void accept(visitor& v) {
        v.visit_node(this);
    }
};

class expression : public node {
    public:
    void accept(visitor& v) override { v.visit_expression(this); }
};

class statement : public node {
    public:
    void accept(visitor& v) override { v.visit_statement(this); }
};

class expr_statement : public statement {
   public:
    std::unique_ptr<expression> expr;
    expr_statement(std::unique_ptr<expression> expr) : expr(std::move(expr)) {}
    expr_statement();

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_expr_statement(this); }
```

```
};

class decimal : public expression {
   public:
    long long value;
    yy::location loc;

    decimal(long long value, yy::location loc) : value(value), loc(loc) {}
    decimal() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_decimal(this); }
};

class fraction : public expression {
   public:
    long double value;
    yy::location loc;

    fraction(long double value, yy::location loc) : value(value), loc(loc) {}
    fraction() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_fraction(this); }
};

class stringlit : public expression {
   public:
    std::string value;
    yy::location loc;

    stringlit(std::string value, yy::location loc) : value(value), loc(loc) {}
    stringlit();

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_stringlit(this); }
};

class binary_operator : public expression {
   public:
    char op = 0;
    std::unique_ptr<expression> lhs = nullptr;
    std::unique_ptr<expression> rhs = nullptr;
    yy::location loc;

    binary_operator(char op, std::unique_ptr<expression> lhs, std::unique_ptr<expression> rhs, yy::location loc) : op(op)
, lhs(std::move(lhs)), rhs(std::move(rhs)), loc(loc) {}
    binary_operator() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_binary_operator(this); }
};

class unary_operator : public expression {
   public:
    char op = 0;
    std::unique_ptr<expression> expr = nullptr;
    yy::location loc;
```

```
    unary_operator(char op, std::unique_ptr<expression> expr, yy::location loc) : op(op), expr(std::move(expr)), loc(loc)
{}
    unary_operator() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_unary_operator(this); }
};

class identifier : public expression {
   public:
    std::string name;
    yy::location loc;

    identifier(const std::string& name, yy::location loc) : name(name), loc(loc) {}
    identifier() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_identifier(this); }

};

class block : public expression {
   public:
    std::vector<std::unique_ptr<statement>> statements;
    block() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_block(this); }
};

class assignment : public expression {
   public:
    std::unique_ptr<identifier> lhs = nullptr;
    std::unique_ptr<expression> rhs = nullptr;
    assignment(std::unique_ptr<identifier> lhs, std::unique_ptr<expression> rhs) : lhs(std::move(lhs)), rhs(std::move(rhs)
) {}
    assignment() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_assignment(this); }
};

class function_call : public expression {
   public:
    std::unique_ptr<identifier> ident;
    std::unique_ptr<std::vector<std::unique_ptr<expression>>> args_list;

    function_call(std::unique_ptr<identifier> ident, std::unique_ptr<std::vector<std::unique_ptr<expression>>> args_list)
 : ident(std::move(ident)), args_list(std::move(args_list)) {}
    function_call() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_function_call(this); }
};

class variable_declaration : public statement {
   public:
    std::unique_ptr<identifier> type;
    std::unique_ptr<identifier> ident;
```

```
    std::unique_ptr<expression> assign_expr;

    variable_declaration(
        std::unique_ptr<identifier> type,
        std::unique_ptr<identifier> ident,
        std::unique_ptr<expression> assign_expr) : type(std::move(type)),
                                                   ident(std::move(ident)),
                                                   assign_expr(std::move(assign_expr)) {}

    variable_declaration(
        std::unique_ptr<identifier> type,
        std::unique_ptr<identifier> ident) : type(std::move(type)),
                                             ident(std::move(ident)) {}

    variable_declaration() {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_variable_declaration(this); }
};

class comp_operator : public expression {
    public:
    std::string op;
    std::unique_ptr<expression> lhs;
    std::unique_ptr<expression> rhs;

    comp_operator(std::string op, std::unique_ptr<expression> lhs, std::unique_ptr<expression> rhs) : op(op), lhs(std::mov
e(lhs)), rhs(std::move(rhs)) {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_comp_operator(this); }
};

class conditional : public statement {
    public:
    std::unique_ptr<expression> comp_expr;
    std::unique_ptr<expression> then_expr;
    std::unique_ptr<expression> else_expr;

    conditional(std::unique_ptr<expression> comp_expr,
    std::unique_ptr<expression> then_expr,
    std::unique_ptr<expression> else_expr = nullptr) : comp_expr(std::move(comp_expr)),
                                                       then_expr(std::move(then_expr)),
                                                       else_expr(std::move(else_expr)) {}

    llvm::Value* code_gen(codegen_context* ctx);

    void accept(visitor& v) override { v.visit_conditional(this); }

};

class for_loop : public statement {
    public:
    std::unique_ptr<expression> init_expr;
    std::unique_ptr<expression> cond_expr;
    std::unique_ptr<expression> update_expr;
    std::unique_ptr<expression> then_expr;

    for_loop(std::unique_ptr<expression> init_expr,
    std::unique_ptr<expression> cond_expr,
    std::unique_ptr<expression> update_expr,
    std::unique_ptr<expression> then_expr) : init_expr(std::move(init_expr)),
```

```
                                        cond_expr(std::move(cond_expr)),
                                        update_expr(std::move(update_expr)),
                                        then_expr(std::move(then_expr)) {}

    void accept(visitor& v) override { v.visit_for_loop(this); }
};

class for_range : public statement {
    public:
    std::unique_ptr<identifier> ident;
    std::unique_ptr<decimal> range_lim;
    std::unique_ptr<expression> then_expr;

    for_range(std::unique_ptr<identifier> ident, std::unique_ptr<decimal> range_lim, std::unique_ptr<expression> then_exp
r) : ident(std::move(ident)), range_lim(std::move(range_lim)), then_expr(std::move(then_expr)) {}

    void accept(visitor& v) override { v.visit_for_range(this); }
};

class array_access : public expression {
    public:
    std::unique_ptr<identifier> var_name;
    std::unique_ptr<expression> index;
    // arr[0][1] -> here arr[0] is the next_dimension
    std::unique_ptr<expression> next_dimension;

    // single index access
    array_access(std::unique_ptr<identifier> var_name, std::unique_ptr<expression> index) : var_name(std::move(var_name))
, index(std::move(index)) {}

    // multi dimension access
    array_access(std::unique_ptr<expression> next_dimension, std::unique_ptr<expression> index) : next_dimension(std::mov
e(next_dimension)), index(std::move(index)) {}

    void accept(visitor& v) override { v.visit_array_access(this); }
};

// }
```

ast_structures.cpp

```
#include "ast_structures.hpp"
#include "../codegen/code_generator.hpp"

#include "parser.hpp"
#include "location.hh"
#include "../visitor/visitor_pprint.hpp"
#include "../external/loguru.hpp"

#include <typeinfo>

llvm::Value* block::code_gen(codegen_context* ctx) {
    llvm::Value* last = nullptr;
    // LOG_S(INFO) << "[ found " << this->statements.size() << " statements ]";
    for (auto& stmt : this->statements) {
        // LOG_S(INFO) << "[ generating code for: " << typeid(*stmt).name() << " ]";
        last = stmt->code_gen(ctx);
    }

    return last;
}

llvm::Value* expr_statement::code_gen(codegen_context* ctx) {
    // LOG_S(INFO) << "[ generating code for: " << typeid(*expr).name() << " ]";
```

```
    return this->expr->code_gen(ctx);
}

llvm::Value* decimal::code_gen(codegen_context* ctx) {
    // LOG_S(INFO) << "[ producing decimal for: " << value << " ]";
    return llvm::ConstantInt::get(codegen_context::TheContext,
                                  llvm::APInt(64, value));
}

llvm::Value* fraction::code_gen(codegen_context* ctx) {
    // LOG_S(INFO) << "[ producing fraction for: " << value << " ]";
    return llvm::ConstantFP::get(codegen_context::TheContext,
                                 llvm::APFloat((double)this->value));
}

llvm::Value* stringlit::code_gen(codegen_context* ctx) {
    // LOG_S(INFO) << "[ producing string literal for: " << value << " ]";
    // return
    // codegen_context::Builder.CreateGlobalString(llvm::StringRef(this->value));

    using namespace llvm;

    // generate the type for the globale var
    ArrayType* ArrayTy_0 = ArrayType::get(
        IntegerType::get(codegen_context::TheContext, 8), value.size() + 1);
    // create global var which holds the constant string.
    GlobalVariable* gvar_array__str = new GlobalVariable(
        *codegen_context::TheModule.get(),
        /*Type=*/ArrayTy_0,
        /*isConstant=*/true, GlobalValue::PrivateLinkage,
        /*Initializer=*/0,  // has initializer, specified below
        ".str");
    gvar_array__str->setAlignment(1);
    // create the contents for the string global.
    Constant* const_array_str =
        ConstantDataArray::getString(codegen_context::TheContext, value);
    // Initialize the global with the string
    gvar_array__str->setInitializer(const_array_str);

    // generate access pointer to the string
    std::vector<Constant*> const_ptr_8_indices;
    ConstantInt* const_int = ConstantInt::get(codegen_context::TheContext,
                                              APInt(64, StringRef("0"), 10));
    const_ptr_8_indices.push_back(const_int);
    const_ptr_8_indices.push_back(const_int);
    Constant* const_ptr_8 = ConstantExpr::getGetElementPtr(
        ArrayTy_0, gvar_array__str, const_ptr_8_indices);

    return const_ptr_8;
}

llvm::Value* binary_operator::code_gen(codegen_context* ctx) {
    using namespace llvm;

    llvm::Value* L = this->lhs->code_gen(ctx);
    // std::cout << "[producing binary_operator for: " << op << " ]"
    //           << "\n";
    llvm::Value* R = this->rhs->code_gen(ctx);

    // check if the value TypeIDs are same for left and right
    // if they are different cast them to doubles, since we only have
    // 2 data types, this works
    if (L->getType()->getTypeID() != R->getType()->getTypeID()) {
        LOG_S(WARNING) << "data types of binary operands different at, " << visitor_pprint::get_loc(this);
```

```
        auto doubleTy = ctx->Builder.getDoubleTy();

        // cast RHS
        auto cast_instr = CastInst::getCastOpcode(R, true, doubleTy, true);
        // CastOp, Value*, Type*, Twine
        R = ctx->Builder.CreateCast(cast_instr, R, doubleTy, "cast_double");

        // cast LHS
        cast_instr = CastInst::getCastOpcode(L, true, doubleTy, true);
        L = ctx->Builder.CreateCast(cast_instr, L, doubleTy, "cast_double");
    }

    bool is_double = R->getType()->isFloatingPointTy();

    Instruction::BinaryOps op_instr;
    switch (this->op) {
        case '+':
            op_instr = is_double ? Instruction::FAdd : Instruction::Add;
            break;
        case '-':
            op_instr = is_double ? Instruction::FSub : Instruction::Sub;
            break;
        case '*':
            op_instr = is_double ? Instruction::FMul : Instruction::Mul;
            break;
        case '/':
            op_instr = is_double ? Instruction::FDiv : Instruction::SDiv;
            break;
        // these are short circuited logical operators
        case '&':
            op_instr = Instruction::And;
            break;
        case '|':
            op_instr = Instruction::Or;
            break;
        default: {
            LOG_S(ERROR) << "unknown operator !" << this->op;
            return nullptr;
        }
    }

    Value* bin_op = ctx->Builder.CreateBinOp(op_instr, L, R, "math_tmp");

    return bin_op;
}

llvm::Value* unary_operator::code_gen(codegen_context* ctx) {
    using namespace llvm;

    // std::cout << "[producing unary operator for: " << this->op << " ]"
    //           << "\n";
    LOG_S(WARNING) << "Producing BITWISE operator at, " << visitor_pprint::get_loc(this);

    Value* expr = this->expr->code_gen(ctx);

    // Instruction::UnaryOps op_instr;
    Value* un_op = nullptr;

    switch (this->op) {
        // does bitwise not
        case '!': {
            Value* neg_one = llvm::ConstantInt::get(ctx->TheContext,
                                                    llvm::APInt(64, -1));
```

```
            auto instr = Instruction::Xor;
            un_op = ctx->Builder.CreateBinOp(instr, neg_one, expr, "not_temp");
        } break;
        default: {
            LOG_S(ERROR) << "unknown operator !" << this->op;
            return nullptr;
        }
    }

    // Value* un_op = ctx->Builder.CreateUnOp(op_instr, expr, "unary_tmp");

    return un_op;
}

llvm::Value* identifier::code_gen(codegen_context* ctx) {
    using namespace llvm;

    // std::cout << "[producing identifier for: " << this->name << " ]"
    //           << "\n";

    // check if the variable does not exist in the current locals
    if (ctx->current_block()->locals.find(this->name) == ctx->current_block()->locals.end()) {
        LOG_S(ERROR) << "undeclared variable " << this->name << ", at: " << visitor_pprint::get_loc(this);

        return nullptr;
    }

    // load the variable
    Value* loaded_var = ctx->Builder.CreateLoad(ctx->current_block()->locals[this->name], this->name.c_str());

    return loaded_var;
}

llvm::Value* assignment::code_gen(codegen_context* ctx) {
    using namespace llvm;

    // std::cout << "[ producing assignment for: " << lhs->name << " ]"
    //           << "\n";
    // check if the variable does not exist in the current locals
    if (ctx->current_block()->locals.find(lhs->name) == ctx->current_block()->locals.end()) {
        LOG_S(ERROR) << "undeclared variable " << lhs->name << " !";

        return nullptr;
    }

    StoreInst* si = ctx->Builder.CreateStore(this->rhs->code_gen(ctx), ctx->current_block()->locals[this->lhs->name]);

    return si;
}

llvm::Value* function_call::code_gen(codegen_context* ctx) {
    using namespace llvm;

    // std::cout << "[producing function call for: " << this->ident->name << " ]"
    //           << "\n";

    // fetch the function from the module
    Function* function = codegen_context::TheModule->getFunction(this->ident->name.c_str());

    std::vector<Value*> args;

    // put all the parameters into the vector
    for (auto& arg : *(args_list)) {
        args.push_back(arg->code_gen(ctx));
```

```
    }

    // create the instruction call
    CallInst* call = CallInst::Create(function, args, "", ctx->blocks.top()->block);

    return call;

    // experiment on printf function call directly from code
    // i then realised about external function, so rather use that
    // its more generic
    // but i'm keeping this here as a reference, might use later ?
    // - shadowleaf
    //
    // if (this->ident->name == "print") {
    //     FunctionType* funcType = FunctionType::get(
    //         IntegerType::getInt32Ty(codegen_context::TheContext),
    //         PointerType::get(Type::getInt8Ty(codegen_context::TheContext), 0),
    //         true);
    //     FunctionCallee CalleeF =
    //         codegen_context::TheModule->getOrInsertFunction("printf", funcType);

    //     return codegen_context::Builder.CreateCall(CalleeF, args, "printfCall");
    // } else {
    //     return nullptr;
    // }
}

llvm::Value* variable_declaration::code_gen(codegen_context* ctx) {
    using namespace llvm;

    // std::cout << "[producing variable declaration for: " << this->ident->name << " ]"
    //           << "\n";

    if (ctx->current_block()->locals[this->ident->name] != nullptr) {
        std::cout << "error ! " << this->ident->name << " already declared" << '\n';

        return nullptr;
    }

    IRBuilder<> TmpB(ctx->current_block()->block, ctx->current_block()->block->begin());

    AllocaInst* alloc = TmpB.CreateAlloca(ctx->type_of(this->type.get()), nullptr, this->ident->name);

    ctx->current_block()->locals[this->ident->name] = alloc;

    // now create an assignment operation for the above allocation
    if (this->assign_expr != nullptr) {
        assignment assign(std::make_unique<identifier>(this->ident->name, this->loc), std::move(this->assign_expr));
        assign.code_gen(ctx);
    }

    return alloc;
}

llvm::Value* comp_operator::code_gen(codegen_context* ctx) {
    using namespace llvm;

    Value* lhs_val = this->lhs->code_gen(ctx);
    Value* rhs_val = this->rhs->code_gen(ctx);

    if (lhs_val == nullptr or rhs_val == nullptr) {
        LOG_S(FATAL) << "error ! lhs or rhs of comp operator null !";

        return nullptr;
```

```
    }

    // TODO: write code to check if the lhs and rhs are types that can be compared
    // also check if they are of differnt types, for now assume they are going to be same
    bool is_double = lhs_val->getType()->isFloatingPointTy();

    Instruction::OtherOps oinstr = is_double ? Instruction::FCmp : Instruction::ICmp;

    CmpInst::Predicate predicate;

    if (this->op == ">=") {
        predicate = is_double ? CmpInst::FCMP_OGE : CmpInst::ICMP_SGE;
    } else if (this->op == ">") {
        predicate = is_double ? CmpInst::FCMP_OGT : CmpInst::ICMP_SGT;
    } else {
        LOG_S(ERROR) << "operator: " << this->op << " not supported!";
        return nullptr;
    }

    return CmpInst::Create(oinstr, predicate, lhs_val, rhs_val, "cmp_tmp", ctx->current_block()->block);
}

llvm::Value* conditional::code_gen(codegen_context* ctx) {
    using namespace llvm;

    Value* comp_val = this->comp_expr->code_gen(ctx);

    return nullptr;
}
```

extern_func.hpp

```
#pragma once

#define DECLSPEC

extern "C" DECLSPEC void display(char* str, ...);

extern "C" DECLSPEC void displayln(char* str, ...);

// extern "C" DECLSPEC void read(char* str, ...) {
```

extern_func.cpp

```
#include "extern_func.hpp"
#include <stdarg.h>
#include <stdio.h>
#include <cstring>
#include <iostream>

extern "C" DECLSPEC void display(char* str, ...) {
    va_list argp;
    va_start(argp, str);
    vprintf(str, argp);
    va_end(argp);
}

extern "C" DECLSPEC void displayln(char* str, ...) {
    char* outstr;
    va_list argp;
    va_start(argp, str);
    outstr = (char*)malloc(strlen(str) + 2);
    strcpy(outstr, str);
    strcat(outstr, "\n");
    vprintf(outstr, argp);
```

```
    va_end(argp);
    free(outstr);
}

// TODO: implement read
// extern "C" DECLSPEC void read(char* str, ...) {

// }
```

code_generator.hpp

```
#pragma once

#include "../ast/ast_structures.hpp"

#include <algorithm>
#include <cctype>
#include <cstdio>
#include <cstdlib>
#include <map>
#include <memory>
#include <stack>
#include <string>
#include <vector>
#include "llvm/ADT/APFloat.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/ExecutionEngine/GenericValue.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Verifier.h"

#include "extern_func.hpp"

class basic_block {
   public:
    llvm::BasicBlock* block;
    std::map<std::string, llvm::AllocaInst*> locals;

    basic_block(llvm::BasicBlock* block) : block(block) {}
};

class codegen_context {
   public:
    std::stack<std::unique_ptr<basic_block>> blocks;
    llvm::Function* main_function;
    // store the llvm::Function*, pointer to the extern function
    std::vector<std::pair<llvm::Function*, void*>> inbuilts_info;

    // as explained in llvm tutorial
    static llvm::LLVMContext TheContext;
    static llvm::IRBuilder<> Builder;
    static std::unique_ptr<llvm::Module> TheModule;
    // static llvm::Module* TheModule;
    static std::map<std::string, llvm::Value*> NamedValues;
    static std::unique_ptr<llvm::legacy::FunctionPassManager> TheFPM;

    void generate_code(std::shared_ptr<block> root, bool dump_ir);
```

```
    llvm::GenericValue run_code();

    void setup_inbuilts();

    codegen_context();

    llvm::Type* type_of(const identifier* type);

    basic_block* current_block() { return this->blocks.top().get(); }
};
```

code_generator.cpp

```
#include "code_generator.hpp"

#include "../external/loguru.hpp"

#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/ExecutionEngine/GenericValue.h"
#include "llvm/ExecutionEngine/Orc/CompileUtils.h"
#include "llvm/ExecutionEngine/SectionMemoryManager.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include "llvm/Support/TargetSelect.h"

// declare all the static variables
// this will also define their storage size
// and that's how static variables are instantiated

// LLVMContext holds a lot of LLVM Data Structures like type and constant value tables
llvm::LLVMContext codegen_context::TheContext;

// Builder makes it easy to generate LLVM IR, IR Builder class keeps track of wehere to insert instrs. to
llvm::IRBuilder<> codegen_context::Builder(codegen_context::TheContext);

// Module contains functions and global variables
std::unique_ptr<llvm::Module> codegen_context::TheModule = std::make_unique<llvm::Module>("barium-
jit", codegen_context::TheContext);
// llvm::Module* codegen_context::TheModule = nullptr;

// NamedValues keeps track of which values are defined in the current scope. It's a symbol table for the code
std::map<std::string, llvm::Value*> codegen_context::NamedValues;

std::unique_ptr<llvm::legacy::FunctionPassManager> codegen_context::TheFPM;

codegen_context::codegen_context() {
    // Initialize Native Target
    llvm::InitializeNativeTarget();
    llvm::InitializeNativeTargetAsmParser();
    llvm::InitializeNativeTargetAsmPrinter();
}

void codegen_context::generate_code(std::shared_ptr<block> root, bool dump_ir = false) {
    LOG_S(INFO) << "Generating LLVM IR";

    LOG_S(INFO) << "setting up in-builts";

    // add the inbuilt functions to the module
    setup_inbuilts();

    // create the argument list
    std::vector<llvm::Type*> argTypes;
```

```
    // create the function prototype/signature
    llvm::FunctionType* funcType = llvm::FunctionType::get(codegen_context::Builder.getInt64Ty(), argTypes, false);

    // create the main_function
    this-
>main_function = llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", codegen_context::TheModule.get()
);

    // create the entry block and fill it with appropriate code
    llvm::BasicBlock* entryBlock = llvm::BasicBlock::Create(codegen_context::TheContext, "entry", this->main_function, 0);

    this->blocks.emplace(std::make_unique<basic_block>(entryBlock));
    // the code will be inserted into entry block now
    codegen_context::Builder.SetInsertPoint(entryBlock);
    root->code_gen(this);  // generate code for the entire tree
    // codegen_context::Builder.CreateRet(blocks.top()->block);

    // set the return of main function to decimal 1
    codegen_context::Builder.CreateRet(llvm::ConstantInt::get(codegen_context::TheContext,
                                                              llvm::APInt(64, 1)));
    this->blocks.pop();

    // print the IR
    if (dump_ir) {
        LOG_S(INFO) << "Generated IR";
        codegen_context::TheModule->print(llvm::errs(), nullptr);
    }
}

llvm::GenericValue codegen_context::run_code() {
    using namespace llvm;
    LOG_S(INFO) << "Running Code!";

    TargetOptions Opts;

    auto* module = this->TheModule.get();
    std::unique_ptr<RTDyldMemoryManager> MemMgr(new llvm::SectionMemoryManager());

    // Create the JIT Engine
    EngineBuilder factory(std::move(this->TheModule));
    factory.setEngineKind(EngineKind::JIT);
    factory.setTargetOptions(Opts);
    factory.setMCJITMemoryManager(std::move(MemMgr));

    // setup the execution engine
    auto execution_engine = std::unique_ptr<ExecutionEngine>(factory.create());

    // set the memory layout of the module to same as of the engine
    module->setDataLayout(execution_engine->getDataLayout());

    // add the inbuilt functions to the execution engine
    for (auto [fun, fun_addr] : this->inbuilts_info) {
        execution_engine->addGlobalMapping(fun, fun_addr);
    }

    execution_engine->finalizeObject();

    std::vector<GenericValue> noargs;

    // fetch the returned value of the main funciton
    GenericValue val_ret = execution_engine->runFunction(this->main_function, noargs);

    LOG_S(INFO) << "code was run!";
```

```
    return val_ret;
}

void codegen_context::setup_inbuilts() {
    using namespace llvm;

    // setup "display" function

    // arg: int8 pointer
    std::vector<llvm::Type*> display_arg_types(1, codegen_context::Builder.getInt8PtrTy());
    // return: Void, Params: int8 pointer, isVarArg: true
    FunctionType* display_ft = FunctionType::get(codegen_context::Builder.getVoidTy(), display_arg_types, true);

    Function* display_func = Function::Create(display_ft, Function::ExternalLinkage, "display", codegen_context::TheModule.
get());

    auto i = display_func->arg_begin();
    if (i != display_func->arg_end()) {
        i->setName("format_str");
    }

    this->inbuilts_info.push_back({display_func, (void*)display});

    // setup "displayln" function
    Function* displayln_func = Function::Create(display_ft, Function::ExternalLinkage, "displayln", codegen_context::TheMod
ule.get());

    i = displayln_func->arg_begin();
    if (i != displayln_func->arg_end()) {
        i->setName("format_str");
    }

    this->inbuilts_info.push_back({displayln_func, (void*)displayln});
}

llvm::Type* codegen_context::type_of(const identifier* type) {
    if (type->name.compare("decimal") == 0) {
        return this->Builder.getInt64Ty();
    } else if (type->name.compare("fraction") == 0) {
        return this->Builder.getDoubleTy();
    }

    return this->Builder.getVoidTy();
}
```

driver.hpp

```
#pragma once

#include <map>
#include <string>
#include "parser.hpp"

// declare the YY_DECL as our custom parser driver
#define YY_DECL yy::parser::symbol_type yylex(driver& drv)

YY_DECL;

class driver {
  public:
    driver();

    std::map<std::string, int> variables;
```

```
    int result;

    // to run the parser on a given file
    int parse(const std::string& f);

    // name of the file being parsed
    std::string file;

    // handling the scanner
    // NOTE: defined in tokens.l
    void scan_begin();
    void scan_end();

    // token location
    yy::location location;
};
```

driver.cpp

```
#include "driver.hpp"

#include "parser.hpp"

driver::driver() { }

int driver::parse(const std::string& f) {
    file = f;
    location.initialize(&file);

    // scan_begin and scan_end are defined in tokens.l
    scan_begin();

    yy::parser parse(*this);

    // int res =
    parse();

    scan_end();

    // return res;
    return 0;
}
```

tokens.l

```
%{
    #include <string>
    #include <cerrno>
    #include <climits>
    #include <cstdlib>
    #include <cstring> // strerror

    #include "driver/driver.hpp"
    #include "parser.hpp"
    #include "ast/ast_structures.hpp"

    // temporary for storing the string literal
    std::string g_str;

%}

%option noyywrap nounput noinput batch

%x str
%s normal
```

```
%{
    yy::parser::symbol_type make_DECIMAL(const std::string& s, const yy::parser::location_type& loc);
    yy::parser::symbol_type make_FRACTION(const std::string& s, const yy::parser::location_type& loc);
    yy::parser::symbol_type make_IDENT(const std::string& s, const yy::parser::location_type& loc);
%}

ident       [a-zA-Z_][a-zA-Z_0-9]*
num         [0-9]
blank       [ \t\r]

%{
    // runs each time a pattern is matched
    #define YY_USER_ACTION  loc.columns(yyleng);
%}

%%

%{
    yy::location& loc = drv.location;
    loc.step();
%}
 //<- one leading blank space; comments should start one blank space after
 /* state automata for string literal */

'                   { g_str = ""; BEGIN(str); } /*eat '*/
<str>\'             { BEGIN(normal); return yy::parser::make_STRINGLIT(std::make_unique<stringlit>(g_str, loc), loc); }
<str>\\n            g_str += "\n";
<str>\\t            g_str += "\t";
<str>\\r            g_str += "\r";
<str>\\\'           g_str += "'";
<str>\\(.|\n)       g_str += yytext[1];
<str>[^\\']+        g_str += std::string(yytext);

{blank}+            { loc.step(); }
\n+                 { loc.lines(yyleng); loc.step(); }
"and"               { return yy::parser::make_AND(loc); }
"or"                { return yy::parser::make_OR(loc); }
"not"               { return yy::parser::make_NOT(loc); }
"if"                { return yy::parser::make_IF(loc); }
"else"              { return yy::parser::make_ELSE(loc); }
"for"               { return yy::parser::make_FOR(loc); }
"in"                { return yy::parser::make_IN(loc); }
"range"             { return yy::parser::make_RANGE(loc); }
"+"                 { return yy::parser::make_PLUS(loc); }
"-"                 { return yy::parser::make_MINUS(loc); }
"*"                 { return yy::parser::make_MUL(loc); }
"/"                 { return yy::parser::make_DIV(loc); }
"="                 { return yy::parser::make_ASSIGN(loc); }
">"                 { return yy::parser::make_GRT(loc); }
">="                { return yy::parser::make_GRTEQ(loc); }
"<"                 { return yy::parser::make_LES(loc); }
"<="                { return yy::parser::make_LESEQ(loc); }
"!="                { return yy::parser::make_NOTEQ(loc); }
"=="                { return yy::parser::make_EQUAL(loc); }
{num}+\.{num}*      { return make_FRACTION(yytext, loc); }
-?{num}+            { return make_DECIMAL(yytext, loc); }
{ident}             { return make_IDENT(yytext, loc); }
"("                 { return yy::parser::make_LPAREN(loc); }
")"                 { return yy::parser::make_RPAREN(loc); }
"{"                 { return yy::parser::make_LBRACE(loc); }
"}"                 { return yy::parser::make_RBRACE(loc); }
"["                 { return yy::parser::make_LBRACKET(loc); }
"]"                 { return yy::parser::make_RBRACKET(loc); }
```

```
","                 { return yy::parser::make_COMMA(loc); }
#.*                 /* eat everything; single line comment */
.                   { throw yy::parser::syntax_error
                      (loc, "invalid character: " + std::string(yytext));
                    }
<<EOF>>             { return yy::parser::make_END(loc); }

%%

yy::parser::symbol_type make_DECIMAL(const std::string& s, const yy::parser::location_type& loc) {
  std::unique_ptr<decimal> temp = std::make_unique<decimal>(std::strtoll(yytext, NULL, 10), loc);
  return yy::parser::make_DECIMAL(std::move(temp), loc);
}

yy::parser::symbol_type make_FRACTION(const std::string& s, const yy::parser::location_type& loc) {
  std::unique_ptr<fraction> temp = std::make_unique<fraction>(std::strtold(yytext, NULL), loc);
  return yy::parser::make_FRACTION(std::move(temp), loc);
}

yy::parser::symbol_type make_IDENT(const std::string& s, const yy::parser::location_type& loc) {
  std::unique_ptr<identifier> temp = std::make_unique<identifier>(s, loc);
  return yy::parser::make_IDENT(std::move(temp), loc);
}

// code from bison manual: https://www.gnu.org/software/bison/manual/html_node/Calc_002b_002b-Scanner.html

void driver::scan_begin() {
  if (file.empty() || file == "stdin")
    yyin = stdin;
  else if (!(yyin = fopen(file.c_str(), "r"))) {
      std::cerr << "cannot open " << file << ": " << strerror(errno) << '\n';
      exit (EXIT_FAILURE);
    }
}

void driver::scan_end() {
  fclose(yyin);
}
```

parser.y

```
%skeleton "lalr1.cc"
%require "3.5"
%language "c++"

%defines

// variant will make sure we can use our non-trivial types
%define api.value.type variant
%define api.token.constructor
%define parse.assert

// this will be added to the parser.cpp file, cyclic-dependecy is resolved by using
// forward declaration of the driver class, this is added verbatim
// if you want to declare any variables do not do in this requires section
%code requires {
    #include <string>
    #include <memory>
    #include <typeinfo>

    class driver;

    #include "ast/ast_structures.hpp"
    // love you c++ gods, g++ gave me much help in debugging
    // <3
```

```
    #include "visitor/visitor.hpp"
    #include "visitor/visitor_pprint.hpp"
    #include "external/loguru.hpp"

    static int cnt = 0;
}

// parsing context
%param { driver& drv }

// for location tracking
%locations
%verbose

// because we'll be using the driver class methods
%code {
    #include "driver/driver.hpp"

    std::shared_ptr<block> program_block;

    visitor_pprint v_pprint;
}

// to make sure there are no conflicts prepend TOK_
%define api.token.prefix{TOK_}
%token
    END  0  "end of file"
    AND     "and"
    OR      "or"
    NOT     "not"
    FOR     "for"
    IN      "in"
    RANGE   "range"
    IF      "if"
    ELSE    "else"
    ASSIGN  "="
    PLUS    "+"
    MINUS   "-"
    MUL     "*"
    DIV     "/"
    LPAREN  "("
    RPAREN  ")"
    LBRACE  "{"
    RBRACE  "}"
    LBRACKET "["
    RBRACKET "]"
    COMMA   ","
    GRT     ">"
    GRTEQ   ">="
    LES     "<"
    LESEQ   "<="
    NOTEQ   "!="
    EQUAL   "=="

%token  <std::unique_ptr<identifier>>   IDENT       "identifier"
%token  <std::unique_ptr<decimal>>      DECIMAL     "decimal"
%token  <std::unique_ptr<fraction>>     FRACTION    "fraction"
%token  <std::unique_ptr<stringlit>>    STRINGLIT   "stringlit"
%nterm  <std::unique_ptr<identifier>>   identifier // add this for verbosity
%nterm  <std::unique_ptr<expression>>   expr
%nterm  <std::unique_ptr<expression>>   literals
%nterm  <std::unique_ptr<expression>>   binop_expr
%nterm  <std::unique_ptr<expression>>   unaryop_expr
%nterm  <std::unique_ptr<expression>>   compare_expr
```

```
%nterm  <std::unique_ptr<block>>        stmts
%nterm  <std::unique_ptr<block>>        program
%nterm  <std::unique_ptr<block>>        block
%nterm  <std::unique_ptr<statement>>    stmt
%nterm  <std::unique_ptr<statement>>    conditional
%nterm  <std::unique_ptr<statement>>    for_loop
%nterm  <std::unique_ptr<statement>>    for_range
%nterm  <std::unique_ptr<std::vector<std::unique_ptr<expression>>>> call_args
%nterm  <std::unique_ptr<variable_declaration>> var_decl
%nterm  <std::unique_ptr<array_access>> array_access

%printer { yyo << $$; } <*>;

%start program;

%code {
    #define DEBUG_PARSER
    #undef DEBUG_PARSER
}

%%

// left associativity

%left "+" "-";
%left "*" "/";

// program consists of statements

program     : stmts {

                program_block = std::move($1);
                program_block->accept(v_pprint);

                }
            ;

// statements can consist of single or multiple statements

stmts[block]       : stmt {

                $block = std::make_unique<block>();

                $block->statements.emplace_back(std::move($1));
                $$->accept(v_pprint);

                }
            | stmts[meow] stmt {

                $meow->statements.emplace_back(std::move($2));

                // i added this because i std::move everytime and this moves the $block also
                // so i std::move back $meow to block to retain the address of main block
                // it was becoming null before, added null check in main.cpp as well
                // - shadowleaf

                $block = std::move($meow);

                }
            ;

// statement can be an expression or an variable declaration

stmt        : expr {
```

```
                $$ = std::make_unique<expr_statement>(std::move($1));
                $$->accept(v_pprint);
                }
            | var_decl {
                $$ = std::move($1);
                }
            | conditional {
                $$ = std::move($1);
                 }
            | for_loop {
                $$ = std::move($1);
                $$->accept(v_pprint);
                }
            | for_range {
                $$ = std::move($1);
                $$->accept(v_pprint);
            }

            ;

// for loops

for_loop    : "for" "(" expr "," expr "," expr ")" block {
                $$ = std::make_unique<for_loop>(std::move($3), std::move($5), std::move($7), std::move($9));
            }
            ;

for_range   : "for" identifier "in" "range" "decimal" block {
                $$ = std::make_unique<for_range>(std::move($2), std::move($5), std::move($6));
            }
            ;

// a block

block       : "{" stmts "}" {
                    $$ = std::move($2);
                    $$->accept(v_pprint);
                    }
            ;

// conditional statement

conditional     : "if" expr block "else" block {
                    $$ = std::make_unique<conditional>(std::move($2), std::move($3), std::move($5));
                    $$->accept(v_pprint);
                    }
                | "if" expr block {
                    $$ = std::make_unique<conditional>(std::move($2), std::move($3));
                    $$->accept(v_pprint);
                }
                ;

// variable declaration and/or assignment

var_decl    : "identifier" "identifier" {
                $$ = std::make_unique<variable_declaration>(std::move($1), std::move($2));
                $$->accept(v_pprint);
            }
            | "identifier" "identifier" "=" expr {
                $$ = std::make_unique<variable_declaration>(std::move($1), std::move($2), std::move($4));
                $$->accept(v_pprint);
            }
            ;
```

```
// all the literals, like integers, fractions and string literals

literals    : "decimal"  {

                $$ = std::move($1);
                // LOG_S(INFO) << "found decimal at " << @1.begin.line << "." << @1.begin.column;
                $$->accept(v_pprint);

                }
            | "fraction" {

                $$ = std::move($1);
                $$->accept(v_pprint);

                }
            | "stringlit" {

                $$ = std::move($1);
                $$->accept(v_pprint);

            }
            ;

// all the expression statements

expr        : identifier "=" expr {

                $$ = std::make_unique<assignment>(std::move($1), std::move($3));
                $$->accept(v_pprint);

                }

            | identifier "(" call_args ")" {
                // function call

                $$ = std::make_unique<function_call>(std::move($1), std::move($3));
                $$->accept(v_pprint);

                }
            | identifier {
                // just an identifier

                $$ = std::move($1);
                $$->accept(v_pprint);

                }

            | literals {

                // literal, either decimal or fractional

                $$ = std::move($1);

                }
            | binop_expr {

                // some binary operation (numeric, not boolean)

                $$ = std::move($1);
                $$->accept(v_pprint);
                }
            | unaryop_expr {
                // a and or not, unary boolean expression
```

```
                $$ = std::move($1);
                }
            | compare_expr {
                // a comparison expression

                $$ = std::move($1);
                }
            | array_access {
                // accessing an element of array

                $$ = std::move($1);
                }
            | "(" expr ")" {
                $$ = std::move($2);
                $$->accept(v_pprint);
                }
            ;

identifier  : "identifier" { $$ = std::move($1); $$->accept(v_pprint); }

// call arguments of a function
// can be blank

call_args[args_list]   : /*blank*/ {
                $args_list = std::make_unique<std::vector<std::unique_ptr<expression>>>();
                }
            | expr {
                $args_list = std::make_unique<std::vector<std::unique_ptr<expression>>>();
                $args_list->push_back(std::move($1));
                }
            | call_args[arg] "," expr {
                $arg->push_back(std::move($3));
                $args_list = std::move($arg);
                }
            ;

// array access for arr[0], arr[<some expr that evaluate to decimal>]
// or for the future can also be arr['string'] for maps
array_access    : identifier "[" expr "]" {
                    $$ = std::make_unique<array_access>(std::move($1), std::move($3));
                    $$->accept(v_pprint);
                    }
                | array_access "[" expr "]" {
                    $$ = std::make_unique<array_access>(std::move($1), std::move($3));
                    $$->accept(v_pprint);
                }
                ;

// binary operators

binop_expr  : expr "and" expr {

                $$ = std::make_unique<binary_operator>('&', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);

                }
            | expr "or" expr {

                $$ = std::make_unique<binary_operator>('|', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);

                }
            | expr "+" expr {
                $$ = std::make_unique<binary_operator>('+', std::move($1), std::move($3), @$);
```

```
                $$->accept(v_pprint);

                }
            | expr "-" expr {
                $$ = std::make_unique<binary_operator>('-', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);
                }
            | expr "*" expr {
                $$ = std::make_unique<binary_operator>('*', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);
                }
            | expr "/" expr {
                $$ = std::make_unique<binary_operator>('/', std::move($1), std::move($3), @$);
                $$->accept(v_pprint);
                }
            ;

// binary boolean comparison operators
compare_expr    :   expr ">" expr {
                        $$ = std::make_unique<comp_operator>(">", std::move($1), std::move($3));
                        $$->accept(v_pprint);
                    }
                |   expr ">=" expr {
                        $$ = std::make_unique<comp_operator>(">=", std::move($1), std::move($3));
                        $$->accept(v_pprint);
                    }
                |   expr "<" expr {

                        $$ = std::make_unique<comp_operator>("<", std::move($1), std::move($3));
                        $$->accept(v_pprint);
                    }
                |   expr "<=" expr {

                        $$ = std::make_unique<comp_operator>("<=", std::move($1), std::move($3));
                        $$->accept(v_pprint);
                    }
                |   expr "==" expr {

                        $$ = std::make_unique<comp_operator>("==", std::move($1), std::move($3));
                        $$->accept(v_pprint);
                    }
                |   expr "!=" expr {

                        $$ = std::make_unique<comp_operator>("!=", std::move($1), std::move($3));
                        $$->accept(v_pprint);
                }
                ;

// unary operations

unaryop_expr    : "not" expr {
                    $$ = std::make_unique<unary_operator>('!', std::move($2), @$);
                    $$->accept(v_pprint);
                    }
                ;

// // boolean expression

// boolean_expr    : expr "and" expr {

//                     }
//                 | expr "or" expr {

//                     }
```

```
//                | expr "xor" expr {

//                    }

/* testing out a grammar */
/*
program     : expr { std::cout << "expr: " << cnt++ << "\n"; }
            ;

expr        : "decimal" { std::cout<< "decimal: " << cnt++ << "\n"; $$ = std::move($1); }
            | expr "+" expr { std::cout << "expr + expr: " << cnt++ << "\n"; $$ = std::make_unique<binary_operator>('+', st
d::move($1), std::move($3)); }
            ;
*/
%%

void yy::parser::error (const location_type& l, const std::string& m) {
  std::cerr << l << ": " << m << '\n';
}
```