Columbia University

Computer Science Department

COMS W4115 Programming Languages and Translators

Matcab: Matrix Manipulation Language

Author:
Cheng Xiang
Yu Qiao
Ran Yu
Tianchen Yu

Supervisor:
Prof. Stephen A. Edwards

December 19, 2012

Contents

1 Introduction

The MatCab is a programming language that simplifies and accelerates matrix manipulations. It is a C-style language that particularly aims at easier and faster matrix computing for programmers. Matrix computing is one of the most common linear algebra operations used in scientific computation. Operations like image processing and data mining both require considerable amount of highly efficient matrix calculations. For example, in the multi-dimensional regression, a basic algorithm in Machine Learning, we need to compute the inverse of the matrix. Instead of writing multiple lines to implement the inverse function and maybe error-prone due to the unfamiliar with inverse concept. Our MatCab language, however, provides an easy way to apply basic arithmetical operation to matrices. Like that you can simply type 'A"' to get the inverse matrix of A. And its C-like syntax can help C programmers get rid of the burden of hundreds of lines of c-code for matrix computing, making the computation simple, compact and easy to read.
MatCab simplifies matrix processing, through domain-specific data types and operators. A basic data type in MatCab is a $FloatMat$, which is a 2-dimension matrix consist of float numbers. And $FloatMat$ support various basic Matrix computation, such as $inverse$, ,$transpose$, ,$convolution$, ... and etc.
With the feature of matrix manipulation, MatCab can be widely use in programming about calculation. The intuitive, robust and portable MatCab can do a lot in programming.

2 Language Tutorial

2.1 General Purpose Programs

At the beginning, I would like to demonstrate our MatCab language with a simple general purpose program that calculates the greatest common divisor.

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
```

```
            b = b - a;
        endif;
    }
    return a;
}

int entry() {
    int a;
    a = gcd(2, 14);
    print(a);
}

}
```

In this gcd.mcb program, language features like function declaration, function call(gcd(2, 14)), while loop, if-else branching, variable declaration are presented. More precise syntax specification is given in later part of this report.

2.2 MAE

In order to demonstrate the simplicity in syntax of matrix manipulations, the code for mae.mcb program is given as below. MAE stands for Mean-Average-Error. This is an indicator of the quality of decomposing one huge matrix into approximately the product of two matrices with much smaller dimension.

```
int entry() {
    int i;
    int j;
    int cnt;
    floatMat A;
    floatMat B;
    floatMat C;

    Init B[20][4];
    Init C[4][10];
    Init A[20][10];
```

```
B = [[
    1, 2, 3, 4  ?
    7, 3, 9, 1  ?
    5, 6, 7, 3  ?
    3, 9, 1, 4  ?
    8, 3, 5, 3  ?
    6, 3, 2, 7  ?
    2, 8, 5, 3  ?
    4, 2, 9, 4  ?
    6, 3, 8, 4  ?
    7, 9, 5, 2  ?
    3, 8, 6, 9  ?
    3, 0, 3, 1  ?
    4, 2, 2, 7  ?
    1, 6, 8, 7  ?
    3, 0, 4, 3  ?
    6, 5, 7, 2  ?
    2, 7, 7, 3  ?
```

In this program you may see language features including matrix multiplication operation (simply achieved by overriding * operator) , matrix subtraction operation (unlike C-style syntax that iterates through the 2-dimensional array, the difference of corresponding elements of two matrix of same type and dimensions can be directly returned using - operator).

2.3 Gaussian Filter

2.4 Compiling and Running Your Program

First make the MatCab compiler by running

`make`

. Then use

`./matcab -e < gcd.mcb`

to compile and run your program. If you just want to compile and do not want to run the program, use "-e" parameter. If you want to view the AST of your program, use the "-a" parameter.

3 Language Reference Manual

3.1 Program Definition

A MatCab Program consists of any number of global variable declarations and several function definitions. The compiler will find the function with function name "entry" and start executing the program from this entry function. ThereâĂŹs no requirement for the order of such variable definitions or function definitions. Users can arrange the order of appearance of them in any way they would like to. Our compiler is able to fetch the right function that a user is pointing to no matter this function is defined before or after the definition of entry function.

There are two now existing limitation about MatCab programs. The first is that now the compiler cannot process arguments given from command line, which means that the behavior of one MatCab program is pretty determinant and predictable, lacking some extend of flexibility. The second flaw of our current implementation of MatCab language is that now our compiler will accept only one input source program. If a user would like to take advantage of some existing implemented function, he/she would have to copy the code of such functions to the new source MatCab program.

Each variable declaration consists of a type specification and a variable name designation. Each function definition is the combination of return type specification, function name designation, formal list declaration and function body specification. The rules for naming a variable or a function are given in later discussion of this MatCab language specification document. Formal list of a function may contain none, one or more than one arguments (with the only exception of entry function that can take in no arguments). MatCab programmers are required to explicitly specify the type of the formal arguments. The function body is simply a list of MatCab language statements. Local variables are declared at the head of the function body. Though the order of variable declarations and function definitions within the source program is of no limit, all the declarations of variables

have to appear before any other statements within that function body scope.

Please refer to the gcd.mcb program in section 2.1. This program calculates the greatest common divisor of two integers. A function named gcd, with return type int, is declared before the entry function. And it is later called by the entry function to give out the greatest common divisor of 2 and 14.

3.2 Lexical Conventions

There are six kinds of tokens: comments, identifiers, keywords, constants, strings, and operators. Comments, blanks, tabs, tabs, new lines and are ignored. The token is recognized until the parser sees a separator.

3.2.1 Comments

Program segments enclosed in the "/*" and "*/" are considered as the comments of MatCab program. Our compiler will simply ignore and discard such code segments. No nested "/*"-"*/" pairs are allowed, otherwise a syntax error will occur.

*/*This is a line of comment.*/*

3.2.2 Identifiers

An identifier is used to distinguish one variable or function from another. It consists of a sequence of letters and numbers, which are demanded to start with a letter. The identifiers are case-sensitive, i.e. Seen by the compiler, "move" and "Move" are different identifiers.

foo,bar,BAR

3.2.3 Keywords

Like almost every other programming language on earth, MatCab has a series of reserved keywords. MatCab programmers cannot use them to name their variables nor functions. They are serving for different purposes. Names of built-in types, built-in functions are preserved. Some keywords are for flow-control purposes.

$$Keyword \rightarrow \mid float \mid int \mid floatMat \mid intMat \mid intRowVec \mid floatRowVec \mid intColVec \mid floatColVec \mid sum \mid init \mid if \mid else \mid endif \mid for \mid while \mid entry \mid true \mid return \mid true \mid false$$

3.2.4 Constants

MatCab language has two primitive types: integers and floats. Thus the constant literals in MatCab are either integer literals or float literals. An integer literal is a series of number digits. A float constant contains an integer part, a decimal point, and a fraction part.

A numeric constant can be an integer constant, a float constant. The integer is a series of numbers. The float constant contains an integer part, a decimal point, a fraction part and a character 'f'.

$$Numeric\ Constant \rightarrow Integer\ Constant \mid Float\ Constant$$
$$Integer\ Constant \rightarrow [0\text{-}9]+$$
$$Float\ Constant \rightarrow [0\text{-}9]+\ .\ [0\text{-}9]+\ 'f'$$

3.3 Operators

Being a language that aiming at simplifies the syntax of carrying out arithmetic operations, especially matrix calculations, the operators defined in the MatCab language is of great importance.

The precedence and associativity are in correspondence with the equivalent operators in other common general purpose programming language, C for instance.

3.4 Types

MatCab now supports integer and float variables as primitive type variables. As of the matrix part, all elements of a matrix must be float numbers.

3.5 Expressions

Here the definition of MatCab expressions follows the steps in C Language Reference Manual defining expressions.

Table 1: Operators

Symbol	Explanation
+, -	Binary operator that performs addition/subtraction operation. Operands could be int-int, int-float, float-int, float-float, matrix-int, matrix-float, int-matrix, float-matrix, and matrix-matrix pairs
-	Unary operator that gives the value with opposite sign of given constant literal or variable. Operand is a integer or a float number.
*, /	Binary operator that performs multiplication/division operation. Operands could be int-int, int-float, float-int, float-float, matrix-int, and matrix-float pairs
%	Binary operator that performs modular operation. Operands could be int-int pairs.
>, >=, <, <= ,==, !=	Binary operator that performs greater-than/no-less-than/less-than/no-greater-than/equals/not-equals comparisons. Operands could be int-int or float-float pairs
**	Binary operator that performs convolution operation. Operands are matrix-matrix pairs.
âĂŸ	Unary operator that performs transposition operation. Operand is a matrix.
âĂIJ	Unary operator that performs matrix-inverse calculation. Operand is a matrix.
&&, ||	Binary operator that performs AND/OR Boolean operation. Operands are Boolean expressions.
!	Unary operator that performs NOT Boolean operation. Operand is Boolean expression.

3.5.1 Assignment expressions

Assignment expressions are of form:

Identifier AssignOperator expr

3.5.2 Built-in function call expressions

Calling a built-in function with argument of a matrix is identified as an expression by our parser. For example, | A | is asking for the determinant of the matrix A. Such function calls are:

| expr |
tr (expr)
submat (expr, expr, expr, expr, expr)
sum (expr, expr, expr, expr, expr)

3.5.3 User defined function calls

User defined functions are called in the same manner as in built-in function calls.User defined functions are called in the same manner as in built-in function calls.

Identifier (argument_list_optional)

3.5.4 Declarations

There are two kinds of declaration statements in MatCab: variable declarations and function declarations. Variable declarations are of form:

Type Identifier ;

Function declarations are of form:

Type Identifier (formal_list) function_body

A function may take in no arguments, which means that the formal list is an optional part for a function declaration. The function body consists of local variable declarations and statements. Any local variables being used within this function body must be declared first before any of the statements. The statements are parsed and executed line by line.

3.6 Statements

Statements in MatCab language can be classified as four types: simple statements, grouped statements, matrix initialization statements and flow control statements.

A single semicolon or an expression followed by a semicolon are parsed as simple statements.

A series of statements enclosed by a pair of curly brackets are considered as one single statement. It is like:

statement-1; statement-2;... statement-n;

Before applying operations to a declared matrix, it must be initialized with its dimensions first. The matrix initialization statements are of form:

Init Identifier [expr] [expr]

In which the first expression specifies the number of rows of this matrix, and the second expression specifies the number of columns of this matrix.

3.7 Flow Control Statements

Basic flow control statements including if, ifelse, for, while, return statements are supported in MatCab. The formal definition is given as below:

If (Boolean_expression) statement endif ;
If (Boolean_expression) statement else statement endif ;
For (expression; Boolean_expression; expression) statement
While (Boolean_expression) statement
Return expression ;

3.8 Program Definition

A MatCab program consists of several definition statements and exactly one entry point. The entry point itself is a function declaration with a specified form:

```
program
        /* Global variables; Function declarations; Entry point. */
        : { [], [] }
```

```
        | program VarDecl  { ( $2 :: fst $1 ), snd $1 }
        | program FuncDecl { fst $1, ( $2 :: snd $1 ) }
;

VarDecl_list
        : /* nothing */ { [] }
        | VarDecl_list VarDecl { $2 :: $1 }
;

VarDecl
        : Type IDENTIFIER SEMICOLON { { var_type = $1; var_name = $2 } }
;

FuncDecl
        : Type IDENTIFIER LPAREN formal_list_opt RPAREN LBRACE
        VarDecl_list Statement_list RBRACE

formal_list_opt
        : { [] }
        | formal_list { $1 }

formal_list
        : Type IDENTIFIER formal_list_rest { { var_type = $1; var_name = $2; } :: $
;

formal_list_rest
        : /* nothing */ { [] }
        | COMMA formal_list { $2 }
;

Type
        : INT { Int }
        | FLOAT { Float }
        | INTROWVEC { IntRowVec }
```

```
        | FLOATROWVEC { FloatRowVec }
        | INTCOLVEC { IntColVec }
        | FLOATCOLVEC { FloatColVec }
        | INTMAT        { IntMatrix }
        | FLOATMAT      { FloatMatrix }
;

Statement_list
        : /* nothing */ { [] }
        | Statement_list Statement { $2 :: $1 }
;

Statement
        : expr SEMICOLON { Expr($1) }
        | SEMICOLON { Empty }
        | LBRACE Statement_list RBRACE { Block($2) }
        | IF LPAREN Bool_expr RPAREN Statement ELSE Statement ENDIF SEMICOLON
        | IF LPAREN Bool_expr RPAREN Statement ENDIF SEMICOLON
        | FOR LPAREN expr SEMICOLON Bool_expr SEMICOLON expr RPAREN Statement
        | WHILE LPAREN Bool_expr RPAREN Statement
        | EXPORT LPAREN IDENTIFIER RPAREN SEMICOLON
        | IDENTIFIER ASSIGN IMPORT LPAREN IDENTIFIER RPAREN SEMICOLON
        | RETURN expr SEMICOLON
        | INIT IDENTIFIER LBRACKET expr RBRACKET LBRACKET expr RBRACKET
;

data
        : FLOAT_LITERAL { Float_lit($1) }
        | INT_LITERAL { Int_lit($1) }
        | TRUE { Boolean($1) }
        | FALSE { Boolean($1) }
;

Int_Row
```

```
        : { [] }
        | expr { [$1] }
               | Int_Row COMMA expr { $3 :: $1 }

IntMat_Init
        : Int_Row { [List.rev $1] }
        | IntMat_Init QUES Int_Row { $1 @ [List.rev $3] }

IdList_opt
        : { [] }
        | IdList { $1 }

IdList
        : expr { [$1] }
        | IdList COMMA expr { $1 @ [$3] }

expr
        : data { $1 }
        | IDENTIFIER ASSIGN expr { VarAssign($1, $3) }
        | BAR expr BAR { Det($2) }
        | SLASH IntMat_Init BACKSLASH { Im_init($2) }
        | TRACE LPAREN expr RPAREN { Trace($3) }
        | SUBMAT LPAREN expr COMMA expr COMMA expr COMMA expr COMMA expr RPAREN
        | SUM LPAREN expr COMMA expr COMMA expr COMMA expr COMMA expr RPAREN
        | IDENTIFIER LBRACKET expr RBRACKET LBRACKET expr RBRACKET
        | IDENTIFIER { Id($1) }
        | LPAREN expr RPAREN { $2 }
        | IDENTIFIER LPAREN IdList_opt RPAREN { Call($1, $3) }
        | expr TIMES expr { Binary_op($1, Times, $3) }
        | expr DIVIDES expr { Binary_op($1, Divides, $3) }
        | expr MODE expr { Binary_op($1, Mode, $3) }
        | expr MTIMES expr { Binary_op($1, Mtimes, $3) }
        | expr CONVOLUTION expr { Binary_op($1, Convolution, $3) }
        | expr PLUS expr { Binary_op($1, Add, $3) }
```

```
        | expr MINUS expr { Binary_op($1, Sub, $3) }
        | MINUS expr %prec UMINUS { Unary_op($2, Neg) }
        | expr RhsUnaryOp { Unary_op($1, $2) }
        | IDENTIFIER LBRACKET expr RBRACKET LBRACKET expr RBRACKET ASSIGN expr
;
Bool_expr
        : expr EQ expr { Bool_expr1($1, Eq, $3) }
        | expr NEQ expr { Bool_expr1($1, Neq, $3) }
        | expr LT expr { Bool_expr1($1, Lt, $3) }
        | expr GT expr { Bool_expr1($1, Gt, $3) }
        | expr LEQ expr { Bool_expr1($1, Leq, $3) }
        | expr GEQ expr { Bool_expr1($1, Geq, $3) }
        | LPAREN Bool_expr RPAREN { $2 }
        | Bool_expr AND Bool_expr { Bool_expr2($1, And, $3) }
        | Bool_expr OR Bool_expr { Bool_expr2($1, Or, $3) }
        | NOT Bool_expr { Bool_expr2($2, Not, $2) }
;

RhsUnaryOp
        : INVERSE { Inverse }
        | MINVERSE { Minverse }
        | TRANSPO { Transpo }
;
```

4 Project Plan

4.1 Overview

This is an team project that involves 4 people, so we have make clear each team member's responsibility at the very beginning. At first we tried to use CUDA to speedup matrix calculation and did some research on it, but it did not come into use at last. All four team members participated in designing, coding, and testing,

Table 2: Responsibility of Each Member

Cheng Xiang	Team management, Syntax design, Parsing and AST, Test suites
Tianchen Yu	Semantic analysis, Code generation
Ran Yu	Scanner and Matrix library
Yu Qiao	Architecture design, Glue everything together, Automatic Testing

Table 3: Project Plan

Cutting Time	Deliverable
Sept 26	Language proposal finished, background language learnt
Oct 2	Team member responsibility defined, scanner started
Oct 18	Scanner completed, language convention started
Oct 31	Language reference manual finished, parser started
Nov 15	Parser finished, proceed to AST
Dec 2	AST done, proceed to code generation
Dec 16	Code generation finished, proceed to test
Dec 19	Turnover

while each one has more specific tasks per the table below.

At the beginning of this semester, we came up with a project proposal and then every one has his/her share of responsibility and worked towards it.

4.2 Administration

We set several milestone for important deliverables for time management.

4.3 Develop environment

Programming Language We use the latest OCaml 4.00.0 as developing language.

Scannner We use OCamllex as scanner.

Parser We use OCamlyacc as parser.

Automatic Tests We use a Perl script and a bash script to test do regression test periodically.

Automatic Build We use GNU make.

Compile As our compiler outputs Java code, we use Java 1.6.0_35 as runtime support.

Version Control We use SVN to do the version control. The repository is in google code and is publicly available. The URL is `http://code.google.com/p/matcab/`

5 Architecture Design

5.1 Overview

The MatCab compiler is nothing different than a common compiler. It has four parts: scanner, parser, AST and code generator. The only difference between MatCab compiler and the commercial compiler is that it does not generate assemble code. Instead, it generates Java source code and makes use of Java as runtime support.

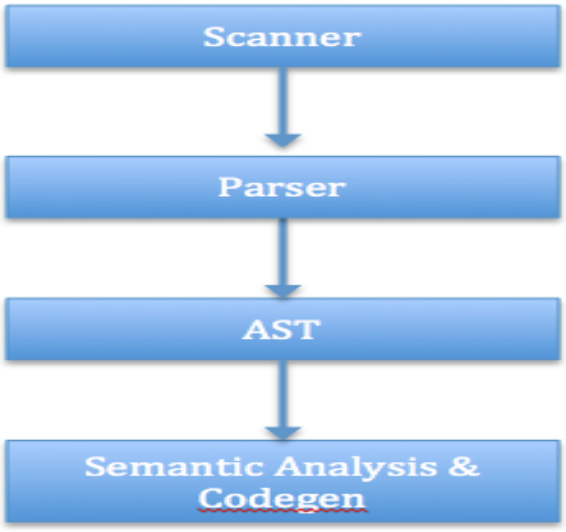

Figure 1: MatCab Architecture

5.2 Scanner

The scanner is used to tokenize the input file into token streams. The scanner will eliminate all kinds of useless tokens, such as tabs, white spaces, tabs, etc, and will raise an error if the scanner meets an illegal token. We used ocamllex to build the scanner.

5.3 Parser

The parser accepts the token stream generated by the scanner and parses it to build an AST. If the input token stream does not meet the requirements of the predefined syntax, it will raise an error. We used ocamlyacc to build the parser.

5.4 AST

The abstract syntax tree eliminates some unnecessary parts of elements at the parsing phase to the rest compiling job easier. Figure 4 is the type checking list.

5.5 Compiler

The compiler accepts an AST and recursively traverse each node of it, translating it into Java code while doing the type checking at the same time. If there is something wrong, it will raise an error.

6 Test Plan

Our project can be roughly divided into two phases: 1. The design and implementation of the parser and Ast
2. The construction of the compiler (also translater).

Tests were involved in both phases and every single feature was tested with a corresponding test file.

The construction of the compiler did not start until we had run sufficient tests on our parser and ast and fixed all detected bugs. Started from here, new test file would be added into our test suite when our compiler got a new functionality.

Table 4: Type checking list

Accessing an undeclared variable
Invoking an undefined function
The arguments of a function call does not match the formal list
Calculating the determinant of a non-matrix variable
Calculating the trace of a non-matrix variable
Returning the submatrix of a non-matrix variable
Returning the area sum of a non-matrix variable
Returning the convolution of a non-matrix variable
Returning the inverse of a non-matrix variable
Returning the transpose of a non-matrix variable
Accessing an non-integer index of a matrix
Assigning a non-matrix variable to a matrix
Adding/Substrcting an integer to(from) a matrix
No entry function
The entry function has arguments

Automation was used during our tests. A bash script was written to processing all test files inside the test suite. The direct result and intermediate java source code was recoded for possible future inspection. Our Test Suites was divided into several sub suites, each focused on one domain of our program. Here is a list of all the sub suites, with self-explanatory names: basic comprehensive, expr, func, stmt, var

6.1 basic

This suite tested the most basic functionalities and top level structure of our language. Sample program: Basic5.mcb This program aimed to test: variable and function declaration, both local and global, the return statement, the type checking of return value, and the recognition of the entry() function.

```
int entry() {
    int a;
    int b;
}
```

```
5
float f1;
float foo(int bar) {
    return 0;
}
10
int baz;
```

6.2 expr

This suite tested the parsing and implementation of normal and nested expressions. The tests of arithmetic calculations and matrix operations were also include in this suite. Sample: expr10.mcb This program mainly tested matrix initialization, which was achieved through nested expressions. Same for the print function.

```
int entry() {
    int a;
    floatMat b;
    floatMat d;
5
    Init b[3][3];
    b = [[ 1,2,3?4,5,6?9,8,7]];
    Init d[3][3];
    d = [[ 10, 0, 30 ? 44, 2, 92 ? 3, 49, 72 ]];
10  b = b * 11;
    print(b[2][2]);
    d = d ** b;
    print(d[1][1]);
}
```

6.3 func

This suite basically tested the function call and declaration. Sample: fun4.mcb This program tested the call and declaration of the foo function, which returns an element of a matrix. Note that foo is called inside the call to print as parameter.

```
float foo(floatMat A, int x, int y) {
    return A[x][y];
}

int entry() {
    floatMat B;
    int xx;
    int yy;
    xx = 3;
    yy = 4;
    Init B[10][10];
    B[1][1] = 9;
    print(foo(B, xx, yy) + 33);
}
```

6.4 stmt

This suite tested the parsing and implementation of statements Sample: stmt8.mcb This program tested the "for" loop, which is a statement.

```
int entry() {
    int a;
    for (a = 0; a < 10; a = a + 1)
        print(333);
}
```

6.5 var

This suite tested the initialization and declaration of variables Sample: var9.mcb This program tested the initialization of a matrix. Note that we assigned Mat A with an illegal matrix expression, which has inconsistent number of elements on each row. So we were expecting an error to be thrown here.

```
int entry() {
    floatMat A;
    Init A[3][5];
```

```
    A = [[ 1, 2 ? 3, 4, 5 ]];
}
```

6.6 comprehensive

This suite tested many features our language as a whole. Test programs in this suite were relatively complicated and did some actual stuff. Sample: fib.mcb This program found and printed out the fifth Fibonacci number. Recursive call, if-else flow, along with some basic features were tested.

```
int fib(int n) {
    int tmp;
    if (n == 1)
        return 1;
    endif;

    if (n == 2)
        return 1;
    endif;

    if (n >= 3)
    {
        tmp = fib(n-1) + fib(n-2);
        return tmp;
    }
    endif;
    return 0;
}

int entry() {
    print(fib(5));
}
```

7 Lesson Learned

This section gives the lessons learned by each member.

7.1 Cheng Xiang

Originally I was quite optimistic about our project. Because it seems that nobody, as far as I know at least, had ever developed language that exploited the simplicity and beauty of Matlab-like matrix manipulation syntax, and exploited the concurrable nature of many matrix calculations to accelerate the calculation process at the same time. However our development didnâĂŹt go quite smoothly. At last we submitted a runnable compiler using syntax close to Matlab, but did not produce CUDA code. Being the âĂIJleaderâĂİ of this project team, I thought there are several reasons for this.

First of all, the team leader has to be able to do the time management wisely. It would be best if all the team members could agree on a fixed time in each week to sit down and work something serious out. Secondly, appropriate milestones and timeline should be set up by the team leader. We got stuck at developing parser and constructing the AST for quite long a time. This is the primary reason we didnâĂŹt choose to output CUDA executable code and selected relatively simpler Java code instead. And last, the team member should always be good at communicate with team members. He/She should play as a role model and encourage other team members to steadily move on along the path of development. Advice for future teams

Implementing a translator or compiler for a given language is comparably simple. The difficult and fun part lies in developing your own language. With so many long-existing languages it may seem quite hard to come up with a brilliant idea. Even you think you get an extraordinary idea, keep in mind that idea is worthless unless it is carried out. Try to group up with people with different specialties and academic backgrounds, and consult friends (especially those PhD ones working in areas besides computer science) during the design of your own language.

7.2 Tianchen Yu

For me, the most valuable lesson learned during our project is that coordination among team members can greatly improve the team efficiency. We have a very clear role for each member of our team. After the parser and ast were done, everybody did their tasks concurrently. And we knew who to ask if we encountered a problem. This made the second phase of our project went smoothly and fast. We did a lot of thing within only a few days.

Also, it is much better to find a rather long time period when everybody is available, ideally a whole day. In this way the whole team would be more devoted into the tasks. Having some one leaving in the middle of a meeting sometimes diminish the passion of other team members.

Advice for Future Teams:Spread the task evenly throughout the semester. DonâĂŹt push everything to the very last week. The parser and ast should be done around mid term so you will have plenty of time to implement the rest and do tests without having to cut off some features could be pretty awesome.

7.3 Ran Yu

Although I have implemented a compiler in my undergraduate study, MatCab is a really different one. Thinking in the O'Caml way is such a different experience for me.

Except Functional language, I learned in detail how the AST, scanner, parser and compiler are worked together to translate a programming language into executable. It is much more hard than I expected, but I am very glad that our team work together and make it.

Also I learned how to design unit test that discover all kind of dark corner and fix the problem with teammates.

Last but not least, the project teach me a lot in team working. How to deal with each other and how to make the project developed more efficient. In our project, everyone of us gave the best effort to implement the compiler.

7.4 Yu Qiao

The first lesson I learnt is start early, although this has been repeatedly stated by so many times by different teams. OCaml is a language that is very difficult to command, and it requires far more time than expected to write just a few lines of code. Starting early means getting a deep understanding of OCaml and thus make the following work smoothly.

The second lesson is to define LRM accurately first. We did not come up with a crystal clear LRM at first, and that proves to be a mistake. We had to revise the parser and AST again and again when doing the real coding, which is a great waste of time.

The third lesson is to resolve the technical risk at the very begging. We failed to generate CUDA code because we could not finish it in a very short time. Learning something new is always time-consuming, and the best way is to let someone resole this risk at first.

References

[Dennis M. Ritchie, 1973] Dennis M. Ritchie (1973). C Reference Manual. *Bell Telephone Laboratories*

[Jeremy, Robert, Kevin and Yongxu, 2011] Jeremy Andrus and Robert Martin and Kevin Sun and Yongxu Zhang (2011). CLAM: The Concise Linear Algebra Manipulation Language. *Columbia University*