**CS105:**

**COMPILER DESIGN PROJECT**

**Group C4:**

Arioja, Ryan

Dela Pena, Ezekiel

Lara, Andrea

Leabres, Kimberly

Porciuncula, Bianca

Saliba, Allison

Zurita, Kevin

**4CSC**

**Phase I. Programming Language Design**

**Requirements:**

1. ***Character set***

The language uses a subset of the UTF-8 encoding, which includes uppercase and lowercase letters, numbers, and some special characters.

∑ = {[A-Z], [a-z], [0-9], [; = + - \* / \ | & < > ( ) { } # : ‘ . , % ^ ! @ ~ ` $ “] }

* ***Case-sensitivity***

The language is case-sensitive.

* ***Control characters***

The \n is for newline.

The \t is for tab.

1. ***White spaces:***

* ***Spaces, tabs, end-of-line, and end-of-file***

The language allows spaces, tabs, end-of-line char, and end-of-file char.

* ***Comments***

Line comments are indicated by a #comment at the beginning.

Ex.

#comment This is a comment.

#comment This is also a comment.

#comment This line is indented.

#comment  
 This is  
 not a valid comment.

* ***Control characters***

Each line must end with a line break or a newline character. No trailing whitespaces must appear after each line. The file must end with a newline.

1. ***Start, end, and program name:***

The reserved word #login indicates the start of the program while the reserved word #logout indicates the end of the program. The name of the program will be placed after #login.

Ex.

#login HelloWorld  
 …  
#logout

1. ***Tokens***

* A statement (except #login and #logout) usually begins with an even number of spaces before them because of the two-space indention.
* Reserved words usually begin with a hash (#) symbol.
* Tokens may end with a space, EOL, EOF, or a delimiter.

**Division Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| #login | START | Program start |
| #logout | END | Program end |
| #newsfeed | MAIN | Main function |
| #trending | PROC\_CALL | Function call |
| #throwback | PROC\_RET | Function return |
| #share | ASSIGN | Assignment statement |
| #comment | COMMENT | Comment |
| areFriendsWith | CONCAT | String concatenate |
| identifiers | VAR | Variable name |

**Data Type Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| #ooti | DATATYPE\_INT | Integer data type |
| #ootf | DATATYPE\_FLOAT | Float data type |
| #ootc | DATATYPE\_CHAR | Character data type |
| #oots | DATATYPE\_STRING | String data type |
| #ootb | DATATYPE\_BOOL | Boolean data type |
| #ootv | DATATYPE\_VOID | Void data type |

**Loop Statement Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| #like | DO | do |
| #status | WHILE | while |
| #unfollow | BREAK | break |
| #follow | CONTINUE | Continue |

**Conditional Statement Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| #tweet | IF | if statement |
| #retweet | ELSE\_IF | else if statement |
| #reply | ELSE | else statement |

**Input and Output Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| #inbox | INPUT | Stream input |
| #outbox | OUTPUT | Stream output |

**Constants Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| ([0-9])\* | INT\_CONST | Integer constant |
| ([0-9])\*.([0-9])\* | FLOAT\_CONST | Float constant |
| ‘([A-Z]|[a-z]|[0-9]|[SPEC\_CHARS])’ | CHAR\_CONST | Character constant |
| “([A-Z]|[a-z]|[0-9]|[SPEC\_CHARS])\*” | STRING\_CONST | String constant |
| accept | BOOL\_CONST\_TRUE | Boolean true |
| decline | BOOL\_CONST\_FALSE | Boolean False |
| null | NULL | Null constant |

**Arithmetic Operator Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| ++ | INC\_OP | Increment operator |
| -- | DEC\_OP | Decrement operator |
| + | ADD\_OP | Addition operator |
| - | DIF\_OP | Difference operator |
| \* | MUL\_OP | Multiplication operator |
| % | MOD\_OP | Modulo operator |
| / | DIV\_OP | Division operator |
| ^ | EXP\_OP | Exponential operator |
| ( | LEFT\_PAREN | Left Parenthesis |
| ) | RIGHT\_PAREN | Right Parenthesis |

**Relational and Logical Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| || | OR\_OP | Short circuit OR |
| && | AND\_OP | Short circuit AND |
| < | LESS\_OP | Less than operator |
| > | GREAT\_OP | Greater than operator |
| <= | LESS\_EQ\_OP | Less than or equal operator |
| >= | GREAT\_EQ\_OP | Greater than or equal operator |
| == | EQUAL\_OP | Equal operator |
| != | NOT\_EQUAL\_OP | Not equal operator |
| ! | NOT\_OP | NOT operator / Boolean toggle operator |

**Other Tokens**

|  |  |  |
| --- | --- | --- |
| **Lexemes** | **Token Names** | **Description** |
| = | ASSIGN\_OP | Assignment operator |
| , | PARAM\_SEP | Parameter separator |
| ; | STMT\_SEP | Statement separator |
| two spaces (“ “) | INDENT | Scope indent / Nesting indent |
| “ | DQUOTE | Double quote |
| ‘ | SQUOTE | Single quote |

* ***Reserved words***

|  |  |  |
| --- | --- | --- |
| areFriendsWith | ootb | retweet |
| comment | ootc | share |
| follow | ootf | status |
| inbox | ooti | throwback |
| like | oots | trending |
| login | ootv | tweet |
| newsfeed | outbox | unfollow |
| null | reply |  |

1. ***Data types for identifiers and constants***

The data types included are ooti, ootf, ootc, oots, ootb, and ootv.

|  |  |
| --- | --- |
| ooti | integers |
| ootf | float |
| ootc | char |
| oots | string |
| ootb | boolean |
| ootv | void |

* ooti is the data type for whole numbers.
* ootf is the data type for bigger and more precise numbers. Its precision will be at most 15 digits.
* ootc is the data type that holds a single character enclosed with single quotation (‘ ‘) marks.
* oots refers to a string of characters, or an array of ootc, and enclosed with double quotation (“ ”) marks.
* ootb refers to a logical type of data that only have two values: accept which means true, and decline which means false.
* ootv is the data type for the result of a function that returns normally, but does not provide a result value to its caller.
* ***Identifiers***

They must consist of alphanumeric characters and an underscore. The first character can start only with a letter or an underscore. Camel case convention is highly encouraged.

***Regular Expression:***

([a-z] | [A-Z] | \_) ● ([a-z] | [A-Z] | [0-9] | \_)\*

***Declaration:***

#<data\_type> <identifier>

Ex.

#ooti sum  
#ootf salary

* ***Constants***

Initialization of constants are not allowed while declaration. Assignment of values must be done explicitly with an assignment operator right after declaring the variable.

***Regular Expression:***

ooti *(integers):* ([0 – 9])\*

ootf *(float):* ([0 – 9])\*.([0 – 9])\*

ootc *(char): ‘*([a-z] | [A-Z] | [0-9] | [@,#,$,…])’

oots *(string): “*([a-z] | [A-Z] | [0-9] | [@,#,$,…] | whitespaces)\*”

ootb *(boolean):* (accept | decline)

* ***Initialization***

#<data\_type> <identifier> = <value> | <expression>

Ex.

#oots name = “Cecil”  
#ootc yes = ‘Y’

1. ***Statement seperators***

A statement can end with a newline or line break. If two or more statements are present in a line, semicolons (;) must be used as a delimiter to separate each statement. Commas (,) are used to separate expressions and parameters inside parentheses ().

Ex.

#outbox “Hi!”; #outbox “Hello!”

#inbox response; #comment This line is a valid comment.

#ootv saiyan(#oots goku, #oots vegeta)

1. ***Blocks***

Off-side rule is implemented to express blocks. Any non-whitespace token to the left of the first such token on the previous line is taken to be the start of a new declaration. Two spaces ( ) is the standard token to be used for indention. The depth of indention indicates scope of a code block. Variables declared in a block have their scope only limited within the block.

Ex.

#ootb isEven(#ooti a)

#tweet (a % 2 == 0)

#outbox “Even!”

#throwback accept

#outbox “Odd!”; #throwback decline

1. ***Data conversion***

All cast operations are implicit depending on the declared data type used for the variable. I/O operations are all done in string. Void data type cannot be converted to anything.

* + ***String***

All outputs are implicitly converted into string, regardless of data type. Values of arithmetic expressions are solved before outputted. Boolean literals used for I/O are converted to true for accept, and false for decline. Any data type concatenated with areFriendsWith will be automatically converted to string.

* + ***Integer, Float***

Any input of numbers is automatically converted to integer or float (depending on the declared data type).

1. ***I/O statements***
   * ***Input***

#inbox <identifier>

Ex.

#numbers age

#inbox age

* + ***Output***

#outbox <expression>

Ex.

#characters hello = “Hello World!”

#outbox hello; #comment This prints “Hello World!”

#outbox 3 + 7; #comment This prints 10

#outbox 3 areFriendsWith 7; #comment This prints 37

1. ***Assignment Statements***

#share <identifier> <value>|<expression>

Ex.

#share one 1

#share sum #trending add(num1, num2)

1. ***Conditional Statements***
   * ***tweet-then statement***

It tells your program to execute a certain section of code only if a particular test evaluates to accept.

#tweet (condition)

<statement> | <statements>

Ex.

#ootv brakeCar() {

#comment the "tweet" clause: car must be moving

#tweet (isMoving)

#comment the "then" clause: decrease current speed

currentSpeed--

It can be executed in one line, provided that the “then” clause contains only a statement.

Ex.

#ootv brakeCar() {

#tweet (isMoving) currentSpeed--;

* + ***tweet-then-reply statement***

This statement provides a secondary path of execution when a tweet clause evaluates to decline.

#tweet (condition)

<statement> | <statements>

#reply

<statement> | <statements>

Ex.

#ootv brakeCar() {

#tweet (isMoving)

currentSpeed--;

#reply

#outbox “The car has stopped.”

Provided with a condition for the secondary path, a retweet clause can be used.

#tweet (condition)

<statement> | <statements>

#retweet (condition)

<statement> | <statements>

Ex.

#ooti testScore = 76

#ootc grade

#tweet (testscore >= 90) grade = 'A'

#retweet (testscore >= 80) grade = 'B'

#retweet (testscore >= 70) grade = 'C'

#retweet (testscore >= 60) grade = 'D';

#reply

grade = 'F'

Using the most closely nested rule, any reply shall be associated with the nearest tweet without a matching reply.

1. ***Looping Statements***

* ***status loop***

The status statement continually executes a block of statements while a particular condition is true.

status (expression)

<statement> | <statements>

The status statement evaluates expression, which must return a boolean value. If the expression evaluates to accept, the status statement executes the statement(s) in the status block. It continues testing the expression and executing its block until the expression evaluates to decline.

Ex.

#ooti count = 1

#status (count < 11)

#outbox “Count is: “ areFriendsWith count

count++

* ***like-status loop***

The like-status statement is a variant of the status statement.

#like

<statement> | <statements>

#status (condition)

The statement block is first executed, and then the condition is evaluated. If the condition returned a accept value, the first statement in the like-status will be again executed. The statement block will be executed continuously until the condition returns a decline value.

Ex.

#ooti count = 1

#like

#outbox “Count is: “ areFriendsWith count

count++

#status (count < 11)

1. ***Branching Statements***
   * ***unfollow statement***

If the unfollow statement is encountered inside any looping constructs, the rest of the statement block will be skipped and the statement following the statement block will be executed.

Ex.

#ooti count = 1

#status (count < 11)

#outbox “Count is: “ areFriendsWith count

#tweet (count == 3) #unfollow

count++; #comment Will be skipped at count = 3

* + ***follow statement***

It allows prematurely terminating the current loop body and returning the program control back to the beginning of the loop for a new iteration.

Ex.

#ooti count = 1

#status (count < 11)

#outbox “Count is: “ areFriendsWith count

#tweet (count == 3) #follow

count++; #comment Will NOT be skipped at count = 3

1. ***Expressions***

An expression is a construct made up of variables, operators, and function invocations, which are constructed according to the syntax of the language, which evaluates to a single value. The data type of the value returned by an expression depends on the elements used in the expression.

* ***Arithmetic Expressions***

These are composed of integers/floats, arithmetic operators, parentheses, and function invocations that return mathematical value.

Ex.

x^2 + 2x + 1

2 \* (2 \* 2(2 \* 2(2 \* 2(2 \* 2)))))

1 + 2 – 3 \* 4 / 5 ^ 6 (7)

* ***String Expressions***

A string literal can be evaluated in these expressions.

Ex.

“Foobar”

“I “ areFriendsWith “Love “ areFriendsWith #trending you()

“This variable has this “ areFriendsWith value

* ***Conditional/Boolean Expressions***

Any relational or logical expressions that returns boolean values.

Ex.

you && me

!(brandy) || (beer)

this != funny

***Arithmetic Operators***

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ++ | Increment operator |
| -- | Decrement operator |
| + | Addition operator |
| - | Difference operator |
| \* | Multiplication operator |
| / | Division operator |
| % | Modulo operator |
| ^ | Exponential operator |
| ( | Left Parenthesis |
| ) | Right Parenthesis |

***Relational Operators***

|  |  |
| --- | --- |
| **Operator** | **Description** |
| < | Less than operator |
| > | Greater than operator |
| <= | Less than or equal operator |
| >= | Greater than or equal operator |

***Logical Operators***

|  |  |
| --- | --- |
| **Operator** | **Description** |
| || | Short circuit OR |
| && | Short circuit AND |

***Equality Operators***

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | Equal operator |
| != | Not equal operator |

***Inverse Operators***

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ! | NOT operator / Boolean toggle operator |

***Operator Precedence Table***

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Level** | **Associativity** |
| expr++  expr--  () | post-increment  post-decrement  parentheses | 1 | L to R |
| ++expr  --expr  +  -  ! | pre-increment  pre-decrement  unary plus  unary minus  logical NOT | 2 | R to L |
| ^ | exponential | 3 | R to L |
| \*  /  % | multiplicative | 4 | L to R |
| +  - | additive | 5 | L to R |
| < <=  > >= | relational | 6 | L to R |
| ==  != | equality | 7 | L to R |
| && | logical AND | 8 | L to R |
| || | logical OR | 9 | L to R |
| = | assignment | 10 | R to L |

1. ***Limitations***

* The language is not object oriented
* Strict rules for nesting must be followed for proper blocking
* newsfeed is always required for the program

**PROGRAM ::=** START ID **<FUNCTIONS>** END

**<FUNCTIONS>** **::=** NEST **<MAIN\_FUNCTION>** **<SUB\_** **FUNCTIONS>**

**<MAIN\_FUNCTION> ::=** MAIN <**STATEMENTS>**

**<SUB\_FUNCTIONS> ::=** NEST **<SUB\_** **FUNCTION>**

| NEST **<SUB\_** **FUNCTION> <SUB\_FUNCTIONS>**

| ε

**<SUB\_FUNCTION>** **::=** **<DATATYPE>** ID LEFT\_PAREN **<DECL\_PARAM>** RIGHT\_PAREN **<STATEMENTS>**

**<DECL\_PARAM> ::=** **<DATATYPE>** ID PARAM\_SEP **<DECL\_PARAM>** | **<DATATYPE>** ID

**<STATEMENTS> ::=** NEST **<STATEMENT> <STATEMENTS>**

| NEST **<MORE\_STATEMENT> <STATEMENTS>**

| NEST **<MORE\_STATEMENT>**

| NEST **STATEMENT**

**<MORE\_STATEMENT> ::=** **<STATEMENT>** STMT\_SEP **<STATEMENT>**

| **<STATEMENT>** STMT\_SEP **<MORE\_STATEMENT>**

**<STATEMENT> ::= <DECLARATION>**

| **<ASSIGNMENT>**

| **<IO>**

| **<CONTROL\_FLOW>**

| <**COMMENTS>**

| **<EXPR\_STATEMENTS>**

| **<BRANCHING>**

| **<RETURN>**

**<DECLARATION> ::= <DATATYPE>** ID | <**DATATYPE>** ID ASSIGN­\_OP CONSTANT

**<ASSIGNMENT> ::=** ASSIGN ID **<EXPRESSIONS>**

**<IO> ::=** **<INPUT\_STMT>** | **<OUTPUT\_STMT>**

**<CONTROL\_FLOW> ::= <CONDITIONAL>** | **<LOOPING>**

**<COMMENTS> ::=** COMMENT **<GIBBERISH>**

**<EXPR\_STATEMENTS> ::= <INC\_STMT>** | **<DEC\_STMT>**

| PROC\_CALL ID LEFT\_PAREN CALL\_PARAM RIGHT\_PAREN

**<BRANCHING> ::=** BREAK | CONTINUE

**<RETURN> ::=** PROC\_RETURN **<EXPRESSIONS>** | PROC\_RETURN **<VALUE>**

**<DATATYPE> ::=** DATATYPE\_INT

| DATATYPE\_FLOAT

| DATATYPE\_CHAR

| DATATYPE\_STRING

| DATATYPE\_BOOL

| DATATYPE\_VOID

**<INPUT\_STMT> ::=** INPUT ID

**<OUTPUT\_STMT> ::=** OUTPUT **<EXPRESSIONS>**

**<CONDITIONAL> ::= <IF\_STMT>** | **<IF\_ELSEIF\_STMT>** | **<IF\_ELSE\_STMT>**

**<LOOPING> ::= <WHILE\_STMT>** | **<DO\_WHILE>**

**<GIBBERISH> ::= <GIBBER>** | NEST **<GIBBER> <GIBBERISH>**

**<EXPRESSIONS> ::= <MATH\_EXPR>** | **<STRING\_EXPR>** | **<BOOL\_EXPR>**

| PROC\_CALL ID LEFT\_PAREN **<CALL\_PARAM>** RIGHT\_PAREN

**<IF\_ELSEIF\_STMT> ::= <IF\_STMT> <ELSEIF\_STMTS>**

**<IF\_ELSE\_STMT> ::= <IF\_STMT> <ELSE\_STMT>**

**<ELSEIF\_STMTS> ::= <ELSEIF\_STMT> <ELSEIF\_STMTS>** | **<ELSEIF\_STMT>**

**<IF\_STMT> ::=** IFLEFT\_PAREN **<BOOL\_EXPR>** RIGHT\_PAREN **<STATEMENTS>**

|IFLEFT\_PAREN **<BOOL\_EXPR>** RIGHT\_PAREN **<STATEMENT>**

**<ELSEIF\_STMT> ::=** ELSE\_IF LEFT\_PAREN BOOL\_EXPR RIGHT\_PAREN **<STATEMENTS>**

| ELSE\_IF LEFT\_PAREN BOOL\_EXPR RIGHT\_PAREN **<STATEMENT>**

**<ELSE\_STMT> ::=** ELSE **<STATEMENTS>** | ELSE **<STATEMENT>**

**<WHILE\_STMT> ::=** WHILE LEFT\_PAREN BOOL\_EXPR RIGHT\_PAREN **<STATEMENTS>**

| WHILE LEFT\_PAREN BOOL\_EXPR RIGHT\_PAREN **<STATEMENT>**

**<DO\_WHILE> ::=** DO **<STATEMENTS>** WHILE LEFT\_PAREN **<BOOL\_EXPR>** RIGHT\_PAREN

**<INC\_STMT> ::= <PRE\_INC>** | **<POST\_INC>**

**<DEC\_STMT> ::= <PRE\_DEC>** | **<POST\_DEC>**

**<MATH\_EXPR> ::= <MATH\_EXPR>** ADD\_OP **<MATH\_EXPR2>**

| **<MATH\_EXPR>** DIF\_OP **<MATH\_EXPR2>** | **<MATH\_EXPR2>**

**<MATH\_EXPR2> ::= <MATH\_EXPR2>** MUL\_OP **<MATH\_EXPR3>**

| **<MATH\_EXPR2>** DIV\_OP **<MATH\_EXPR3>**

| **<MATH\_EXPR2>** MOD\_OP **<MATH\_EXPR3>** | **<MATH\_EXPR3>**

**<MATH\_EXPR3> ::= <MATH\_EXPR4>** EXP\_OP **<MATH\_EXPR3>** | **<MATH\_EXPR4>**

**<MATH\_EXPR4> ::=** LEFT\_PAREN **<MATH\_EXPR>** RIGHT\_PAREN | **<VALUE>**

**<STRING\_EXPR> ::= <VALUE>** | **<VALUE>** CONCAT **<STRING\_EXPR>**

**<REL\_EXPR> ::= <REL\_EXPR>** OR\_OP **<REL\_EXPR2>** | **<REL\_EXPR2>**

**<REL\_EXPR2> ::= <REL\_EXPR2>** AND\_OP **<REL\_EXPR3>** | **<REL\_EXPR3>**

**<REL\_EXPR3> ::= <REL\_EXPR3>** EQUAL\_OP **<REL\_EXPR4>**

| **<REL\_EXPR3>** NOT\_EQUAL\_OP **<REL\_EXPR4>**

**<REL\_EXPR4> ::= <REL\_EXPR4>** GREAT\_OP **<REL\_EXPR5>**

| **<REL\_EXPR4>** LESS\_OP **<REL\_EXPR5>**

| **<REL\_EXPR4>** GREAT\_EQ\_OP **<REL\_EXPR5>**

| **<REL\_EXPR4>** LESS\_EQ\_OP **<REL\_EXPR5>**

| **<REL\_EXPR5>**

**<REL\_EXPR5> ::=** NOT\_OP **<REL\_EXPR>** | **<VALUE>**

**<PRE\_INC> ::=** INC\_OP ID

**<POST\_INC> ::=** ID INC\_OP

**<PRE\_DEC> ::=** DEC\_OP ID

**<POST\_DEC> ::=** ID DEC\_OP

**<GIBBER> ::=** STRING\_CONST

**<VALUE> ::= <CONSTANTS>** |ID

**<CONST> ::=** INT\_CONST | FLOAT\_CONST | CHAR\_CONST

| STRING\_CONST | **<BOOL\_CONST>** | NULL

**<BOOL\_CONST>** **::=** BOOL\_CONST\_TRUE | BOOL\_CONST\_FALSE