Jacal

Language Summary Version 0.01

Connor Johnson

Theory of Programming Languages

Alan Labouseur

5 May 2023

1. Introduction

Jacal is a programming language that follows a class-based, strongly-typed, and object-oriented paradigm. Its primary source of inspiration is the Pascal language. However, Jacal also incorporates several characteristics and functionalities found in other programming languages. These features were looked upon as potentially troublesome or ‘annoying’, however were still considered to be included in this new revolutionary programming language . Notably, Jacal is pronounced like the animal "Jackal." The new features incorporated into Pascal to make the Jacal language can be show as the following:

1.) Jacal took inspiration from languages like Erlang or Cobolt with their implementation of ending each line with either a ‘.’ or a ‘,’. Within Jacal’s syntax, we require you to have every line of code start and end with a ‘$’.

2.) Inside of the Pascal syntax, you would need a ‘writeln’ statement for printing purposes. However, in Jacal we gained inspiration from Java to change this phrase to ‘system.send.print.write.out()’ or with a new line instance ‘system.send.print.write.out.ln()’.

3.) Just like examples from the 1st generation programming languages, we gained inspiration to completely negate any conditional or loop statement with its actual word. Now these loops will be represented by one character and one number that you must memorize to use.

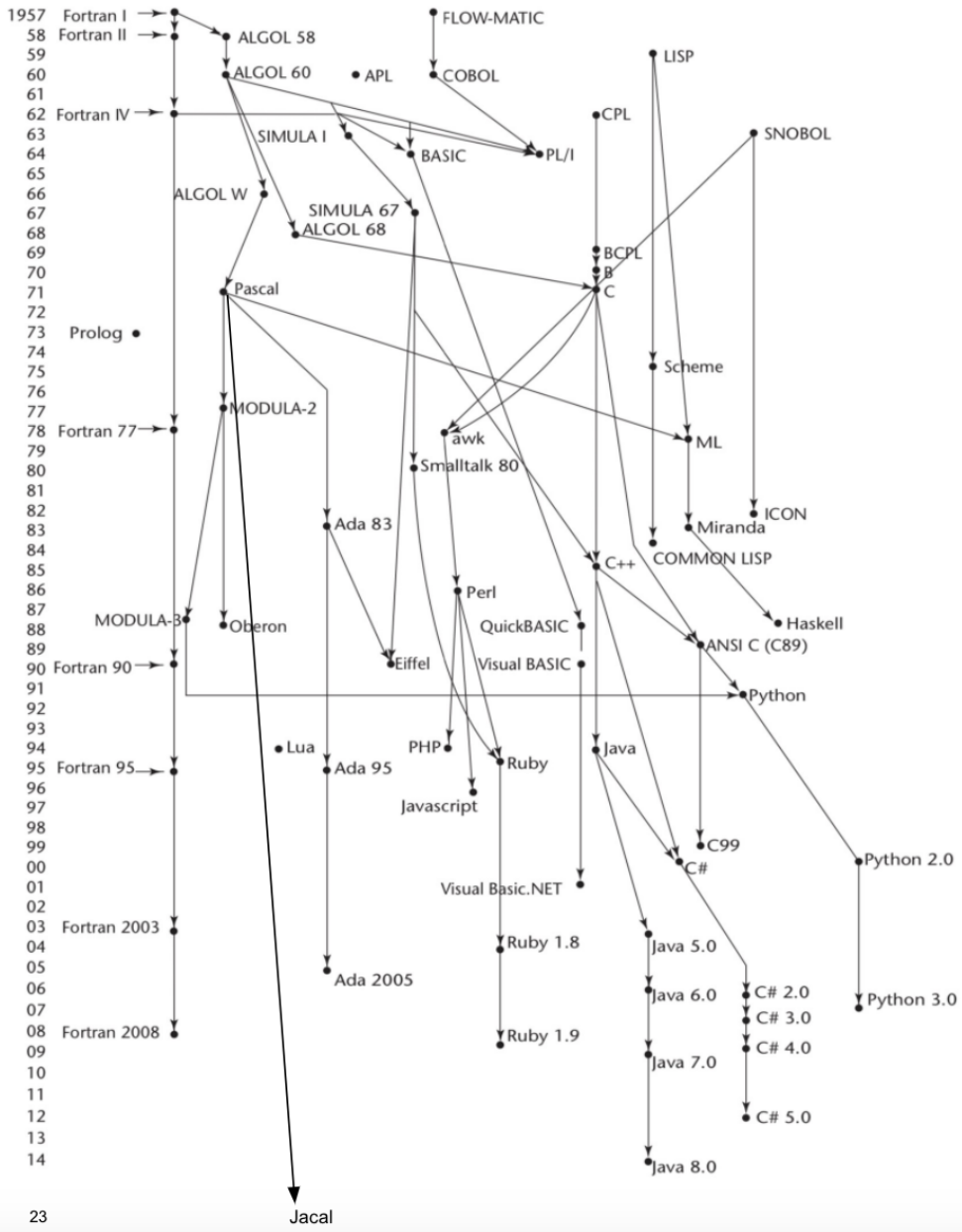
4.) Pascal provides statements like ‘begin’ and ‘end’ in its syntax to allocate organization within the code. However, in Jacal, we decided to build upon these phrases by making it now \{begin} and \{end}, similarly to how it is typed out in a LaTex document.

5.) Jacal does not support the ‘<=’ or ‘>=’ operators due to having to type out two different symbols. Now all you need to insert is the ‘≤’ and the ‘≥’ to represent a greater than/less than equal to

6.) With inspiration from relevant and riveting coding languages like Python, we decided to be nice and remove the type declarations inside of a function's conditions and its return type. All variables must still be declared inside of the var parameter.

7.) As seen in some languages like Python, Java and or C, Jacal saw that the infamous ‘walrus’ operator was too bland for our liking. In result, the new assignment operator will now be ‘>>’.

1.1. Genealogy



1.2.Hello world

```
$program helloWorld$

$var$
   $str: phrase$

$\{begin}$
    $str >> 'Hello World'$

    $system.send.print.write.out.ln(str)$

$\{end}$
```

1.3.Program structure

The key organizational concepts in Jascal are as follows:

A. All code must be inside the object block that is named after the filename of the program.
B. Execution of instructions is sequential.
C. Format is based off of the beginning var that initializes private variables being used inside of the program or a separate function. After var begins a function declaration and or its $\{begin}$ and $\{end}$ to initiate the desired code.
D. Code is not based on indentations or brackets, however warnings will still be issued for unorganized code.
E. Variables must be declared and typed in its respective var before calling

Example code:

In this program, we use all the changes made for Jacal. We have a car and year initialized, and in this small program we print out the car and its year and call for the PrintStatement function to print out another print statement to determine if the car's model year was a rare year:

```
$var$
  $car: phrase$
  $year: int$

$procedure PrintStatement(year)$
$\{begin}$
  $A7 year ≤ 2014 J2$
    $system.send.print.write.out.ln('This Toyota was manufactured before
boring 2015')$
  $H2 year ≥ 2017 J2$
    $system.send.print.write.out.ln('This Toyota was manufactured after
boring 2016')$
  $J9$
    $system.send.print.write.out.ln('This Toyota was manufactured in the
rare years 2015 and 2016')$
$\{end}$

$\{begin}$
  $car >> 'Toyota'$
  $year >> 2015$
  $system.send.print.write.out.ln('Car:', car, ', Year:', year)$
  $PrintStatement(year)$
$\{end}$
```

1.4.Types and Variables

In Jacal, there exist two distinct categories of data types, namely value types and reference types. Variables of value types directly store their data within their own memory allocation, while variables of reference types store a reference to the location in memory where the data is stored, commonly referred to as an object. Due to the nature of reference types, it is possible for two or more variables to reference the same object in memory, and as a result, operations performed on one variable may have an impact on the object referred to by the other variable(s).

1.5.Visibility

In Jacal, the declaration of variables can be done either in a function scope, meaning that the variable will only be accessible within that particular function, or globally scoped, meaning that the variable can be accessed throughout the entire program. This concept of scoping in variable declaration is similar to what is observed in Pascal, where variables can also be declared either locally, within a function, or globally, outside of any function. In other words, the visibility of a variable in Jacal, just like in Pascal, is determined by the scope in which it is declared.

1.6. Statements Statements Differing from Pascal

Statement	***Example***

Expressions	`$program ExpressionExample$`
	
	`$var$`
	`  $$name: phrase$`
	`  $birthdayMonth: int$`
	
	`$\{begin}$`
	`  $name >> 'Connor'$`
	`  $birthdayMonth >> 2$`
	`  $system.send.print.write.out.ln('Name:', car, ', Birth month in number:', year)$`
	`$\{end}$`

If - else
 	
If : A7
then: J2
else : J9
Else-if : H2
	`$program if-ElseExample$`
	
	`$var$`
	`  $$name: phrase$`
	`  $birthdayMonth: int$`
	
	`$\{begin}$`
	`  $name >> 'Connor'$`
	`  $birthdayMonth >> 2$`
	`  $system.send.print.write.out.ln('Name:', car, ', Birth month in number:', year)$`
	
	`    $A7 Length(birthdayMonth.) ~=~ 0 J2$`
	`        $system.send.print.write.out.ln("No arguments")$`
	`    $J9$`
	`        $system.send.print.write.out.ln("One or more arguments")$`
	
	`$\{end}$`

For
 	
For : C6
In : U7
Do : Z3
	`$program ForExample$`
	`$var$`
	`  $arr: array[1..5] of int ~=~ (1, 2, 3, 4, 5)$`
	`  $i: int$`
	
	`$C6 i U7 arr Z3$`

	$\{begin}$ $system.send.print.write.out.ln(i)$ ${end}$

While
 	
While: F2
	$program WhileExample$
	var
	  $i: int$
	
	$i >> 1$
	
	$F2 i ≤ 10 Z3$
	
	$\{begin}$
	
	  $system.send.print.write.out.ln(i)$
	  $i >> i + 1$
	
	${end}$

Comments
 	
Single line:
?Comment?
 	
Multiline:
?*Comment*?
	$program CommentExample$
	var
	
	  ?Comment?
	  $i: int$
	
	$i >> 1$
	
	$F2 i ≤ 10 Z3$
	
	?*Comment
	$\{begin}$*?
	
	  $system.send.print.write.out.ln(i)$?Comment?
	  $i >> i + 1$
	
	${end}$

2. Lexical structure

2.1.Programs

A Jacal program is composed of one or more source files, which are an ordered sequence of characters, potentially in the Unicode format. To compile a Jacal program, the process involves three steps:

1.) Transformation - this step converts a file from a specific character repertoire and encoding scheme into a sequence of Unicode characters.
2.) Lexical analysis - during this step, a stream of Unicode input characters are translated into a stream of tokens.
3.) Syntactic analysis - the stream of tokens is converted into executable code.

2.2.Grammars

This specification presents the syntax of Jacal where it differs from Pascal.

2.2.1.Lexical grammar (tokens) in Jacal

The Jacal lexical grammar is as follows:

<Assignment operator>	→	>>
<Mathematical operators>	→	+ | - | * | /
<Comparison operators>	→	truth | false | ~=~ | !=! | < , ≤ | > , ≥
<keyword>	→	language defined
		variable defined
<begin block>	→	\{begin}
<end block>	→	\{end}
<single line comment >	→	?Comment?
<begin multi line comment>	→	?*
<end multi line comment>	→	*?
<concatenation>	→	+
<new line>	→	$
<Input>	→	readln()
<Output>	→	system.send.print.write.out.ln()
		system.send.print.write.out()

2.2.2.Syntactic (“parse”) grammar in Jacal

The BNF grammar production for Jacal begins at the $program Example$ block, which serves as the entry point for the program. The program block must include all code, functions, and variable declarations that are to be used in the program, and must be defined before any code is developed. Statements that appear randomly outside of any defined function within a file are ignored and will not be executed by the program.

In essence, the $program Example$ block serves as the starting point for the Jacal program and defines the structure and organization of the program code. It is essential that all necessary code, functions, and variables are declared within the program block before any other code is developed, in order for the program to execute properly. Any statements that are not defined within a function will be ignored and will not contribute to the program's functionality.

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.

- Delimited comments start with the characters ?* and end with the characters *?. 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
real-literal

character-literal
string-literal
operator-or-punctuator

2.4.1.Keywords different from Pascal

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

New keywords:
C6, F2, U7, Z3, J2, M1, Q9, A7, J2, H2, J9, W2 >>, ≤, ≥, \{begin}, \{end}, ??, ?**?, ~=~, !=!, int, phrase

Removed keywords:
for, while, in, do, then, repeat, until, if, then, else-if, else, to, :=, <=, >=, begin, end, {}, {**}, ==, !=, integer, string

3. Type System

Jacal employs a robust static type system, which is characterized by its strong typing properties. This means that during compilation, type errors are detected and explicitly communicated to the programmer. Additionally, Jacal's type system is statically typed, which allows for compile-time type checking and early binding.

3.1.Type Rules

The type rules for Jacal are as follows:

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 >> e_2 : T} \qquad \frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 \sim\!=\!\sim e_2 : T}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 \; !\!=\!! \; e_2 : T} \qquad \frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 < e_2 : T}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 > e_2 : T}$$

3.2.Value types (different from Pascal)

It is important to know that value types are similar assigned as it was from Pascal

Data Type	***Description***	***Example***

int	32-bit signed value. Range -2147483648 to 2147483647	`$program intExample$`
		`$var$`
		`  $intexample: int$`
		
		`$\{begin}$`
		
		`  $intexample >> 2$`
		
		`$\{end}$`

float	double-precision floating-point binary numbers with up to 16 significant digits	`$program floatExample$`
		`$var$`
		`  $floatexample: float$`
		
		`$\{begin}$`
		
		`  $floatexample >> 2.0$`
		
		`$\{end}$`

boolean	Either the literal true or the literal false	`$program booleanExample$`
		`$var$`
		`  $booleanexample: boolean$`
		
		`$\{begin}$`
		
		`  $booleanexample >> true$`
		
		`$\{end}$`

char	fixed-length character string data type	`$program charExample$`
		`$var$`
		`  $charexample: char$`
		
		`$\{begin}$`

		$charexample >> 'a'$ $\{end}$

3.3.Reference types (differing from Pascal)

Data Type	*Description*	*Example*
phrase	variable-length, dynamically allocated character string data type, without limitation	$program phraseExample$ var $phraseexample: phrase$ $\{begin}$ $phraseexample >> 'Hello World'$ $\{end}$
array	Multi-Typed collection of data that has a variable length	$program arrayExample$ var $arr: array (1, 'x', 3, "Test", 5)$ $C6 i U7 arr Z3$ $\{begin}$ $system.send.print.write.out.ln(i)$ ${end}$

4. Example Programs

4.1.1 Caesar Cipher encrypt

```
$program ceaserCipher$


?Setting public variables?
$var$
   $str: phrase$
   $shiftAmount, maxShiftValue: int$

$function encrypt(str; shiftAmount)$

?Setting a private variable?
$var$
    $character: int$
$\{begin}$

    ?Loop to acquire each character in the str?
    $C6 character >> 1 W2 length(str) Z3$
        ?make sure it is a capital letter?
        $A7 (ord(str[character]) ≥ 65) and (ord(str[character]) ≤ 90) J2$
            $\{begin}$
                $str[character] >> chr(((ord(str[character]) - 65 +
shiftAmount) mod 26 + 26)mod 26 + 65)$
            $\{end}$
        $J9$
            ?If character is a special character?
            $\{begin}$
                $str >> str[character]$
            $\{end}$
    $encrypt >> str$
$\{end}$


$\{begin}$
    $str >> 'HAL'$
    $shiftAmount >> 26$
```

```
    $system.send.print.write.out.ln('Encrypted: ', encrypt(str,
shiftAmount))$

$\{end}$
```

4.1.2 Caesar Cipher decrypt

```
$program ceaserCipher$


?Setting public variables?
$var$
   $str: phrase$
   $shiftAmount, maxShiftValue: int$

$function encrypt(str; shiftAmount)$

Setting a private variable?
$var$
    $character: int$
$\{begin}$

    ?Loop to acquire each character in the str?
    $C6 character >> 1 W2 length(str) Z3$
        ?make sure it is a capital letter?
        $A7 (ord(str[character]) ≥ 65) and (ord(str[character]) ≤ 90) J2$
            $\{begin}$
                $str[character] >> chr(((ord(str[character]) - 65 +
shiftAmount) mod 26 + 26)mod 26 + 65)$
            $\{end}$
        $J9$
            ?If a character is a special character?
            $\{begin}$
                $str >> str[character]$
            $\{end}$
    $encrypt >> str$
$\{end}$

$function decrypt(str; shiftAmount)$
```

```
$var$
    $character: int$
    $decryptedString: phrase$
$\{begin}$

    ?Calls encrypt to properly decrypt?
    $decryptedString >> encrypt(str, shiftAmount)$

    $C6 character >> 1 W2 length(decryptedString) Z3$
        $A7 (ord(decryptedString[character]) ≥ 65) and
(ord(decryptedString[character]) ≤ 90) J2$
            $\{begin}$
                $decryptedString[character] >>
chr(((ord(decryptedString[character]) - 65 - shiftAmount) mod 26 + 26)mod
26 + 65)$
            $\{end}$
        $J9$
            $\{begin}$
                $decryptedString >> decryptedString[character]$
            $\{end}$
    $decrypt >> decryptedString$

$\{end}$


$\{begin}$
    $str >> 'HAL'$
    $shiftAmount >> 26$

    $system.send.print.write.out.ln('Encrypted: ', encrypt(str,
shiftAmount))$

$\{end}$
```

4.1.3 Factorial

```
$program factorial(input, output)$
```

```
$function factorial(n)$
$\{begin}$
    $A7 n ≤ 1 J2$
        $factorial >> 1$
    $J9$
        ?Common recursion example?
        $factorial >> n * factorial(n-1)$
$\{end}$


$var$
    $n: int$
$\{begin}$
    $system.send.print.write.out('Enter a non-negative integer to calculate
its factorial: ')$
    $readln(n)$
    $A7 n < 0 J2$
        $system.send.print.write.out.ln('Error: input must be
non-negative')$
    $J9$
        $system.send.print.write.out.ln(n, '! = ', factorial(n))$
$\{end}$
```

4.1.4 Insertion Sort

```
$program insertionSort$

?Uses these for public variables?
$const$
  $MAXSIZE ~=~ 10$

$type$
  $arrtype ~=~ array[1..MAXSIZE]$

$var$
  $arr: arrtype ~=~ (3, 9, 1, 7, 5, 8, 6, 2, 4, 0)$
  $n, i, j, key: int$

$procedure insertionSort(a, n)$
```

```
$var$
  $i, j, key: int$
$\{begin}$
  C6 i >> 2 W2 n Z3
  $\{begin}$
    $key >> a[i]$
    $j >> i - 1$
    ?compares the current element with the previous element?
    F2 (j > 0) and (a[j] > key) Z3
    $\{begin}$
      $a[j+1] >> a[j]$
      $j >> j - 1$
    $\{end}$
    ?found position of current element?
    $a[j+1] >> key$
  $\{end}$
$\{end}$

$\{begin}$
  $n >> MAXSIZE$

  $insertionSort(arr, n)$

  $system.send.print.write.out.ln('Sorted array:')$
  $C6 i >> 1 W2 n Z3$
  $\{begin}$
    $system.send.print.write.out(arr[i], ' ')$
  $\{end}$
$\{end}$
```

4.2.1 Stack

```
$program StackExample$


$\{begin}$
  $Stack ~=~ record$
    $top: integer$
    $data: array [1..100] of integer$
$\{end}$

$procedure InitStack(s)$
$\{begin}$
  $s.top >> 0$
$\{end}$

?Simple function to see if the stack has nothing in it?
$function IsStackEmpty(s)$
$\{begin}$
  $IsStackEmpty >> s.top ~=~ 0$
$\{end}$


?Simple function to see if the stack has no more space?
$function IsStackFull(s)$
$\{begin}$
  $IsStackFull >> s.top ~=~ 100$
$\{end}$

?Push to have items added to the stack?
$procedure Push(s, x)$
$\{begin}$
  A7 not IsStackFull(s) J2$
  $\{begin}$
    $s.top >> s.top + 1$ ?Get to the top of the stack?
    $s.data[s.top] >> x$ ?Add element?
  $\{end}$
$\{end}$

?Pop for items to leave the stack?
$function Pop(s)$
$\{begin}$
```

```
  A7 not IsStackEmpty(s) J2
  $\{begin}$
    $Pop >> s.data[s.top]$ ?Get the element at the top of the stack?
    $s.top >> s.top - 1$ ?Delete the last item?
  $\{end}$
$\{end}$

$var$
  $s: Stack$
  $i: int$

$\{begin}$
  $InitStack(s)$
  $C6 i >> 1 W2 5 Z3$
  $\{begin}$
    $Push(s, i)$
  $\{end}$
  F2 not IsStackEmpty(s) Z3$
  $\{begin}$
    $system.send.print.write.out.ln(Pop(s))$
  $\{end}$
$\{end}$
```

4.2.2 Queue

```
$program QueueExample$

  $\{begin}$
  $Queue ~=~ record$
    $data: array[1..100]$ ?data array to hold the elements of the queue?
    $front, rear: int$ ?front and rear indices of the queue?
  $\{end}$

?function to check if the queue is empty?
$function IsEmpty(q)$
$\{begin}$
  $IsEmpty >> (q.front ~=~ 0) and (q.rear ~=~ 0)$
```

```
$\{end}$

? function to check if the queue is full ?
$function IsFull(q)$
$\{begin}$
  $IsFull >> q.rear ~=~ 100$
$\{end}$

? procedure to add an element to the rear of the queue ?
$procedure Enqueue(q, x)$
$\{begin}$
  $A7 IsFull(q) J2$
    $system.send.print.write.out.ln('Error: Queue is full')$
  $H2 IsEmpty(q) J2$
  $\{begin}$
    $q.front >> 1$
    $q.rear >> 1$
    $q.data[q.rear] >> x$
  $\{end}$
  J9
  $\{begin}$
    $q.rear >> q.rear + 1$
    $q.data[q.rear] >> x$
  $\{end}$
$\{end}$

? procedure to remove an element from the front of the queue ?
$procedure Dequeue(q,x)$
$\{begin}$
  $A7 IsEmpty(q) J2$
    $system.send.print.write.out.ln('Error: Queue is empty')$
   $H2 q.front >> q.rear J2$
  $\{begin}$
    $x >> q.data[q.front]$
    $q.front >> 0$
    $q.rear >> 0$
  $\{end}$
  J9
  $\{begin}$
    $x >> q.data[q.front]$
    $q.front >> q.front + 1$
  $\{end}$
$\{end}$
```

```
$var$
  $q: Queue$
  $x: Integer$
$\{begin}$
  $q.front >> 0$
  $q.rear >> 0$

  ? enqueue  elements ?
  $Enqueue(q, 1)$
  $Enqueue(q, 2)$

  ? dequeue  elements ?
  $Dequeue(q, x)$
  $system.send.print.write.out.ln(x)$


  ? dequeue remaining elements ?
  $F2 not IsEmpty(q) Z3$
  $\{begin}$
    $Dequeue(q, x)$
    $system.send.print.write.out.ln(x)$
  $\{end}$
$\{end}$
```