|  |  |
| --- | --- |
| **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

[Declaration Sheet ii](#_Toc36613803)

[Contents iii](#_Toc36613804)

[List Of Figures iv](#_Toc36613805)

[1 Question A 5](#_Toc36613806)

[1.1 Introduction 5](#_Toc36613807)

[1.2 Identification and grouping of Tokens 6](#_Toc36613808)

[1.2.1 Keywords 6](#_Toc36613809)

[1.2.2 Operators 6](#_Toc36613810)

[1.2.3 Special Symbols 6](#_Toc36613811)

[1.2.4 Literals 7](#_Toc36613812)

[1.2.5 Identifier 7](#_Toc36613813)

[1.3 Implementation in Lex 7](#_Toc36613814)

[1.4 Design of Context Free Grammar 11](#_Toc36613815)

[1.5 Implementation in Yacc 13](#_Toc36613816)

[1.6 Testing 22](#_Toc36613817)

[1.7 Results and Comments 28](#_Toc36613818)

[1.7.1 Limitations 30](#_Toc36613819)

[1.7.2 Further Improvements 30](#_Toc36613820)

[Bibliography 31](#_Toc36613821)

[Appendix A (Testing and Logs) 32](#_Toc36613822)

[1.7.3 test\_all\_ir.bar 32](#_Toc36613823)

[1.7.4 test\_array.bar 33](#_Toc36613824)

[1.7.5 test\_branch.bar 34](#_Toc36613825)

[1.7.6 test\_io.bar 34](#_Toc36613826)

[1.7.7 test\_loop.bar 35](#_Toc36613827)

[1.7.8 test\_shape\_area.bar 36](#_Toc36613828)

[Appendix B (Source Code) 39](#_Toc36613829)

# List Of Figures

[Figure 1‑1 Compiler Recipe 5](#_Toc36613830)

[Figure 1‑2 LLVM Optimizer 5](#_Toc36613831)

[Figure 1‑3 Parsing Simple Arithmetic Expression 23](#_Toc36613832)

[Figure 1‑4 AST for 2 + 3 \* 4 - 1 24](#_Toc36613833)

[Figure 1‑5 Fractional Number Syntax Error 25](#_Toc36613834)

[Figure 1‑6 String Syntax Error 25](#_Toc36613835)

[Figure 1‑7 Parse Error, unrecognized character 25](#_Toc36613836)

[Figure 1‑8 Simple JIT Compile and Execute 26](#_Toc36613837)

[Figure 1‑9 Semantic Analyzer data type cast warnings 26](#_Toc36613838)

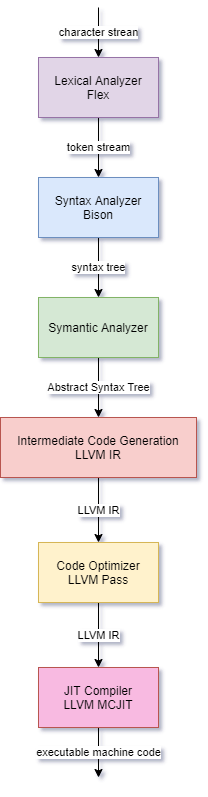
[Figure 1‑10 Undeclared Variable Error 27](#_Toc36613839)

[Figure 1‑11 ProjektBarium help screen 28](#_Toc36613840)

# Question A

Solution to Question A

## 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 ‑ 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]

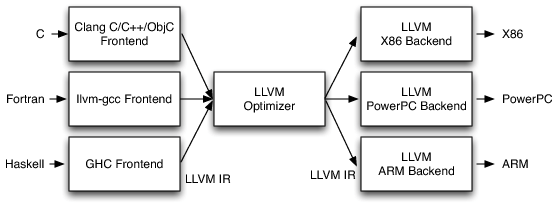


Figure ‑ LLVM Optimizer

## Identification and grouping of Tokens

### Keywords

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

### 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 |

### Special Symbols

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

|  |  |
| --- | --- |
| TOKEN | EOF |
| RE | <<eof>> |

### 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.

### Identifier

|  |  |
| --- | --- |
| TOKEN | IDENTIFIER |
| RE | [a-zA-Z\_][a-zA-Z\_0-9]\* |

## 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\_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);

}

**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

## 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]

## 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), 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>('+', std::move($1), std::move($3)); }

            ;

\*/

**%%**

*void* yy**::**parser**::**error (*const* location\_type*&* **l**, *const* std**::**string*&* **m**) {

  std**::**cerr **<<** l **<<** ": " **<<** m **<<** '\n';

}

## 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.

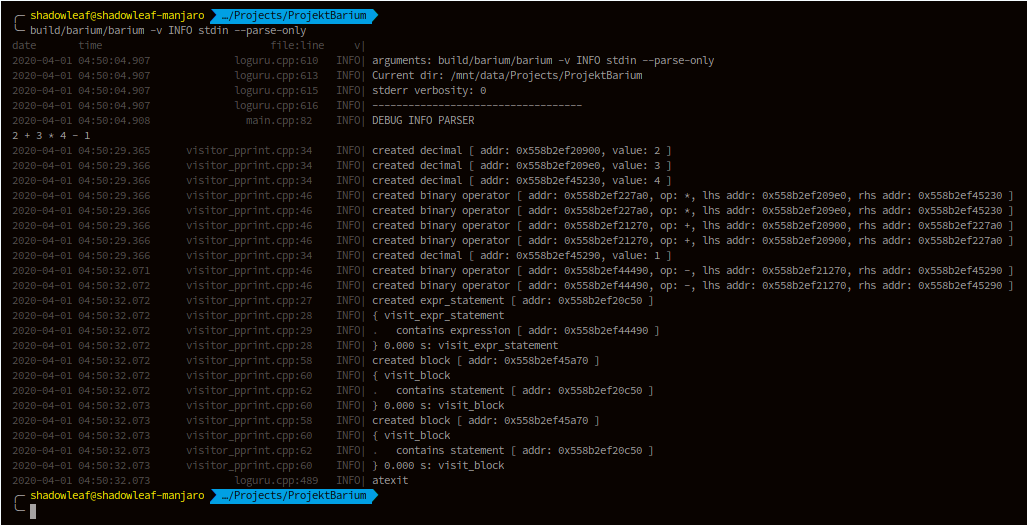


Figure ‑ Parsing Simple Arithmetic Expression

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.

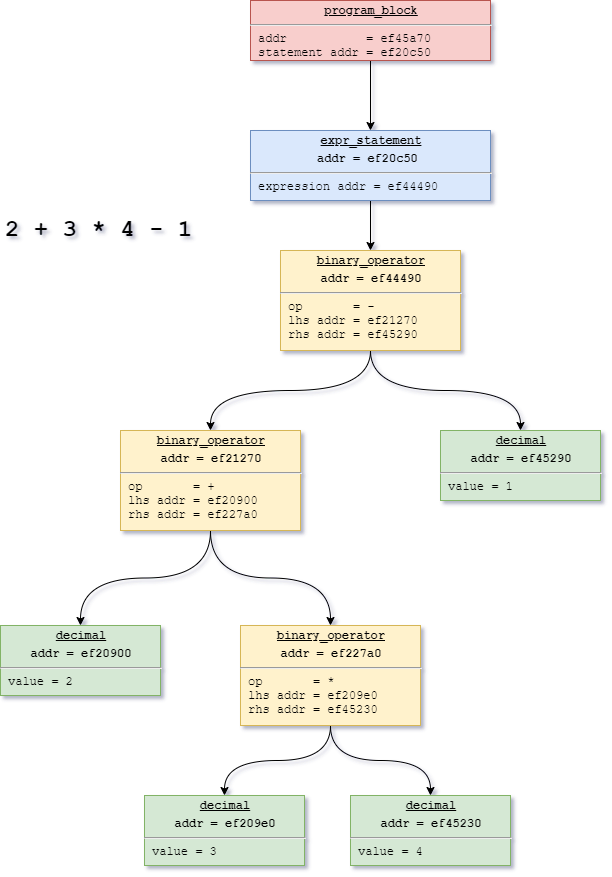


Figure ‑ AST for 2 + 3 \* 4 - 1

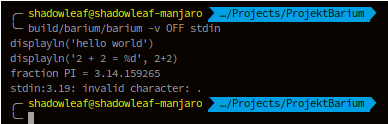


Figure ‑ Fractional Number Syntax Error

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

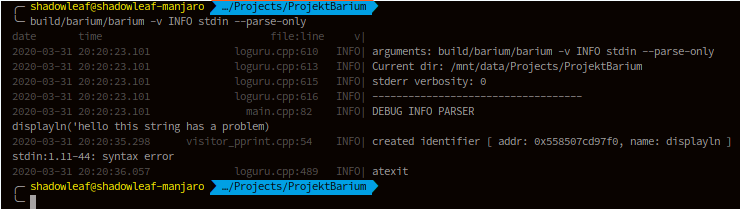


Figure ‑ String Syntax Error

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

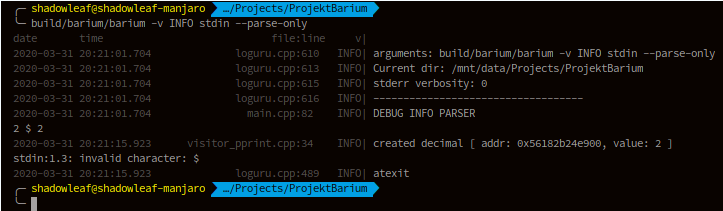


Figure ‑ Parse Error, unrecognized character

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

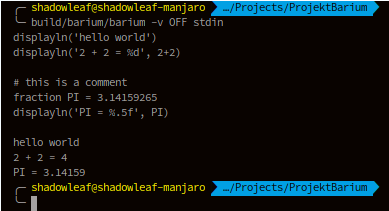


Figure ‑ Simple JIT Compile and Execute

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.*

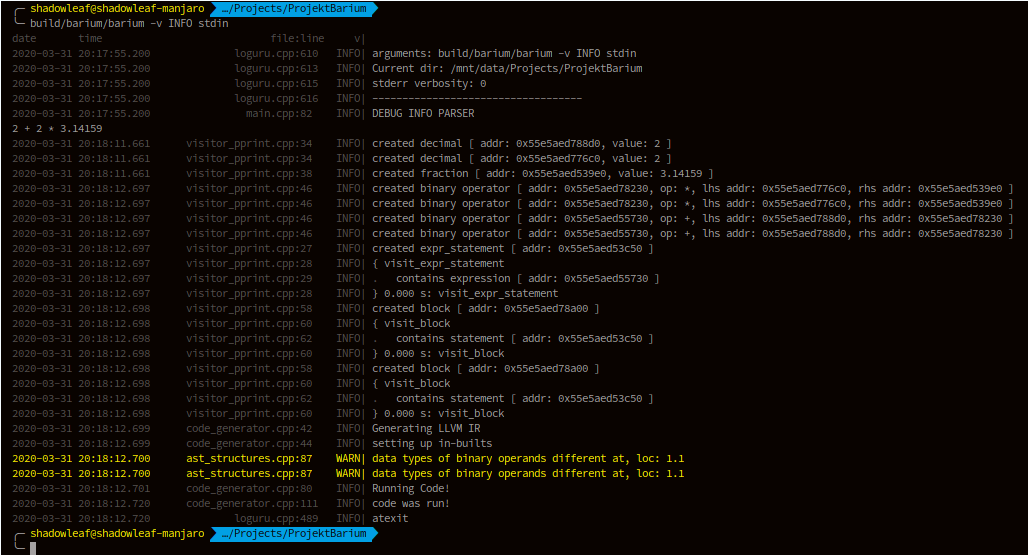


Figure ‑ Semantic Analyzer data type cast warnings

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.

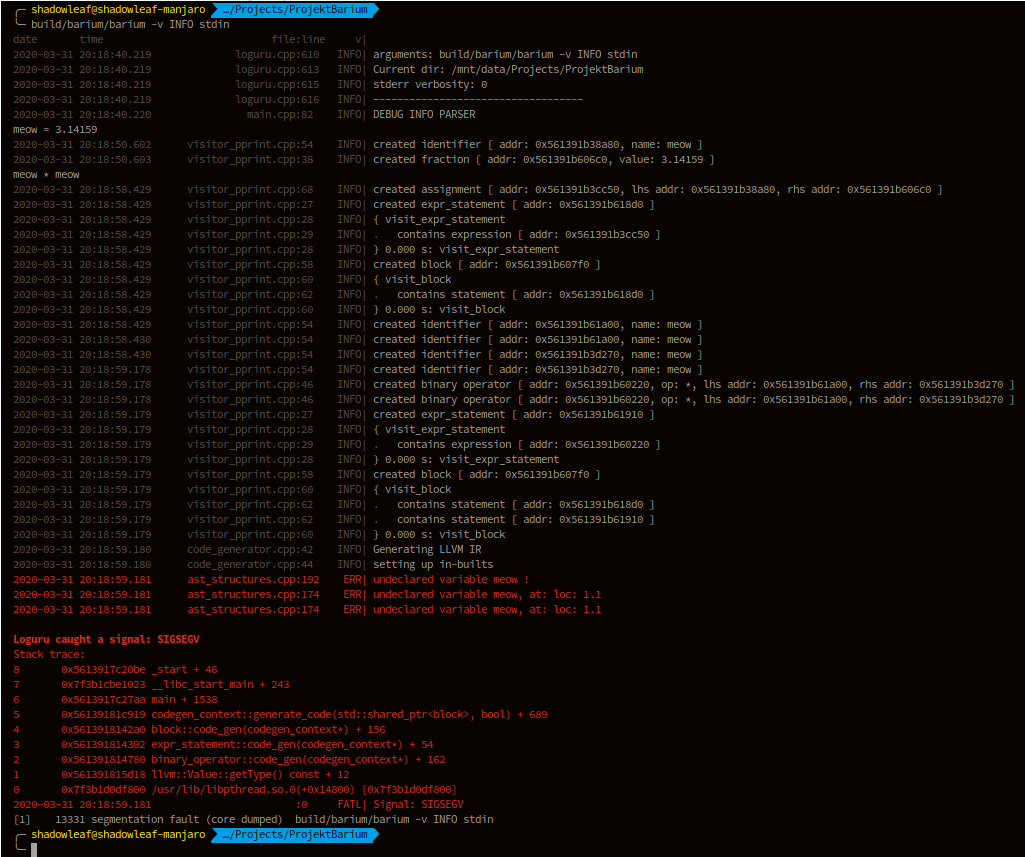


Figure ‑ Undeclared Variable Error

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.

## Results and Comments

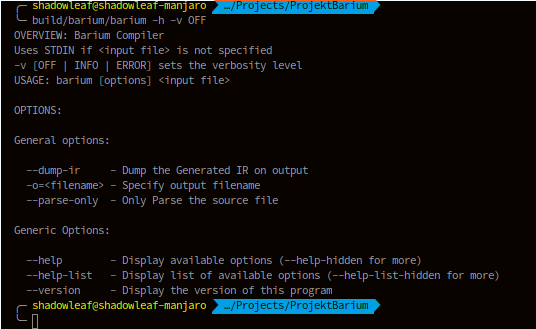


Figure ‑ ProjektBarium help screen

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 = 3.141592653589793238462643383279502884197169

# 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.

### 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.

### 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)

### 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

### 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

### 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

### 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

### 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

### test\_shape\_area.bar

# calculate the area of a shapes

fraction area = 0.0

fraction PI = 3.141592653589793238462643383279502884197169

# 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**](https://github.com/satyajitghana/ProjektBarium)

Source Code

**main.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***::**move(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\_expr**) : 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***::**move(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**::**TheModule.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>('+', std::move($1), std::move($3)); }

            ;

\*/

**%%**

*void* yy**::**parser**::**error (*const* location\_type*&* **l**, *const* std**::**string*&* **m**) {

  std**::**cerr **<<** l **<<** ": " **<<** m **<<** '\n';

}