Compiler construction language design

Krieolin Navindran Naidoo[219003234]

University of Johannesburg, Auckland Park, 5 Kingsway Ave, Rossmore, Johannesburg, 2092, South Africa
krieolin@gmail.com
219003234@student.uj.ac.za

Abstract. This document contains the grammatical analysis of the jollyjam language. It delves into the language overview which covers factors such as the languages influence and structure, sample programs and token definitions are also included.

Table of contents

1. Language description - Language overview

JollyJam or “JJ” for short is a simplified programming language that gains inspiration from the Baker’s biscuit Jolly Jammers. The biscuit is essentially a raspberry and vanilla filling sandwiched by two smiley face biscuits, all simple ingredients combined to create something fun and enjoyable for all people and ages, a delightful treat that can even be baked at home. This ideology is what the JollyJam programming language tries to achieve, simple aspects working together to create something great that can be picked up relatively easily like following a recipe to bake a biscuit. The major driving factor that determined the grammar for the language was based on feedback from non-programmers. Essentially what I did was talk to a few people who have no programming experience and described the problem and asked them which command or grammar was easier for them to pick up on, because for someone with programming experience it is relatively easier to learn a language but for someone without any prior knowledge it is difficult to understand programming concepts which is why user input from none programmers was an important driving factor. As a result majority of the grammar is in line with baking terms like “recipe”, “method”, “do”.

The JollyJam language takes inspiration from various areas from people’s input to programming languages already in existence such as in the procedural and imperative programming language families, like the BASIC language which utilizes English-like commands which are easier to understand and remember for the user, however even though it takes inspiration from BASIC which is a procedural programming language JollyJam is a imperative programming language where the user would control the programs steps.

The language supports branching flow control via if statements, while also supporting common loop structures such as while and for loops. The language has a clear type system that is enforced by the compiler, however at the machine code stages on the lower level some types may become the same after its context in analysed.

Basic data types that the language supports includes numbers such as integers and floating point numbers, characters such as string literals and Booleans which indicate a true or false. There is no support for complex data types such as objects which is more commonly seen in object oriented programming languages. There is no support for custom operators, however string concatenation is possible using the + operand. Each complete statement has to end with a semi-colon ;

JollyJam has support for user-defined functions for both return types (functions that return something such as an integer or string). The language allows for functions to be used before they are defined, so the functions implementation can be before or after the function call, however the function call should be called with a “@” symbol so for example if the functions name is Jump() the call for the function would be @Jump(). Recursive functions are allowed.

There is no support for pointers or direct memory access so memory manipulation cannot be addressed directly. As such all parameters are passed by value. Every runnable program must include a main function as this will act as the starting point of the program, however the main function does not take in any command line arguments. As the language was user feedback driven the program terms are all in capital as shown in the sample program sections.

2. Summarised in point form for easier referencing:

- **Inspiration and Language Category:**
  - JollyJam language draws inspiration from both procedural and imperative programming language families.
  - It takes cues from BASIC, using English-like commands for user-friendliness.
  - Despite BASIC influence, JollyJam is an imperative language where users control program steps.
- All program terms are capitalized.

- Flow Control and Looping:
  - JollyJam supports branching flow control using "if" statements.
  - It provides common loop structures: "while" and "for" loops.

- Type System and Compilation:
  - JollyJam enforces a clear type system through the compiler.
  - In low-level machine code, certain types may become the same due to context analysis.

- Data Types:
  - Supported basic data types: integers, floating-point numbers, string literals , and Booleans (true/false).
  - No Support for fixed-length arrays and complex data types like objects.

- Operators and Concatenation:
  - No support for custom operators.
  - String concatenation is possible using the "+" operand.

- User-Defined Functions:
  - JollyJam allows user-defined functions with return types (returning data or nothing).
  - Functions can be used before they're defined, denoted by "@FunctionName()".
  - Supports recursive functions.

- Memory and Pointers:
  - No support for pointers or direct memory access.
  - All parameters are passed by value.

- Main Function and Entry Point:
  - Every runnable JollyJam program must include a main function.
  - The main function doesn't accept any command line arguments.

3. Sample programs

```
**This is a basic hello world program that will ask for the users name and then print that name**
RECIPE HelloWorld
DO
   METHOD Main() AS INTEGER
   DO
      INGREDIENT name AS STRING;
      ASK(name);
      INGREDIENT message AS STRING;
      message = "Hello, " + name + "!";
      SPEAK(message);
      SHARE 0;
   DONE
DONE
```

```
**This program shows the basics of loops and if statements**

RECIPE CountToTen
DO
   METHOD Main() AS INTEGER
   DO
      INGREDIENT count AS INTEGER;
      count = 1;

      LOOP WHILE count <= 10 DO
         IF count == 3 THEN
            SPEAK("3 spoons of sugar");
         ELSE IF count == 5 THEN
            SPEAK("Five cups of flour");
         ELSE
            SPEAK(count);
         DONE
         count = count + 1;
      DONE

      SHARE 0;
   DONE
DONE
```

```
**This program shows the basics of function calls**

RECIPE FunctionCall
DO
   METHOD Main() AS INTEGER
   DO
      INGREDIENT num1 AS INTEGER;
      INGREDIENT num2 AS INTEGER;
      num1 = 5;
      num2 = 7;

      INGREDIENT result AS INTEGER;
      result = @AddNumbers(num1, num2);
```

```
        SPEAK("The sum is: " + result);
        SHARE 0;
    DONE

    METHOD AddNumbers(a AS INTEGER, b AS INTEGER) AS INTEGER
    DO
        SHARE a + b;
    DONE
DONE
```

```
** This program shows various math operations**

RECIPE MathOperations
DO
    METHOD Main() AS INTEGER
    DO
        INGREDIENT a AS INTEGER;
        INGREDIENT b AS INTEGER;
        a = 14;
        b = 2;

        INGREDIENT sum AS INTEGER;
        sum = a + b;
        INGREDIENT difference AS INTEGER;
        difference = a - b;
        INGREDIENT product AS INTEGER;
        product = a * b;
        INGREDIENT quotient AS INTEGER;
        quotient = a / b;

        SPEAK("Sum: " + sum);
        SPEAK("Difference: " + difference);
        SPEAK("Product: " + product);
        SPEAK("Quotient: " + quotient);
        SHARE 0;
    DONE
DONE
```

```
** This program shows string manipulation **

RECIPE StringManipulation
DO
    METHOD Main() AS INTEGER
    DO
        INGREDIENT name AS STRING;
        ASK(name);

        INGREDIENT greeting AS STRING;
        greeting = "Hello, " + name + "!";

        SPEAK(greeting);
    SHARE 0;
  DONE
DONE
```

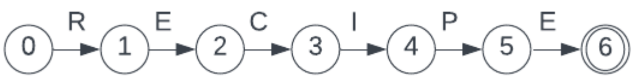
```
** This program shows logical operators **

RECIPE LogicalOperators
DO
  METHOD Main() AS INTEGER
  DO
    INGREDIENT x AS INTEGER;
    x = 7;

    IF x > 5 AND x < 10 THEN
      SPEAK("use 1 cup sugar");
    ELSE
      SPEAK("Don’t add sugar");
    DONE

    SHARE 0;
  DONE
DONE
```

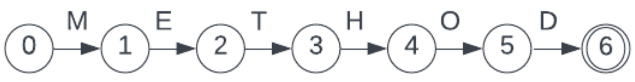
```
** This program shows booleans **

RECIPE BooleanExample
DO
  METHOD Main() AS INTEGER
  DO
    INGREDIENT isWet AS BOOL;
    isWet = TRUE;

    INGREDIENT isCake AS BOOL;
    isCake = FALSE;

    IF isWet THEN
      SPEAK("Add more flour");
    ELSE
      SPEAK("Bake");
    DONE

    IF isCake THEN
      SPEAK("Time to eat");
    ELSE
      SPEAK("no cake as yet :(");
    DONE

    SHARE 0;
  DONE
DONE
```

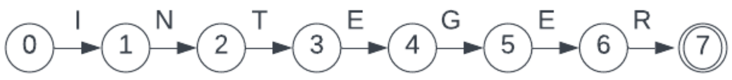
```
** This program shows the basic of a function call as well as arrays in use**
RECIPE FunctionCallWithArray
DO
```

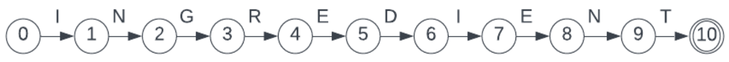
```
    METHOD Main() AS INTEGER
    DO
        INGREDIENT values[] AS STRING;
        values[0] = 1;
        values[1] = 2;
        values[2] = 3;

        INGREDIENT i AS INTEGER;
        I = 0;
        LOOP WHILE i < 2 DO
            IF values[i] == 1 THEN
                @Print("cake");
            DONE
            i = i + 1;
        DONE

        SHARE 0;
    DONE

    METHOD Print(message AS STRING) AS INTEGER
    DO
        SPEAK(message);
        SHARE 0;
    DONE
DONE
```

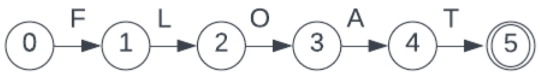
4. Token Definitions

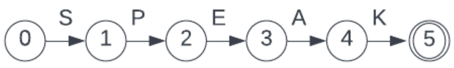
Token type : RECIPE	Description : Keyword “RECIPE”	Regular expression : RECIPE
 0 R 1 E 2 C 3 I 4 P 5 E 6		

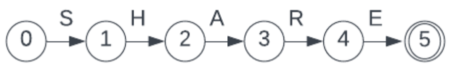
Token type : METHOD	Description : Keyword “METHOD”	Regular expression : METHOD
 0 M 1 E 2 T 3 H 4 O 5 D 6		

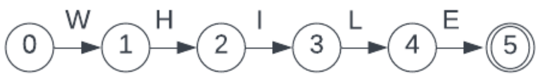
Token type : INTEGER	Description : Keyword “INTEGER”	Regular expression : INTEGER
		

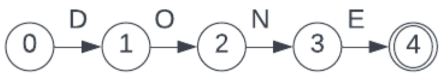
Token type : INGREDIENT	Description : Keyword “INGREDIENT”	Regular expression : INGREDIENT
		

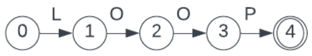
Token type : FLOAT	Description : Keyword “FLOAT”	Regular expression : FLOAT
		

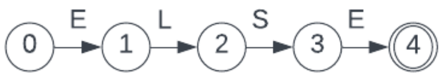
Token type : SPEAK	Description : Keyword “SPEAK”	Regular expression : SPEAK
		

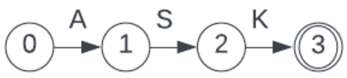
Token type : SHARE	Description : Keyword “SHARE”	Regular expression : SHARE
		

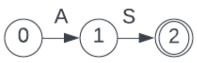
Token type : WHILE	Description : Keyword “WHILE”	Regular expression : WHILE
		

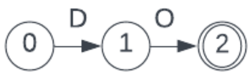
Token type : DONE	Description : Keyword “DONE”	Regular expression : DONE
		

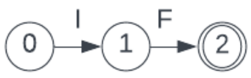
Token type : LOOP	Description : Keyword “LOOP”	Regular expression : LOOP
		

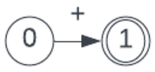
Token type : ELSE	Description : Keyword “ELSE”	Regular expression : ELSE
		

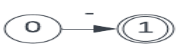
Token type : ASK	Description : Keyword “ASK”	Regular expression : ASK
		

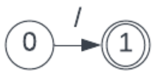
Token type : AS	Description : Keyword “AS”	Regular expression : AS
		

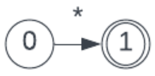
Token type : DO	Description : Keyword “DO”	Regular expression : DO
		

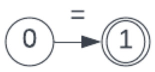
Token type : IF	Description : Keyword “IF”	Regular expression : IF
		

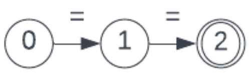
Token type : PLUS	Description : Keyword “ADDITION OPERATOR”	Regular expression : \+
		

Token type : MINUS	Description : Keyword “MINUS OPERATOR”	Regular expression : \-
		

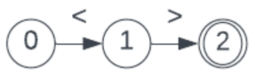
Token type : FORWARD SLASH	Description : Keyword “DIVISION OPERATOR”	Regular expression : /
		

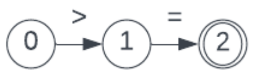
Token type : STAR	Description : Keyword “MULTIPLICATION OPERATOR”	Regular expression : *
		

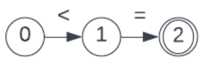
Token type : ASSIGN	Description : Keyword “ASSIGNMENT OPERATOR”	Regular expression : =
		

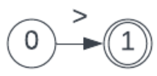
Token type : EQUAL	Description : Keyword “EQUALITY OPERATOR”	Regular expression : ==
		

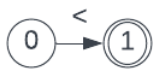
Token type : NEQ	Description : Keyword “INEQUALITY OPERATOR”	Regular expression : <>



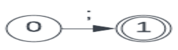
Token type : GREATEREQUAL	Description : Keyword “GREATER OR EQUAL”	Regular expression : >=
		

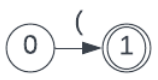
Token type : LESSEREQUAL	Description : Keyword “LESSER OR EQUAL”	Regular expression : <=
		

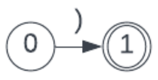
Token type : GREATER	Description : Keyword “GREATER”	Regular expression : >
		

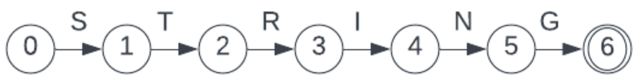
Token type : LESSER	Description : Keyword “LESSER”	Regular expression : <
		

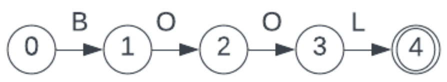
Token type :	Description :	Regular expression :

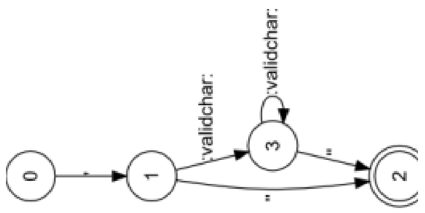
SEMICOLON	Keyword “SEMI COLON”	;
		

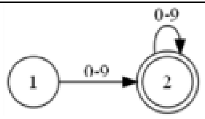
Token type : LEFTPARENTHESIS	Description : Keyword “LEFT PARENTHESIS”	Regular expression : (
		

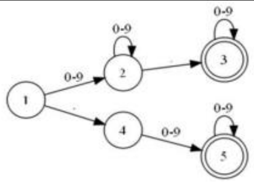
Token type : RIGHTPARENTHESIS	Description : Keyword “RIGHT PARENTHESIS”	Regular expression :)
		

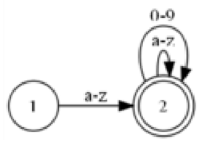
Token type : STRING	Description : Keyword “STRING”	Regular expression : STRING
		

Token type : BOOL	Description : Keyword “BOOLEAN”	Regular expression : BOOL
		

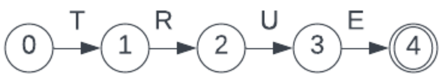
Token type : STRINGLIT	Description : Keyword “STRING LITERAL”	Regular expression : \”(:VALIDCHAR:)*\” WHERE VALIDCHAR = [a-z][A-Z][0-9][~!@#$%^&*\`(\)_\-+=\\|{\[}\];:<>,.?\\\]
		

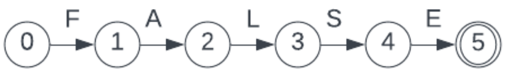
Token type : INTEGERLIT	Description : Keyword “INTEGER LITERAL”	Regular expression : [0-9]+
		

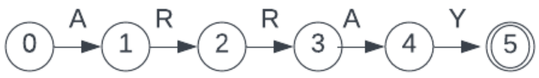
Token type : FLOATLIT	Description : Keyword “FLOAT LITERAL”	Regular expression : [0-9]*.[0-9]*
		

Token type : ID	Description : Keyword “IDENTIFIER”	Regular expression : ([a-z]|[A-Z])([a-z] | [A-Z] | [0 – 9])*
		

Token type :	Description :	Regular expression :

BOOLLITTRUE	Keyword “BOOLEAN LITERAL”	TRUE
		

Token type : BOOLLITFALSE	Description : Keyword “BOOLEAN LITERAL”	Regular expression : FALSE
		

Token type : ARRAY	Description : Keyword “ARRAY”	Regular expression : ARRAY
		

5. Language grammar – production rules

```
RECIPE ⟶ RECIPE ID DO METHOD_MAIN DONE
METHOD_MAIN ⟶ METHOD MAIN LEFTPARENTHESIS RIGHTPARENTHESIS DO STMNT_BLOCK DONE
STMNT_BLOCK ⟶ STMNT STMNT_BLOCK
STMNT_BLOCK ⟶ ε
STMNT ⟶ ASK LEFTPARENTHESIS ID RIGHTPARENTHESIS SEMICOLON
STMNT ⟶ SPEAK LEFTPARENTHESIS STRINGLIT RIGHTPARENTHESIS SEMICOLON
STMNT ⟶ SHARE INTEGERLIT SEMICOLON
STMNT ⟶ ID ASSIGN EXPR SEMICOLON
STMNT ⟶ ID ASSIGN ARRAY_ACCESS SEMICOLON
STMNT ⟶ IF EXPR THEN STMNT_BLOCK ELSE_BLOCK DONE
ELSE_BLOCK ⟶ ELSE STMNT_BLOCK
ELSE_BLOCK ⟶ ε
STMNT ⟶ FUNCTION_CALL SEMICOLON
EXPR ⟶ TERM EXPR_PRIME
EXPR_PRIME ⟶ PLUS TERM EXPR_PRIME
EXPR_PRIME ⟶ MINUS TERM EXPR_PRIME
EXPR_PRIME ⟶ AND TERM EXPR_PRIME
EXPR_PRIME ⟶ GREATER TERM EXPR_PRIME
EXPR_PRIME ⟶ GREATEREQUAL TERM EXPR_PRIME
EXPR_PRIME ⟶ LESSER TERM EXPR_PRIME
EXPR_PRIME ⟶ LESSEREQUAL TERM EXPR_PRIME
EXPR_PRIME ⟶ EQUAL TERM EXPR_PRIME
EXPR_PRIME ⟶ NEQ TERM EXPR_PRIME
EXPR_PRIME ⟶ ε
TERM ⟶ FACTOR TERM_PRIME
TERM_PRIME ⟶ STAR FACTOR TERM_PRIME
TERM_PRIME ⟶ FORWARD_SLASH FACTOR TERM_PRIME
TERM_PRIME ⟶ AND FACTOR TERM_PRIME
TERM_PRIME ⟶ GREATER FACTOR TERM_PRIME
TERM_PRIME ⟶ GREATEREQUAL FACTOR TERM_PRIME
TERM_PRIME ⟶ LESSER FACTOR TERM_PRIME
TERM_PRIME ⟶ LESSEREQUAL FACTOR TERM_PRIME
TERM_PRIME ⟶ EQUAL FACTOR TERM_PRIME
TERM_PRIME ⟶ NEQ FACTOR TERM_PRIME
TERM_PRIME ⟶ ε
FACTOR ⟶ ID
FACTOR ⟶ INTEGERLIT
FACTOR ⟶ STRINGLIT
FACTOR ⟶ ARRAY_ACCESS
FACTOR ⟶ FUNCTION_CALL
ARRAY_ACCESS ⟶ ID LEFTBRACKET EXPR RIGHTBRACKET
FUNCTION_CALL ⟶ AT_SIGN ID LEFTPARENTHESIS ARGUMENT_LIST RIGHTPARENTHESIS
ARGUMENT_LIST ⟶ EXPR ARGUMENT_LIST_PRIME
ARGUMENT_LIST_PRIME ⟶ COMMA EXPR ARGUMENT_LIST_PRIME
ARGUMENT_LIST_PRIME ⟶ ε
TYPE ⟶ INTEGER
TYPE ⟶ FLOAT
TYPE ⟶ STRING
TYPE ⟶ BOOLEAN
```

6. Meaning of terms

RECIPE: Represents the start of a recipe block (program,).
METHOD_MAIN: Represents the main method within a recipe.
STMNT_BLOCK: Represents a block of statements.
STMNT: Represents a statement.
ASK: Represents the input command to ask for user input.
SPEAK: Represents the command to output text.
SHARE: Represents sharing a value with the environment.
ID: Represents an identifier (variable name).
ASSIGN: Represents the assignment operator (=).
IF: Represents the start of an if statement.
THEN: Represents the "then" keyword in an if statement.
ELSE_BLOCK: Represents the else block in an if statement.
ELSE: Represents the "else" keyword.
DONE: Represents the end of a block or statement.
EXPR: Represents an expression.
EXPR_PRIME: Represents additional expressions to form complex expressions.
TERM: Represents a term in an expression.
TERM_PRIME: Represents additional terms to form complex terms in an expression.
FACTOR: Represents a factor in an expression, which can be an identifier or a literal value.
TYPE: Represents data types like INTEGER, FLOAT, STRING, BOOLEAN.