**POLYTECHNIC UNIVERSITY OF THE PHILIPPINES** Anonas, Sta.Mesa, Maynila,

Kalakhang Maynila



**Snek Mini Programming Language Documentation**

**BSCS 3-4**

**Department of Computer Science**

Chacon, Alissa Carla

Miranda, Allyza

Obias, Jovelyn

Cepe, Carl Joseph

Mojado, Mathew

Parilla, Aaron

Schandler, Jim

**Mr. Montaigne Molejon**

Professor

**I. INTRODUCTION**

Snek is a newly conceptualized mini programming language designed by computer science students in Polytechnic University of the Philippines . This new programming language is heavily inspired by famous programming languages such as C and Python. It was mainly developed as a system programming language to perform basic functions. Main feature of Snek language is it has a simple set of keywords and clean style. Most notable part of Snek is its ability to distinguish reserved words from user-generated entities with the help of apostrophe (‘). These features make this programming language suitable for beginners. It emphasizes code readability and minimal learning curve for newcomers in the field of computer programming.

Snek has a programming paradigm which is almost similar to a structured language. Given that Snek is merely a mini language, fundamental instructions namely repetition, selection and sequence are at most tasks that can be performed.

**II. SYNTACTIC ELEMENTS OF A LANGUAGE**

1. Character Set

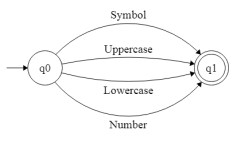
<Character> 🡪 {Letter, Number, Symbol}

<Letter> 🡪 {uppercase, lower-case}

<Uppercase>🡪 {A…Z}

<Lowercase>🡪 {a…z}

<Number> 🡪 {0,1,2,3,4,5,6,7,8,9}

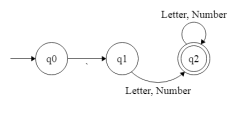
<Symbol> 🡪 { (, ), {, }, !, =, >, <, |, /, ”, ,, + , - , %, \*, ; ,. , :} 

2. Identifiers

∙ Always starts with the symbol “ ` ”.

∙ Followed by series of letters or numbers

∙ It must be case sensitive.



3. Operation Symbols

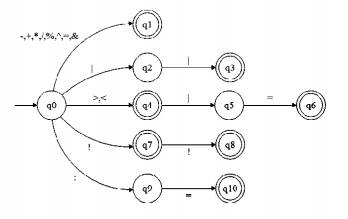
<Arithmetic> 🡪 {+, -, /, \*, %, ^}

<Logic> 🡪 {||, &, !}

<Relational> 🡪 {>, <, =, !!, >|=, <|=}

<Assignment>🡪 {:=}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ARITHMETIC** | | **LOGIC** | | **RELATIONAL** | | **ASSIGNMENT** | |
| + | Addition | || | Logic or | < | Less than | := | Takes |
| - | Subtraction | & | Logic and | > | Greater than |  |  |
| / | Division | ! | Logic not | = | Equal |  |  |
| \* | Multiplicatio n |  |  | <|= | Less than or equal |  |  |
| ^ | Exponential |  |  | >|= | Greater than or equal |  |  |
|  |  |  |  | !! | Not equal |  |  |



4. Noise words

|  |  |
| --- | --- |
| **NOISE WORDS** | **DESCRIPTION** |
| start  Syntax: Main (start) | Indicates the start of the program |
| end  Syntax: Main (end) | Indicates the start of the program |

S t a r tQ6 Q1 Q2 Q3 Q4 Q5 Q6 

E n d 

Q1 Q2 Q3 Q4

Q4

5. Keywords/ Reserved words

|  |  |  |
| --- | --- | --- |
| **Keywords** | **Syntax** | **Description** |
| Main | Main(start) | Starting point for program execution. |
| Outdis | outdis (“text”);  outdis (“text” – identifier); | Is used to print string literals and values of the identifiers onto the output screen. |
| inscan | inscan identifier := value | Is used to read character, string, numeric data from keyboard. |
| integral | integral | Define numeric variables holding both positive and negative numbers. |
| decimal | decimal | Define numeric variables holding numbers with decimal points. |
| text | text | A character or sequence of characters |
| incase | incase(condition) | Statement is responsible for modifying the flow of execution of a program. incase statement is always used with a condition.  The condition is evaluated first before executing any statement inside the body of incase. Statement evaluates the test expression inside the  parenthesis () |
| then | then {statements} | If the condition in incase will be evaluated as true, statements under “then” will be performed. |
| else | else {statements} | Will be executed if the condition/s in the incase block is/are not met. |
| amid | amid (condition) | Repeatedly executes a target  statement as long as a given condition is true. |
| act | act {statements} | Similar to amid, except that it is guaranteed to execute at least one time. |
| forloop | Forloop (initialization;  condition; arithmetic  expression) | Used for executing a block of  statements repeatedly until a given condition returns false |

M a i nQ5Q5 

Q1 Q2 Q3 Q4

oi sQ5 Q5 Q6 

u t dQ7

Q1 Q2 Q3 Q4

i n sQ6 Q5 ca nQ7 Q1 Q2 Q3 Q4 Q5 

s e 

i n c aQ6

Q1 Q2 Q3 Q4 Q5 

Q7Q7

m a

d e c iQ6 Q7 Q1 Q2 Q3 Q4 Q5 

l

Q8 

Q8

t e x tQ1Q5 

Q1 Q2 Q3 Q4

i n t e g r 

Q1 Q2 Q3 Q4

Q5 Q6 Q7 a

l

t h

Q1 

Q9

e nQ5

Q8

Q3 Q1 Q2 Q3 Q4 

el s eQ5 Q2 Q1 Q2 Q3 Q4 

a m i dQ5 Q4 Q1 Q2 Q3 Q4 

a c t

Q1 Q2 Q3

Q4

6. Comments

∙ Comments are used to provide supplementary information making it easier to understand the source code of the computer program. The comments are generally ignored by the interpreter.

∙ One line comment should start with the symbol “>>”, followed by any combinations of letters and numbers, ends in next line

∙ Multiple line comment should start with the symbol “>>” and end with “<<”

∙ All the strings after this symbol “>>” would be ignored by the compiler and starts to be recognized again after “<<” for multiple lines and end of line for single line comment

|  |  |  |
| --- | --- | --- |
| **Comment Style** | **Syntax** | **Sample** |
| **Line comments** – delimit a region of source code for a single line only.  The symbol “>>” indicate the beginning of a  comment.  A newline character  indicates the end of a line comment. | >> | >> This is a line comment |
| **Block comments** – delimit a region of source code which may span multiple lines.  The symbol “ >>” indicate the beginning and the symbol “<< “ for the end of the block comment. | >> << | >>This is a  Block  comment << |

7. Blanks

∙ The use of blank spaces can improve the style of a program. It improves the readability of a program. A blank space is

required between consecutive words in a program line.

∙ Blank space does not correspond to a visible mark, but it actually occupies an area on a page. When the lexical analyzer encounters a blank space, it indicates the end of a lexeme

thus, the lexical analyzer will look up for that certain lexeme from the list.

8. Delimiters and Braces

|  |  |  |
| --- | --- | --- |
| **DELIMITERS** | **DESCRIPTION** | **SAMPLE** |
| ; | Semicolons are used to identify the end of line in a line of code | Integral ‘a; |
| {} | Curly Brackets are used to signify a block of code | Main(START){  …...  }Main(END) |
| “” | Double quotes are used to identify String Literals | outdis(“Random”); |
| , | Commas are used to  separate data data fields | outdis(“text” ~ var ~ “text, ‘var); |
| () | Open and Closed  Parenthesis are used as a field for parameters,  String Literals, etc. | outdis(“Text”); |
| >><<  >> | Double Angle Brackets are used for representing comments | >>Multiple-line  comment<<  >>One line comment |

9. Expression

9.1 Arithmetic Expression

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Precedence** | **Operators** | **Order of Evaluation** | **Example** | **Result** |
| 4 | () | If the parentheses are  nested, expression in the innermost pair is evaluated first. If there is several pair of parentheses on the same  level, they are evaluated from left to right. | ((3+2)\*(8/2))  5 \* 4  20 | 20 |
| 3 | ^ | If there are no parentheses in the expression, this will always be evaluated first | 6/3+2^2  6/3+ 4  6/3+ 4  2 + 4  6 | 6 |
| 2 | \*, / % | If there are several on the same level, they are  evaluated from left to right. | 10%7\*6/9  3 \*6/9  18/9  2 | 2 |
| 1 | +, - | If there are several on the same level, they are  evaluated from left to right. | 2+3-1-2+10  5 -1-2+10  4 -2+10  2 +10  12 | 12 |

|  |
| --- |
| **Example combination of all arithmetic operators**  ((45-12\*2) + (20/5%1) \* 3^2) 🡪 Inner parentheses first, left to right  ((45- 24)) + (20/5%1) \* 3^2) 🡪 \* has a higher precedence  ( 21 + (20/5%1) \* 3^2) 🡪 First inner parentheses has been evaluated  ( 21 + ( 4 %1) \* 3^2) 🡪 All operators on the second inner parentheses is on the same level of precedence, left to right  ( 21 + 0 \* 3^2) 🡪 Second inner parentheses has been evaluated  ( 21 + 0 \* 9) 🡪 ^ has the highest precedence among all the remaining operators ( 21 + 0) 🡪 \* has higher precedence than +  20 🡪 Done evaluating all operators |

9.2 Relational Expression

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Precedence** | **Operators** | **Order of Evaluation** | **Example** | **Result** |
| Equal  priority | <  >  >|=  <|=  =  !! | Can only have one  relational operator in a pair of parentheses.  Relational expressions can only be performed  through logical operator. It doesn’t matter what relational expression is evaluated first, it will  always result to the same answer. | (2 > 4) & (5 < 10) || (4 !! 3)  false & (5 < 10) || (4 !! 3)  false & true || (4 !! 3)  false & true || true  false || true  true  (2 > 4) & (5 < 10) || (4 !! 3)  (2 > 4) & (5 < 10) || true  false & (5 < 10) || true  false & true || true  false || true  true | true |

9.3 Logical expression

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Precedence** | **Operators** | **Order of Evaluation** | **Example** | **Result** |
| 4 | ! | Highest precedence among all operators. | (2>1) & !(6=2) || (5<10)  (2>1) & !(false) || (5<10)  (2>1) & true || (5<10)  true & true || (5<10)  true || (5<10)  true || true  true | true |
| 3 | () | If the parentheses are nested, expression in the innermost pair is evaluated first. If there is several pair of parentheses on the same level, they are evaluated from left to right. Relational expressions being evaluated by the logical operator should always be  enclosed in parentheses. | (((9 < 2) || (6 >|= 2)) & (4=1)) (( false || (6 >|= 2)) & (4=1)) (( false || true) & (4=1)) (( false || true) & (4=1)) ( true & (4=1)) ( true & false )  false | false |
| 2 | & | After all the expressions inside parentheses have been evaluated, this will always be evaluated first if  there are other logical operators on the same level pair of parentheses. If all are the same operators on the  same level, it will be evaluated from left to right. | ((6 > 4) || (1 < 10) & (2 >|=5)) ( true || true & false ) ( true || false ) true | true |
|  | || | Last to be evaluated. Left to right evaluation also if same operators on the same level pair of  parentheses. | ((6 > 4) || (1 < 10) & (2 >|=5)) ( true || true & false ) ( true || false ) true | true |
| **Example combination of all logical operators**  (((8 >|= 4) || (30 < 8) & (2 >|=5)) & !((3 >6)) 🡪 logic not to be evaluated first  (((8 >|= 4) || (30 < 8) & (2 >|=5)) & !(false)) 🡪 perform the expression inside  (((8 >|= 4) || (30 < 8) & (2 >|=5)) & true ) 🡪 invert its value  (( true || false & false ) & true ) 🡪 parentheses on the same level evaluated (( true || false ) & true ) 🡪 & has higher precedence but || is inside parentheses ( true & true ) 🡪 & is last to be evaluated since it is outside the parentheses **true** which has higher precedence | | | | |

9.4 All expressions

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operators** | **Order of Evaluation** | **Example** |
| 6 | ! | Highest precedence on all operators. | X = 5  Y = 2  Z = 3  (y = x + 3 ) > 10 || 2= Z & (2+Y <|= X & !(Z = Y + X\*2 > 20 )) (y = x + 3 ) > 10 || 2= Z & (2+Y <|= X & !(Z = Y + 10 > 20 )) (y = x + 3 ) > 10 || 2= Z & (2+Y <|= X & !(Z = 12 > 20 ))  X = 5  Y = 2  Z =12  (y = x + 3 ) > 10 || 2= Z & (2+Y <|= X & !( 12 > 20 )) (y = x + 3 ) > 10 || 2= Z & (2+Y <|= X & !false ) (y = x + 3 ) > 10 || 2= Z & (2+Y <|= X & true ) (y = 8 ) > 10 || 2= Z & (2+Y <|= X & true )  X = 5  Y = 8  Z =12  8 > 10 || 2= Z & (2+Y <|= X & true ) 8 > 10 || 2= Z & ( 10 <|= X & true ) 8 > 10 || 2= Z & ( false & true ) 8 > 10 || 2= Z & false  false || 2= Z & false  false || false & false  false || false  **false** |
| 5 | () | If the parentheses are nested,  expression in the  innermost pair is  evaluated first. If  there is several pair of parentheses on the same level, they are evaluated from left to right. |
| 4 | Arithmetic Operators | Precedence of  arithmetic operators are also applied. |
| 3 | Relational Operators | Precedence of  relational operators is also applied. |
| 2 | Logical  Operators | Precedence of logical operators is also  applied. |
| 1 | = | Lowest precedence among all operator hence, last to be  evaluated |

∙ In an expression, operator with the highest precedence is grouped with its operands first, then the next highest operator will be grouped with its operands and so on. If there are several operators of the same precedence, they will be examined left to right.

∙ All expression should always start with either a brace(parenthesis only), symbol “ ’ ” (for variable/identifier), numbers (for constant values)

Example:

⮚ (‘x + 3)

⮚ ‘x \* 3

⮚ 3 / ‘x

∙ Every opening braces should have its corresponding close braces

⮚ (('Y + ‘X – 3) > 5)

∙ First operand which can be letters or numbers will be followed by an operator and ends with another operand composed of letters or numbers, or can also end with braces to partner its corresponding open brace.

⮚ ‘Y % 4

⮚ (‘Y > ‘X) && (‘Z <|= ‘X)

∙ It can only start with an operator in logic expression using the operator “!” followed by a relational expression or logical expression which inverts their resulting logical value – true or false.

⮚ !(4 < 5)

∙ Arithmetic operators cannot perform operation between relational expressions and logic expressions. Only for letters/numbers

⮚ (3 > 4) +( 6 < 4) 🡪 wrong

⮚ (3 > 4 && 6 =2) - (‘X < 4 || 3 >|= ‘Y) 🡪 wrong

∙ Relational operator cannot perform operation between logic expressions ⮚ (3 > 4 && 6 =2) >|= (‘X < 4 || 3 >|= ‘Y) 🡪 wrong

∙ Logic operator can only perform logic and relational expressions

⮚ (Y > X) && (Z <|=X)

⮚ (Y > X) || (Z <|=X && 3 > 4)

⮚ ! (‘X = 2)

⮚ !(‘X + 3) 🡪

10.Statements

|  |  |
| --- | --- |
| **Syntax** | **Example** |
| Comment | |
| >> <character> one line | >> uno is life |
| >>  <character> two or more lines << | >>  This code block  computes for the  total price  << |
| Declaration | |
| <data\_type> <identifier>; | integral `a; |
| Data type | |
| <data\_type> <identifier>; | integral `catNumber; |
| <data\_type> <identifier>; | decimal `average; |
| <data\_type> <identifier>; | text `studentName; |
| Declaration plus initialization/assignment | |
| <data\_type> <identifier> <assignment\_operator> literal\_value; | decimal `dogNumber := 32.3; |
| <data\_type> <identifier> <assignment\_operator>  “string\_literal\_value”; | text `dogName := “Manny”; |
| <data\_type> <identifier> <assignment\_operator> <identifier>; | Integral `tigerNumber := `lionNumber; |
| <data\_type> <identifier> <assignment\_operator>  <arithmetic\_expression>; | decimal `mouseNumber := 32\*2+3; |
| Initialization/assignment  (assigning of initial value separated from the declaration) | |
| <identifier> <assignment\_operator> literal\_value; | `dogNumber := 3; |
| <identifier> <assignment\_operator> “string\_literal\_value”; | `catName := “Chico”; |
| <identifier> <assignment\_operator> <identifier>; | `tigerNumber := `dogNumber; |
| <identifier> <assignment\_operator> <arithmetic\_expression>; | `mouseNumber := 2+10/5; |
| Assignment  (assigning of new value to a variable with a value already) | |
| <identifier> <assignment\_operator> literal\_value; | `deerNumber := 13; |
| <identifier> <assignment\_operator> “string\_literal\_value”; | `catName := “elChoco”; |
| <identifier> <assignment\_operator> <identifier>; | `birdNumber := `eagleNumber; |
| <identifier> <assignment\_operator> <arithmetic\_expression>; | `average := 2+10/5; |
| Input | |
| inscan <identifier> <assignment\_operator> literal\_value; | inscan a := 28; |
| inscan <identifier> <assignment\_operator>  “string\_literal\_value”; | inscan b := “Argentina”; |
| Output | |

|  |  |  |
| --- | --- | --- |
| Syntax | Example | Output |
| outdis (“string\_literal\_value” ); | outdis (“text”); | text |
| outdis (“string\_literal\_value” ~<identifier>~  “string\_literal\_value”, `<identifier>); | x = "This";  outdis("that" ~ x ~ "there", 'x); | thatThisther |
| outdis(<identifier>); | x=1;  outdis(x); | 1 |
| Conditional | | |
| incase, then, else | | |
| incase(condition) then{statements} | incase(x=3) then{  y:= “cat”;  z:= 3;  x:= z+1;  } | |
| Incase(condition) then{statements} else{statements} | incase(x=3) then{  y:= “cat”;  z:= 3;  x:= z+1;  } else { y:= “dog”;  z:= 4;  x:= z+2;  } | |
| Loop/iteration | | |
| amid, act | | |
| amid(condition){statements} | amid(x=z)  { y:= zebra  integral total := 1+2+3+4;  x:=x+1;} | |
| act{statements}amid(condition); | act  {y:= zebra  integral total := 1+2+3+4;  x:=x+1;}  amid(x=z); | |
| for | | |
| forloop (initialization; condition; arithmetic expression){statements} | forloop (integral x:=0;x<|=10;x:=x+1) {:= zebra  integral total := 1+2+3+4;} | |

III. Design Issues

Upon constructing Snek programming language, drawbacks and potential downsides to the implemented structural design have been anticipated by Snek developers. These drawbacks are:

• Readability for identifiers with extensive use of numbers or only use numbers.

• Lacks capital letters for keywords/reserved words and basic syntax. • Case Sensitive, same name with different capital letters would result in different meanings.

• Identifiers do not support the usage of symbols.

• Has no support for switches.

• Does not support terminating keywords and relies on symbols and brackets.

• Indentations don’t affect code structure, could potentially result in messy code blocks.

• Very few syntactic sugar and syntactic salt.

• Built to avoid overly verbose code.

• Potentially too terse.

• Has no candy grammar feature.

• Lacks support for abstraction.

• Lacks support for extensibility.