OTH Regensburg
Computer Science and Mathematics

Study Project

Interpreter for the programming language STYX

Course: Compiler Construction

Finn Artmann

Winter Semester 2022/2023

Contents

1 Introduction

The programming language ”SŦYX” is an esoteric, interpreted programming language, which mainly uses UTF-8 special characters. It is therefore inherently hard to read. The syntax is oriented on the programming language C. The Interpreter is for SŦYX written in C using ”flex” and ”bison” as a lexer and parser generator. The interpreter builds an abstract syntax tree (AST) while parsing the source code and executes the AST afterwards. The following chapters will describe the syntax of SŦYX and the implementation of the interpreter.

2 Lexer implementation

The lexer is implemented in the file *src/styx.l*. Regarding the lexer implementation it is important to know how UTF-8 characters are encoded. Otherwise, there might be hard to debug issues when using regular expressions to match these characters. Flex scans the input byte-wise, but UTF-8 characters can be encoded with up to 4 bytes. So when we want to match a UTF-8 character, we need to make sure we match the whole sequence of bytes.
Additionally to recognizing keywords, identifiers, operators and other special character the lexer is also able to recognize numbers in different formats and supports linecomments and blockcomments. For those, the lexer tries to take as much work away from the parser as possible for example by completely ignoring the content of comments and converting numbers to decimal numbers when possible. Special number formats will be discussed in more detail in the specific chapter.
Also supported are String literals, which can still make use of special characters used elsewhere in the language by escaping them using the defined escape character. They are also able to span multiple lines with the use of newline symbols. Strings can also include arbitrary ascii characters by using the escape character.
Another unconventional feature compared to other programming languages is that lexemes are not separated by whitespace, but by the special character # or a newline to better fit in with the cryptic theme of the language.

3 Syntax

To keep this document in the requested length, only a few selected tokens are listed here explicitly. To review all tokens recognized by the lexer, please refer to the lexer implementation (*src/styx.l*).

3.1 Number formats

The lexer can recognize the following number formats:

- Decimal numbers (e.g. 42)
- Hexadecimal numbers (e.g. Ex2A)
- Real numbers (e.g. 3.14)
- Babylonian numbers (e.g. EB{YY <<}BE)

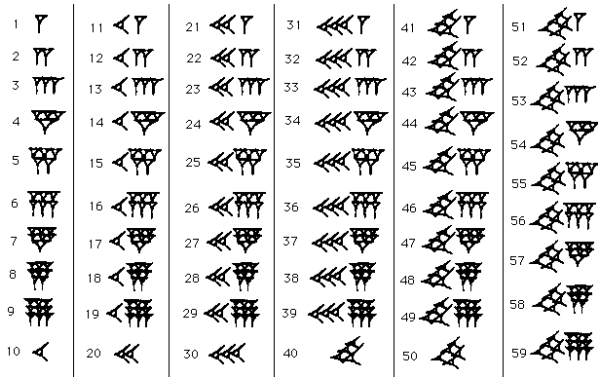
Decimal and Real numbers can be used in code in the expected way. Hexadecimal numbers need to be escaped with the Ex prefix.
While the first three are self-explanatory, the Babylonian numbers are a bit more complicated. They can be used in code like this:

EB{insert_babylonian_number_here}BE

The Babylonian number is then converted to a decimal number by the lexer. Valid characters for the babylonian numbers are:

- Y equals 1
- < equals 10
- Empty space is delimiter for the digits

The babylonian number system is a positional number system, where the base is 60. The following table show the representation of the numbers 1 to 59:



[1]

3.2 Data types

The supported data types are:

- ı’Ŧ (integer)
- ÐØ↑‘Ł€ (double)
- ßŦ®ı’ĵ (string) Usage: ⅜insert_string_here⅜
- ©ĦÆ® (character) Usage: Coming soon, see lexer file

- ,ØıÐ (void)

The language has a strong type system, which means that every variable has a type and the type of a variable can not be changed. The type of a variable is determined on declaration. Trying to assign a value of a different type to a variable will result in a type mismatch error.

3.3 Program structure

A SŦYX program is required to have a main function called ºÆı', which is the entry point of the program. The main function is required to have the return type ı'Ŧ and can not have any parameters. Functions have to be defined above the main function. Any statements or declarations that are not control structures are terminated by a semicolon.

3.4 Functions

SŦYX supports functions which can have parameters and a return type. The return type can be any of the supported data types. Functions can be called from within the main function or other functions and can be called recursively. They do not have to be defined in a specific order (except before the main function).
At the current state of the language, the return statement does not immediately stop the execution of the function, but instead returns the value to the caller at the end of the function. The returned value from a function call can directly be assigned to a variable of the same type or used as a parameter for another function call.
Here is some example code which shows the functions ”foo” and ”bar” and ”main”. The function ”main” has a print statement with the function call to ”foo” as argument, which itself has the integer 4 as argument. The function ”foo” calls the function ”bar”, which increments the given integer by 1 and returns it.

```
1  ı'Ŧ#'Æ®(ı'Ŧ#Ŋ)°
2  ####Þ®ı'Ŧ(⅜bar⅜);
3  ####®€Ŧ↑®'#Ŋ#¤#1;
4  °°
5  ı'Ŧ#ª∅∅(ı'Ŧ#Æ)°
6  ####Þ®ı'Ŧ(⅜foo⅜);
7  ####®€Ŧ↑®'#'Æ®(Æ);
8  °°
9  ı'Ŧ#ºÆı'()°
10 ####Þ®ı'Ŧ(ª∅∅(4));
11 °°
```

3.5 Variables

Variables can be of the data types described earlier. The variable has to be declared before it can be used. Declarations can be done anywhere in a function body or in the main function. A variable can also be declared as global variable using the ŋŁØ‘ÆŁ keyword, which makes it accessible from anywhere in the program. It is possible to directly assign a value to a variable on declaration. Local variables can not be accessed from outside the function or the scope they are declared in. Scopes can be created using ° and °° in any function body or the main function.

Examples Declarations:

- Standard declaration: ÐØ↑‘Ł€#,Æ®;
- Declaration with assignment: ÐØ↑‘Ł€#,Æ®#§#3.14;

3.6 Control flow

Implemented control flow statements are:

- If-Statement : ıª(condition)°body°°
- If-Else-Statement: ıª(condition)°body°°€ŁßŠ€°body°°
- For-Loop: ªØ®(declaration;condition;increment)°body°°

3.7 User I/O and other built-in functions

SŦYX has built-in functions for user input and output that work with all supported data types. ßŠ©Æ’(variable) reads a value from the standard input and assigns it to the variable. Þ®ı’Ŧ(variable) prints the value of the variable to the standard output. There also is the possibility to call the print function with an additional width parameter like this: Þ®ı’Ŧ(width?variable) This results in the output being formatted to the specified width.

Random integers beween 0 and a specified value can be generated using the ®Æ’Ðı’Ŧ(variable_max_number) function, which returns the generated number.

Another built-in function is ßŠ›ßŦ€º(variable_string), which allows the user to call a system command with a string as parameter. The return value of this function is 0 if the command was executed successfully and non zero otherwise.

4 Abstract Syntax Tree

The parser builds an abstract syntax tree (AST) while parsing the source code and executes the AST afterwards. Available functions can be found in the *include/ast.h* header file or in the *src/ast.c* source file.

4.1 AST nodes

Each node contains the following fields:

- **id** (int): A unique identifier for the node that is assigned on creation.
- **type** (int): The type of the node specifies the token type of the node.
- **data_type**: Used to determine the data type of the value stored in the val field.
- **val** (val_t): Stores the value of the node.
- **is_const** (int): Used to determine if the node is constant.
- **child** (struct astnode_t*): Stores pointers to the child nodes.

4.2 Debugging

After the AST is executed and fully traversed the parser will generate a Graphviz file, which can be used to visualize the AST. To display node names in a human-readable form, token IDs are converted to their corresponding token names using the internal token table provided by bison. There also is a function to print the AST to console.
By setting the pre-compiler directive *#define DEBUG_AST* in the header file each node of the AST will be printed to the console while traversing it.

4.3 Optimization

Constant folding is implemented for arithmetic operations. The optimization takes place while the AST is being traversed and is implemented in the *operation* function in the *src/ast.c* source file. When the AST is traversed, and an operation is encountered for the first time, the operation is evaluated, and the result is stored in the *value* field of the node. This optimization is only possible if all child nodes have a constant value. For this reason each ast node has a *is_const* field. A node is marked as constant if all child nodes are constant. When the original operation node is encountered again, the node is marked as constant and *value* field can be used instead of traversing the child nodes again.

References

[1] J O'Connor **and** E F Robertson. *Babylonian numerals*. https://mathshistory.st-andrews.ac.uk/HistTopics/Babylonian_numerals/. [Online; accessed 13-January-2023]. 2000.