GLYPHIC Language Description

Louis Ebneth

GLYPHIC Language Description

Faculty:	Informatik und Mathematik
Course of study:	Allgemeine Informatik
Student:	Louis Ebneth
Matriculation number:	3345711

Contents

Chapter 1

Introduction

1.1 General

The language GLYPHIC was inspired by the natural language. Therefore, all keywords and operators are written out in English.

Operations like assigning or accessing a variable or calling a function are clearly differentiated using a different operator symbol.

GLYPHIC is dynamically typed, which means that the type of a variable is determined at runtime and can be changed over the course of the program's execution.

The parser first builds an abstract syntax tree (AST) from the input code. Then, the AST is printed to a ".gv" file using the Graphviz library. Following this step, the AST is optimized by evaluating constant expressions and removing unreachable nodes. The AST is then printed again in this optimized form. Finally, the AST is traversed and executed by the executor function.

Variables and functions are stored in their individual stores and are both accessed using an identifier. The stores are both implemented as stacks.

When calling a function, the parameters are added to a separate queue before jumping to the AST node representing the start of the function body. There, the parameters are retrieved from the queue and assigned to the parameters in the given order.

1.2 Project Structure

```
GLYPHIC/
    |-- build/                    # Compiled intermediate files
    |   |-- *.o
    |   |-- *.tab.c
    |   |-- *.tab.h
    |   +-- *.yy.c
    |-- graphs/                   # AST graphs in .gv and .png format
    |   |-- *.gv
    |   +-- *.png
    |-- include/                  # Include directory
    |   |-- ast.c
    |   |-- ast.h
    |   |-- function_params.c
    |   |-- function_params.h
    |   |-- function_storage.c
    |   |-- function_storage.h
    |   |-- variable_storage.c
    |   +-- variable_storage.h
    |-- programs/                 # Programs
    |   |-- frame.glyph
    |   |-- rpsls.glyph
    |   |-- towersOfHanoi.glyph
    |   +-- languageDemo.glyph
    |-- executor.sh               # Shell script for shorter cmd inputs
    |-- GLYPHIC                   # Executable
    |-- GLYPHIC.l                 # lex lexer file
    |-- GLYPHIC.y                 # GNU Bison file
    |-- makefile                  # Makefile
    +-- testfile.glyph            # Test program
```

Chapter 2

Language Specification

2.1 Comments

Line comments are specified by `~|` and block comments by `~{` and `}~` .

`~| This is a line comment.`

`~{ This is a block comment. }~`

2.2 Basic Data Types

2.2.1 Numbers

Numbers are always stored using the C data type double. If a number without decimals is printed, the “%g” formatting option is used.

2.2.2 Regular Notation

The number is equal to it’s face value.

`12345` `12.345` `-123.45`

2.2.3 Value-Base Notation

The base for the number is determined by the decimal number following the `r` .

`1C7r16` = 455 (base 16)

`541r11` = 650 (base 11)

2.2.4 Symbol Notation

The number is equal to the number of `#` following the `u` .

`u###` = 3

`u##########` = 10

2.2.5 Strings

Strings are stored using the C data type char*. They are always enclosed in `«` and `»`.

`«Hello, World!»` `«This is a string.»`

Control sequences are defined by `~`.

`«~n»` = a newline

`«~t»` = a tab

`«~< »` and `«~> »` for literal < and >

ASCII characters are defined by `~x` followed by the ASCII code.

`«~x41»` = 'A'

2.2.6 Boolean

Boolean values are stored using the C data type char*. They are equal to their written value `true` or `false`

2.2.7 Arrays

Arrays are identified by the `{` and `}` delimiter characters. They are available for the data types number, string, and boolean. Internally, they are stored as a struct containing a list of pointers to the respective data type (double* or char**) and an integer with the length of the array.

`{1 2 3 4 5}` `{«Hello» «World» «!»}` `{true false true}`

The length of an array is final once declared and cannot be changed.

The value of a field can be changed like a regular assignment:

`[myArray](2) equals 10`

If the requested index is out of bounds, the program will automatically print an error message.

2.3 Identifiers and Value Access

Valid identifiers have to be enclosed in `[` and `]`.

$[myVariable]$ $[x]$ $[myArray]$

To access the value of an identifier, the `~` operator is used.

`~[myVariable]` `~[x]` `~[myArray]`

To access the value of an array, the index is enclosed in `(` and `)` and placed after the identifier access. If the requested index is out of bounds, the program will automatically print an error message.

`~[myArray](2)` `~[myArray](~[index])`

Inside the index brackets, no further calculations can be performed. The following is therefore invalid:

`~[myArray](2plus1)`

2.4 Operators

Operators are defined by their literal string value.
`plus` : +
`minus` : -
`times` : *
`divby` : /
`and` : && (logical and)
`or` : || (logical or)
`equals` : Equality or Assignment operator
`isnt` : Inequality
`smaller` : <
`bigger` : >
`modulo` : %

Operations can be performed on the following data pairs:

number and number (all operators)

boolean and boolean (all operators, for mathematical operations: true = 1, false = 0)

number and boolean (boolean is converted to number: true = 1, false = 0)

string and string (plus for concatenation, equals and isnt for Equality, smaller and bigger for length comparison, modulo for substring check)

2.5 Keywords

The following keywords are available in GLYPHIC:
`if`
`do`
`otherwise`
`end`

repeat
give

2.6 Control Structures

2.6.1 Functions

This is an example of a function declaration for the function add that adds two numbers in GLYPHIC:

```
->[add]: ([x] [y]) =>
    [result] equals ~[x] plus ~[y]
    give ~[result]
end
```

The -> operator is used to declare a function. The function name is written as a regular identifier using [and] .

The parameter list is enclosed in (and) . The function body is enclosed in => and end .

Functions always need a give statement to return a value, even if it is 0 and unused.

A function is called by using the & operator followed by the identifier and passing the parameters in (and) .

```
    [result] equals &[add](10 20)
```

Inside the parameter list, no further calculations can be performed. The following is therefore invalid:

```
    [result] equals &[add](10 plus 1 20)
```

2.6.2 Loop

This is an example of a loop:

```
    [counter] equals 0
    repeat: <~[counter] smaller 10> do
        [counter] plus 1
    end
```

The repeat keyword is used to start a loop. The loop condition being evaluated to true / false is enclosed in < and > and is followed by the do keyword.

If the condition contains only a single number, the condition is true if the number is not 0.

The condition can consist of multiple statements chained together with `and` or `or`. As brackets are not allowed, the order of operations is from left to right.

```
    [counter] equals 0
    repeat: <~[counter] smaller 10 and ~[counter] bigger 0> do
        [counter] plus 1
    end
```

Each statement can consist of multiple calculations chained together.

```
    [counter] equals 0
    repeat: <~[counter] plus 1 smaller 10> do
        [counter] plus 1
    end
```

These two points also apply to the conditions of the condition and check structures detailed below.

2.6.3 Condition

This is an example of a condition:

```
    if: <~[x] equals 10>
        &[print] (<<Value of x is 10>>)
    otherwise:
        &[print] (<<x is not 10>>)
    end
```

Similar to the loop, the condition is enclosed in `<` and `>`.

2.6.4 Check

This is an example of a check:

```
    if: <~[x] equals 10>
        &[print] (<<Value of x is 10>>)
    end
```

It is a special version of a condition that does not have an `otherwise` branch.

2.7 Library Functions

The following functions are available in the GLYPHIC standard library and do not need to be declared by the user:

&[print]() prints any number of any parameter passed to it to stdout (string, number, boolean, variables, function returns)

&[readnum]() returns the next number read from stdin

&[readstr]() returns the next string read from stdin

&[random](double double) returns a random int (inclusive with the bounds)

&[strsplit](string) returns a list containing the parts of the string that were separated by whitespaces

&[strlen](string) returns the length of the given string (does not count the null terminator)

&[arrlen](array) returns the number of objects in the given array. The value is simply retrieved from the field of the corresponding struct.

2.8 Language "Quirks"

2.8.1 Shorter Assignment

Like assigning values to variables using [number] equals 10 operations like [number] plus 10 are also possible.

The previous statement is therefore equal to [number] equals ~[number] plus 10 .

2.8.2 Dynamic Typing

GLYPHIC is dynamically typed. This means that the following is possible:

```
    [counter] equals 0
    &[print](~[counter] <<~n>>) ~| prints 0

    [counter] equals false
    &[print](~[counter] <<~n>>) ~| prints false
```