Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2023/2024

Compilador Forth
Grupo 14

Duarte Leitão(a100550) Diogo Barros(a100600) Carlos Costa(a88551)

12 de maio de 2024

PL

Data de Receção	
Responsável	
Avaliação	
Observações	

Compilador Forth
Grupo 14

Duarte Leitão(a100550) Diogo Barros(a100600) Carlos Costa(a88551)

12 de maio de 2024

Índice

1 Introdução

1.1 Contextualização

O Forth é uma linguagem de programação de baixo nível, baseada numa stack, que foi criada por Charles H. Moore na década de 1960. É conhecida pela sua simplicidade e eficiência, sendo amplamente utilizada em sistemas embebidos, controlo de hardware, sistemas operativos e outras aplicações que requerem desempenho e compactação.

A linguagem Forth tem como principais caraterísticas:

- **Stack:** Todas as operações em Forth são realizadas utilizando uma stack de dados. Os valores são empilhados e desempilhados para execução das operações;
- **Notação pós-fixa (RPN - Reverse Polish Notation):** Forth utiliza a notação pós-fix, onde os operadores são colocados após os seus operandos. Por exemplo, 2 3 + realiza a operação de adição entre 2 e 3;
- **Extensível:** Forth é uma linguagem extremamente extensível, permitindo que novas palavras (ou funções) sejam definidas pelo utilizador de forma rápida e fácil. Isso permite uma grande flexibilidade na criação de programas específicos para determinadas aplicações;
- **Interpretada ou compilada:** Forth pode ser interpretada diretamente a partir do código fonte ou compilada para código nativo, dependendo da implementação. No nosso caso, iremos compilá-la;
- **Eficiência e compactação:** Forth é conhecida pela sua eficiência e compactação de código. Isso a torna uma escolha popular para sistemas com recursos limitados de hardware.

1.2 Objetivos

Neste projeto o objetivo foi criar um compilador capaz de converter código forth em código máquina a ser interpretado pela EWVM.

Com a realização deste projeto, os seus principais objetivos foram:

- aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capaci- dade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora;
- desenvolver um compilador gerando código para um objetivo específico;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

2 Concepção

A implementação do programa baseia-se em duas partes fundamentais, a análise léxica e a análise sintática, feitas através da utilização de dois módulos, *lex* e *yacc* da ferramenta **PLY** do Python.

2.1 Análise Léxica

Primeiramente definiram-se os *tokens* da linguagem FORTH:

- **FUNCTION** $\longrightarrow$ Token que representa o nome de uma função de FORTH.
- **STRING** $\longrightarrow$ Tipo de valor delimitado por aspas.
- **NUMBER** $\longrightarrow$ Valor numérico
- **PLUS** $\longrightarrow$ Token para o operador + de soma aritmética.
- **MINUS** $\longrightarrow$ Token para o operador - de subtração aritmética.
- **TIMES** $\longrightarrow$ Token para o operador * de multiplicação.
- **DIVIDE** $\longrightarrow$ Token para o operador / de divisão.
- **MOD** $\longrightarrow$ Token para o operador % do resto.
- **NOT** $\longrightarrow$ Token para a operação booleana **NOT**.
- **INF** $\longrightarrow$ Token para o operador lógico $<$.
- **SUP** $\longrightarrow$ Token para o operador lógico $>$.
- **INFEQ** $\longrightarrow$ Token para o operador lógico $<=$.
- **SUPEQ** $\longrightarrow$ Token para o operador lógico $>=$.
- **DOT** $\longrightarrow$ Token indicador do operador . que em FORTH escreve no terminal o topo da *stack*.
- **STDOUT**

- **CHAR** $\longrightarrow$ Função de FORTH que devolve o valor ASCII do carácter no topo da *stack*.
- **COMMENT_LINE** $\longrightarrow$ Token que inicia um comentário em linha.
- **COMMENT_BLOCK** $\longrightarrow$ Token que inicia ou termina um bloco de comentário.
- **NEWLINE** $\longrightarrow$ Uma ou mais quebras de linha.
- **IF** $\longrightarrow$ Token que inicia um *statement* condicional.
- **ELSE** $\longrightarrow$ Token que indica o *statement* a executar caso a condição falhe.
- **THEN** $\longrightarrow$ Token que indica o *statement* a executar caso a condição seja verdadeira.
- **DO** $\longrightarrow$ Token que indica o trecho de código a executar no loop.
- **LOOP** $\longrightarrow$ Token para indicar um loop.
- **DROP** $\longrightarrow$ Função de manipulação de *stack* que remove e escreve em terminal o último elemento da *stack*.
- **SWAP** $\longrightarrow$ Função de manipulação de *stack* que troca os últimos dois elementos da *stack*.
- **ROT** $\longrightarrow$ Função de manipulação de *stack* que troca os últimos três elementos da *stack* de forma rotativa.
- **OVER** $\longrightarrow$ Função de manipulação de *stack* que duplica o penúltimo elemento da *stack* para o seu topo.
- **CONCAT** $\longrightarrow$ Função que concatena duas *Strings*.
- **DUP** $\longrightarrow$ Função de manipulação de *stack* que duplica o elemento no topo.
- **EMIT** $\longrightarrow$ Função que, mediante uma representação *ASCII* na *stack*, imprime o caráter correspondente.
- **CR** $\longrightarrow$ Função de FORTH com o propósito de *Carriage Return*.
- **KEY** $\longrightarrow$ Função FORTH que recebe input de um caráter do teclado.
- **SPACE** $\longrightarrow$ Função FORTH que imprime um espaço em branco.
- **SPACES** $\longrightarrow$ Função FORTH que imprime **N** espaços em branco.
- **2DUP** $\longrightarrow$ Função que copia os últimos 2 elementos da stack.
- **FUNC_BODY** $\longrightarrow$ O corpo de uma função definida em FORTH.
- **VARIABLE_DEFENITION** $\longrightarrow$ O nome de uma variável definida em FORTH.

- **VARIABLE_ASSIGNMENT** $\longrightarrow$ O valor da varíavel definida.
- **VARIABLE_FETCH** $\longrightarrow$ Token que inicia a *busca* da variável definida.
- **VARIABLE_PRINT** $\longrightarrow$ Token para a escrita da variável em terminal.

Em seguida, vamos explicar alguns tokens e as suas construções:

- **NUMBER**: esta função de token permite representar todas as combinações de números, sejam eles, **negetivos, positivos, inteiros ou até mesmo negativos**;

```
def t_NUMBER(t):
    r'-?\d+(\.\d+)?'
    t.value = float(t.value) if '.' in t.value else int(t.value)
    return t
```

- **STRING**: esta função de token permite representar todas as strings possíveis com as aspas;

```
def t_STRING(t):
    r'"([^"]|\s)*"'
    t.value = str(t.value)
    return t
```

- **STDOUT**: esta função permite representar o token de stdout, que aparece no formato de **."texto"**, para depois o texto seja transmitido como output;

```
def t_STDOUT(t):
    r'\.\s*"([^"]*)"\s*'
    t.value = t.value[3:-1]
    return t
```

- **COMMENT_LINE E COMMENT_BLOCK**: estas funções de tokens permitem representar os dois tipos de comentários da linguagem **Forth**;

```
def t_COMMENT_LINE(t):
    r'\\.*'
    return t

def t_COMMENT_BLOCK(t):
    r'\(.*?\)'
    return t
```

- **Expressões para as variáveis**: estas funções permitem representar a maneira de como podemos trabalhar com as variáveis;

```
    # Este token define uma variável, VARIABLE DATE

    def t_VARIABLE_DEFENITION(t):
        R'VARIABLE\s+[^\s]+'
        t.value = t.value[9:]
        return t

    # Este token atribui um valor à variável chamada, 12 DATE !

    def t_VARIABLE_ASSIGNMENT(t):
        r'\w+\s[A-Z]+\s!'
        parts = t.value.split()
        if parts[0].isdigit():
            t.value = (int(parts[0]), parts[1])
        elif parts[0].isfloat():
            t.value = (float(parts[0]), parts[1])
        else:
            t.value = (parts[0], parts[1])
        return t

        # Este token vai encontrar o valor da variável chamada, DATE @

    def t_VARIABLE_FETCH(t):
        r'[A-Z]+\s@'
        parts = t.value.split()
        if isinstance(parts[0], str):
            t.value = str(parts[0])
        elif isinstance(parts[0], int):
            t.value = int(parts[0])
        elif isinstance(parts[0], float):
            t.value = float(parts[0])
        return t

    # Este token vai dar print do valor da variável chamada, DATE ?

    def t_VARIABLE_PRINT(t):
        r'[A-Z]+\s\?'
        parts = t.value.split()
        t.value = parts[0]
        return t
```

- **DO**: esta função permite representar o token **DO**, ou seja um ciclo. Tem dois valores com este token, o **limite de ciclos a efetuar** e o **valor do ínicio do ciclo**;

```
def t_DO(t):
    r'[0-9]+\s[0-9]+\sDO'
    limit = t.value.split()[0]
    start = t.value.split()[1]
    t.value = (int(limit), int(start))
return t
```

- **CHAR**: esta função permite representar o token da função **CHAR**. Este token tem ainda um char na sua frente, para que se consiga determinar o seu valor da **tabela ASCII**;

```
def t_CHAR(t):
    r'CHAR\s+.'
    t.value = str(f'{t.value[5]}')
    return t
```

- **EMIT**: esta função permite representar o token da função **EMIT**. Este token tem ainda um valor numérico atrás, para que se esta consiga determinar o char que representa na **tabela ASCII**;

```
def t_EMIT(t):
    r'\d*\s*EMIT'
    parts = t.value.split()
    if parts[0].isdigit():
        t.value = int(parts[0])
    else:
        t.value = None
    return t
```

2.2 Gramática e Análise Sintática

O analisador sintático utiliza o módulo *ply.yacc* para definir a gramática da linguagem FORTH. A gramática é definida como um conjunto de regras de produção que descrevem como os *tokens* podem ser combinados para formar expressões e instruções válidas.

Na implementação da gramática do interpretador, optou-se pela seguinte estruturação da linguagem[1]:

- **Programa**: Um conjunto de comandos ou instruções;

```
def p_programa(p):
  '''programa : comandos'''
  p[0] = p[1]
```

- **Comandos**: Um conjunto de comandos;

```
def p_comandos(p):
  '''comandos : comando
              | comandos comando'''
  if len(p) == 2:
      p[0] = p[1]

  else:
      p[1][1].append(p[2])
      p[0] = p[1]
```

- **Comando**: Uma expressão, que geralmente implica a presença de valores ou algo a ser "empurrado"para a stack, a sua posterior avaliação e a geração de código máquina correspondente. Pode ser algo a avaliar, como:

```
def p_comando(p):
  '''comando : exp_aritmeticas
             | exp_relacionais
             | functions
             | values
             | creating_funcs
             | variable
             | flow_control'''

  p[0] = p[1]
```

[1]O código está apresentado em trechos para reduzir a saturação na leitura.

– Expressões aritméticas: Faz a avaliação de expressões aritméticas, manipulando a stack de acordo com o resultado. Gera as instruções correspondentes para a máquina virtual.

```
def p_exp_aritmeticas(p):
    '''exp_aritmeticas : comando comando PLUS
                       | comando comando MINUS
                       | comando comando TIMES
                       | comando comando DIVIDE
                       | comando comando MOD'''
```

– Expressões lógicas: Faz a avaliação de expressões booleanas, manipulando a stack de acordo com o resultado. Gera as instruções correspondentes para a máquina virtual.

```
def p_exp_relacionais(p):
    '''exp_relacionais : comando comando NOT
                       | comando comando INF
                       | comando comando SUP
                       | comando comando INFEQ
                       | comando comando SUPEQ'''
```

– Funções: Um conjunto de funções já existentes em FORTH, com diversos propósitos. Manipulam a stack, ou o Input/Output.

```
def p_functions(p):
    '''functions : stdout
                 | dot
                 | space
                 | dup
                 | comment
                 | drop
                 | swap
                 | rot
                 | over
                 | concat
                 | cr
                 | emit
                 | char
                 | key
                 | spaces
                 | 2dup'''
```

– Valores: Aceita ou um número ou uma string, que de seguida é adicionado à stack. Gera instruções *PUSH*.

```
def p_values(p):
    '''values : NUMBER
              | STRING'''
```

– Condicionais e expressões de controlo de fluxo: Representa um *if statement* ou um *loop*. No caso de loop, existe manipulação de stack, gerando instruções *PUSH* e *STORE* para a gestão dos saltos.

```
def p_flow_control(p):
    '''flow_control : if
                    | else
                    | then
                    | do
                    | loop'''
```

As funções definidas, para além de estabelecerem as regras gramaticais do parser, têm o papel fulcral de adicionar elementos à stack mediante as operações exigidas e gerar o código máquina correspondente, a utilizar na máquina virtual apresentada pela equipa docente.

A estruturação desenhada pelo grupo pretende captar e emular duas funcionalidades *vitais* do FORTH:

- **Extensibilidade**: Criando um *"dialeto"* baseado na simplicidade, o FORTH possibilita a composição recursiva de instruções menores para criar programas robustos e eficazes;
- **Uso de Stack**: No interpretador desenvolvido, também se utiliza uma *Data Stack* - representada por uma lista, cujo conteúdo é manipulado constantemente, seja por escrita direta, a função *pop()* das listas, ou a função *append()*.

3 Testes

3.1 Números e aritmética

O objetivo deste teste foi verificar que é reconhecida a diferença entre ints e floats e, ao mesmo tempo, testar o funcionamento das operações aritméticas com vários tipos de números. Aproveitamos ainda para verificar que o DOT reconhece o tipo do elemento da stack e escreve a devida instrução de WRITE.

Input:

```
-2 5 + 2.2 + -1.1 - .
```

Output:

```
PUSHI -2
PUSHI 5
ADD
PUSHF 2.2
FADD
PUSHF -1.1
FSUB
WRITEF
```

3.2 Strings

Neste teste, o objetivo foi verificar que as strings estão a ser introduzidas na stack corretamente, assim como algumas funções que atuam sobre strings, como o DOT, CONCAT e SPACE.

Input:

```
"MUNDO" "OLA" CONCAT "FORTH" . SPACE .
```

Output:

```
PUSHS "MUNDO"
PUSHS "OLA"
CONCAT
PUSHS "FORTH"
WRITES
PUSHS " "
WRITES
WRITES
```

3.3 Print de caracteres

Neste teste, o objetivo foi testar as operações de print, em específico o STDOUT, EMIT e CHAR. Verificamos já nos testes anteriores que o DOT funciona corretamente.

Input STDOUT:

```
." hello"
```

Output STDOUT:

```
PUSHS "hello"
WRITES
```

Input EMIT:

```
66 EMIT
65 EMIT
```

Output EMIT:

```
PUSHI 66
WRITECHR
PUSHI 65
WRITECHR
```

Input CHAR:

```
CHAR W .
```

Output CHAR:

```
PUSHI 87
WRITEI
```

3.4 Condicionais

O objetivo deste teste foi verificar o funcionamento dos condicionais. Neste ponto podemos verificar também o funcionamento das operações de comparação.

Input:

```
2 3 > IF "TRUE" . ELSE "FALSE" . THEN
```

Output:

```
PUSHI 2
PUSHI 3
SUP
JZ else0
PUSHS "TRUE"
WRITES
JUMP endif0
else0:
PUSHS "FALSE"
WRITES
JUMP endif0
endif0:
```

3.5 Ciclos

Para testar os loops, usamos um pequeno programa que irá dar print a uma string 10 vezes.

Input:

```
10 0 DO "HELLO" . LOOP
```

Output:

```
PUSHG 0
PUSHI 10
STOREG 0
PUSHG 1
PUSHI 0
STOREG 1
WHILE0:
PUSHG 0
PUSHG 1
SUP
jz ENDWHILE0
PUSHS "HELLO"
WRITES
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP WHILE0
ENDWHILE0:
```

3.6 Variáveis

Neste teste, o objetivo foi testar a declaração de variáveis, assim como a atribuição de valores às mesmas e o *fetch* dos seus valores.

Input:

```
VARIABLE DATE
12 DATE !
VARIABLE NUMBER
56 NUMBER !
DATE ?
NUMBER ?
```

Output:

```
PUSHG 0
PUSHI 12
STOREG 0
PUSHG 1
PUSHI 56
STOREG 1
PUSHG 0
WRITEI
PUSHG 1
WRITEI
```

3.7 Funções

Para testar as funções, declaramos várias funções que são todas chamadas em outra função. Esta última função é chamada para executar o conteúdo das outras.

Input:

```
: FUNC1 12 5 + . 34 14 - . ;
: FUNC2 16 2 * . 50 10 % . ;
: FUNC3 100 5 / . ;
: FUNC4 FUNC1 CR FUNC2 CR FUNC3 ;
FUNC4
```

Output:

```
PUSHA FUNC4
CALL

FUNC4:
PUSHA FUNC1
CALL

FUNC1:
PUSHI 12
PUSHI 5
ADD
WRITEI
PUSHI 34
PUSHI 14
SUB
WRITEI
WRITELN
PUSHA FUNC2
CALL

FUNC2:
PUSHI 16
PUSHI 2
MUL
WRITEI
PUSHI 50
PUSHI 10
MOD
WRITEI
WRITELN
PUSHA FUNC3
```

```
CALL

FUNC3:
PUSHI 100
PUSHI 5
FDIV
WRITEF
```

3.8 Comentários

O seguinte teste teve como objetivo verificar que os comentários são devidamente reconhecidos.

Input:

```
\ TESTE TESTE 1 2 3
( 150=30*5 )
```

Output:

```
// \ TESTE TESTE 1 2 3
// ( 150=30*5 )
```

4 Conclusão

Através do desenvolvimento deste compilador, conseguimos explorar a linguagem Forth e a partir dela perceber alguns detalhes que são tidos em conta na engenharia das linguagens de programação. Por exemplo, no que toca a sintaxe, é importante que esta esteja bem estruturada para não haver ambiguidade.

Conseguimos ainda aprofundar o nosso conhecimento sobre expressões regulares na criação dos tokens utilizados pelo compilador (com recursos ao Lex), e consequentemente, sobre a definição de gramáticas (com recurso a Yacc).

Este projeto exigiu ainda do grupo bastante criatividade de elaboração de testes. Desta forma foram elaborados testes com vários níveis de "dificuldade"que avaliaram o funcionamento e robustez da solução implementada.