Square

Mini-Programming Language Project Documentation

COSC 303

Bonifacio, Regina S.
Carlos, Henry James R.
Domingo, Syruz Ken C.
Lagman, Fervicmar D.
Marzan, Duane Kyros DR.
Ong, Ylana Lorelie I.
Queja, Hans Christian N.
Rosales, Acelle Krislette L.

November 2024

Introduction

Square is a programming language designed with beginners in mind, particularly targeting high school students, computer science and IT students especially freshmen, and career shifters who are new to coding. Its syntax is built around square brackets, which is where its name is derived from, maintaining simplicity and clarity. This general programming language emphasizes ease of learning and readability, making it an ideal starting point for those who are just beginning their programming journey. With a balance of powerful features and an intuitive structure, Square allows newbies to focus on understanding core programming concepts without feeling overwhelmed by complex syntax.

The programming language draws inspiration from C for its robust static typing and fast compile time, which ensure efficient performance and reduced runtime errors. By incorporating type hints, annotations in a programming language that specify the expected data types of variables, function parameters, and return values, the language facilitates clarity in defining functions and variables, making the code easier to read and debug.

From Python, the language adopts a focus on simplicity and readability through the use of straightforward keywords and syntax to lower the barrier to entry for users. The inclusion of an f-string-inspired concept allows for clear string interpolation, making it easier to present calculations and results in an accessible format. This combination of C's performance-oriented features and Python's user-friendly principles makes the language well-suited for users new to programming with both accuracy and ease of use.

One of the problems that Square wants to solve is the repetitive typing of variable and condition value for conditional OR statements. For instance,

```
if x == 12 or x == 13 or x == 0
    print("Valid value")
```

In this programming language, the concept of junctions will be used. The syntax is as follows:

```
when x [
    == 12 | 13 | 0 :: print("Valid value")
]
```

With this approach, there is no need for the programmer to repeatedly type the variable `x` and the condition values `== 12`, `== 13`, and `== 0`. In just one run, the junction allows these values to just be separated by the vertical bar (|).

Another issue that Square aims to address is the repetitive accessing of fields in a structure. For instance, Square has a similar collection of variables with `struct` in C. It is called models (`mod`) and has the following syntax:

```
mod <identifier> [
    <identifier1>: <data_type> = <value1>.
    <identifier2>: <data_type> = <value2>.
    <identifier3>: <data_type> = <value3>.
    ...
    <identifiern>: <data_type> = <valuen>.
]
```

Normally, such as what is conventional in other programming languages, programmers need to access the elements or fields individually, using a dot operator. For instance,

```
user.name = "Juan Dela Cruz".
user.age = 20.
```

In Square, batch accessing is introduced. With this, there is no need to access elements or fields individually. Instead, the programmer can just do this:

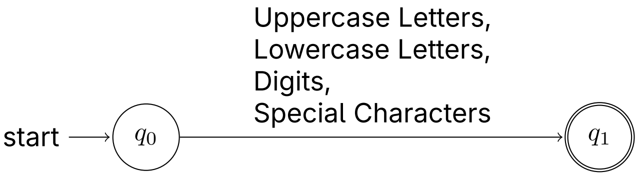
```
user.[name, age] = ["Mary Dela Cruz", 22].
```

Syntactic Elements of Language

I. Character Set

Characters = { Alphabet, Digits, Special Characters }

- **Alphabet** = { Uppercase Letters, Lowercase Letters }
  - **Uppercase Letters** = { 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 }
  - **Lowercase Letters** = { 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** = { ., +, -, *, /, %, <, >, =, ", ’, „ ;, |, !, (,), [,], _, ^, ~, & }



II. Identifiers

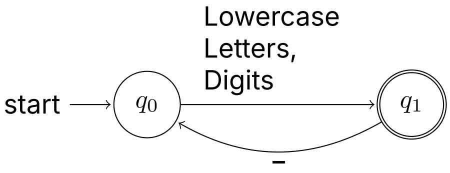
1. Identifiers can contain lowercase letters, digits, and underscores (_).
2. Identifiers must always start with a lowercase letter.
3. Identifiers must not end with an underscore (_).
4. Identifiers must only use lowercase letters.
5. Identifiers cannot be reserved words (e.g., `if`, `for`, `while`, etc.).
6. Identifiers must not contain spaces or tabs.

Table 1: Examples of Valid and Invalid Identifiers

Identifier	Validity	Reason (if invalid)
`gender`	Valid	-
`temp_value`	Valid	-
`numbers123`	Valid	-
`x`	Valid	-
`abc_xyz`	Valid	-
`1academy`	Invalid	Starts with a digit, not a lowercase letter
`bestBuy`	Invalid	Contains an uppercase letter
`batch@09`	Invalid	Contains unsupported special characters like `@`

Regular Expression: (`<Lowercase Letters>` | `<Digits>` | _)* (`<Lowercase Letters>` | `<Digits>`)

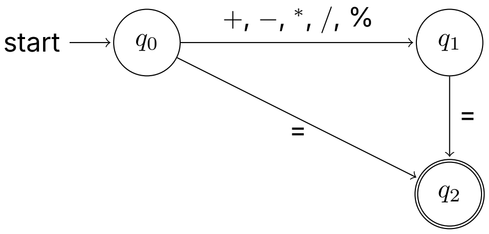
State Diagram:



III. Operation Symbols

A. Assignment Operator

Assignment Operator



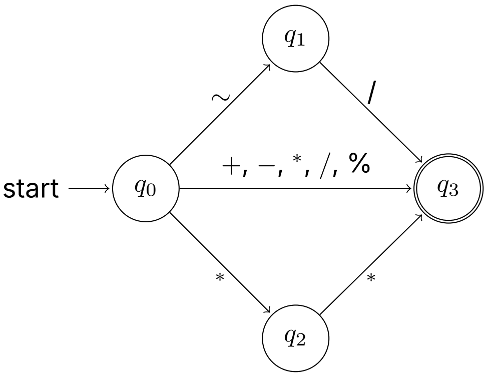
<assignment_operator> ::= = | += | -= | *= | /= | %=

Table 2: Assignment Operators and Descriptions

Operator	Example Expression	Description
=	`score: int = 57`	Assignment Operator: Assigns a variable/value to the variable. In the example, it assigns 57 to integer `score`.
+=	`age: += 2`	Addition Assignment: Adds a variable/value to the current value of the variable and updates its value. In the example, it adds 2 to the value of integer `age` and assigns the sum to `age`.
-=	`energy: int -= 25`	Subtraction Assignment: Subtracts a variable/value from the current value of the variable and updates its value. In the example, it subtracts 2 from the value of integer `energy` and assigns the difference to `energy`.
*=	`price: int *= 2`	Multiplication Assignment: Multiplies the current value of the variable to a variable/value and updates its value. In the example, it multiplies 2 to the value of integer `price` and assigns the product to `price`.
/=	`amount: int /= 2`	Division Assignment: Divides the current value of the variable by a variable/value and updates its value. In the example, it divides the integer `amount` by 2 and assigns the quotient to `amount`.
%=	`items: int %= 5`	Modulus Assignment: Divides the current value of the variable by a variable/value and updates its value with the remainder. In the example, it divides the integer `items` by 2 and assigns the remainder to `items`.

B. Arithmetic Operator

Arithmetic Operator



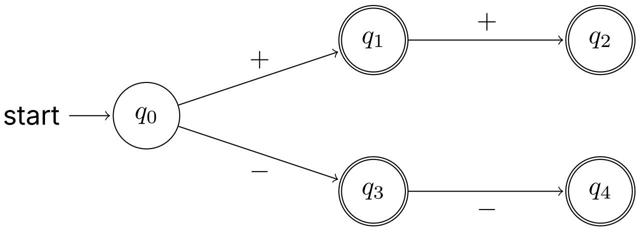
`<arithmetic_operator>` ::= + | - | * | / | % | ~/ | **

Table 3: Arithmetic Operators and Descriptions

Operator	Example Expression	Description
+	`principal + interest`	Addition: Adds the values of two variables together. In the example, it produces the sum of `principal` and `interest`.
-	`budget - expenses`	Subtraction: Subtracts the value of one variable from another. In the example, it returns the difference between `budget` and `expenses`.
*	`hours_worked * hourly_rate`	Multiplication: Multiplies the values of two variables together. In the example, it produces the product of `hours_worked` and `hourly_rate`.
/	`distance / time`	Division: Divides the value of one variable by another. In the example, it returns the quotient of `distance` divided by `time`.
%	`total_items % items_per_box`	Modulo: Finds the remainder when one variable is divided by another. In the example, it returns the remainder when `total_items` was divided by `items_per_box`.
~/	`total_pages ~/ pages_per_day`	Integer Division: Divides one number by another and returns the largest integer less than or equal to the result. In the example, it divided `total_pages` by `pages_per_day` and returns the integer part of the result.
**	`hypotenuse ** 2`	Exponent: Raises a number to the power of another. In the example, it raises `hypotenuse` to the power of 2.

C. Unary Operator

Unary Operator



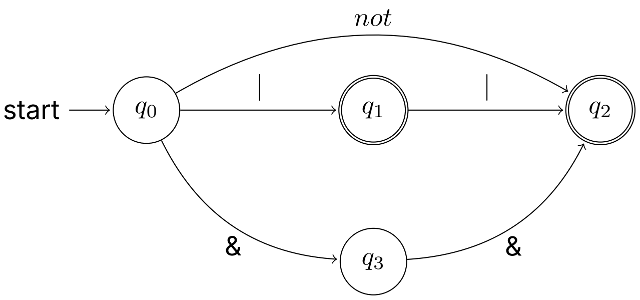
<unary_operator> ::= + | - | ++ | -

Table 4: Unary Operators and Descriptions

Operator	Example Expression	Description
+	+index	Unary Plus: Indicates that the value of a variable is positive. In the example, index is positive.
-	-index	Unary Minus: Negates the value of a variable, indicating a negative value. In the example, index is negative.
++	index++	Increment: Increases the value of the variable by 1. In the example, increases the value of index by 1.
--	index--	Decrement: Decreases the value of the variable by 1. In the example, it decreases the value of index by 1.

D. Logical Operator

Logical Boolean Operator



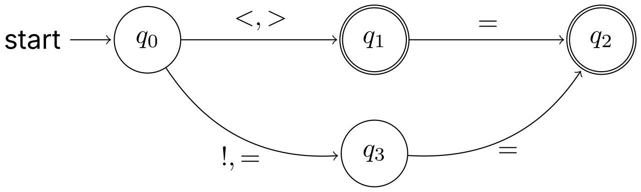
<logical_operator> ::= | | || | && | not

Table 5: Boolean Operators and Descriptions

<table>
<tr><th>Operator</th><th>Example Expression</th><th>Description</th></tr>
<tr><td>|</td><td>when a == 12 | 13 | 0
:: print("Valid value").</td><td>Junction OR: Checks if a variable matches any of the specified values. If the variable matches any value, the condition evaluates to true.</td></tr>
<tr><td>||</td><td>a: bool = true.
b: bool = false.
c: bool = a || b.</td><td>Logical OR: Returns a value true if either of the statements in the condition is true, otherwise, returns false. Therefore c is false.</td></tr>
<tr><td>&&</td><td>a: bool = true.
b: bool = false.
c: bool = a && b.</td><td>Logical AND: Returns a value true if both statements in the condition are true, otherwise, returns false. Therefore, c is false.</td></tr>
<tr><td>not</td><td>a: bool = true.
b: bool = not a.</td><td>Logical NOT: Returns the opposite value of the expression. Therefore, b is false.</td></tr>
</table>

E. Relational Operator

Relational Operator



`<relational_operator>` ::= < | > | <= | >= | != | == |

Table 6: Relational Operators and Descriptions

Operator	Example Expression	Description
==	`x == y`	Equal: Checks if the values of the variables are equal. In the example, it checks if `x` and `y` are equal.
!=	`age != required_age`	Not Equal: Checks if the values of the variables are not equal. In the example, it evaluates if `age` and `required_age` are not equal.
>	`temperature > threshold`	Greater Than: Checks if the value of a variable is greater than the other. In the example, it checks if `temperature` is greater than `threshold`.
<	`speed < speed_limit`	Less Than: Checks if the value of the variable is less than the other. In the example, it checks if `speed` is less than speed_limit.
>=	`score >= passing_score`	Greater Than or Equal To: Checks if the value of the variable is greater than or equal to the other. In the example, it evaluates if `score` is greater than or equal to `passing_score`.
<=	`height <= max_height`	Less Than or Equal To: Checks if the value of the variable is less than or equal to the other. In the example, it checks if `height` is less than or equal to `max_height`

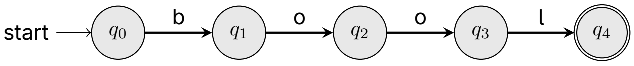
IV. Keywords and Reserved Words

A. Keywords

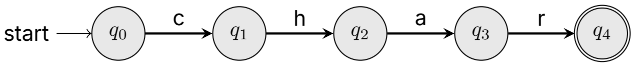
Table 7: Square Language Keywords

Keyword	Definition
`bool`	Indicates a data type used to represent true or false values.
`char`	Represents the character data type used to store a single character.
`const`	Declares a variable whose value cannot be changed after it is initialized.
`double`	Defines a double-precision floating-point number, allowing for greater accuracy than float (typically 8 bytes).
`exit`	Exits the current loop, ending the iteration.
`float`	Represents floating-point numbers, used for real numbers with decimal points or those that require exponential notation.
`get`	Used to collect input from the user and store it in a variable. It allows the program to receive data, such as numbers or text, for further use.
`int`	Declares integer variables, used for whole numbers (size is platform-dependent but typically 4 bytes).
`long`	Declares a long integer variable.
`print`	Used to show output to the user. It displays the value of a variable, expression, or a message on the screen, letting the program provide information or results.
`return`	Exits a function and optionally returns a value.
`short`	Declares a smaller integer type (typically 2 bytes), used for saving memory in constrained environments.
`signed`	Explicitly declares that a variable can hold both positive and negative values.
`skip`	Skips the rest of the current loop iteration and immediately starts the next iteration.
`static`	Keeps a variable or function's value persistent across function calls or limits scope to the current file when applied to global declarations.
`unsigned`	Declares a variable that can only store non-negative values, effectively doubling the positive range.
`void`	Specifies that a function does not return a value or a pointer has no type.

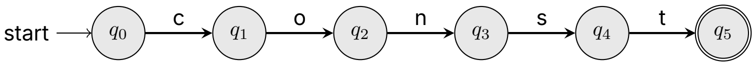
bool



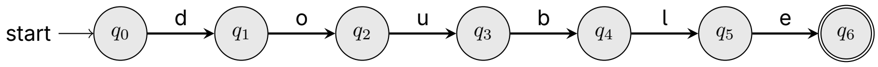
char



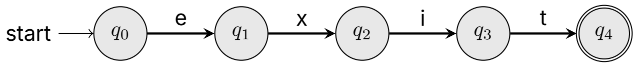
const



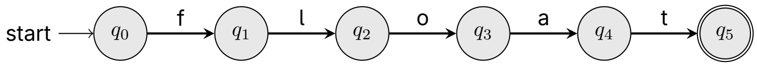
double



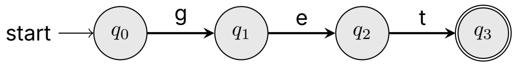
exit



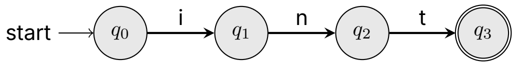
float



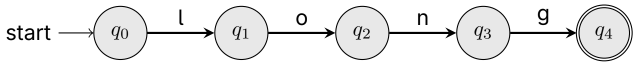
get



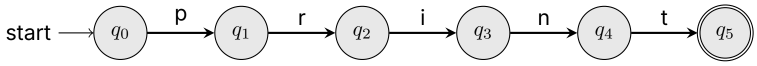
int



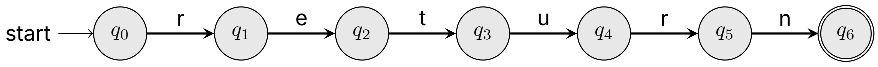
long



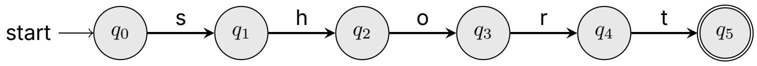
print



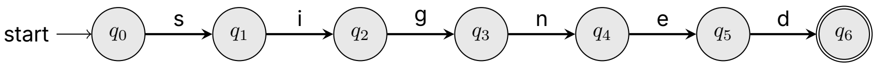
return



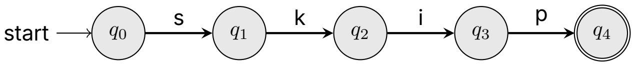
short



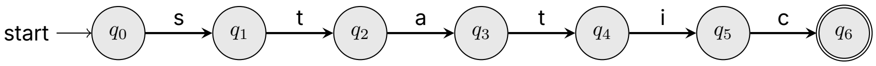
signed



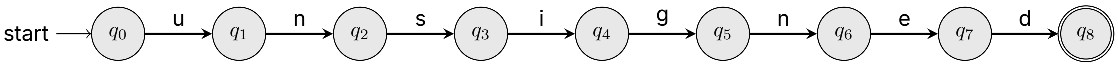
skip



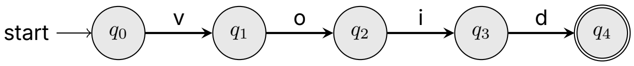
static



unsigned



void

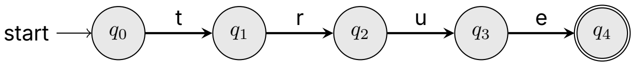


B. Reserved Words

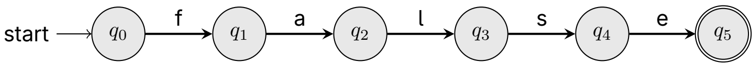
Table 8: Square Language Reserved Words

Reserved Word	Definition
`true`	Represents the boolean value "true," indicating correctness or activation.
`false`	Represents the boolean value "false," indicating incorrectness or deactivation.
`main`	The entry point function where program execution begins.

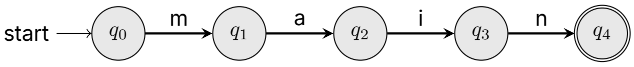
true



false



main

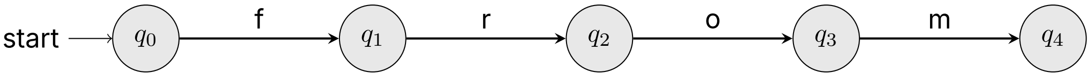


V. Noise Words

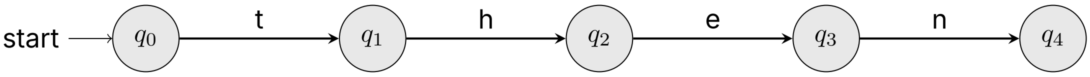
Table 9: Square Language Noise Words

Noise Word	Definition
`from`	It is typically used to indicate an origin point or a starting value in a given possible domain.
`then`	It is used to improve the flow in conditional statements. It is mainly used after a given condition.
`to`	It is used to indicate a transition or range in a given set of values.

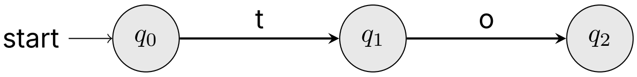
from



then



to

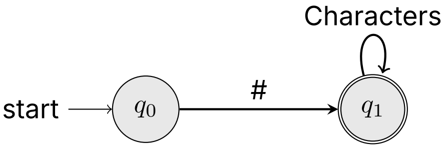


VI. Comments

Single-line Comments

Single-line comments make use of a singular hash (#) symbol to indicate a comment.

State Diagram



Example

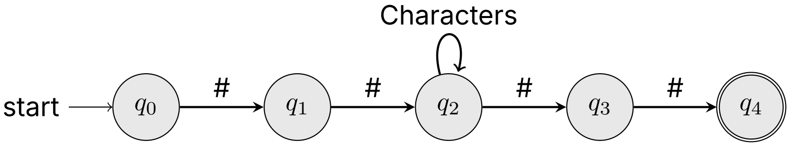
```
# This is a single-line comment in Square

input: int = 5.

```

```
# This is the second single-line comment.
```

Multiple-line Comments

Multiple-line comments make use of two sets of two hashes (#) encapsulating a given code snippet.

State Diagram



Example

```
##
This is a multiple-line comment!
I love multiple line comments, they allow me
to write tons of things.
##
```

VII. Blanks (Spaces)

Blanks are whitespaces primarily used to separate syntax such as identifiers, values, keywords, and more. The following must be adhered to:

- After using a keyword or a reserved word, a space must be used.

  Example

  ```
  coins: int = 5.
  ```

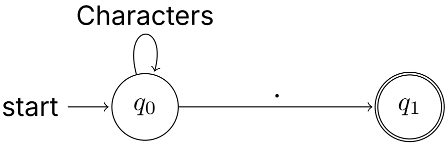
- If the programmer wants to code multiple statements in one line, they have to use proper delimiters and brackets.
- Code blocks for iterative and conditional statements can also be written in a single line, provided that they use proper delimiters and brackets.

VIII. Delimiters and Brackets

Dot (.)

The dot (`.`) is used as a statement terminator to indicate the end of an instruction or command in the code.

State Diagram



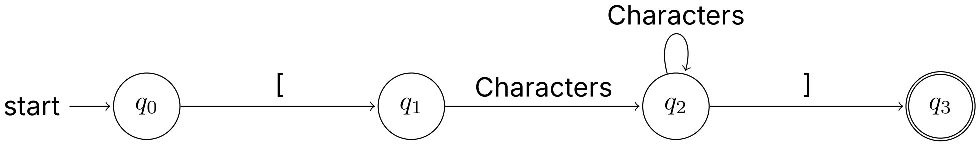
Example

```
population: int.
```

Square Brackets ([ ])

Square brackets ([ ]) serve as delimiters for defining code blocks. They encapsulate the code that belongs to constructs such as functions, loops, or conditional statements.

State Diagram



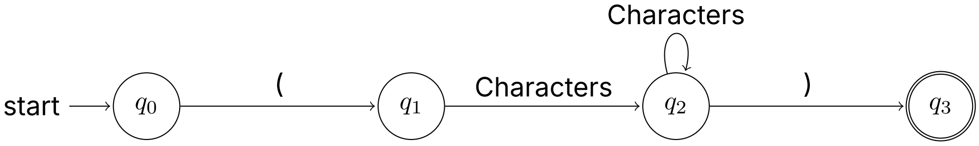
Example

```
def add(num1: int, num2: int) [
    return num1 + num2.
]
```

Parentheses (( ))

Parentheses (( )) enclose expressions, arguments for functions, or control conditions in statements to ensure logical grouping and clarity in operations.

State Diagram



Example

```
when true [
    return.
]
```

IX. Free-and-fixed-field formats

Square follows the **free-field format** as this allows for more freedom and flexibility in writing codes. It does not require specific positioning or alignment in a line for any statement as developers can position their code freely, as long as they follow the basic structures and rules of the language (i.e., syntax and logic).

X. Expression (Rules of Evaluating Expressions)

Mathematical/Arithmetic Expressions

Unary Expression

Table 10: Unary Expression Operators

Precedence	Operator	Description	Associativity
1	++, --	Unary operators for increment and decrement (prefix)	Right to left
2	++, --	Unary operators for increment and decrement (postfix)	Left to right

Arithmetic Expressions

Table 11: Arithmetic Expression Operators

Precedence	Operator	Description	Associativity
1	()	Parentheses used for grouping expressions	Left to right
2	**	Exponentiation operator for raising numbers to a power	Left to right
3	*, /, %, ~/	Multiplication, division, modulus, and integer division operators, respectively	Left to right
4	+, -	Addition and subtraction operators, respectively	Left to right

Ternary Expression

Table 12: Ternary Expression Operators

Precedence	Operator	Description	Associativity
1	?, :	Ternary conditional operator used for conditional expressions	Right to left

Boolean Expressions

Logical Expressions

Table 13: Logical Operators

Precedence	Operator	Description	Associativity
1	not	Logical NOT	Right to left
2	&&	Logical AND	Left to right
3	|, ||	Junction OR and Logical OR, respectively	Left to right

Relational Expressions

Table 14: Relational Operators

Precedence	Operator	Description	Associativity
1	>, <, >=, <=	Comparative operators (greater than, less than, greater than or equal to, less than or equal to)	Left to right
2	==, !=	Equality operators (equal, not equal)	Left to right

Operator Precedence Levels

Table 15: Precedence of All Operators

Precedence	Operators
1	Unary Operators
2	Arithmetic Operators
3	Relational Operators
4	Logical Operators

XI. Statements

A. Declaration Statement

Table 16: Declaration Statement Syntax and Examples

Syntax	Example
where: `<value> := <identifier> | <constant>`	
`<identifier>:<data_type>.`	`value1:  int.`
`<identifier>:<data_type> = <value>.`	`value1:  int = 5.`
`<identifier>:<data_type> = <value>.`	`value2:  float = value3.`
`<identifier>, <identifier>, ..., <identifier>:<data_type>.`	`value1, value2, _value3:  int.`
`<identifier1>, <identifier2>, ..., <identifier_n>: <data_type> = <value1>, <value2>, ..., <valueN>.`	`value1, value2, value3:  int = 5, 4, 6.`

`mod<identifier> [`
`  <identifier1>:  <data_type> = <value1>.`
`  <identifier2>:  <data_type> = <value2>.`
`  <identifier3>:  <data_type> = <value3>.`
`  ...`
`  <identifiern>:  <data_type> = <valuen>.`
`]`
	`mod book [`
	`  title:  string = "When Haru Was Here".`
	`  author:  string = "Dustin Thao".`
	`  published_year:  int = 2024.`
	`  genre:  string = "Fiction".`
	`  is_available:  bool = true.`
	`  pages:  int = 304.`
	`  rating:  float = 3.8.`
	`]`

B. Input/Output Statements

Input Statement

Table 17: Input Statement Syntax and Examples

Syntax	**Example**
<identifier> = get().	temp: double.
<identifier> = get("<string>").	temp: get("Enter temperature (Celsius): ").
print("<string>", <identifier>);	print("The temperature is", temp).

Output Statement

Table 18: Output Statement Syntax and Examples

Syntax	**Example**
print(<identifier>);	car_acceleration: double = 8.10.
print(<identifier>);	print(car_acceleration).
print(f"<string> identifier");	print(f"Acceleration of the car: {car_acceleration}").
print(f"<string> {identifier} <string> {identifier} ...");	print("Acceleration of the car: {car_acceleration} m/s^2").

C. Assignment Statement (Expressions)

Assignment Statement

Table 19: Assignment Statement Syntax and Example

Syntax	**Example**
where <identifier>: <data_type> <assignment_operator> <value>.	
<identifier>: <data_type> <assignment_operator> <value>.	var: int = 100.

Note: Type Annotation using colon (:) is implemented to explicitly specify the data type of variables, improving the code's readability.

D. Conditional Statement

If Statement

Syntax

1. Single-line statement

```
when var: type [
    condition :: statement.
]
```

2. Multi-line statements

```
when var: type [
    condition :: [
        statement.
        statement.
        statement.
    ]
]
```

Example Usage

1. Single-line statement

```
is_raining: bool = True

when is_raining: bool [
    true :: print("Don’t forget your umbrella!").
]
```

2. Multi-line statements

```
is_raining: bool = True

when is_raining: bool [
    true :: [
        print("It’s raining outside.").
        print("Take an umbrella before heading out.").
        notify("Weather alert: Rainy day!").
    ]
]
```

Example Output

1. Single-line statement

```
Don’t forget your umbrella!
```

2. Multi-line statements

```
It’s raining outside.
Take an umbrella before heading out.
Weather alert: Rainy day!
```

If-else Statement

Syntax

1. Single-line statement

```
when var: type [
    condition :: statement.
    _ :: statement.
]
```

2. Multi-line statements

```
when var: type [
    condition :: [
        statement.
        statement.
        statement.
    ]
    _ :: [
        statement.
        statement.
        statement.
    ]
]
```

Example Usage

1. Single-line statement

```
age: int = 19

when age: int [
    >= 18 :: print("You are eligible to vote.").
    _ :: print("You are not eligible to vote.").
]
```

2. Multi-line statements

```
score: int = 57

when score: int [
    >= 38 :: [
        print("Congratulations, you passed!").
        award_certificate("Certificate of Achievement").
        notify("Your passing grade is recorded.").
    ]
    _ :: [
        print("Unfortunately, you did not pass.").
        recommend("Review the material and try again.").
        log_event("Student scored below passing threshold.").
    ]
]
```

Example Output

1. Single-line statement

```
You are eligible to vote.
```

2. Multi-line statements

```
Congratulations, you passed!
Your passing grade is recorded.
```

If-else if-else/Switch Statement

Syntax

1. Single-line statement

```
when var: type [
    condition1 :: statement.
    condition2 :: statement.
    ...
    conditionn :: statement.
    _ :: statement.
]
```

2. Multi-line statements

```
when var: type [
    condition1 :: [
        statement.
        statement.
        statement.
    ]
    condition2 :: [
        statement.
        statement.
        statement.
    ]
    ...
    conditionn :: [
        statement.
        statement.
        statement.
    ]
    _ :: [
        statement.
        statement.
        statement.
    ]
]
```

Example Usage

1. Single-line statement

```
temperature: int = 25

when temperature: int [
    > 30 :: print("It’s hot outside!").
    > 20 :: print("It’s warm outside!").
    > 10 :: print("It’s cool outside!").
    _ :: print("It’s cold outside!").
]
```

2. Multi-line statements

```
order_status: string = "shipped"

when order_status: string [
    "pending" :: [
        print("Your order is being processed.").
        notify("You will be updated when it’s shipped.").
        log_event("Order is pending.").
    ]
    "shipped" :: [
        print("Your order has been shipped.").
        track_shipment("Tracking number: #1234ABCD").
        notify("You can track your order online.").
    ]
    "delivered" :: [
        print("Your order has been delivered.").
        update_status("Marking as complete in the system.").
        notify("Thank you for shopping with us!").
    ]
    "canceled" :: [
        print("Your order has been canceled.").
        refund("Processing your refund.").
        notify("We are sorry for the inconvenience.").
    ]
    _ :: [
        print("Unknown order status. Please contact support.").
        log_event("Invalid order status.").
    ]
]
```

Example Output

1. Single-line statement

```
It's warm outside!
```

2. Multi-line statements

```
Your order has been shipped.
You can track your order online.
```

E. Iterative Statement

Infinite Loop

Syntax

```
loop [
    # Code to execute
]
```

Example Usage

```
loop [
    print["This is an infinite loop"].
]
```

Example Output

```
This is an infinite loop
This is an infinite loop
This is an infinite loop
...
```

Controlled Loop

Syntax

```
loop i: int in range(start, end, step_exp) [
    # Code to execute
]
```

Example Usage

```
loop i: int in range(0, 5, _i++) [
    print[i].
]
```

Example Output

```
0
1
2
3
4
5
```