```
                     (
                       )     (
                ___...(-------)-....___
            .-""       )    (          ""-.
      .-'``'|-._             )         _.-|
     /  .--.|   `""---...........---""`   |
    /  /    |                             |
    |  |    |                             |
     \  \   |                             |
      `\ `\ |   World’s Best Programmer   |
        `\ `|                             |
        _/ /\                             /
       (__/  \                           /
    _..---""` \                         /`""---.._
 .-'           \                       /          '-.
:               `-.__             __.-'              :
:                  ) ""---...---"" (                 :
 '._               `"--...___...--"`              _.'
   \""--..__                              __..--""/
    '._     """----.....______.....----"""     _.'
       `""--..,,_____            _____,,..--""`
                     `"""----"""`
```

liteRoast

A Language Design Project for CMPT330L

Andrew Hatch

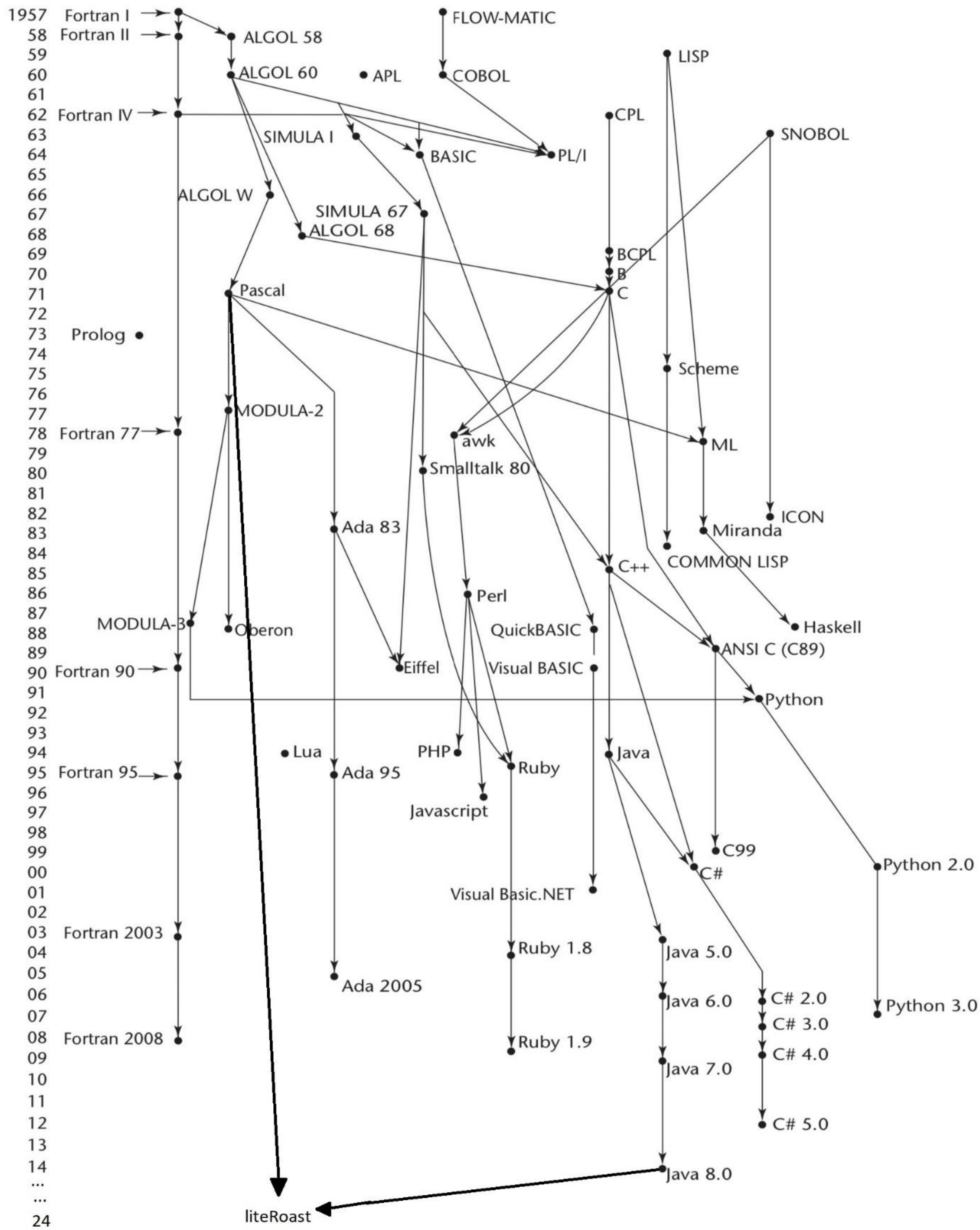
1.Introduction

liteRoast is a simple(ish) type-safe programming language, created to incorporate elements from both Java and Pascal in order to make a more readable ("lite") blend of both. Does it accomplish this? Not perfectly – but it was at least fun to plan out.

liteRoast differs from its forefathers in the following ways:

1. Approximately 300% more "java"-related functionality - Keywords like int, string, array have been replaced by more fitting terms (*see sections 2.2.1 - 2.2.2 for more information*).
2. Functions, Classes and Objects must be declared with the "brew" operator placed before them, taken as inspiration from Python's decl.
3. All functions and classes (whether public or private) must be declared with their respective identifiers:
    a. "lite" for public.
    b. "dark" for private.
4. Variable assignment is done through the ":=" operator.
5. Primitive number types (int, float, double) have been consolidated under the "cream" keyword. All cream variables are compatible with each other and can be modified freely.
6. Several other formatting-related changes (*refer to section 1.3 for more information*)

1.1.Genealogy



1.2.Hello world

```
brew Order HelloWorld
pour
    brew lite void main()
    pour
        receipt("Hello World!");
    sip
sip
```

1.3.Program structure

The key organizational concepts in liteRoast are as follows:

1. Brackets ("{}") have been replaced by "pour" and "sip" for opening and closing blocks, respectively. These do not need the usual end-of-line terminating symbol to work.
    a. "for" and "while" functions do not require a "pour" token to work, instead needing a "do" token immediately following the expression. They *definitely* need a "sip" token to terminate, though.
2. Borrowing from Pascal, liteRoast methods require a "toppings" block where variables are initialized, placed before the driver code. Variables can ***only*** be declared in these blocks, but can be used in other methods so long as they are called properly and are public.
3. At least one Order ("class") must exist in each program, borrowed from Java. Orders can coexist peacefully in the brewing process so long as they do not share the same identifier.
4. A main function must be declared within every Order.

An example program:

```
dark order ventiCaramel
pour
    brew lite iced getTheOrder(customerName, number)
    toppings
        cream i;
        blend sugar custList;
    pour
        custList := {"Abigail Owens", "James Bond", "John Smith", "Star Bucks"};
        for (i := 0 to custList.flavor()) do
            if (custList(i) == customerName) do
                return true;
            sip
            else
                return false;
            sip
        sip
    sip

    brew lite void main()
    toppings
        sugar firstName;
        sugar lastName;
        cream ID;
        iced validCustomer;

    pour
        receipt("Here's our latest customer...");
        validCustomer := getTheOrder("John Smith", 31); \/D call our other method to validate customer
        receipt("Is our customer registered? " + validCustomer);
    sip
sip
```

Declares a new Order named ventiCaramel using the "brew" keyword, which includes two methods: getTheOrder and our main method. getTheOrder contains parameters which accepts two variables of type String and Int, respectively., and returns a Boolean value true/false.

1.4.Types and Variables

There are two kinds of types in liteRoast: ***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

liteRoast supports both Public and Private visibility for functions, with the keyword "lite" used to denote Public functions, and "dark" for Private. Variables are also subject to this visibility, but are assumed Public until otherwise specified.

1.6.Statements Differing from Java and Pascal

Statement	Example
Expression statement	brew Order expressionEx pour brew lite void main() toppings syrup firstInitial; syrup secondInitial; sugar fullName; pour firstInitial := “A”; secondInitial := “H”; fullName := firstInitial + secondInitial; receiptln(fullName); sip sip
`if` statement	brew Order ifEx pour brew lite void main() Toppings Cream firstNum; Cream secondNum; pour firstNum := 5; secondNum := 7; if(firstNum == secondNum) then receiptln(“first number is equal to second”); else firstNum := 7; receiptln(“first number is NOT equal to second”); sip sip
For statement	brew Order forEx pour brew lite void main() toppings blend S; cream i; pour numbers := [1, 2, 3, 4, 5, 8, 9, 10]; for (i := 1 to S.flavor()) do receiptln(“Current number: “ + numbers[i]); sip sip sip

<table>
<tr><td>While statement</td><td>

```
brew Order whileEx
pour
    brew lite void main()
    toppings
        Grounds newNumbers;
        Cream i;
     pour
         i := 1;
         newNumbers := [10, 20, 30, 40, 20, 60, 80];
         While (newNumbers[i] != 60) do
             receiptln("Current number is " + newNumbers[i]);
             i++;
          sip
     sip
sip
```

</td></tr>
<tr><td>Comments</td><td>

```
Brew Order wheresthefiller
Pour
   Brew lite void main()

   \/D~ Toppings to initialize our ingredients
   We're still in a comment here, so I can say whatever I want!
   I love coffee ~\/D
   toppings
       cream cawfee;
       cream greentea;
    pour
       cawfee := 20; \/D initialize cawfee to 20
       greentea = 30; \/D initialize green tea to 30

       \/D~ If we've got more coffee, print a message
       else say we've got more green tea
       ~\/D
       if (cawfee > greentea) do
           receiptln ("I'd like some more coffee");
       else
           receiptln("I'd like some more green tea");
    sip \/D end our main function
sip
```

</td></tr>
</table>

2.Lexical structure

2.1.Programs

A liteRoast program consists of one or more ***source files***. A source file is an ordered sequence of (probably 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 liteRoast programming language where it differs from Java and Pascal.

2.2.1.Lexical grammar (tokens) where different from Java and Pascal

<Assignment Operator> → :=

<Math Operator> → + | * | / | -

<Print> → receipt() | receiptln()

<Boolean Operator> → == | != | <= | >=

<Open Block> → pour

<Close Block> → sip

<Single-Line Comment> → \/D

<Delimited Comment> → \/D~ | ~\/D

<int, float, double> → cream

<string> → sugar

<char> → syrup

<array> → blend

<arraylength> → flavor

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

<Order declaration> → Order <Identifier>

<Function Declaration> → brew <visibility> <object type> <identifier> <parameter list> | brew <visibility> <object type> <identifier>

<Blend declaration> → blend <object type> <identifier>

<Parameter> → <object type> <identifier>

2.3.Lexical analysis

2.3.1.Comments

liteRoast supports two forms of comments: single-line comments and delimited comments.

- ***Single-line comments*** start with the characters `\/D` (representing a nice cup of joe) and extend to the end of the source line.
- ***Delimited comments*** start with the characters `\/D~` and end with the characters `~\/D`. Delimited comments may span multiple lines.

Comments do not nest – attempting to do so will ruin the drink.

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
  cream-literal
  syrup-literal
  sugar-literal
  iced-literal
  operator-or-punctuator

2.4.1.Keywords different from Java and Pascal

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

New keywords:

```
Order, brew, pour, sip, lite, dark, iced, sugar, cream, syrup,
receipt, flavor, blend
```

Removed keywords:

```
   Class, new, Public, Private, begin, end, bool, string, int, float,
double, char, print, println, length, size, array
```

3.Type System

liteRoast uses a **strong static** type system – variables are declared as a specific type and cannot be transmuted to any other type. Strong typing means that type errors are caught and expressed to the programmer during compilation. Static typing means early binding compile-time type checking.

3.1.Type Rules

The type rules for liteRoast are as follows:

$S \vdash e_1: T$

$S \vdash e2: T$

T is a Primitive type

$S \vdash e1 := e2: T$

$S \vdash e_1: T$

$S \vdash e2: T$

T is a Primitive type

$S \vdash e1 \mathrel{!=} e2: \text{boolean}$

$S \vdash e_1: T$

$S \vdash e2: T$

T is a Primitive type

$S \vdash e1 == e2: \text{boolean}$

$S \vdash e_1: T$

$S \vdash e2: T$

T is a Primitive type

$S \vdash e1 \mathrel{!=} e2: \text{boolean}$

$S \vdash e_1: T$

$S \vdash e2: T$

T is a Primitive type

$S \vdash e1 == e2: \text{Boolean}$

$S \vdash e_1: T$

$S \vdash e2: T$

T is a Primitive type

$S \vdash e1 < e2: \text{boolean}$

$S \vdash e_1 : T$

$S \vdash e2 : T$

T is a Primitive type

$S \vdash e1 * e2 : T$

$S \vdash e_1 : T$

$S \vdash e2 : T$

T is a Primitive type

$S \vdash e1 + e2 : T$

$S \vdash e_1 : T$

$S \vdash e2 : T$

T is a Primitive type

$S \vdash e1 > e2 : boolean$

$S \vdash e_1 : T$

$S \vdash e2 : T$

T is a Primitive type

$S \vdash e1 / e2 : T$

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

3.2.Value types (different from Java and Pascal)

Cream – any real number, positive or negative. Decimals points are considered when combining cream variables (ex. Adding 1 + 3.5 would have liteRoast treat the first cream as 1.0).

Syrup – a single Unicode character. Able to be concatenated to another syrup variable or to any available sugar variables.

Iced – Do we want our coffee served hot or with ice? Binary value that returns true or false depending on the customer's specifications.

3.3.Reference types (differing from Java and Pascal)

Sugar – Sequence of syrup variables with a non-fixed length. Able to be concatenated through other sugar or syrup variables.

Blend – container object that holds a fixed amount of variables of a certain type. Since the art of brewing is a dangerous and highly precise practice, the specific blend of ingredients we initialize is fixed length and cannot be changed.

4.Example Programs

1. Caesar Cipher encrypt / decrypt

```
brew lite Order caeserCipher
pour
    brew lite sugar caeserEncrypt(newCipher, shift)
    toppings
        cream i;
    pour
        for (i:= 1 to newCipher.flavor()) do
            \/D Check for uppercase first
            'A'..'Z': newCipher[i] := chr(ord('A') + (ord(newCipher[i]) - ord('A') + shift) mod 26);

            \/D Then check for lowercase
            'a'..'z': newCipher[i] := chr(ord('a') + (ord(newCipher[i]) - ord('a') + shift) mod 26);
        sip
    return newCipher;
    sip

    brew lite sugar caeserDecrypt(newCipher, shift)
    pour
        newCipher := caeserEncrypt(newCipher, shift * -1);
        return newCipher;
    sip

    brew lite void main()
    toppings
        sugar newCipher;
        cream shift;
    pour
        receipt("Caeser Encrypt and Decrypt");
        newCipher := "Everyone likes Starbucks, Im more of a Dunkin kind of guy";
        newCipher := caeserEncrypt(newCipher, shift);
        receipt("Encrypted Cipher is: " + newCipher);

        newCipher := caeserDecrypt(newCipher, shift);
        receipt("Decrypted Cipher is: " + newCipher);
    sip
sip
```

2. Factorial

```
brew lite Order factorial
pour
    brew lite cream factorial(number)
    toppings
        cream factpuccinorial;
        cream i;
    pour
        factpuccinorial := 1;
        for (i := 2 to number) do
            factpuccinorial := factpuccinorial * i;
        sip
        return factpuccinorial;
    sip

    brew lite void main()
    toppings
        cream num;
    pour
        num := 7;
        receipt("The factorial of 7 is " + factorial(num));
    sip
sip
```

3. Bubble Sort

```
brew lite Order bubbleSort
pour
    brew lite void bubbles(bubbleArr)
    toppings
        cream x
        cream y
        cream temp
    pour
        for (x := 0; x < bubbleArr.flavor() - 1; x++) do
            for (y := 0; y < bubbleArr.flavor() - 1; y++) do
                if (bubbleArr[x] > bubbleArr[x + 1]) do
                    temp := bubbleArr[y];
                    bubbleArr[y] := bubbleArr[y + 1];
                    bubbleArr[y + 1] := temp;
                sip
            sip
        sip
        return bubbleArr;
    sip

    brew lite void main()
    toppings
        blend cream bubbleArr;
        blend cream i;
        cream j;
    pour
        bubbleArr := {1, 2, 4, 6, 7, 3, 8, 10, 9}

        i := bubbles(bubbleArr);

        for (j := 0 to i.flavor()) do
            receipt(i[j]); \/D prints out our sorted array
        sip
    sip
sip
```

4. Insertion Sort

```
brew lite Order insertionSort
pour
    brew lite cream insert(insertArr, i)
    toppings
        cream j;
        cream x;
        cream y;
    pour
        for (j := 1 to i) do
            y := insertArr[j];
            x := j - 1;

            while (j >= 0 && insertArr[x] > y) do
                insertArr[x + 1] = insertArr[x];
                x := x - 1;
            sip
            insertArr[x + 1] := y;
        sip
        return insertArr;
    sip

    brew lite void main()
    toppings
        blend insertArr;
        blend sortedInsert;
        cream i;
        cream n;
    pour
        insertArr := {1, 2, 4, 6, 7, 3, 8, 10, 9};
        i := insertArr.flavor();

        sortedInsert := insert(insertArr, i);

        for (n := 0 to i) do
            receipt("Sorted ingredients are: " + sortedInsert[n]);
        sip
    sip
sip
```

5. Quick Sort

```
brew lite Order quickSort
pour
    brew lite cream partitioning (quickArr, mild, strong)
    toppings
        cream pivot;
        cream j;
        cream x;
        cream temp;
    pour
        x := mild - 1;
        for(j := mild to strong - 1) do
            if (quickArr[j] < pivot) do
                temp := quickArr[x];
                quickArr[x] := quickArr[j];
                quickArr[j] := temp;
                j++;
                x++;
            sip
        sip
        return quickArr[j + 1];
    sip


    brew lite blend quick (quickArr, mild, strong)
    toppings
        cream pivot;
        blend superBean;
    pour
        pivot := quickArr[strong];
        if(mild < strong) do
            pivot := partitioning(quickArr, mild, strong);
            superBean := quick(quickArr, mild, pivot - 1);
            superBean := quick(quickArr, pivot + 1, strong);
        sip
        return superBean;
    sip

    brew lite void main()
    toppings
        blend quickArr;
        blend sortedQuickly;
        cream i;
        cream n;
    pour
        quickArr := {1, 2, 4, 6, 7, 3, 8, 10, 9};
        i := quickArr.flavor();

        sortedQuickly := quick(quickArr, 0, i - 1);

        for (n := 0 to i) do
            receipt("Sorted ingredients are: " + sortedQuickly[n]);
        sip
    sip
sip
```