TAPE: A File Handling Language

Language Reference Manual

Tianhua Fang (tf2377)
Alexander Sato (as4628)
Priscilla Wang (pyw2102)
Edwin Chan (cc3919)

Programming Languages and Translators
COMSW 4115 Fall 2016

Table of Contents

1. Introduction
    1.1. Running a tape Program

2. Data Types

3. Lexical Elements
    3.1 Identifiers
    3.2 Tokens
    3.3 Literals
        3.3.1 Numeric Literals
        3.3.2 Character Literals
        3.3.3 String Literals
    3.5 Comments
    3.6 Whitespace
    3.7 Operators
        3.7.1 Arithmetic Operators
        3.7.2 Assignment Operators
        3.7.3 Relational Operators
        3.7.4 Logical Operators
        3.7.5 Other Operators

4. Regular Expressions
    4.1 Basic Syntax
    4.2 Search Patterns

5. Functions
    5.1 User-Defined Functions
    5.2 Built-In Functions
    5.3. Search Function

6. Variables
        6.1 Built-In Variables
    6.2 User-Defined Variables

7. Statements
    7.1 Condition statement
    7.2 Looping
        7.2.1 While Loop
        7.2.2 For Loop

8. Another Sample Program

9. References

1. Introduction

`Tape` is a data-driven scripting language that allows users to read, write, and manipulate documents easily. We draw inspiration from `awk` because `awk` is a great example of a data-driven language. Like `awk`, `tape` allows the users to describe the data they wish to work with and what they want to do to the data. This is the ideal language to use for file manipulation because it perform actions based on patterns. `Tape` compiles to LLVM.

1.1. Running a `tape` Program

All `tape` programs are written in files. Users can run the program by running the following command:

```
tape myTapeProgram.tp inputFile1.txt
```

2. Data Types

Data types are used for declaring variables or functions of different types. The type of a variable determines how the variable is interpreted.

The types in `tape` can be classified as follows:

`file`
File object contains filename and content

`int`
4 byte signed value (-2147483648 to 2147483647)

`float`
4 byte digit (3.4E +/- 38)

`char`
ASCII character

`string`
Sequence of char

`bool`
Value of 0 (false) and 1 (true)

`null`
The absence of values of all data types

3. Lexical Elements

Each lexical element in tape is formed from a sequence of characters. Lexical elements can be identifiers, tokens, literals, built-in variables, comment, or operator.

3.1 Identifiers

Identifiers are sequences of ASCII characters used for naming `tape` entities. Identifiers cannot have the same character sequence as a `tape` keyword.

Rules for `tape` identifiers:

1. The first character in an identifier must be an alphabet or an underscore and can be followed by digits.
2. Keywords can not be used as an identifier
3. Commas, periods, or quotes cannot be in identifiers.

3.2 Tokens

Tokens are special identifiers reserved for use as part of the programming language. Users will not be able to use them for other purposes. Tokens will be reserved by the compiler and hold by `tape`.

List of tokens recognized by `tape`:

```
file int float char string if else while for return after before
open close scan copy count readline write replace delete
```

3.3 Literals

Literals are numeric, characters or string values.

3.3.1 Numeric Literals

Numeric literals stands for a number. Numeric literals are defined by: sign part, a number part, an optional decimal point, and a fraction part. The sign part indicates whether an integer or float is positive or negative. The sign part is represented by the ASCII characters + and -. The number and fraction parts are defined by a single digit 0 or one digit from 1 to 9 followed by more digits from 0 to 9 .

Example:

```
# Valid numeric literals
1
3.14
4.23
0.3
```

```
+3.14

# Invalid numeric literals
.1
1e+3
e-2
```

3.3.2 Character Literals

Character literals are single ASCII characters enclosed in single quotes.

Example:

```
# Valid character literals
'a'
'9'

# Invalid character literals
a
3
```

3.3.3 String Literals

A string literal are a sequence of characters enclosed in double quotes, " and ". A string is considered as an array of characters.

String literals also contain the below backslash escaping in string:

- `\n` Represents a new line
- `\t` Represents a horizontal tab
- `\\` Represents a literal backslash
- `\s` Represents white space

Example:

```
# Valid string literals
"hello"
"6789"
```

3.5 Comments

Comments are followed by #. Anything followed by # will be ignored.

Example:

```
# This is a comment
```

```
# 12345 This is a comment
```

3.6 Whitespace

Whitespace is defined as tab character, space character, newline character. It will be ignored by compiler.

3.7 Operators

`Tape` allows users to use operators for different data types. All relational operators have the same precedence.

3.7.1 Arithmetic Operators

Arithmetic operators take numeric values (`int` or `float`) as their operands and return a single numeric value.

`+`
: Adds two operands.

`–`
: Subtracts right operand from the left.

`*`
: Multiplies both operands.

`/`
: Divides left operand by right operand.

`++`
: Increment operator increases the integer value by one.

`/`
: Decrement operator decreases the integer value by one.

3.7.2 Assignment Operators

Assignment operators allows users to assign a value to an identifier.

`=`
: Simple assignment operator. Assign values from right side operands to left side operand.

`+=`
: Add and assignment operator. Values of operands must be of type `int` or `float`.

`-=`
Subtract and assignment operator. Values of operands must be of type `int` or `float`.

`*=`
Multiply and assignment operator. Values of operands must be of type `int` or `float`.

`/=`
Divide and assignment operator. Values of operands must be of type `int` or `float`.

3.7.3 Relational Operators

Relational operators test for relations between the two operands. `Tape` has the below relational operators:

`==`
Checks if the values of two operands are equal or not. If yes, then condition becomes true

`!=`
Checks if the values of two operands are equal or not. If values are not equal then the condition becomes true.

`<`
Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.

`>`
Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.

`<=`
Checks if value of left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.

`>=`
Checks if value of right operand is less than or equal to the value of the left operand. If yes, then the condition becomes true.

3.7.4 Logical Operators

Logical Operators or only used to compare booleans.

`&&`
Returns true if both operands are true.

`||`
Returns true if one of the operands is true.

3.7.5 Other Operators

`~`
: Checks if two regular expressions of the operands matches. If yes, then the condition becomes true.

`!~`
: Checks if two regular expressions of the operands are not equal. If yes, then the condition becomes true.

`?:`
: Conditional expression.

`$`
: Field reference.

4. Regular Expressions

Since pattern matching plays a big role in `tape`, regular expressions are often used. Regular expressions are used to describe sets of strings.

A regular expression is enclosed in slashes, '`/`', is a pattern that matches every input record whose text belongs to that set.

4.1 Basic Syntax

Declaration of variables, and opening/closing of files is done in a similar manner to `C`.
Most of the body of the program will be done in a similar manner to `awk`: pattern matching followed with actions.

```
/ pattern / { action; } file_to_parse ;
```

The pattern can be a literal. `Char` literals must be enclosed in single quotes while string literals must be enclosed in double quotes.

Example:

```
/'c'/ { foo() } 'file.txt'; #do foo() if 'c' is found
/var/ { bar() } 'file.txt'; #do bar() if var is found
```

Pattern matching can also extend to regular expressions.

Pattern matching will occur for each line in a given file, and the action will occur for each line where the pattern matches.

4.2 Search Patterns

Tape allows users to combine regular expressions with regular expression operators to increase the power and versatility of regular expressions.

`/cat/`
: This searches for the pattern enclosed in the forward slashes, "`/`". This example searches for any line that contains the string "cat". This will not match "Cat" since `tape` is case sensitive. However, it will match words like "wildcats" or "bobcat".

`/^cat/`
: Preceding the string with a "`^`" tells tape to search for the string at the *beginning* of input line. This example matches any line that *begins* with the string "cat".

`/cat$/`
: Ending the string with "`$`" searches for lines that *end* with the string. This example matches any line that *ends* with the string "cat".

`/apple|banana/`
: "`|`" allows the program to match with either of the regular expressions next to it. This example matches with any line that contains the string "apple" or the string "banana".

`/(wild|bob)cat/`
: "`()`" allows users to group in regular expressions as in arithmetic. Notice how parentheses are used to group two expressions. This example matches with any line that contains the string "wildcat" or the string "bobcat".

`/cat*/`
: "`*`" is used when the preceding regular expression is to be repeated as many times as necessary to find a match. This example matches string "ca", "cat", catt", and any other variation of "cat" with any number of 't's at the end of the string.

`/\(cat\)/`
: "`\`" is used to suppress the special meaning of a character when matching. It matches any one of the characters that are enclosed in the square brackets. This example matches any line that contains the string "(cat)".

Note: In addition we can also apply a search to specific fields. Refer to Section 5.3.

5. Functions

5.1 User-Defined Functions

User defined functions are very similar to the ones found in `C`. In `tape`, we declare functions by using the function keyword. There is no need to specify the return type of the function or the type of the parameters.

User-defined functions consist of a name, parameter-list, and body-of-function. The A valid function name is also a valid variable name. Parameter-list is a list of the function's arguments and local variable names, separated by commas. The body-of-function is consists of statements about what the function does.

Example:
The below user-defined function prints a `str num` times.

```
function myFunction(str, num){
    for (i=0; i<num; ++i) {
        print(str);
    }
}
```

5.2 Built-In Functions

`Tape` has some built-in functions that users can use in their code. To call a built-in function, write the name of the function followed by arguments in parentheses. Each built-in functions accepts a certain number of arguments. If users omit or adds extra arguments, they will get an error message.

`index(String a, String b)`
The `index` function will search the string a for the first occurrence of the string `b`, and returns the position in character where that occurrence begin in `a`.

`substr(String s1, int index1, int index2)`
The `substr` function returns a length-character-long substring of `s1`. It starts at the index `index1` or `s1` and ends at `index2` of `s1`..

`tolower(string s)`
Returns a copy of string s, with each upper-case in the string replaced with its lower-case character.

`toupper(string s)`
Returns a copy of string s, with each lower-case in the string replaced with its upper-case character.

5.3. Search Function

Tape also allows users to apply a search to specific fields.

`$1 == "hello"`

This would search in the first field for the string "hello"

`$2 == 100`

This would search in the second field for the int 100.

6. Variables

6.1 Built-In Variables

`Tape` allows a simple way of searching and manipulating a file by using its built in variables.

`Tape` uses $ and an integer as variables to keeps track of the tokens in a given line. $1 is the first literal that appears in a given line, $2 is the second literal that appears, and so forth. Undefined values for these field variables would yield a null value. $0 can be used to represent the entire line.

`FS`

Input field separator variable. `Tape` reads and parses each line from input based on whitespace character by default and set the variables $1, $2, and etc. `FS` is used to set the field separator for each record. `FS` can be changed any number of times, it retains its values until it is explicitly changed.

Example:

```
# In a given names.txt we have the following data:

#Chan, Edwin: 1
#Fang, Tianhua: 2
#Sato, Alexander: 3
#Wang, Priscilla: 4

#To use this file with the Tape program "example.tp"

TAPE example.tp names.txt




#In example.tp

FS= /, |:/; #Fields are separated with a comma followed by a space
                #or by a ":"
```

```
#Print the last names.
print $1;
#Print the first names.
print $2;
#Print the first then the last name
print $2” “$1;
#Print the number, “: ”, first name, and then last name.
print $3”: ”$2” “$1;
```

OFS

Output Field Separator Variable. By default, OFS is a blank “” character.

Example:

```
# Using names.txt again…

FS= /, |:/;
OFS= ”_”;

print $1$2;
# Output in the following format: FIRSTNAME_LASTNAME_NUMBER
print $2$1$3;
```

RS

Record Separator variable. This field determines when to decide when the next record is. By default, a new line is a new record, so in TAPE, the newline character (\n) is the default value of RS.

ORS

Output Record Separator Variable. Each record in the output will be printed with this delimiter. Once again, by default, this value is set to the newline character (\n).

NR

Number of Records Variable. NR keeps a current count of the number of input lines. It gives you the total number of records being processed or line number. In our names.txt, NR = 4.

NF

Number of Fields in a record. NF keeps a count of the number of words in an input line. In our names.txt, NF = 3.

6.2 User-Defined Variables

Variables are declared in the following order: type, variable name, and value.

Example:

```
#file x => "C:/user/doc.txt";
int y = 1;
float z = 2.0;
char c = 'a';
string i = "hello world";
```

7. Statements

7.1 Condition statement

`If`, `else if`, and `else` statements are used to check the conditions. If the conditions return true, the actions within the parentheses will be performed.

Example:

```
if(x == y){
   print("true");
}else if(x != y){
   print("false");
}else {
   print("error");
}
```

7.2 Looping

Loop statements allow users to execute a statement or group of statements multiple times as long as the conditions.

7.2.1 While Loop

`While` loops repeatedly execute the statements in `{}` as long as the condition within the `()` is true.

Example:

```
int i = 0;
while(i < 5){
    print(i);
    i++;
}
```

7.2.2 For Loop

`For` loops iterate through a range of values and execute the statements in `{}` each time.

Example:

```
for(int j = 0; j < 5; j++){
    print(j);
}
```

8. Another Sample Program

Below is a sample `tape` program. The program opens a file and searches for a word in the file. It prints out the entire file if the word is found in the file.

```
1  # This is a simple TAPE program!
2  #We use this by entering "TAPE sample.tp myfile.txt" into the terminal.
3  string myWord = "test";
4  int myInt = 0;
6
7  /test/ {print $0;
8          print myWord "end.";} #Searches for the literal "test"
9                                #in the currently open file.
10                               #prints out the entire file if it is found.
11                               #Prints out test end afterwards.
12
```

9. References

https://www.chemie.fu-berlin.de/chemnet/use/info/gawk/gawk_3.html
https://en.wikipedia.org/wiki/Data-driven_programming
https://www.gnu.org/software/gawk/manual/html_node/Numeric-Functions.html
https://www.math.utah.edu/docs/info/gawk_13.html
https://www.tutorialspoint.com/awk/awk_arithmetic_functions.htm
http://vc.airvectors.net/tsawk_2.html
http://www.thegeekstuff.com/2010/01/8-powerful-awk-built-in-variables-fs-ofs-rs-ors-nr-nf-filename-fnr/
https://www.chemie.fu-berlin.de/chemnet/use/info/gawk/gawk_toc.html#TOC25