Compilador de Forth

Projeto final da UC Processamento de Linguagens

Universidade do Minho
Diogo Abreu, Luís Figueiredo, e Rodrigo Monteiro
{a100646, a100549, a100706}@alunos.uminho.pt

Grupo 21

1. Introdução

Neste projeto, desenvolvemos um compilador da linguagem Forth que gera código para a máquina virtual EWVM. Para isso, utilizamos um gerador de compiladores baseado em gramáticas tradutoras, concretamente o Yacc, e o gerador de analisadores léxicos Lex, versão PLY do Python.

2. Lex: Análise léxica

2.1. Tokens

```
# words
'COLON', # start
'SEMICOLON', # end

# math
'UCOMPARISON',
'COMPARISON',
'ARITHMETIC',

# numbers
'INTEGER',
'FLOAT',

# functions
'WORD',

# comments
'LPAREN',
'RPAREN',
'BACKSLASH',
'COMMENT',

# for loop
'DO', # start
'LOOP', # end
'PLUSLOOP', # end
```

```
# while loop
'BEGIN', # start
'UNTIL', # end
'AGAIN', # end
'WHILE', # middle
'REPEAT', # end

# conditional logic
'IF',  # start
'ELSE', # middle
'THEN', # end

# strings
'STRING',
'CHAR',
'KEY',

# variables
'VARIABLE',
'STORE',  # '!'
'PUSH', #  '@'
'CONSTANT'
```

2.2. Estados

```
states = (
    ('word', 'exclusive'),  # function/ word declaration
    ('commentp', 'exclusive'),  # comment with parentheses
    ('commentb', 'exclusive'),  # comment with backslash
    ('forloop', 'exclusive'),
    ('whileloop', 'exclusive'),
    ('ifstatement', 'exclusive'),
)
```

A gestão dos estados é feita a partir das funções `t.lexer.push_state` e `t.lexer.pop_state`. Deste modo, é possível entrar num estado e voltar para o anterior facilmente.

De modo a permitir que qualquer estado utilize uma dada regra, utilizamos a keyword `ANY` no nome da função. No entanto, certas regras são mais restritas e só podem ser utilizadas em certos estados. Por exemplo, a regra para reconhecer `LOOP` só pode ser utilizada no estado `forloop`.

Para além disso, a ordenação das regras é importante e proposital, isto é, certas regras precisam de ser definidas antes de outras para que o reconhecimento dos tokens seja feito corretamente. Por exemplo, a regra para reconhecer um `float` deve ser definida antes da regra para reconhecer um `integer` devido ao funcionamento do regex destas duas regras.

2.3. Regras

State	Function name	RegEx
Word	t_COLON	:\B
Word	t_word_SEMICOLON	\B;\B
For loop	t_ANY_DO	(do|DO)
For loop	t_forloop_LOOP	(loop|LOOP)
For loop	t_forloop_PLUSLOOP	\+(loop|LOOP)
While loop	t_ANY_BEGIN	(begin|BEGIN)
While loop	t_whileloop_WHILE	(while|WHILE)
While loop	t_whileloop_REPEAT	(repeat|REPEAT)
While loop	t_whileloop_UNTIL	(until|UNTIL)
While loop	t_whileloop_AGAIN	(again|AGAIN)
If statement	t_ANY_ifstatement_IF	(if|IF)
If statement	t_ifstatement_ELSE	(else|ELSE)
If statement	t_ifstatement_THEN	(then|THEN)

Comment	t_ANY_BACKSLASH	\\
Comment	t_commentb_COMMENT	[^\n]+
Comment	t_commentb_NEWLINE	\n
Comment	t_ANY_LPAREN	\(
Comment	t_commentp_COMMENT	[^)\n]+
Comment	t_commentp_RPAREN	\)
ANY	t_ANY_FLOAT	(\-?(?:0|[1-9]\d*)(?:\.\d+){1}(?:[eE]\d+)?)
ANY	t_ANY_UCOMPARISON	\d+(>=|<=|>|<|=)
ANY	t_ANY_INTEGER	\d+(?!\S)
ANY	t_ANY_ARITHMETIC	(\+|\-|*|\/|\%|\^|MOD|mod)
ANY	t_ANY_COMPARISON	(<=|>=|<>|=|<|>|AND|and|OR|or)
ANY	t_ANY_CHAR	(char|CHAR)\s+(?P<char>\S+)?
ANY	t_ANY_STRING	(?P<type>.){1}\"\s(?P<string>.+?)\"
ANY	t_ANY_VARIABLE	variable(?:.+?)(?P<var>\S+)
ANY	t_ANY_PUSH	@
ANY	t_ANY_STORE	!
ANY	t_ANY_CONSTANT	constant(?:.+?)(?P<var>\S+)
ANY	t_ANY_KEY	key
ANY	t_ANY_WORD	\S+
ANY	t_ANY_newline	\n+

2.4. **Testes**

O analisador léxico foi testado com os dados do ficheiro `tests.yaml` da diretoria `testing`.

```
with open("testing/tests.yaml", "r") as f:
        yaml_data = yaml.safe_load(f)

tests = yaml_data['tests']

for test in tests:
    print(f"Test: {test['name']}\n")
    lexer.input(test['input'])
    for tok in lexer:
        print(tok)
```

3. Yacc: Análise sintática

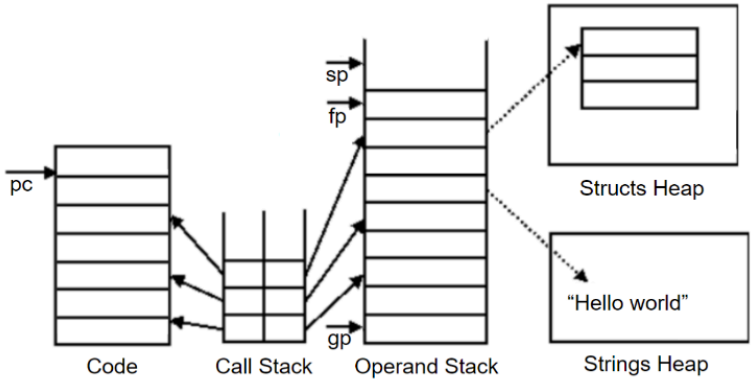
3.1. Gramática

```
All : Elements

Elements : Elements Element
         | &

Element : WordDefinition
        | Variable
        | Char
        | String
        | Arithmetic
        | Comparison
        | Integer
        | Float
        | IfStatement
        | WhileLoop
        | ForLoop
        | Store
        | Push
        | Word

BodyElement : Integer
            | Char
            | String
            | Arithmetic
            | Comparison
            | Float
            | IfStatement
            | ForLoop
            | WhileLoop
            | Store
            | Push
            | Word

Integer : INTEGER
Float : FLOAT
Arithmetic : ARITHMETIC
Comparison : COMPARISON
```

```
WordDefinition :
  COLON WORD WordBody SEMICOLON
WordBody : WordBodyElements
WordBodyElements :
  WordBodyElements BodyElement
  | &


ForLoop : DO FLBody LOOP
FLBody : FLBodyElements
FLBodyElements :
  FLBodyElements BodyElement
  | &


IfStatement :
     IF ISBody THEN
  | IF ISBody ELSE ISBody THEN
ISBody : ISBodyElements
ISBodyElements :
  ISBodyElements BodyElement
  | &


WhileLoop : BEGIN WLBody UNTIL
WLBody : WLBodyElements
WLBodyElements :
  WLBodyElements BodyElement
  |


Char : CHAR
String : STRING
Word : WORD
Variable : VARIABLE
Store : STORE
Push : PUSH
```

3.2. **Stack**

A linguagem Forth é *stack oriented*, o que significa que todas as operações são realizadas sobre uma pilha de dados. A EWVM possui duas stacks (*call stack* e *operand stack*), e duas heaps (*structs heap* e *strings heap*).



No entanto, quando é chamada uma *label*, não é possível modificar elementos adicionados anteriormente à chamada, i.e., entre o *global pointer* e o *frame pointer*.

Portanto, decidimos criar uma stack adicional, de modo a que fosse possível utilizar mais funcionalidades da EWVM, como *labels*, *calls* e *jumps*.

```
ALLOC 20

MYPUSH:
    PUSHG 0 DUP 1 DUP 2
    LOAD 0 PADD PUSHFP LOAD -1 STORE 1
    LOAD 0 PUSHI 1 ADD STORE 0
    RETURN

MYPOP:
    PUSHG 0 DUP 1
    LOAD 0 PUSHI 1 SUB DUP 1
    PUSHG 0 SWAP STORE 0 PADD LOAD 1
    RETURN
```

O *stack pointer* é guardado na posição 0 da struct alocada. A função `MYPUSH` é responsável por adicionar um elemento à stack, e a função `MYPOP` por retirar um elemento da stack.

O aspeto negativo desta abordagem é que são geradas mais instruções ao utilizar a stack adicional, o que pode tornar o código menos eficiente.

3.3. **Funções**

O código das palavras definidas é guardado num dicionário, `parser.words`, onde a chave é o nome da *label* – as *words* são convertidas para *labels* através da função `get_next_word_label`. Assim, quando uma palavra é chamada, basta chamar a *label* correspondente, `PUSHA <label> CALL`.

```
def p_WordDefinition(p):
    """WordDefinition : COLON WORD WordBody SEMICOLON"""
    p2_to_lower = p[2].lower()
    if p2_to_lower not in parser.reserved_words:

        if p2_to_lower in parser.variables:
            parser.variables.pop(p2_to_lower)

        word_label = get_next_word_label()
        parser.word_to_label[p2_to_lower] = word_label
        parser.words[word_label] = p[3]
    else:
        raise Exception("Reserved word")

    p[0] = []
```

3.4. **Expressões aritméticas**

Para efetuar operações aritméticas, é necessário retirar os dois elementos do topo da stack, efetuar a operação e adicionar o resultado à stack.

```
def p_Arithmetic(p):
    """Arithmetic : ARITHMETIC"""
    temp = ""
    if p[1] == "+":
        temp = "\tADD"
    # ...
    p[0] = [
      "PUSHA MYPOP CALL",
      "PUSHA MYPOP CALL",
      "SWAP", temp,
      "PUSHA MYPUSH CALL POP 1"
    ]
```

3.5. Caracteres e strings

As funções `cr` e `emit` são definidas como *reserved words* no dicionário `parser.reserved_words`.

```
parser.reserved_words = {
    "cr" : ["WRITELN"],
    "emit" : ["PUSHA MYPOP CALL","WRITECHR",],
}
```

Utilizamos a operação `CHRCODE` para converter um caracter para o seu código ASCII, e a operação `WRITES` para imprimir uma string.

```
def p_Char(p):
    """Char : CHAR"""
    p[0] = [
      "PUSHS \"" + str(p[1]) + "\" CHRCODE",
      "PUSHA MYPUSH CALL POP 1"
    ]

def p_String(p):
    """String : STRING"""
    if p[1][0] == '.':
        p[0] = "PUSHS \"" + p[1][1] + "\" WRITES"
```

3.6. Condicionais

O código corresponde às partes do *if statement* é guardado no dicionário `parser.if_statements`, onde a chave é a *label* correspondente ao *if statement*.

```
parser.if_statements = { "EMPTYELSE": [] }
parser.if_statement_idx = 0
```

Depois de retirado o elemento do topo da stack, é utilizada a operação de controlo `JZ` que verifica se este é zero. Se for, salta para a *label* especificada (com o código correspondente), caso contrário, retorna (se não houver um `else`), ou salta para a *label* correspondente ao `else`.

```
def p_IfStatement(p):
    """IfStatement : IF ISBody THEN
                   | IF ISBody ELSE ISBody THEN"""
    if len(p) == 4:
        label = next_if_statement_label()
        p[0] = ["PUSHA " + label + " CALL"]
        parser.if_statements[label] = [
          "PUSHA MYPOP CALL", "JZ EMPTYELSE"] + p[2]
    else:
        # ...
```

3.7. Ciclos

3.7.1. Inicialização do ciclo

Os ciclos `for` utilizam uma struct que guarda dois valores: o índice em que começa e o limite do ciclo.

Utilizamos a *operand stack* para guardar os valores atuais do índice e do limite do ciclo, os quais são restaurados no final do ciclo. Usamos esta estratégia para que `nested loops` funcionem corretamente utilizando a mesma struct.

Assim, os dois valores do topo da stack adicional são guardados na struct, a *label* do ciclo é chamada, e no final esses valores são substituídos/ "restaurados" pelos valores da iteração do ciclo anterior (caso exista).

```
# load struct values
"PUSHG 0",
"LOAD 0",
"PUSHG 0",
"LOAD 1",

# pop idx and limit
"PUSHA MYPOP CALL",
"PUSHA MYPOP CALL",
"SWAP",

# store idx
"PUSHG 0",
"SWAP",
"STORE 0",
```

```
# store limit
"PUSHG 0",
"SWAP",
"STORE 1",

# call loop
"PUSHA " + for_loop_label,
"CALL",

# restore struct values
"PUSHG 0",
"SWAP",
"STORE 1",
"PUSHG 0",
"SWAP",
"STORE 0",
```

3.7.2. Corpo do ciclo

Primeiramente, é verificado se o índice é menor que o limite. Caso seja, o corpo do ciclo é executado, e, no final, o índice é incrementado e o ciclo é repetido.

```
'PUSHG 0',
'LOAD 0',
'PUSHG 0',
'LOAD 1',
'INF',
'JZ ' + 'ENDLOOP',
```

```
'PUSHG 0',
'DUP 1',
'LOAD 0',
'PUSHI 1',
'ADD',
'STORE 0',
```

```
for_loop += ['JUMP ' + for_loop_label]
parser.for_loops[for_loop_label] = for_loop
p[0] = init
```

3.8. Variáveis

Cada variável é guardada num dicionário, `parser.variables`, onde a chave é o nome da variável e o valor é o índice da variável.

```
def p_Variable(p):
    """Variable : VARIABLE"""
    variable_to_lower = p[1].lower()
    # ...
    parser.variables[variable_to_lower] = parser.next_variable_idx
    parser.next_variable_idx += 1

    p[0] = []
```

As variáveis são guardadas numa struct com um dado número de posições. Quando se adiciona o endereço de uma variável à stack, esta é identificada como *WORD*, e é necessário somar o índice da variável ao endereço base da struct.

```
def p_Word(p):
    """Word : WORD"""
    # ...
        elif p1_to_lower in parser.variables:
            variable_number = parser.variables[p1_to_lower]
            p[0] = [
                "PUSHG " + str(VARIABLES_GP),
                "PUSHI " + str(variable_number),
                "PADD",
                "PUSHA MYPUSH CALL POP 1"
            ]
```

Para guardar um valor numa variável, é necessário retirar o valor do topo da stack e guardá-lo na posição correspondente da struct. E, para obter o valor de uma variável, é necessário utilizar a operação `LOAD`.

```
def p_Store(p):
    """Store : STORE"""
    p[0] = ["PUSHA MYPOP CALL", "PUSHA MYPOP CALL", "STORE 0"]

def p_Push(p):
    """Push : PUSH"""
    p[0] = ["PUSHA MYPOP CALL", "LOAD 0",  "PUSHA MYPUSH CALL POP 1"]
```

3.9. **Funções adicionais**

Algumas funções adicionais foram implementadas e estão guardadas no dicionário `parser.reserved_words`: `dup`, `2dup`, `drop`, `depth`, `spaces`, `key`, etc.

```
"drop": [
    "PUSHA DECPOINTER CALL",
],
"depth": [
    f"PUSHG {GP} LOAD 0",
    "PUSHA MYPUSH CALL POP 1"
],
```

3.10. **Testes**

3.10.1. Programa de testes

Os testes foram efetuados com o programa `test.py` na diretoria `testing`. Este programa:

1. Lê o ficheiro `tests.yaml` com recurso à biblioteca `yaml`, que contém os testes a efetuar.
2. Para cada teste, executa o programa `forth_yacc.py` através da biblioteca `subprocess` com o input do teste.

```
subprocess.run(
  ["python3", "forth_yacc.py", test['input']], cwd="../", check=True
)
with open("../output.txt", "r") as output_file:
    ewvm_code = output_file.read()
```

3. Para cada teste, chama a função `get_result` com o código gerado pelo `forth_yacc.py`. Esta função faz *web scraping* do site `https://ewvm.epl.di.uminho.pt/run` para obter o resultado do código gerado, com recurso à biblioteca `selenium`.

```
def get_result(code: str) -> str:
    textarea = driver.find_elements(By.NAME, "code")[0]
    textarea.send_keys(code)

    run_input = driver.find_element(By.XPATH, ...)
    run_input.click()

    result = driver.find_elements(By.XPATH, ...)
    result_str = ''.join(
      [r.text if r.text != "" else '\n' for r in result]
    )
    return result_str
```

4. Por fim, imprime os resultados para o stdout.

3.10.2. Testes efetuados e resultados[1]

Input	Resultado

`: EGGSIZE ( n -- )`
`DUP 18 < IF  ." reject "      ELSE`
`DUP 21 < IF  ." small "       ELSE`
`DUP 24 < IF  ." medium "      ELSE`
`DUP 27 < IF  ." large "       ELSE`
`DUP 30 < IF  ." extra large " ELSE`
`    ." error "`
`THEN THEN THEN THEN THEN DROP ;`
`23 EGGSIZE`
	`medium`

`2 0 DO`
`  1 . 2 0`
`  DO 2 . 2 0`
`  DO 3 . LOOP 2 0 DO 4 .`
`  LOOP`
`  LOOP`
`LOOP`
	`1233442334412334423344`

`: somatorio 0 swap 1 do i + loop ;` `11 somatorio .`	`55`

`." hello"`
`cr`
`." hello again"`
`cr 97 emit`
	`hello`
	`hello again`
	`a`

`: tofu ." Yummy bean curd!" ;`
`: sprouts ." Miniature`
`vegetables." ;`
`: menu CR tofu CR sprouts CR ;`
`menu`
	`Yummy bean curd!`
	`Miniature vegetables.`

`: ?FULL 12 = IF 391 .  THEN ;` `12 ?FULL`	`391`

`: ?DAY  32 < IF  ." Looks good "`
`ELSE  ." no way " THEN ;`
`33 ?DAY`
	`no way`

[1]Devido à mudança do limite de instruções de 7000 para 1000 na EWVM, alguns dos testes que envolvem ciclos deixaram de funcionar. No entanto, funcionam corretamente sem esse limite imposto.

```
: maior2 2dup > if swap then ;
: maior3 maior2 maior2 . ;
2 11 3 maior3
```

```
11
```

```
: maior2 2dup > if drop else swap
drop then ;
: maior3 maior2 maior2 ;
: maiorN depth 1 do maior2 loop ;
2 11 3 4 45 8 19 maiorN .
```

```
45
```

```
: RECTANGLE 25 0 DO I 5 MOD 0 = IF
CR  THEN ." *" LOOP ;
RECTANGLE
```

```
*****
*****
*****
*****
*****
```

```
: A  CR 4 1 DO  DUP I * .  LOOP
DROP ;
: B  CR 4 1 DO  I A  LOOP ;
B
```

```
123
246
369
```

```
: testing 10 9 8 7 begin . 10 =
until ;
testing
```

```
79
```

```
CHAR W .
CHAR % DUP . EMIT
CHAR A DUP .
32 + EMIT
```

```
8737%65a
```

```
1 . 10 spaces 1 .
```

```
1          1
```

```
variable x
5 x !
x @ .
variable y
4 y !
y @ .
```

```
54
```

4. Conclusões

Para concluir, achamos que conseguimos cumprir os requisitos do projeto, tendo sido implementado um analisador léxico e um analisador sintático para a linguagem Forth, que gera código para a EWVM, com suporte a expressões aritméticas, criação de funções, caracteres, strings, condicionais, ciclos e variáveis.

References

1. Documentação do PLY (Python Lex-Yacc), https://ply.readthedocs.io/en/latest/ply.html
2. EWVM manual, https://ewvm.epl.di.uminho.pt/manual
3. Forth Glossary, https://forth-standard.org/standard/core
4. Forth Loops, https://www.forth.com/starting-forth/6-forth-do-loops/
5. Teixeira, S. A.: EWVM - an Educational Web Virtual Machine. (2022)