Macaw Reference Manual

William Hom - wh2307
Joseph Baker - jib2126
Yi Jian - yj2376
Christopher Chang - cyc2136

Aug 11, 2016

Summary

Chapter 1

Introduction

Many commercial languages support basic mathematical types like integers, floats, and vectors (as arrays typically). However, popular languages often require additional libraries to allow users to perform matrix operations, even if the language supports multi-dimensional arrays. Often only specialized mathematical languages like Matlab or R have full support for matrix calculations. Macaw is a Matlab and R inspired language that provides a matrix data type and by allowing a simple operator overloading mechanism is able to provide a standard library of matrix manipulation functions.

Macaw is a strongly typed, imperative language that supports simple flow control, looping, and user defined function constructs. Users can define function-like constructs for operators that allow an operator to function on types that the language doesn't natively support.

Credit for the development of Macaw goes to William Hom, Joseph Baker, Yi Jian, and Christopher Chang, who designed the language and implemented its compiler. Special thanks are also given to their professor Stephen Edwards, PhD., and their advisor, Graham Gobieski.

Inquiries into the language can be made to the following addresses:

- William Hom - wh2307@columbia.edu
- Joseph Baker - jib2126@columbia.edu
- Christopher Chang - cyc2136@columbia.edu
- Yi Jian - yj2376@columbia.edu

Chapter 2

Tutorial

Information on how to use the Macaw language compiler can be found in readme.md, but in brief:

1. Make sure LLVM is installed.
2. Run `make` from the command line.
3. Run `./mcaw [-cla] <source_file> > <destination_file>`
    - -c: compiles to LLVM and is the default,
    - -a: generates and prints the AST and SAST for program,
    - -l: compiles the LLVM but doesn't check that the LLVM is valid
4. Run `lli <destination_file>` to execute.

2.1 Hello World Example

An example of a 'Hello World' program written in Macaw is provided below:

```
void foo(string s) {
  print(s);
}
foo("Hello World");
```

2.2 How to Write a Macaw Program

A Macaw program is written as a series of functions and imperative statements. It is helpful to think of imperative statements outside of the scope of function definitions to be part of an overarching 'main' function, with one major difference: variables declared this way are considered global.

Function definitions and variable declarations must be made prior to referencing them. As an example, the below program will not compile:

```
foo();
void foo() {
  print("Hello World!");
}
```

Functions are identified by name and parameter list, so overloading them is possible. In addition, operator overloading is supported for a variety of arithmetic operations; consult the language reference manual to see the full list.

2.2.1 Data Types

Macaw has 3 primitive data types, which are:

- number - Floating point numbers used for arithmetic, logical, and relational operations. When printed, these numbers are rounded to the nearest thousandth. Floating point numbers were chosen for their mathematical properties.
- string - Character strings are used for printing statements to the console. They can be stored in variables or used as constants. A built-in function strcat(dest, source) is provided to concatenate any two strings together. Note, however, that doing so is destructive for the first argument.
- matrix - Two-dimensional arrays of numbers. Built-in support includes initialization of a `m×n` matrix of zeroes, initialization of a `m×n` matrix with a known matrix, indexing, and insertion.

Standard library functions available for matrix operations include, but are not limited to, pairwise addition, pairwise subtraction, dot product, and transposition.

Matrix data types can be accessed using [row, column] or [flattened] indexing. When using [flattened] indexing, the indices are counted across columns one row at a time. Thus, accessing a 4×4 matrix with [7] is the equivalent of accessing it with [2, 3].

Note that Macaw uses one-based indexing for its matrix datatype.

2.2.2 Control Flows

Macaw supports 4 common control flows which are if, while, for, and return. The syntax for each one is as follows:

```
if (predicate) {
  # statement block
  ...
}

while (predicate) {
  # statement block
  ...
}
```

For loops in Macaw are unique in that they are used to traverse matrices. They take the form of:

```
for (<number var> in <matrix var>) {
  # statement block
  ...
}
```

Each iteration over this loop changes the value of `<number var>` to the cell contents in `<matrix var>` as it sequentially indexes over `<matrix var>`. The `range(number start, number end)` function is part of the standard library and can be used to define a one-dimensional matrix with a range from start to end. Using this will allow you to mimic traditional for loops in other languages like C.

2.2.3 Define Functions

Functions in Macaw must declare a return type in their declarations. Allowable return types are:

- string
- matrix
- number
- void

All functions not declared void types return a value. As such, return statements are recommended in any function that is not defined as a void function. The data type of the return must match the datatype of the function. Macaw will automatically insert a return statement in the absence of one with default values as follows:

- number -> 0.000
- string -> (null)
- Matrix -> (null)

A (null) string can still be printed. However, referencing a null matrix will result in a segmentation fault. Examples of valid function definitions are below:

```
void foo(string s) {
  print(s);
}

number bar(int a) {
  return a + 8;
}
```

Chapter 3

Language Reference Manual

3.0.1 Compilation Target

Macaw compiles to LLVM which can then be further compiled into an executable to run on a variety of systems.

3.1 Lexical Conventions

In Macaw, there are seven kinds of tokens: Comments, Separators, Whitespace, Identifiers, Keywords, Literals, and Operators.

3.1.1 Comments

Comments in Macaw are denoted with the pound or hash character '`#`'. Any character between a '`#`' and a newline character is considered a comment. Only single-line comments are supported.

3.1.2 Separators

Separators are characters that separate and/or group tokens. Whitespace characters, such as space, tab, vertical tab, and form-feed, are only used to separate tokens and are ignored by the compiler (except inside string literals). Newline characters are only used to terminate a comment block and are otherwise ignored.

Other separators that are recorded by the scanner and used in parsing are: `( ) [ ] { } < > ; ,` Some of these have differing uses depending on the context so further explanation of their meaning in Macaw will be provided in the sections below.

3.1.3 Identifiers

Identifiers are sequences of characters used for naming variables and functions. Letters, numbers, and the underscore character '_' are allowed to be used in this manner. Note that the first character of an Identifier must be a letter and identifiers are case-sensitive such that `macaw`, `Macaw`, and `MACAW` all represent different identifiers.

3.1.4 Keywords

Keywords in Macaw are special identifiers to be used as part of the programming language itself. A keyword may not be used or referenced in any other way; function definitions and variable naming cannot override keywords.

```
number          void            else            while
matrix          operator        for             return
string          if              in
```

3.1.5 Literals

Numeric Literal

A number literal is a sequence of digits and optionally has a fractional part. Macaw only recognizes base-10 numeric representations. Examples of numbers are: '89', '4.2', '0.95'. Note that '8.' and '.89' are not valid representations of numbers.

- *[0-9]+(.[0-9]+)?*

String Literal

A string literal is a sequence of zero or more characters, digits, and symbols. Symbols are permitted except the newline character and the double quote character.

- *"[^\n "]*"*

Matrix Literal

Matrices are a collection of `number` objects. There are two ways to define a `matrix`:

- *[(expr_list ;)*]* for example [1, 2, 3; 8+9, 4-2, 4]
- *[ ] (expr, expr)* for example [ ] (2, 3)

The first creates a matrix where each cell is the result of the expression given for that cell. The '`,`' character separates cells on a row while '`;`' separates the rows. The size of the first row determines the number of columns, and all subsequent rows must have the same number of columns. The second creates an empty matrix with the row and column dimensions according the the result of the expressions in the parenthesis. All cells in this matrix are given the value '`0`'.

3.2 Identifiers

Identifiers are used to reference a variable or function. Variable identifiers cannot be directly followed by parens '`( )`', while function identifiers necessarily must be followed by parens. Both variable and function identifiers have an associated type which can be an type such as `number`, `string`, or `matrix`; or in the case of functions `void`.

3.3 Conversions

In Macaw you cannot convert between `number`, `string`, and `matrix`. For `number`, there is no type distinction between integer and floating point numbers. For calculations all numbers are considered floating point and when indexing into a matrix Macaw will convert the number into an integer representation to find the correct cell.

3.4 Expressions

The following expressions categories are ordered by precedence with the highest precedence first

3.4.1 Primary Expressions

Primary expressions group left to right.

- *identifier* - An identifier that is not immediately followed by parentheses '`( )`' is taken to be a variable identifier. The result of the expression is the value that the identifier 'points' to.
- *literal* - The value of a literal expression is simply the value corresponding to the literal.

- `(` *expression* `)` - Any expression can be wrapped in parentheses '`( )`' to create another expression. This is useful in overriding implicit precedence.
- *identifier* `(` *expression-list* `)` - An identifier followed by parentheses '`( )`' is taken to be a function invocation. Each expression inside the expression list is evaluated and passed into the function that is identified by the identifier.

Matrix Access

Square brackets are used to access a particular cell in a `matrix`. Each expression inside the brackets must evaluate to a `number` and any fractional part of the `number` is discarded. The indices of `matrix` types are 1-based, meaning that for `matrix A`, `A[1,1]` would return the value in the first cell. The string outside the brackets refers to a variable that is of type `matrix`. There are two variants, the first of which allows `matrix` to also behave intuitively as a vector construct.

- *identifier* `[` *expression* `]` - Returns contents of the cell of the `matrix` in a linear ordering of the `matrix` which iterates through the rows. (That is, all values in the first row are before any value in the second row).
- *ientifier* `[` *expression* `,` *expression* `]` - Returns the contents of the cell in the row equal to the result of the first expression and column equal to the result of the second expression.

3.4.2 Unary Expressions

- −*expression* - Negation of a `number`
- *expression*′ - Transpose of a matrix (not implemented by language, but by standard library)
- `!` *expression* - Logical 'not'. Takes an `number` value and returns 0 if the `number` is not 0, and 1 otherwise

3.4.3 Multiplicative Expressions

- *expression* `*` *expression* - Multiplication of `number` types
- *expression* / *expression* - Division of `number` types
- *expression* `%` *expression* - Modulus of `number` types

3.4.4 Additive Expressions

- *expression* + *expression* - Addition of `number` types
- *expression* − *expression* - Subtraction of `number` types

3.4.5 Relational Expressions

All relational operators are non associative, meaning that they can't be chained like `a < b < c`. They all yield 1 if the specified relation is true and 0 otherwise. Valid for `number` types

- *expression* < *expression*
- *expression* > *expression*
- *expression* <= *expression*
- *expression* >= *expression*

3.4.6 Equality Operators

Equality operators behave like the relational expressions but have a lower precedence. Valid for `number` types.

- *expression* = *expression*
- *expression* ! = *expression*

3.4.7 Logical Operators

Logical operators are left associative, though because of the nature of the operators order of evaluation actually doesn't matter. Both `&` and `|` have the same precedence, so if they both chained together they are applied left to right. Use parens to make the precedence order explicit.

- *expression* `&` *expression* - Returns 1 if both *expression*s are non 0, 0 otherwise. Valid only for `number` types
- *expression* `|` *expression* - Returns 1 if either *expression*s are non 0, 0 otherwise. Valid only for `number` types

3.4.8 Assignment Expression

Assignment is right associative and returns the assigned value. So for variables `a`, `b`, and `c`, the following is valid `a <- b <- c` and assigns the value of `c` to `b` and then to `a`

- *lvalue* <− *expression* - Assigns the result of *expression* to the identifier corresponding to the *lvalue*.

Function expressions that return `void` cannot be used in assignment expressions.

L Values

In assignment expressions the left hand side of the assignment must be a variable declaration, a variable identifier, or a cell identifier for a `matrix`.

- *type identifier* - Declares a variable with name *identifier* and type *type*. For more information see the Declarations section below.
- *identifier* - References an already declared variable.
- *identifier* `[` *expression* `]`
- *identifier* `[` *expression* `,` *expression* `]` - Reference a already declared and initialized `matrix` variable and assign the rvalue of the assignment to the cell indicated. The two variants of `matrix` access behave as described in the Matrix Access section.

Note that this grammar allows the following statement: `number a <- number b <- 4 + 5;` which evaluates `4 + 5` and then assigns that value to the newly declared variables `a` and `b`.

Value vs. Reference

`number` types are considered 'value' types and as such when a `number` variable is assigned to another `number` variable they both have the same value, but in different memory locations and further changes to either variable will not affect the other one. `matrix` and `string` types however are considered 'reference' types. When a `matrix` variable is assigned to another `matrix` variable, they both refer to the same `matrix` in memory. This means that a cell assignment made via one variable, affects the values in both variables (because they are referencing the same underlying `matrix`).

3.5 Statements

Statements control execution flow and allow for function definitions. Except where indicated, statements are executed in order

3.5.1 Statement Blocks

- `{` *(statement)** `}` - Statement blocks allow for multiple statements to be used where only one is expected

3.5.2 Single Statements

- *expression* `;` - Any expression can be turned into a statement by adding a semicolon. This is generally useful for assignment expressions or function calls that have side effects.
- *type identifier* `;` - A variable can be declared but left uninitialized as a statement.
- `if (` *expression* `)` *statement-block* - Conditional Statement evaluates the expression and if the result is non 0 executes the statement block
- `if (` *expression* `)` *statement-block* `else` *statement-block* - Behaves like the normal conditional but executes the else statement block if the *expression* evaluates to `0`.
- `while (` *expression* `)` *statement-block* - While loop executes the *statement-block* as long as the *expression* evaluates to non 0. The *expression*'s value is checked prior to each *statement-block* execution.
- `for (` *lvalue* `in` *expression* `)` *statement-block* - For loop. *expression* must return a `matrix` and the *lvalue* must be either `number` variable declaration or an *identifier* of an already declared `number`. Before each loop of the *statement-block* the *lvalue* is set to the element of the *expression* `matrix` corresponding to that loop. The loops are executed in order of the elements and continue until all elements have been assigned to the *lvalue*. The order of the elements follows the same ordering as the single coordinate access described in the matrix access section.
- `return;` - Returns control to the function's caller, additional code in the function past the `return` statement is not allowed. Can only be used with functions with return type `void`.
- `return` *expression* `;` - Similar to `return;` but also provides an expression to be evaluated and passed back to the caller. The type of *expression* should match that of the function's declared return type.

3.6 Declarations

Both variables and functions must be declared before they can be used as expressions. Variables can be declared and assigned to as part of a single assignment statement, while functions must be completely defined before they can be called. The main types in Macaw are `number`, `string`, and `matrix`.

3.6.1 Variable Declarations

- *type identifier* - Declares a variable with name *identifier* and type *type*. This statement can appear by itself (terminated by a `;`) or as the left hand side of an assignment expression. A identifier can only be declared once per scope (see scoping section below) and once declared the type of the variable cannot be changed.

3.6.2 Function Declarations

Functions must declare a return type as part of their declaration. This type represents the type of value that can be returned from the function when it is called in the code. In addition to the standard types, functions are also allowed to have a `void` type which allows them to indicate that they do not return anything. The compiler is then able to ensure that the function is not uses as part of an expression which expects a value like assignment or mathematical operators.

Functions have an argument list with any fixed number of arguments, including none. This list is made up of comma separated variable declarations. Finally the *statement-block* of a function is executed whenever the function is called.

- *type identifier* `(` *argument-list* `)` *statement-block*

– where *argument-list* is *(type identifier)**

Functions are identified by the combination of their identifier and the number and types of its argument list. This means that Macaw supports function overloading where two functions share the same identifier, but take in different types of arguments. This is a useful way to write mathematical functions that use different types (for instance equality between `number` types and equality between `matrix` types) without requiring support for polymorphic types.

3.7 Scope

There are two major scoping levels in Macaw. There is the global scope and function scope. A Macaw program is a collection of top-level statements which can be statements or function declarations. Any variable declared outside of the function declarations is considered inside the global scope. Because Macaw doesn't support nested functions, all functions are also in the global scope.

Function declarations create their own scope and variables declared in the argument list and in the body of the function can only be read from and written to during execution of that function. Global variables are accessible and can be assign to within functions. Function arguments and local variables (variables declared inside the body of a function) can share names with global variables. If an argument or local variable shares the name of a global variable, it 'shadows' that variable during function execution so that any reads or writes affecting that variable actually affect the argument or local variable, not the global variable. Note that because arguments and local variables share the function scope, all *identifiers* must be unique for arguments and local variables.

Functions can be invoked at any point after they are declared. Attempting to execute a function before it has been declared results in a compilation error. The code inside a function is declaration-checked, when the compiler sees an invocation of that function. This means that functions can reference other functions and global variables that were defined after the function, as long as all functions and variables referenced are declared prior to the function being invoked.

Variables declared inside of control constructs (like `if, else, for,` and `while`) are considered declared after these constructs end. This means within a scope they cannot be re-declared. Because Macaw does not enforce that a variable is assigned to before it is read from, declaring and assigning to variables inside a flow control construct and using it outside may result in a runtime error if at runtime the flow control construct was not entered. Due to this we encourage users of Macaw to place their declarations and initial assignments outside control constructs directly in the body of the function

3.8 Operator Overloading

In order to present the users of Macaw with a calculator like experience, Macaw has the ability to overload certain operators so that they have defined behavior for additional types. For instance, the language only supports the `number` type for the + operator. To add support for `matrix` types, we can easily define a function that performs the addition operation of the two matrices. Operator overloading allows us to tell the compiler to attach that function to the behavior of the + symbol. Like functions, the signature of an operator overload is the symbol used as an identifier and the parameter list. These must be unique and cannot duplicate language provided operators (for instance you cannot define a + overload that takes in two `number` params).

During type checking and code generation, the compiler determines if the types for the operands of a function are valid for the language. If they aren't, it then checks the operator overload table and sees if there has been a function defined to handle those types for that operator. If it does find such a function it accepts those types during type checking and uses that function for the code generation for that operator. Otherwise it presents a compile time error. This allows us to write a standard library greatly expanding the mathematical support for matrices without having to add undue complexity to the language.

To define an operator overload use the following alternate function declaration syntax:

- *type* `operator` *operator-symbol* `(` *argument-list* `)` *statement-block* - The *operator-symbol* is the operator to overload, the *type*, *argument-list*, and *statement-block* behave similarly to their roles in function declarations.

Operator overloading is fairly restricted. Overloading does not change any of the precedence rules for an operator and nor can it change a unary operator into a binary operator or vice versa. It also does not change side of the expression that a unary operator appears on. Only the following operators defined in the language can be overloaded:

- Unary: `'  !  -`
- Binary: `^ * / % .* ./ + - < > <= >= = != & |`

3.9 Language Functions

Language functions behave like user defined functions in how they are invoked. They are provided for tasks that are difficult or impossible to otherwise do with the language. While they are not keywords, users cannot define functions that use these names.

3.9.1 Print

`print` takes in one argument, and prints the value of that argument to the console. Function overloads for `number` and `string`.

3.9.2 Shape

`shape` takes in a single `matrix` argument and returns a `matrix` with 1 row and 2 columns. The first value (accessible with `[1]`) is the number of rows of the `matrix`. The second value (accessible with `[2]`) is the number of columns of the `matrix`. For example:

```
matrix A <- [1,2,3; 4,5,6];
matrix s <- shape(A);
print(s[1]); # Prints 2 (the number of rows) to the console
print(s[2]); # Prints 3 (the number of columns) to the console
```

3.10 Standard Library

The standard library that comes with Macaw has the following functions:

- `print(matrix)` - function overload that prints the passed in matrix
- `len(matrix)` - returns the total number of cells in a matrix
- `range(number, number)` - returns a matrix when the cell values in the matrix correspond to every number from the first argument to the second argument, inclusive. Useful for `for` loops.

It also contains the following operator overloads:

- `number ^ number` - Exponentiation operator
- `matrix '` - Matrix transpose
- `- matrix` - Matrix negation
- `matrix + number` - Scalar / Matrix addition
- `number + matrix` - Commutative overload
- `matrix + matrix` - Matrix addition
- `number - matrix` - Scalar / Matrix subtraction

- `matrix - number` - Matrix / Scalar subtraction
- `matrix - matrix` - Matrix subtraction
- `matrix * matrix` - Matrix dot product
- `number * matrix` - Scalar / Matrix multiplication
- `matrix * number` - Commutative overload
- `matrix ./ matrix` - Pairwise division of matrices
- `matrix .* matrix` - Pairwise multiplication of matrices

By using the build script to compile a program, the standard library is automatically built and linked to the program and available for use.

3.11 Grammar

```
- Literal
  number: [0-9]+(.[0-9]+)?
  string: [^ \n " ]*
  matrix: [ (expr_list;)+ ]
  empty_matrix: [ ] (expr, expr)

expr_list
  expr
  expr, expr

- ID ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
- Type
  number
  string
  matrix

- Return Type
  Type
  Void

- expr
  - expr
  expr '
  ! expr
  expr ^ expr
  expr * expr
  expr / expr
  expr % expr
  expr + expr
  expr - expr
  expr < expr
  expr > expr
  expr <= expr
  expr >= expr
  expr = expr
  expr != expr
  expr & expr
```

```
  expr | expr
  lvalue <- expr
  ID [ expr ]
  ID [ expr, expr ]
  ( expr )

- lvalue
  ID
  type ID
  ID [ expr, expr ]

- statement-block
  { statement list }

- statement
  expr;
  type ID;
  if ( expr ) statement_block
  if ( expr ) statement_block else statement_block
  while ( expr ) statement_block
  for ( lvalue = expr to expr ) statement_block
  return;
  return expr;

- function_decl
  type ID ( argument_list ) statement_block

- argument_list
  [ ]
  (type ID) list

- operator_overload
  type operator operator_symbol ( argument_list) statement_block

operator_symbol:
+
-
*
/
^
%
.*
./
=
!=
<
<=
>
>=
&
|
!
'
```

```
- program
  global_statement list

- global_statement
  statement
  function_decl
  operator_overload
```

Chapter 4

Project Plan

4.1 Group Meeting Schedule

Because this semester was an abbreviated one (only 5 weeks), we had to get started on the project immediately and needed to spend substantial time each week together as a group. We found that meeting before class worked the best for everyone and initially met at 4:00 until class at 5:30 and later extended the meetings to start at 2:30 as we progressed in the project. Meeting before class allowed us to ensure that any blocking issues that required answers from the teacher were known so that we could ask the teacher during a break in class. We also planned for everyone to be on Skype on Friday afternoon for an additional progress check, though this check was much more informal.

These meetings enabled us to ensure that everyone was up to date on the next goals for the project and to ensure that any blockers could be addressed quickly so we remain productive.

4.2 Timeline of the Project

We tried to have the deliverables finished ahead of the required timeline. Based on that timeline we set rough deadlines for ourselves. Naturally these deadlines moved around as we worked due to some areas being easier than expected and many more areas being more difficult than expected. Our initial optimistic project timeline was:

Date	Milestone
7/11	Complete Proposal
7/17	Finish LRM
7/20	Complete scanner, parser, and AST
7/27	Compile Hello World to LLVM
08/01	Semantic Checking/SAST generation finished
8/05	Matrix support finished
08/08	Test suite finished
08/11	Project Report and Presentation Finished

4.3 Project Log

The actual timeline of achievements of our project was:

Date	Milestone
7/11	Completed Proposal
7/20	Finalized LRM
7/25	Scanner/Parser/AST finished
7/28	Hello World in LLVM
08/08	Semantic Checker/SAST finished. Matrix support in CodeGen finished
08/09	Test Suite finished, various bug fixes. Compiler ready for demo
08/11	Project Report and Presentation Ready

4.4 Group Member Roles

Roles of the group members are posted below. Note that there is significant overlap between roles as group members often assisted others in completing their work. The roles are listed loosely based on most time spent in a particular role to the least time spent. We also indicate 'researcher' below to describe activities aimed towards identifying resources to help our progress. This was critical to ensuring that the project was completed on time and didn't neatly fit into any of the 4 existing roles.

Member	Roles
William Hom	Researcher, System Architect, Tester, Manager
Yi Jian	Tester
Joseph Baker	Language Designer, System Architect, Researcher
Christopher Chang	Tester, Manager

4.5 Development Environment

In addition to our twice weekly meetings we tried to be on Skype as much as possible. A few group members lived far from campus, so an effort was made to be 'on-call' whenever we weren't able to meet. We were able to respond immediately to each other if a situation arose.

All group members had MacBook Pros for development purposes, which simplified things as far as installation of LLVM. Because of the consistent environment we were able to assist each other with regard to system problems.

We created a Git repository in BitBucket to keep track of our changes. All members had full access. There was a soft requirement that all commits needed to be approved by at least one other group member as a way of making sure that code was being looked at.

For editing, we used primarily Vim or Sublime, depending on what we were comfortable with.

There was no official code style enforced during development. It was recommended that everyone stay consistent with their own formatting. Effort was made to make our Ocaml code legible and consistent when it came to formatting our blocks.

Chapter 5

Language Evolution

At the outset, the goals for this language were relatively ambitious. We had always intended to implement matrices of some sort, but we intended to include as built-in functions a large variety of matrix operations (pairwise addition, pairwise subtraction, dot product, transposition, etc.). As we revisited the idea and project proposal, we realized that it would be much simpler (and make our language more interesting at the same time) if we implemented enough support for matrices such that a standard library would be able to cover any matrix operation a user would need. We decided that matrix initialization, access, and insertion should be sufficient to implement other necessary operations.

With this in mind, we decided to shift gears into supporting operator overloading for our language. While we could've accomplished most of our goals using only functions, we felt that a seamless user experience would be more useful.

The scanner, parser, and AST went through several iterations before we arrived at what we believe represented our language. In the beginning, while we were trying to launch our prototype, most of our attention was put towards creating a simple enough grammar that would allow us to continue moving forward. Then, by using careful branching of our Git repository, some of us were able to continue moving forward with the system architecture while others refined the scanner, parser, and AST.

For the evaluator and compilator, we would find ourselves going back and forth deciding how to accomplish our language goals. We realized that we couldn't just group the source commands into lists of function and statements and hope everything was declared in the proper order. Not only did we have to transform our AST into a something that represented the executable we wanted, we also needed specific logic handling what to do, for example, if our compiler sees a '+' symbol; do we add them like numbers or look for an overloaded function? After careful thought and deliberation, we opted to use the evaluator to transform our parsed tree into a semantically checked tree representation of our program. This included support for operator and function overloading by parsing our functions into unique `id_parameter_list` signatures, parsing out global statements, and rolling everything else together into a 'main' function. This is all done after our evaluator successfully checked the program sequentially to ensure that it abided by our language conventions. As a result, our code generation is relatively simple and straight-forward as the bulk of the work is being done by the evaluator.

Our first function overloading test was to combine all of our print functions. Our first operator overloading test was to have addition of string data types to concatenate them. Once we had those working, we knew we were on the right track.

The last major challenge once everything else was done was the matrix implementation. While we uncovered a way to implement it using Ocaml LLVM bindings, we felt things would be far simpler if we wrote C code to support this, then link it into the LLVM. Once we were able to confirm that we could do this, it was a matter of making sure the types matched up correctly. In the end, all of our matrix implementation and support (initialization, access, and insertion) were taken care of with additional C code.

A lot of others things were minor decisions in comparison. We also decided to make all numbers 64-bit floating point because of issues with auto-casting in LLVM and felt that floating-point math was more useful. In addition, since for loops are while loops with some additional syntax, we opted to exclude them and instead give users a way to iterate over their matrices. The last consideration we had was concerning print statements. We wanted to keep things simple by not allowing users to specify a format string, but we didn't want to always force a newline on our

users. For this, we simply created both a print (with newline) and and printnl (without newline).

Once our language was feature-complete, we needed to write our standard library functions, which was relatively straight-forward. In doing this, we encountered small quirks that we had to fix. In the end, we felt we were able to deliver on all of our primary language requirements.

Chapter 6

Translator Architecture

6.1 Scanner, Parser, and AST

The scanner reads in our source files and tokenizes them appropriately. The parser will take the tokens and transforms it into an abstract syntax tree.

If an invalid keyword is passed, a parsing error will occur.

The abstract syntax tree serves as a representation of what a typical Macaw program looks like.

6.2 Semantic Checker

The semantic checker (called evaluator in the source) has the task of ensuring that the program we are going to be generating code from is completely valid (though it can't ensure correctness). The primary things it is doing are ensuring that all variables and functions that are used are properly defined before their use and ensuring that the types used in various locations match what is expected in those scenarios. Once it has done some of its checks, it can reshape the AST into a more convenient shape.

The evaluator starts by initializing an empty 'app state'. This is a structure which contains all the functions, variables, operator overloads, and state information about the application at a moment in time. It then starts to traverse the AST as if it were executing the program. Each time it sees a declaration, it adds it to the app state. Every time it sees a statement requiring a variable or function, it ensures that such a variable or function exists in the app state. When it sees a function invocation, it starts scanning the body of that function if the body hasn't been scanned before. It performs a similar check on operators. If an operator is known by the language it continues, but if an operator has been overloaded, it will check the overload's function body at this time. Note that to determine which function or operator overload to call it needs to know the types of the expressions involved. This means that the declaration checker requires limited type checking abilities.

Once declaration checking is finished, the semantic checker uses the completely built application state to generate a Semantically checked Abstract Syntax Tree (SAST). Note that this tree hasn't been fully type checked yet. The main changes in this tree are that declarations are grouped into lists by scope and removed from the statements themselves. This allows easy lookup of a variable or function by the type checker and codegen. The other major change in the program tree is that all functions with parameters and all operator overloads are replaced with unique function calls. The unique function name is built by taking the function name and appending all the parameter types to it. This means that after this stage the compiler can assume that operators are known by the language and that despite function overloading, each function has a unique name.

At this stage for loops and matrix initialization blocks are translated into other operations. The for loop subtree in the AST is replaced with a code block that uses a while loop and some system-named variables to achieve the for loop behavior. Matrix initialization blocks are translated into function calls to generated functions that create a matrix of the correct size and assign the passed in values to the appropriate cells in the matrix.

Using the SAST the semantic checker then moves on to type checking. This is where it ensures that all assignments are of the appropriate type for the variable being assigned to, all matrix initializations are only using numbers, that predicates for if/for/while constructs are correct for the language, and that return expressions match

the return type of the function where the return expression occurs. It is also where we ensure that no statements follow the return statement and other smaller checks that the language requires. Note that because the declaration checker has already ensured that only functions with the correct type are invoked and that operators are all recognized by the compiler, we don't have to do much additional checking to ensure that the correct type of parameter is passed to a function or operator. Though we do still have to recurse into those expressions to ensure they are type checked.

Once the type checking is finished the program is considered semantically valid. At this point the evaluator cleans out any language-based declarations that were added to the SAST like built in operators or built in functions. This is to ensure that the codegen doesn't attempt to automatically generate anything for these language features and so it doesn't need to worry about filtering them out itself.

6.3 SAST

The SAST represents a semantically checked Macaw program that has been restructured to eas code generation. At this point all function names are unique and declarations have been associated with the scope they are valid in.

6.4 Codegen

Our compilator takes the SAST given to it from the semantic checker and proceeds rather mechanically from there. The group made a decision to not process any logic during this phase or make any decisions. With the AST fully transformed, the compilator essentially transcribes what it receives into LLVM bytecode. For example, since overloaded operators are transformed into function calls, when the compilator encounters an addition symbol, it doesn't need to decide whether we're dealing with numbers, matrices, or strings. It can safely assume that we're always dealing with numbers. A similar situation occurs when we have function overloading; the compilator doesn't need to figure out which function to call because every function signature is unique.

Our language makes use of LLVM linking to provide matrix support. This support is written entirely in C. This would be compiled to LLVM bytecode and linked into our compiler when passed a source file.

The compilator first allocates all global variables and assigns any values to them (if applicable). It then iterates over each function, adding any local function variables to a separate map (to ensure proper scoping). It then proceeds to build the body of each function based on what it sees in the SAST. Transformed function calls can be handled with a single generic match, but our built-in functions written in C needed special cases written for them in instances where we need them because these aren't actually defined by the user and thus wouldn't be part of the SAST.

6.5 Standard library

The standard library for Macaw is written in the _includes/stdlib.ma file. It contains standard library functions. This includes many overloaded operators and functions used for manipulating matrices. The mcaw script takes this file and inserts it to the head of the source file before compiling to LLVM. To see how this works, you can try to compile the source code using the compiler 'macaw.native' directly. If your code is reliant on anything in our standard library (e.g., matrix pairwise addition), it will return a compilation error.

Chapter 7

Test Plan and Scripts

Having a thorough LRM made it easier for us to put together a complete test suite. While we were waiting for a functioning programming language to be implemented, the testers began devising test cases and scenarios that they could use once we were ready to begin.

The order of battle when it came to testing was that the system architects would first implement some feature. They would then write a few toy programs to make sure it was working correctly. After this was done, these programs would be passed off to the testers who would then break them down into component unit tests.

As the language was nearing completion, the testers began writing unit tests in earnest. A test script was written to run all tests at once. The process of writing these tests uncovered bugs in our compiler that we then had to patch. In the end, there were 139 tests written to test the Macaw compiler. All success and failing cases are meeting expected outputs.

While we did write some programs that used many features simultaneously, we did not encode those as integration tests.

Chapter 8

Conclusions

8.1 William Hom

This was a stressful class! We chose LLVM as our target which proved to be challenging as not only were we trying to learn Ocaml but understand the LLVM bindings at the same time. While we hit several of our milestones ahead of schedule, the core parts of our project weren't making much progress. It was only by diving deep into the LLVM documentation was I able to find information about the linker and write some C code that helped simplify our code generation significantly. Everything came together towards the end of the week of August 1st. This led to a mad rush to implement features, write tests, fix bugs, write more tests, fix other bugs, etc. In the end, we accomplished most of what we set out to do.

Managing proved to be challenging as well, as three members of the group all had jobs. Combined with the tight time constraints, it was difficult finding the perfect balance of spending time in group sessions while respecting that some of us had our own daily schedules.

8.2 Yi Jian

This was my first time being a tester. Testing is much harder than I thought. Testers must be really detail oriented. The hardest part is you have to think about all the ways to make the language break for the failure tests, which requires a lot of experience in programming. Tester is an important role to know our language better and make it more complete.

8.3 Joseph Baker

I learned a lot about how parsers and compilers actually work. I enjoyed being able to write the semantic checker and get into the details of how to transform the raw input from a user into an organized structure that can be used by code generation.

I absolutely see why test driven development can work so well for compiler projects and think that we should have leveraged that better. Although I was writing throw-away code to test behavior as I went, having a test-suite already built (or being built concurrently) would have greatly increased my confidence of the code correctness as we went along.

8.4 Christopher Chang

Working so closely with an awesome team to complete a seemingly huge project has allowed me to learn so much in a short amount of time. It was a bit tough in the beginning because the other three members of my original group ended up dropping the class so I was already a week or so behind by the time I joined this one. I think one of the things was great with my team was that they were always very communicative, even when we were not together. As a result, for most of this project, the development of the semantic checker, codegen, etc. seemed to

be very much in sync. However, with the testing process, this wasn't always the case. A lot of our final tests were written in the last few days as we were waiting for certain features to be completed. In the future, I'd like testing to be more of an iterative process. As the project manager, I felt like a lot of what I learned had to do with the overall development process. To be able to see the various parts of the project develop from a high level made me realize that certain design decisions could make a big difference later down the road even if it didn't seem like a big deal at the time. For example, one of the bottlenecks we faced was the matrices functionality. Because of certain design decisions we made, it didn't hold back the rest of our development since the semantic check didn't depend on that.

8.5 Overall

Throughout the project, we probably could've made more of an effort to make sure the system architects and the testers were in sync. It wasn't until the language was nearing feature-completion that we began writing most of the tests.

The group consisted of people with different backgrounds and experiences when it came to programming. As the semester progressed, we all settled into roles that hid our weaknesses and accentuated our strengths. At the same time, we did our best to ensure that everyone was able to contribute to all parts of the project.

We should have read the LLVM documentation more thoroughly. The bindings were relatively tricky and we would've benefited spending a little bit more time understanding how LLVM works. A lot of resources were tied up trying to implement matrices because we couldn't seem to figure out the correct bindings to use. The discovery of linking really made a big difference in helping us complete the project.

Large parts of the project seemed confusing at first, but as we wrote the scanner, parser, AST, etc. it became clearer as to what we were doing and how we were accomplishing it. In the end, the group learned quite a lot about programming languages and translators.

We employed a lot of iterative programming techniques that proved very helpful. For example, when we realized we needed to transform the AST into a list of global variables and functions, one of us implemented a very crude version of it in our compilator. Once we had it working, we were able to pass the code back to another system architect who was working on something else to implement it more fully. This was employed while putting together our scanner, parser, and AST as well.

Chapter 9

Full Code Listing

9.1 Tests

../src/testall.sh

```
#!/bin/sh

#  Regression testing script for Macaw
#  Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Macaw Compiler. If this doesn't work, try using ./macaw.native.
# However, keep in mind that tests relying on standard library functions
# will fail.
MACAW="./mcaw"
#MACAW="_build/macaw.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.ma files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to diff file
```

```
Compare() {
    generatedfiles="$generatedfiles␣$3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
        SignalError "$1␣differs"
        echo "FAILED␣$1␣differs␣from␣$2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    if [ $1 -eq "./mcaw" ]; then
        eval $* || {
        SignalError "$1␣failed␣on␣$*"
        return 1
        }
    else
        eval $*
        return 0
    fi
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
        SignalError "failed:␣$*␣did␣not␣report␣an␣error"
        return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.ma//'`
    reffile=`echo $1 | sed 's/.ma$//'`
    basedir="`echo␣$1␣|␣sed␣'s/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "######␣Testing␣$basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles␣${basename}.ll␣${basename}.out" &&
    Run "$MACAW␣-c" $1 "␣>␣${basename}.ll" &&
    Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
        fi
        echo "OK"
        echo "######␣SUCCESS" 1>&2
    else
        echo "######␣FAILED" 1>&2
        globalerror=$error
    fi
}
```

```
CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.ma//'`
    reffile=`echo $1 | sed 's/.ma$//'`
    basedir="`echo␣$1␣|␣sed␣'s/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "######␣Testing␣$basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles␣${basename}.err␣${basename}.diff" &&
    RunFail "$MACAW␣-c" $1 "2>␣${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
        fi
        echo "OK"
        echo "######␣SUCCESS" 1>&2
    else
        echo "######␣FAILED" 1>&2
        globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
        k) # Keep intermediate files
            keep=1
            ;;
        h) # Help
            Usage
            ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could␣not␣find␣the␣LLVM␣interpreter␣\"$LLI\"."
  echo "Check␣your␣LLVM␣installation␣and/or␣modify␣the␣LLI␣variable␣in␣testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ ! -f "macaw.native" ]; then
    echo "Please␣run␣make."
    exit 1
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.ma␣tests/fail-*.ma"
fi

for file in $files
do
    case $file in
```

```
        *test-*)
            Check $file 2>> $globallog
            ;;
        *fail-*)
            CheckFail $file 2>> $globallog
            ;;
        *)
            echo "unknown␣file␣type␣$file"
            globalerror=1
            ;;
    esac
done

exit $globalerror
```

all_tests_output.txt

```
fail-conversion1.ma
==========
# test number to matrix conversion
number a;

matrix b <- [ 1, 2, 3; 8+9, 4-2, 4];

a <- b;  # should fail

---
Fatal error: exception Failure("number type expected from b!")
---

fail-conversion2.ma
==========
# test string to number conversion
number failedNum <- 5;

string failedString;
failedString <- "num to string fail";

failedNum <- failedString;  # should return error

---
Fatal error: exception Failure("number type expected from failedString!")
---

fail-conversion3.ma
==========
# test string to matrix conversion

string failedStringMatrix;
failedStringMatrix <- "[ 1, 2, 3; 8+9, 4-2, 4]";
matrix faildString;

faildString <- failedStringMatrix; # should return error message

---
Fatal error: exception Failure("matrix type expected from failedStringMatrix!")
---

fail-declaration1.ma
==========
# test number declaration

number Num <- "15"; #should return error message

---
Fatal error: exception Failure("number type expected from "15"!")
---
```

```
fail-declaration2.ma
==========
#test string declaration
string s1 <- hi; #should return error message

---
Fatal error: exception Failure("Use of undeclared variable hi!")

---

fail-declaration3.ma
==========
#test matrix declaration

matrix b <- [ 1, 2, 3;; 8+9, 4-2, 4];  #should return error message

---
Fatal error: exception Parsing.Parse_error
---

fail-declaration4.ma
==========
# can't declare void variables!

void foo;

print("HELLO!");

---
Fatal error: exception Failure("Illegal void variable foo!")

---

fail-declaration5.ma
==========
# jagged matrix!

matrix b <- [1,2,3,4;1,2,3];

print(b);

---
Fatal error: exception Failure("Matrix initialization cannot be jagged! 1, 2, 3, 4; 1, 2, 3")
---

fail-expressions1.ma
==========
# testing failing matrix access

string a <- "Hello World";

matrix b <- [1,2,3;a,5,6];

print(b[4]);


---
Fatal error: exception Failure("Only number type allowed in matrix init block. string used!")
---

fail-expressions2.ma
==========
# failing test for expressions, cannot have booleans in print statements

number a <- 1;

number b <- 1;
```

```
print(a&b);
---
Logical operators only work with integral types!
  %tmp = and double %a, %b
LLVM ERROR: Broken module found, compilation aborted!

---

fail-expressions3.ma
==========
# failing test for expressions, cannot have booleans in print statements

number a <- 1;

number b <- 1;

print(a|b);
---
Logical operators only work with integral types!
  %tmp = or double %a, %b
LLVM ERROR: Broken module found, compilation aborted!

---

fail-expressions4.ma
==========
# failing test for unary expressions, logical not

number a <- 1;

if (!a = 0){
        print("Success");
}



---
Logical operators only work with integral types!
  %tmp = xor double %a, 0xFFFFFFFFFFFFFFFF
LLVM ERROR: Broken module found, compilation aborted!

---

fail-expressions5.ma
==========
#testing string expressions, cannot print double quotes, only single quotes

print("""");
---
Fatal error: exception Parsing.Parse_error

---

fail-expressions6.ma
==========
# testing expressions, newline

print("
");
---
Fatal error: exception Parsing.Parse_error

---

fail-expressions7.ma
==========
# testing relational operator, cannot be chained
```

```
number a <- 1;

number b <- 2;

number c <- 3;

if ( a<b<c ){
        print("Success");
}
---
Fatal error: exception Parsing.Parse_error

---

fail-expressions8.ma
==========
# failing test with string and matrix types

matrix a <- [1,2,3; 4,5,6];

string b <- "Hello World";

a[4] <- b;

print(a[4]);


---
Fatal error: exception Failure("number type expected from b!")

---

fail-functions1.ma
==========
# failed function overloading
# identical signatures

void foo(string s) {
        print(s);
}

void foo(string s) {
        print(s);
}

print("I shouldn't be printed!");
---
Fatal error: exception Failure("Duplicate function foo(string)!")
---

fail-functions11.ma
==========
# multiple returns from functions!

number foo() {
    return 0;
    return 1;
}

print("YOU SHALL NOT PASS!");

---
Fatal error: exception Failure("No statements are allowed after return inside a block!")
---

fail-functions12.ma
==========
# Parsing error when declaring on return!
```

```
number foo() {
    return number a;
}

print(a);

---
Fatal error: exception Parsing.Parse_error

---

fail-functions13.ma
==========
# referencing an undeclared variable in a function!
# variable declared after function

void foo(string s) {
    print(a + 5);
    print(s);
}

foo("NOPE!");
number a <- 10;

---
Fatal error: exception Failure("Use of undeclared variable a!")

---

fail-functions2.ma
==========
# defining a shape function
# overriding a system call isn't allowed

matrix shape(matrix a) {
        print(a);

        return shape(a);
}

matrix b <- shape([](5, 6));

print(5);
---
Fatal error: exception Failure("Duplicate function shape(matrix)!")
---

fail-functions3.ma
==========
# no return type specified in function declaration

foo() {
        number a <- 5;
        return a;
}

number a <- foo();
---
Fatal error: exception Parsing.Parse_error
---

fail-functions4.ma
==========
# declare a return type but return something else!
# LOL

number foo(number a, number b) {
```

```
        return [](5, 6);
}

number a <- foo(6, 6);
---
Fatal error: exception Failure("number type expected from [ ](5, 6)!")

---

fail-functions5.ma
==========
number foo(string s) {
    print(s);
    return 1;
}

void foo(string s) {
    print(s);
}

number a <- foo("hello");
number b <- foo("world");
---
Fatal error: exception Failure("Duplicate function foo(string)!")
---

fail-functions6.ma
==========
# function return doesn't match return type!

number foo(string s) {
        matrix b <- [](5, 6);

        return b;
}

print("Hello!");
---
Fatal error: exception Failure("number type expected from b!")
---

fail-functions7.ma
==========
# assignment on return not right!

number foo(number a) {
        return a;
}

matrix b <- foo(5);
---
Fatal error: exception Failure("matrix type expected from foo_number(5)!")
---

fail-functions8.ma
==========
# foo should not return anything!

void foo() {
        return 0;
}

number a <- foo();

print(a);
---
Fatal error: exception Failure("number type expected from foo_()!")
---
```

```
fail-functions9.ma
==========
# can't call functions before they're declared!

foo();

void foo() {
    print("YO!");
}

---
Fatal error: exception Failure("Use of undeclared function foo!")

---

fail-overload1.ma
==========
# too many variables!

number operator .* (number a, number b, number c) {
        return a + b + c;
}

print(5 .* 7 .* 3);
---
Fatal error: exception Failure("Use of unknown operator number .* number!")
---

fail-overload2.ma
==========
# declare overloaded operator, called with wrong types

number operator .* (number a, number b) {
        return a + b;
}

string s <- "Hello!";
number b <- 6;

print(s .* b);
---
Fatal error: exception Failure("Use of unknown operator string .* number!")

---

fail-overload3.ma
==========
# same function signatures!

number operator ^ (number a, number b) {
        return a * b;
}

number operator ^ (number a, number b) {
        return a * b;
}

print("Don't see me!");
---
Fatal error: exception Failure("Duplicate function _pow(number,number)!")

---

fail-parser1.ma
==========
# invalid keyword!
```

```
int a;
---
Fatal error: exception Parsing.Parse_error
---

fail-parser2.ma
==========
# if statements always require brackets!

number a <- 10;

if (a = 10)
        print("Nope.");
---
Fatal error: exception Parsing.Parse_error
---

fail-parser3.ma
==========
# no semi-colons :(

void foo() {
        print("hello");
}

foo()
---
Fatal error: exception Parsing.Parse_error
---

fail-parser4.ma
==========
# capitalization matters!

String s <- "Hello World";

print(s);
---
Fatal error: exception Parsing.Parse_error
---

fail-parser5.ma
==========
# for loops also must be in brackets!

matrix a <- [1;2;3;4;5;5];

for (number b in a)
        print(b);
---
Fatal error: exception Parsing.Parse_error
---

fail-parser6.ma
==========
# while loops must be in brackets!

while (1 = 1)
        print("I'm stuck!");
---
Fatal error: exception Parsing.Parse_error
---

fail-scope1.ma
==========
# testing different function scopes

number a(){
```

```
    number first <- 6;
    print (first); # function a scope, prints 6
}

number b(){
    print (first); # failed test: can't find "first"
}
---
Fatal error: exception Failure("Use of undeclared variable first!")

---

fail-scope2.ma
==========
# failing scope test

string a() {

        string s;
    s <- "6";
    print (s); # function a scope, prints "6"

}

string b(){
    print (s); # failed:can't find s
}

---
Fatal error: exception Failure("Use of undeclared variable s!")

---

fail-scope3.ma
==========
# testing scoping with matrices

matrix a(){
    matrix m <- [1,2];
    print (m); # function a scope, prints [1,2]
}

matrix b(){
    print (m); # failed:can't find m
}

---
Fatal error: exception Failure("Use of undeclared variable m!")

---

fail-statements1.ma
==========
# main program returns void, so this shouldn't be allowed.

number a;
a <- 5;

return a;
---
Fatal error: exception Failure("void type expected from a!")

---

fail-statements2.ma
==========
# numbers don't evaluate to anything
```

```
while (1) {
        print("yep.");
}
---
Branch condition is not 'i1' type!
  br double 1.000000e+00, label %while_body, label %merge
double 1.000000e+00
LLVM ERROR: Broken module found, compilation aborted!
---

fail-statements3.ma
==========
# string equality doesn't work.

string s <- "Hello!";

while (s = "Hello!") {
        print("Yep!");
        s <- "Nope.";
}
---
Fatal error: exception Failure("Use of unknown operator string = string!")
---

fail-statements4.ma
==========
# nested ifs must be in else blocks!

number b <- 99;

if (b = 88) {
        print(b);
} else if {
        print("Nope.");
}
---
Fatal error: exception Parsing.Parse_error
---

fail-statements5.ma
==========
# flipping numbers and matrix in for loop

number a;
matrix b <- [1;2;3;4;5];

for (b in a) {
        print(b);
}

---
Fatal error: exception Failure("check_expr_typ: Can't find len_number")
---

fail-statements6.ma
==========
# flipping numbers and matrix in for loop
# define a len_number function

number len(number b) {
        return b;
}

number a;
matrix b <- [1;2;3;4;5];

for (b in a) {
        print(b);
```

```
}

---
Fatal error: exception Failure("matrix type expected from a[_tmp_for_i18, 0]!")
---

fail-statements7.ma
==========
# Cannot chain relational operators!

number a <- 10;

if ((a = 10) = 1) {
        print("Nope.");
}
---
Both operands to FCmp instruction are not of the same type!
  %tmp1 = fcmp oeq i1 %tmp, double 1.000000e+00
LLVM ERROR: Broken module found, compilation aborted!
---

test-conversion1.ma
==========
# test number to number conversion
number successNum <- 15;
number number2 <- 35;

successNum <- number2;

print(successNum); #should print 35

---
35.000

---

test-conversion2.ma
==========
# test string to string conversion

string successString <- "Failure";
string string2 <- "Success";

successString <- string2;

print(successString); # should print "Success"


---
Success
---

test-conversion3.ma
==========
#test matrix to matrix conversion

matrix successMatrix <- [ 1, 2, 3; 8+9, 4-2, 4];
matrix matrix2 <- [ 2, 1, 5; 10, 9, 8; 500-3, 1234, 100];

successMatrix <- matrix2;

print(successMatrix); # should print out [ 2, 1, 5; 10, 9, 8; 500-3, 1234, 100];
```

```
---
2.000, 1.000, 5.000;
10.000, 9.000, 8.000;
497.000, 1234.000, 100.000;

---

test-declaration1.ma
==========
# test number declaration

number Num1 <- 15;

print(Num1); #should print 15


---
15.000

---

test-declaration2.ma
==========
#test string declaration

string s <- "hi";

print(s); #should print "hi"


---
hi
---

test-declaration3.ma
==========
#test matrix declaration

matrix b <- [ 1, 2, 3; 8+9, 4-2, 4];

print(b); #should print [ 1, 2, 3; 17, 2, 4];


---
1.000, 2.000, 3.000;
17.000, 2.000, 4.000;

---

test-declaration4.ma
==========
# test declaration in separate lines

number a;
a <- 8;
print(a);
---
8.000
---

test-declaration5.ma
==========
string s;

print(s);
```

```
---
(null)

---

test-declarations6.ma
==========
# numbers are initialized to zero!

number a;

print(a);


---
0.000

---

test-declarations7.ma
==========
matrix a <- [3,4];

print(a[2]);

---
4.000

---

test-expressions1.ma
==========
# testing unary expressions

number a <- 50;

if(!(a=40)){
        print ("Success");
}


---
Success
---

test-expressions10.ma
==========
# unary expressions, matrix transpose

matrix a <- [1, 3, 5; 2, 4, 6];

matrix b <- a';

print(b[4]);


---
4.000
---

test-expressions11.ma
==========
# testing primary expressions, matrix access, logical not

number a <- 1;

if(!(a=0)){
        print("Success");
```

```
}


---
Success

---

test-expressions12.ma
==========
# testing multiplicative expressions

matrix a <- [1, 2, 3];

matrix b <- [4; 5; 6];

print(a * b);



---
32.000;
---

test-expressions13.ma
==========
# testing multiplicative expressions

number a <- 5;

number b <- 2;

print(a*b);


---
10.000
---

test-expressions14.ma
==========
# testing multiplicative expresssions, division

number a <- 12;

number b <- 2;

print(a / b);

---
6.000

---

test-expressions15.ma
==========
# testing multiplicative expressions, modulo

number a <- 11 ;

number b <- 2 ;

print(a%b);


---
1.000
```

```
---

test-expressions16.ma
==========
# additive expressions, expressions + expression

string a <- "Hello ";

string b <- "World";

print(strcat(a,b));

---
Hello World

---

test-expressions17.ma
==========
# testing additive expressions, numbers and matrices

matrix a <- [1,2,3;4,5,6];

matrix b <- [1,2,3;4,5,6];

matrix c <- a + b;

print(c[4]);

---
8.000
---

test-expressions18.ma
==========
# testing additive expressions, subtraction

matrix a <- [1,2,3;4,5,6];

matrix b <- [1,2,3;4,5,6];

matrix c <- a - b;

print(c[5]);



---
0.000
---

test-expressions19.ma
==========
# testing relational expressions

number a <-5;

number b <-10;

if(a<b){

        print("Success");
}


---
Success
```

```
---

test-expressions2.ma
==========
# testing primary expressions

matrix a <- [1, 2, 3; 8+9, 4-2, 4];

matrix b <- [4, 5, 3; 7, 5, 3];

number c <- a[4] - b[1]; #[-3, -3, 0; 10, -3, 1]

print(c);


---
13.000
---

test-expressions20.ma
==========
# testing relational expressions

number a <- 5;

number b <- 10;

if(!(a>b)){
        print("Success");
}

---
Success
---

test-expressions21.ma
==========
# testing relational expressions

number a <- 5;

number b <- 5;

if (a<=b){
        print("Success");
}




---
Success

---

test-expressions22.ma
==========
# testing relational expressions

number a <- 5;

number b <- 5;

if (a>=b){
        print("Success");
}
```

```
---
Success

---

test-expressions23.ma
==========
# testing equality operators

number a <- 5;

number b <- 5;

if(a=b){

        print("Success");
}


---
Success

---

test-expressions24.ma
==========
# testing equality operators

number a <- 5;

number b <- 5;

print (a!=b);


---
0.000
---

test-expressions25.ma
==========
# testing expressions, logical operators

number a <- 10;

number b <- 12;

if((a=10)&(b=12)){
        print("Success");
}

---
Success

---

test-expressions26.ma
==========
# testing expressions, logical operators

number a <- 0;

number b <- 12;

if((a=5)|(b=12))
{
```

```
        print("Success");
}


---
Success

---

test-expressions27.ma
==========
# testing expressions, assignment expression

number a <- 5 ;

number b <- 10 ;

number c <- 15 ;

number d <- a <- b <- c ;

print(d);


---
15.000
---

test-expressions28.ma
==========
# testing expressions, lvalues

number a;

a <- 5;

print(a);


---
5.000

---

test-expressions29.ma
==========
# testing assingment expressions, L Values

number a <- 5;

number b <- 10;

matrix c <- [1,2,3;4,5,6];

print(c[b-a]);



---
5.000
---

test-expressions3.ma
==========
# testing primary expressions

number a <-5;
```

```
print(a);
```

```
---
5.000
---
```

```
test-expressions30.ma
==========
# testing expressions, value versus reference

number a <- 5;

number b <- 10;

number c <- a+b;

a <- 10;

print(c);
```

```
---
15.000
---
```

```
test-expressions31.ma
==========
# testing expressions, when /n in strings, will print "/n" and not new line

print("Hello /n World");
---
Hello /n World
---
```

```
test-expressions32.ma
==========
# testing additive expressions, adding number to matrix

number a <- 5;

matrix b <- [1,2,3;4,5,6];

matrix c <- a + b;

print(c[6]);

---
11.000
---
```

```
test-expressions34.ma
==========
# assigning number to matrix

matrix a <- [1,2,3;4,5,6];

number b <- 100;

a[4] <- b;

print(a[2,1]);

---
```

```
100.000
---

test-expressions35.ma
==========
# creating 2 by 3 matrix filled with zeros

matrix a <- [](2,3);

print(a[6]);

---
0.000
---

test-expressions36.ma
==========
# matrix insertion returns the value replaced!

matrix a <- [1,2,3,4;5,6,7,8];

number b <- a[2, 3] <- 19;

print(b);
print(a[2,3]);
---
7.000
19.000
---

test-expressions37.ma
==========
# matrix insertion returns the value replaced!

matrix a <- [1,2,3,4;5,6,7,8];

number b <- a[3] <- 34;

print(b);
print(a[3]);
---
3.000
34.000
---

test-expressions38.ma
==========
# test 2d matrix multiplication

matrix A <- [1,2,9;5,2,1];

matrix B <- [8,1;4,2;4,2];

print(A * B);
---
52.000, 23.000;
52.000, 11.000;
---

test-expressions4.ma
==========
# testing primary expressions

number a <- 5;

number b <- 10;

number c <- (a + b);
```

```
print(c);


---
15.000
---

test-expressions5.ma
==========
# testing primary expressions

string a <- "Success";

void foo (string s)
{
        print(s);
}

foo(a);


---
Success
---

test-expressions6.ma
==========
# testing primary expressions, matrix access

matrix a <- [1, 2, 3; 4, 5, 6];

print(a[1]);


---
1.000
---

test-expressions7.ma
==========
# testing primary expressions, matrix access

number b <- 5;
matrix a <- [1,2,3;4,5,6];

print(a[b]);


---
5.000
---

test-expressions8.ma
==========
# testing unary expressions

number a <- 5;

print (-a);


---
-5.000
---

test-expressions9.ma
==========
```

```
# testing unary operators

matrix a <- [1,2,3;4,5,6];

matrix b <- -a;

print(b[1]);

---
-1.000
---
```

test-functions1.ma

```
==========
# passing an argument to a void function

void foo(string s) {
        print(s);
}

string s <- "Hello World!";
foo(s);

---
Hello World!
---
```

test-functions10.ma

```
==========
number foo() {
        number a <- 5;
        return a;
}

print(foo());

number bar(number a) {
        number x <- foo() + a;
        return x;
}

print(bar(8));

---
5.000
13.000
---
```

test-functions11.ma

```
==========
# return a matrix from a function!

matrix foo() {
    return [](5, 5);
}

matrix a <- foo();
print(a);

---
0.000, 0.000, 0.000, 0.000, 0.000;
0.000, 0.000, 0.000, 0.000, 0.000;
0.000, 0.000, 0.000, 0.000, 0.000;
0.000, 0.000, 0.000, 0.000, 0.000;
0.000, 0.000, 0.000, 0.000, 0.000;
---
```

test-functions12.ma

```
==========
# return a matrix from a function!

matrix foo() {
        matrix b <- [1,2,3,4,5;1,2,3,4,5];
    return b;
}

matrix a <- foo();
print(a);
---
1.000, 2.000, 3.000, 4.000, 5.000;
1.000, 2.000, 3.000, 4.000, 5.000;
---

test-functions13.ma
==========
# return a matrix from a function!

matrix foo() {
        return [1,2,3,4,5;1,2,3,4,5];
}

matrix a <- foo();
print(a);
---
1.000, 2.000, 3.000, 4.000, 5.000;
1.000, 2.000, 3.000, 4.000, 5.000;
---

test-functions14.ma
==========
number foo() {
    return number a <- 6;
}

print(foo());

---
6.000
---

test-functions15.ma
==========
number factorial (number i) {
        if (i = 0) {
                return 1;
        } else {
                return i * factorial(i - 1);
        }
}

print(factorial(5));

print(factorial(10));
---
120.000
3628800.000
---

test-functions2.ma
==========
# function overloading

void foo(string s) {
        print(s);
}
```

```
void foo(string s, string t) {
        print(s);
        print(t);
}

foo("Hello, sir!");
foo("How are you doing?", "Well, I presume?");

---
Hello, sir!
How are you doing?
Well, I presume?
---

test-functions3.ma
==========
# variable scoping in functions

void foo() {
        string s <- "Hello";
        print(s);
}

string s <- "World!";

foo();
print(s);

---
Hello
World!

---

test-functions4.ma
==========
# test return variables are correct for numbers

number foo(number a, number b) {
        return a * b - a + b * 5;
}

number a <- foo(4, 6);

print(a);

---
50.000

---

test-functions5.ma
==========
# global variable modifications are sticky
# variables are passed by value

number a <- 7;

number foo(number x) {
        x <- x * 5;

        print(x);

        return 2;
}

number b <- foo(a);
```

```
print(a);
print(b);

---
35.000
7.000
2.000
---

test-functions6.ma
==========
# matrices are passed by reference/pointer

matrix a <- [](2, 1);

print(a[1]);

void foo(matrix x) {
    x[1] <- 5;
    x[2] <- 18;
}

foo(a);

print(a[1]);
print(a[2]);

---
0.000
5.000
18.000
---

test-functions7.ma
==========
# string concatenation is destructive

string foo(string s) {
        return strcat("Hello ", s);
}

print(foo("World!"));

---
Hello World!

---

test-functions8.ma
==========
# string concatenation is destructive!

string s <- "Hello!";
string t <- " World!";

string x <- strcat(s, t);

print(x);
print(s);
print(t);
x <- strcat(t, s);

print(x);
---
Hello! World!
Hello! World!
World!
World!Hello! World!
```

```
---

test-functions9.ma
==========
# referencing an undeclared variable in a function!
# variable declared after function

void foo(string s) {
    print(a + 5);
    print(s);
}

number a <- 10;
foo("NOPE!");

---
15.000
NOPE!

---

test-overload1.ma
==========
# basic operator overloading

number operator .* (number a, number b) {
        number c <- 1;

        while (b > 0) {
                c <- c * a;
                b <- b - 1;
        }

        return c;
}

number a <- 5 .* 2;
print(a);

---
25.000
---

test-overload2.ma
==========
# unary operators!

string operator - (string s) {
        return strcat("Hello ", s);
}

string a <- "World";
print(-a);
---
Hello World
---

test-overload3.ma
==========
# sequential chaining of overloaded operator

number operator .* (number a, number b) {
        number c <- 0;

        while (b > 1) {
                c <- c + a * a;
                b <- b - 1;
```

```
        }

        return c;
}

number z <- 5 .* 2;
number a <- z .* 8;
print(a);

---
4375.000
---

test-overload4.ma
==========
# mixed variables in overloads

number operator + (string s, number b) {
    print(b);
    print(s);

    return 0;
}

number a <- "Hello World!" + 6;
print(a);
---
6.000
Hello World!
0.000

---

test-overload5.ma
==========
# Overloading overloaded operators

string operator + (string s, number b) {
    print(b);
    print(s);

    return "what?";
}

string operator + (string s, string t) {
        print(strcat(s, t));

        return "lol";
}

string a <- "Meh" + 8.2;
print(a);
string k <- "Hello ";
string l <- "World!";
string s <- k + l;
print(s);
---
8.200
Meh
what?
Hello World!
lol

---

test-overload6.ma
==========
# Overloading overloaded operators
```

```
# This is the same as test-overload5.ma

string operator + (string s, number b) {
    print(b);
    print(s);

    return "what?";
}

string operator + (string s, string t) {
        print(strcat(s, t));

        return "lol";
}

string a <- "Meh" + 8.2;
print(a);
string s <- "Hello " + "World! 2";
print(s);

---
8.200
Meh
what?
Hello World! 2
lol

---

test-overload7.ma
==========
# chain two different overloaded operators
# really the power function, but we're using this symbol because power is defined in stdlib
number operator ./ (number a, number b) {
    number c <- 1;

    while (b > 0) {
        c <- c * a;
        b <- b - 1;
    }

    return c;
}

number operator .* (number a, number b) {
        return a - b * 10;
}

print(5 ./ 10 .* 5);
---
9765575.000

---

test-overload8.ma
==========
# Define a custom operator to test daisy chaining of operators
number operator .* (number a, number b) {
        return a + b * a;
}

number a <- 5 .* 2 .* 8; # 135 expected
print(a);

---
135.000
---
```

test-scope1.ma
==========

```
# testing different function scopes

number first <- 5;

print (first); # global scope, should print 5

number a(){

    return first;

}

number b(){

    return first;
}

print(a());
print(b());
```

```
5.000
5.000
5.000
```

---

test-scope2.ma
==========

```
# testing scope with strings

string s <- "Hello World";

print(s);

string a(){
    print(s);
    return "Success";
}

string b(){
    print(s);
    return "Success";
}

print(a());
print(b());
```

```
Hello World
Hello World
Success
Hello World
Success
```

---

test-scope3.ma
==========

```
# scope passing matrix


matrix m <- [ 1, 2, 3; 8+9, 4-2, 4];
```

```
print(m);  # global scope, should print [1,2,3;17,2,4]

matrix a() {

    return m;
}

matrix b() {
    return m;
}

print(a());
print(b());


---
1.000, 2.000, 3.000;
17.000, 2.000, 4.000;
1.000, 2.000, 3.000;
17.000, 2.000, 4.000;
1.000, 2.000, 3.000;
17.000, 2.000, 4.000;

---

test-scope4.ma
==========
# scope testing

number a <- 1;
number c <- 5;
print(c);

number foo() {
    c <- 10;
    print(c);

    number a <-10;

    number p <- 99;

    return p;
}
print (c);
print (a);
c <- foo();
print(c);

---
5.000
5.000
1.000
10.000
99.000

---

test-scope5.ma
==========
# scope testing

string s;

s <- "Outside";


string foo () {
```

```
        string s;
        s <- "Inside";
        return s;


}
print(foo());
print(s);
---
Inside
Outside

---

test-statements1.ma
==========
# test an if-else statement

number foo(number a) {
    if (a = 1) {
        print(a);
    } else {
        print(a + 1);
    }

    return 0;
}

print(foo(1));

---
1.000
0.000

---

test-statements11.ma
==========
# nested while loop

number b <- 10;

while (b != 5) {
        b <- 5;
}

print(b);

while (b != 11) {
        number c <- 1;
        while (c < 111) {
                c <- c * b;
                print(c);
                b <- b + 1;
        }
        print(b);
}
---
5.000
5.000
30.000
210.000
8.000
8.000
72.000
720.000
11.000
---
```

```
test-statements12.ma
==========
# compare numerals

if (1 = 1) {
        print("1 is equal to 1.");
}
---
1 is equal to 1.
---

test-statements13.ma
==========
# this program should not loop.

number a <- 10;

if (a = 10) {
        print("I'm okay!");
} else {
        while (1 = 1) {
                print("HELP ME!");
        }
}
---
I'm okay!
---

test-statements14.ma
==========
# nested if statements

number b <- 99;

if (b >= 92) {
        if (b >= 93) {
                if (b >= 95) {
                        if (b >= 99) {
                                print(b);
                        }
                }
        }
}

if (b < 99) {
        print("Nope.");
} else {
        if (b < 94) {
                print("Nope.");
        } else {
                print(b);
        }
}
---
99.000
99.000
---

test-statements15.ma
==========
# for loop test

matrix a <- [1,2,3,4,5,6,7,8];
number b <- 0;

for (b in a) {
        print(b);
```

```
}

---
1.000
2.000
3.000
4.000
5.000
6.000
7.000
8.000
---

test-statements16.ma
==========
# var declaration in for loop

matrix a <- [1;2;3;4;5];

for (number b in a) {
        print(b);
}
---
1.000
2.000
3.000
4.000
5.000
---

test-statements2.ma
==========
# greater than test

number a <- 8;

if (a > 6) {
        print("a is greater than 6.");
}
---
a is greater than 6.

---

test-statements3.ma
==========
# less than test

number a <- 8;

if (a < 10) {
        print("a is less than 10.");
}
---
a is less than 10.

---

test-statements4.ma
==========
# testing both printing functions

print("Hello World");
printnl("Hello ");
print("World");

---
Hello World
```

```
Hello World

---

test-statements5.ma
==========
# test a while statement

number a <- 0;

while (a <= 10) {
        print(a);
        a <- a + 1;
}
---
0.000
1.000
2.000
3.000
4.000
5.000
6.000
7.000
8.000
9.000
10.000
---

test-statements6.ma
==========
# test LET

number b <- 5;

if (b <= 5) {
        print("Yep.");
}
---
Yep.
---

test-statements7.ma
==========
# test GET

number b <- 5;

if (b >= 5) {
        print("Yep.");
}

b <- 6;

if (b >= 5) {
        print("Yep.");
}
---
Yep.
Yep.
---

test-statements8.ma
==========
# compare two numbers

number a <- 6;
number b <- 6;
```

```
if (a = b) {
        print("a and b are equal.");
}

print("Success!");

---
a and b are equal.
Success!
---

test-statements9.ma
==========
# if else if else if else statements

number b <- 99;

if (b = 88) {
        print(b);
} else {
        if (b = 77) {
                print(b);
        } else {
                if (b = 99) {
                        print(b);
                }
        }
}
---
99.000
---
```

9.2 Source Files

../src/scanner.mll

```
{
open Parser

(* authors: William Hom and Joseph Isaac Baker *)
}

rule token = parse
(* whitespace *)
[' ' '\t' '\r' '\n'] { token lexbuf }
(* comments *)
| '#' { comment lexbuf }
(* arithmetic operators *)
| '+'       { PLUS     }   | '-'       { MINUS   }
| '*'       { TIMES    }   | '/'       { DIVIDE  }
| '%'       { MOD      }   | '\''      { TRAN    }
(* relational operators *)
| '='       { EQUAL    }  | "!="       { NOTEQUAL}
| '<'       { LESS     }  | "<="       { LEQ     }
| '>'       { GREATER  }  | ">="       { GEQ     }
(* logical operators *)
| '!'       { NOT      }   | '&'       { AND     }
| '|'       { OR       }
(* Assignment *)
| "<-"      { ASSIGN   }
(* bracketing *)
| '{'       { LCURLY   }   | '}'          { RCURLY  }
| '('       { LPAREN   }   | ')'          { RPAREN  }
| '['       { LBRACK   }   | ']'          { RBRACK  }
(* expression markers *)
| ';'       { SEMI     }   | '.'          { DOT     }
| ','       { COMMA    }   | '"'␣␣␣␣␣␣␣␣␣{␣DQUOTE␣␣}
```

```
(* data types for variables anf functions *)
| "matrix" { MAT_TYP }   | "void"     { VOID_TYP }
| "number" { NUM_TYP }   | "string"   { STR_TYP }
(* block expressions *)
| "if"     { IF }       | "else"     { ELSE }
| "while"  { WHILE }    | "return"   { RETURN }
| "for"    { FOR }      | "in"       { IN }
(* overloadable operators *)
| '^'      { POW }      | ".*"       { DOTTIMES }
| "./"     { DOTDIVIDE } | "operator" { OPERATOR }
(* numeric types *)
| ['0'-'9']+ as num { INT(int_of_string num) }
| ['0'-'9']+'.'['0'-'9']+ as num { FLOAT(float_of_string num) }
(* strings *)
| '"'[^'\n' '"']+'"' as str { STRING(str) }
(* identifiers *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lit { ID(lit) }
(* meh *)
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
(* end-of-file *)
| eof { EOF }
(* comments *)
and comment = parse
"\n" { token lexbuf }
| _ { comment lexbuf } (* eat it all! *)
```

../src/ast.ml

```
(* author: Joseph Isaac Baker *)

type b_op = Add | Sub | Mul | Div | Mod | Pow | DotMul | DotDiv
          | Equal | Neq | Less | Leq | Greater | Geq
          | And | Or
type u_op = Not | Neg | Tran

type op = Bop of b_op | Uop of u_op

type typ =
  NumberTyp
| StringTyp
| MatrixTyp
| VoidTyp

type num_typ = IntTyp | FloatTyp

type binding = typ * string

type lvalue =
  VarDecl     of binding
| IdAsn       of string
| MatCellAsn  of string * expr * expr

and expr =
  Number        of num_typ * int * float
| Str           of string
| Id            of string
| MatAcc        of string * expr * expr
| MatInit       of expr list list
| MatEmptyInit  of expr * expr
| Binop         of expr * b_op * expr
| Unop          of u_op * expr
| Assign        of lvalue * expr
| Func          of string * expr list
| Noexpr

type stmt =
  Block   of stmt list
| Expr    of expr
```

```
| VDecl   of binding
| Return  of expr
| If      of expr * stmt * stmt
| For     of lvalue * expr * stmt * string
| While   of expr * stmt

type func_decl = {
    fdtype  : typ;
    fname   : string;
    fparams : binding list;
    fbody   : stmt;
  }

type oper_decl = {
    opdtype   : typ;
    operator  : op;
    opparams  : binding list;
    opbody    : stmt;
  }

type array = {
    array   : typ list;
    dtype   : typ;
    rows    : int;
    columns : int;
  }

type global_stmt =
  Stmt     of stmt
| FuncDecl of func_decl
| OperDecl of oper_decl

type program = global_stmt list

(* Complete dump from MicroC.
 * Need to retrofit it to our purposes
 * Pretty-printing functions
 *)
let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mul -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Pow -> "^"
  | DotMul -> ".*"
  | DotDiv -> "./"
  | Equal -> "="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"
  | Tran -> "'"

let name_of_op = function
    Bop(o) -> (match o with
                | Add -> "add"
                | Sub -> "sub"
                | Mul -> "mul"
                | Div -> "div"
                | Mod -> "mod"
```

```
                | Pow -> "pow"
                | DotMul -> "dotmul"
                | DotDiv -> "dotdiv"
                | Equal -> "equal"
                | Neq -> "notequal"
                | Less -> "less"
                | Leq -> "leq"
                | Greater -> "greater"
                | Geq -> "geq"
                | And -> "and"
                | Or -> "or")
  | Uop(o) -> (match o with
                | Neg -> "neg"
                | Not -> "not"
                | Tran -> "tran")

let rec string_of_typ = function
    NumberTyp -> "number"
  | MatrixTyp -> "matrix"
  | StringTyp -> "string"
  | VoidTyp   -> "void"

let string_of_vdecl (t, id) = string_of_typ t ^ "␣" ^ id ^ ";␣"

let rec string_of_lvalue = function
    VarDecl(b) -> string_of_typ (fst b) ^ "␣" ^ (snd b)
  | IdAsn(s) -> s
  | MatCellAsn(s, e1, e2) -> s ^ "[" ^ string_of_expr e1 ^ ",␣" ^ string_of_expr e2 ^ "]"

and string_of_expr = function
    Number(t, i, f) -> if t = IntTyp then string_of_int i else string_of_float f
  | Str(l) -> l
  | Id(s) -> s
  | MatAcc(s, e1, e2) -> s ^ "[" ^ string_of_expr e1 ^ ",␣" ^ string_of_expr e2 ^ "]"
  | MatInit(ell) -> "[" ^ string_of_expr_list_list ell ^ "]"
  | MatEmptyInit(e1, e2) -> "[␣](" ^ string_of_expr e1 ^ ",␣" ^ string_of_expr e2 ^ ")"
  | Binop(e1, o, e2) ->
      "(" ^ string_of_expr e1 ^ "␣" ^ string_of_op o ^ "␣" ^ string_of_expr e2 ^ ")"
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> string_of_lvalue v ^ "␣<-␣" ^ string_of_expr e
  | Func(f, el) ->
      f ^ "(" ^ String.concat ",␣" (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

and string_of_expr_list e_list = String.concat ",␣" (List.map string_of_expr e_list)
and string_of_expr_list_list e_list_list = String.concat ";␣" (List.map string_of_expr_list e_list_list)

let rec string_of_stmt p = function
    Block(stmts) ->
      p ^ "{\n" ^ String.concat "" (List.map (string_of_stmt (p ^ "\t")) stmts) ^ p ^ "}\n"
  | Expr(expr) -> p ^ string_of_expr expr ^ ";\n";
  | VDecl((t, i)) -> p ^ string_of_typ t ^ "␣" ^ i ^ ";\n";
  | Return(expr) -> p ^ "return␣" ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> p ^ "if␣(" ^ string_of_expr e ^ ")\n" ^ string_of_stmt p s
  | If(e, s1, s2) ->  p ^ "if␣(" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt p s1 ^ p ^ "else\n" ^ string_of_stmt p s2
  | For(lv, e, s, _) ->
      p ^ "for␣(" ^ string_of_lvalue lv ^ "␣in␣" ^ string_of_expr e ^ ")␣" ^ string_of_stmt p s
  | While(e, s) -> p ^ "while␣(" ^ string_of_expr e ^ ")\n" ^ string_of_stmt p s

let string_of_fdecl fdecl =
  string_of_typ fdecl.fdtype ^ "␣" ^
  fdecl.fname ^ "(" ^ String.concat ",␣" (List.map (fun (t, n) -> string_of_typ t ^ "␣" ^ n) fdecl.fparams) ^ ")\
  string_of_stmt "" fdecl.fbody

let string_of_odecl odecl =
  string_of_typ odecl.opdtype ^ "␣operator␣" ^
  name_of_op odecl.operator ^ "(" ^ String.concat ",␣" (List.map (fun (t, n) -> string_of_typ t ^ "␣" ^ n) odecl.
```

```
  string_of_stmt "" odecl.opbody

let string_of_glbl_stmt = function
  Stmt(st) -> string_of_stmt "" st
| FuncDecl(f) -> string_of_fdecl f
| OperDecl(o) -> string_of_odecl o

let string_of_program (glbl_stmts) =
  String.concat "" (List.map string_of_glbl_stmt glbl_stmts)
```

../src/parser.mly

```
%{ open Ast
(* authors: William Hom and Joseph Isaac Baker *)

let for_counter =
  let count = ref (0) in
  fun () -> incr count; string_of_int !count;;

  %}

%token PLUS MINUS TIMES DIVIDE MOD POW DOTTIMES DOTDIVIDE
%token EQUAL NOTEQUAL LESS LEQ GREATER GEQ
%token AND OR
%token NOT NEG TRAN
%token ASSIGN
%token LCURLY LPAREN LBRACK RCURLY RPAREN RBRACK
%token SEMI COMMA DOT SQUOTE DQUOTE
%token VOID_TYP STR_TYP NUM_TYP MAT_TYP
%token IF ELSE FOR IN WHILE RETURN
%token OPERATOR

%token <int> INT
%token <float> FLOAT
%token <string> STRING
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc EQUAL NOTEQUAL LESS LEQ GREATER GEQ
%right ASSIGN
%left OR AND
%left PLUS MINUS
%left TIMES DIVIDE MOD DOTTIMES DOTDIVIDE
%right NOT NEG POW
%left TRAN

%start program
%type <Ast.program> program

%%

program:
  global_stmts EOF { List.rev $1 }

global_stmts:
  /* nothing */           { [] }
| global_stmts stmt       { Stmt($2) :: $1 }
| global_stmts func_decl  { FuncDecl($2) :: $1 }
| global_stmts oper_decl  { OperDecl($2) :: $1 }

func_decl:
  typ ID LPAREN param_list RPAREN stmt_block
  {{
    fdtype    = $1;
    fname     = $2;
    fparams   = $4;
```

```
    fbody     = $6
  }}

oper_decl:
  typ OPERATOR oper_symbol LPAREN param_list RPAREN stmt_block
  {{ opdtype = $1; operator = $3; opparams = $5; opbody = $7 }}

oper_symbol:
  PLUS        { Bop(Add)     }
| MINUS       { Bop(Sub)     }
| TIMES       { Bop(Mul)     }
| DIVIDE      { Bop(Div)     }
| POW         { Bop(Pow)     }
| MOD         { Bop(Mod)     }
| DOTTIMES    { Bop(DotMul)  }
| DOTDIVIDE   { Bop(DotDiv)  }
| EQUAL       { Bop(Equal)   }
| NOTEQUAL    { Bop(Neq)     }
| LESS        { Bop(Less)    }
| LEQ         { Bop(Leq)     }
| GREATER     { Bop(Greater) }
| GEQ         { Bop(Geq)     }
| AND         { Bop(And)     }
| OR          { Bop(Or)      }
| NOT         { Uop(Not)     }
| TRAN        { Uop(Tran)    }

param_list:
  /* nothing */          { []          }
| param                  { List.rev $1 }

param:
  typ ID                 { [($1, $2)]   }
| param COMMA typ ID      { ($3, $4)::$1 }

typ:
  VOID_TYP { VoidTyp       }
| STR_TYP  { StringTyp     }
| NUM_TYP  { NumberTyp     }
| MAT_TYP  { MatrixTyp     }

stmt_list:
    /* nothing */  { []       }
  | stmt_list stmt { $2 :: $1 }

stmt_block:
| LCURLY stmt_list RCURLY                            { Block(List.rev $2)    }

stmt:
  expr SEMI                                          { Expr($1)              }
| typ ID SEMI                                        { VDecl(($1, $2))       }
| RETURN SEMI                                        { Return Noexpr         }
| RETURN expr SEMI                                   { Return $2             }
| IF LPAREN expr RPAREN stmt_block %prec NOELSE      { If($3, $5, Block([])) }
| IF LPAREN expr RPAREN stmt_block ELSE stmt_block   { If($3, $5, $7)        }
| FOR LPAREN lvalue IN expr RPAREN stmt_block        { For($3, $5, $7, for_counter()) }
| WHILE LPAREN expr RPAREN stmt_block                { While($3, $5)         }

/* comment on the FOR statement: */
/* you can't do this: for (int i <- 5 to 10) {} */
/* it has to be int i; for (i <- 5 to 10) {} */
/* due to the way the expressions are set up. */
/* can remedy this changing 'ID ASSIGN expr' to 'expr' */
/* it can lead to something nonsensical like for (3 + 3 to 8) {} */
/* but it's how microC does it. */

lvalue:
  typ ID                          { VarDecl(($1, $2))          }
```

```
| ID                                { IdAsn($1)                    }
| ID LBRACK expr RBRACK             { MatCellAsn($1, $3, Number(IntTyp, 0, 0.0)) }
| ID LBRACK expr COMMA expr RBRACK  { MatCellAsn($1, $3, $5)      }

expr:
/* data types */
  INT     { Number(IntTyp, $1, 0.0)      }
| FLOAT   { Number(FloatTyp, 0, $1)      }
| STRING  { Str($1)                      }
| ID      { Id($1)                       }
/* Matrix expressions */
| ID LBRACK expr RBRACK                       { MatAcc($1, $3, Number(IntTyp, 0, 0.0)) }
| ID LBRACK expr COMMA expr RBRACK            { MatAcc($1, $3, $5)     }
| LBRACK matrix_init_vars RBRACK              { MatInit($2)            }
| LBRACK RBRACK LPAREN expr COMMA expr RPAREN { MatEmptyInit($4, $6)   }
/* arithmetic expressions */
| expr PLUS expr       { Binop($1, Add, $3)      }
| expr MINUS expr      { Binop($1, Sub, $3)      }
| expr TIMES expr      { Binop($1, Mul, $3)      }
| expr DIVIDE expr     { Binop($1, Div, $3)      }
| expr MOD expr        { Binop($1, Mod, $3)      }
| expr POW expr        { Binop($1, Pow, $3)      }
| expr DOTTIMES expr   { Binop($1, DotMul, $3)   }
| expr DOTDIVIDE expr  { Binop($1, DotDiv, $3)   }
| MINUS expr %prec NEG { Unop(Neg, $2)           }
| expr TRAN            { Unop(Tran, $1)          }
/* Relational expressions */
| expr EQUAL expr      { Binop($1, Equal, $3)    }
| expr NOTEQUAL expr   { Binop($1, Neq, $3)      }
| expr LESS expr       { Binop($1, Less, $3)     }
| expr LEQ expr        { Binop($1, Leq, $3)      }
| expr GREATER expr    { Binop($1, Greater, $3)  }
| expr GEQ expr        { Binop($1, Geq, $3)      }
/* logical expressions */
| expr AND expr        { Binop($1, And, $3)      }
| expr OR expr         { Binop($1, Or, $3)       }
| NOT expr             { Unop(Not, $2)           }
/* assignment expression */
| lvalue ASSIGN expr   { Assign($1, $3)          }
/* function call expression */
| ID LPAREN args_opt RPAREN { Func($1, $3)       }
/* parenthetical expressions */
| LPAREN expr RPAREN   { $2                      }

args_opt:
  /* nothing */ { []          }
| args_list     { List.rev $1 }

args_list:
  expr                 { [$1]   }
| args_list COMMA expr { $3::$1 }

matrix_init_vars:
| rows { List.rev $1 }

rows:
  columns            { [ List.rev $1 ]   }
| rows SEMI columns  { (List.rev $3)::$1 }

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

../src/evaluator.ml

```
(* Semantic checking for the MicroC compiler *)

(*
```

```
system functions
prints (string a)
printf (float a)
print (int a)
str_cat(string a, string b)
*)

(* author: Joseph Isaac Baker *)

open Ast
open Sast

module StringMap = Map.Make(String)
module StringSet = Set.Make(String)
type func_id = {  func_name: string;
                  func_params: typ list; }
module FuncMap = Map.Make(struct type t = func_id let compare = compare end)
module FuncSet = Set.Make(struct type t = func_id let compare = compare end)
type oper_id = {  oper_symbol: op;
                  oper_params: typ list;}
module OperMap = Map.Make(struct type t = oper_id let compare = compare end)

module Transforms =
  struct
    let func_id_of_func_decl func_decl = {
      func_name = func_decl.fname;
      func_params = (List.map fst func_decl.fparams); }

    let string_of_func_id func_id =
      let param_string params = String.concat "," (List.map string_of_typ params) in
      func_id.func_name ^ "(" ^ param_string func_id.func_params ^ ")"

    let oper_id_of_oper_decl oper_decl = {
      oper_symbol = oper_decl.operator;
      oper_params = (List.map fst oper_decl.opparams); }

    let func_string_of_oper_id oper_id =
      let param_string params = String.concat "," (List.map string_of_typ params) in
      name_of_op oper_id.oper_symbol ^ "(" ^ param_string oper_id.oper_params ^ ")"

    let string_of_oper_id oper_id =
      let binary_oper_string binop params = (string_of_typ (List.nth params 0)) ^ "␣" ^
                                            string_of_op binop ^ "␣" ^ (string_of_typ (List.nth params 1)) in
      let pre_unary_oper_string unop param = string_of_uop unop ^ string_of_typ param in
      let post_unary_oper_string unop param = string_of_typ param ^ string_of_uop unop in
      match oper_id.oper_symbol with
          Bop(o) -> binary_oper_string o oper_id.oper_params
        | Uop(o) -> (match o with
                      | Neg | Not -> pre_unary_oper_string o (List.hd oper_id.oper_params)
                      | Tran      -> post_unary_oper_string o (List.hd oper_id.oper_params))

    let operator_overload_name op_symbol = "_" ^ (name_of_op op_symbol)

    let func_of_oper oper = {
      fdtype  = oper.opdtype;
      fname   = operator_overload_name oper.operator;
      fparams = oper.opparams;
      fbody   = oper.opbody; }
  end
open Transforms

module AppState =
  struct
    type app_state = {  vars: typ StringMap.t;
                        funcs: func_decl FuncMap.t;
                        func_names: StringSet.t;
                        func_bodies_checked: FuncSet.t;
                        func_locals: typ StringMap.t FuncMap.t;
```

```
                  opers: oper_decl OperMap.t;
                  decl_allowed: bool;
                  local_context: func_id option; }
let empty = { vars = StringMap.empty;
              funcs = FuncMap.empty;
              func_names = StringSet.empty;
              func_bodies_checked = FuncSet.empty;
              func_locals = FuncMap.empty;
              opers = OperMap.empty;
              decl_allowed = true;
              local_context = None; }
(* Adds the given variable to the app state and returns the new app state *)
let add_global_var prev_state var_id var_typ =
  { prev_state with vars = StringMap.add var_id var_typ prev_state.vars; }
(* Adds a local variable for a function *)
let add_local_for_func prev_state func_id var_id var_typ =
  let new_locals_map = StringMap.add var_id var_typ (try FuncMap.find func_id prev_state.func_locals
                                                      with Not_found -> raise (Failure
                                                       ("add_local_for_func: Can't find function " ^
                                                      string_of_func_id func_id))) in
  { prev_state with func_locals = FuncMap.add func_id new_locals_map prev_state.func_locals; }
let get_local_var state func_id var_id =  StringMap.find var_id (FuncMap.find func_id state.func_locals)
(* Adds variable to the proper scope depending on the prev app_state *)
let add_var prev_state var_id var_typ =
  match prev_state.local_context with
    | Some(f) -> add_local_for_func prev_state f var_id var_typ
    | None    -> add_global_var prev_state var_id var_typ
(* Adds the given function to the app state and returns the new app state *)
let add_func prev_state func_sig func_decl = { prev_state with
  funcs       = FuncMap.add func_sig func_decl prev_state.funcs;
  func_names  = StringSet.add func_sig.func_name prev_state.func_names;
  func_locals = FuncMap.add func_sig StringMap.empty prev_state.func_locals; }
(* Adds the given operator overload to the app state and returns the new app state *)
let add_oper prev_state oper_id oper_decl =
  { prev_state with opers = OperMap.add oper_id oper_decl prev_state.opers; }
(* Adds the given func name to the func names set and returns the new app state *)
let add_func_name prev_state func_name =
  { prev_state with func_names = StringSet.add func_name prev_state.func_names; }
(* Adds the given func sig to the bodies checked let and returns the new app state *)
let add_func_body_checked prev_state func_sig =
  { prev_state with func_bodies_checked = FuncSet.add func_sig prev_state.func_bodies_checked; }
(* Returns an appstate based on prevstate with declarations prohibited *)
let prohibit_decl prev_state = { prev_state with decl_allowed = false; }
let enable_decl prev_state = { prev_state with decl_allowed = true; }
(* Set whether var declarations should be global or local *)
let set_local_context prev_state func_id = { prev_state with local_context = func_id; }
let print state =
  let param_string params = String.concat "_" (List.map string_of_typ params) in
  let string_of_func_id f = f.func_name ^ "(" ^ param_string f.func_params ^ ")" in
  let func_string_of_oper_id o = name_of_op o.oper_symbol ^ "(" ^ param_string o.oper_params ^ ")" in
  let var_string = StringMap.fold (fun k v a -> a ^ string_of_typ v ^ " " ^ k ^ "; " ) state.vars "" in
  let local_context_string =
    match state.local_context with
      | Some(f) -> string_of_func_id f
      | None    -> "None"
  in
  let func_string = FuncMap.fold (fun k _ a -> a ^ string_of_func_id k ^ "; ") state.funcs "" in
  let oper_string = OperMap.fold (fun k _ a -> a ^ func_string_of_oper_id k ^ "; ") state.opers "" in
  let locals_string =
    let local_vars_string locals_map =
      StringMap.fold (fun k v a -> a ^ string_of_typ v ^ " " ^ k ^ ", ") locals_map "" in
  FuncMap.fold (fun k lcls a -> a ^ "\t" ^ string_of_func_id k ^ ": " ^
                                local_vars_string lcls ^ "\n") state.func_locals "" in
  let checked_string =
    FuncSet.fold (fun f a -> a ^ string_of_func_id f ^ "; " ) state.func_bodies_checked "" in

  "Vars: " ^ var_string ^ "\n" ^ "Funcs: " ^ func_string ^ "\n" ^ "local_context: " ^
    local_context_string ^ "\n" ^
```

```
      "Declarations␣Allowed:␣" ^ string_of_bool state.decl_allowed ^ "\n" ^
      "Function␣Locals:␣\n" ^ locals_string  ^ "Checked␣Functions:␣" ^ checked_string ^ "\n" ^
      "Known␣Operators:␣\n" ^ oper_string
  end
open AppState

module ExceptionMessages =
  struct
    (* Exception statement functions *)
    let except_void_var n               = "Illegal␣void␣variable␣" ^ n ^ "!"
    let except_dupe_var n               = "Duplicate␣variable␣" ^ n ^ "␣for␣this␣scope!"
    let except_dupe_func n              = "Duplicate␣function␣" ^ n ^ "!"
    let except_dupe_oper n              = "Duplicate␣operator␣definition␣" ^ n ^ "!"
    let except_undeclared_var n         = "Use␣of␣undeclared␣variable␣" ^ n ^ "!"
    let except_undeclared_func n        = "Use␣of␣undeclared␣function␣" ^ n ^ "!"
    let except_unknown_operator n       = "Use␣of␣unknown␣operator␣" ^ n ^ "!"
    let except_declaration_prohibited n = "Declaration␣of␣variable␣" ^ n ^ "␣prohibited␣in␣this␣context!"
    let except_type_binary_unknown n    = "Don't␣recognize␣types␣for␣arithemetic␣operator␣" ^ n ^ "!"
    let except_type_unary_unknown n     = "Don't␣recognize␣types␣for␣unary␣operator␣" ^ n ^ "!"
    let except_type_req_type t n        = t ^ "␣type␣expected␣from␣" ^ n ^ "!"
    let except_type_mat_init t          = "Only␣number␣type␣allowed␣in␣matrix␣init␣block.␣" ^ t ^ "␣used!"
    let except_jagged_mat t             = "Matrix␣initialization␣cannot␣be␣jagged!␣" ^ t
    let except_stmt_after_return        = "No␣statements␣are␣allowed␣after␣return␣inside␣a␣block!"
  end
open ExceptionMessages

module TypeCh =
  struct
    let check_types global_vars func_decls app_state =
      let global_var_map = List.fold_left (fun map (t, i) ->
        StringMap.add i t map) StringMap.empty global_vars in
      let func_map = List.fold_left (fun map func_decl ->
        StringMap.add func_decl.cname func_decl map) StringMap.empty func_decls in

      (* Confirm no global, local, or parameter variables are marked as 'void' type *)
      let check_void_vars var_list = List.iter (fun (t, i) -> if t =
        VoidTyp then raise (Failure (except_void_var i))) var_list in
      check_void_vars global_vars;
      List.iter (fun func_decl ->
        check_void_vars func_decl.clocal_vars; check_void_vars func_decl.cparams ) func_decls;


      let confirm_typ t target_typ e =
        if t != target_typ
        then raise (Failure (except_type_req_type (string_of_typ target_typ) (string_of_expr e)))
      in
      (* Performs various type checks on the expression and returns the expression's type *)
      let rec check_expr_typ avail_vars = function
          Number(_,_,_)       ->  NumberTyp
        | Str(_)              ->  StringTyp
        | Noexpr              ->  VoidTyp
        | Id(i)               ->  (try StringMap.find i avail_vars
                                   with Not_found -> raise (Failure ("Can't␣find␣" ^ i)))
        (* Ensure all the expressions have the same type*)
        | MatInit(ell)        ->  check_expr_list_list_typ avail_vars ell NumberTyp;
                                  MatrixTyp
        (* Ensure both expressions are numbers *)
        | MatEmptyInit(e1,e2) ->  confirm_typ (check_expr_typ avail_vars e1) NumberTyp e1;
                                  confirm_typ (check_expr_typ avail_vars e2) NumberTyp e2;
                                  MatrixTyp;
        (* Ensure both expressions are numbers *)
        | MatAcc(_, e1, e2)   ->  confirm_typ (check_expr_typ avail_vars e1) NumberTyp e1;
                                  confirm_typ (check_expr_typ avail_vars e2) NumberTyp e2;
                                  NumberTyp;
        (* Simply getting their type determines if we recognize the types of the operands *)
        | Binop(e1,op,e2)     ->  let oper_id = { oper_symbol=Bop(op);
                                                  oper_params=[ check_expr_typ avail_vars e1;
                                                                check_expr_typ avail_vars e2]} in
```

```
                            (try (OperMap.find oper_id app_state.opers).opdtype
                             with Not_found ->
                             raise (Failure ("check_expr_typ: Can't find operator " ^
                                             string_of_oper_id oper_id)))
  (* Simply getting its type determines if we recognize the types of the operands *)
  | Unop(op, e)         ->  let oper_id = { oper_symbol=Uop(op);
                                            oper_params=[check_expr_typ avail_vars e;]} in
                            (try (OperMap.find oper_id app_state.opers).opdtype
                            with Not_found ->
                            raise (Failure ("check_expr_typ: Can't find operator " ^
                                            string_of_oper_id oper_id)))
  (* Confirm that the result of the assignment expression matches what is being assigned to *)
  | Assign(l, e)        ->  check_assignment avail_vars l e;
  (* Simply getting the function's type determines if the parameters are for a valid function *)
  | Func(s, es)         ->  ignore(List.map (check_expr_typ avail_vars) es);
                            try (StringMap.find s func_map).cdtype
                            with Not_found -> raise (Failure ("check_expr_typ: Can't find " ^ s))

and check_expr_list_list_typ avail_vars ell target_typ=
  let check_expr_list_typ el =
    List.iter (fun e -> confirm_typ (check_expr_typ avail_vars e) target_typ e) el
  in
  List.iter check_expr_list_typ ell
and l_value_target_typ avail_vars = function
    VarDecl (t, _)      ->  t
  | IdAsn(i)            ->  (try StringMap.find i avail_vars
                             with Not_found -> raise (Failure ("check_assignment: Can't find " ^ i)))
  | MatCellAsn(_,e1,e2) ->  confirm_typ (check_expr_typ avail_vars e1) NumberTyp e1;
                            confirm_typ (check_expr_typ avail_vars e2) NumberTyp e2;
                            NumberTyp;
and check_assignment avail_vars lv expr =
  let target_typ = l_value_target_typ avail_vars lv in
  let expr_typ = check_expr_typ avail_vars expr in
  confirm_typ expr_typ target_typ expr; expr_typ
in

let confirm_return_block_end stmts =
  let len = List.length stmts in
  let check i = function
      Return(e) -> if (i != (len - 1)) then raise (Failure except_stmt_after_return)
    | _         -> ()
  in
  List.iteri check stmts
in

let rec check_stmt func_target_typ avail_vars = function
    Block(stmts)      ->  confirm_return_block_end stmts;
                          List.iter (check_stmt func_target_typ avail_vars) stmts
  | Expr(e)           ->  ignore(check_expr_typ avail_vars e)
  | Return(e)         ->  confirm_typ (check_expr_typ avail_vars e) func_target_typ e
  | If(e, s1, s2)     ->  confirm_typ (check_expr_typ avail_vars e) NumberTyp e;
                          check_stmt func_target_typ avail_vars s1;
                          check_stmt func_target_typ avail_vars s2;
  | While(e, s)       ->  confirm_typ (check_expr_typ avail_vars e) NumberTyp e;
                          check_stmt func_target_typ avail_vars s;
  | For(lv,e,s,_)     ->  confirm_typ (l_value_target_typ avail_vars lv) NumberTyp (Assign(lv, Noexpr));
                          confirm_typ (check_expr_typ avail_vars e) MatrixTyp e;
                          check_stmt func_target_typ avail_vars s;
  | _                 -> ()
in

let check_function func_decl =
  let avail_vars = List.fold_left (fun map (t, i) -> StringMap.add i t map)
                     global_var_map func_decl.cparams in
  let avail_vars = List.fold_left (fun map (t, i) -> StringMap.add i t map)
                     avail_vars func_decl.clocal_vars in
  check_stmt func_decl.cdtype avail_vars func_decl.cbody
in
```

```
      List.iter check_function func_decls
  end

module DeclarationCh =
  struct
    let add_system_func_decls app_state =
      let stub_func name param_bindings return_typ =
        { fdtype = return_typ; fname = name; fparams = param_bindings; fbody = Block([]); } in
      let add_func prev_state (name, params, return_typ) =
        let func_decl = stub_func name params return_typ in
        AppState.add_func prev_state (func_id_of_func_decl func_decl) func_decl  in
      let system_funcs =
        [("main",  [], VoidTyp);
        ("print",  [ (NumberTyp, "n") ], VoidTyp);
        ("print",  [ (StringTyp, "s") ], VoidTyp);
        ("printnl",  [ (NumberTyp, "n") ], VoidTyp);
        ("printnl",  [ (StringTyp, "s") ], VoidTyp);
        ("shape",  [ (MatrixTyp, "m") ], MatrixTyp);
        ("len",    [ (MatrixTyp, "m") ], NumberTyp);
        ("strcat", [ (StringTyp, "s"); (StringTyp, "s2"); ], StringTyp); ]
      in List.fold_left add_func app_state system_funcs

    let add_system_oper_decls app_state =
      let stub_oper op param_bindings return_typ =
      { opdtype = return_typ; operator = op; opparams = param_bindings; opbody = Block([]); } in
      let add_oper prev_state (op, params, return_typ) =
        let oper_decl = stub_oper op params return_typ in
        AppState.add_oper prev_state (oper_id_of_oper_decl oper_decl) oper_decl  in
      let system_opers =
        [(Bop(Add),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Sub),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Mul),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Div),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Mod),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Equal),   [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Neq),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Less),    [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Leq),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Greater), [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Geq),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(And),     [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Bop(Or),      [(NumberTyp, "n"); NumberTyp, "n2"], NumberTyp);
         (Uop(Not),     [(NumberTyp, "n")], NumberTyp);
         (Uop(Neg),     [(NumberTyp, "n")], NumberTyp); ]
      in List.fold_left add_oper app_state system_opers

    let check_func_param_types app_state func_id =
      if FuncMap.mem func_id app_state.funcs then app_state
      else raise (Failure (except_undeclared_func (string_of_func_id func_id)))

    (*  Focused search of the expression types search just for the types of the parameters
        of the function in question *)
    let rec func_id_of_name_params app_state func_name func_params =
      let get_var_typ var_name =
        let is_global_var = StringMap.mem var_name app_state.vars in
        let is_local_var =
          match app_state.local_context with
            | Some(f) ->  (try StringMap.mem var_name (FuncMap.find f app_state.func_locals)
                           with Not_found -> raise (Failure ("get_var_typ:␣Can't␣find␣local␣context")))
            | None    -> false
        in
        if is_local_var then
          ( let func_context = let (Some f) = app_state.local_context in f in
            try StringMap.find var_name ( try FuncMap.find func_context app_state.func_locals
                                          with Not_found ->
                                          raise (Failure ("get_var_typ:␣Can't␣find␣local␣context")))
            with Not_found -> raise (Failure ("get_var_typ:␣Can't␣find␣variable␣in␣local␣context")))
```

```
    else
      if is_global_var then try StringMap.find var_name app_state.vars
                           with Not_found ->
                           raise (Failure ("get_var_typ: Can't find global variable"))
      else raise (Failure (except_undeclared_var var_name))
  in
  let get_oper_typ oper params =
    let oper_id = { oper_symbol=oper; oper_params=params } in
    if OperMap.mem oper_id app_state.opers
    then try (OperMap.find oper_id app_state.opers).opdtype
         with Not_found -> raise (Failure ("get_oper_typ: Can't find operator " ^
                                           string_of_oper_id oper_id))
    else raise (Failure (except_unknown_operator (string_of_oper_id oper_id)))
  in
  let rec get_expr_typ = function
    (* cases not using a variable or function return simple type *)
      Number(_,_,_)       ->  NumberTyp
    | Str(_)              ->  StringTyp
    | Noexpr              ->  VoidTyp
    (* id has the type that it was declared as *)
    | Id(i)               ->  get_var_typ i
    (* recurse into the first expression and use its type as the inner type for the matrix *)
    | MatInit(ell)        ->  MatrixTyp
    (* empty matrices have an inner type of NumberTyp by default *)
    | MatEmptyInit(_,_)   ->  MatrixTyp
    (* matrix access has the same type as the inner type of the matrix *)
    | MatAcc(i, _, _)     ->  NumberTyp
    (* determine return type of binary op based on the types of the expressions *)
    | Binop(e1,op,e2)     ->  get_oper_typ (Bop(op)) [(get_expr_typ e1); (get_expr_typ e2)]
    (* determine return type of unary op based on the types of the expression *)
    | Unop(op, e)         ->  get_oper_typ (Uop(op)) [(get_expr_typ e)]
    (* assign returns the same type as the expression *)
    | Assign(_, e)        ->  get_expr_typ e
    (* func has the same type as the return type from the function declaration *)
    | Func(s, es)         ->  let func_id = func_id_of_name_params app_state s es in
                              (try (FuncMap.find func_id app_state.funcs).fdtype
                               with Not_found ->
                               raise (Failure ("get_expr_typ: Can't find function " ^
                                               string_of_func_id func_id)))
  in
  { func_name=func_name; func_params=(List.map (get_expr_typ) func_params) }

let func_id_of_symbol_params app_state oper_symbol oper_params =
  let possible_func_id =
    func_id_of_name_params app_state (operator_overload_name oper_symbol) oper_params in
  let oper_id = { oper_symbol=oper_symbol; oper_params=possible_func_id.func_params; } in
  let valid_operator =
    if OperMap.mem oper_id app_state.opers then true
    else raise (Failure (except_unknown_operator (string_of_oper_id oper_id)))
  in
  if (valid_operator && FuncMap.mem possible_func_id app_state.funcs)
  then Some(possible_func_id)
  else None (* Is just a system function that has no corresponding overload function *)

(*  Iterate over the global statements, as if we're evaluating them.
    Build a application declarations objects, ensuring that whenever we
    try to use a variable, function, or oper overload, it has been
    declared. *)
let check_declarations (global_statements) =
  let confirm_variable_declared app_state var_name =
    let is_global_var = StringMap.mem var_name app_state.vars in
    let is_local_var =
      match app_state.local_context with
        | Some(f) -> StringMap.mem var_name (try FuncMap.find f app_state.func_locals
                                             with Not_found ->
                                             raise (Failure ("confirm_variable_declared: Can't find local con
        | None    -> false in
    if (is_local_var || is_global_var) then app_state
```

```
  else raise (Failure (except_undeclared_var var_name))
in
let confirm_func_declared app_state func_name =
  if StringSet.mem func_name app_state.func_names then app_state
  else raise (Failure (except_undeclared_func func_name))
in
let try_add_var_decl app_state var_name var_typ =
  if app_state.decl_allowed
  then
    let is_global_var = StringMap.mem var_name app_state.vars in
    let is_in_local_context = app_state.local_context != None in
    let is_local_var =
      match app_state.local_context with
        | Some(f) -> StringMap.mem var_name (try FuncMap.find f app_state.func_locals
                                             with Not_found ->
                                             raise (Failure ("try_add_var_decl: Can't find local context"))
        | None    -> false
    in
    let will_add_locally = is_in_local_context && not(is_local_var) in
    let will_add_globally = not(is_in_local_context) && not(is_global_var) in

    if will_add_locally || will_add_globally
    then AppState.add_var app_state var_name var_typ
    else raise (Failure (except_declaration_prohibited var_name))
  else raise (Failure (except_declaration_prohibited var_name))
in
let try_add_func_decl app_state func_sig func_decl =
  if not (FuncMap.mem func_sig app_state.funcs)
  then AppState.add_func app_state func_sig func_decl
  else raise (Failure (except_dupe_func (string_of_func_id func_sig)))
in
let try_add_oper_decl app_state oper_sig oper_decl =
  if not (OperMap.mem oper_sig app_state.opers)
  then AppState.add_oper app_state oper_sig oper_decl
  else raise (Failure (except_dupe_oper (func_string_of_oper_id oper_sig)))
in
let handle_mat_init app_state ell =
  let row_count = List.length ell in
  let col_count = List.length (List.hd ell) in
  let mat_is_rect =
    let check_row r = if ((List.length r) != col_count)
                      then raise (Failure (except_jagged_mat (string_of_expr_list_list ell))) in
  List.iter check_row ell
  in
  let fname = "_mat_init_" ^ string_of_int row_count ^ "_by_" ^ string_of_int col_count in
  let mat_id = "tmp_mat" in
  let flattened_el = List.fold_left (fun a l -> a @ l) [] ell in
  let el_types =
    let param_types = (func_id_of_name_params app_state "" flattened_el).func_params in
    let confirm_typ t = if (t != NumberTyp)
                        then raise (Failure (except_type_mat_init (string_of_typ t))) in
    List.iter confirm_typ param_types
  in
  let params =
    let rec gen_list acc = function 0 -> acc | _ as l  -> gen_list (l::acc) (l - 1) in
    let len = row_count * col_count in
    List.map (fun e -> (NumberTyp, "c" ^ string_of_int e)) (gen_list [] len)
  in
  let assigns = List.mapi (fun i (_, id) -> Expr(Assign(
      MatCellAsn(mat_id, Number(IntTyp, (i + 1), 0.0), Number(IntTyp, 0, 0.0)), Id(id)))) params in
  let return = [Return(Id(mat_id))] in
  let body = Block([
    Expr(Assign(VarDecl((MatrixTyp), mat_id),
              MatEmptyInit(Number(IntTyp, row_count, 0.0), Number(IntTyp, col_count, 0.0))));
  ] @ assigns @ return) in
  let func_decl = { fdtype = MatrixTyp; fname = fname; fparams = params; fbody = body; } in
  let func_id =
    let func_params = List.map (fun _ -> Number(IntTyp, 0, 0.0)) params in
```

```
    func_id_of_name_params app_state fname func_params in
  AppState.add_func app_state func_id func_decl
in

let handle_for_loop app_state id =
  let temp_var_name = "_tmp_for_i" ^ id in
  let mat_len_name = "_tmp_mat_len" ^ id in
  AppState.add_var (AppState.add_var app_state temp_var_name NumberTyp) mat_len_name NumberTyp
in
let rec check_expression app_state = function
  (* cases not using a variable or function *)
    Number(_,_,_) | Str(_) | Noexpr  -> app_state
  (* confirm variable is declared *)
  | Id(i)               ->  confirm_variable_declared app_state i
  (* recurse into expressions,
     because we turn off declaration for inside the expressions, return the previous state*)
  | MatInit(ell)        ->  ignore(check_expression_list_list (AppState.prohibit_decl app_state) ell);
                            handle_mat_init app_state ell;
  (* recurse into expressions,
     because we turn off declaration for inside the expressions, return the previous state*)
  | MatEmptyInit(e1,e2) ->  ignore(check_expression (check_expression
                                      (AppState.prohibit_decl app_state) e1) e2);
                            app_state;
  (* confirm variable is declared, recurse expressions,
     because we turn off declaration for inside the expressions, return the previous state *)
  | MatAcc(i, e1, e2)   ->  ignore(confirm_variable_declared app_state i);
                            ignore(check_expression (check_expression
                                       (AppState.prohibit_decl app_state) e1) e2);
                            app_state;
  (* recurse expressions, confirm that we have operator overload if appropriate *)
  | Binop(e1,op,e2)      ->  check_operator app_state (Bop(op)) [e1; e2]
  (* recurse expression, confirm that we have operator overload if appropriate *)
  | Unop(op, e)          ->  check_operator app_state (Uop(op)) [e]
  (* check lvalue, recurse expression *)
  | Assign(l, e)        ->  check_lvalue (check_expression app_state e) l
  (* confirm function is declared then check function body *)
  | Func(s, es)         ->  ignore(check_expression_list app_state es);
                            ignore(confirm_func_declared app_state s);
                            check_func_body_by_name app_state s es

and check_expression_list app_state el =
  List.fold_left check_expression app_state el
and check_expression_list_list app_state ell =
  List.fold_left check_expression_list app_state ell

and check_lvalue app_state = function
  (* add variable declaration *)
    VarDecl((t, i))       ->  try_add_var_decl app_state i t
  (* check variable declared *)
  | IdAsn(i)              ->  confirm_variable_declared app_state i
  (* check variable declared and recurse expressions *)
  | MatCellAsn(i, e1, e2) ->  ignore(confirm_variable_declared app_state i);
                              ignore(check_expression (check_expression
                                        (AppState.prohibit_decl app_state) e1) e2);
                              app_state;
and check_operator app_state oper params =
  let app_state = List.fold_left check_expression app_state params in
  let oper_func_id = func_id_of_symbol_params app_state oper params in
  match oper_func_id with
    | Some(func_id) -> (let func_decl = (try FuncMap.find func_id app_state.funcs
                                         with Not_found ->
                                         raise (Failure ("check_operator: Can't find function " ^
                                                         string_of_func_id func_id))) in
                        check_func_body app_state func_id func_decl)
    | None          -> app_state

(* Check the body statements of a function and confirm everything is declared before use *)
and check_func_body_by_name app_state func_name func_params =
```

```
  let func_id = func_id_of_name_params app_state func_name func_params in
  let func_decl =
    ignore(check_func_param_types app_state func_id);
    (try FuncMap.find func_id app_state.funcs
     with Not_found -> raise (Failure ("check_func_body_by_name:␣Can't␣find␣function␣" ^
                                       string_of_func_id func_id))) in
  check_func_body app_state func_id func_decl

and check_func_body app_state func_id func_decl =
  let add_params_to_state app_state params =
    List.fold_left (fun a p -> AppState.add_var a (snd p) (fst p)) app_state params in

  if not (FuncSet.mem (func_id_of_func_decl func_decl) app_state.func_bodies_checked)
  then
    ( let func_app_state = AppState.add_func_body_checked app_state func_id in
      let func_app_state = AppState.set_local_context func_app_state (Some(func_id)) in
      let func_app_state = add_params_to_state func_app_state func_decl.fparams in
      let func_app_state = d_check_statement func_app_state func_decl.fbody in
      (* Restore the previous context *)
      AppState.set_local_context func_app_state app_state.local_context)
  else app_state

(* debug check statement call, prints the application state after processing the passed in state *)
and d_check_statement app_state s = check_statement app_state s

and check_statement app_state = function
  (* check each statement in the block *)
    Block(stmts)      ->  List.fold_left d_check_statement app_state stmts
  (* check the expression *)
  | Expr(e)           ->  check_expression app_state e
  (* add the variable declared to the app_state *)
  | VDecl((t, i))     ->  try_add_var_decl app_state i t
  (* check the expression, since decl is prohibited inside the expression return the original state *)
  | Return(e)         ->  check_expression app_state e
  (* check the expression and both statements,
     note decl is prohibited inside the expression but not inside the statement blocks *)
  | If(e, s1, s2)     ->  ignore(check_expression (AppState.prohibit_decl app_state) e);
                          d_check_statement (d_check_statement app_state s1) s2;
  (* check the expression and statement,
     note decl is prohibited inside the expression but not inside the statement block *)
  | While(e, s)       ->  ignore(check_expression (AppState.prohibit_decl app_state) e);
                          d_check_statement app_state s;
  (* replace with final for syntax, will need to check variable declaration,
      matrix expression, and statement *)
  | For(lv,e,s,id)    ->  ignore(check_expression (AppState.prohibit_decl app_state) e);
                          d_check_statement (handle_for_loop (check_lvalue app_state lv) id) s;
in

let add_func_decl app_state func_decl =
  try_add_func_decl app_state (func_id_of_func_decl func_decl) func_decl
in

(* Add the operator to the operator map and also add it as a function entry *)
let add_oper_decl app_state oper_decl =
  let fix_unary_minus oper_decl =
    if (oper_decl.operator = Bop(Sub) && (List.length oper_decl.opparams) = 1)
    then { oper_decl with operator=Uop(Neg); }
    else oper_decl
  in
  let oper_decl = fix_unary_minus oper_decl in
  let func_decl = (func_of_oper oper_decl) in
  let new_app_state = add_func_decl app_state func_decl in
  try_add_oper_decl new_app_state (oper_id_of_oper_decl oper_decl) oper_decl
in

let check_global_statement app_state = function
    Stmt(st)    ->  d_check_statement app_state st
  | FuncDecl(f) ->  add_func_decl app_state f
```

```
        | OperDecl(o) ->  add_oper_decl app_state o
      in

      let check_remaining_funcs app_state =
        let remaining_func_ids = FuncMap.filter (fun id _ ->
            not (FuncSet.mem id app_state.func_bodies_checked)) app_state.funcs in
        FuncMap.fold (fun func_id func_decl app_state -> check_func_body app_state func_id func_decl)
          remaining_func_ids app_state
      in

      let initial_app_state = (add_system_oper_decls
                               (add_system_func_decls AppState.empty)) in
      (* Check variable and function declarations based on invocation ordering *)
      let checked_app_state = List.fold_left check_global_statement initial_app_state global_statements in
      (* Check any functions that were declared but not checked by previous step *)
      let checked_app_state = check_remaining_funcs checked_app_state in

      checked_app_state
  end

module SASTGenerator =
  struct
    let generate_sast app_state global_statements =
      let rec gen_sast_func_name func_context name exprs =
        let func_app_state = AppState.set_local_context app_state (Some(func_context)) in
        let func_id = DeclarationCh.func_id_of_name_params func_app_state name exprs in
        gen_sast_func_name_of_func_id func_id

      and gen_sast_func_name_of_func_id func_id =
        let param_string = String.concat "_" (List.map string_of_typ func_id.func_params) in
        func_id.func_name ^ "_" ^ param_string
      in


      let gen_sast_oper func_context op_symbol exprs orig_exprs =
        let func_app_state = AppState.set_local_context app_state (Some(func_context)) in
        let func_id = DeclarationCh.func_id_of_symbol_params func_app_state op_symbol orig_exprs in
        match func_id with
          | Some(f) ->  Func(gen_sast_func_name_of_func_id f, exprs)
          | None    ->  match op_symbol with
                          | Bop(op) ->  Binop((List.nth exprs 0), op, (List.nth exprs 1))
                          | Uop(op) ->  Unop(op, (List.hd exprs))
      in
      let rec gen_sast_expr_list func_context el = List.map (gen_sast_expr func_context) el

      and gen_sast_expr func_context = function
          MatAcc(s,e1,e2)     ->  MatAcc(s, gen_sast_expr func_context e1, gen_sast_expr func_context e2)
        | MatInit(ell)        ->  gen_sast_mat_init func_context ell
        | MatEmptyInit(e1,e2) ->  MatEmptyInit(gen_sast_expr func_context e1, gen_sast_expr func_context e2)
        | Binop(e1,op,e2)     ->  gen_sast_oper func_context (Bop(op)) [gen_sast_expr func_context e1;
                                  gen_sast_expr func_context e2] [e1; e2]
        | Unop(op, e)         ->  gen_sast_oper func_context (Uop(op)) [gen_sast_expr func_context e] [e]
        | Assign(l, e)        ->  Assign(gen_sast_lvalue func_context l, gen_sast_expr func_context e)
        | Func(s, es)         ->  Func(gen_sast_func_name func_context s es,
                                       gen_sast_expr_list func_context es)
        | _ as e              ->  e

      and gen_sast_lvalue func_context = function
        (* No var decls in SAST because the declaration are all in global and local var lists *)
          VarDecl(_,i)          ->  IdAsn(i)
        | IdAsn(i)              ->  IdAsn(i)
        | MatCellAsn(s,e1,e2)   ->  MatCellAsn(s, gen_sast_expr func_context e1,
                                                  gen_sast_expr func_context e2)

      and gen_sast_mat_init func_context ell =
        let row_count = List.length ell in
        let col_count = List.length (List.hd ell) in
        let fname = "_mat_init_" ^ string_of_int row_count ^ "_by_" ^ string_of_int col_count in
```

```
  let params =
    let rec gen_list acc = function 0 -> acc | _ as l  -> gen_list (l::acc) (l - 1) in
    let len = row_count * col_count in
    List.map (fun _ -> NumberTyp) (gen_list [] len)
  in
  let func_id = { func_name=fname; func_params=params } in
  let func_name = gen_sast_func_name_of_func_id func_id in
  let flattened_el = List.fold_left (fun a l -> a @ l) [] ell in
  Func(func_name, gen_sast_expr_list func_context flattened_el)
in

let rec gen_sast_stmt func_context = function
    Block(stmts)      ->  Block(List.map (gen_sast_stmt func_context) (remove_vdecl_stmts stmts))
  | Expr(e)           ->  Expr(gen_sast_expr func_context e)
  | Return(e)         ->  Return(gen_sast_expr func_context e)
  | If(e, s1, s2)     ->  If(gen_sast_expr func_context e, gen_sast_stmt func_context s1,
                             gen_sast_stmt func_context s2)
  | While(e, s)       ->  While(gen_sast_expr func_context e, gen_sast_stmt func_context s)
  | For(lv,e,s,id)    ->  gen_sast_for_statement func_context lv e s id
  | _ as s            ->  s

and remove_vdecl_stmts stmts =
  let is_not_vdecl = function VDecl(_) -> false | _ -> true in
  List.filter is_not_vdecl stmts

and gen_sast_for_statement func_context lvalue expr statement id =
  let lvalue = gen_sast_lvalue func_context lvalue in
  let mat_id = let (Id s) = expr in s in
  let temp_var_name = "_tmp_for_i" ^ id in
  let mat_len_name = "_tmp_mat_len" ^ id in
  let num n = Number(IntTyp, n, 0.0) in
  let new_statement_block =
    Block([
      Expr(Assign(VarDecl(NumberTyp, temp_var_name), num 1));
      Expr(Assign(VarDecl(NumberTyp, mat_len_name), Func("len", [expr])));
      While(Binop(Id(temp_var_name), Leq, Id(mat_len_name)),
        Block([
          Expr(Assign(lvalue, MatAcc(mat_id, Id(temp_var_name), num 0)));
          statement;
          Expr(Assign(IdAsn(temp_var_name), Binop(Id(temp_var_name), Add, num 1)));
        ]))
    ]) in
  gen_sast_stmt func_context new_statement_block


in

let remove_func_decls glbl_statements =
  (* count both FuncDecl and OperDecl as function declarations *)
  let is_not_fdecl = function Stmt(_) -> true | _ -> false in
  let get_stmt = function Stmt(s) -> s | _ -> Expr(Noexpr) in
  List.map get_stmt (List.filter is_not_fdecl glbl_statements)
in

let remove_params_from_locals locals params =
  let is_not_param p = not(List.mem p params) in
  List.filter is_not_param locals
in

let var_bindings_of_stringmap map =
  StringMap.fold (fun i t l -> (t, i)::l) map []
in

let main = { cdtype=VoidTyp; cname="main";
  cparams=[]; clocal_vars=[]; cbody=gen_sast_stmt {func_name="main"; func_params=[]}
    (Block(remove_func_decls global_statements)) }
in
```

```
      let global_vars =
        var_bindings_of_stringmap app_state.vars;
      in

      let func_decls =
        let gen_sast_func_decl func_id func_decl local_vars =
          let fname = gen_sast_func_name_of_func_id func_id in
          { cdtype=func_decl.fdtype;
            cname=fname;
            cparams=func_decl.fparams;
            clocal_vars=remove_params_from_locals (var_bindings_of_stringmap local_vars) func_decl.fparams;
            cbody=gen_sast_stmt func_id func_decl.fbody; }
        in
        let local_vars func_id = (try FuncMap.find func_id app_state.func_locals
                                  with Not_found ->
                                  raise (Failure ("SAST␣func_decls:␣Can't␣find␣function␣" ^
                                 string_of_func_id func_id))) in
        let process_func func_id func_decl l =
          (gen_sast_func_decl func_id func_decl (local_vars func_id))::l in
        FuncMap.fold process_func app_state.funcs []
      in

      { global_vars=global_vars; func_decls=main::func_decls }

    let clean_up_sast program =
      let remove_sys_functions functions =
        let is_not_sys_function f =
          if  (f.cname = "main_") ||
              (f.cname = "print_number") ||
              (f.cname = "print_string") ||
              (f.cname = "printnl_string") ||
              (f.cname = "printnl_number") ||
              (f.cname = "strcat_string_string") ||
              (f.cname = "shape_matrix") ||
              (f.cname = "len_matrix") then false
          else true
        in
        List.filter is_not_sys_function functions
      in
      { program with func_decls=(remove_sys_functions program.func_decls); }

  end

(* check globals *)
let check (global_statements) =
  let checked_app_state = DeclarationCh.check_declarations global_statements in
  let sast = SASTGenerator.generate_sast checked_app_state global_statements in
  TypeCh.check_types sast.global_vars sast.func_decls checked_app_state;
  SASTGenerator.clean_up_sast sast
```

../src/sast.ml

```
(* Semantically checked Abstract Syntax Tree *)
(* author: Joseph Isaac Baker & William Hom *)


open Ast

type checked_func_decl = {
  cdtype      : typ;
  cname       : string;
  cparams     : binding list;
  clocal_vars : binding list;
  cbody       : stmt;
}

type checked_program = {
  global_vars : binding list;
```

```
  func_decls  : checked_func_decl list;
}


let string_of_cfdecl cfdecl =
  string_of_typ cfdecl.cdtype ^ "␣" ^
  cfdecl.cname ^ "(" ^ String.concat ",␣" (List.map (fun (t, n) ->
      string_of_typ t ^ "␣" ^ n) cfdecl.cparams) ^ ")␣" ^
  "local␣vars:␣" ^ String.concat "" (List.map string_of_vdecl cfdecl.clocal_vars) ^ "\n" ^
  string_of_stmt "" cfdecl.cbody

let string_of_checked_program program =
  "global␣vars:␣" ^ String.concat "" (List.map string_of_vdecl program.global_vars) ^ "\n" ^
  "functions:␣\n" ^ String.concat "\n" (List.map string_of_cfdecl program.func_decls)
```

../src/compilator.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

author: William Hom

*)

module L = Llvm
module A = Ast
module S = Sast

open Llvm.MemoryBuffer
open Llvm_bitreader
open Llvm_linker

module StringMap = Map.Make(String)

(* compilator received a semantically accurate AST from evaluator.
 * It mechanically emits LLVM code from here.
 *)
let translate (blargh) =
  let context = L.global_context () in
  let the_module = L.create_module context "Macaw"
  and i32_t  = L.i32_type   context
  and d64_t  = L.double_type context
  and i8_t   = L.i8_type    context
  and void_t = L.void_type  context in

  let matrix_t = L.struct_type context
    [| L.pointer_type (L.pointer_type d64_t); i32_t; i32_t |] in
  let rec ltype_of_typ = function
      A.NumberTyp -> d64_t
    | A.StringTyp -> L.pointer_type i8_t
    | A.VoidTyp -> void_t
    | A.MatrixTyp -> L.pointer_type matrix_t (* PLACEHOLDER *) in

  let filename = "_includes/matrix.bc" in
  let llctx = Llvm.global_context() in
  let llmem = Llvm.MemoryBuffer.of_file filename in
  let llm = Llvm_bitreader.parse_bitcode llctx llmem in
  ignore(Llvm_linker.link_modules' the_module llm);

  let globals = blargh.S.global_vars and functions = blargh.S.func_decls in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m (t, n) =
      let init = (match t with
        A.NumberTyp -> L.const_float (ltype_of_typ t) 0.0
      | A.StringTyp -> L.const_pointer_null (ltype_of_typ t)
      | A.MatrixTyp -> L.const_null (L.pointer_type matrix_t)
```

```
    | A.VoidTyp -> L.const_null (ltype_of_typ t))
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals

(* print functions *)
in let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func = L.declare_function "printf" printf_t the_module in
let printf_s = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func_s = L.declare_function "printf" printf_s the_module in
let printf_f = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func_f = L.declare_function "printf" printf_f the_module in

(* string concatenation function *)
let strcat_t = L.function_type (L.pointer_type i8_t)
  [| L.pointer_type i8_t; L.pointer_type i8_t |] in
  let strcat_func = L.declare_function "strcat" strcat_t the_module in

(* Initialize a m x n matrix of zeroes *)
let zero_init_f = L.function_type (L.pointer_type matrix_t)
  [| d64_t; d64_t|] in
  let zero_init_func = L.declare_function "zero_matrix_init"
  zero_init_f the_module in

(* Access a specific element of a provided matrix *)
let matrix_access_f = L.function_type d64_t
  [| L.pointer_type (L.pointer_type matrix_t); d64_t; d64_t |] in
  let matrix_access_func = L.declare_function
  "access_element" matrix_access_f the_module in

(* Replace a specific element of a provided matrix *)
let matrix_replace_f = L.function_type d64_t
  [| L.pointer_type (L.pointer_type matrix_t); d64_t; d64_t; d64_t |] in
  let matrix_replace_func = L.declare_function "replace_element"
  matrix_replace_f the_module in

(* Matrix shape *)
let matrix_shape_f = L.function_type (L.pointer_type matrix_t)
  [| L.pointer_type matrix_t |] in
  let matrix_shape_func = L.declare_function "shape"
  matrix_shape_f the_module in

(* Matrix length *)
let matrix_length_f = L.function_type d64_t
  [| L.pointer_type matrix_t |] in
  let matrix_length_func = L.declare_function "matrix_len"
  matrix_length_f the_module in

let function_decls =
  let function_decl m fdecl =
    let name = fdecl.S.cname
    and formal_types = Array.of_list (List.map (fun (t,_) ->
        ltype_of_typ t) fdecl.S.cparams)
    in let ftype = L.function_type
    (ltype_of_typ fdecl.S.cdtype) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty functions in

let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.S.cname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let str_format_str = L.build_global_stringptr "%s\n" "str" builder in
  let float_format_str = L.build_global_stringptr "%.3f\n" "float" builder in

  let str_format_str_nl = L.build_global_stringptr "%s" "str" builder in
  let float_format_str_nl = L.build_global_stringptr "%.3f" "float" builder in
  (* Construct the function's local variables: param arguments and locally
     declared variables.  Allocate each on the stack, initialize their
```

```
   value, if appropriate, and remember their values in the map *)
let local_vars =
  let add_param m (t, n) p = L.set_value_name n p;
  let local = L.build_alloca (ltype_of_typ t) n builder in
    ignore (L.build_store p local builder);
    StringMap.add n local m in
  let add_local m (t, n) =
  let local_var = L.build_alloca (ltype_of_typ t) n builder
    in StringMap.add n local_var m in
  let params = List.fold_left2 add_param StringMap.empty fdecl.S.cparams
      (Array.to_list (L.params the_function)) in
      List.fold_left add_local params fdecl.S.clocal_vars in

(* Return the value for a variable or param argument *)
let lookup n = try StringMap.find n local_vars
  with Not_found -> StringMap.find n global_vars
in

(* Construct code for an expression; return its value *)
let rec expr builder = function
    A.Number(t, n1, n2) -> (match t with
      A.IntTyp -> L.const_sitofp (L.const_int i32_t n1) d64_t
    | A.FloatTyp -> L.const_float d64_t n2)
  | A.Noexpr -> L.const_int i32_t 0
  | A.Id s -> L.build_load (lookup s) s builder
  (*
  | A.Id s -> let x = lookup s in (match (fst x) with
      A.NumberTyp -> L.build_load (L.const_fptosi (snd x) i32_t) s builder
    | _ -> L.build_load (snd x) s builder)
   *)
  | A.Str s -> L.build_global_stringptr
    (String.sub s 1 ((String.length s) - 2)) "" builder
  | A.Binop (e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add     -> L.build_fadd
    | A.Sub     -> L.build_fsub
    | A.Mul     -> L.build_fmul
    | A.Div     -> L.build_fdiv
    | A.Mod     -> L.build_frem
    | A.And     -> L.build_and
    | A.Or      -> L.build_or
    | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
    | A.Neq     -> L.build_fcmp L.Fcmp.One
    | A.Less    -> L.build_fcmp L.Fcmp.Olt
    | A.Leq     -> L.build_fcmp L.Fcmp.Ole
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt
    | A.Geq     -> L.build_fcmp L.Fcmp.Oge) e1' e2' "tmp" builder
  | A.Unop(op, e) ->
    let e' = expr builder e in (match op with
      A.Neg     -> L.build_fneg
    | A.Not     -> L.build_not) e' "tmp" builder
  | A.Assign (l, e) -> (match l with
    | A.MatCellAsn (m, r, c) -> L.build_call matrix_replace_func
        [| (lookup m); (expr builder r);
        (expr builder c); (expr builder e) |]
        "access_element" builder
    | A.VarDecl (t, n) -> let e' =
        (expr builder e) in ignore (L.build_store e' (lookup n) builder); e'
    | A.IdAsn k -> let e' =
        expr builder e in ignore (L.build_store e' (lookup k) builder); e')
  | A.MatEmptyInit (r, c) -> L.build_call zero_init_func
    [| (expr builder r); (expr builder c)|] "zero_init" builder
  | A.MatAcc (e, r, c) -> L.build_call matrix_access_func
    [| (lookup e); (expr builder r); (expr builder c) |] "mat_acc" builder
  | A.Func ("print_string", [e]) -> L.build_call printf_func_s
    [| str_format_str; (expr builder e) |] "printf" builder
```

```
    | A.Func ("print_number", [e]) -> L.build_call printf_func_f
      [| float_format_str; (expr builder e) |] "printf" builder
    | A.Func ("printnl_string", [e]) -> L.build_call printf_func_s
      [| str_format_str_nl; (expr builder e) |] "printf" builder
    | A.Func ("printnl_number", [e]) -> L.build_call printf_func_f
      [| float_format_str_nl; (expr builder e) |] "printf" builder
    | A.Func ("strcat_string_string", [e1; e2]) -> L.build_call strcat_func
      [| (expr builder e1); (expr builder e2) |] "strcat" builder
    | A.Func ("shape_matrix", [e]) -> L.build_call matrix_shape_func
      [| (expr builder e) |] "shape" builder
    | A.Func ("len_matrix", [e]) -> L.build_call matrix_length_func
      [| (expr builder e) |] "matrix_len" builder
    | A.Func (f, act) ->
       let (fdef, fdecl) = StringMap.find f function_decls in
   let actuals = List.rev (List.map (expr builder) (List.rev act)) in
   let result = (match fdecl.S.cdtype with A.VoidTyp -> ""
                                         | _ -> f ^ "_result") in
       L.build_call fdef (Array.of_list actuals) result builder
 in

 let add_terminal builder f =
   match L.block_terminator (L.insertion_block builder) with
     Some _ -> ()
   | None -> ignore (f builder) in

 let rec stmt builder = function
     A.Block sl -> List.fold_left stmt builder sl
   | A.VDecl (t, n) -> builder
   | A.Expr e -> ignore (expr builder e); builder
   | A.Return e -> ignore (match fdecl.S.cdtype with
       A.VoidTyp -> L.build_ret_void builder
     | _ -> L.build_ret (expr builder e) builder); builder
   | A.If (predicate, then_stmt, else_stmt) ->
      let bool_val = expr builder predicate in
let merge_bb = L.append_block context "merge" the_function in

let then_bb = L.append_block context "then" the_function in
add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
  (L.build_br merge_bb);

let else_bb = L.append_block context "else" the_function in
add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
  (L.build_br merge_bb);

ignore (L.build_cond_br bool_val then_bb else_bb builder);
L.builder_at_end context merge_bb

   | A.While (predicate, body) ->
 let pred_bb = L.append_block context "while" the_function in
 ignore (L.build_br pred_bb builder);

 let body_bb = L.append_block context "while_body" the_function in
 add_terminal (stmt (L.builder_at_end context body_bb) body)
   (L.build_br pred_bb);

 let pred_builder = L.builder_at_end context pred_bb in
 let bool_val = expr pred_builder predicate in

 let merge_bb = L.append_block context "merge" the_function in
 ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
 L.builder_at_end context merge_bb
 in

 (* Build the code for each statement in the function *)
 let builder = stmt builder fdecl.S.cbody in

 (* Add a return if the last block falls off the end *)
 add_terminal builder (match fdecl.S.cdtype with
```

```
        A.VoidTyp -> L.build_ret_void
      | t -> L.build_ret (L.const_float (ltype_of_typ t) 0.0))
in List.iter build_function_body functions;
the_module
```

../src/macaw.ml

```
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

type action = Ast | Compile | LLVM_IR

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);  (* Print the AST only *)
            ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
            ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast = Evaluator.check ast in
  match action with
    Ast ->  print_string (Ast.string_of_program ast);
            print_string "\n\nSemantically␣Checked␣AST:␣\n";
            print_string (Sast.string_of_checked_program sast)
  | LLVM_IR -> let p = (Compilator.translate sast) in
    print_string (Llvm.string_of_llmodule p)
  | Compile -> let p = (Compilator.translate sast) in
    Llvm_analysis.assert_valid_module p;
    print_string (Llvm.string_of_llmodule p)
```

../src/_includes/stdlib.ma

```
# General Helper functions
matrix range(number start, number end) {
        number len <- end - start + 1;
        number i <- 0;
        matrix r <- [](1, len);
        while (i < len) {
                r[i + 1] <- start + i;
                i <- i + 1;
        }
        return r;
}


void print(matrix A) {
        matrix shape <- shape(A);
        number rs <- shape[1];
        number cs <- shape[2];

        matrix ris <- range(1,rs);
        matrix cis <- range(1,cs);
        for (number ri in ris) {
                for (number ci in cis) {
                        printnl (A[ri, ci]);
                        if (ci < cs) {
                                printnl (",␣");
                        }
                }
                print(";␣");
        }
}

# Number Operations
number operator ^ (number a, number b) {
        number c <- 1;
        while (b > 0) {
```

```
                c <- c * a;
                b <- b - 1;
        }

        return c;
}

# Matrix Operations

# returns true if the matrix shapes are equal.
number operator = (matrix A, matrix B) {
        matrix shapeA <- shape(A);
        matrix shapeB <- shape(B);

        if ((shapeA[1] = shapeB[1]) & (shapeA[2] = shapeB[2])) {
                return 1;
        } else {
                return 0;
        }
}

matrix operator ' (matrix A) {
        matrix shape <- shape(A);
        number rs <- shape[1];
        number cs <- shape[2];
        matrix ris <- range(1,rs);
        matrix cis <- range(1,cs);

        matrix new <- [](cs, rs);
        for (number ri in ris) {
                for (number ci in cis) {
                        new[ci,ri] <- A[ri,ci];
                }
        }

        return new;
}

matrix operator - (matrix A) {
    matrix shape <- shape(A);
    number rs <- shape[1];
    number cs <- shape[2];
    matrix ris <- range(1,rs);
    matrix cis <- range(1,cs);

    matrix new <- [](cs, rs);
    for (number ri in ris) {
        for (number ci in cis) {
            new[ci,ri] <- -A[ri,ci];
        }
    }

    return new;
}

matrix operator + (number a, matrix B) {
        matrix shape <- shape(B);
        matrix new <- [](shape[1], shape[2]);

        number x <- 1;
        for (number i in B) {
                new[x] <- a + i;
                x <- x + 1;
        }

        return new;
}
matrix operator + (matrix B, number a) {
```

```
        return a + B;
}

matrix operator + (matrix A, matrix B) {
        matrix shape <- shape(A);
        matrix new <- [](shape[1], shape[2]);
        number x <- (A = B);

        if (x = 1) {
                matrix r <- range(1, len(A));

                for (number q in r) {
                        new[q] <- A[q] + B[q];
                }
        }

        return new;
}

matrix operator - (number a, matrix B) {
        matrix shape <- shape(B);
        matrix new <- [](shape[1], shape[2]);

        number x <- 1;
        for (number i in B) {
                new[x] <- a - i;
                x <- x + 1;
        }

        return new;
}
matrix operator - (matrix A, number b) {
        matrix shape <- shape(A);
        matrix new <- [](shape[1], shape[2]);

        number x <- 1;
        for (number i in A) {
                new[x] <- i - b;
                x <- x + 1;
        }

        return new;
}
matrix operator - (matrix A, matrix B) {
        matrix shape <- shape(A);
        matrix new <- [](shape[1], shape[2]);
        number x <- (A = B);

        if (x = 1) {
                matrix r <- range(1, len(A));

                for (number q in r) {
                        new[q] <- A[q] - B[q];
                }
        }

        return new;
}

matrix operator .* (matrix A, matrix B) {
        matrix shape <- shape(A);
        matrix new <- [](shape[1], shape[2]);
        number x <- (A = B);

        if (x = 1) {
                matrix r <- range(1, len(A));

                for (number q in r) {
```

```
                        new[q] <- A[q] * B[q];
                }
        }
        return new;
}
matrix operator * (matrix A, number b) {
        matrix shape <- shape(A);
        matrix new <- [](shape[1], shape[2]);

        matrix r <- range(1, len(A));

        for (number q in r) {
                new[q] <- A[q] * b;
        }
        return new;
}
matrix operator * (number a, matrix B) {
        return B * a;
}

matrix operator * (matrix A, matrix B) {
        matrix Ashape <- shape(A);
        matrix Bshape <- shape(B);
        matrix new <- [](Ashape[1], Bshape[2]);

        if (Ashape[2] = Bshape[1]) {
                matrix ars <- range(1, Ashape[1]);
                matrix ins <- range(1, Ashape[2]);
                matrix bcs <- range(1, Bshape[2]);
                for (number ari in ars) {
                        for (number bci in bcs) {
                                number sum <- 0;
                                for (number i in ins) {
                                        sum <- sum + A[ari, i] * B[i, bci];
                                }
                                new[ari, bci] <- sum;
                        }
                }

        }
        return new;
}


matrix operator ./ (matrix A, matrix B) {
        matrix shape <- shape(A);
        matrix new <- [](shape[1], shape[2]);
        number x <- (A = B);

        if (x = 1) {
                matrix r <- range(1, len(A));

                for (number q in r) {
                        new[q] <- A[q] / B[q];
                }
        }
        return new;
}
```

../src/_includes/matrix.h

```
/*
 * matrix header file
 */

struct matrix {
    double **vectors;
    int rows;
```

```
    int columns;
};

// initialize a matrix (here the matrix itself may already be defined, but
// initialization guarantees we always know the shape).
struct matrix* matrix_init(double **vectors, double rows, double columns);

// in the case in which the matrix itself isn't defined, this function may be
// used to initialize a zero-matrix of size m x n.
struct matrix* zero_matrix_init(double rows, double columns);

// access an element in the matrix and return it.
double access_element(struct matrix **mat, double row, double column);

// replaces the element specified by the row and column of the matrix.
// returns the replaced value.
double replace_element
    (struct matrix **mat, double row, double column, double value);

// returns r x c of matrix
double matrix_len(struct matrix *mat);

// returns shape of the matrix in the form of a 2x1 matrix.
struct matrix *shape(struct matrix *mat);

// prints our matrix
void print_matrix(struct matrix **mat);

// delete the matrix and free up memory.
void delete_matrix(struct matrix* matrix);
```

../src/_includes/matrix.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "matrix.h"

/*
 * matrix implementation
 * author: William HOm
 */

int count_digits(int n) {
    int count = 0;
    while (n != 0) {
        n /= 10;
        count++;
    }

    return count;
}

int find_largest_digits(struct matrix *matrix) {
    int largest_digits = 1;
    for (int i = 0; i < matrix->rows; i++) {
        for (int j = 0; j < matrix->columns; j++) {
            int digit_count = count_digits(matrix->vectors[i][j]);
            if (digit_count > largest_digits) {
                largest_digits = digit_count;
            }
        }
    }

    return largest_digits;
}

double matrix_len(struct matrix *mat) {
```

```
    struct matrix *matrix = mat;

    return ((double) matrix->rows * (double) matrix->columns);
}

struct matrix *shape(struct matrix *mat) {
    struct matrix *s = zero_matrix_init(2, 1);
    struct matrix *matrix = mat;

    replace_element(&s, 1, 1, (double) matrix->rows);
    replace_element(&s, 2, 1, (double) matrix->columns);

    return s;
}

struct matrix *zero_matrix_init(double rows, double columns) {
    int r = (int) rows;
    int c = (int) columns;

    struct matrix *matrix = malloc(sizeof(struct matrix));

    matrix->rows = r;
    matrix->columns = c;
    matrix->vectors = (double **) malloc(sizeof(double *) * r);
    for (int i = 0; i < r; i++) {
        matrix->vectors[i] = (double *) malloc(sizeof(double) * c);
    }

    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            matrix->vectors[i][j] = 0.0;
        }
    }

    return matrix;
}

struct matrix *matrix_init(double **vectors, double rows, double columns) {
    struct matrix *mat = malloc(sizeof(struct matrix));
    mat->vectors = vectors;
    mat->rows = (int) rows;
    mat->columns = (int) columns;

    return mat;
}

double access_element(struct matrix **mat, double row, double column) {
    int r = (int) row;
    int c = (int) column;
    struct matrix *matrix = *mat;

    if (c == 0) {
        int coord = r;
        r = (coord - 1) / matrix->columns;
        c = (coord - 1) % matrix-> columns;
    } else {
        --r; --c;
    }

    double x = matrix->vectors[r][c];

    return x;
}

double replace_element(struct matrix **mat, double row, double column, double value) {
    int r = (int) row;
    int c = (int) column;
    struct matrix *matrix = *mat;
```

```
    if (c == 0) {
        int coord = r;
        r = (coord - 1) / matrix->columns;
        c = (coord - 1) % matrix->columns;
    } else {
        --r; --c;
    }

    double x = matrix->vectors[r][c];
    matrix->vectors[r][c] = value;

    return x;
}

void delete_matrix(struct matrix *matrix) {
    for (int i = 0; i < matrix->rows; i++) {
        free(matrix->vectors[i]);
    }

    free(matrix->vectors);
    free(matrix);
}

void print_matrix(struct matrix **mat) {
    struct matrix *matrix = *mat;

    int width = find_largest_digits(matrix);

    for (int i = 0; i < matrix->rows; i++) {
        for (int j = 0; j < matrix->columns; j++) {
           printf("%*f␣", width, matrix->vectors[i][j]);
        }
        printf("\n");
    }
}
```

../src/_includes/matrix_test.c

```
#include <stdlib.h>
#include <stdio.h>
#include "matrix.h"

/*
 * matrix "test" file.
 * author: William Hom
 */

int main() {
// test a zero-matrix.
    struct matrix *my_matrix = zero_matrix_init(66, 34);
    for (int i = 1; i < 6; i++) {
        for (int j = 1; j < 6; j++) {
            printf("%d␣%d␣-␣%.3f\n", i, j, access_element(&my_matrix, i, j));
        }
    }

    struct matrix *m = shape(my_matrix);
    print_matrix(&m);

    printf("%.3f\n", matrix_len(my_matrix));

// testing out a matrix initialization where an array is already defined.
    double **my_vec = (double **) malloc(sizeof(double *) * 2);
    for (int i = 0; i < 2; i++) {
        my_vec[i] = (double *) malloc(sizeof(double) * 3);
    }
```

```
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            my_vec[i][j] = (i + j) * (j + 1);
        }
    }

    struct matrix *new_matrix = matrix_init(my_vec, 2, 3);
    for (int i = 1; i < 3; i++) {
        for (int j = 1; j < 4; j++) {
            printf("%d␣%d␣-␣%.3f\n", i, j, access_element(&new_matrix, i, j));
        }
    }

    double replaced = replace_element(&new_matrix, 2, 2, 1242);

    printf("%.3f␣replaced␣with␣%.3f\n",
            replaced,
            access_element(&new_matrix, 2,2));

    print_matrix(&new_matrix);
    delete_matrix(my_matrix);
    delete_matrix(new_matrix);
}
```