TWT Programming Language

(Talagang Walang Tulog)

Language Design and Functionalities

Jahziel Rae Arceo
Gabriel Kelly Navarro
Mikaela Jun Lenon
Maria Rosario Gueco
Marbille Juntado

CS 150 HTWX

December 2015

Chapter 1

Introduction

1.1 The TWT programming language

TWT is simply a programming language designed out of curiosity. There were really many challenges of deciding on which fits best on the category of a usable language, which the designers took so much time for considering and creating such.

1.1.1 Language Name

Halfway before the appropriate parser was created for the language, there was no appropriate meaning that the programmers can imply on the initials TWT, except that it was Twitter-inspired. However, since they were inspired by the countless nights that they weren't of good sleep, they made a way for the meaning of the name. TWT refers to "Talagang Walang Tulog", or "really no sleep", the phenomenon that happened to them during the creation of the language.

1.1.2 Paradigm

The TWT programming language is imperative and functional. The programming language works in a fashion nearly similar to C, Python, and Pascal. Function declarations are C-styled. Also, keywords that begin and end a program are used in the language, which seems to be like Pascal.

The language also borrows features from Python, which was the language used to create the new language.

1.1.3 Inspiration

Twitter, a social networking site, was the main inspiration on creating the language. As seen on the next chapter, the keywords and reserved words of the language mostly came from the jargons seen on the website.

Other than Twitter, the programmers also made the intricacy of programming an inspiration to create the language. Syntaxes on the language are simpler compared to other languages, which makes new programmers learn the language easily.

Chapter 2

Grammar Definition

In this chapter, we get to learn about the grammar of the language using its Backus-Naur form (BNF). By this, we can get a hint of what the language looks like on encoding.

2.1 Identifiers

The following conventions apply to variables and values entered in the programming language:

```
<INT> ->          <Digit><INT> | <Digit>
<Digit> ->        "0" | "1" | "2" | "3" | "4" | "5" | "6"
                  | "7" | "8" | "9"

<CHAR> ->         "'"<ASCII>"'"

<FLOAT> ->        <Digit>"."<Digit>
                  (Note: Strictly followed)

<STRING> ->       """<STRINGEXT>"""

<STRINGEXT> ->    <CHAR><STRINGEXT> | <CHAR> | EPSILON

<VARIABLE> ->     <ALPHABET><VARIABLEEXT>

<VARIABLEEXT> -> <ALPHANUMERIC><VARIABLEEXT>
                  | <ALPHANUMERIC> | EPSILON
```

Note that the grammar for float type values is strict in the programming language, so it will not accept an integer as float. The data must explicitly indicate that it is indeed a floating-point value.

2.2 Data Types

The TWT language is statically-typed. The programmer should explicitly state the data type he is to use.

Data types were defined in the language grammar as follows:

```
<Dtype> -> "@INT" | "@CHIRP" | "@COKE" | "@MSG" | "@TRALSE"
```

Data types are as follows:

- **@INT:** integer
- **@CHIRP:** character (ASCII)
- **@COKE:** float
- **@MSG:** string
- **@TRALSE:** boolean

The '@' symbol is used to easily identify words as types. By inspiration, it takes the @-mention paradigm of Twitter as its data type recognition mechanism.

2.3 Expressions and Assignment Statements

The following rules apply to expressions and assignments:

```
<Assignment> ->  <DType> VARIABLE "=" VARIABLE
                | <DType> VARIABLE "=" <Reading>
                | "@INT" VARIABLE "=" INT
                | "@CHIRP" VARIABLE "=" CHAR
                | "@COKE" VARIABLE "=" FLOAT
                | "@MSG" VARIABLE "=" STRING
                | "@TRALSE" VARIABLE "=" "YES"
                | "@TRALSE" VARIABLE "=" "NO"
```

```
                | <DType> VARIABLE "=" "(" <Exp> ")"

<Reading> ->    "REPLY" VARIABLE

<Exp> ->        <Term> <ExpPrime>

<ExpPrime> ->   "+" <Term> <EspPrime>
                | "-" <Term> <ExpPrime>

<Term> ->       <Fact> <TermPrime>

<TermPrime> ->  "*" <Fact> <TermPrime>
                | "/" <Fact> <TermPrime>

<Fact> ->       INT | CHAR | FLOAT | VARIABLE | "(" <Exp> ")"
```

2.4 Statement Level Control Structures

The following rules apply to control structures available in the language:

```
<State> ->      <Loop> "#" <StatePrime>
                | <If> "#" <StatePrime>
                | <Assignment> "#" <StatePrime>
                | <Call> "#" <StatePrime>
                | <Printing> "#" <StatePrime>
                | <Control> "#" <StatePrime>

<StatePrime> ->  <Loop> "#" <StatePrime>
                | <If> "#" <StatePrime>
                | <Assignment> "#" <StatePrime>
                | <Call> "#" <StatePrime>
                | <Printing> "#" <StatePrime>
                | <Read> "#" <StatePrime>
                | <Control> "#" <StatePrime>

<Loop> ->       "RT" <Condition> {" <Block> "}"

<If> ->         "IF" <Condition> "FOLLOW" "{" <Block> "}"
                <ElseIf>
                <Else>
                | "IF" <Condition> "FOLLOW" "{" <Block> "}"
                <Else>
                | "IF" <Condition> "FOLLOW" "{" <Block> "}"

<ElseIf> ->     "ELSEIF" <Condition> "FOLLOW" "{" <Block> "}"
```

```
                <Elseif>

<Else> ->       "ELSE" <Condition> "FOLLOW" "{" <Block> "}"
```

2.5 Subprograms

This is how functions or subprograms are declared in the language:

```
<Declaration> -> <Dtype> VARIABLE "(" <Args> ")"
                "{" <Block> <Return> "}"
                | <Dtype> VARIABLE "(" <Args> ")" "{" <Block> "}"
```

2.6 Other syntaxes not listed on previous sections

```
<Program> ->    <Declaration> <Main>
                | <Main>

<Main> ->       "LOGIN" <Block> "LOGOUT"
                | "LOGIN" "LOGOUT"

<Block> ->      <State>

<Call> ->       "HOOT "VARIABLE "(" <Args> ")"

<Printing> ->   "TWEET" VARIABLE
                | "TWEET" "(" <EXP> ")"
                | "TWEET" INT
                | "TWEET" FLOAT
                | "TWEET" CHAR
                | "TWEET" STRING
                | "TWEET" TRUE
                | "TWEET" FALSE
<Reading> ->    "REPLY" VARIABLE

<Control> ->    "UNFOLLOW" | "LIKE" | "BLOCK"

<Condition> ->  "(" <Conditional> ")" | "~" <Condition>

<Conditional> -> VARIABLE <Conditional> VARIABLE
                | { VARIABLE | INT | CHAR | FLOAT } <CondOp>
                    <Condition>
```

```
                | <Conditional> <CondOp> <Conditional>
                | <Condition> <CondOp> { VARIABLE | INT | CHAR |
                    FLOAT }
                | { VARIABLE | INT | CHAR | FLOAT } <CondOp>
                { VARIABLE | INT | CHAR | FLOAT }

<CondOp> ->     ">=" | "<=" | "==" | ">" | "<"

<Args> ->       <Dtype> <Vname> "," <Args>

<Return> ->     "REPORT" { INT | CHAR | FLOAT | STRING | VARIABLE }
                | "REPORT" <Exp>
```

Chapter 3

Lexical and Syntax Analysis

3.1 Parser

The parser for the language uses LR parsing, so no parse tables were used to parse the strings for the language. By that, it is obvious in the grammar that revisions for left recursion were done, so that no conflict will happen on the parsing process.

On the next lines of code is the Python code for the lexical analyzer, together with the token numbers assigned to them.

```
EOF = -1
int_dec = 3
float_dec = 4
char_dec = 5
string_dec = 6
bool_dec = 7
login = 1
logout = 0
if_state = 20
elseif_state = 21
else_state = 22
loop_state = 30
break_state = 31
continue_state = 32
exit_state = 33
exec_state = 34
read_state = 10
print_state = 11
return_state = 12
TRUE = 13
```

```
FALSE = 14
NOT = 15
INT = 71
FLOAT = 72
CHAR = 73
STRING = 74
openBrace = 41
closeBrace = 42
plusSign = 43
minusSign = 44
divSign = 45
mulSign = 46
asSign = 47
openParen = 48
closeParen = 49
ENDOFSTATE = 50
commaSign = 51
lessEqSign = 52
greatEqSign = 53
eqSign = 54
lesserSign = 55
greaterSign = 56
VARIABLE = 100

nextToken = 0

def isfloat(str):
    try:
        float(str)
    except ValueError:
        return False
    return True

def lex():
    global input
    global nextToken
    nextString = input[0]
    if nextString == 'EOF':
        nextToken = EOF
    elif nextString == 'LOGIN':
        nextToken = login
    elif nextString == 'LOGOUT':
        nextToken = logout
    elif nextString == '@INT':
        nextToken = int_dec
    elif nextString == '@COKE':
        nextToken = float_dec
    elif nextString == '@CHIRP':
        nextToken = char_dec
```

```
elif nextString == '@MSG':
    nextToken = string_dec
elif nextString == '@TRALSE':
    nextToken = bool_dec
elif nextString == 'IF':
    outfile.write('if ')
    nextToken = if_state
elif nextString == 'ELSEIF':
    outfile.write('elif ')
    nextToken = elseif_state
elif nextString == 'ELSE':
    outfile.write('else ')
    nextToken = else_state
elif nextString == 'RT':
    outfile.write('while ')
    nextToken = loop_state
elif nextString == 'UNFOLLOW':
    outfile.write('break')
    nextToken = break_state
elif nextString == 'LIKE':
    outfile.write('continue')
    nextToken = continue_state
elif nextString == 'BLOCK':
    outfile.write('exit')
    nextToken = exit_state
elif nextString == 'FOLLOW':
    nextToken = exec_state
elif nextString == 'REPLY':
    nextToken = read_state
elif nextString == 'TWEET':
    outfile.write('print ')
    nextToken = print_state
elif nextString == 'REPORT':
    nextToken = return_state
elif nextString == 'YES':
    nextToken = TRUE
elif nextString == 'NO':
    nextToken = FALSE
elif nextString == '~':
    nextToken = NOT
elif nextString.isdigit():
    outfile.write(nextString)
    nextToken = INT
elif nextString[0] == '-' and nextString[1:].isdigit():
    outfile.write(nextString)
    nextToken = INT
elif isfloat(nextString):
    outfile.write(nextString)
    nextToken = FLOAT
```

```
elif (nextString[0] == '\'') and (nextString[-1] == '\'') and
    (nextString.len() == 1):
    outfile.write(nextString)
    nextToken = CHAR
elif (nextString[0] == '\"') and (nextString[-1] == '\"'):
    outfile.write(nextString)
    nextToken = STRING
elif nextString == '{':
    nextToken = openBrace
elif nextString == '}':
    nextToken = closeBrace
elif nextString == '+':
    outfile.write(nextString)
    nextToken = plusSign
elif nextString == '-':
    outfile.write(nextString)
    nextToken = minusSign
elif nextString == '/':
    outfile.write(nextString)
    nextToken = divSign
elif nextString == '*':
    outfile.write(nextString)
    nextToken = mulSign
elif nextString == '=':
    outfile.write(nextString)
    nextToken = asSign
elif nextString == '(':
    outfile.write(nextString)
    nextToken = openParen
elif nextString == ')':
    outfile.write(nextString)
    nextToken = closeParen
elif nextString == '#':
    outfile.write('\n')
    nextToken = ENDOFSTATE
elif nextString == ',':
    outfile.write(nextString)
    nextToken = commaSign
elif nextString == '>=':
    outfile.write(nextString)
    nextToken = lessEqSign
elif nextString == '<=':
    outfile.write(nextString)
    nextToken = greatEqSign
elif nextString == '==':
    outfile.write(nextString)
    nextToken = eqSign
elif nextString == '>':
    outfile.write(nextString)
```

```
    nextToken = lesserSign
elif nextString == '<':
    outfile.write(nextString)
    nextToken = greaterSign
else:
    outfile.write(nextString)
    nextToken = VARIABLE
print("Next token is: " + str(nextToken))
print("Next string is: " + nextString)
del input[0]
```

All reserved words are explicitly stated in the analyzer.

Chapter 4

Names, Binding, and Scoping

4.1 Case sensitivity

The TWT programming language was designed in a way that the user must exactly write the language in whichever case it is. By that, TWT is case-sensitive. For example, **@int** is not the same with **@INT**.

4.2 Reserved words

The following is a list of the reserved words in the language:

LOGIN	LOGOUT	REPORT	TWEET	IF
ELSEIF	ELSE	HOOT	REPLY	UNFOLLOW
@INT	@CHIRP	@COKE	@MSG	@TRALSE
LIKE	BLOCK	YES	NO	

4.3 Name form

By convention, as explained in the grammar, the names must be alphabetic at the start of each string, for variables. The language follows C-style rules.

4.4 Binding

The language follows Python-based binding, but on the programming language scope, it was designed to be statically bound. As noticed, Python does dynamic binding, however, the programmers chose to require having types specified before each run.

4.5 Lifetime and scope

Conventions were also followed on variable lifetime and scope. That is, what Python generally has on variable lifetime and scope applies to TWT as well.

4.6 Blocks

In TWT, blocks can be determined by curly braces, as what C has. Note the following code example:

```
RT ( X < 4 ) {
     X = ( X + 1 ) #
     Y = ( Y + 2 ) #
     TWEET X #
 } #
```

The example above is a loop. It has the three statements ”X = (X + 1) #”, ”Y = (Y + 2) #”, and ”TWEET X #” in the block example.

Chapter 5

Data types

5.1 Primitive data types

As referenced to the grammar in Chapter 2, these are the data types available in the language:

- **@INT:** integer
- **@CHIRP:** character (ASCII)
- **@COKE:** float
- **@MSG:** string
- **@TRALSE:** boolean

5.2 Strings

The programming language supports strings, and is a primitive data type. Like C, the strings are determined by double quotation marks.

There are no limitations on string declarations, and everything that applies to every other language also applies to TWT.

5.3 Note on user-defined types and type-checking

As of writing, user-defined typing is not available in the language. The user must strictly use the available data types.

On type-checking and coercion, Python conventions are used, that is, the features that Python has are used in the language.

Chapter 6

Expression and Assignment Statements

6.1 Arithmetic expressions and overloading

This is the list of mathematical symbols used in the language.

```
+   -   *   /
=   (   )
```

Listed are its functions:

+	addition
-	subtraction
*****	multiplication
/	division
=	assignment
(	grouping (opening)
)	grouping (closing)

Also the language is using the infix notation for expressions. For the operators, there is no overloading. Operations are defined as to what they represent.

6.2 Boolean expressions

This is the list of boolean symbols used in the language.

YES NO == <=
>= < > ~

Listed are its functions:

YES	TRUE
NO	FALSE
==	equal
<=	less than or equal
>=	greater than or equal
<	less than
>	greater than
~	NOT

As what applies in most languages on boolean statements, the conventional approach to boolean expressions apply to TWT, specifically the operator precedence.