Assembly Lexical Analyzer

Ramírez Castillo Daniela Guadalupe – 170659@upslp.edu.mx
Universidad Politécnica de San Luis Potosí

Abstract – This Project consist in the creation of a lexical analyzer for the Assembly programming language in Rust, by using finite state machines and regular expressions. The structures created in Rust to sort the tokens made for Assembly language into different categories in order to categorize all of them.

Index Terms: Programming, Rust Language, Assembly Language, Lexical analizar.

I. INTRODUCTION

In computer science, lexical analysis, lexing or tokenization is the process of converting a sequence of characters into a sequence of tokens (strings with an assigned and thus identified meaning).

A lexical token or simply token is a string with an assigned and thus identified meaning. It is structured as a pair consisting of a token name and an optional token value. The token name is a category of lexical unit.

Rust is a multi-paradigm programming language designed for performance and safety, especially safe concurrency.

Rust was originally designed by Graydon Hoare at Mozilla Research, with contributions from Dave Herman, Brendan Eich, and others The designers refined the language while writing the Servo layout or browser engine, and the Rust compiler.

Assembly language is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions.

Because assembly depends on the machine code instructions, every assembly language is designed for exactly one specific computer architecture. Assembly language may also be called symbolic machine code.

II. LEXICAL ANALYSIS

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

A. Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

III. ASSEMBLY LANGUAGE

Assembly language (or Assembler) is a compiled, low-level computer language. It is processor-dependent, since it basically translates the Assembler's mnemonics directly into the commands a particular CPU understands, on a one-to-one basis. These Assembler mnemonics are the instruction set for that processor. In addition, an Assembler provides commands that control the assembly process, handle initializations, and allow the use of variables and labels as well as controlling output.

A. Code

Most assembly language instructions require operands to be processed. An operand address provides the location, where the data to be processed is stored. Some instructions do not require an operand, whereas some other instructions may require one, two, or three operands.

This is a simple example of a ‘Hello World!”

```
.global main

.text
main:                          # This is called by C library's
startup code
 mov    $message, %rdi         # First integer (or pointer)
parameter in %rdi
      call    puts             # puts(message)
      ret                      # Return to C library code
message:
      .asciz "Hola, mundo"          # asciz puts a 0 byte at the
end
```

Where, label is the target label that identifies the target instruction as in the jump instructions. The LOOP instruction assumes that the ECX register contains the loop count. When

the loop instruction is executed, the ECX register is decremented and the control jumps to the target label, until the ECX register value, i.e., the counter reaches the value zero.

This is an example of a loop in Assambly Language.

```
section .text
  global _start ;must be declared for using gcc

_start: ;tell linker entry point
  mov ecx,10
  mov eax, '1'

l1:
  mov [num], eax
  mov eax, 4
  mov ebx, 1
  push ecx

  mov ecx, num
  mov edx, 1
  int 0x80

  mov eax, [num]
  sub eax, '0'
  inc eax
  add eax, '0'
  pop ecx
  loop l1

  mov eax,1 ;system call number (sys_exit)
  int 0x80 ;call kernel
section .bss
num resb 1
```

IV. IMPLEMENTATION

A. *Finite State Machine*

It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines.A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

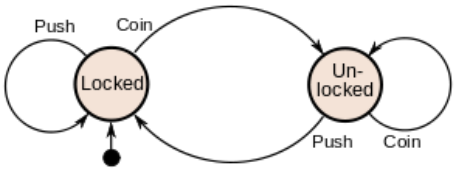

Ilustración 1 Example of a State Machine

B. *Constucting the lexer.*

First, we used functions and structures in rust to check if the analized token is a reserved word, identifier, operator, symbol or a value.

Next are the functions that were created to:

Reserved words

```
keywords: vec![
                String::from("global main"),
                String::from("text"),
                String::from("main"),
                String::from("mov"),
                String::from("section"),
                String::from("int"),
                String::from("push"),
                String::from("_start"),
                String::from("sub"),
                String::from("inc"),
                String::from("add"),
                String::from("pop"),
                String::from("loop"),
            ]
```

Identifier

```
fn token_is_identifier(&self, token: &str) -> bool
    {
        let regex = Regex::new(r"^[a-zA-
Z|_]*$").unwrap();
        regex.is_match(token)
    }
```

Operator

```
fn token_is_operator(&self, token: &str) -> bool
{
        let regex = Regex::new(r"^[>|<|.|=|+|\-
|*|/]$").unwrap();
        regex.is_match(token)
    }
```

Value

```
fn token_is_value(&self, token: &str) -> bool
{
        let regex =
Regex::new(r"^(['].*[']|\d+|\d*\.\d+)$").unwrap();
        regex.is_match(token)
    }
```

Symbol

```
fn token_is_symbol(&self, token: &str) -> bool
{
        let regex =
Regex::new(r"^[:|;|,|(|)]$").unwrap();
        regex.is_match(token)
    }
```

V. GLOSSARY

- Programming language: A programming language is a vocabulary and set of grammatical rules for instructing a computer or computing device to perform specific tasks. The term programming language usually refers to high-level languages.
- Syntax: refers to the rules that specify the correct combined sequence of symbols that can be used to

form a correctly structured program using a given programming language.

- Interpreter: An interpreter is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages. The interpreter transforms the high-level program into an intermediate language that it then executes
- Pattern: is a general, reusable solution to a commonly occurring problem within a given context in software design.
- Token: A token is a group of characters having collective meaning: typically a word or punctuation mark, separated by a lexical analyzer and passed to a parser.
- Pattern: A rule that describes the set of strings associated to a token. Expressed as a regular expression and describing how a particular token can be formed.
- Lexeme: is a sequence of characters in the source text that is matched by the pattern for a token.

VI. CONCLUSIÓN

The main task of a lexical analyzer are reading the input characters of the source code, creating groups of lexemes and produce an output of tokens for each lexeme.

Basically, we created a lexer sisnce the beginning, and even if its not as difficult as it seems, comprehending and finding Good sources for assembly language was the real problema here.

Even if Rust doesn't have that much documentation and almost all of the examples and articles about assembly language are old or discontinued, we still develop a basic lexer analyzer.

.

REFERENCES

[1] M´ırian Halfeld-Ferrari. Compilers - Lexical Analysis. TLC. https://www.univorleans.fr/lifo/Members/Mirian.Halfeld/Cours/TLComp/l3-0708- LexA.pdf

[2] Göteborgs universitet. 12. Finite-State Machines. 2013. http://www.cse.chalmers.se/~coquand/AUTOMATA/book.pdf.

[3] Assembly Programming Tutorial - Tutorialspoint. (s. f.). Tutorialspoint. Recuperado 6 de diciembre de 2020, de https://www.tutorialspoint.com/assembly_programming/index.htm

[4] Loyola Maymount University. (s. f.). Assembly Language Programming. Computer Science Department. Recuperado 5 de diciembre de 2020, de https://cs.lmu.edu/%7Eray/notes/x86assembly/