Mattricks

Final Project Report

Yuxin Yang, Weisheng Wang, Yuanhan Tian, Jack Wang,

Danny Hou, Hang Yuan, Yuhao Dong

Introduction

Mattricks is an imperative programming language focused on mathematical calculations similar to Matlab, with a particular focus on matrix manipulation. The motivation behind our language is that matrix calculation is widely used in linear algebra and machine learning models, and there is no existing language that provides the perfect solution for such calculation. Existing languages either need the support of additional libraries or their complicated grammar makes it less friendly for new users to perform matrix calculation. Therefore, we want to design a language that provides users with the ability to perform matrix manipulation with peace of mind. In Mattricks, we will include mathematical symbols and syntax, so users do not need to install any math packages. Moreover, the matrix syntax is designed to be easily understandable for users such that users without much prior knowledge of computer science will also be able to write their own calculations using our language. It also provides more readability so that users can easily understand the programmers' intentions.

Static Scoping could improve the readabilities and help users to avoid confusion about where each variable (number) is accessed from. It also requires less run-time overhead. Strict evaluation specifies the order of executing different function arguments to avoid side-effects. A strongly and statically typed system sets everything clear to users on what type of elements they are using and allows less confusion. With such paradigms and features incorporated into itself, Mattricks gives the users a way to perform small to large scale matrix calculations with more efficiency and flexibility.

Language Tutorial

Data Types:

int: 32-bit int. Explicit declaration: `a = int 1;`.
float: 32-bits float. Explicit declaration: `a = float 0.4;`
bool: binary value of true or false. Int values can be casted to boolean.
Mat: a fixed-sized, fixed dimensioned data structure representing matrices.
The syntax is `<var> <assignment_op> <type> [<dimension>];`.
For example: `mat1 = mat int [3];` We provide users with a random access operator to manipulate the matrix, so users need to manually assign the variables. If the values are not initialized, the behavior of the language is unpredictable(undefined behavior).

Type Casting:

The primitive type variable can be statically cast to another type and type casting must be explicitly noted. For instance, we can console both boolean and integer type values.

Operators:

Arithmetic: `'+', '-'. '*', '/', '%'`. The operators follow the same precedence as they are in C.

Assignment:

Assignment operator does specify the type of variables being assigned. The type will need to be specified at the right hand side of assignment.
For example: `D = int 1; E = float 2.0; F = bool true;`

Declaration:

Once declared, the assignment operator does not specify the type of variables being assigned.
For example: `int A; float B; bool C; A = 1; B = 2.0; C = true;`

Equivalence:

`==, !=, >, <, >=` and `<=` are right associative and the return of equivalence expressions is int 0 or int 1.

Logical:

The AND(`&&`) and OR(`||`) operation is supported in this language. AND has higher precedence than OR. Negation and '!' operator(unary) is not supported in this language. Negation is to turn some value of float and int to its negative value. ! is used to denote logical NOT.

Stream:

The stream operator is an operator used to print the data to the console. It is paired with the "`console`" and "`consolef`" keyword. The "console" is used to print integer and boolean value, whereas the latter is used to print the floating point value. The right hand side of the stream operator is the data which will be sent to the console.

Function For Matrix:

Here present an example of how to define a function that takes a 1d integer matrix with size 10(m) and a value(v). Adds each value of the matrix by v and print it out. Finally, the function returns the 1d integer matrix with size 10. The return type is specified after the `gives` keyword.

```
/* Sample Function declaration */
function addval_print_1d_int_10(mat int [10] m, int v) gives mat int [10] {
    i = 0;
    while (i < 10) {
        m[i] = m[i] + v;
        console << m[i]
        i = i + 1;
    }
    return m;
}
```

if / if-else / while

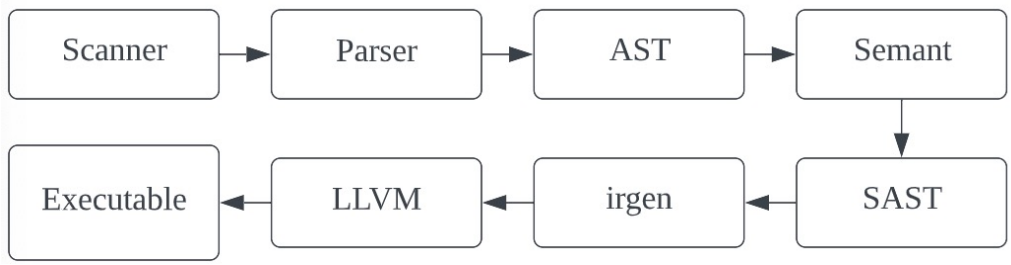
While-loop is supported by our language. Use the keyword “while” followed by condition statements. If and if-else statements are also supported.

```
/* While */
while (expression) {
   (block)
}

/* If */
if (expr) {
   (block)
}

/* IfElse */
if (expr) {
   (block)
}
else {
   (block)
}
```

Architectural Design



Scanner

Developed by: Yuanhan Tian, Weisheng Wang, Hang Yuan, Yuhao Dong

The **scanner.mll** file is used to define all the tokens that serve the purpose of lexical analysis. We built it on top of MicroC and added, modified, and removed some tokens correspondingly for our matrix-oriented language. For instance, we designed *mat, console, consolef, gives,* etc. It also raises an exception if it encounters an invalid token.

Parser

Developed by: Yuanhan Tian, Weisheng Wang, Hang Yuan, Yuhao Dong

The **parse.mly** file uses the tokenize rule defined in the scanner.mll to generate the tokens. It then takes the generated tokens to perform syntax analysis and generates an abstract syntax tree(AST) based on the grammar rules we created and it throws a parsing error if the input is syntactically incorrect.

Semantic Checking

Developed by: Yuanhan Tian, Weisheng Wang, Hang Yuan, Yuhao Dong

The **semant.ml** file checks if there is any semantic error by checking the AST generated by the parser and generates an SAST. It uses the parse function defined in the parser module to generate the parsed program. Semantic checking takes the parsed program as the input and checks if the input follows our design of grammar. For example, an integer cannot be declared if the assignment contains a floating point. If primarily ensures the data types for each declaration are applied correctly. We also use the StringMap module to create a symbol table for all the variables that users create and because of the immutability of StringMap, everytime a new StringMap is created when a new variable is created.

It guarantees that none of the semantics rules are violated, for example

1. No duplicates
2. All variables must be declared and in their scope.
3. Assignment can only be performed among variables and expressions of the same type.

If one or more rules are violated, the semantic checker will throw an exception to prohibit further compilation.

Code Generation:

Developed by: Yuanhan Tian, Weisheng Wang, Hang Yuan, Yuxin Yang, Yuhao Dong

The **irgen.ml** file uses LLVM to generate IR code based on the SAST. In the last phase of running our program, the IR code is converted to assembly code and an executable is then created. We also use the StringMap module in ir generation to store new variables in a way similar to the process mentioned in semantic checking.
The IR generator calls the semant functions defined in the previous section and takes the semantically checked program as the input to let llvm's IRBuilder to transform the semantically-checked code into lower-level, intermediate instruction of target language, which could be understood by the LLVM compiler.

Back-end(LLVM)

After this phase is done, the generated IR will be compiled by LLVM compiler and produce the final executable binary code. The output of the target program will be printed as strings to the console.

Test Plan

Developed by: *Hang Yuan, Yuxin Yang*

Based on the designs of the LRM, multiple positive and negative test cases of features for each phase. Each test case contains the source code and expected output for different phases. We integrated the test suite with the testing automation: a testing job will be triggered to run all the test cases after the pull request is enabled. Developers are also required to manually trigger the test job frequently after they implement the new features.

Test-case Example 1

Source Code	IR Generated Code	Output

/*
./test_cases/runnable/pass_stream.mc */
 	
function main() gives int {
  i = int 1;
  console << i;
  f = float 111.2;
  consolef << f;
  b = bool true;
  console << b;
  m = mat int [4];
  m[0] = i;
  console << m[0];
  console << 1;
  return i;
}
	......
	define i32 @main() {
	entry:
	  ......
	  %printf5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i1 %b4)
	  %tmp = alloca [4 x i32], i32 0, align 4
	  %i6 = load i32, i32* %i, align 4
	  %tmp7 = getelementptr [4 x i32], [4 x i32]* %tmp, i32 0, i32 0
	  store i32 %i6, i32* %tmp7, align 4
	  %tmp8 = getelementptr [4 x i32], [4 x i32]* %tmp, i32 0, i32 0
	  %tmp9 = load i32, i32* %tmp8, align 4
	  %printf10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %tmp9)
	  %printf11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 1)
	  %i12 = load i32, i32* %i, align 4
	  ret i32 %i12
	}
		1
		111.200000
		1
		1
		1

Test-case Example 2

Source Code	IR Generated Code	Output

/*
./test_cases/runnable/pass_calc_mat.mc */
 	
/* Addition */
function main() gives int {
  mat1 = mat int [2];
  mat1[0] = 123;
  console << mat1[0];
 	
  mat2 = mat int [2];
  mat2[0] = 321;
  console << mat2[0];
 	
  mat3 = mat int [2];
  mat3[0] = mat1[0] + mat2[0];
  console << mat3[0];
 	
  return 0;
}
	......
	define i32 @main() {
	entry:
	  ......
	  %tmp9 = alloca [2 x i32], i32 0, align 4
	  %tmp10 = getelementptr [2 x i32], [2 x i32]* %tmp, i32 0, i32 0
	  %tmp11 = load i32, i32* %tmp10, align 4
	  %tmp12 = getelementptr [2 x i32], [2 x i32]* %tmp4, i32 0, i32 0
	  %tmp13 = load i32, i32* %tmp12, align 4
	  %tmp14 = add i32 %tmp11, %tmp13
	  %tmp15 = getelementptr [2 x i32], [2 x i32]* %tmp9, i32 0, i32 0
	  store i32 %tmp14, i32* %tmp15, align 4
	  %tmp16 = getelementptr [2 x i32], [2 x i32]* %tmp9, i32 0, i32 0
	  %tmp17 = load i32, i32* %tmp16, align 4
	  %printf18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %tmp17)
	  ret i32 0
	}
		123
		321
		444

Summary

Weisheng Wang (ww2609):

Contribution: I participated in group meetings to plan short term goals and distribute workloads. In “Hello World”, I did pair programming with Yuhao Dong on ast, parser and scanner parts; I also designed the Mat type and realized its rules. In the final phase, I did pair programming with Yuanhan Tian on redesigning the Matrix type(reimplemented the entire workflow from scanner to codegen), and I helped with miscellaneous features.
Takeaway: With hands-on experience in developing my own language, I gained a deeper understanding of the lecture materials.
Advice: Plan ahead and Set weekly and monthly milestones.

Yuanhan Tian (yt2825):

Contribution:I reused some of the microc's example code and implemented part of the ast , the semantics checker and sast and the IR generator. I discussed the plans with the group members in the Zoom meetings.I collaborated with Yuhao Dong, Hang Yuan and Yuxin Yang and implemented part of the IR generation and enabled the program to bind assignments(initialize the variables during declarations). I did pair programming with Weisheng Wang on supporting matrices.
Takeaway: The project gives me a chance to thoroughly understand how a compiler is implemented and works. This is a very valuable experience for me. Good teammates are often more helpful than writing alone.
Advice: Think recursively.

Yuhao Dong (yd2626):

Contribution: I participated in group meetings to discuss the team plans, individual roles, and difficulties we faced. I worked with Weisheng Wang together on adding and modifying rules in ast, parser, and scanner files according to our own language features. I also worked with Weisheng Wang to fix the 2d matrix parsing rules to make it work as designed and to eliminate the conflicts. For the final part, I worked together with Yuanhan Tian and Yuxin Yang on implementing the IR generation.
Takeaway: Efficiently planning and dividing the work saved us a lot of time. Communicating with teammates would make the project more organized. Using what we learned in class to design and build our own programming language is also a valuable experience.
Advice: Start the project early and always communicate with teammates.

Yuxin Yang (yy3277):

Contribution: 1. Created test cases for testing the parser of the program matching with the features mentioned in LRM; 2. Organized and managed this repository and group meetings 3. Helped identify some issues and existing bugs in the grammar; 4. Updated LRM 5. Helped Yuanhan and Yuhao to work on ir generation 6. Created presentation slides and wrote final project report
Takeaway: I learned that clear division of labor is important in teamwork especially in a group with many people. I also learned to make plans early will help save time and keep things organized. I also gained a more in depth understanding of compiler design and implementation.
Advice: Set up your agenda ahead of time and go to office hour as needed will help you become closer to success in this class

Hang Yuan (hy2784):

Contribution: 1. Be responsible for the entire testing plan and automation for all phases; 2. Write and revise all test cases for all phases (scanner, parser, semant, irgen); 3. Report potential bugs based on the test suites; 4. Work with Yuanhan Tian and Weisheng Wang to add new features and fix multiple bugs/issues through all the phases; 6. Participated in all discussions, meetings, documentations and presentations.
Takeaway: Learned a lot of knowledge of how the compiler works behind the scene.
Advice: Start earlier.

Danny Hou (dh3034):

Contribution: I discussed our plan with team members during group meetings and collaborated with Yuanhan to work on the tokens and rules of our scanner reusing some of the microc's example. I also worked on final reports and presentation slides with team members.
Takeaway: A deeper understanding of the programming language.
Advice: The sample code provided by instructors is very helpful. Ask questions on piazza if you get stuck on anything.

Jack Wang (yw4014):

Contribution: I participated in discussion on implementation, ast, and semant. The discussions finalized key items for submission. I went to office hours to ask the requirements and expectations and shared them with the team. I contributed to the presentation in architecture design.
Takeaway: This program helps me better understand the working mechanism of a modern compiler. Very interesting project!
Advice: Do things earlier.