Programming Languages

Ciel

Jezreel J. Maldonado Ruiz

Mariely Ocasio Rodríguez

Comp4036-070

14th May, 2021

Introduction

The Ciel programming language was designed with usability and readability in mind, without making compromise in performance. The ciel scanner is written in C using the C99 standard, whilst the Ciel Parser is written in C++ using the C++11 standard. The Ciel programming language is multi-paradigm, focusing on a strictly statically typed experience with simple and basic structural programming. The general idea of the language is for it to be fun and simple to program in while still exploring fresh and new concepts. The main purpose of this Ciel programming language is to remove all unnecessary complexity and maximize developer efficiency. The appearance is visually pleasant for programmers, especially because of how the syntax is very straightforward, our syntax is highly inspired by C++ syntax but still brings a different and new feel to programming languages. The language also helps to produce clean and consistent applications which can be easily read and understood, making usability one of its strongest features. Therefore, users can program with The Ciel programming language' s aesthetically designed syntax while still making efficient and productive programs.

Language Tutorial

Compiling the language interpreter

As the Ciel interpreter is written in C & C++, it needs to be compiled using a C99-compatible C-compiler and a C++11-compatible C++-compiler. Refer to the project repository for further instructions on how to install them. You also need the latest versions of the Flex & Bison tools installed and on your path. Once installed, you can use `make`, which is the build system used by the project to maintain and easily keep track of the multi-stage compilation process. Once you have installed every dependency, you can compile the interpreter just by running `make` in the root of the directory tree. An executable with the name `ciel` will be constructed, this is the interpreter executable used to interpret the Ciel language.

Running the interpreter

After compiling the language interpreter, there are two ways to run the interpreter: interactively or in a file. To use the interpreter interactively you need to run the executable file without any arguments like this: `./ciel` , which will then let you convenience from using the Ciel programming language interactively. On the other hand, if you want to use the interpreter with your code already saved in a file you will have to add the file's name right next to the to executable, like so: `./ciel <filename>`

Basic syntax & rules

Ciel's syntax consists of statements followed by a "." (dot) at the end of the line. Statements can either be assignments, constructions, macros, or blocks. Ciel's blocks characteristically begin with a right arrow symbol "->" and end with a complementing "<-". Blocks provide context for our language, as all variables that are declared inside a context are only available in that block not outside it, and therefore are "destroyed" after the "<-" symbol is encountered.

If you want to have multiple translation units (files) to be interpreted from a single entrypoint, you can "link" them all together by utilizing the `link` keyword and then specifying that file's relative path to the `ciel` interpreter's original location. An example like so: `link "test.ci".`

To make inline comments in your code you can start the comment line with a "|" symbol and then close the comment line with the same "|" symbol.

Examples

Example Hello World:

```
stdout put “Hello World”.
```

Example of Contexts:

```
int a = 1.

->

     float b = 2.5.

     stdout put a.

<-

stdout put b.
```

In this example, an error should occur saying that b the identifier ‘b’ doesn’t exist when trying to put it into the standard output.

Example of basic arithmetic:

```
int a = 23.

int b = 180.

int c = 4 + b - a.

stdout put c.
```

The result of this example should be: 161

Example of I/O

```
int a.

int b.

stdout put “Enter a number: ”.

stdin put a.

stdout put “Enter another number: ”.

stdin put b.

stdout put a * b.
```

Example of Strings and Characters

```
string name.

char middleLetter.

stdout put "Enter your name: ".

stdin put name.

stdout put "Enter your middle initial: ".

stdin put middleLetter.

stdout put name + " " + middleLetter.
```

Language reference manual

Expressions

Expressions are quantifiable arithmetical expressions.

The associativity of the operators used are:

- left - sum, minus, multiplication, division
- right - exponentiation

Unary operators:

- -
    - 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.

Conditions

Conditionals consist of expressions and relational operators to compare and return a true or false result.

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.

Statements

Statements can either be blocks of code, assignments, declarations, or constructions.

Statements always end with a "." (dot) symbol at the end of each line.

Examples

```
int a = 1.

float b = 2.5.

bool c = true.

char d = “a”.

string e = “hello”.
```

These are all assignment statements for each of the different primitive types implemented in the language.

An assignment can become just a declaration by omitting the use of an equals sign and an expression like so:

```
int a.

float b.

bool c.

char d.

string e.
```

The interpreter will automatically assign default values to each of the different variables depending on their type. This value is 0 for int, float & bool. Char's have a default value of "\0" or null and strings have an empty string by default.

Construction examples

Block construction

```
->

int localVar = 0.

<-
```

If construction

```
if (2 < 5)

->

stdout put “two is less than five”.

<-
```

While construction

```
int iteration = 0.

while (true)

->

Stdout put iteration.

iteration += 1.

<-
```

Do while construction

```
int iteration = 0.

do

->

stdout put iteration.

iteration += 1.

<- while (iteration < 10).
```

Until construction

```
int iteration = 0.

until ( iteration == 10)

->

stdout put iteration.

Iteration += 1.
<-
```

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.

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

Macros

Macros are input/output keywords for managing internal system calls like printing to stdout or listening to stdin. Streams are identified by using “stdout” and “stdin” keywords, they must always be followed by the “put” keyword.

Output Streams:

- **stdout**
    - standard output text stream

Input Streams:

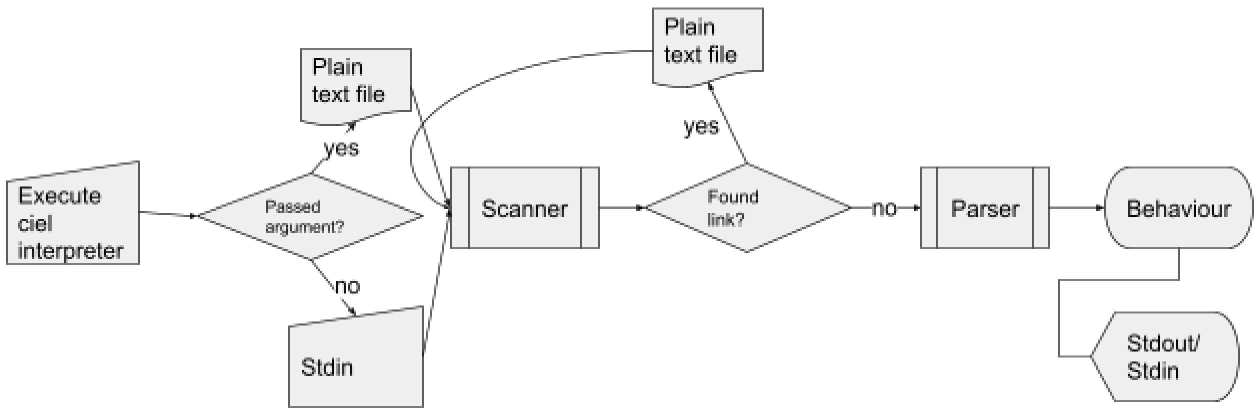
- **stdin**
    - standard input text stream

I/O operators:

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

Language Development

Translator architecture



The translator architecture of the ciel interpreted language is simple, only consisting of a Scanner to identify all the different tokens and a parser to construct the necessary syntax tree in order to evaluate each expression.

Scanner

Regular expressions

- [0-9]+
    - This regular expression is used to identify a chain of 1 or more occurrences of digits.
- [0-9]+[.][0-9]+
    - This regular expression is used to identify floating point numbers.
- \"([^"\\\n]|\\.)\"
    - This regular expression is used to identify a character.
- \"(\\.|[^"\\])*\"
    - This regular expression is used to identify a chain of strings.
- true | false
    - This regular expression is used to identify if something is true or false which is denoted by boolean expressions in which true is identified by the integer 1 and false by 0.
- char | string | int | float | bool | null

    - This regular expression is used to identify a primitive which will be used as a type for defining variables.
- [A-Za-z][A-Za-z0-9]*
    - This regular expression is used to identify identifiers.
- \|[^|]*\|
    - This regular expression is used to identify inline comments.
- [\t\n]+
    - This regular expression is used to identify whitespace.

“link” keyword:

- <lnk>\"
    - Initializes the flex command BEGIN(INITIAL), goes back to the initial state.
- <lnk>[\t]*
    - Eats the whitespace after the “link” keyword.
- <lnk>[^\"]+
    - Begins to enable the user to link multiple translation units (files) to be interpreted from a single entrypoint by using the “link” keyword and specifying that file’s relative path.

Parser

The only terminal symbols in the parser are expressions (exp) and terms (terms), these are the ones that get passed down to the different states of the CFG automata.

Grammar rules

Ciel defines a translation unit as a “file” or a sequence of imperative declarations with the following grammar rule:

```
translation_unit    :   statement statements

                    |   statement

                    ;
```

A single translation unit can have a collection of statements which are defined by the following grammar rule:

```
statement    : assignment

             | macro
```

```
            | block

            | ifelse

            | while

            | until

            | dowhile

            | TOK_LABEL_IDENTIFIER

            | TOK_GOTO_KEY TOK_IDENTIFIER TOK_DOT_SBL

            ;
```

As we can see here, statements can be either assignments, macros, blocks, if else, while loops, until loops, do while loops, labels, and gotos. We will be defining each of these afterwards.

```
statements  : statement

            | statements statement

            ;
```

A collection of statements is simply called statements.

```
block       : TOK_LARROW_SBL TOK_RARROW_SBL

            | TOK_LARROW_SBL statements TOK_RARROW_SBL

            ;
```

A block of code is simply a collection of statements that are delimited by -> and <- symbols.

```
while      : TOK_WHILE_KEY TOK_LPAR_SBL condition TOK_RPAR_SBL block ;
```

A while construction is defined by a while clause followed by a block.

```
until      : TOK_UNTIL_KEY TOK_LPAR_SBL condition TOK_RPAR_SBL block ;
```

A until construction is the same pattern for a while construction just using a different keyword.

Here we can see that almost all the different loop constructions use the same grammatical pattern, and thus all the rest are just variants using different keywords.

```
dowhile     :
```

```
TOK_DO_KEY  block   TOK_WHILE_KEY TOK_LPAR_SBL condition TOK_RPAR_SBL
TOK_DOT_SBL ;
```

Do while rule is one of the exceptions. A “.” (dot) symbol is required right after the while clause.

```
ifelse      :   if

            |   if else

            ;

if          :   TOK_IF_KEY TOK_LPAR_SBL condition TOK_RPAR_SBL block

            ;

else        :   TOK_ELSE_KEY block   ;
```

If and else statements are defined by a union of the if clause followed by a block. In which there might be an else clause followed by a block after the initial if.

Expressions, Conditions, terms and macros are a little more complex. These follow a repetitive pattern definition for each of the different types and their combinations to be able offer the different types’ desired behaviour.

Symbol Architecture

Each ciel symbol has an enum (internally an int in c/c++) defining it’s type and a “container” for its value which can be either boolean, integer, floating_point, string or character. These are defined by the following c structure below.

```
typedef struct symrec {

       int type;

       union {

           int integer;

           double floating_point;

           int boolean;

           char* string;

           char character;

       } value;

   } symbol;
```

An uncommon feature of each “symrec” (short for symbol record), is that it uses the union user-defined datatype. A union is a special declaration clause for defining a data-type, in which only one of its members can be held at once. This was an intentional decision to improve performance and yield a smaller memory footprint much like its brother language C.

A symbol table implemented as an std::unordered_map type in c++ is used to store each of the symbols encountered while parsing. Each encountered symbol’s identifier is used as the key to the symbol table structure.

To maintain context, a Context Stack is used to push new contexts in, and pop already interpreted contexts. Refer to the Example of Contexts: for more information. This is done by pushing new symbol tables in, once a new block has been encountered and popping off the stack once the “<-” character is encountered.

Software development environment

The software development environment used to create our translator consists of:

- make build system
- C99/C++11 standard programming languages
- Lex/Flex library and toolchain as a scanner generator in C
- Yacc/Bison library and toolchain as a parser generator in C++

The systems used and compatible compilers versions are:

- Windows 10 64-bit
  - Windows subsystem for Linux (WSL) version 2, Ubuntu
  - GNU Make version 4.2.1
  - C/C++ compilers gcc & g++ version 9.3.0
  - Flex version 2.6.4
  - Bison version 3.5.1
- MacOS Big Sur 11.2.3
  - Terminal
  - GNU Make 3.81
  - Apple clang version 12.0.0 (clang-1200.0.32.29) Target: x86_64-apple-darwin20.3.0
  - Flex 2.5.35 Apple(flex-32)
  - Bison (GNU Bison) 3.7.6

Test Methodology

The test methodology during development consisted of visualizing the lexer and parser as separate units and then linking them together at compile time. Warnings from the compiler were taken into consideration, although some are still part of the last version of the project. Graphs were generated using Bison to visualize the complexity of the Parser Automata Generated by Bison (see graph). Once a feature was implemented into the parser, it was tested using the interactive interpreter and verified before committing code into the different feature or fix branches. A Continuous Integration Github Actions workflow was implemented early on in the development of the project to make sure that all code compiled and didn't have any errors before merging into the master branch (see actions). A lot of care was poured into making the repository master branch pristine and clear of errors by having github pull request code reviews and description templates.

Test Programs

All examples shown previously in pages 3-4 (Examples) have been tested thoroughly with our translator.

Conclusion

The Ciel programming language has considerable usability when it comes to making easy to understand programs. Imperative programmers can take advantage of its unique design and simplicity, especially because of Ciel's similarity in structure to C++. While developing our programming language we used Flex and Bison as scanner and parser generators, both being completely new for us. Thus getting to contribute in the making of a programming language was a first and gratifying experience. In the end, we have discovered the complexity of building a programming language from scratch. We still believe that Ciel has much potential that can be expanded upon given more time and effort poured into the project. We ended with a competent implementation of the programming language we desired and although many more things could be added in, we believe that the initial version we have is a good v0.1.