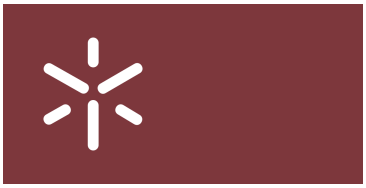

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Compilador Forth

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2023/2024

Os Intrépidos Engenheiros de Compiladores da Noite Sob o Manto Estrelado da Semântica Eterna

José Correia	**Pedro Lopes**	**José Oliveira**
a100610	a100759	a100547

\PL

Índice

1. Introdução

No âmbito da unidade curricular de Processamento de Linguagens, foi-nos proposta a conceção e desenvolvimento de um compilador de Forth que gera pseudo-código máquina para uma $\mathrm{máquina\ virtual}$[1] disponibilizada pela equipa docente.

O desenvolvimento deste compilador foi baseado em gramáticas independentes de contexto (GIC). Nesta implementação recorremos ao uso da ferramenta PLY do Python para construção dos analisadores léxicos e sintáticos (Lex e Yacc).

2. Descrição do problema

O objetivo do nosso compilador é traduzir a linguagem Forth para a liguagem da máquina virtual. Para alcançarmos este objetivo de forma eficaz e completa, é crucial explorarmos a fundo ambos os ambientes para percebermos como é que estes funcionam, principalmente a nível sintático e semântico.

2.1. Forth

A linguagem de programação Forth é uma linguagem de baixo nível, baseada em stack. É também uma linguagem extremamente flexível e eficiente.

Porém a característica mais discrepante desta linguagem é a sua notação, pois esta utiliza uma notação pós-fixa (RPN - Reverse Polish Notation), o que significa que, os operadores são colocados após os seus operandos.

```
2 10 /
: maior2 ( a b - maior ) 2dup > if drop else nip then ;
```

2.2. Máquina Virtual

A máquina virtual utiliza pseudo-código Assembly, e tal como Forth também é uma máquina de stack.

Esta compreensão é o que nos vai permitir estabelecer com precisão os componentes da nossa gramática **G**.

$$G = (T, N, S, P)$$

T - Alfabeto da linguagem Forth (símbolos terminais/*tokens*)

N - Símbolos não terminais

S - Símbolo Inicial

P - Produções

De forma a definir estes componentes, o nosso alfabeto é defindo por uma lista de *tokens* no Lexer. Enquanto que o resto será definido no Yacc.

A gramática dá origem a uma árvore de derivação de um dado X (raiz da árvore), onde os nodos são símbolos terminais ou símbolos não terminais. De uma forma genérica podemos dividir o tipo de reconhecedores, `TOP-DOWN` ou `BOTTOM-UP`. Após analisarmos ambas as linguagens, percebemos que a melhor abordagem seria `BOTTOM-UP` (de baixo para cima, da direita para a esquerda) devido a Forth ser uma linguagem pós-fixa.

3. Arquitetura do compilador

Como referido anteriormente, o nosso programa irá converter código Forth para pseudo-código máquina, que será depois executado numa máquina de stack virtual, ou seja vamos dar "parse" de código Forth e traduzir para o código da máquina virtual. A arquitetura do nosso compilador é constituída por 3 módulos principais: Analisador Léxico, Analisador Sintático e Translator.

O nosso programa passa por várias etapas desde que recebe o Input (código Forth) até obter o Output (código máquina virtual). Podemos generalizar da seguinte forma:

```
Input -> (Analisador Léxico + Analisador Sintático) <-> Translator -> Output
```

O Analisador Léxico utiliza o PLY/LEX, responsável por definir regras de expressões regulares que especificam padrões para identificar os *tokens* no código fonte. Os *tokens* incluem operadores, números, identificadores, palavras-chave, entre outros elementos da linguagem.

O Analisador Sintático utiliza o PLY/YACC, define as regras da gramática e analisa a estrutura sintática de acordo com estas. Este processo envolve a construção de uma árvore sintática que representa a estrutura hierárquica do código fonte.

O Translator é um componente criado por nós, que vai armazenando o código traduzido ao longo da análise sintática.

Primeiramente inicializamos o Lexer e o Yacc, para que o nosso *parser* "herde" todos os *tokens* definidos no Lex. Seguidamente passamos o código fonte para o nosso *parser*. No fim, o Translator possui o código máquina resultante.

4. Gramática

4.1. Gramática desenvolvida

De acordo com os requisitos pedidos pelo enunciado deste TP, depois de muitas mudanças, de forma a nossa gramática não conter quaisquer tipos de conflitos, sejam estes do tipo **shift/reduce** ou **reduce/reduce**, chegando então à gramática representada abaixo:

```
Line : Line Cmd
     | Line Function
     |

Expressao : Expressao Cmd
          |

Cmd : COMMENT1
    | COMMENT2
    | NAME SET
    | NAME GET
    | SWAP
    | VARIABLE NAME
    | NAME
    | KEY
    | CR
    | STRING
    | EMIT
    | DOT
    | CHR CHAR
    | CHR MATH_OPERATOR
    | DUP
    | PRINTSTRING
    | NUMBER
    | CHAR
    | MATH_OPERATOR
    | Loop
    | WHILE
    | IF
    | ELSE
    | THEN

Begin : BEGIN

Until : UNTIL

Do : DO

Repeat : REPEAT

LooP : LOOP

Colon : COLON
```

```
Loop : Begin Expressao Until
     | Begin Expressao Repeat
     | Do Expressao LooP

Function : Colon NAME Expressao SEMICOLON
```

Outro motivo pelo qual a gramática é demonstrada desta maneira aqui é que, graças à maneira como o *Translator* é relacionado com o *Analisador Sintático* e os seus métodos, a gramática acabou por ficar com uma forma bastante dividida, algo que é possível verificar mais abaixo.

5. Implementação

Na nossa implementação, seguimos uma abordagem baseada em classes, com o objetivo de estruturar e modularizar o projeto. O Lexer, o Parser e o Translator do nosso programa estão definidos em classes.

A ferramenta YACC é um gerador de analisadores sintáticos LALR(1) (Look-Ahead Left-to-Right, Rightmost derivation) em Python. O YACC é notável pela sua capacidade de gerar analisadores sintáticos (parsers), a partir de uma especificação formal de gramática

5.1. Main

O ficheiro main.py do projeto tem algumas características a notar. Para o correr, podemos executar o seguinte comando:

```
> python main.py [nome do ficheiro de input]
```

Este programa é responsável pelo funcionamento de todos os programas, fazendo essencialmente o único parse feito ao código Forth no ficheiro enviado como argumento, na sua execução, da seguinte forma:

```
def main(args):
    i = 0
    lexer = Lexer()
    translator = Translator()
    parser = Parser(lexer, translator)
    if len(args) > 1:
        with open(args[1], 'r') as file: # Leitura do ficheiro de Input
            data = file.readlines()
        data = treat_data(data)
        for line in data:
            result = parser.parse(line)
            i+=1
            if result:
                print(result)

    translator.code_to_file()
    translator_code_aux = translator.code
```

Porém, anteriormente a isto, de forma a conseguirmos dar o parse correto de todos os casos, especialmente ficheiros com múltiplas linhas ou comentários, decidimos implementar uma função responsável por este pré-tratamento do ficheiro de input, que vai juntar, se possível, todas as linhas do ficheiro numa, possibilitando um parse sem preocupações com quebras de linha:

```
def treat_data(data):
    treated_data = []
    buffer = ''
    for line in data:
        if '\\' in line:
            if buffer:
                treated_data.append(buffer)
```

```
                buffer = ''
        buffer += line.strip() + ' '
        if ';' in line or '\\' in line:
            treated_data.append(buffer)
            buffer = ''
    if buffer:
        treated_data.append(buffer)
    return treated_data
```

Após isto, caso este código contenha Loops ou Funções, o código destas vai ser adicionado também ao ficheiro de Output, através da limpeza e armazenamento do código das estruturas no código do *Translator*, ficando então composto o ficheiro de Output do programa, com o código correspondente da Máquina Virtual:

```
    # Parte das funções
    if translator.functions:
        i = 0
        while i < len(translator.functions):
            function_name = translator.function_names[i]
            data = translator.functions[i]
            if ("pusha " + function_name) in translator_code_aux:
                translator.code = []
                translator.code.append(f"\n{function_name}:")
                for line in data:
                    translator.code.append(line)

                translator.function_to_file()
            i += 1

    # Parte dos loops
    if translator.loops:
        i = 0
        while i < len(translator.loops):
            data = translator.loops[i]
            translator.code = []
            translator.code.append("\nloop" + str(i+1) + ":")
            for line in data:
                translator.code.append(line)
            translator.loop_to_file(i+1)
            i += 1

    filename = "../outputs/output.txt"
    with open(filename, 'a') as file:
        file.write(f"stop\n")
```

5.2. Lexer

Quanto ao Lexer, ou seja, o analisador léxico, definimos os tokens necessários para o analisador sintático e, também, de forma a reconhecer as variáveis no código Forth, definimos também as suas expressões regulares para detecção,tendo assim este programa a seguinte estrutura, resumidamente:

```
    tokens = [
        'NUMBER',
        'STRING',
        'COLON',
        'SEMICOLON',
        'DOT',
        'EMIT',
        'KEY',
        'CHAR',
        'CHR',
        'CR',
        'VARIABLE',
        'GET',
        'SET',
        'NAME',
        'IF',
        'ELSE',
        'THEN',
        'COMMENT1',
        'COMMENT2',
        'MATH_OPERATOR',
        'PRINTSTRING',
        'DUP',
        'DO',
        'LOOP',
        'SWAP',
        'BEGIN',
        'UNTIL',
        'WHILE',
        'REPEAT'
    ]

    def t_PRINTSTRING(self, t):
        r'\.\"\s[^"]+\"'
        return t

    t_COLON = r':'
    t_SEMICOLON = r';'
    t_DOT = r'\.'

    def t_BEGIN(self, t):
        r"[Bb][Ee][Gg][Ii][Nn]"
        return t

    def t_UNTIL(self, t):
        r"[Uu][Nn][Tt][Ii][Ll]"
        return t

...
```

5.3. Yacc

Chegamos então ao *Analisador Sintático*, que é o programa que vai relacionar a gramática desenvolvida e demonstrada acima com o *Translator*, tornando assim possível o parse de ficheiros, caso a gramática esteja corretamente desenvolvida para tal. Antes de continuarmos, o Analisador Sintático, de forma a funcionar corretamente, contém estes elementos essenciais na sua classe:

```
      self.lexer = lexer #Lexer
      self.tokens = self.lexer.tokens #TOKENS
      self.parser = yacc.yacc(module=self, start='Line') #Declaração do Parser
      self.translator = translator #Translator associado ao Parser
      self.func_count = 0 # Contadores de Funções
      self.loop_count = 0 # Contadores de Loops
      self.in_func = False # Bool indicativo de presença numa função
      self.in_loop = False # Bool indicativo de presença num loop
```

Dito isto, passemos então à explicação e demonstração da implementação deste programa.

5.3.1. Tratamento de Variáveis

Neste excerto abaixo está representada uma porção do código responsável pelo tratamento da leitura de variáveis, juntamente com algumas funções que são chamadas do Translator:

```
    def p_string(self, p):
        '''Cmd : STRING'''
        p[0] = p[1]
        self.translator.push(p[1], self.in_func, self.in_loop)

    def p_print2(self, p):
        '''Cmd : EMIT
        '''
        p[0] = p[1]
        self.translator.emit(self.in_func, self.in_loop)

    def p_print(self, p):
        '''Cmd : DOT
        '''
        p[0] = p[1]
        self.translator.print(self.in_func, self.in_loop)

    def p_char(self, p): #Possivelmente vai sair
        '''Cmd : CHR CHAR
               | CHR MATH_OPERATOR
        '''
        p[0] = p[1]
        self.translator.char(p[2])

    def p_dup(self,p):
        '''Cmd : DUP
        '''
        p[0] = p[1]
        self.translator.dup(self.in_func, self.in_loop)
...
```

Estando o programa estruturado desta forma, mal a leitura referida é completa, para cada função p, é chamada uma função, caso necessária, do Translator, que irá utilizar os valores booleanos e a informação de leitura para a tradução do código. Isto será explicado melhor no capítulo do Translator.

5.3.2. Tratamento de Loops

De forma a termos margem de manobra no início, durante e no fim dos loops, decidimos então guardar alguns tokens em tokens não-terminais, podendo assim chamar funções nestas alturas previamente mencionadas no loop, sendo assim, o tratamento dos Loops ficou da seguinte forma:

```
    def p_begin(self, p):
        '''Begin : BEGIN'''
        p[0] = p[1]
        self.in_loop = True
        self.translator.init_loop() # Entrada de Um Loop

    def p_until(self, p):
        '''Until : UNTIL'''
        p[0] = p[1]
        self.translator.until()

    def p_do(self, p):
        '''Do : DO'''
        p[0] = p[1]
        self.in_loop = True
        self.translator.init_loop() # Entrada de Um Loop

    def p_while(self, p):
        '''Cmd : WHILE'''
        p[0] = p[1]
        self.translator._while()

    def p_Repeat(self, p):
        '''Repeat : REPEAT'''
        p[0] = p[1]
        self.translator.repeat()

    def p_endLoop(self, p):
        '''LooP : LOOP
        '''
        self.translator.until()
        return p

    def p_loop1(self, p):
        '''
        Loop : Begin Expressao Until
        '''
        self.in_loop = False # Saida de um Loop

        return p

    def p_loop2(self, p):
        '''
        Loop : Begin Expressao Repeat
        '''
        self.in_loop = False # Saida de um Loop

        return p

    def p_loop3(self, p):
        '''
        Loop : Do Expressao LooP
        '''
        self.in_loop = False # Saida de um Loop

        return p
```

Como se pode ver, como decidimos utilizar uma estratégia de saber quando estamos num Loop ou fora, de forma a saber onde guardar o código traduzido, decidimos a implementação de 2 valores booleanos *in_loop* e *in_func*, o que permite ao parser saber quando se situa a ler dentro de uma função ou um loop, permitindo assim a distinção de scopes.

5.3.3. Tratamento de Funções

De uma forma bastante semelhante aos Loops, as funções tiveram a seguinte implementação, seguindo o mesmo raciocínio:

```
    def p_colon(self, p):
        '''Colon : COLON'''
        p[0] = p[1]
        self.in_func = True
        self.translator.init_func() # Entrada de uma função

    def p_function(self, p):
        '''
        Function : Colon NAME Expressao SEMICOLON
        '''
        self.in_func = False # Saída de uma função

        if p[2] not in self.translator.function_names:
            self.translator.function_names.append(p[2]) # Tratamento dos nomes das
funções
        else:
            print(f"Function {p[2]} already exists")
            self.translator.functions[self.func_count] = ""
            return None
        return p
```

O que diferencia as funções dos loops é o tratamento de nomes necessário a fazer. Como uma função tem de ter um nome no momento da sua definição, de forma a ser possível a sua tradução e escrita no ficheiro de output, nós guardamos estes nomes numa lista no *Translator*, que vai ser utilizada na *Main* para a escrita correta das funções no output.

5.4. Translator

De entre todos os programas, graças aos tratamentos que este têm que fazer aos elementos lidos pelo Analisador Sintático, este acaba por ser o programa mais pesado, sendo ele responsável pela correta tradução do código Forth para o código da Virtual Machine pedido.

Sendo assim, a classe do Translator, para o seu correto funcionamento, vai conter os seguintes atributos na sua definição:

```
class Translator:
    def __init__(self):
        self.stack = [] # Stack
        self.variables = {} # Variáveis criadas
        self.functions = [] # Funções criadas
        self.loops = [] # Loops criados
        self.function_names = [] # Nomes de funções
        self.code = [] # Código principal
        self.code.append("start")
        self.code.append("") # Arranque do código
        self.var_counter = 0
        self.ifcounter = 0
```

```
        self.func_count = -1
        self.loop_count = -1 # Contadores necessários
```

Com isto, de forma ao *Translator* estar em união com o *Analisador Sintático*, foram então criados os métodos de tradução do parsing feito pelo Analisador, chegando então a métodos como estes abaixo:

```
    def push(self, value, in_func : bool, in_loop : bool):
        self.stack.append(value)
        if type(value) == str:
            if in_func and not in_loop:
                self.functions[self.func_count].append(f"pushs \"{value}\"")
            elif (in_loop and not in_func) or (in_func and in_loop):
                self.loops[self.loop_count].append(f"pushs \"{value}\"")
            elif not in_func and not in_loop:
                self.code.append(f"pushs \"{value}\"")
        if type(value) == int:
            if in_func and not in_loop:
                self.functions[self.func_count].append(f"pushi {value}")
            elif (in_loop and not in_func) or (in_func and in_loop):
                self.loops[self.loop_count].append(f"pushi {value}")
            elif not in_func and not in_loop:
                self.code.append(f"pushi {value}")
        if type(value) == float:
            if in_func and not in_loop:
                self.functions[self.func_count].append(f"pushf {value}")
            elif (in_loop and not in_func) or (in_func and in_loop):
                self.loops[self.loop_count].append(f"pushf {value}")
            elif not in_func and not in_loop:
                self.code.append(f"pushf {value}")
        return value

    def pop(self, in_func : bool, in_loop : bool):
        if in_func and not in_loop:
            self.functions[self.func_count].append("pop")
        elif (in_loop and not in_func) or (in_func and in_loop):
            self.loops[self.loop_count].append("pop")
        elif not in_func and not in_loop:
            self.code.append("pop")

    def dup(self, in_func : bool, in_loop : bool):
        value = self.stack[-1]
        self.stack.append(value)
        if in_func and not in_loop:
            self.functions[self.func_count].append("dup 1")
        elif (in_loop and not in_func) or (in_func and in_loop):
            self.loops[self.loop_count].append("dup 1")
        elif not in_func and not in_loop:
            self.code.append("dup 1")
        return value
...
```

Como se pode ver por estas funções, graças ao tratamento dos 2 valores booleanos no parser mencionados anteriormente, conseguimos assim tratar das diferentes scopes no código do Forth inserido como Input no ficheiro de Input.

Fazendo isto, criamos então, ao contrário do que possa parecer, 4 possíveis scopes diferentes:

- **Dentro de uma função**
- **Dentro de um Loop**
- **Dentro de um Loop, dentro de uma função**
- **Fora de qualquer Loop ou função**

Isto é feito através das sequências de ifs, que se conseguem ver dentro de várias das funções de tradução, permitindo assim o armazenamento do código nos seus sítios devidos, ajudando então na tradução e escrita que é feita na *Main*.

Outro destaque neste programa, além deste tratamento mencionado, é a utilização de uma "Stack", de forma a que o programa possa ter uma boa ideia do que se está a passar no código, dando assim assistência na tradução.

Por outro lado, apesar dos benefícios que esta stack nos dá, também nos deu bastantes dificuldades em alguns casos de funções, onde, na leitura do seu código respetivo, graças a ser uma função, que espera uma variável/número como argumento, o tradutor não conseguia chegar até ao fim da tradução do código, graças a estar à espera de um valor da stack, que só é fornecido na sua chamada mais abaixo.

Apesar disto, sem esta stack, não conseguiríamos tratar de muitas das traduções mencionadas atrás, tendo assim sendo um método de tradução um pouco diferente do tradicional, como se trata de uma tradução, à primeira vista, nem deveria ser necessária uma stack de simulação.

6. Exemplos de utilização

Finalmente, com a nossa implementação explicada, temos aqui então alguns casos de exemplo, representativos do que nos foi pedido pelo enunciado para implementar:

6.1.1. Aritmética

Input:

```
30 5 + 7 / .
```

Output

```
start

pushi 30
pushi 5
add
pushi 7
div
writei
stop
```

6.1.2. Strings em funções

Input:

```
: TOFU ." Yummy bean curd!" ;
TOFU
: TOFU2 ." YUMMYYYYY bean curd !" ;
TOFU2
```

Output:

```
start

pusha TOFU
call
pusha TOFU2
call

TOFU:
pushs "Yummy bean curd!"
writes
return

TOFU2:
pushs "YUMMYYYYY bean curd !"
writes
return
stop
```

6.1.3. Condicionais

Input:

```
69                \ Empilha 69 na Stack
420 <             \ Compara o topo da pilha com 420
IF                \ Se for verdadeiro (topo da pilha menor que 420)
    ." Está Certo"  \ Imprime "Está Certo"
ELSE              \ Senão
    ." Está Errado" \ Imprime "Está Errado"
THEN              \ Fim da estrutura condicional
```

Output:

```
start

pushi 69
pushi 420
inf
jz else1
pushs "Está Certo"
writes
jump then1
else1:
pushs "Está Errado"
writes
then1:
stop
```

6.1.4. Variáveis

Input:

```
VARIABLE ASa
ASa @
10 ASa !
.
```

Output:

```
pushi 0
start

pushi 0
storeg 0
pushg 0
pushi 10
storeg 0
writei
stop
```

6.1.5. Loops

6.1.5.1. Loop Repeat

Input:

```
1
BEGIN
    DUP .
    1 +
```

```
    DUP 10 =
WHILE
    1 .
REPEAT
```

Output:

```
start

pushi 1

loop1:
dup 1
writei
pushi 1
add
dup 1
pushi 10
equal
jz endloop1
pushi 1
writei
jump loop1
endloop1:
stop
```

6.1.5.2. Loop Until

Input:

```
1
BEGIN
    DUP .
    1 +
    DUP 10 =
UNTIL
```

Output:

```
start

pushi 1

loop1:
dup 1
writei
pushi 1
add
dup 1
pushi 10
equal
jz loop1
stop
```

6.1.5.3. Loop DO com Função

Infelizmente, este tipo de Loops não foi implementado totalmente, devido a problemas de tempo e outros problemas que tivemos ao longo do projeto. Ainda assim, ficam aqui os exemplos de Input e Output, tal e qual como nos exemplos anteriores:

Input:

```
: sum ( n -- sum )
    0 swap 1 do
    i +
    loop ;
5 sum .
```

Output:

```
start

pushi 5
pusha sum
call
writei

sum:
pushi 0
swap
pushi 1
return

loop1:
add
jz loop1
stop
```

7. Problemas encontrados ao longo do Projeto

- Sendo este já mencionado anteriormente, não conseguimos tratar totalmente de um dos tipos de loops, tendo este ficado com funcionalidades por tratar.
- Acabamos por perder bastante tempo também com a restauração e reestruturação da nossa gramática anterior, visto que esta estava cheia de conflitos, o que permitiu, pelo lado positivo, uma melhor sabedoria sobre a forma correta da criação de gramáticas e as melhores formas de as corrigir.

8. Conclusão

Este projeto permitiu-nos colocar os conceitos ensinados na UC em prática numa forma muito interessante, o que nos deu uma ideia um bocado mais clara de como os compiladores funcionam e todo o estudo que está por trás.

Aquilo que se mostrou mais desafiante neste projeto foi deixar o nosso pensamento tradicional (notação infixa) de lado e analisar os problemas com a lógica de uma máquina de stack. Isto permitiu-nos relembrar como é que as linguagens de máquina realmente funcionam, principalmente no que toca a variáveis de ambiente e jumps.

De uma forma geral, apesar de algumas dificuldades no caminho, consideramos que os objetivos propostos no enunciado foram alcançados com sucesso, tendo também noção que existe espaço para melhoria na nossa implementação.

Referências

[1] Máquina Virtual para testar o código máquina: EWVM
Biblioteca Python PLY, uma ferramenta para a construção de analisadores léxicos e sintáticos (Lex e Yacc).
Manual para iniciantes na linguagem Forth: Starting Forth
IDE online para teste e desenvolvimento de código Forth: Jdoodle

Column 1	Column 2	Column 3	Column 4	Column 5
Column 1	Column 2	Column 3	Column 4	Column 5
Column 1	Column 2	Column 3	Column 4	Column 5
Column 1	Column 2	Column 3	Column 4	Column 5