BURGer Language Reference Manual

name	uni	role
Jacqueline Kong	jek2179	Language Guru
Jordan Lee	jal2283	Manager
Ashley Nguyen	akn2121	Tester
Adrian Traviezo	aft2115	System Architect

Introduction

BURGer is a programming language whose goal is to make text-based adventure games easy to create. While it can be used to develop real-world applications as a result of being a general-purpose language, BURGer makes writing a text-based game more intuitive by offering native structures and syntactical elements that optimize structuring and writing clean code for sequences of game scenes that are to be displayed on a console. In this spirit, we have decided to prioritize readability and avoided making the language strongly typed so that the BURGer can appeal to users who are not necessarily familiar with stricter programming conventions and backend concerns, just as burgers themselves often do.

1. Lexical elements

1.1. Identifiers

Identifiers are used for naming data types. Identifiers consist of any combination of alphanumeric characters. The first character of an identifier must be a letter.

1.2. Comments

Comments are denoted like so:

```
// text; this is a comment!!
```

OR

```
/* text
wow look at me
multiple lines
incredible */
```

1.3. Keywords, Symbols, and Operators

Keywords

`if`	`int`	`Scene`
`else`	`char`	`Inventory`

for	bool	Item
while	String	Player
float	List	Option
def	has	return
else if	break	continue

Symbols and Operators

+	-=	[ ]	==	{ }
-	+=	<<	=	&&
*	*=	>>	.	||
/	/=	<=	( )	" "
++	--	>=	%	' '
,	;	!	!=	< >
->	%=			

The standard library for BURGer also includes the <, >, and -> operators, which are used to instantiate scenes in the format shown in the sample code.

1.4. Constants

1.4.1. Integer constants

An integer constant consists of one or more digits. Integers are optionally signed and default to positive if they are unsigned.

1.4.2. Floating constants

Floating constants in BURGer have an integer part and decimal part. Floating constants, like integers, are optionally signed.

1.4.3. Character constants

A character constant is 1 character (or 2 characters in the case of an escape sequence) enclosed by single quotes or double quotes. BURGer does not distinguish between the use of single quotes or double quotes. Escape sequences in BURGer are identical to those in C.

1.4.4. Boolean constants

Boolean constants can be `true` or `false`, which respectively correspond to logical true and false values.

1.5. Strings

A string is a sequence of characters enclosed in single or double quotes. BURGer strings are mutable and iterable.

2. Data Types

2.1. Primitive types

There are five primitive data types: `int, float, char, bool`, and `null`.

Most numbers will be denoted as an `int` type, which stores up to 4 bytes (to refer to numbers of items in inventory, for example). Numbers can also be stored as a `float`, a 32-bit numerical value.

`char` holds a single ASCII character as its value. `bool` holds a Boolean value of either `true` or `false`. `null` is a type used for uninitialized variables.

2.2. Non-primitive types

2.2.1. `String` and `List`

Strings, as discussed in Section 1.5, are a non-primitive type supported in BURGer. A `String` is an ordered and iterable list of `char`s.

A `List` is an ordered, iterable list of any data type. This encompasses data types common in other programming languages such as arrays. `List`s are resizable, meaning that the number of objects assigned to the `List` when it's declared can change afterwards. Data can be added and removed from a `List`.

2.2.2. Custom Objects

`Scene`	An object containing these values:
	`text: (String)`
	`options: (List)`
	`next: (Scene)`

`Inventory`	An object containing these values:
	`items: (list of Items)`
	`capacity: (int)`
	`amount: (int)`

	`display()`

`Item`	something that each Character can have in their Inventory; an object containing these values:
	`name: (String)`
	`quantity: (int)`
	`use()`

`Option`	An object containing these values:
	`selector: (String)` must be unique
	`text: (String)`
	`action()`

3. Expression and Statement Syntax

3.1. Operators

3.1.1. Relational Operators

BURGer has the following relational operators:

- `<<` less than
- `>>` greater than
- `<=` less than or equal to
- `>=` greater than or equal to

Note that BURGer uses a different symbol for the less than and greater than operators than most commonly-used programming languages.

3.1.2. Equality Operators

BURGer has the following equality operators:

- `==` equals
- `!=` not equals

3.1.3. Logical Operators

BURGer has the following logical operators:

- `&` and
- `|` or
- `!` not

3.1.4. Assignment Operators

BURGer has the following assignment operators:

- `=` sets the left operand equal to the right operand
- `+=` adds the right operand to the value of the left operand and sets the left operand equal to the total

-= subtracts the right operand from the left operand value and sets the left operand equal to the new value
*= multiplies the surrounding values and sets the left operand equal to the product
/= divides the surrounding values and sets the left operand equal to the result
%= divides the surrounding values and sets the left operand equal to the remainder

3.1.5. Arithmetic Operators

BURGer supports the standard arithmetic operations + (addition), – (subtraction), * (multiplication), / (division), and % (modulo).

3.1.6. Operator Precedence

The order of precedence for classes of operators is as follows, from the highest to the lowest: arithmetic, logical, assignment, equality/relational. Equality and relational operators have the same level of precedence. For arithmetic operations, multiplication and division take precedence over addition and subtraction.

Expressions contained within parentheses always take precedence. Otherwise, operators of equal precedence levels will take precedence from left to right.

3.2. Delimiters

3.2.1. Parentheses

BURGer uses parentheses to determine the operation precedence and for enclosing function calls.

3.2.2. Whitespace

BURGer uses whitespace to separate tokens. However, the amount of whitespace has no other bearing on the language.

3.2.3. Semicolons

BURGer uses semicolons to terminate statements.

3.2.4. Square brackets

BURGer uses the square brackets to enclose the values in a list.

3.2.5. Angled brackets

BURGer uses the angled brackets to denote an option. It can be used with <[selector], [text], [action]> where [selector], [text], [action] are replaced with the option's respective selector, text, and action functions.

3.3. Declaration and Initialization of Variables and Functions

3.3.1. Variables

Variables are declared without the `def` keyword or a return type, but must have a variable name. In the example below, since 20 does not have a decimal point, the value is automatically determined to be an `int`.

```
x = 20;
```

Variable must be assigned a value, even if that value is null. Multiple variables can also be declared on the same line and be initialized to the same value, like so:

```
x, y, z = null;
```

3.3.2. Functions

A function is declared with the `def` keyword, a function name, and a list of parameters between parentheses. The return type of the function and of its parameters are not specified on declaration, but a function that does return a value is assigned that return type. A function may be declared with curly braces specifying what the function does, like so:

```
def adder (x, y) {
      return x + y;
};
```

Or, a function may be declared without any curly braces, like so:

```
def adder (x, y);
```

In the latter case, the function does not perform any operations, but is considered initialized and in the scope of any declarations or operations that come after it (see 4.2 Scope).

3.4. Built-in Functions

3.4.1. `print()`

BURGer uses the `print()` function to print to the console.

3.4.2. `exit()`

BURGer uses the `exit()` function to exit the program.

3.4.3. `input()`

BURGer uses `input()` to read in a string of data.

3.4.4. `options()`

BURGer uses the `options()` function by taking in a comma separated list of options and displays the options for the player.

3.5. Control Flow Expressions

3.5.1. `if...else`

BURGer uses the if else statements in a similar way to other languages. if(expression) { statement; } else{ statement; }

3.5.2. `for(){}`

BURGer performs `for` loops with the same syntax as Java.

3.5.3. `while(){}`

BURGer performs `while` loops with the same syntax as Java.

3.5.4. return statements

BURGer uses the `return` statement to return values from a function.

4. Program Structure and Scope Rules

4.1. Program Structure

A BURGer program must be contained in a single source file, whose extension is ".bun" and has an analog to the C language's `main` method, called `start`. The function labeled `start` will execute at runtime.

4.2. Scope

The scope of a declared object (a variable, function, or struct) refers to the parts of the program from which it is visible. An object may only be visible from within the .bun file where it was declared.

4.2.1.1. Global objects may be declared independently of any code block - as in, they do not need to exist within any function or as part of another declaration. By convention, they should be declared at the top of the file. These objects are visible from any point of the file.

4.2.1.2. Local objects are visible only from within the function where they are defined. If a helper function must access an object declared in the outer function from which it is called, the object must be passed into the helper function as a parameter.

4.2.1.3. An object is not visible to any operations or declarations that have come before it - for example, a variable x_1 may not form part of the declaration of x_2 unless x_2 was previously formally declared.

5. Grammar

Terminals are written in bold.

program ➜
| program vdecl
| program fdecl

num ➜
id
| **constant**

fdecl ➜ **def id (** formals **) {** vdecls stmts **}**

formals ➜ **id**
| formals **, id**

vdecls ➜ vdecl
| vdecls vdecl

vdecl ➜ **id ;**

expr ➜ **stringlit**
| num binary_arith num
| bool_expr
| **id (** actuals **)**
| num unary_arith
| game_expr
| scene_nav
| assign_expr
| cast_expr

cast_expr ➜ **char (** num **)**

| **int (** num **)**
| **float (** num **)**

game_expr → **< stringlit , stringlit , id >**
| **< stringlit , stringlit ,** fdecl **>**

actuals → expr
| actuals **,** expr

statements →
| statements statement

statement → expr **;**
| end_function **;**
| **{** statements **}**
| iter_statement
| cond_statement
| jump_statement **;**

iter_statement → **while (** expr **)** statement
| **for (** expr **;** expr **;** expr **)** statement

cond_statement → **if (** bool_expr **)** statement
| **if (** bool_expr **)** statement **else** statement
| **if (** bool_expr **)** statement **else if (** bool_expr **)** statement
| **if (** bool_expr **)** statement **else if (** bool_expr **)** statement **else** statement

jump_statement → **continue**
| **break**

end_function → **return id**
| **return**

bool_expr → **id** binary_log num
| **id** binary_log **stringlit**

| **id** binary_log **bool**
| **constant** binary_log **constant**
| **stringlit** binary_log **stringlit**
| **bool** binary_log **bool**
| **id** binary_rel num
| **constant** binary_rel **constant**

list ➔ num
| **id ,** list
| **constant ,** list

list_access ➔ id **[** num **]**
| **id [** num **:** num **]**
| id **[** num **:]**
| id **[:** num **]**

assign_expr ➔ num = num
| **id** = **stringlit**
| **id** = **bool**
| num arith_assign num
| **id** = **null**
| **id** = **cast_expr**
| **id** = **[** list **]**

scene_nav ➔ **id -> id**

binary_arith ➔ **+**
| **-**
| *
| **/**
| **%**

arith_assign ➔ **+=**
| **-=**
| ***=**
| **/=**

| %=
| =

unary_arith → ++
| --

binary_rel → ==
| >>
| <<
| >=
| <=

binary_log → &&
| ||
| !=

6. Sample Code

```
money = 5;
inventory = Inventory(“laptop”, “bookbag”, money);

/* Initialize a few scenes */
Scenes(Bed, CS_Lounge, PLT, GameOver);

/* Define the text they display upon entering scene */
Bed.text = "Good morning! It’s time for class.";
CS_Lounge.text = “You came to the CS Lounge and spot a pizza, but
don’t know if you can take any. What will you do?”;
PLT.text = “You’re in class. Do you fall asleep or pay attention?”
GameOver.text = “You died, game over. Play again?”

/* Create a path of scenes */
Bed(<0, Keep sleeping.>)->Bed(<0, Run to the CS lounge for food.>)->
CS_Lounge(<0, Just go to class.>)->PLT(<0, Fall asleep.>);

/* Fill in the above scenes with more options in a different way */
PLT.options(
     <attention,
```

```
      Pay attention.,
      FINISH(){
            print("Good job, you win!");
            exit();
      }>
);
CS_Lounge.options(
      <pizza,
      Eat the pizza.,
      FINISH(){
            print("You stole a pizza and got arrested! GAME OVER.");
            exit();
      }>
);

/* This function executes at runtime and begins the game.
def START(){
  Bed();
};
```