Macaw Reference Manual

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

July 20, 2016

Contents

1 Introduction

This is the language reference manual for Macaw implemented as part of the Programming Languages and Translators team at Columbia University. The intent of this manual is to document the features of this language, including data type, syntax, and control flows.

Macaw was originally conceived as a calculator language that would support matrices as a primitive data type while having control constructs familiar to a C programmer. It has been heavily inspired by C, Matlab, and R. The end result is distinct from C in the flexibility of the source files (code need not be in a function, no 'main') while still retaining the imperative ease of use that programmers of any C descendent language will recognize.

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

2 Lexical Conventions

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

2.1 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

2.2 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. 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.

2.3 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.

2.4 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`	`function`	`for`	`print`
`matrix`	`operator`	`to`	`shape`
`string`	`if`	`while`	
`void`	`else`	`return`	

2.5 Literals

2.5.1 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]+)?*

2.5.2 String Literal

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

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

2.5.3 Matrix Literal

Matrices are a collection of `number`, `string`, or `matrix` 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 of type `number` and are given the value '`0`'.

3 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 atomic type such as `number`, `string`, or in the case of functions `void`; or a composite type such as `matrix<number>` or `matrix<string>`. Because `matrix` objects can contain `matrix` objects, there are an conceptually infinite number of composite types via nesting.

4 Conversions

In Macaw you cannot convert between `number`, `string`, and any kind of `matrix`. For `number`, internally Macaw stores additional information about whether the type has a fractional part which allows counters and other integer based number to remain integers in memory. Macaw's type checking only enforces that a `number` is provided where a `number` is expected, it makes no distinction between integer and float. Any implicit conversion from integer to float or vice versa is handled by the runtime environment.

5 Expressions

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

5.1 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.

5.1.1 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. Typically the expression outside the brackets will be an identifier of type `matrix`, but it could be the result of another expression such as a function that returns a `matrix` or a mathematical operation that returns a `matrix`. There are two variants, the first of which allows `matrix` to also behave intuitively as a vector construct.

- *expression* `[` *expression* `]` - Returns contents of the cell of the `matrix` in a linear ordering of the `matrix` which iterates through the columns. (That is, all values in the first column are before any value in the second column).
- *expression* `[` *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.

5.2 Unary Expressions

- −*expression* - Negation of a `number` or `matrix`
- *expression*′ - Transpose of a matrix
- `!` *expression* - Logical 'not'. Takes an `number` value and returns 0 if the `number` is not 0, and 1 otherwise

5.3 Multiplicative Expressions

- *expression* `*` *expression* - Multiplication of `number` types and dot product of `matrix` types
- *expression* / *expression* - Division of `number` types
- *expression* `%` *expression* - Modulus of `number` types

5.4 Additive Expressions

- *expression* + *expression* - Addition of `number` and `matrix` types and concatenation for `string` types
- *expression* − *expression* - Subtraction of `number` and `matrix` types

5.5 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*

5.6 Equality Operators

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

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

5.7 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

5.8 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.

5.8.1 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`.

5.8.2 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 effect the other one. `matrix` 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`). `string` types are immutable and so the distinction is irrelevant for them.

6 Statements

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

6.1 Statement Blocks

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

6.2 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.
- `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* = *expression* `to` *expression* `)` *statement-block* - For loop. The first *expression* is evaluated and assigned to the *lvalue* and the *statement-block* is executed. After the *statement-block*'s execution the *lvalue's identifier*'s value is incremented by one and compared to the `to` *expression*. If the *lvalue's identifier* is less than or equal to the `to` *expression*, the *statement-block* is executed again. This continues until the *lvalue's identifier*'s value is greater than the `to` *expression*.
- `return;` - Returns control to the function's caller, any remaining code in the function after the `return` statement is not executed. 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.

7 Declarations

Both variables and function 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 atomic types in Macaw are `number` and `string`, and the only composite type is `matrix<?>`, where `?` here represents another type. When declaring a `matrix` users will instead use `matrix<number>` or similar.

7.1 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.

7.2 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 use as part of an expression which expects a value like assignment or mathematical operators.

Functions are also allowed to have an optional argument list. 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*

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.

8 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 variables 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 and between themselves. If an argument 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, not the global variable. Similarly local variables can 'shadow' global variables and argument 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. Similarly, if a variable is declared inside a conditional control flow that only executes under certain conditions, such as `if, else for,` and `while`, it cannot be used outside of that block because the compiler will not be able to determine if the variable has been declared. Although not enforced, due to this we encourage developers using Macaw to place their declarations outside such control constructs directly in the body of the function.

9 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<number>` types, we can easily define a function that performs the equality comparison of the two matrices. Operator overloading allows us to tell the compiler to attach that function to the behavior of the + symbol.

During type checking and code generation, if the compiler sees that 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 functions.

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. Language implementations take precedence over overloaded operators. Only the following operators defined in the language can be overloaded:

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

10 Language Functions

Language function 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.

10.1 Print

`print` takes in one argument, and prints the value of that argument to the console. Works for `number`, `string`, `matrix` types.

10.2 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 the passed in matrix has. The second value (accessible with `[2]`) is the number of columns of the matrix.

11 Grammar

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

- lvalue
ID
type ID

- statement-block
{ statement list }

- statement
expr;
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
```