Language Design Document

Order Up

Programming Language

Designed By: Joseph Cruz

1. Language Data Types

The names of each data type in the “Order Up” language are named in reference to breakfast foods. The standard names of these data types and their “Order Up” equivalent are given below:

|  |  |
| --- | --- |
| **Standard Data Type** | **Order Up Data Type** |
| String | BreakfastItem[<numElem>] |
| Character | BreakfastItem |
| Integer | wholeMilk |
| Float | percentMilk |
| Bool, true, false | BaconOrSausage, bacon, sausage |

Variable declaration and assignment is as such:

**Declarations :** <DataType> @ <VariableName>;

**Assignment :** <VariableName> <- <Value>;

**Array Assignment:** <VariableName>[<expr>] <- <value>

**Example:**

BreakfastItems[20] @ helloWorld;

helloWorld <- “Hello World”;

As shown in the example above, the language will support string literals and other object literals.

2. Language Code Blocks

Blocks of code are enclosed within double curly braces like many languages with C like syntax are.

These blocks are used to inclose conditionals, loops, and functions.

2.1 Language Functions

Functions in “Order Up” are called breakfastOrders. In order to return a value these functions use the “orderUp” keyword. See the Structure below:

**Definition Structure:**

breakfastOrder <funcName>(<paramType> @ <paramName>) orderOf <retType>

{

orderUp <retData>;

}

**Definition Example:**

breakfastOrder addNumbers(wholeMilk @ a, wholeMilk @ b) orderOf wholeMilk

{

wholeMilk sum <- a + b;

orderUp sum;

}

**Calling the Function:**

wholeMilk @ Num1<- 2;

whileMilk @ Num2<- 3;

wholeMilk @ Sum <- addNumbers(num1, num2);

3. Language Control Structures

Language control structures for “Order Up” include loops and conditional statements.

3.1 Loops

roll(<counterVar> : <startNum> -> <endNum> stepBy <incPerLoop>)

{

//Loop Logic

}

Example:

wholeMilk @ i= 0;

roll(i : 0 -> 12 stepBy 1)

{

//Loop Logic

}

3.2 Conditionals

fresh(<bool>)

{

//Conditional statement is true (bacon)

}

?fresh(<bool>)

{

//This is like “else if”, <bool> is true (bacon)

}

expired

{

//All above conditions are false (sausage)

}

Conditional “is equal to” statement: <val> <==> <val>

4. IO

Simple IO is done by the following:

Print to standard output:

serveOut(<breakfastItems>);

Read from standard input:

<dataType> @ myInData <- orderIn();

5. EBNF

Key:

* [ ] means optional
* \* means infinitely repeatable
* E is the empty set
* Red text is terminal

<program> -> <functions>

<functions> -> <declarations> <function> [<declarations><function>]\*

<function> -> breakfastOrder id([<declaration>], [<declaration>]\* ) [orderOf <type>]

<block>

<block> -> { [<declarations>] [<stmnts>]}

<stmts> -> <stmnt> [<stmnt>]\*

<stmnt> -> orderUp (<expr> | E);

| id <- <expr>;

| break;

| roll(id : (<expr>) -> (<expr>) stepBy (<expr>))<block>

| fresh(<bool>) <block> [?fresh(<bool>) ]\* [expired <block>]

<expr> -> <mult> | id | <factor> | <sum>

<mult> -> <expr> (\* | / ) (<expr> | (<sum>))

<sum> -> <expr> (+ | -) ((<expr>))

<declarations> -> <declaration> [<declaration>]\*

<declaration> -> <type> @ id

<bool> -> (bacon | sausage | <equality> | <logicBool>) [ || <bool>]

<logicBool> -> <bool> && <bool>

<equality> -> <bool> <equalityOp> <bool>

<equalityOp> -> <==> | <=!=> | <= | >= | > | <

<factor> -> <bool> | num | real | char | str

<type> -> breakfastItem | wholeMilk | percentMilk | baconOrSausage | <array>

<array> -> <type>[<expr>]