

BSCS 3-4

Group 3

Members: Concepcion, Freanne Lei
Crisostomo, Joseph Karl
Delos Santos, Rons Marie
Fernandez, Justine Mikaela
Martinez, John Lloyd
Reoloso, Jirro
Villanueva, Ian Kirk

NAME: sPyC

INTRODUCTION:

sPyC is a strongly typed, static, high-level programming language. The goal of the said language is to enable non-technical users to code without much knowledge and to help them be ready if they want to learn harder existing languages. It is great for beginners and it makes coding more intuitive. Another purpose of this language is to emphasize productivity and code readability by adapting features from different well established programming languages like Python and C.

The name “sPyC” derived from the word spicy, and the programming languages, Python and C. The name sPyC presents a clever twist to the word ‘spicy’ that gives a comical zest for a language that is mostly based on Python and C languages. Like its conventional word counterpart, sPyC aims to add flavor and spice to these existing languages, by providing efficiency, portability and readability. Allowing the user to be immersed in such tasteful and well ordered programming language without being overwhelmed. A combination of both organization and productivity surely boosts the user-friendliness of this language.

sPyC is based and influenced by C and Python. C is a general purpose procedure-based programming language and Python is a backend, high-level, and an object-oriented programming language. The syntax of the proposed language is Python based but it has a C-like structure. With the increased productivity of Python combined with the readability of the C structure, programmers will surely love the efficiency it provides, making it an ideal language for those who are new to programming. It is a good programming language that serves as the building block towards understanding key programming concepts. sPyC is in a good enough state to function well, placing it in a position where it

can be easily improved. Its functions can accept parameters and return values and perform a variety of tasks like input from the user, displaying the information, etc. In summary, it is simple and easy to learn and use with its builtin functions, operators, and keywords that are relatively shorter and more portable. sPyC will work best in small projects where performance is important, undoubtedly, it helps beginners because of its easy syntax and wide range applications/opportunities.

sPyC would be a simple, powerful and an adaptable programming language. It's readability would make it an excellent programming language for beginners because it allows them to think like a programmer without getting slowed down by complicated syntax. To make it more approachable for all users, we aim for sPyC's keywords to be comprehensible and we refrain from using technical terms.

SYNTACTIC ELEMENTS OF LANGUAGE

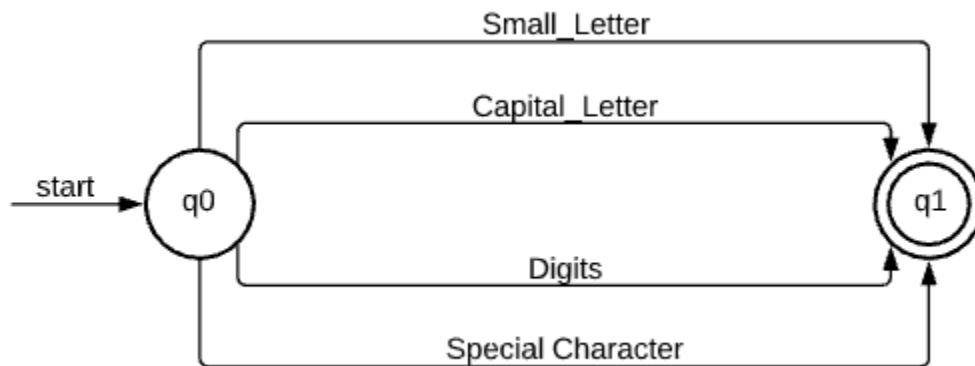
1. Character Set

- **Character** = {Alphabet, Digits, Special Characters}
- **Alphabet** = {Small_Letter, Capital_Letter}
- **Small_Letter** = { a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
- **Capital_Letter** = { A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}
- **Digits** = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- **Special Characters**

Symbol	Name	Symbol	Name
,	comma	(left parenthesis
.	period)	right parenthesis
;	semicolon	%	percent sign
#	number sign	&	ampersand
/	slash	^	caret
\	backslash	*	asterisk
'	apostrophe	-	minus
“	quotation mark	+	plus
!	exclamation mark	[left bracket
	vertical bar]	right bracket

<	opening angle bracket	{	left brace
>	closing angle bracket	}	right brace
	space		

Machine for Character Set



2. Identifiers

Rules:

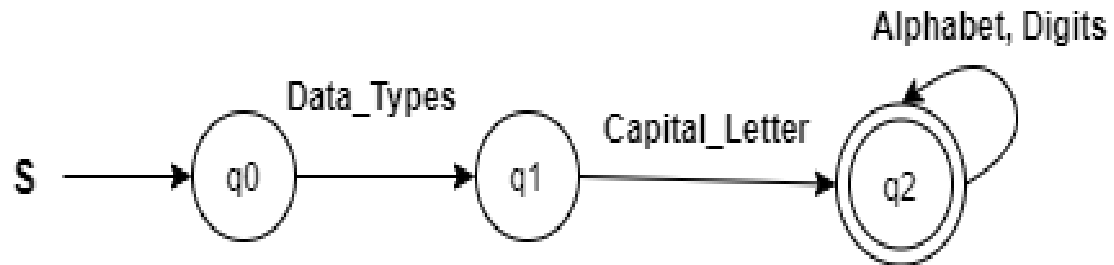
- Identifiers can only have alphanumeric characters (a-z, A-Z, 0-9).
- Identifiers must begin with its data type followed by a capital letter (A-Z).
- After the data type and the capital letter, the case of the letters is significant. For example, the identifiers intAGE and intAge will be recognized as two unique identifiers.
- Keywords and reserved words are not allowed to be used as identifiers.
- There are no special characters allowed in or as an identifier, such as underscore (`_`), a semicolon (`;`), period (`.`), whitespaces (), slash (`/`), comma (`,`), and so on.
- The length of an identifier is unlimited.
- For functions and class declarations, rules b and c do not apply.

Alphabet = {Small_Letter, Capital_Letter}

Digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

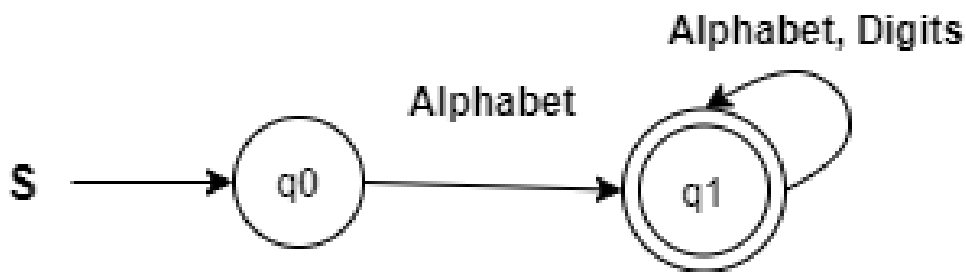
Data_Types = {int, float, double, char, str}

Machine for Identifiers



Regular Expression: (Data_Types)(Capital_Letter)(Alphabet+Digits)*

Machine for Class and Function Identifiers



Regular Expression: Alphabet(Alphabet+Digits)*

3. Operation Symbols

a. Arithmetic operations

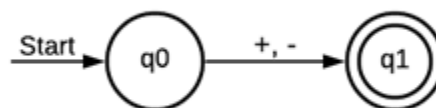
Unary Operations unary op: integer -> integer

Operator	Description	Example
+	Indicates a value is a positive number; hence, not showing much significance	+4 returns the result 4

-	Negates a value. Returning the negated value as its operand: zero to zero, positive to negative, and negative to positive	-3 returns the result -3
---	--	--------------------------

- The + operator in Python can be used in both the binary and unary form. The binary form means add, returning a result that is the standard arithmetic sum of its operands. The unary form means identity, returning the same value as its operand. [2]
- The - operator in Python can be used in both the binary and unary form. The binary form means subtract, returning a result that is the standard arithmetic difference of its operands: left operand minus right operand. The unary form means negate, returning the negated value as its operand: zero to zero, positive to negative, and negative to positive. [2]

Machine for Unary Operators:

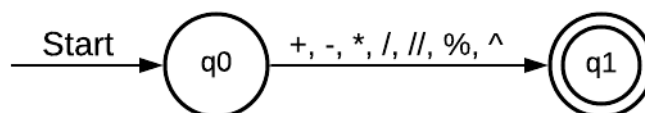


Binary Operations binary op: integer + integer -> integer [3]

Operator	Name	Description	Example in Symbols
+	Addition Operator	Used to add two operands	$x + y$
-	Subtraction Operator	Used to subtract the second operand from the first operand.	$x - y$
*	Multiplication Operator	Used to multiply the operands	$x * y$

/	Division Operator	Used to divide left operand by the right one	x / y
//	Floor Division Operator	Normal division operation and returns floor value for both integer and floating point arguments	$x // y$
%	Modulus Operator	Outputs the remainder of the division of left operand by the right	$x \% y$ (remainder of x divided by y)
^	Exponentiation Operator	Used to express exponents where left operand raised to the power of right	$x ^ y$ (x to the power of y)

Machine for Binary Operators:



b. Boolean operations

b.1. Relational ex. <,>==

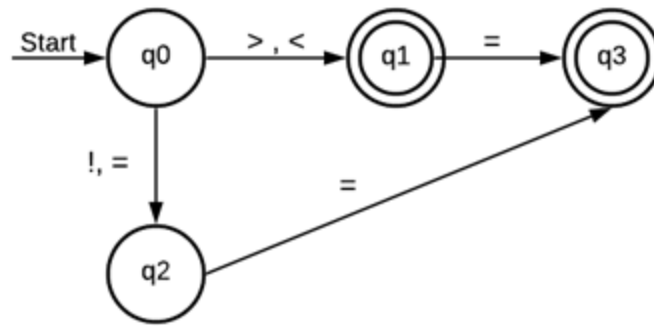
Operator	Name	Description	Example
!=	Not Equal to	returns true if the value on the left is not equal to the value on the right, otherwise it returns false.	$x != y$

==	Equal to	returns true if the value on the left is equal to the value on the right, otherwise it returns false.	$x = y$
>	Greater Than	returns true if the value on the left is greater than the value on the right, otherwise it returns false.	$x > y$
<	Less Than	returns true if the value on the left is less than the value on the right, otherwise it returns false.	$x < y$
<=	Less Than or Equal	returns true if the value on the left is less than or equal than the value on the right, otherwise it returns false.	$x <= y$
>=	Greater Than or Equal	returns true if the value on the left is greater than or equal than the value on the right, otherwise it returns false.	$x >= y$

Note: Relational operators have left to right associativity. Left to right associativity means that when two operators of the same precedence are adjacent, the leftmost operator is evaluated first. It is noted as necessary, to enclose relational expressions in parentheses to improve readability.

Characters and Strings can also be compared using relational characters. When a relational operator is used with strings, the integer value of each character of the left operand is compared to the integer value of each character of the right operand working from left to right.

Machine for Relational Operators:



b.2 Logical ex. Not, and, or

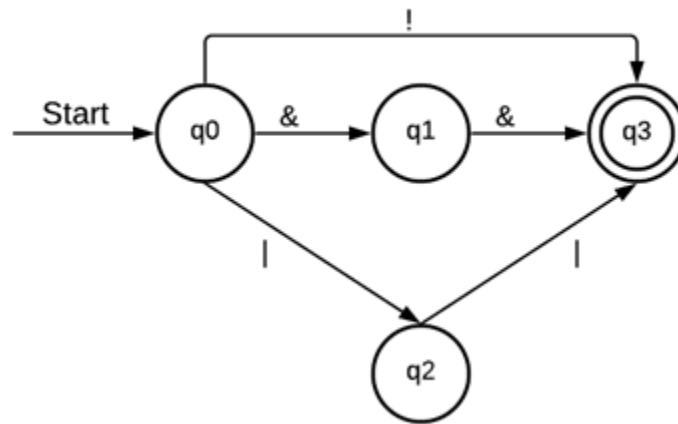
Operator	Description	Example
!	True if operand is false	not x
&&	True if both the operands are true	x and y
	True if either of the operands is true	x or y

The logical operators can be used to build complex logical expressions. The use of parentheses is necessary/recommended to nest or group expressions.

Example of mix logical and relational operators:

(6 > 4) and (2 <= 14)

Machine for Logical Operators:

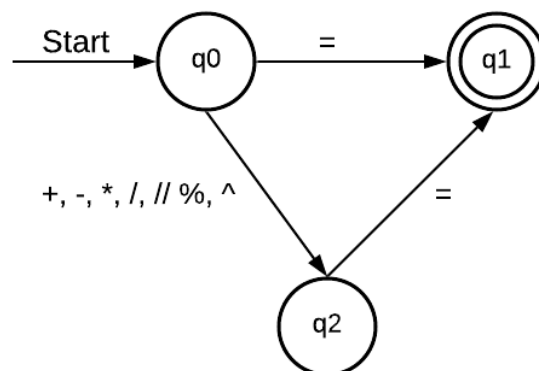


C. Assignment Operators [4]

Operator	Description	Example
=	Assign value of right side of expression to left side operand	x = 5
+=	Add and Assign: Add right side operand with left side operand and then assign to left operand	x += 5
-=	Add and Assign: Add right side operand with left side operand and then assign to left operand	x -= 5
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	x *= 5

/=	Divide AND: Divide left operand with right operand and then assign to left operand	x /= 5
%=	Modulus AND: Takes modulus using left and right operands and assign result to left operand	x %= 5
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	x //= 5
^=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	x ^= 5

Machine for Assignment Operators:



4. Keywords and Reserved Words [5][6]

Explicitly state your set of keywords and set of reserved words (data types) - with machine.

Reserved Words

Name	Description
as	Used to create an alias
assert	The assert keyword is used when debugging code. It tests if a condition in the code returns True, if not, the program will raise an AssertionError.
break	Terminates the loop containing it
bool, boolean	Data type
char, character	Data type
class	Used to create a class
continue	Skips the rest of the code inside a loop for the current iteration only.
def, define	keyword for defining a function
del, delete	used to delete objects, lists, dictionaries, variables, etc., in Python
otherwise	Used with 'whenever' structures. It catches anything which isn't caught by the preceding conditions.
except	used to catch and handle the exception(s) that are encountered in the try clause
finally	Executed after try and except blocks.
False	Boolean value

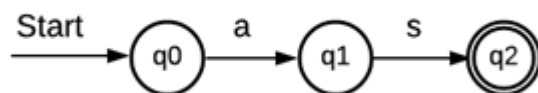
for	Used in loop structures
from	Used to import only a specific section from a module
global	Allows to modify the variable outside of the current scope
whenever	Begins "whenever" structures
import	Imports code from one module into another program
in	Used to check if a value is present in a sequence (list, range, string etc.). Also used to iterate through a sequence in a for loop
int, integer	Data type
is	Used to test if two variables refer to the same object.
lambda	used to create small anonymous functions
nonlocal	Used to make the variable which refers to the variable bounded in the nearest scope. Scope to which variable it bound should not be global or local scope
None	Used to define a null variable or an object
pass	used as a placeholder for future code, to execute nothing
raise	used to raise an exception

return	exits a function and instructs Python to continue executing the main program
True	Boolean value
try	Used in try... except blocks. It defines a block of code test if it contains any errors
with	Used for resource management and exception handling.
while	Used in while loop structures
yield	Returns value from a function without finishing the states of a local variable.

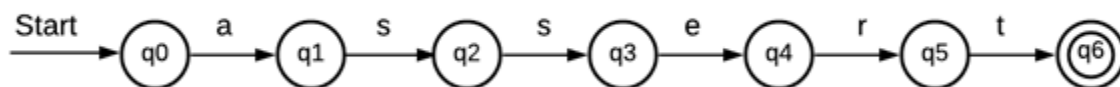
This language's reserved words are case-sensitive. These words are formulated to be portable and understandable; the length and simplicity of the words were considered.

To prevent errors, the programmer should be aware that these keywords are reserved and should not be used as variable names. If the programmer attempts to use any of the words above as identifiers in programs, an error will appear.

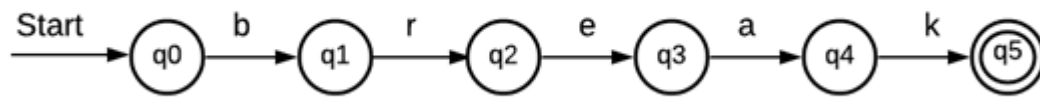
as



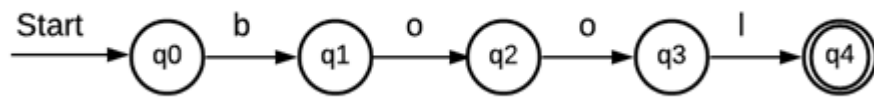
assert



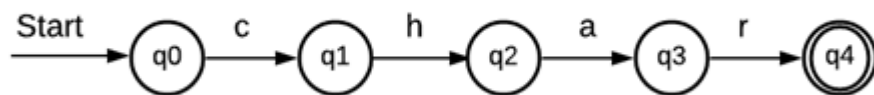
break



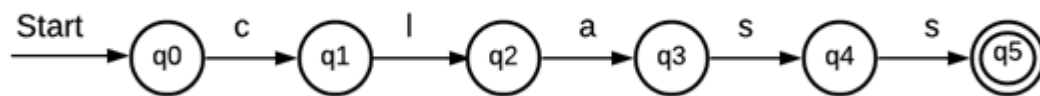
bool



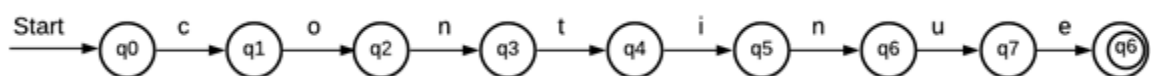
char



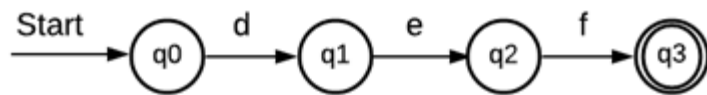
class



continue



def



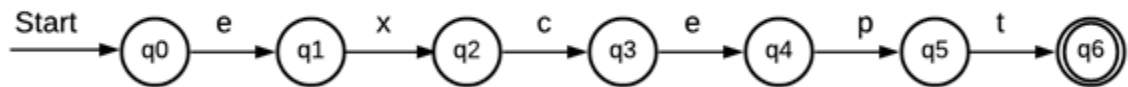
del



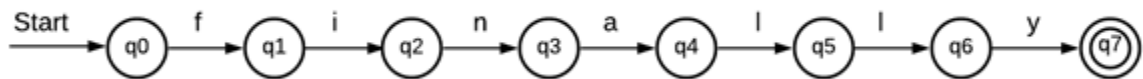
otherwise



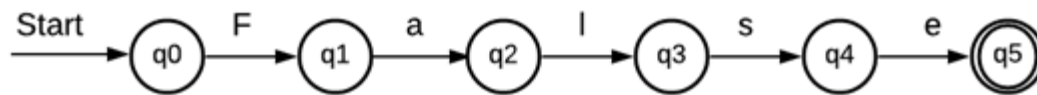
except



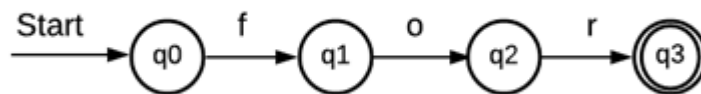
finally



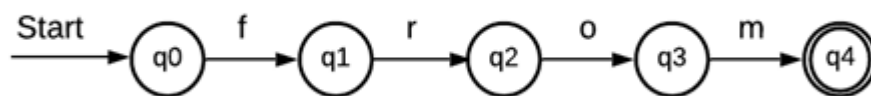
False



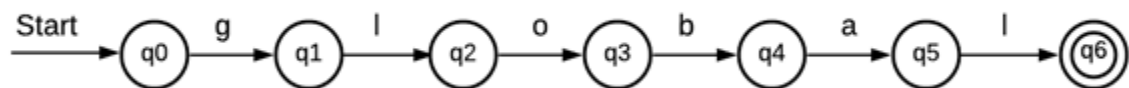
for



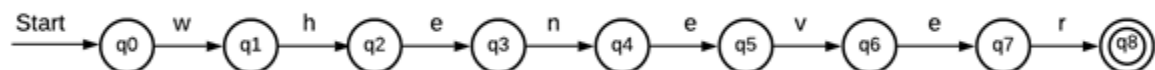
from



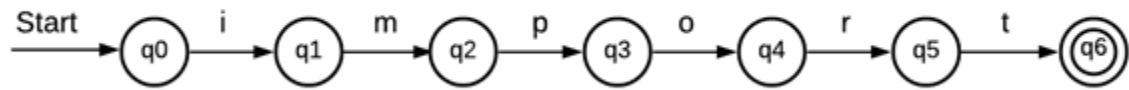
global



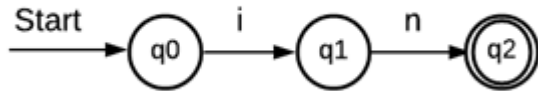
whenever



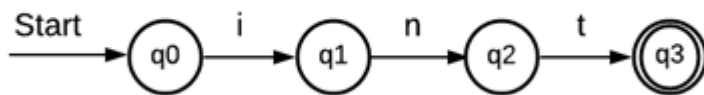
import



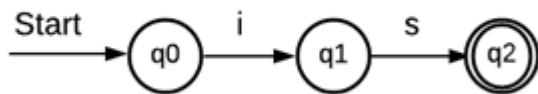
in



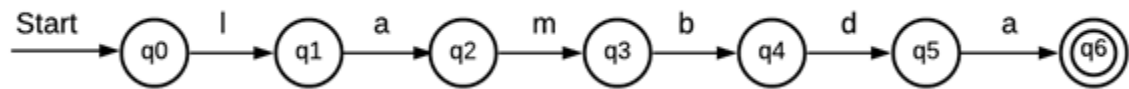
int



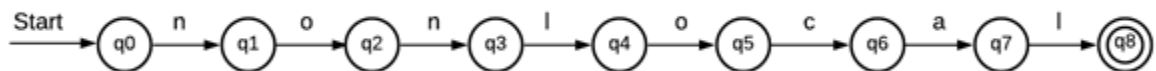
is



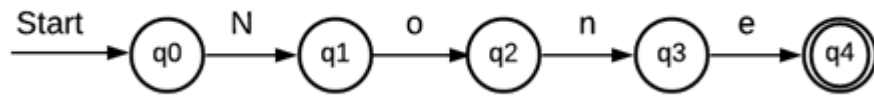
lambda



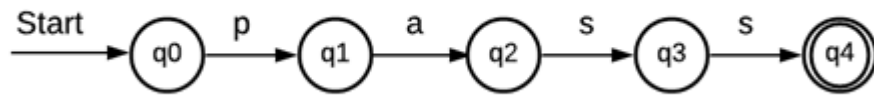
nonlocal



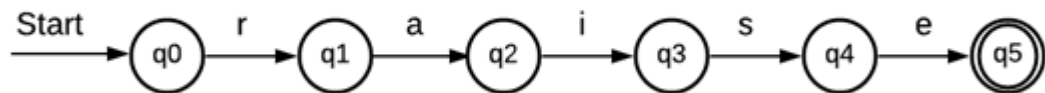
none



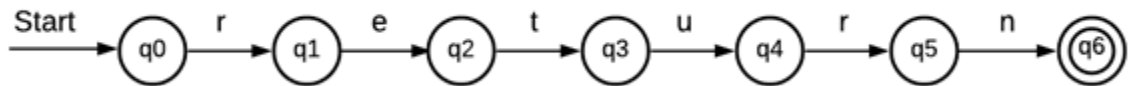
pass



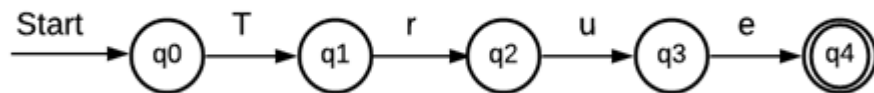
raise



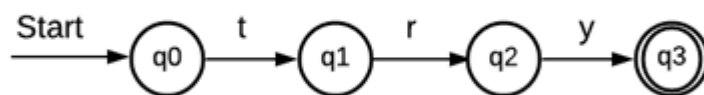
return



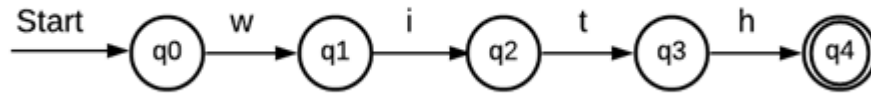
True



try



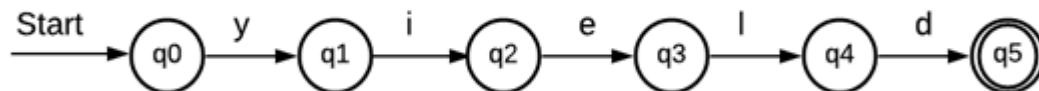
with



while



yield



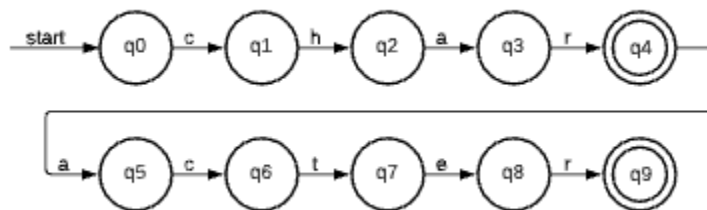
5. Noise words

Noise Words	Shorthand	Original Notation	Definition
acter	char	character	Full notation for the data type character, can be short handed as char by removing the noise word “acter”. For clearer reading and definition, typing it in full notation is recommended
eger	int	integer	Full notation for the data type integer, can be short handed as int by removing the noise word “eger”. For clearer reading and definition, typing it in

			full notation is recommended
ean	bool	boolean	Full notation for the data type boolean, can be short handed as bool by removing the noise word “ean”. For clearer reading and definition, typing it in full notation is recommended
def	ine	define	Full notation for the function define, can be short handed as def by removing the noise word “ine”. For clearer reading and definition, typing it in full notation is recommended
del	ete	delete	Full notation for the function delete, can be short handed as del by removing the noise word “ete”. For clearer reading and definition, typing it in full notation is recommended

Machine for Noise Words:

character (char + acter)



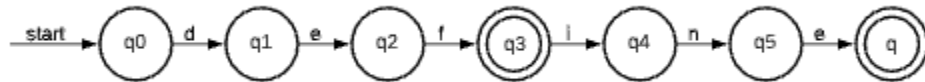
integer (int + eger)



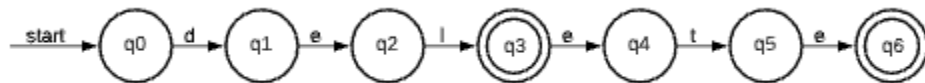
boolean (bool + ean)



define (def + ine)



delete (del + ete)



6. Comments

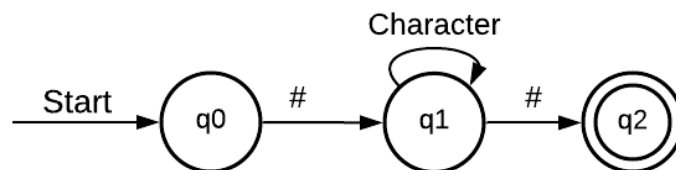
We have two types of comments here, the end-of-line/single line comment and the block comment. Comments can go before or after the program.

A comment must be preceded by a single number sign (#) that indicates that the following line is a comment and should be ignored. All these comments can start any place on the line and can be placed after commands. These comments are terminated automatically by encountering another number sign. The example below shows how comments are indicated

This is a comment # - single line

This is

Also a comment # - multiple line



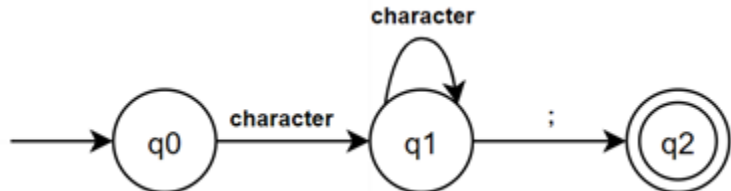
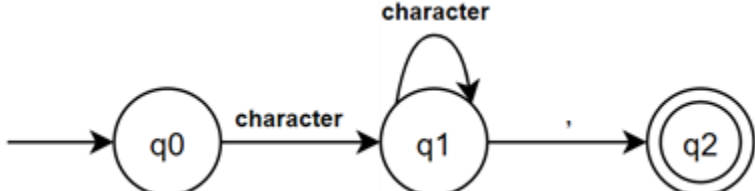
7. Blanks (spaces)

- White-space characters including space, tab, enter (newline) serve the same purpose as the spaces between words and lines on a printed page; they make reading easier. [1]
- White-space is not necessary for separating the identifiers, operators, delimiters, and values in this programming language.
- White-space is necessary after using a keyword or reserved word.
- White-space characters are ignored by the C compiler when parsing code unless they are used as separators or as components of character constants or string literals.
- Comments are also considered by the compiler as white space.

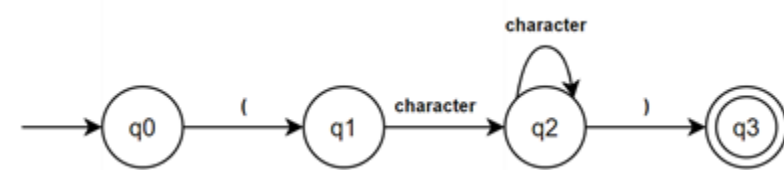
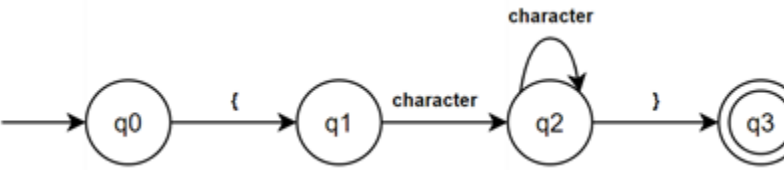
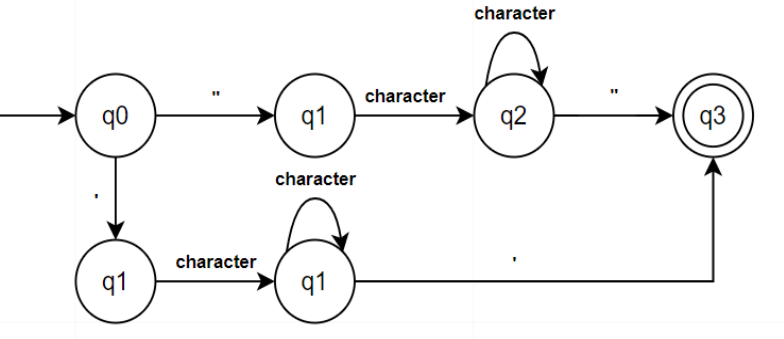
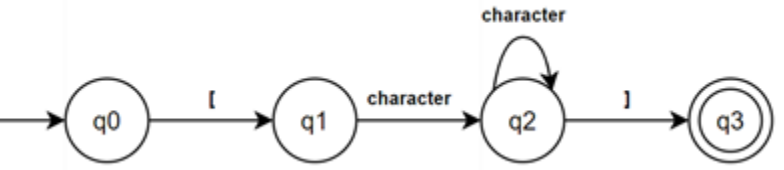
8. Delimiters and Brackets

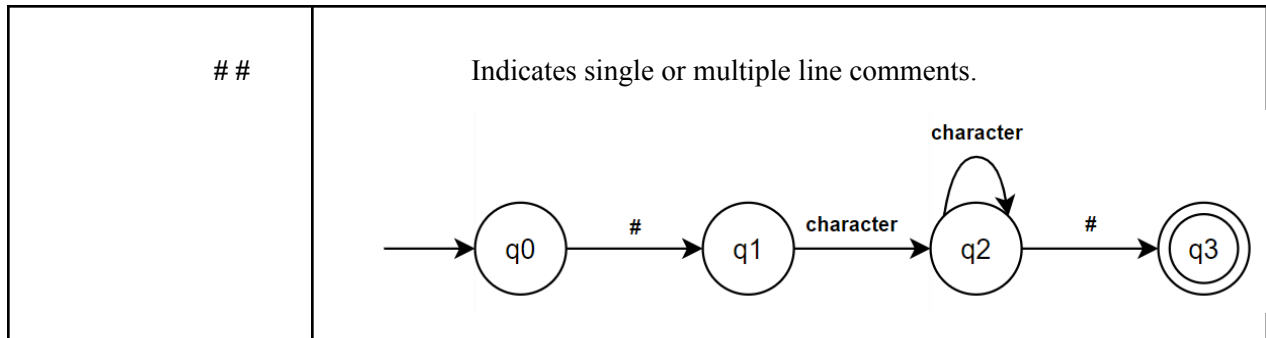
Delimiter is a single character or sequence of characters that marks both beginning and end of a statement, as well as sets boundaries between distinct units of data. It is used to organize and structure the code for it to be functional. [8]

Field and Record Delimiters [7]

Delimiter	Function
;	<p>Used to mark the end of a statement and serve as a separator between lines.</p> 
,	<p>Separates elements of a list and indicates separation between fields.</p> 

Bracket Delimiters [7]

Delimiter	Function
()	<p>Used to introduce parameters into a function or method, and to give a higher priority to an operation in an arithmetic expression.</p>  <pre> graph LR start(()) --> q0((q0)) q0 -- "(" --> q1((q1)) q1 -- "character" --> q2((q2)) q2 -- "character" --> q2 q2 -- ")" --> q3(((q3))) style start fill:none,stroke:none </pre>
{ }	<p>Used to construct a local scope by enclosing a set of statements.</p>  <pre> graph LR start(()) --> q0((q0)) q0 -- "{" --> q1((q1)) q1 -- "character" --> q2((q2)) q2 -- "character" --> q2 q2 -- "}" --> q3(((q3))) style start fill:none,stroke:none </pre>
‘ ’ or “ ”	<p>Indicates character or string literals.</p>  <pre> graph TD start(()) --> q0((q0)) q0 -- "\"" --> q1_1((q1)) q0 -- "'" --> q1_2((q1)) q1_1 -- "character" --> q2_1((q2)) q1_2 -- "character" --> q1_2 q2_1 -- "\"" --> q3(((q3))) q1_2 -- "character" --> q1_2 q1_2 -- "'" --> q3 style start fill:none,stroke:none </pre>
[]	<p>Indicates subscript.</p>  <pre> graph LR start(()) --> q0((q0)) q0 -- "[" --> q1((q1)) q1 -- "character" --> q2((q2)) q2 -- "character" --> q2 q2 -- "]" --> q3(((q3))) style start fill:none,stroke:none </pre>



9. Free-and-fixed-field formats

sPyC is both inspired from C and Python but we based the format from C language, which is a free-field format language. sPyC will also be a free-field format language since we program on interactive devices. Users can technically write multiple statements in the same line as long as they are syntactically and lexically correct.

10. Expression

a. Mathematical/Arithmetic Expressions [9][10]

Precedence Level	Operator	Description	Associability
High	() []	Parentheses/Brackets	Left-to-Right
	* / %	Multiplication/Division/Modulus	Left-to-Right
Low	+ -	Addition/Subtraction	Left-to-Right

b. Boolean expression [9][10]

Precedence Level	Operator	Description	Associability
High	<	less than,	Left-to-Right
	<=	less than or equal, greater than, greater than or equal	

Low	>		
	=>		
	==	equal, not equal	Left-to-Right
	!=		
	&&	AND	
		OR	

11. Statements

a. Declaration Statement

The declaration statement is almost similar to the declaration in existing languages. The only difference is that there is no white space between the data type and the variable name.

<data type><variable name>;

Example: intSalary;

b. Input/Output Statements

Input Statement

<identifier> = read();

Output Statement

Syntax	Example:	Output:
write(<identifier>);	strB = 'Chaeyoung'; write(strB);	Chaeyoung
write('<string>'); write("&<string>");	write("I got the feels for you.");	I got the feels for you.
write('<string>' + <identifier>);	strGG = "TWICE";	Hello, we are TWICE

write("<string>" + <identifier>);	write("Hello, we are" + strGG);	
--------------------------------------	---------------------------------	--

Example:

```
intAge = read('Age is: ');
```

```
write(intAge);
```

Output:

Age is: 20

20

c. Assignment Statement (expressions)

Assignment_Operators = {=, +=, *=, /=, %=, //, ^=}

Syntax	Definition	Example:
<identifier> <Assignment_Operators> <value>;	Assignment by value - the value is assigned to the identifier.	intStayc = 6;
<identifier> <Assignment_Operators> <identifier>;	Assignment by variable - assigns the right identifier's value to the left identifier	doubleRawr += doubleMika;
<identifier> <Assignment_Operators> <expression>;	Assignment by expression - the expression's value is assigned to an identifier	intAnswer = 150 - 7;

d. Conditional Statement (Nested)

Instead of the terms "if", "else" and "else if", we'll use the synonyms "whenever", "otherwise" and "otherwise whenever" in the conditional statement.

i. **if** (whenever)

If you only need to execute one statement, put it on the same line.

_____ **whenever** (expression) #Executes when the expression is true

Example:

```
intT = 8;
intW = 10;
whenever (intT < intW) write("T is less than W");
```

Output:

T is less than W

If you need to execute multiple statements, put it inside curly braces ({ }).

```
whenever (expression)
    { statements } #Executes when the expression is true
```

Example:

```
intTs = 22;
whenever (intTs == 22)
{
    write("Ts is equivalent to");
    write("22");
}
```

Output:

Ts is equivalent to 22

ii. **if else** (whenever ; otherwise)

If you only need to execute one statement, one for each, you can put everything on the same line.

<execute when the expression is true> **whenever** (expression) **otherwise** <execute when expression is false>;

Example:

```
intT = 8;
intW = 10;
write("It is true.") whenever (intT < intW) otherwise write("It is false.");
```

Output:

It is true.

If you need to execute multiple statements, for both, you can put it inside curly braces ({ }).

whenever (expression)

```
{ statements } #Executes when the expression is true
```

otherwise

```
{ statements } #Executes when the expression is false
```

Example:

```
intT = 88;
intW = 10;
whenever (intT < intW)
    { write("It is true. ");
      write("T is less than W."); }
otherwise
    { write("It is false. ");
      write("T is not less than W."); }
```

Output:

It is false. T is not less than W.

iii. if-else if-else

If you only need to execute one statement, one for each, you can put everything on the same line per condition.

whenever (expression) #Executes when the first expression is true

otherwise whenever (expression) #Executes when the 2nd expression is true

otherwise #Executes when the 1st and 2nd expressions are false

If you need to execute multiple statements, you can put it inside curly braces ({ }).

whenever (expression)

{ statements } #Executes when the first expression is true

otherwise whenever (expression)

{ statements } #Executes when the 2nd expression is true

otherwise

{ statements } #Executes when the 1st and 2nd expressions are false

Example:

```
intT = 24;
```

```
whenever (intT == 8) write("T is 8");
```

```
otherwise whenever (intT == 24) write("T is 24");
```

```
otherwise write("T is not 8 and 24");
```

Output:

T is 24

iv. **nested if**

whenever (expression1)

whenever (expression2)

whenever (expression3)

{ statement(s) } #Executes when the 3rd expression is true

otherwise

{ statement(s) } #Executes when the 3rd expression is false

otherwise

{ statement(s) } #Executes when the 2nd expression is false

otherwise

{ statement(s) } #Executes when the 1st expression is false

Example:

intX = read("Input a number: ");

whenever (intX <= 55)

whenever (intX < 25)

whenever (intX < 10)

{ write("The number is less than 10."); }

otherwise

{ write("The number is not less than 10."); }

otherwise

{ write("The number is not less than 25."); }

otherwise

```
{ write("The number is not less than or equal to 55."); }
```

Input a number: 99

Output: The number is not less than or equal to 55.

Input a number: 5

Output: The number is less than 10.

v. **nested if else**

whenever (expression1)

Execute inner whenever-otherwise when the first expression is true

whenever (expression2)

Execute when the second expression is true

otherwise

Execute when the second expression is false

otherwise

Execute when the first expression is false

Example:

```
intX = read("Input a number: ");
```

```
whenever (intX <= 99)
```

```
    whenever (intX < 99)
```

```
        write("The number is less than 99.");
```

```
    otherwise
```

```
        write("The number is equal to 99.");
```

```
otherwise
```

```
write("The number is not less than and equal to 99.");
```

Input a number: 99

Output: The number is equal to 99.

Input a number: 101

Output: The number is not less than and equal to 99.

vi. **nested if else if**

```
whenever (expression1)
```

```
    # Execute when expression1 is true
```

```
    whenever (expression3)
```

```
        # Execute when expression3 is true
```

```
    otherwise whenever (expression4)
```

```
        # Execute when expression3 is false AND expression4 is true
```

```
    otherwise
```

```
        # Execute when expression3 is false AND expression4 is false
```

```
otherwise whenever (expression2)
```

```
    # Execute when expression1 is false AND expression2 is true
```

```
    whenever (expression5)
```

```
        # Execute when expression5 is true
```

```
    otherwise whenever (expression6)
```

```
        # Execute when expression5 is false AND expression6 is true
```

```
    otherwise
```

```
        # Execute when expression5 is false AND expression6 is false
```


otherwise

Execute when expression1 is false AND expression2 is false

whenever (expression7)

Execute when expression7 is true

otherwise whenever (expression8)

Execute when expression7 is false AND expression8 is true

otherwise

Execute when expression7 is false AND expression8 is false

Example:

intX = read("Input a number: ");

whenever (intX < 10)

whenever (intX == 3)

write("The number is 3.");

otherwise whenever (intX == 7)

write("The number is 7.");

otherwise

write("The number is not 3 and 7.");

otherwise whenever (intX > 20)

whenever (intX == 25)

write("The number is 25.");

otherwise whenever (intX == 50)

write("The number is 50.");

otherwise

write("The number is not 25 and 50.");

otherwise

```

whenever (intX == 10)

    write("The number is 10.");

otherwise whenever (intX == 15)

    write("The number is 15.");

otherwise

    write("The number is not 10 or 15.");

```

Input a number: 25

Output: The number is 25.

Input a number: 6

Output: The number is not 3 and 7.

Input a number: 10

Output: The number is 10.

e. Iterative Statements

i. for

for loop with statements

Sequence = {list, tuple, array}

for <variable> within <sequence>

{ statements }

Example:

```
arrayBpink = ["Jisoo", "Jennie", "Rose", "Lisa"];
```

```
for intX within arrayBpink
```

```
{ write(intX + "is a member."); }
```

Output:

Jisoo is a member.

Jennie is a member.

Rose is a member.

Lisa is a member.

for loop with iterators

for <variable> within <iterator>

{ statements }

Example:

for intX within scope(4)

write("The number is: " + intX);

Output:

The number is 1

The number is 2

The number is 3

The number is 4

ii. nested for

nested for loop with statements

for <variable> within <sequence>

for <variable> within <sequence>

{ statements }

Example:

```
arrayColor = ["green", "yellow"];
arrayFruits = ["banana", "mango"];
for intX within arrayColor
{
    for intY within arrayFruits
    {
        write(intX, intY);
    }
}
```

Output:

```
green banana
green mango
yellow banana
yellow mango
```

nested for loop with iterators

```
for <variable> within <iterator>
    for <variable> within <iterator>
        { statements }
```

Example:

```
for intX within scope(2)
{
    for intY within scope(2)
    {
```

```
        write(intX, intY);  
    }  
}
```

Output:

0 0

0 1

1 0

1 1

Citations:

- [1] <https://docs.microsoft.com/en-us/cpp/c-language/white-space-characters?view=msvc-160>
- [2] <https://www.ics.uci.edu/~pattis/ICS-31/lectures/opexp.pdf>
- [3] https://www.tutorialspoint.com/python/python_basic_operators.htm
- [4] https://isip.piconepress.com/courses/temple/ece_3822/resources/tutorials/python/python_basic_operators.pdf
- [5] <https://realpython.com/lessons/reserved-keywords/>
- [6] https://www.w3schools.com/python/python_ref_keywords.asp
- [7] <https://en.wikipedia.org/wiki/Delimiter>
- [8] <https://www.computerhope.com/jargon/d/delimit.htm>
- [9] <https://www.tutorialcup.com/cprogramming/operator-precedence-associativity.htm>
- [10] <https://www.youtube.com/watch?v=5JXcX0IqRUo&t=134s>