int64

Ariel Ortiz – ariel.ortiz@itesm.mx

Table of Contents

1. Introduction

This document contains the complete technical specification of the *int64* programming language. The language is called *int64* because it only supports one data type: a 64-bit signed integer.

This language specification was developed for the spring semester TC3048 *Compiler Design* course at the Tecnológico de Monterrey, Campus Estado de Mexico.

2. Lexicon

In the following sections, a letter is any character from the English alphabet from `A` to `Z` (both lowercase and uppercase). A digit is any character from `0` to `9`.

2.1. Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Spaces, tabulators, newlines, and comments (collectively, “white space”) are used as delimiters between tokens, but are otherwise ignored.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

2.2. Comments

Comments can be either single or multi-line. Single line comments start with two slashes (`//`) and conclude at the end of the line. Multi-line comments start with a slash and an asterisk (`/*`) and end with an asterisk and a slash (`*/`). Comments cannot be placed inside string literals. Multi-line comments cannot nest.

2.3. Identifiers

An identifier is composed of a letter and a sequence of zero or more letters, digits and the underscore character (`_`). Uppercase and lowercase letters are considered different. Identifiers can be of any length. An identifier token appears as an ‹*id*› terminal symbol in the language grammar.

2.4. Keywords

The following fifteen identifiers are reserved for use as keywords, and may not be used otherwise:

`break`	`else`	`return`
`case`	`false`	`switch`
`continue`	`for`	`true`
`default`	`if`	`while`
`do`	`in`	`var`

2.5. Literals

There are four kinds of literals: booleans, integers, characters, and strings.

2.5.1. Booleans

A boolean literal is either `true` or `false` . These are equivalent to 1 and 0, respectively.

A boolean literal token appears as a ‹*lit-bool*› terminal symbol in the language grammar.

2.5.2. Integers

An integer literal can be represented using bases 2, 8, 10 and 16:

- A sequence of one or more digits is considered to be in base 10. Valid range: 0 to $2^{63} - 1$.
- Base 2 literals must start with `0b` or `0B` (a digit zero followed by the upper or lower case letter "b") followed by one or more binary digits (0 and 1). Valid range: 0 to $2^{64} - 1$.
- Base 8 literals must start with `0o` or `0O` (a digit zero followed by the upper or lower case letter "o") followed by one or more octal digits (0 to 7). Valid range: 0 to $2^{64} - 1$.
- Base 16 literals must start with `0x` or `0X` (a digit zero followed by the upper or lower case letter "x") followed by one or more hexadecimal digits (digits 0 to 9 and the upper or lower case letters "a" to "f"). Valid range: 0 to $2^{64} - 1$.

An integer literal token appears as a ‹*lit-int*› terminal symbol in the language grammar.

2.5.3. Characters

A character literal is a Unicode character enclosed in single quotes, as in `'x'`. The compiler translates the specified character into its corresponding Unicode integer code point.

Character literals do not contain the quote character (`'`) or newlines; in order to represent them, and certain other characters, the following escape sequences may be used:

Name	Escape Sequence	Code Point
Newline	`\n`	10
Carriage Return	`\r`	13
Tab	`\t`	9
Backslash	`\\`	92
Single Quote	`\'`	39
Double Quote	`\"`	34
Unicode Character	`\u` *hhhhhh*	*hhhhhh*

The escape `\u` *hhhhhh* consists of the backslash, followed by the lower case letter “u”, followed by six hexadecimal digits (digits 0 to 9 and the upper or lower case letters “a” to “f”), which are taken to specify the code point of the desired character.

A character literal token appears as a ‹*lit-char*› terminal symbol in the language grammar.

2.5.4. Strings

A string literal is a sequence of zero or more Unicode characters delimited by double quotes, for example: `"this is a string"`. String literals do not contain newline or double-quote characters; in order to represent them, the same escape sequences as for character literals are available.

A string literal is stored in memory as an array list (accessible through a 64-bit handle (https://en.wikipedia.org/wiki/Handle_(computing))) containing zero or more int64 values. Each value is the code point of the character in the corresponding position of the given string. In other words, a string is stored using an UTF-32 (https://en.wikipedia.org/wiki/UTF-32) encoding but using 64-bit values instead.

A string literal token appears as a ‹*lit-str*› terminal symbol in the language grammar.

3. Syntax

The following BNF context free grammar defines the syntax of the *int64* programming language. The **red** elements represent explicit terminal symbols (tokens).

‹*program*›	→	‹*def-list*›
‹*def-list*›	→	‹*def-list*› ‹*def*›
‹*def-list*›	→	ε
‹*def*›	→	‹*var-def*›
‹*def*›	→	‹*fun-def*›
‹*var-def*›	→	**var** ‹*var-list*› **;**
‹*var-list*›	→	‹*id-list*›
‹*id-list*›	→	‹*id*› ‹*id-list-cont*›

‹*id-list-cont*›	→	**,** ‹*id*› ‹*id-list-cont*›
‹*id-list-cont*›	→	ε
‹*fun-def*›	→	‹*id*› **(** ‹*param-list*› **)** **{** ‹*var-def-list*› ‹*stmt-list*› **}**
‹*param-list*›	→	‹*id-list*›
‹*param-list*›	→	ε
‹*var-def-list*›	→	‹*var-def-list*› ‹*var-def*›
‹*var-def-list*›	→	ε
‹*stmt-list*›	→	‹*stmt-list*› ‹*stmt*›
‹*stmt-list*›	→	ε
‹*stmt*›	→	‹*stmt-assign*›
‹*stmt*›	→	‹*stmt-fun-call*›
‹*stmt*›	→	‹*stmt-if*›
‹*stmt*›	→	‹*stmt-switch*›
‹*stmt*›	→	‹*stmt-while*›
‹*stmt*›	→	‹*stmt-do-while*›
‹*stmt*›	→	‹*stmt-for*›
‹*stmt*›	→	‹*stmt-break*›
‹*stmt*›	→	‹*stmt-continue*›
‹*stmt*›	→	‹*stmt-return*›
‹*stmt*›	→	‹*stmt-empty*›
‹*stmt-assign*›	→	‹*id*› **=** ‹*expr*› **;**
‹*stmt-fun-call*›	→	‹*fun-call*› **;**

‹fun-call›	→	*‹id›* **(** *‹expr-list›* **)**
‹expr-list›	→	*‹expr› ‹expr-list-cont›*
‹expr-list›	→	ε
‹expr-list-cont›	→	**,** *‹expr› ‹expr-list-cont›*
‹expr-list-cont›	→	ε
‹stmt-if›	→	**if (** *‹expr›* **) {** *‹stmt-list›* **}** *‹else-if-list› ‹else›*
‹else-if-list›	→	*‹else-if-list›* **else if (** *‹expr›* **) {** *‹stmt-list›* **}**
‹else-if-list›	→	ε
‹else›	→	**else {** *‹stmt-list›* **}**
‹else›	→	ε
‹stmt-switch›	→	**switch (** *‹expr›* **) {** *‹case-list› ‹default›* **}**
‹case-list›	→	*‹case-list› ‹case›*
‹case-list›	→	ε
‹case›	→	**case** *‹lit-list›* **:** *‹stmt-list›*
‹lit-list›	→	*‹lit-simple› ‹lit-list-cont›*
‹lit-list-cont›	→	**,** *‹lit-simple› ‹lit-list-cont›*
‹lit-list-cont›	→	ε
‹lit-simple›	→	*‹lit-bool›*
‹lit-simple›	→	*‹lit-int›*
‹lit-simple›	→	*‹lit-char›*
‹default›	→	**default :** *‹stmt-list›*
‹default›	→	ε

‹stmt-while›	→	**while (** *‹expr›* **) {** *‹stmt-list›* **}**
‹stmt-do-while›	→	**do {** *‹stmt-list›* **} while (** *‹expr›* **) ;**
‹stmt-for›	→	**for (** *‹id›* **in** *‹expr›* **) {** *‹stmt-list›* **}**
‹stmt-break›	→	**break ;**
‹stmt-continue›	→	**continue ;**
‹stmt-return›	→	**return** *‹expr›* **;**
‹stmt-empty›	→	**;**
‹expr›	→	*‹expr-cond›*
‹expr-cond›	→	*‹expr-or›* **?** *‹expr›* **:** *‹expr›*
‹expr-cond›	→	*‹expr-or›*
‹expr-or›	→	*‹expr-or›* **||** *‹expr-and›*
‹expr-or›	→	*‹expr-and›*
‹expr-and›	→	*‹expr-and›* **&&** *‹expr-comp›*
‹expr-and›	→	*‹expr-comp›*
‹expr-comp›	→	*‹expr-comp› ‹op-comp› ‹expr-rel›*
‹expr-comp›	→	*‹expr-rel›*
‹op-comp›	→	**==**
‹op-comp›	→	**!=**
‹expr-rel›	→	*‹expr-rel› ‹op-rel› ‹expr-bit-or›*
‹expr-rel›	→	*‹expr-bit-or›*
‹op-rel›	→	**<**
‹op-rel›	→	**<=**

‹op-rel›	→	>
‹op-rel›	→	>=
‹expr-bit-or›	→	‹expr-bit-or› ‹op-bit-or› ‹expr-bit-and›
‹expr-bit-or›	→	‹expr-bit-and›
‹op-bit-or›	→	|
‹op-bit-or›	→	^
‹expr-bit-and›	→	‹expr-bit-and› **&** ‹expr-bit-shift›
‹expr-bit-and›	→	‹expr-bit-shift›
‹expr-bit-shift›	→	‹expr-bit-shift› ‹op-bit-shift› ‹expr-add›
‹expr-bit-shift›	→	‹expr-add›
‹op-bit-shift›	→	<<
‹op-bit-shift›	→	>>
‹op-bit-shift›	→	>>>
‹expr-add›	→	‹expr-add› ‹op-add› ‹expr-mul›
‹expr-add›	→	‹expr-mul›
‹op-add›	→	+
‹op-add›	→	-
‹expr-mul›	→	‹expr-mul› ‹op-mul› ‹expr-pow›
‹expr-mul›	→	‹expr-pow›
‹op-mul›	→	*
‹op-mul›	→	/
‹op-mul›	→	**%**

‹*expr-pow*›	→	‹*expr-unary*› `**` ‹*expr-pow*›
‹*expr-pow*›	→	‹*expr-unary*›
‹*expr-unary*›	→	‹*op-unary*› ‹*expr-unary*›
‹*expr-unary*›	→	‹*expr-primary*›
‹*op-unary*›	→	`+`
‹*op-unary*›	→	`-`
‹*op-unary*›	→	`!`
‹*op-unary*›	→	`~`
‹*expr-primary*›	→	‹*id*›
‹*expr-primary*›	→	‹*fun-call*›
‹*expr-primary*›	→	‹*lit*›
‹*expr-primary*›	→	`(` ‹*expr*› `)`
‹*lit*›	→	‹*lit-simple*›
‹*lit*›	→	‹*lit-str*›
‹*lit*›	→	‹*array-list*›
‹*array-list*›	→	`{` `}`
‹*array-list*›	→	`{` ‹*lit-list*› `}`

4. Semantics

- The language only supports a 64-bit signed integer (int64) data type. This is the data type for every variable, parameter and function return value.
- Every program starts its execution in a function called `main`. It is an error if the program does not contain a function with this name.

- Any variable defined outside a function is a global variable. The scope of a global variable is the body of all the functions in the program, even those defined before the variable itself.
- It is an error to define two global variables with the same name.
- A function definition is visible from the body of all the functions in a program, even from itself (thus, functions can be defined recursively).
- It is an error to define two functions with the same name.
- Function names and global variables exist in different namespaces. This means that you can have a global variable with the same name as a function.
- Within a function, parameter and local variable names exist in their own namespace, so they must be unique. It is valid to have a parameter or local variable name with the same name as a global variable. In that case the local name shadows (https://en.wikipedia.org/wiki/Variable_shadowing) the global variable.
- A function returns zero by default, except if it executes an explicit `return` statement with some other value.
- All statements have the same behavior as in the C programming language, with the following differences:
    - A `case` in a `switch` statement does not fall-through. So, for example the following C code:

      INT64
      ```
      switch (x) {
      case 1:
      case 2:
      case 3:
          y = 0;
          break;
      case 4:
          y = 1;
          break;
      default:
          y = 2;
          break;
      }
      ```

      has to be written in *int64* in a shorter and safer way:

```
INT64
switch (x) {
case 1, 2, 3:
    y = 0;
case 4:
    y = 1;
default:
    y = 2;
}
```

This means that the `break` statement is only used to terminate early an enclosing *while*, *do-while*, or *for* statement.

- The `for` statement allows iterating (similar to the `foreach` statement in C#) over an array list given its handle. So, for example, the following snippet adds the code points of all the characters in a string:

```
INT64
var sum, str, ch;
sum = 0;
str = "some characters";
for (ch in str) {
    sum = sum + ch;
}
```

The `for` statement throws an exception if the given handle is not valid.

- The following are the supported operators. Precedence and associativity are established in the language grammar.

Table 1. Arithmetic operators

Operator	Syntax	Semantics
Unary minus	$- x$	Like C. An exception is thrown if the result does not fit in an int64.
Unary plus	$+ x$	Like C.
Power	x ** y	Like Python. An exception is thrown if the result does not fit in an int64.

Operator	Syntax	Semantics
Multiplication	*x* * *y*	Like C. An exception is thrown if the result does not fit in an int64.
Division	*x* / *y*	Like C.
Remainder	*x* % *y*	Like C.
Addition	*x* + *y*	Like C. An exception is thrown if the result does not fit in an int64.
Subtraction	*x* − *y*	Like C. An exception is thrown if the result does not fit in an int64.

Table 2. Bitwise operators

Operator	Syntax	Semantics
Bitwise NOT	~ *x*	Like C.
Bitwise AND	*x* & *y*	Like C.
Bitwise OR	*x* | *y*	Like C.
Bitwise XOR	*x* ^ *y*	Like C.
Bitwise Shift Left	*x* << *n*	Like C.
Bitwise Shift Right	*x* >> *n*	Like C.
Bitwise Unsigned Shift Right	*x* >>> *n*	Like Java.

Table 3. Logical operators

Operator	Syntax	Semantics
Logical NOT	! *x*	Like C.

Operator	Syntax	Semantics
Logical AND	x && y	Like C.
Logical OR	x || y	Like C.

Table 4. Comparison and relational operators

Operator	Syntax	Semantics
Equal to	$x == y$	Like C.
Not equal to	$x != y$	Like C.
Greater than	$x > y$	Like C.
Less than	$x < y$	Like C.
Greater than or equal to	$x >= y$	Like C.
Less than or equal to	$x <= y$	Like C.

Table 5. Other operators

Operator	Syntax	Semantics
Function call	$f\,(arg_1, arg_2, \ldots, arg_n)$	Like C.
Ternary conditional	$x\ ?\ y : z$	Like C.

5. API

This section documents all the functions from the *int64* application programming interface (API).

Table 6. Input/Output Operations

Signature	Description
`printi(i)`	Prints `i` to stdout as a decimal integer. Does not print a new line at the end. Returns 0.

Signature	Description
`printc(c)`	Prints a character to stdout, where `c` is its Unicode code point. Does not print a new line at the end. Returns 0.
`prints(s)`	Prints `s` to stdout as a string. `s` must be a handle to an array list containing zero or more Unicode code points. Does not print a new line at the end. Returns 0.
`println()`	Prints a newline character to stdout. Returns 0.
`readi()`	Reads from stdin a signed decimal integer and return its value. Does not return until a valid integer has been read.
`reads()`	Reads from stdin a string (until the end of line) and return a handle to a newly created array list containing the Unicode code points of all the characters read.

Table 7. Array List Operations

Signature	Description
`new(n)`	Creates a new array list object with `n` elements and returns its handle. All the elements of the array list are set to zero. Throws an exception if `n` is less than zero.
`size(h)`	Returns the size (number of elements) of the array list referenced by handle `h`. Throws an exception if `h` is not a valid handle.
`add(h, x)`	Adds `x` at the end of the array list referenced by handle `h`. Returns 0. Throws an exception if `h` is not a valid handle.
`get(h, i)`	Returns the value at index `i` from the array list referenced by handle `h`. Throws an exception if `i` is out of bounds or if `h` is not a valid handle.
`set(h, i, x)`	Sets to `x` the element at index `i` of the array list referenced by handle `h`. Returns 0. Throws an exception if `i` is out of bounds or if `h` is not a valid handle.

6. Code Examples

Source File	Description
hello.int64	Prints "Hello World!" to the stdout.
binary.int64	Converts decimal numbers into binary.
palindrome.int64	Determines if a string is a palindrome.
factorial.int64	Computes factorials using iteration and recursion.
arrays.int64	Implementation of typical array operations.
next_day.int64	Given the date of a certain day, determines the date of the day after.
vars.int64	Example using global and local variables.
literals.int64	Verifies that the implementation of literal values meet the specified requirements.
operators.int64	Verifies that the implementation of all the operators meet the specified requirements.
break_continue.int64	Verifies that the implementation of `break` and `continue` meet the specified requirements.

Last updated 2017-08-16 23:29:02 UTC