Casper

Language Reference Manual

Michael Makris

UNI: mm3443

COMS W4115 Programming

Languages and Translators

October 15, 2018

Introduction

Casper is a rather limited in scope general-purpose imperative language that resembles the C language, but with emphasis on the high level than the traditional C low level capabilities. For example, it includes a String data type and library functions to manipulate strings. In this respect, the language should be able to implement many of the usual algorithms for applications that are programmed in C, Java, and Python.

A. Lexical Conventions

A1. Identifiers

An identifier is a sequence of letters, uppercase or lowercase, including the underscore _, and digits, with the first character always a letter. Identifiers are used as tokens to identify variables and functions.

A2. Comments

Comments are ignored by the language compiler.

a) Line comments are signified from // to the end of the line.
b) Block comments are enclosed by /* *some comment* */ and can span multiple lines.

A3. Reserved words

The following tokens are reserved for use as keywords:

int float bool str void true false null if else for while do until break continue return print Input

A4. Whitespace

Newline \n, carriage return \r, horizontal tab \t, and space are considered whitespace and together with the comments they are ignored by the language compiler.

A5. Literals

A5.1 Integer literals

Sequence of digits 0 … 9, optionally signed (prefixed with + or -) representing integers.

A5.2 Floating point literals

Sequence of digits 0 … 9, optionally signed (prefixed with + or -), representing the integer part, followed by a period . and another sequence of digits representing the fraction part. Either the integer or the fraction part may be missing but not both.

A5.3 String literals

A sequence of characters surrounded by double quotes as in "*abc*" or single quotes as in '*abc*'. This allows for one type of quote to be included in a string defined by the other type. Strings are immutable.

A5.3 Logical literals

true and false are tokens used in Boolean expressions.

A5.3 Null

The token null can be used for comparison expressions or assignments to any data type and represents the lack of any value.

B. Data Types

Type	Description	Declaration syntax
Integer	an integer depended on host machine	int x = 0;
Floating point	a floating point number	float x=3.14;
Boolean	reserved words true and false	bool x = true;
String	variable length sequence of characters	str x = “abc”; str x = ‘abc’;
Void	representing the empty set or no value	void x;

C. Expressions

C1. Variables

Variables are declared as shown in the data type section above for the five data types and can be assigned an equivalent type literal or null.

C2. Arrays

Single dimensional list of elements of the same type as the declared array for the Integer, Floating point and String types. Can be initialized with an equivalent type literal or null, or by a same size and type array. Elements can be accessed by position starting from 0 and are enclosed in brackets and separated by commas.

Type	Description	Declaration syntax	
Integer	an integer array	int x[n]=0;	x[1]=1;
Floating point	a floating point array	float x[2]=null;	x[0] == x[1]
String	a string array	str x[2] = [‘abc’, “123”];	x[0] = x[1] _ x[2];

C3. Functions

C3.1 User-defined functions

Functions return a value of the data type they are declared as, except type void which returns null, and take a number of arguments of any type. The argument list is enclosed in parenthesis and arguments are separated by commas. In the function definition the body of statements is in braces and the keyword return with a value can exit the function and return the value. Statements are terminated by a semicolon. As in C, main () is the special function that executes first.

Type	Declaration syntax	Definition syntax
Integer	int myFun(myArg1, …);	myFun (int x) {return x + 1;}
Floating point	float myFun(myArg1, …);	myFun(){return 3.14;}
Boolean	bool myFun(myArg1, …);	myFun(){return true;}
String	string myFun(myArg1, …);	myFun(){return “hello world”;}
Void	void myFun(myArg1, …);	myFun(){return null;}

C3.2 Built-in functions (I/O)

print(str) to print to standard output
str = input() to read from standard input

C4. Operators

Operator	Description	Syntax
_	binary string concatenation	‘a’ _ “b”
?	binary character at position	“abc”?0 == “a”
+	binary arithmetic addition	1 + 2 1.0 + 2.0
-	binary arithmetic subtraction	1 – 2 1.0 - 2.0
*	binary arithmetic multiplication	1 * 2 1.0 * 2.0
/	binary arithmetic float division	1.5 / 2.5
%	binary arithmetic modulus	1 % 2
^	binary arithmetic exponentiation	2 ^ 2 2.0 ^ 0.5
>	binary relational greater than	1 > 2
>=	binary relational greater than or equal	1 >= 2
<	binary relational less than	1 < 2
<=	binary relational less than or equal	1 <= 2
==	binary relational equal	1 == 2
!=	binary relational not equal	1 != 2
-	unary negation	-1
++	unary increment (pre or post) an integer	int i = 0; i++; ++i;
--	unary decrement (pre or post) an integer	int i = 0; i--; --i;
=	assignment of right-hand expression to left-hand side	int i = 0; str x = “abc”;
+=	assignment of the sum of the two sides to the left-hand side	int i = 0; i += 1;
-=	assignment of the difference of the two sides to the left-hand side	int i = 0; i -= 1;
&&	binary logical AND	x && y
||	binary logical OR	x || y
!	unary logical NOT	!x

Precedence

Operator	Associativity
() []	left to right
- ! ++ -- ?	right to left
^	right to left
* / %	left to right
+ -	left to right
_	left to right
> >= < <=	left to right
== !=	left to right
&&	left to right
||	left to right
= += -=	right to left

D. Control Flow

D1. Structure

a) White space is ignored
b) Statements terminated by ;
c) Expressions defined by () with no ; after
d) Compound statements/blocks and scope defined by {} with no ; after

D2. Conditional block

if (expression1) {statement1;}
else if (expression2) {statement2;}
else {statement3;}

D3. Loops

for (<optional initialization>; <optional termination expression is true>; <optional increment>)
{ <statements> }

while (<test expression is true>) { <statements> }

do { <statements> } while (<test expression is true>)

do { <statements> } until (<test expression is true>)

with keyword break allowed in statement block to exit loop and keyword continue to jump to the next iteration.