GrAlgo

Final Report

Team 5

Sahil Chandra - Project Manager - CS20BTECH11033

P Ganesh Nikhil Madhav - System Architect - CS20BTECH11036

Gorantla Pranav Sai - System Integrator - CS20BTECH11018

Suraj Telugu - Language Guru - CS20BTECH11050

Umesh Kalvakuntla - System Architect - CS20BTECH11024

Vanga Aravind Shounik - Tester - CS20BTECH11055

Adepu Vasisht - Tester - CS20BTECH11002

November 25, 2022

Contents

Introduction

With GrAlgo we aim to bring the user closer to the Graph Data Structure which is not available in the general STL library for C++. Since our language is built on helping the user to use the power of graphs without writing any of his own functions which saves time, we have used syntax which is close to C++ i.e like C++ we also write the main part of our code which is to be executed in the main function call. If the user wants to write custom functions he can do so like C++ which needs the return type to be mentioned. Our programming language intuitively uses inbuilt functions and inbuilt data types which makes GrAlgo a powerful tool. Using our language the user can easily define different types of graphs and also traverse using Breadth-First and Depth-First methods using our inbuilt functions we also provide a wide range of methods to represent graphs such as "Node List", "Edge List" along with the normal usage.

While the main USP of GrAlgo is the graph data structure provided and the functions which we implement upon them, the user can also use GrAlgo like a general programming language making it versatile for general usage as well.

1.1 Uses

- Since C/Cpp does not have a dedicated library for graphs we have created a dsl solely for the usage of graphs.
  - Using a new data type named 'graphs' to represent graphs.
  - We are trying to implement 4 types of commonly used graphs and they are, Directed Graphs, Undirected Graphs, Weighted Graphs, and Unweighted Graphs.
  - Common kinds of graph traversals like DFS and BFS are inbuilt functions provided to user. We return a sequence of nodes in these traversal algorithms.
  - In real life graphs have to be used in different scenarios so our language provides an interface exclusively for graphs to tackle these problems

Language Tutorial

2.1 Getting Started

Lets get started with GrAlgo our language, user needs to have basic knowledge on C language and concepts of graphs. Graphs are used to solve many real life problems but contemporary languages does not provide graph specified interfaces. Our language provides a graph oriented interface with a graph type so that user gets a better experience in using graph algorithms. User has to have command on graphs and where to use them. Our program helps user in solving the real life programs involving graphs and variety of algorithms of graphs can be implemented more easily using our compiler. Further we provide basic tutorial of GrAlgo.

2.2 Basic Program Structure

We have a main function with int return type. All the variables are strongly typed and we have a semi colon (;) at end of each line. Basic int, float, char, string data types are used. graph datatype is created which stores a list of edges during initialization and each edge is int a, int b pair data structure which means the edge from a -¿ b.

```
int main()
{
    graph G = {
                2:3,
                1:2,
                3:1
              }
    return 0;
}
```

2.3 Variable Declaration

Following are the syntaxes of declaring variables used in GrAlgo. We must explicitly declare the variable with its datatype before use. we can also assign value while declaration. Multiple variables can be declared in a single line with same datatype name by separating each identifier with ','

```
int a , b , c; // variable declaration
float f = 5.5; //variable declaration with initialization
string s; char c; // string and char variables
dgraph g = {
            1:2,
```

```
              2:3
        } // graph declared with initialization
```

2.4 Function Declaration

The User can define his own functions similar to C and Cpp in which the return type needs to be mentioned. return type function name (function arguments) is the usual pattern for declaration. The code of function is written in curly braces. All the variables initialized in function are in function scope and cannot be used outside. Given below is a sum function defined with int, int arguments and int return type.

```
int sum(int a, int b){
    return a + b;
}
```

2.5 Compilation and Running

To build the compiler, the prerequisite softwares are

- Flex, Bison
- GCC, clang++
- LLVM-suite
- Make

The above softwares can be installed by the following commands in terminal

```
$ sudo apt install build-essential

$ sudo apt-get install flex bison clang-format clang-tidy clang-tools clang clangd
  libc++-dev libc++1 libc++abi-dev libc++abi1 libclang-dev libclang1 liblldb-dev
  libllvm-ocaml-dev libomp-dev libomp5 lld lldb llvm-dev llvm-runtime llvm python3
  -clang
```

Now, to build the compiler and run an examle , we have to go to the Semantics_and_CodeGen directory and run the following commands in bash

```
$ make
$ make test
$ ./a.out
```

Language Reference Manual

3.1 Lexical Conventions

3.1.1 Comments

The comments in this language follow the general C comment syntax.

- With the line ending comments starting with two forward slashes `// This is a valid comment`
- MultiLine comments are written in the following way **"/* Matter */"**.
- Nested comments and comments between strings are not allowed

```
// This is a comment
int a = 3; // This is also a comment
/*
This is a multiline comment
*/
 string a = "Aravind /* This is not a comment*/ Shounik" // This is considered a
    comment
```

3.1.2 Identifiers

Identifiers in our language need to follow certain rules, and they are

- The identifier should start with a letter (both capital and small included) or an underscore.
- Then any of the following can be used
  - Another character.
  - Another underscore.
  - Digit
- The regex for the identifier is given as follows `[_a-zA-Z][_0-9a-zA-Z]*`.
- Keywords cannot be used as identifiers

3.2 Keywords

The following are keywords in our language and cannot be used as identifers

```
if          else       int        float      return
for         while      string     char       break
continue    void       graph      dgraph     node
node_set    edge_set   edge_seq   BFS        DFS
neighbours  nodes      levels
```

3.2.1 Operators

Unary Operators

Purpose	Symbol	Associativity	Valid Operands
Parentheses for grouping of operations	( )	left to right	int, float
Increment	++	Right to Left	int, float
Decrement	--	Right to Left	int, float

Example:

```
int a = ++b;          // returns b added by 1
int a = --b;          // returns b substracted by 1
```

Arthimetic Operators

Purpose	Symbol	Associativity	Valid Operands
Modulo	%	Left to Right	int, float
Multiplication	*	Left to Right	int, float
Division	/	Left to Right	int, float
Addition	+	Left to Right	int, float, string
Subtraction	-	Left to Right	int, float

Example:

```
int a = 3 + 5;             // adds 3 and 5 to 8
float c = a - 1.2;         // subtracts a and 1.2 an int to get 7.8
int b = 9*5 ;              // multiplies 9 and 5 to get 45
int c = 8/3;               // divides 8 by 3 and returns the quotient 2
int d = 8%3;               // returns the remainder 2 when 8 is divided by 3
int a = (3+5)*4 + 3        // returns 35 as '()' has more taken precedence
```

Other Operators

Purpose	Symbol	Associativity	Valid Operands
Relational Operators	`<=`, `<`, `>=`, `>` `==`, `!=`	left to right	all data types
Bitwise Operators	`&`, `|`, `^`	left to right	bool
Logical Operators	`&&` `||`	left to right	bool
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`	right to left	wherever the operator is valid

Example:

```
//Relational operators
bool a = (3 < 5);               // returns true if 3 is less than 5
bool b = (17 <= 8);             // returns false as 17 is not less than or equal to 8
bool d = (4 == 4);              // return true as 4 is equal to 4
bool a = (2 > 1);               // return true as 2 is greater than 1
bool b = (4 >= 3);              // return true as 4 is greater than or equal to 3

// Logical Operators
bool a = (3<5)&&(4>2);          // returns AND of both the boolean expressions - true
bool b = (3>5)||(4<2);          // returns OR of both the boolean expressions - false
// Bitwise operators
int a = 3&5;                    // returns the bitwise AND of both the operators - 1
int b = 3|5;                    // returns the bitwise OR of both the operators - 7
//Assignment operators
int a = 1;                      // Assigns the value of 1 to the variable
int a += 1;                     // This adds 1 to the value of a and assigns it again
    giving 2
int a *= 4;                     // Assigns (a=2)*4 to a
int a /= 2;                     // Assigns (a=8)/2 to a
```

Project Plan

- **Week 0**
  - Discussed various ideas on languages came up with a creative idea which is feasible
  - Decided on the tools to be used, the structure of the compiler
  - Finalised the roles of the team members
- **Week 1**
  - Discussed various features of the language and their feasibility
  - Then finalised the language and compiled the language specification document
  - Decided the syntax of the language along with graph implementation
  - Studied through these reference papers thoroughly on graph DSLs 1. Green marl 2. Ligra
- **Week 2**
  - Changes in the syntax of the language
  - Finding ambiguities in our grammar
  - Designing the lexer
  - Testing the lexer
- **Week 3**
  - Considered ways in which lexer could be written and decided on flex
  - Finished writing 70% of lexer using Flex
  - Added more test cases for the lexer
- **Week 4**
  - Made ppt and videos about the implementation of our lexer
  - Fixed more bugs in Parser
  - Studied and analysed c and cpp parsers
- **Week 5**
  - Working on the parser, integrating it with lexer
  - Learnt how to use Bison for parsing
  - Completed 30% of the parser

- **Week 6**
  - Completed Parser
  - Studied **Generating AST in yacc from flex and bison - orielly**
  - Working on generating AST
- **Week 7**
  - Completed AST generation
  - Exploring different methods to implement semantic phase
  - Explored features of attribute grammar
- **Week 8**
  - Created symbol table
  - integrating symbol table with the parser and lexer
  - inspected semantic analysers of basic languages
- **Week 9**
  - Shifted from C in bison to CPP in bison
  - Revised the lexer and parser (ver 2.0) in cpp
  - Implemented symbol table
  - Working on AST
- **Week 10**
  - Progressed with semantic analysis
  - Worked on error displays and design principles
  - Considered on using Constant folding optimization
- **Week 11**
  - Integrated symbol table with ast
  - Worked on type checking in semantic analysis
  - Studied llvm's kaleidoscope manual for code generation
- **Week 12**
  - Added more on type checking in semantics
  - Worked on code generation
  - Explored LLVM documentation on the symbol table provided by LLVM

- **Week 13**
  - Discussions to implement data structures
  - Worked on code generation for language specific features
- **Week 14**
  - Completed Code Generation
  - Made Final - Report and Presentation
  - Made Demo Video and presentation videos

Language Evolution

Our Language aim is to provide a simpler user interface to implement graphs and its algorithms with graph specified implementations. We followed syntax similar to C to maintain integrity and for easy use. Throughout the timeline we made necessary modifications in our grammar and syntax to reach our goal. First we chose orielly book as our resource for lexical analysis and parsing. We faced lot of complications in the process but successfully done until parsing phase. We faced issues with orielly resource to perform semantic analysis , though we implemented symbol table it was not sufficient to perform all our functions.

We changed our resource and started using bison and flex with cpp. Then we progressed our steadily and implemented semantic checks. I made necessary changes in our syntax and grammar time to time based on use and time. Though initially a variety of datatypes were decided to be implemented in the long run we only retained data types which have more significance in our project. Our testers provides with different testcases and helped us to correct our mistakes which were overlooked. Our Language was dynamic with a lot of trail and errors we added features that were necessary.

The code generation part was one of the challenging phase of the project it was not easy to integrate our code with llvm IR. We worked thoroughly on this phase and implemented code generation of most part of our language. We did not use any other language like a transpiler, we successfully made our code into IR representation and executed the output file. Finally we gave our best efforts in this project for timeline of 7 weeks and implemented GrAlgo's lexical analysis, parsing, semantic analysis and code generation. Our language was thoroughly evolved and made with optimal grammar and syntax.

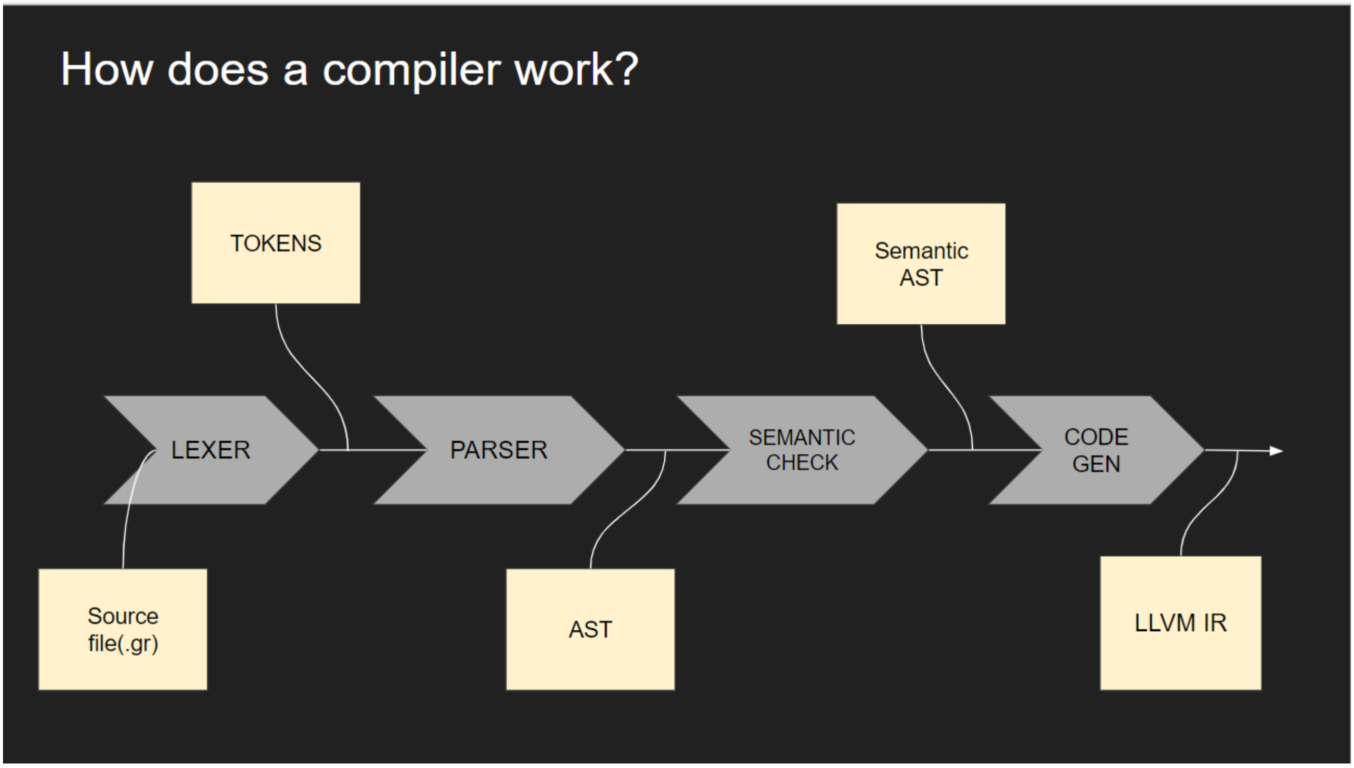
Stages of Evolution in GrAlgo

- Lexer with variety of keywords and operations
- Eliminated unnecessary operations and keywords and parsed language optimally
- Implemented semantic checks
- Generated LLVM IR for our language and added inbuilt functions

Compiler Architecture

The components of the compiler are

1. Lexer
2. Parser
3. Semantic
4. Code Generation



6.1 Lexer

Lexer converts a sequence of characters into a sequence of tokens.errors like invalid characters,incomplete multi line comments are handled with white spaces getting ignored. This generated token stream is taken to the next step i.e parsing.

6.2 Parser

Parser takes the tokens produced by lexer and matches with grammar rules to form Abstract Syntax Tree. Syntax errors will be handled here, the abstract syntax tree generated will still contain semantic errors and we need to do semantics to generate a better form of abstract syntax tree.

6.3 Semantic Analyzer

The semantic check takes an AST and semantically checks it. Semantic Analysis is the process of drawing meaning from a text, Ensuring the declarations and statements of a program is done in this process. Functions of semantic analysis are:-

- **Type Checking**: Makes sure that each operator has matching operands or in other words ensures that data types are used in a way consistent with their definition.
- **Label Checking**: Every program must contain labels references.
- **Flow Control Check**:Keeps a track of whether the control structures are used in proper manner or not. It occurs during compile time and run time.

It also checks traditional conditions such as the existence of a variable within a specified scope or type consistencies for assignments. The final output is semantically checked **AST**

6.4 Intermediate Code Generation

The code generator takes in the semantically checked ast to generate LLVM IR code which we can use alongside with the LLVM compiler to generate machine specific assembly code.

Development Environment

7.1 GNU Make

We used GNU make as one of the tool as we had many files and many things to compile and **make** made our work easy from compiling and converting all the files into objects to cleaning all the intermediaries, we used **make** from lexical phase since we always have multiple commands to execute make made our work easier by executing the file and getting a.out with single command The advantages that we learned about make are:

- Make enables the end user to build and install your package without knowing the details of how that is done – because these details are recorded in the makefile that you supply.
- Make figures out automatically which files it needs to update, based on which source files have changed. It also automatically determines the proper order for updating files, in case one non-source file depends on another non-source file.
  As a result, if you change a few source files and then run Make, it does not need to recompile all of your program. It updates only those non-source files that depend directly or indirectly on the source files that you changed.
- GNU Make has many powerful features for use in makefiles, beyond what other Make versions have. It can also regenerate, use, and then delete intermediate files which need not be saved.

Our final dependency graph looks like the one in the below figure.

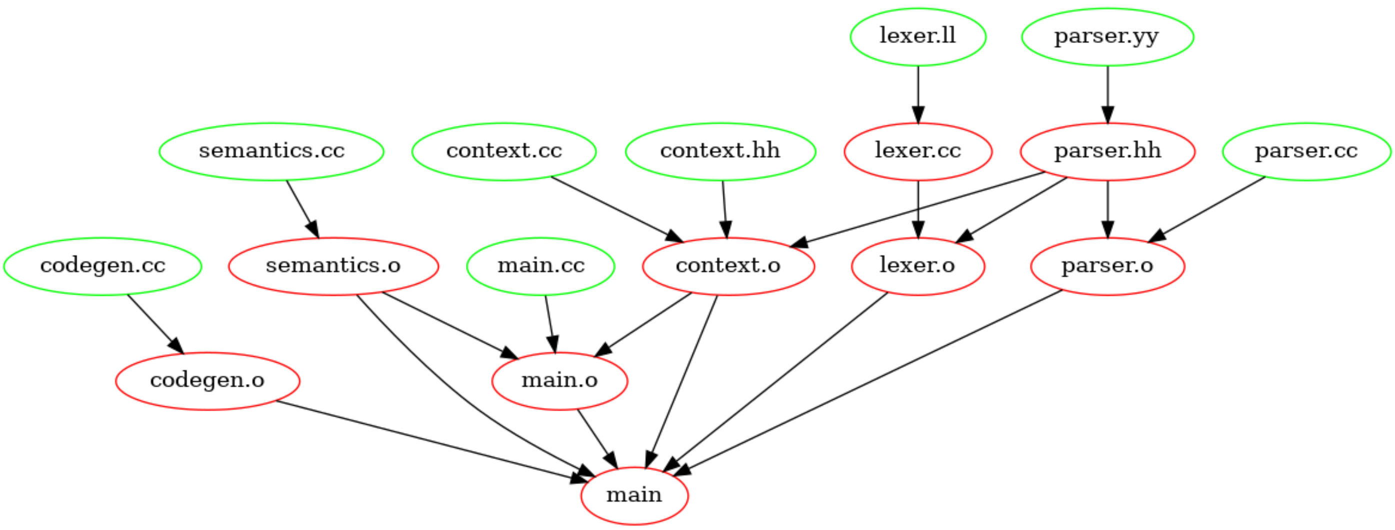


Figure 7.1: Makefile Dependencies

7.2 Git

We used Git as our version control system. Where we pushed our work into a remote on GitHub whenever one of us finished working on some feature, we have done some code review and then modified pulled, pushed etc. We pushed ppts and videos at the end of every phase of our project.

7.3 VSCode

We got into many problems and some of them wanted to be solved at very quickly. All of us Used VSCode particularly **liveshare** when we ran into problems. We were sometimes successful in rectifying the mistakes. As VSCode provided liveshare which had many nice features such as Terminal sharing and Code sharing, syntax highlighting etc, we included it in one of our Tools. It also has source control option where we can use Github features without typing any commands.

Test Plan and Test Suites

8.1 Introduction

We have tested with a good number of testcases which handles all syntax types like arithmetic operations, inbuilt functions, semantic checks etc. The following are few testcases to understand expected output and error cases to understand language properly

8.2 Test Cases

```
/* Basic Program*/
  int main()
{
    return 0;
}
```

```
output: (since we are not doing anything)
none
```

```
/*Initializing the datatypes*/
int main()
{
  int a = 3;
  float b = 10.0;
  graph G = {1:2,2:3,3:4};
  print(a);
  print(b);

  return 0;
}
```

```
3
10.000000
```

```
/* If else conditional statement */
int main()
{
  int a = 10;
  if(a)
  {
    print(a);
  }
  else
  {
```

```
    print(0);
  }
  return 0;
}
```

```
10
```

```
int main()
{
  int a = 10;
  float b = 100.0;
  int c = 100;
  int d = a+c;
  print(d);
  return 0;
}
```

```
110
```

```
int main()
{
  int a = 10;
  print(a);
  a = a + 1;
  print(a);
  return 0;
}
```

```
10
11
```

```
int main()
{
    int a = 10;
    while(a)
    {
        print(a);
        a = a-1;
    }
    return 0;
}
```

```
10
9
8
7
6
5
4
3
2
1
```

```
int main()
{
  graph G = {1:2, 2:3, 3:4};
  BFS(node d :G)
  {
    print(d);
  }
  return 0;
}
```

```
2 1 3 4
```

```
int main()
{
  graph G = {1:2, 2:3, 3:4};
  DFS(node d :G)
  {
    print(d);
  }
  return 0;
}
```

```
2 3 4 1
```

```
int main()
{
  int a = 5;
  print(a, a-2, a + 5);
  return 0;
}
```

```
5 3 10
```

```
int sum(int a, int b)
{

  return a+b;
}

int main()
{
  int x;
  x = sum(10,2);
  int a = 11;
  int b = 2;
  int y = sum(a,2);
  int z = sum(a,b);
  int k = sum(sum(a,b),b);
  print(x,y,z,k);
  return 0;
}
```

```
12 13 13 15
```

8.3 Error Codes

```
int main()
{
  int a = 3

 return 0;
}
```

```
5:2 error: syntax error, unexpected return, expecting COMMA or SEMI_COLON
```

```
int main()
{
  float c =
  return 0;
}
```

```
5:3 error: syntax error, unexpected return
```

```
int main()
{
  graph g = {2:,3:5,1:2};
  return 0;
}
```

```
3:16 error: syntax error, unexpected COMMA, expecting number
```

```
int main()
{
  int x = 5;
  int x = 10;
  return 0;
}
```

```
4:13 error: Duplicate definition <x>
```

```
int main()
{
  int j = 40;
  i = 20;
  return 0;
}
```

```
4:5 error: Undefined identifier <i>
```

Conclusions and Lessons Learnt

This project was a great learning experience. In our interns preparation for graph based questions we were always asked to implement those long algorithms we learnt those algorithms but it is redundant to implement those long algorithms from scratch thus we thought of this idea to create a domain specific language which has inbuilt algorithms and easy to use. So, to achieve our goal we started working on it

We started from lexical phase and concluded at code generation phase we learnt the required technical concepts of that phase at start of week and continued to built upon that knowledge. After deciding grammar of GrAlgo,we began with lexical phase where we learnt **Flex** features and built lexer based on it. Later we started with parsing phase where we learnt **Bison** features, reffered flex-bison Oreilly to integrate lexer with parser in this phase we learnt complications in grammar and tried to avoid shift-reduce and reduce-reduce conflicts here we understood that to build language to achieve our goal we had to make tradeoff over simple operations so we decided to remove few keywords and operations and completed parsing.

Then we have entered semantic phase where we understood that we were naive to think that we can complete everything in C. C language has its limitations in semantic phase we wanted a better environment to provide semantics checks in our language we updated our language with C++ an integrated our lexer and parser with C++ by refering to few examples in bison-flex-cpp example. In cpp we used enum classes and successfully created a symbol table using cpp we concluded our semantic phase by implementing few semantic checks on our language.

Finally we reached code generation phase our project where we inspected llvm - suite and learnt how to link llvm constructs to different data types and functions of our language. In this phase we used the generated IR to compile our code and added inbuilt functions to add our features.

This project was a great journey on compilers we performed required tests at every phase to ensure syntactical correctness of program. We changed grammar and keywords when ever necessary to reach our goal.

Appendix

10.1 Lexer Code (lexer.ll)

```
    /* Regex abbreviations: */
id     [a-zA-Z_][a-zA-Z_0-9]*
int    [0-9]+
blank  [ \t]
EXP  ([Ee][-+]?[0-9]+)

%%

%{
    // Code run each time yylex is called.
    loc.step ();
%}

{blank}+   loc.step ();
[\n]+      {loc.lines (yyleng); loc.step (); }

"-"       return yy::parser::make_MINUS(loc);
"+"       return yy::parser::make_PLUS(loc);
"*"       return yy::parser::make_STAR(loc);
"/"       return yy::parser::make_SLASH(loc);
"("       return yy::parser::make_LPAREN(loc);
")"       return yy::parser::make_RPAREN(loc);
"="      return yy::parser::make_ASSIGN(loc);
"%"      return yy::parser::make_MOD(loc);
"|"      return yy::parser::make_B_OR(loc);
","      return yy::parser::make_COMMA(loc);
";"      return yy::parser::make_SEMI_COLON(loc);
":"      return yy::parser::make_COLON(loc);
"{"      return yy::parser::make_LBRACE(loc);
"}"      return yy::parser::make_RBRACE(loc);
"["      return yy::parser::make_LSB(loc);
"]"      return yy::parser::make_RSB(loc);
"&"      return yy::parser::make_AMPERSAND(loc);


"&&" {return yy::parser::make_AND(loc); }
"||" {return yy::parser::make_OR(loc);}
"++" {return yy::parser::make_PP(loc);}
"--" {return yy::parser::make_MM(loc);}

 /* Assignment Ops */
"+="    {return yy::parser::make_PL_EQ(loc); }
```

```
"-="    {return yy::parser::make_MI_EQ(loc);}
"*="    {return yy::parser::make_MU_EQ(loc);}
"/="    {return yy::parser::make_DI_EQ(loc);}

 /* comparison ops */
">"     { return yy::parser::make_LESS(loc); }
"<"     { return yy::parser::make_GREATER(loc); }
"!="    { return yy::parser::make_NE(loc); }
"=="    { return yy::parser::make_EQ(loc); }
">="    { return yy::parser::make_GEQ(loc); }
"<="    { return yy::parser::make_LEQ(loc); }


"continue"     {return yy::parser::make_CONTINUE(loc);}
"break"     {return yy::parser::make_BREAK(loc);}
"return"     {return yy::parser::make_RETURN(loc);}


 /* type specifiers */
"void"     {return yy::parser::make_VOID(loc);}
"int"      {return yy::parser::make_INT(loc);}
"bool"      {return yy::parser::make_BOOL(loc);}
"float"    {return yy::parser::make_FLOAT(loc);}
"char"    {return yy::parser::make_CHAR(loc);}
"string"    {return yy::parser::make_STRING(loc);}
"graph"    {return yy::parser::make_GRAPH(loc);}
"dgraph"    {return yy::parser::make_DGRAPH(loc);}
"node"    {return yy::parser::make_NODE(loc);}
"node_set"    {return yy::parser::make_NODE_SET(loc);}
"node_seq"    {return yy::parser::make_NODE_SEQ(loc);}
"node_prop"    {return yy::parser::make_NODE_PROP(loc);}
"edge_prop"    {return yy::parser::make_EDGE_PROP(loc);}
"edge_set"    {return yy::parser::make_EDGE_SET(loc);}
"edge_seq"    {return yy::parser::make_EDGE_SEQ(loc);}


 /* keywords */
"if"        { return yy::parser::make_IF(loc); }
"else"      { return yy::parser::make_ELSE(loc); }
"while"     { return yy::parser::make_WHILE(loc); }
"for"       { return yy::parser::make_FOR(loc);}
"BFS"       {return yy::parser::make_BFS(loc);}
"DFS"       {return yy::parser::make_DFS(loc);}
"nodes"     {return yy::parser::make_NODES(loc);}
"levels"    {return yy::parser::make_LEVELS(loc);}
"neighbours" {return yy::parser::make_NEIGHBOURS(loc);}


{int}      {
    errno = 0;
    long n = strtol (yytext, NULL, 10);
```

```
    if (! (INT_MIN <= n && n <= INT_MAX && errno != ERANGE))
        ctx.error (loc, "integer is out of range");
    return yy::parser::make_NUMBER(n, loc);
}

[0-9]+"."[0-9]*{EXP}? |
"."[0-9]+{EXP}? { return yy::parser::make_DOUBLE_CONST(atof(yytext), loc); }

 /* strings */
\"(\\.|[^\\"])*\" {return yy::parser::make_STRING_LITERAL(yytext, loc);}

{id}        return yy::parser::make_IDENTIFIER(yytext, loc);

.           ctx.error (loc, "invalid character");

<<EOF>>     return yy::parser::make_END(loc);
```

10.2 Parser Code (parser.yy)

```
%code
{
#include "context.hh"
#define yylex ctx.lexer.yylex

#define M(x) std::move(x)
#define C(x) node(x)
}

// Tokens:
%define api.token.prefix {TOK_}
%token
  END  0  "end of file"
  ASSIGN  '='
  MINUS   '-'
  PLUS    '+'
  STAR    '*'
  SLASH   '/'
  AMPERSAND '&'
  LPAREN  '('
  RPAREN  ')'
  MOD '%'
  B_OR '|'
  COMMA ','
  SEMI_COLON ';'
  COLON ':'
  LBRACE '{'
  RBRACE '}'
  LSB '['
  RSB ']'
```

```
  AND "&&"
  OR "||"
  PP "++"
  MM "--"
  PL_EQ "+="
  MI_EQ "-="
  MU_EQ "*="
  DI_EQ "/="
  LESS "<"
  GREATER ">"
  NE "!="
  EQ "=="
  GEQ ">="
  LEQ "<="

  BREAK "break"
  CONTINUE "continue"
  RETURN "return"
  IF "if"
  ELSE "else"
  WHILE "while"
  FOR "for"
  BFS "BFS"
  DFS "DFS"
;
%token
  VOID "void"
  INT "int"
  BOOL "bool"
  FLOAT "float"
  CHAR "char"
  STRING "string"
  GRAPH "graph"
  DGRAPH "dgraph"
  NODE "node"
  NODE_SET "node_set"
  NODE_SEQ "node_seq"
  NODE_PROP "node_prop"
  EDGE_PROP "edge_prop"
  EDGE_SET "edge_set"
  EDGE_SEQ "edge_seq"
  NODES "nodes"
  LEVELS "levels"
  NEIGHBOURS "neighbours"
;

// Use variant-based semantic values: %type and %token expect genuine types
%token <std::string> IDENTIFIER "identifier" STRING_LITERAL
%token <int> NUMBER "number"
%token <double> DOUBLE_CONST "double_const"
%type<std::string> identifier
```

```
%type<node> expr exprs stmt selection_stmt jump_stmt expression_stmt iteration_stmt
    vardec_stmt empty_stmt compound_stmt p_expr initializer initializer_list edge
%type<type_name> typename

/* Operator precedence */
/* %left  COMMA */
%right '?' COLON ASSIGN PL_EQ MI_EQ
%left  OR
%left  AND
%left  EQ NE
%left  PLUS MINUS
%left  STAR SLASH MOD
%right AMPERSAND PP MM
%precedence  LPAREN LSB

%precedence LOWER_THAN_ELSE
%precedence ELSE

// No %destructors are needed, since memory will be reclaimed by the
// regular destructors.
/* %printer { yyoutput << $$; } <*>; */

// Grammar:
%%
%start program;

program: { ++ctx; } declarations { --ctx; };
declarations: declarations declaration
|              %empty
;
declaration: function
|             vardec_stmt SEMI_COLON { ctx.add_decl(M($1)); }
;

function: typename identifier { ctx.defun($2); ++ctx; } LPAREN paramdecls RPAREN
   compound_stmt RBRACE { ctx.add_function(M($2), M($7), $1); --ctx; }
;
paramdecls: paramdecl
|            %empty
;
paramdecl:  paramdecl COMMA typename identifier { ctx.defparam($4, $3); }
|            typename identifier { ctx.defparam($2, $1); }
;
typename: VOID { $$ = type_name::VOID; }
|          INT { $$ = type_name::INT; }
|          BOOL { $$ = type_name::BOOL; }
|          CHAR { $$ = type_name::CHAR; }
|          FLOAT { $$ = type_name::FLOAT; }
|          STRING { $$ = type_name::STRING; }
|          GRAPH { $$ = type_name::GRAPH; }
|          DGRAPH { $$ = type_name::DGRAPH; }
```

```
|           NODE_SET { $$ = type_name::NODE_SET; }
|           EDGE_SET { $$ = type_name::EDGE_SET; }
|           NODE_PROP '<' identifier '>'
|           NODE_SEQ '<' identifier '>'
|           EDGE_PROP '<' identifier '>'
|           EDGE_SEQ '<' identifier '>'
;

stmt: compound_stmt RBRACE { $$ = M($1); --ctx; }
|      selection_stmt
|      jump_stmt
|      expression_stmt
|      empty_stmt
|      vardec_stmt SEMI_COLON { $$ = $1; }
|      iteration_stmt
;
expression_stmt: exprs SEMI_COLON { $$ = M($1); }
;
jump_stmt: CONTINUE SEMI_COLON { $$ = n_cont(); }
|           BREAK SEMI_COLON  { $$ = n_br(); }
|           RETURN SEMI_COLON  { $$ = n_ret(); }
|           RETURN expr SEMI_COLON                  { $$ = n_ret(M($2));          }
;
empty_stmt: SEMI_COLON
;
vardec_stmt: typename identifier ASSIGN initializer { ctx.temptype = $1; $$ = n_vardec(); $$.params.push_back(M(ctx.def($2) %= M($4))); }
|             typename identifier { ctx.temptype = $1; $$ = n_vardec(); $$.params.push_back(M(ctx.def($2) %= n_nop())); }
|             vardec_stmt COMMA identifier ASSIGN initializer { $$ = M($1); $$.params.push_back(M(ctx.def($3) %= M($5))); }
|             vardec_stmt COMMA identifier { $$ = M($1); $$.params.push_back(M(ctx.def($3) %= n_nop())); }
;

initializer: expr
|             edge
|             LBRACE initializer_list RBRACE { $$ = M($2); }
;
initializer_list: initializer { $$ = n_init_list(M($1)); }
|                  initializer_list COMMA initializer { $$ = M($1); $$.params.push_back($3); }
;
edge: NUMBER COLON NUMBER { $$ = n_edge($1, $3); }
;

compound_stmt:  LBRACE { $$ = n_comma(); ++ctx; }
|                compound_stmt stmt { $$ = M($1); $$.params.push_back(M($2)); }
;
selection_stmt: IF p_expr stmt %prec LOWER_THAN_ELSE  { $$ = n_cond(M($2), M($3), n_comma()); }
```

```
|                IF p_expr stmt ELSE stmt   { $$ = n_cond(M($2),M($3),M($5)); }
;
iteration_stmt: WHILE p_expr stmt           { $$ = n_loop(M($2), M($3)); }
|                FOR LPAREN expr SEMI_COLON expr SEMI_COLON expr RPAREN stmt { $$ =
    n_loop(M($3), M($5), M($7), M($9)); }
|                FOR LPAREN typename identifier COLON identifier RPAREN stmt { $$ =
    n_loop(M($8)); }
|                BFS LPAREN typename identifier COLON identifier RPAREN stmt { $$ =
    n_loop(M($8)); }
|                DFS LPAREN typename identifier COLON identifier RPAREN stmt { $$ =
    n_loop(M($8)); }
;
p_expr: LPAREN expr RPAREN { $$ = M($2); }
;
exprs: expr                         { $$ = n_comma(M($1)); }
|      exprs COMMA expr             { $$ = M($1); $$.params.push_back(M($3)); }
;

expr: NUMBER                        { $$ = $1;    }
|     DOUBLE_CONST                  { $$ = $1; }
|     STRING_LITERAL                { $$ = M($1); }
|     identifier                    { $$ = ctx.use($1);   }
|     LPAREN exprs RPAREN                 { $$ = M($2); }
|     expr LSB exprs RSB      { $$ = n_deref(n_add(M($1), M($3))); }
|     identifier LPAREN RPAREN                  { $$ = n_fcall(ctx.use($1), n_comma());
    }
|     identifier LPAREN exprs RPAREN            { $$ = n_fcall(ctx.use($1), M($3)); }
|     expr ASSIGN expr                 { $$ = (M($1) %= M($3)); }
|     expr PLUS expr                 { $$ = n_add( M($1), M($3)); @$ = @2; }
|     expr MINUS expr %prec PLUS     { $$ = n_add( M($1), n_neg(M($3))); }
|     expr STAR expr                 { $$ = n_mul( M($1), M($3)); }
|     expr SLASH expr %prec STAR     { $$ = n_div( M($1), M($3)); }
|     expr MOD expr                 { $$ = n_mod( M($1), M($3));}
|     expr "+=" expr                //{ if(!$3.is_pure()) { $$ = ctx.temp() %=
    node_addrof(M($1)); $1 = node_deref($$.params.back()); } $$ = node_comma(M($$),
    M($1) %= node_add(C($1), M($3))); }
|     expr "-=" expr                //{ if(!$3.is_pure()) { $$ = ctx.temp() %=
    node_addrof(M($1)); $1 = node_deref($$.params.back()); } $$ = node_comma(M($$),
    M($1) %= node_add(C($1), node_neg(M($3)))); }
|     "++" expr                     //{ if(!$2.is_pure()) { $$ = ctx.temp() %=
    node_addrof(M($2)); $2 = node_deref($$.params.back()); } $$ = node_comma(M($$),
    M($2) %= node_add(C($2),  1l)); }
|     "--" expr %prec PP      //{ if(!$2.is_pure()) { $$ = ctx.temp() %=
    node_addrof(M($2)); $2 = node_deref($$.params.back()); } $$ = node_comma(M($$),
    M($2) %= node_add(C($2), -1l)); }
|     expr "++"                     //{ if(!$1.is_pure()) { $$ = ctx.temp() %=
    node_addrof(M($1)); $1 = node_deref($$.params.back()); } auto i = ctx.temp(); $$
     = node_comma(M($$), C(i) %= C($1), C($1) %= node_add(C($1),  1l), C(i)); }
|     expr "--" %prec PP      //{ if(!$1.is_pure()) { $$ = ctx.temp() %=
    node_addrof(M($1)); $1 = node_deref($$.params.back()); } auto i = ctx.temp(); $$
     = node_comma(M($$), C(i) %= C($1), C($1) %= node_add(C($1), -1l), C(i)); }
```

```
|     expr OR expr              { $$ = n_cor( M($1), M($3)); }
|     expr AND expr              { $$ = n_cand(M($1), M($3)); }
|     expr EQ expr              { $$ = n_eq(  M($1), M($3)); }
|     expr NE expr %prec EQ { $$ = n_eq(n_eq(M($1), M($3)), 0); }
|     AMPERSAND expr                  { $$ = n_addrof(M($2)); }
|     STAR expr  %prec AMPERSAND        { $$ = n_deref(M($2));  }
|     MINUS expr  %prec AMPERSAND        { $$ = n_neg(M($2));    }
|     '!' expr  %prec AMPERSAND        { $$ = n_eq(M($2), 0); }
|     expr '?' expr COLON expr    //{ auto i = ctx.temp(); $$ = node_comma(node_cor
    (node_cand(M($1), node_comma(C(i) %= M($3), 1l)), C(i) %= M($5)), C(i)); }
;

identifier: IDENTIFIER                 { $$ = M($1); };

%%

// Register errors to the driver:
void yy::parser::error (const location_type& l,
                          const std::string& m)
{
    ctx.error(l, m);
}
```

10.3 Semantics Code (semantics.cc)

```
std::vector<std::string> doSemantics(std::vector<common_list> &ast)
{
  std::vector<std::string> error_list;
  for (auto &cn : ast)
  {
    if (cn.isFunc) //
    {
      auto &f = cn.f;
      func_map[f.name] = &f;
      type_name ret = f.ret_type;
      bool hasRetStmt = false;
      for (auto &stmt : f.code.params)
      {
        try
        {
          // if it returns, do semantics on it
          if (stmt.type == node_type::ret)
          {
            hasRetStmt = true;
            // check return type of function
            if (ret == type_name::VOID && stmt.params.size() > 0)
            {
              throw Exception("Unexpected return statement.");
            }
```

```
            else if (ret != type_name::VOID && stmt.params.size() == 0)
            {
              throw Exception("Return statement doesn't return anything.");
            }
            else if (ret != type_name::VOID && stmt.params.size() > 0)
            {
              if (ret != doSemantics(stmt.params[0]))
              {
                throw Exception("Return types don't match.");
              }
            }
          }
          else
          {
            doSemantics(stmt);
          }
        }
        catch (Exception &e)
        {
          std::cerr << e.loc.begin.line << ":" << e.loc.begin.column << " error: "
    << e.msg << std::endl;
        }
      }
      if (ret != type_name::VOID && !hasRetStmt)
      {
        try
        {
          throw Exception("Expected return statement.");
        }
        catch (Exception &e)
        {
          std::cerr << e.loc.begin.line << ":" << e.loc.begin.column << " error: "
    << e.msg << std::endl;
        }
      }
    }
    else
    {
      // declaration type should match rhs
      try
      {
        doSemantics(cn.n);
      }
      catch (Exception &e)
      {
        std::cerr << e.loc.begin.line << ":" << e.loc.begin.column << " error: " <<
     e.msg << std::endl;
      }
    }
  }
  return error_list;
```

```
}

type_name doSemantics(const node &n)
{
  type_name ret1, ret2;
  ret1 = ret2 = type_name::VOID;
  switch (n.type)
  {
  case node_type::number:
    return type_name::INT;

  case node_type::double_const:
    return type_name::FLOAT;

  case node_type::string:
    return type_name::STRING;

  case node_type::identifier:
    return n.ident.v_type;

  case node_type::add:
    if ((ret1 = doSemantics(n.params[0])) != (ret2 = doSemantics(n.params[1])))
    {
      throw Exception(n.loc, "+ of different types");
    }
    break;

  case node_type::neg:
    return doSemantics(n.params[0]);

  case node_type::mul:
    if ((ret1 = doSemantics(n.params[0])) != (ret2 = doSemantics(n.params[1])))
    {
      throw Exception(n.loc, "* of different types");
    }
    break;

  case node_type::div:
    if ((ret1 = doSemantics(n.params[0])) != (ret2 = doSemantics(n.params[1])))
    {
      throw Exception(n.loc, "/ of different types");
    }
    break;

  case node_type::mod:
    if ((doSemantics(n.params[0]) != type_name::INT) || (doSemantics(n.params[1]) != type_name::INT))
    {
      throw Exception(n.loc, "% of non-integers");
    }
    ret1 = type_name::INT;
```

```
    break;

  case node_type::eq:
    if ((ret1 = doSemantics(n.params[0])) != (ret2 = doSemantics(n.params[1])))
    {
      throw Exception(n.loc, "== of different types");
    }
    ret1 = type_name::BOOL;
    break;

  case node_type::cor:
    if ((ret1 = doSemantics(n.params[0])) != (ret2 = doSemantics(n.params[1])))
    {
      throw Exception(n.loc, "|| different types");
    }
    ret1 = type_name::BOOL;
    break;

  case node_type::cand:
    if ((ret1 = doSemantics(n.params[0])) != (ret2 = doSemantics(n.params[1])))
    {
      throw Exception(n.loc, "&& different types");
    }
    ret1 = type_name::BOOL;
    break;

  case node_type::ret:
    if (n.params.size() > 0)
      ret1 = doSemantics(n.params[0]);
    break;

  case node_type::copy:
  {
    if ((ret1 = doSemantics(n.params[0])) != (ret2 = doSemantics(n.params[1])))
    {
      throw Exception(n.loc, "= different types");
    }
    return type_name::VOID;
  }

  case node_type::vardec:
    for (auto &inits : n.params)
    {
      ret1 = doSemantics(inits);
    }
    return type_name::VOID;

  case node_type::br:
    return type_name::VOID;

  case node_type::cont:
```

```
    return type_name::VOID;

  case node_type::fcall:
  {
    if (n.params[0].ident.type != id_type::function)
    {
      throw Exception(n.loc, "Expected a function name.");
    }
    auto f = func_map[n.params[0].ident.name];

    if (n.params[1].params.size() > f->num_params)
    {
      throw Exception(n.loc, "Expected fewer arguments.");
    }
    else if (n.params[1].params.size() < f->num_params)
    {
      throw Exception(n.loc, "Too few arguments.");
    }

    unsigned i = 0;
    for (i = 0; i < f->num_params; i++)
    {
      if (doSemantics(n.params[1].params[i]) != f->param_types[i])
      {
        throw Exception(n.loc, "Expected argument of type " + toString(f->
  param_types[i]));
      }
    }
    return f->ret_type;
  }

  case node_type::init_list:
  {
    auto prev = node_type::nop;
    unsigned size = 0;
    for (auto &p : n.params)
    {
      doSemantics(p);
      if (prev == node_type::nop)
      {
        prev = p.type;
        size = p.params.size();
      }
      else
      {
        if (p.type != prev)
        {
          throw Exception(n.loc, "Expected init list of same types.");
        }
        if (p.params.size() != size)
        {
```

```
          throw Exception(n.loc, "Expected init lists of same sizes.");
        }
      }
    }
    /*
    pending:
    recursive type check also for children
    */
    return type_name::INT;
  }

  default:
    break;
  }
  return ret1;
}
```

10.4 Code Generation Code (codegen.cc)

```
static void InitializeModuleAndPassManager(void)
{
  TheFPM->add(createCFGSimplificationPass());

  TheFPM->doInitialization();
}

static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction, Type *Ty, StringRef VarName)
{
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(), TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Ty, nullptr, VarName);
}

void doCodeGen(const std::vector<common_list> &ast)
{
  InitializeModuleAndPassManager();

  AddBuiltInFuncs();

  for (auto &cn : ast)
  {
    if (cn.isFunc)
    {
      // std::cout << "## codegen.cc line 21\n";
      HandleFunction(cn.f);
    }
    else
    {
```

```
      HandleNode(cn.n);
    }
  }

  raw_ostream *out = &errs();
  std::error_code EC;
  out = new raw_fd_ostream("test.ll", EC);
  TheModule->print(*out, nullptr);
}

void HandleNode(const node &n)
{
  Value *ir = codegen(n);
}

void HandleFunction(const function &f)
{
  codegen(f);
}

/* Util functions for code generation */
Type *convertType(type_name Ty)
{
  switch (Ty)
  {
  case type_name::INT:
    return Type::getInt32Ty(*TheContext);

  case type_name::FLOAT:
    return Type::getDoubleTy(*TheContext);

  case type_name::BOOL:
    return Type::getInt1Ty(*TheContext);

  case type_name::VOID:
    return Type::getVoidTy(*TheContext);

  case type_name::GRAPH:
  {
    return createGraph();
  }
  case type_name::NODE_SET:
  {
    return ArrayType::get(convertType(type_name::INT), 10);
  }
  default:
    break;
  }
  return nullptr;
}

```

```
Function *codegen(const function &f)
{
  // std::cout << "## Entered function codegen func. line 103\n";
  std::vector<Type *> param_types(f.param_types.size(), nullptr);
  for (auto i = 0; i < f.param_types.size(); i++)
    param_types[i] = convertType(f.param_types[i]);

  FunctionType *FT = FunctionType::get(convertType(f.ret_type), param_types, false);
  funcList[f.name] = FT;

  Function *F = Function::Create(FT, Function::ExternalLinkage, f.name, TheModule.get());

  // Set names for all arguments
  unsigned Idx = 0;
  for (auto &Arg : F->args())
    Arg.setName(f.param_names[Idx++]);

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(*TheContext, "entry", F);
  Builder->SetInsertPoint(BB);

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : F->args())
  {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(F, Arg.getType(), Arg.getName());

    // Store the initial value into the alloca.
    Builder->CreateStore(&Arg, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[std::string(Arg.getName())] = Alloca;
  }
  emit(f.code.params);
  return F;
}

Value *codegen(const node &n)
{
  switch (n.type)
  {
  case node_type::number:
    return ConstantInt::get(*TheContext, APInt(32, n.numvalue, true));

  case node_type::double_const:
    return ConstantFP::get(*TheContext, APFloat(n.doublevalue));

  case node_type::string:
```

```
      break;

  case node_type::identifier:
  {
    Value *V = NamedValues[n.ident.name];
    if (!V)
      fprintf(stderr, "Error: Unknown variable name\n");

    // Load the value.
    return Builder->CreateLoad(convertType(n.ident.v_type), V, n.ident.name.c_str()
  );
  }

  case node_type::add:
  {
    Value *L = codegen(n.params[0]);
    Value *R = codegen(n.params[1]);

    if (!L || !R)
      return nullptr;
    auto Inst = BinaryOperator::CreateAdd(L, R, "addtmp");
    auto block = Builder->GetInsertBlock();
    block->getInstList().push_back(Inst);
    return Inst;
  }

  case node_type::mul:
  {
    Value *L = codegen(n.params[0]);
    Value *R = codegen(n.params[1]);

    if(!L || !R)
      return nullptr;
    auto Inst = BinaryOperator::CreateMul(L,R,"multmp");
    auto block = Builder->GetInsertBlock();
    block->getInstList().push_back(Inst);
    return Inst;
  }

  case node_type::div:
  {
    Value *L = codegen(n.params[0]);
    Value *R = codegen(n.params[1]);

    if(!L || !R)
      return nullptr;
    auto Inst = BinaryOperator::CreateSDiv(L,R,"multmp");
    auto block = Builder->GetInsertBlock();
    block->getInstList().push_back(Inst);
    return Inst;
  }
```

```

case node_type::neg:
{
  return Builder->CreateNeg(codegen(n.params[0]), "subtmp");
}
case node_type::ret:
{
  Value *v = codegen(n.params[0]);
  return Builder->CreateRet(v);
}

case node_type::vardec:
{
  for (auto &var : n.params)
  {
    auto alloca = Builder->CreateAlloca(convertType(var.params[1].ident.v_type));
    NamedValues[var.params[1].ident.name] = alloca;
    if (var.params[0].type != node_type::nop)
    {
      Value *v = codegen(var.params[0]);
      auto store = Builder->CreateStore(v, alloca);
    }
  }
}

case node_type::cond:
{
  // return nullptr;
  Value *CondV = codegen(n.params[0]);
  if (!CondV)
    return nullptr;

  // Convert condition to a bool by comparing non-equal to 0
  CondV = Builder->CreateICmpNE(CondV, Builder->getInt32(0),"ifcond");


  Function *TheFunction = Builder->GetInsertBlock()->getParent();

  BasicBlock *ThenBB = BasicBlock::Create(*TheContext, "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(*TheContext, "else",TheFunction);
  BasicBlock *MergeBB = BasicBlock::Create(*TheContext, "ifcont",TheFunction);

  Builder->CreateCondBr(CondV, ThenBB, ElseBB);

  Builder->SetInsertPoint(ThenBB);
  emit(n.params[1].params);
  Builder->CreateBr(MergeBB);

  Builder->SetInsertPoint(ElseBB);
  emit(n.params[2].params);
  Builder->CreateBr(MergeBB);
```

```

    Builder->SetInsertPoint(MergeBB);
    return CondV;
  }

  case node_type::loop:
  {
    Function *TheFunction = Builder->GetInsertBlock()->getParent();

    BasicBlock *LoopBB = BasicBlock::Create(*TheContext, "loop", TheFunction);
    BasicBlock *AfterBB = BasicBlock::Create(*TheContext, "afterloop",TheFunction);

    Builder->CreateBr(LoopBB);
    Builder->SetInsertPoint(LoopBB);
    emit(n.params[1].params);

    Value* CondV = codegen(n.params[0]);
    CondV = Builder->CreateICmpNE(CondV, Builder->getInt32(0),"ifcond");
    Builder->CreateCondBr(CondV,LoopBB,AfterBB);

    Builder->SetInsertPoint(AfterBB);

    return CondV;


  }
  case node_type::fcall:
  {
    auto& f = n.params[0];
    auto& params = n.params[1].params;

    auto& fname = f.ident.name;

    std::vector<Value*> Args;

    for(auto& p : params)
    {
      Args.push_back(codegen(p));
    }
    CallInst* CallFunc = CallInst::Create(TheModule->getOrInsertFunction(fname,
   funcList[fname]), Args, fname);
    Builder->GetInsertBlock()->getInstList().push_back(CallFunc);
    return CallFunc;
  }

  default:
    break;
  }
  return nullptr;
}

```

```
void emit(const node_vec &stmts)
{
  for (auto &stmt : stmts)
  {
    codegen(stmt);
  }
}

Type *createGraph()
{
  StringRef Name = "graph";
  std::vector<Type *> v = {convertType(type_name::INT), convertType(type_name::INT)
   };
  return StructType::create(*TheContext, v, Name);
}
```

10.5 Code Listing

All other codes can be found our Github Page GrAlgo - Team5