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
        | Key
        | Arithmetic
        | Comparison
        | Integer
        | Float
        | IfStatement
        | WhileLoop
        | ForLoop
        | Store
        | Push
        | Word

BodyElement : Integer
            | Char
            | String
            | Key
            | 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
Key : KEY
Word : WORD
Variable : VARIABLE
Store : STORE
Push : PUSH
```

3.2. Funções

O código das funções, ou *words*, é guardado no dicionário `parser.words`, onde a chave é uma *label*, única para cada função, e o valor é uma lista de instruções.

```
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] = []
```

Quando uma WORD é encontrada, esta pode ser uma palavra reservada, como `cr` ou `emit`, uma variável, ou uma função que tenha sido definida no código.

Se for uma palavra reservada, então é introduzido o código guardado no dicionário `parser.reserved_words`, i.e., `p[0] = parser.reserved_words[p1_to_lower]`.

Se for uma variável, é introduzido na stack o endereço correspondente a essa variável, sendo que cada variável tem um índice associado, guardado no dicionário `parser.variables`. A partir deste índice, é possível aceder à posição da variável na struct.

```
variable_number = parser.variables[p1_to_lower]
p[0] = [
    "PUSHG " + str(VARIABLES_GP),
    "PUSHI " + str(variable_number),
    "PADD"
]
```

Se for uma função definida no código, então é introduzida a *label* correspondente.

```
label = parser.word_to_label.get(p1_to_lower, None)
  if label and label in parser.words:
      p[0] = [label]
```

No fim do *parsing*, estas *labels* são substituídas pelo código correspondente (até não existirem mais labels para substituir). Para além disso, é necessário modificar o nome das *labels* dentro da função (*for loops*, *while loops*, *if statements*), caso contrário, não se poderia chamar a mesma função mais do que uma vez, visto que as *labels* seriam redefinidas.

```
def replace_words(code):
    pattern_word = r'(\<\<(\w+?)(\d+)\>\>)'
    pattern_label = r'(FORLOOP|ENDLOOP|IFSTATEMENT|ENDIF|WHILELOOP)(\d+)'
    word_usage = defaultdict(int)
    max_labels = defaultdict(int)

    # repetir até não haver mais word labels para substituir
    while re.search(pattern_word, code):
        def replace_word(match):
          word = match.group(1)
          word_code = '\n'.join(parser.words[word])

          if word_usage[word] > 0:

            def replace_labels(match2):
              inc = get_label_value(match2.group(1))
              # incrementar o valor antigo da label pelo valor atual
              number = int(match2.group(2)) + inc
              max_labels[match2.group(1)] = max(
                max_labels[match2.group(1)], number
              )
              return match2.group(1) + str(number)

            # reajustar valores das labels
            word_code = re.sub(pattern_label, replace_labels, word_code)
            # incrementar labels de acordo com os máximos encontrados
            increment_labels(max_labels, 1)

          word_usage[word] += 1
          return word_code

      # substituir word labels por código
      code = re.sub(pattern_word, replace_word, code)

  return code
```

3.3. Expressões aritméticas

Para efetuar operações aritméticas, é introduzida a operação da EWVM correspondente à operação aritmética encontrada.

```
def p_Arithmetic(p):
    """Arithmetic : ARITHMETIC"""
    temp = ""
    if p[1] == "+":
        temp = "ADD"
    # ...
    p[0] = temp
```

3.4. Caracteres e strings

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

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

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

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

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

def p_Key(p):
    """Key : KEY"""
    p[0] = ["\tREAD", "\tCHRCODE"]
```

3.5. **Condicionais**

Para os condicionais, são utilizadas *labels* (sem *calls*) de modo a que seja possível saltar para o fim do *if statement* ou para o *else*.

Por exemplo, se a condição for verdadeira, o salto para a *label* do *else* não é efetuado (`JZ IFSTATEMENT`), depois é executado o código correspondente à condição verdadeira, e, por fim, é feito um salto para o fim do *if statement* (`JUMP ENDIF`), para que a *label* do *else* não seja executada. Se a condição for falsa, é executado um salto para a *label* do *else*, onde é executado o código correspondente – no fim deste código, inicia-se a label do `ENDIF`, onde estará o código restante do programa.

```
def p_IfStatement(p):
    """
    IfStatement : IF ISBody THEN
                | IF ISBody ELSE ISBody THEN
    """
    if len(p) == 4:
        endif = next_endif_label()
        p[0] = ["JZ " + endif] + p[2] + [endif + ':']
    else:
        endif = next_endif_label()
        label = next_if_statement_label()
        p[0] = ['JZ ' + label] + p[2] + ['JUMP ' + endif,
                label + ':'] + p[4] + [endif + ':']
```

Código Forth

```
: test 0 = if ." condição
verdadeira " else ." condição
falsa " then ;
0 test
```

Código EWVM

```
START
  PUSHI 0
  PUSHI 0
  EQUAL
  JZ IFSTATEMENT0
  PUSHS "condição verdadeira "
  WRITES
  JUMP ENDIF0
IFSTATEMENT0:
  PUSHS "condição falsa " WRITES
ENDIF0:
STOP
```

3.6. Ciclos

3.6.1. For loop

Os ciclos `for` utilizam uma struct que guarda dois valores: o índice em que começa e o limite do ciclo. Ao inicializar um ciclo, os valores atuais do índice e do limite são guardados numa *custom stack*, depois são inseridos os novos valores na struct, o ciclo é executado, e, no final, restauramos esses valores. Desta forma, é possível executar *nested loops*, visto que sempre que se executa um ciclo dentro de outro, quando este acaba o estado do ciclo anterior é restaurado.

```
init = [
    "PUSHG 0 LOAD 0", # load loop parameter values
    "PUSHG 0 LOAD 1",
    "PUSHA MYPUSH CALL POP 1", # store those in the stack
    "PUSHA MYPUSH CALL POP 1",
    "PUSHG 0 SWAP STORE 0", # store new loop parameter values
    "PUSHG 0 SWAP STORE 1",
]
```

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.

```
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
```

No corpo do ciclo, é verificado se o índice é menor que o limite. Caso seja, o índice é incrementado, o corpo do ciclo é executado, e, no final, é feito um salto para o início do ciclo. Caso contrário, é feito um salto para o fim do ciclo (`ENDLOOP`), onde os valores da struct são restaurados.

```
for_loop = [
    "PUSHG 0 LOAD 0",
    "PUSHG 0 LOAD 1",
    "INF",
    "JZ " + end_loop_label,
    'PUSHG 0', 'DUP 1', 'LOAD 0', 'PUSHI 1', 'ADD', 'STORE 0',
]

for_loop += p[2] + ['\tJUMP ' + for_loop_label]

restore = [
    'PUSHG 0', 'PUSHA MYPOP CALL', 'STORE 0',
    'PUSHG 0', 'PUSHA MYPOP CALL', 'STORE 1'
]

p[0] = init + [for_loop_label + ':'] + for_loop
        + [end_loop_label + ':'] + restore
```

Exemplo do código gerado para um ciclo `for` (`3 2 0 do 1 - dup dup . loop`).

```
ALLOC 2
ALLOC 21
ALLOC 10

START
  PUSHG 1 PUSHI 0 STORE 0
  PUSHG 0 PUSHI 0 STORE 0
  PUSHG 0 PUSHI 0 STORE 1
  PUSHI 3
  PUSHI 2
  PUSHI 0
  PUSHG 0 LOAD 0
  PUSHG 0 LOAD 1
  PUSHA MYPUSH CALL POP 1
  PUSHA MYPUSH CALL POP 1
  PUSHG 0 SWAP STORE 0
  PUSHG 0 SWAP STORE 1
FORLOOP0:
  PUSHG 0 LOAD 0
  PUSHG 0 LOAD 1
  INF JZ ENDLOOP0
  PUSHG 0
  DUP 1
  LOAD 0
  PUSHI 1
  ADD
  STORE 0
  PUSHI 1
  SUB
  DUP 1
  DUP 1
  WRITEI
  JUMP FORLOOP0
ENDLOOP0:
  PUSHG 0
  PUSHA MYPOP CALL
  STORE 0
  PUSHG 0
  PUSHA MYPOP CALL
  STORE 1
STOP
```

3.6.2. While loop

No ciclo `while`, o corpo do ciclo é executado enquanto o valor do topo da stack no final da execução do corpo do ciclo for diferente de zero.

```
def p_WhileLoop(p):
    """
    WhileLoop : BEGIN WLBody UNTIL
    """
    label = next_while_loop_label()
    p[0] = [label + ':'] + p[2] + ['JZ ' + label]
```

Código Forth

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

Código EWVM

```
START
  PUSHI 10
  PUSHI 9
  PUSHI 8
  PUSHI 7
WHILELOOP0:
  WRITEI
  PUSHI 10
  EQUAL
  JZ WHILELOOP0
STOP
```

3.7. **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"]
```

3.8. **Funções adicionais**

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

```
parser.reserved_words = {
    "cr" : ["WRITELN"],
    "emit" : ["WRITECHR"],
    "dup": ["DUP 1"],
    "2dup": ["PUSHA TWODUP CALL"],
    ...
}
parser.auxiliary_labels = {
  "2dup": (False, [
        "TWODUP:",
        "PUSHFP LOAD -2",
        "PUSHFP LOAD -1",
        "RETURN",
    ])
}
```

3.9. Testes

3.9.1. Programa de testes[1]

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.

[1] Com a nova versão do website da EWVM (1.3), este programa de testes deixou de funcionar.

3.9.2. Testes efetuados e resultados obtidos

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 CR`
`2 EGGSIZE CR`
`28 EGGSIZE`
	`medium`
	`reject`
	`extra large`

`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 ;` `100 somatorio .`	`4950`

`." 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`

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

```
11
```

```
: 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
```

```
: A 1 2 + ;
: B 3 0 do dup A + loop ;
: C B 3 0 do . 32 EMIT loop cr ;
3 C  4 C 10 B 10 C 2 A 100 C
```

```
12 9 6
13 10 7
19 16 13
109 106 103
```

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)