CMPT 330L Language Design Project

Sufia Khan

May 9, 2025

Table of Contents

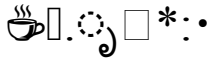
1. Introduction....3
1.1. Genealogy....4
1.2. Hello world....5
1.3. Program structure....5
1.4. Types and Variables....7
1.5. Visibility....7
1.6. Statements Differing from Java and BASIC....7
2. Lexical structure....9
2.1. Programs....9
2.2 Grammars....9
2.2.1 Lexical grammar (tokens) where different from Java and BASIC....9
2.2.2 Syntactic (“parse”) grammar where different from Java and BASIC....10
2.3 Lexical analysis....10
2.3.1 Comments....10
2.4 Tokens....11
2.4.1 Keywords different from Java or BASIC....11
3. Type System....12
3.1 Type Rules....12
3.2 Value types (differing from Java and BASIC)....13
3.3 Reference types (differing from Java and BASIC)....13
4. Example Programs....14
4.1 Caesar Cipher encrypt and decrypt....14
4.2 Factorial....15
4.3 Bubble Sort (Swap function included)....15
4.4 QuickSort....16
4.5 MergeSort....17

☕.◌ ◌*:•

1. Introduction

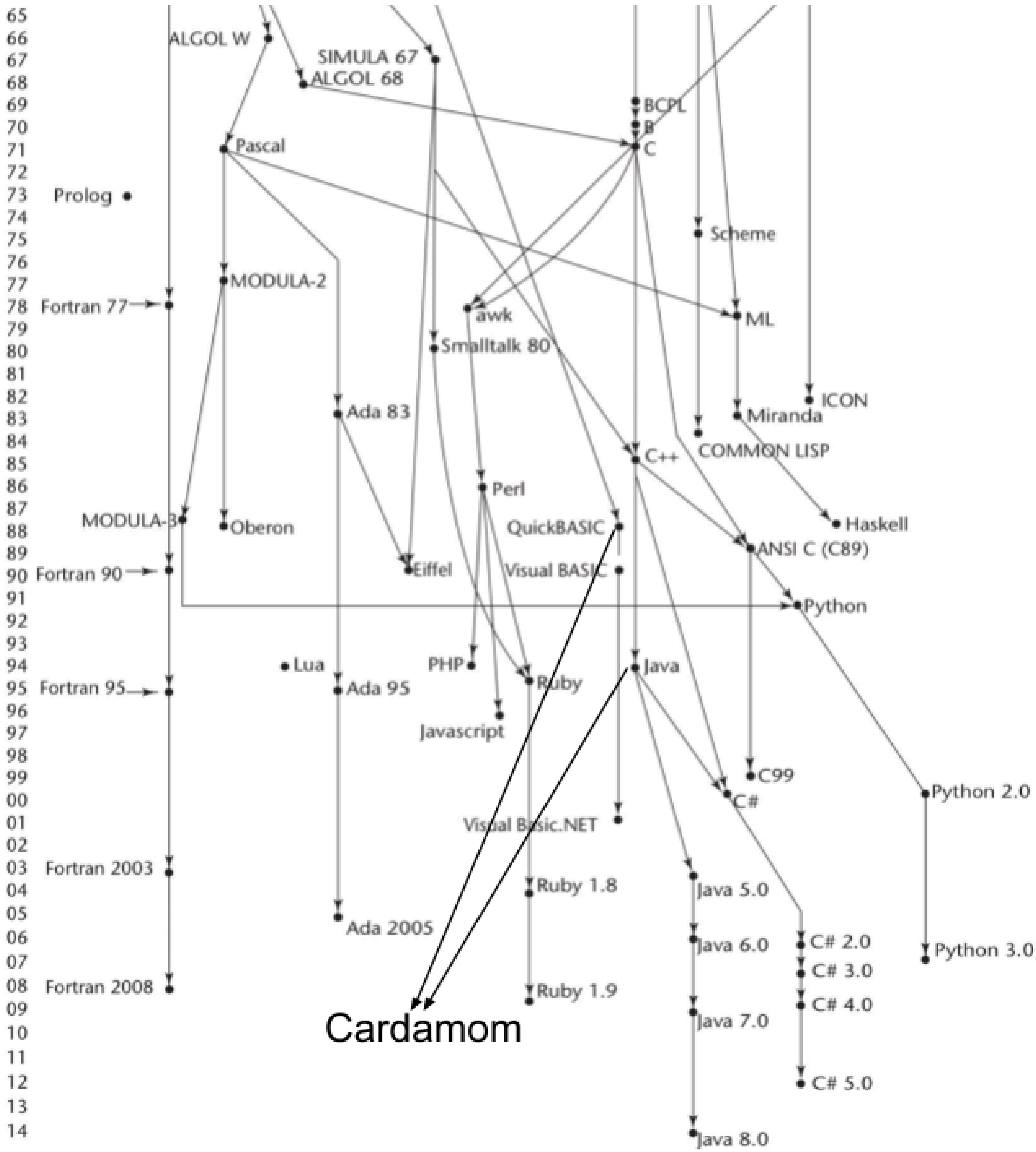
Cardamom (pronounced card−ah−mum) is a simple, comforting, and refreshing language that is both object−oriented and type−safe. It is suitable for beginner and intermediate programmers as it supports both procedural and functional programming. It is named after the "Queen of Spices", cardamom, which is native to South Asia. Like its namesake, it's revitalizing and sweet compared to earlier programming languages and transports the programmer to a cafe−like world.
Cardamom is derived primarily from Java and some BASIC, but has an identity of its own as it differentiates from the two in the following ways:

1. All Cardamom programs **must** have classes, denoted by the `Chai` keyword.
2. There are no one−dimensional lists. Instead, tables/arrays are used for both one−dimensional and multi−dimensional lists, denoted by the `AromaSet` keyword .
3. Variables cannot be declared without being defined. If you know you need a variable, you must only create it at the point in the code where it will be defined as well. The exception to this is function parameters. To define a variable, use the `Fill` keyword.
4. Each line of code within a class/function **must** end with a tilde (~), or else you will get syntax errors.
5. Only `while` loops exist, which can be used to recreate the functionality of other types of loops.
6. Recursion is done with the `useMasala()` built−in function, where the recursive function and its parameters are called as a parameter. This is also how functions are normally called in other parts of your program, regardless of recursivity. This built−in function is not required to call other built−in functions like `sub()` and `add()`.



1.1. Genealogy

The below diagram illustrates Cardamom's place within the programming language genealogy:



1.2. Hello world

```
Public Chai Hello {
    Masala HelloWorld() as void {
        Fill String hi = “Hello, World!”~
        Pour(hi)~
    }
}
```

1.3. Program structure

The key organizational concepts in Cardamom are as follows:

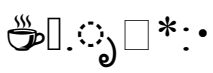
1. Chais (classes) must be located in files with the same name and ending with the `.crdm` extension.
2. Each program must start with a `Chai` (class) within which code is considered its “ingredients”, so to speak.
3. Chais must have Masalas (functions), no matter how short the program is.
4. To create a package (like in Java), use the `Kahwa` keyword. Similarly, to create an interface, use the `ChaiMold` keyword.
5. To import a Chai, use the `Steep` keyword. Similarly, to export a Chai use the `Serve` keyword.
6. Cardamom, like Java, uses {} to denote the beginning and end of classes, functions, loops, and the like.
7. Chais may have a `Masala main()` function to run a program, unless the Chai’s functions are used elsewhere and not within itself.

This example:

```
Kahwa ChaiBrew {
  Public Chai ChaiMaker{
      Masala makeChai() as void {
          Fill AromaSet<String> ingredients = new AromaSet~

          ingredients.insert("Hot water")~
          ingredients.insert("Black tea leaves")~
```

```
            ingredients.insert("A pinch of cardamom")~
            ingredients.insert("A dash of ginger")~
            ingredients.insert("A sprinkle of cinnamon")~
            ingredients.insert("Milk")~
            ingredients.insert("A spoonful of sugar")~

            Pour("Preparing chai...")~
            Fill int i = 0~
            while (i < ingredients.size) {
                Fill String ingredient = ingredients.get(i)~
                Pour("Adding: " + ingredient)~
                i = i + 1~
            }
            Pour("Stirring. . .")~
            Pour("Your Cardamom Chai is ready!")~
        }

        Public Masala main() as void {
           Fill ChaiMaker chaiMaker = new ChaiMaker~
           useMasala(chaiMaker.makeChai())~
        }
    }
}
```

declares a `Chai` (class) named `ChaiMaker` in a `Kahwa` (package) called `ChaiBrew`. The fully qualified name of this class is `ChaiBrew.ChaiMaker`. The `ChaiMaker` class is responsible for simulating the process of making chai. It contains a `Masala` (function) named `makeChai()`, which further contains an `AromaSet` (array/table) of `Strings` named `ingredients`. The `makeChai()` Masala adds ingredients for a traditional chai recipe into the table and then retrieves them using a Java-esque `while` loop to simulate the preparation process. To work with the AromaSet, built-in functions such as `insert()`, `size()`, and `get()` are used. The `ChaiMaker` class also has a `main` `Masala` (function) which serves as the program's entry point, creating a new instance of the `ChaiMaker` class and calling its `makeChai()` Masala to begin the simulation.



1.4. Types and Variables

Like Java, there are two kinds of types in Cardamom: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. See Section 3 for details.

1.5. Visibility

Cardamom has two choices for access modifiers: `Public` and `Private`. Chais and Masalas can be either, depending on whether one plans to use its functions outside of the class. A public Chai's Masalas are automatically considered to be public as well, unless specified. However, similar to Java, only inner Chais (classes) can be private. Private Chais (classes) can have both public or private members, but they can only be accessed within itself and its parent Chai (class).

1.6. Statements Differing from Java and BASIC

Statement	Example

Expression statement	`Masala CloveCounter() as void {`
	`  Fill int cloves = 3~`
	`  Fill int clovesNeeded = 8 - cloves~`
	`  pour("We need " + clovesNeeded + " more cloves.")~`
	`  add(cloves, 5)~`
	`  pour("Great! Now we have " + cloves + " cloves.")~`
	`}`

`if` statement	`Masala sellCakes(size as int; price as float) as void {`
	`  if (size < 20) {`
	`  Fill price = 10.99~`
	`  } orMaybe (size > 20 && size < 35) {`
	`  Fill price = 24.99~`
	`  } else {`
	`  Fill price = 34.99~`
	`  }`
	`}`

While loop

```
Masala DirtyDishes(numDishes as int) as void {
    while (numDishes != 0) {
        pour("Washing the current dish...")~
        sub(numDishes, 1)~
        pour("There are " + numDishes + " dishes left")~
        if (numDishes != 0) {
            pour("KEEP WASHING THOSE DISHES!")~
        }
    }
}
```

For loop

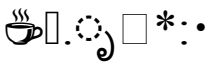
```
^-^ For loops are written using while loops
Masala printNumbers(max as int) as void {
    Fill int i = 0~
    while (i < max) {
        Pour(i)~
        i = add(i, 1)~
    }
}
```

Comments

```
()> This is a delimited comment which spans multiple
 > lines.
 > Parameters: price as int, amountPaid as int
 > Returns: change as int
/()>
Masala CalculateChange(price as int; amountPaid as int)
as void
{
    if (price < amountPaid){
        Fill int change = sub(amountPaid, price)~
    }
    return change~ ^-^ this is a single line comment
}
```

Recursion

```
Masala findSum(n as int) as int {
    if (n == 1) {
        return 1~
    }
    return add(n, useMasala(findsum(n - 1)))~
}
```



2. Lexical structure

2.1. Programs

A Cardamom ***program*** consists of one or more ***source files***. A source file is an ordered sequence of Unicode characters.

Conceptually speaking, a program is compiled using three steps:

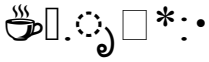
1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the Cardamom programming language where it differs from Java and BASIC.

2.2.1 Lexical grammar (tokens) where different from Java and BASIC

⟨statement_terminator⟩	→	'~'
⟨variable_declaration_keyword⟩	→	'Fill'
⟨function_declaration_keyword⟩	→	'Masala'
⟨return_type_clause⟩	→	'as' ⟨type⟩
⟨else_if_conditional⟩	→	'orMaybe'
⟨single_line_comment⟩	→	'^–^' ⟨comment_text⟩
⟨delimited_comment⟩	→	'( )⟩' ⟨comment_text⟩ '/( )⟩'



2.2.2 Syntactic ("parse") grammar where different from Java and BASIC

⟨class_declaration⟩ → 'Chai' ⟨identifier⟩

⟨function_declaration⟩ → 'Masala' ⟨identifier⟩ '('⟨parameter_list⟩')'⟨return_type_clause⟩ '{'⟨statement_list⟩ '}'

⟨variable_declaration⟩ → 'Fill' ⟨type⟩ ⟨identifier⟩ '=' ⟨expression⟩ '~'

⟨conditional_statement⟩ → 'if' '(' ⟨expression⟩ ')' '{' ⟨statement_list⟩ '}'

| 'if' '(' ⟨expression⟩ ')' '{' ⟨statement_list⟩ '}' 'orMaybe' '('⟨expression⟩ ')' '{' ⟨statement_list⟩ '}'

| 'if' '(' ⟨expression⟩ ')' '{' ⟨statement_list⟩ '}' 'orMaybe' '(' ⟨expression⟩ ')' '{' ⟨statement_list⟩ '}' 'else' '{' ⟨statement_list⟩ '}'

2.3 Lexical analysis

2.3.1 Comments

Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the characters `^-^` and extend to the end of the source line. Notice how this combination of characters creates a cartoon-like "face". *Delimited comments* start with the characters `()>` and end with the characters `/()>`. This combination of characters is supposed to resemble tea cups. Delimited comments may span multiple lines. Comments do not nest.

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

Tokens:

❀ identifier
❀ keyword
❀ integer-literal
❀ float-literal
❀ real-literal
❀ character-literal
❀ string-literal
❀ operator-or-punctuator

2.4.1 Keywords different from Java or BASIC

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.

New Keywords	Removed/Replaced Keywords
• Kahwa • Chai • Masala • pour • orMaybe • AromaSet • Steep • Serve • Fill • ChaiMold	• package • class • System.out.print() • else-if • array • import • exports • interface

☕.◌) *:・

3. Type System

Cardamom uses a **strong static** type system. Strong typing means that type errors are caught and expressed to the programmer during compilation. Static typing means early binding compile-time type checking. If the programmer forgets to specify the type, an error will occur urging them to specify types wherever missing. There is no type inference, nor any illusion of dynamic typing unlike in BASIC (which allows `def type` instead of a specific type).

3.1 Type Rules

The type rules for Cardamom are as follows:

```
S ⊢ e1: T
S ⊢ e2: T
T is a primitive type
-------------------------
S ⊢ e1 = e2: T

S ⊢ e1: T
S ⊢ e2: T
T is a primitive type
-------------------------
S ⊢ e1 == e2: boolean

S ⊢ e1: T
S ⊢ e2: T
T is a primitive type
-------------------------
S ⊢ e1 != e2: boolean

S ⊢ e1: T
S ⊢ e2: T
T is a primitive type
-------------------------
S ⊢ e1 < e2: boolean

S ⊢ e1: T
S ⊢ e2: T
T is a primitive type
-------------------------
S ⊢ e1 > e2: boolean

S ⊢ e1: String
S ⊢ e2: String
-------------------------
S ⊢ e1 + e2: String

S ⊢ e1: int
S ⊢ e2: int
-----------------
S ⊢ e1 + e2: int

S ⊢ e1: int
S ⊢ e2: int
-----------------
S ⊢ e1 - e2: int

S ⊢ e1: int
S ⊢ e2: int
-----------------
S ⊢ e1 * e2: int

S ⊢ e1: int
S ⊢ e2: int
-----------------
S ⊢ e1 / e2: int
```

Cardamom types are divided into two main categories: *Value types* and *Reference types*.

3.2 Value types (differing from Java and BASIC)

The value types present in Cardamom are the same as in Java.

3.3 Reference types (differing from Java and BASIC)

Chai – Similar to a Java "class". It serves as a blueprint for creating objects, defining their properties and behaviors, much like a recipe defines the ingredients and steps for making a specific type of tea.

ChaiMold – Similar to a Java "interface". It defines a contract for Chais (classes), specifying a set of Masalas (functions) that any Chai implementing it must provide, ensuring consistency in behavior across different Chai types.

AromaSet – Similar to a Java "array". It provides a way to store and access multiple values of the same type, organized as either a one-dimensional or a multi-dimensional grid, allowing you to work with complex data sets, like the flavor profiles in a delicious chai blend.

4. Example Programs

4.1 Caesar Cipher encrypt and decrypt

```
Public Chai CaesarCipher {
    ()> Encrypt a given message by shifting each letter a specific
    * number of times.
    * Parameters: Message as String, Shift as int
    * Returns: encryptedMessage as String
    /()>
    Masala Encrypt(Message as String; Shift as int) as String {
        Fill int normalized = ((Shift % 26) + 26) % 26~

        Fill encryptedMessage = Message.map { seed =>
         ^-^ seed is the keyword for character
            if (seed.isUpper) {
            ((seed - 'A' + normalized) % 26 + 'A').toSeed~
            } orMaybe (seed.isLower) {
            ((seed - 'a' + normalized) % 26 + 'a').toSeed~
            } else {
            seed~
            }
        }.join()~ ^-^ Join the seeds back into a string

        return encryptedMessage~
    }

    ()> Decrypts a given message by shifting each letter a given shift
     * amount.
     * Parameters: Message as String, Shift as int
     * Returns: decryptedMessage as String
     /()>
     Masala Decrypt(Message as String; Shift as String) as String {
        Fill String decryptedMessage = useMasala(Encrypt(Message, -Shift))~
     }
}
```

4.2 Factorial

```
Public Chai Arithmetic {
    Masala Factorial(int num) as int {
        Fill int result = 1~
        Fill int i = 2~
        ^-^ Let's recreate a for loop using a while loop
        while (i <=n) {
            ^-^ Multiplies result by i and sets solution to result
            multiply(result, i)~
            add(i, 1)~
        }
        return result~
    }
}
```

4.3 Bubble Sort (Swap function included)

```
Public Chai BubbleSort {
    Masala bubbleSort(input as AromaSet<int>) as AromaSet<int> {
        Fill boolean swapping = true~
        while (swapping) {
            Fill swapping = false~
            Fill int i = 0~
            while (i < input.size - 1) {
                if (input[i] > input[i+1]) {
                    swap(input, i, i+1)~
                    swapping = true~
                }
                add(i, 1)~
            }
        }
        return input~
    }
    Masala swap(array as AromaSet<int>; iOne as int; iTwo as int) as void {
        Fill int temp = array[iTwo]~
        Fill array[iTwo] = array[iOne]~
        Fill array[iOne] = temp~
    }
}
```

4.4 QuickSort

```
Public Chai QuickSort{
    Masala quicksort(array as AromaSet<int>; start as int; end as int) as AromaSet<int>
    {
        if (start == end) {
            return array~
        }

        Fill int index = partition(array, start, end)~

        if (start < index - 1) {
            useMasala(quicksort(array, start, index - 1))~
        }

        if (index < end) {
            useMasala(quicksort(array, index, end))~
        }
        return array~
    }


    Masala partition(array as AromaSet<int>;  left as int; right as int) as int {
        Fill int pivot = array[Calc.floorDiv((left + right), 2)]~

        while (left <= right) {
            while (array[left] < pivot) {
                add(left, 1)~
            }

            while (array[right] > pivot) {
                sub(right, 1)~
            }

            if (left <= right) {
                swap(array, left, right)~
                add(left, 1)~
                add(right, 1)~
            }
        }
        return left;
    }
}
```

4.5 MergeSort

```
Public Chai MergeSort {
    Masala mergeSort(a as AromaSet<int>; n as int) as void {
           if (n < 2) {  ^-^ Base Case
               return~
           }

           Fill int mid = n / 2~
           Fill AromaSet<int> l = new AromaSet[mid]~
           Fill AromaSet<int> r = new AromaSet[n - mid]~

           Fill int i = 0~
           while (i < mid) {
               Fill l[i] = a[i]~
            add(i, 1)~
           }

           Fill int j = mid~
           while (j < n) {
               Fill r[j - mid] = a[j]~
            add(j, 1)~
           }

           useMasala(mergeSort(l, mid))~
           useMasala(mergeSort(r, n - mid))~

           useMasala(merge(a, l, r, mid, n - mid))~
   }

   Masala merge(
           a as AromaSet<int>; l as AromaSet<int>; r as AromaSet<int>; left as int;
            right as int) as void {

           Fill int i = 0~
           Fill int j = 0~
           Fill int k = 0~
           while (i < left && j < right) {
              if (l[i] <= r[j]) {
                     a[k+1] = l[i+1]~
              } else {
                     a[k+1] = r[j+1]~
              }
          }
          while (i < left) {
              a[k+1] = l[i+1]~
          }
          while (j < right) {
              a[k+1] = r[j+1]~
          }
   }

}
```