Ciel Team
Jezreel Maldonado Ruiz
Cindy Méndez Avilés
Mariely Ocasio Rodríguez

Ciel Programming Language

Feb 26, 2020

Introduction

The Ciel programming language shall be a statically typed, generic and structured programming language. The Ciel language will utilize a compiler toolchain complete with preprocessor, scanner, parser, and code generator. The general idea of the language is for it to be fun and simple to program in while still exploring fresh and new concepts. An optional end goal for our Ciel language is to take bitmap images as source code.

Motivations & Reasons

The main motivation behind the Ciel programming language is to explore interesting ideas and introduce them to the programming community in a fresh and simple manner such that it will prove to be fun and intuitive to work on. This is the reason why the Ciel team decided to create a language in which its syntax will be more clear and precise, because it will bring another interesting way of interacting and interfacing with a computer system in a much personalized or stylish way. Therefore, Ciel will bring a more comfortable and orderly feel to its users.

Features

Structural

Data Types:

- char - unsigned 64 bit data type
- string - null-ended array of Char
- int - 64 bit data type
- float
    - Following the IEEE 754:
        - 1-bit will be used to store the sign
        - 11 bits will be used to store the exponent

        - 52 bits will be used to store the fraction

- bool - int wrapper
- null - NULL value
- var - generic 64 bit data type
- ptr - 64 bit data type which stores the memory address of a variable

Keywords:

- function
    - The function keyword will specify the declaration of the function header and the return type of the function
- return
    - The return keyword indicates the end of a function definition body and pops the function out of the call stack.
- until
    - The until keyword will indicate when a loop will break
- while
    - The while keyword will run the loop while the condition is met, tested at the beginning of each iteration
- for
    - The for keyword will run the loop for a specified amount of times while a condition is met.
- do
    - The do keyword will run the loop while the condition is met, tested at the end of each iteration.
- link
    - The link keyword will indicate which files will be prepended to the current translation unit.
- goto
    - Will accept jump directly to a label
- If
    - Will accept a boolean expression and a code block to execute when condition is met.

Operators:

Unary operators:

- **++**
    - The ++ unary operator will increase depending on the affix
        - Prefix: increase by one, then return.
        - Postfix: return, then increase
- **--**
    - The -- unary operator will decrease and return depending on the affix
        - Prefix: decrease by one, then return.
        - Postfix: return then decrease by one.
- **!**
    - Prefix: The ! operator negates its operand
- **?**
    - Postfix: The ? operator will test if a value is null when declared after a variable identifier
    - Postfix: The ? operator will declare a datatype as nullable
- **()**
    - Postfix: The () operator will serve as a functional indicator.
- **[]**
    - Postfix: The [] operator will serve as a subscript for indexing arrays.
- **-**
    - Prefix: The - operator will serve as an indicator for assigning the sign of a literal or identifier.
- **+**
    - Prefix: The + operator will serve as an indicator for assigning the sign of a literal or identifier.

Binary operators:

Arithmetic operators:

- **+**
    - The + operator means addition, it accepts two addends of type (char, string, int, float) which can be either literals, identifiers, or expressions.
- **-**
    - The - operator means subtraction, it accepts a minuend and subtrahend of type (char, string, int, float) which can be either literals, identifiers, or expressions.
- *

    - The * operator means multiplication, it accepts two multipliers of type (char, string, int, float) which can be either literals, identifiers, or expressions.
- /
    - The / operator means division, it accepts the numerator and divisor of type (int, float) which can be either literals, identifiers, or expressions.
- %
    - The % operator means modulus, it accepts the numerator and divisor of type (int, float) which can be either literals, identifiers, or expressions.
- ^^
    - The ^^ operator means exponentiation, it accepts the base and the exponent of type (int, float) which can be either literals, identifiers, or expressions.

Relational operators:

- !=
    - The != operator is used to check if two expressions are not equal to each other.
- ==
    - The == operator is used to check if two expressions are equal to each other.
- >
    - The > operator will check whether the expression in the left is greater than the expression in the right.
- <
    - The < operator will check whether the expression in the left is less than the expression in the right.
- >=
    - The >= operator will check whether the expression in the left is greater than or equal to than the expression in the right.
- <=
    - The <= operator will check whether the expression in the left is less than or equal to than the expression in the right.

Logical operators:

- ||
    - The || (or) operator will serve as a logical OR
- &&

- The && (and) operator will serve as a logical AND

Bitwise operators:

- **^**
    - The ^ operator will serve as bitwise XOR
- **|**
    - The | operator will serve as | bitwise OR
- **&**
    - The & operator will serve as | bitwise AND

Assignment operators:

- **=**
    - The = operator serves to assign a value.
- **+=**
    - The += operator assigns to the operand the result of a sum done to the operand itself.
- **-=**
    - The += operator assigns to the operand the result of a subtraction done to the operand itself.
- **%=**
    - The %= operator assigns to the operand the result of a modulus operation done to the operand itself.
- ***=**
    - The *= operator assigns to the operand the result of a multiplication done to the operand itself.
- **/=**
    - The /= operator assigns to the operand the result of a division done to the operand itself.

I/O operators:

- **put**
    - Accepts an output or input stream and a serializable object to be streamed outwards or inwards related to the program.

Ternary operators

- **?:**
    - The ?: operator will function as a way to declare lambda expressions in the typical fashion of (expression ? expression : expression).

Symbols:

- **.**
    - Specifies the end of a statement or expression
- **(,)**
    - Specify the beginning of a function call and encompasses the argument list
    - Specify and encompasses expressions
- **,**
    - Object delimiter in a list or to parameters in a function
- **->**
    - Beginning of a code block
- **<-**
    - Ending of a code block
- **<, >**
    - Indicate the data type of a var or return type of a function
- **|**
    - Marks the start and end of a comment
- **=>**
    - Used to delimit a variable to an iterable identifier

Architectural

Constructs:

Definitions

For the purposes of these examples, the following symbols will not have valid semantic and syntactic sense:

- expression : represents an expression be it arithmetical, logical or syntactical
- statement : represents a statement of imperative commands.
- identifier : represents an identifier, such as a variable name or a function name
- literal : represents a literal value inscribed in-text

- declaration : represents a variable declaration
- [symbol] : represents optional list of symbol
- DataType : represents a DataType

If construct:

```
if(expression).
->
    statement.
<-
```

While construct:

```
while(condition)
->
    statement.
<-
```

For construct:

```
for(declaration => identifier).
->
    statement.
<-
For (declaration. expression. statement) .
->
    statement.
<-
```

Function Construct:

```
function<DataType> foo([declarations]).
->
    statement.
    [return, yield] [identifier, literal]
<-
```

Until Construct:

```
until (expression).
->
    statement.
<-
```

Do Construct:

```
Do
->
    statement.
<- (expression).
```

Output Streams:

- Stdout - standard output text stream

Input Streams:

- Stdin - standard input text stream

Standard Libraries:

- IOlib - manges input / output
- Mathlib - manages math

End-Goal

The main purpose of this Ciel programming language is to remove all unnecessary complexity and maximize developer efficiency. The language also must help to produce clean and consistent applications which can be easily read and understood.

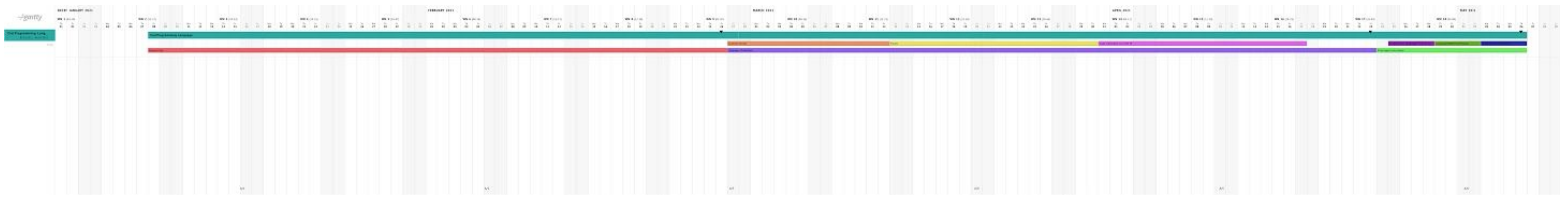
Requirements

- Lex/Flex - as a scanner generator in c/c++
- YACC/Bison - as a parser generator in c/c++
- Make (optional) - as a build system
- LLVM IR - as intermediate representation and machine code generator.

Example program

```
link IOlib.

function<int> main()

->

     stdout put “Hello World!”.

     return 0.

<-
```

Timeline & Gantt Chart



[link to gantt chart]