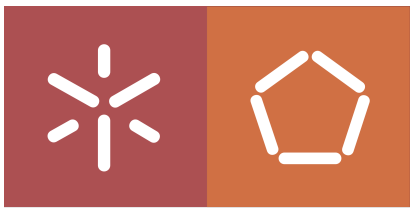
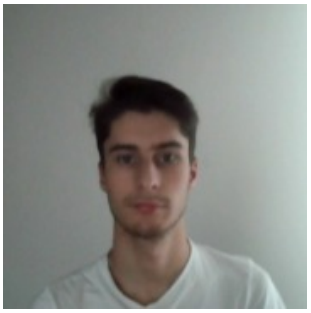


Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Relatório Trabalho Prático
LEI - 3º Ano - 2º Semestre

Braga, 19 de fevereiro de 2025


Lucas Oliveira A98695


Mike Pinto A89292


Rafael Gomes A96208

Conteúdo

Capítulo 1

Introdução

Este relatório tem como propósito descrever detalhar o trabalho prático desenvolvido no âmbito da Unidade Curricular de Processamento de Linguagens do segundo semestre do terceiro ano do curso de Engenharia Informática da Universidade do Minho.

Neste contexto, o trabalho consistiu na implementação de um compilador *Forth*, com o objetivo principal de criar um compilador capaz de gerar código para uma máquina virtual, fornecida pela equipa docente.

A intenção do trabalho é aprimorar as competências em engenharia de linguagens e programação generativa, através da escrita de gramáticas e do desenvolvimento de processadores de linguagens. Recorremos a ferramentas como Yacc e Lex para alcançar tais objetivos.

Capítulo 2

Descrição do Problema

O propósito deste problema é criar um compilador para a linguagem *Forth*, uma linguagem de programação de baixo nível que opera com uma pilha, com o intuito de produzir código para uma máquina virtual. Sendo a linguagem *Forth* reconhecida pela sua simplicidade, eficiência e flexibilidade.

O compilador irá necessitar de compreender instruções escritas em *Forth*, em que irão consistir em uma lista de palavras separadas por espaços. Sendo as operações *Forth* executadas numa *stack*, onde os valores são empilhados e desempilhados para realizar as operações. Além disso a linguagem utiliza notação **Pós-Fix(RPN)**, onde os operadores são colocados após os operandos. O compilador também deverá suportar desde expressões aritméticas simples até funcionalidades mais avançadas, como a criação de funções, estruturas condicionais e ciclos. A implementação deverá ser realizada utilizando os recursos de *YACC* e *LEX*.

Em suma, o objetivo é criar um compilador eficiente e flexível para a linguagem Forth, capaz de gerar código para uma máquina virtual específica.

Capítulo 3

Decisões/Desenho da Solução

Após uma análise detalhada do problema proposto, optou-se pela decisão de elaborar uma estrutura lógica que servirá como base para eliminar quaisquer obstáculos futuros. O objetivo desta etapa foi organizar a estrutura de forma a estabelecer uma visão clara, garantindo assim o sucesso na implementação do *Parser* e do *Lexer*. Esta abordagem segue os princípios lecionados nas aulas, visando aumentar a eficácia na realização do projeto.
A solução obtida pode ser observada na figura 3.1.

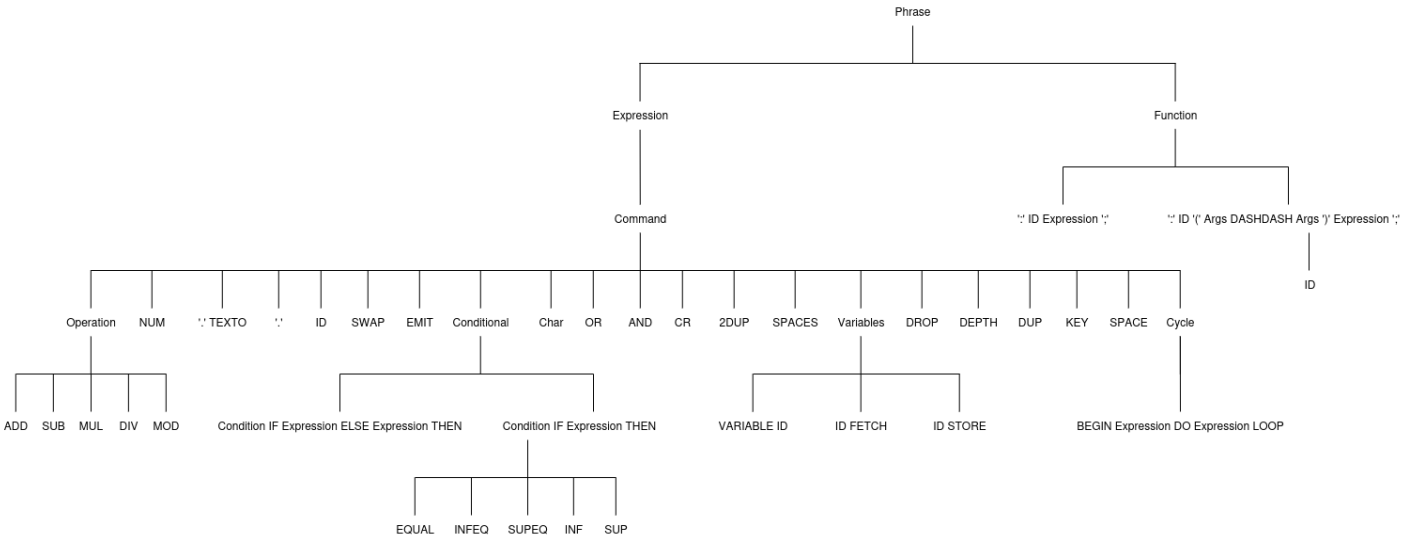


Figura 3.1: Estrutura Lógica da Solução.

Capítulo 4

Implementação

4.1 Análise Léxica

Inicialmente definiu-se os *Tokens* necessários da linguagem, sendo esses:

- **2DUP**: Comando que representa a duplicação dos dois valores no topo da *stack*.
- **NUM**: Tipo de valor numérico.
- **ADD**: Operação da Soma.
- **DASHDASH**: Representação dos caracteres "- -".
- **SUB**: Operação da Subtração.
- **MUL**: Operação da Multiplicação.
- **DIV**: Operação da Divisão.
- **MOD**: Operação do Resto da Divisão.
- **SWAP**: Comando que representa a troca de posições entre dois valores no topo da *stack*.
- **EMIT**: Comando que representa a escrita de um caracter com base no valor numérico *ASCII* do topo da *stack*.
- **KEY**: Comando que representa a leitura de um *input* do teclado e guarda o seu valor numa *String Heap* e empilha o seu endereço na *stack*.
- **CHAR**: Comando que representa a conversão no valor *ASCII* da string e empilha o seu valor na *stack*.
- **OR**: Comando que representa a disjunção entre dois valores no topo da *stack* e empilha o seu resultado.
- **AND**: Comando que representa a conjunção entre dois valores no topo da *stack* e empilha o seu resultado.
- **CR**: Comando que representa a escrita de um "`\n`".
- **SPACES**: Comando que representa a escrita de um certo número de espaços com base no valor numérico no topo da *stack*.

- **SPACE**: Comando que representa a escrita de um único espaço.
- **DROP**: Comando que representa a remoção do valor no topo da *stack*.
- **DEPTH**: Comando que representa o cálculo do tamanho da *stack* no momento.
- **DUP**: Comando que representa a duplicação do valor presente no topo da *stack*.
- **CONDITION**: Tipo de representação de uma condição.
- **IF**: Tipo de representação de um "*if*" em *Forth*.
- **THEN**: Tipo de representação de um "*then*" em *Forth*.
- **ELSE**: Tipo de representação de um "*else*" em *Forth*.
- **VARIABLE**: Representa o início da definição de uma variável.
- **STORE**: Representa o comando "!" do *Forth* que indica o armazenamento do valor do topo da *stack* numa variável.
- **FETCH**: Representa o comando "@" do *Forth* que indica a busca de um valor numa variável e empilha na *stack*.
- **LOOP**: Tipo de representação de um "*loop*" em *Forth*.
- **DO**: Tipo de representação de um "*do*" em *Forth*.
- **BEGIN**: Definido para indicar o início de um ciclo.
- **TEXTO**: Representa um valor delimitado entre aspas.
- **ID**: Representa o nome de uma variável, função, comando.

Para além dos *Tokens*, foram definidos também *literals* que fazem parte da linguagem *Forth*:

- **'(' , ')'**: Representa a indicação da existência de *inputs* e *outputs*.
- **':'**: Representa o início da definição de uma função.
- **';'**: Representa o fim da definição de uma função.
- **'.'**: Representa a indicação de escrita do valor tanto numérico como uma *String*, sendo esse valor o do topo da *stack*.

4.2 Gramática

Após a definição dos *tokens*, seguiu-se para a definição das regras da gramática.

```
Phrase : Phrase Expression
       | Phrase Function
       | Expression
       | Function
Expression : Command
           | Expression Command
Function : ':' ID '(' Args DASHDASH Args ')' Expression ';'
         | ':' ID Expression ';'
Args : ID
     | Args ID
     |
Command : NUM
        | '.' TEXTO
        | '.'
        | ID
        | SWAP
        | EMIT
        | KEY
        | Char
        | OR
        | AND
        | CR
        | SPACE
        | 2DUP
        | SPACES
        | DROP
        | DEPTH
        | DUP
        | Operation
        | Conditional
        | Variables
        | Cycle
Operation : ADD
          | SUB
          | MUL
          | DIV
          | MOD
Conditional :
            | Condition IF Expression ELSE Expression THEN
            | Condition IF Expression THEN
Condition : CONDITION
Variables : VARIABLE ID
          | ID FETCH
          | ID STORE
Cycle : BEGIN Expression DO Expression LOOP
```

Listing 1: Gramática

No que diz respeito à representação do *Function*, foi estabelecido que uma função genérica deve obrigatoriamente indicar se possui ou não argumentos ou saídas, com a exceção das funções que lidam exclusivamente com *Strings*, as quais estão isentas dessa obrigatoriedade.

```
: function_name ( a b -- c ) function_body ;

: function_name ( -- ) function_body ;

: function_name . "Only Strings" ;
```

Listing 2: *Function*

Relativamente ao *Cycle*, foi determinado que, para iniciar um ciclo, é necessário utilizar a palavra-chave *BEGIN* como ponto de partida.

```
BEGIN cycle_expression DO cycle_expression LOOP
```

Listing 3: *Cycle*

Além disso, é importante destacar que o programa inclui um dicionário que contem todas as funções definidas, bem como um dicionário que armazena todas as variáveis definidas durante a execução do programa. Adicionalmente, foi implementada uma *stack* auxiliar para distinguir as instruções *'WRITEI'* e *'WRITES'* quando o *literal '.'* é encontrado. Por fim, a *stack* principal onde é utilizada para armazenar todos os comandos a serem executados na máquina virtual.

```
functions = {
    'args'      # Armazena o número de argumentos da função
    'nOutput'   # Armazena o número de outputs da função
    'body'      # Armazena o corpo da função como uma lista de comandos
    'result'    # Armazena a função para quando for usada
    'index'     # Atribui um index à função
    'activated' # Indica se a função foi ativada ou não
}

variables = {
    'index'     # Atribui um index à variàvel
}

dotAuxiliarStack = []

stack = []
```

Listing 4: Estruturas de Dados

Capítulo 5

Testes/Resultados

5.1 Operações Aritméticas

5.1.1 Adição

Código 5.1: Terminal

```
2 3 + .
```

Código 5.2: Máquina Virtual

```
START
PUSHI 2
PUSHI 3
ADD
WRITEI
STOP
```

5.1.2 Subtração

Código 5.3: Terminal

```
30 5 - .
```

Código 5.4: Máquina Virtual

```
START
PUSHI 30
PUSHI 5
SUB
WRITEI
STOP
```

5.1.3 Multiplicação

Código 5.5: Terminal

```
30 5 * .
```

Código 5.6: Máquina Virtual

```
START
PUSHI 30
PUSHI 5
MUL
WRITEI
STOP
```

5.1.4 Divisão

Código 5.7: Terminal

```
30 5 / .
```

Código 5.8: Máquina Virtual

```
START
PUSHI 30
PUSHI 5
DIV
WRITEI
STOP
```

5.1.5 Resto da Divisão

Código 5.9: Terminal

```
10 20 % .
```

Código 5.10: Máquina Virtual

```
START
PUSHI 10
PUSHI 20
MOD
WRITEI
STOP
```

5.2 Funções

Código 5.11: Terminal

```
: AVERAGE ( a b -- avg ) + 2/ ;
10 20 AVERAGE .
```

Código 5.12: Máquina Virtual

```
PUSHI 0
START
PUSHI 10
PUSHI 20
PUSHA AVERAGE
CALL
POP 2
PUSHG 0
WRITEI
STOP
AVERAGE:
PUSHFP
LOAD -2
PUSHFP
LOAD -1
ADD
PUSHI 2
DIV
STOREG 0
RETURN
```

5.3 Input e Output de Caracteres

5.3.1 Strings

Código 5.13: Terminal

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

Código 5.14: Máquina Virtual

```
START
PUSHA TOFU
CALL
STOP
TOFU:
PUSHS " Yummy bean curd!"
WRITES
RETURN
```

Código 5.15: Terminal

```
: TOFU ." Yummy bean curd!" ;
: SPROUTS ." Miniature vegetables."
: MENU
 CR TOFU CR SPROUTS CR
;
MENU
```

Código 5.16: Máquina Virtual

```
START
PUSHA MENU
CALL
STOP
TOFU:
PUSHS " Yummy bean curd!"
WRITES
RETURN
SPROUTS:
PUSHS " Miniature vegetables."
WRITES
RETURN
MENU:
WRITELN
PUSHA TOFU
CALL
WRITELN
PUSHA SPROUTS
CALL
WRITELN
RETURN
```

5.3.2 Input

Código 5.17: Terminal

```
: TESTKEY ( -- )
  ." Hit a key: " KEY CR
  ." That = " . CR
;
```

Código 5.18: Máquina Virtual

```
START
PUSHA TESTKEY
CALL
STOP
TESTKEY:
PUSHS " Hit a key: "
WRITES
READ
WRITELN
PUSHS " That = "
WRITES
WRITES
WRITELN
RETURN
```

5.4 Condicionais

Código 5.19: Terminal

```
: maior2 ( a b -- ) 2dup > if swap . ." e o maior "
 else . ." e o maior " then ;
77 156 maior2
```

Código 5.20: Máquina Virtual

```
START
PUSHI 77
PUSHI 156
PUSHA maior2
CALL
POP 2
STOP
maior2:
PUSHFP
LOAD -2
PUSHFP
LOAD -1
PUSHSP
LOAD -1
PUSHSP
LOAD -1
SUP
jz ELSE
SWAP
WRITEI
PUSHS " e o maior "
WRITES
JUMP THEN
ELSE:
WRITEI
PUSHS " e o maior "
WRITES
THEN:
RETURN
```

Código 5.21: Terminal

```
: maior2 ( a b -- c ) 2dup > if drop then ;
: maior3 ( a b c -- ) maior2 maior2 . ;
2 11 3 maior3
```

Código 5.22: Máquina Virtual

```
PUSHI 0
START
PUSHI 2
PUSHI 11
PUSHI 3
PUSHA maior3
CALL
POP 3
STOP
maior2:
PUSHFP
LOAD -2
PUSHFP
LOAD -1
PUSHSP
LOAD -1
PUSHSP
LOAD -1
SUP
jz THEN
POP 1
THEN:
STOREG 0
RETURN
maior3:
PUSHFP
LOAD -3
PUSHFP
LOAD -2
PUSHFP
LOAD -1
PUSHA maior2
CALL
POP 2
PUSHG 0
PUSHA maior2
CALL
POP 2
PUSHG 0
WRITEI
RETURN
```

5.5 Ciclos

Código 5.23: Terminal

```
: somatorio ( a -- b ) begin 0 swap 1 do 5 2 * loop ;
11 somatorio .
```

Código 5.24: Máquina Virtual

```
PUSHI 0
START
PUSHI 11
PUSHA somatorio
CALL
POP 1
PUSHG 0
WRITEI
STOP
somatorio:
PUSHFP
LOAD -1
PUSHI 0
SWAP
PUSHI 1
ALLOC 2
SWAP
STORE 0
PUSHST 0
SWAP
STORE 1
CYCLE:
PUSHI 5
PUSHI 2
MUL
PUSHST 0
LOAD 0
PUSHI 1
ADD
DUP 1
PUSHST 0
SWAP
STORE 0
PUSHST 0
LOAD 1
EQUAL
JZ REPEAT
JUMP ENDCYCLE
REPEAT:
JUMP CYCLE
ENDCYCLE:
STOREG 0
RETURN
```

Código 5.25: Terminal

```
: STAR 42 EMIT ;
: STARS 5 0 DO STAR LOOP ;
STARS
```

Código 5.26: Máquina Virtual

```
START
PUSHA STARS
CALL
STOP
STAR:
 PUSHI 42
WRITECHR
RETURN
STARS:
 PUSHI 5
 PUSHI 0
 ALLOC 2
SWAP
STORE 0
PUSHST 0
SWAP
STORE 1
CYCLE:
PUSHA STAR
CALL
PUSHST 0
LOAD 0
 PUSHI 1
ADD
DUP 1
PUSHST 0
SWAP
STORE 0
PUSHST 0
LOAD 1
EQUAL
 JZ  REPEAT
JUMP ENDCYCLE
REPEAT:
JUMP CYCLE
ENDCYCLE:
RETURN
```

Código 5.27: Terminal

```
BEGIN 3 0 DO ." TESTE " CR LOOP
```

Código 5.28: Máquina Virtual

```
START
PUSHI 3
PUSHI 0
ALLOC 2
SWAP
STORE 0
PUSHST 0
SWAP
STORE 1
CYCLE:
PUSHS " TESTE "
WRITES
WRITELN
PUSHST 0
LOAD 0
PUSHI 1
ADD
DUP 1
PUSHST 0
SWAP
STORE 0
PUSHST 0
LOAD 1
EQUAL
JZ REPEAT
JUMP ENDCYCLE
REPEAT:
JUMP CYCLE
ENDCYCLE:
STOP
```

5.6 Variáveis

Código 5.29: Terminal

```
VARIABLE DIA
VARIABLE MES
VARIABLE ANO
: DATA ( m d a -- ) ANO ! DIA ! MES ! ;
05 12 2024 DATA
: PRINTDATA ( -- ) DIA @ . ."/" MES @ . ."/" ANO @ . ;
PRINTDATA
```

Código 5.30: Máquina Virtual

```
PUSHI 0
PUSHI 1
PUSHI 2
START
PUSHI 5
PUSHI 12
PUSHI 2024
PUSHA DATA
CALL
POP 3
PUSHI 5
PUSHI 12
PUSHI 2024
PUSHA DATA
CALL
POP 3
PUSHI 5
PUSHI 12
PUSHI 2024
PUSHA DATA
CALL
POP 3
PUSHA PRINTDATA
CALL
STOP
DATA:
PUSHFP
LOAD -3
PUSHFP
LOAD -2
PUSHFP
LOAD -1
STOREG 2
STOREG 0
STOREG 1
RETURN
PRINTDATA:
PUSHG 0
WRITEI
PUSHS "/"
WRITES
PUSHG 1
WRITEI
PUSHS "/"
WRITES
PUSHG 2
WRITEI
RETURN
```

Capítulo 6

Extras

Com o intuito de enriquecer ainda mais o projeto, decidiu-se desenvolver um compilador adicional. Ao contrário de fornecer comandos para execução na máquina virtual, este compilador é capaz de executar os comandos por si só, demonstrando passo a passo cada iteração do processo. É importante salientar que este compilador é restrito a valores numéricos. Sendo assim o compilador aceita os seguintes comandos:

- **NUM**: Adiciona um valor na *stack*.
- **ADD**: Soma os dois valores no topo da *stack*.
- **SUB**: Subtrai os dois valores no topo da *stack*.
- **MUL**: Multiplica os dois valores no topo da *stack*.
- **DIV**: Divide os dois valores no topo da *stack*.
- **MOD**: Calcula o módulo entre os dois valores no topo da *stack*.
- **DOT**: Imprime o valor que está no topo da *stack*.
- **CONDITION**: Realiza uma condição lógica entre os dois valores no topo da *stack*, retornando 1 *(True)* ou 0 *(False)*.
- **NOT**: Coloca a 0 o valor que está no topo da *stack*.
- **DUP**: Duplica o valor que está no topo da *stack*.
- **POP**: Remove o valor que está no topo da *stack*.
- **SWAP**: Troca a posição dos dois valores que estão no topo da *stack*.
- **NEGATE**: Inverte o sinal do valor que está no topo da *stack*.
- **DEPTH**: Retorna a profundidade da *stack*.
- **EMPTY**: Esvazia a *stack*.

6.0.1 Adição

Código 6.1: Terminal

```
2 3 + .
```

Código 6.2: Resultado

```
Push: 2
Stack: [2]
Push: 3
Stack: [2, 3]
Operation: +
Stack: [5]
Topo da Stack: 5
```

6.0.2 Subtração

Código 6.3: Terminal

```
30 5 - .
```

Código 6.4: Resultado

```
Push: 30
Stack: [30]
Push: 5
Stack: [30, 5]
Operation: -
Stack: [25]
Topo da Stack: 25
```

6.0.3 Multiplicação

Código 6.5: Terminal

```
30 5 * .
```

Código 6.6: Resultado

```
Push: 30
Stack: [30]
Push: 5
Stack: [30, 5]
Operation: *
Stack: [150]
Topo da Stack: 150
```

6.0.4 Divisão

Código 6.7: Terminal

```
30 5 / .
```

Código 6.8: Resultado

```
Push: 30
Stack: [30]
Push: 5
Stack: [30, 5]
Operation: /
Stack: [6]
Topo da Stack: 6
```

6.0.5 Resto da Divisão

Código 6.9: Terminal

```
10 20 % .
```

Código 6.10: Resultado

```
Push: 10
Stack: [10]
Push: 20
Stack: [10, 20]
Operation: %
Stack: [10]
Topo da Stack: 10
```

6.0.6 Condição

Código 6.11: Terminal

```
14 24 = .
32 43 < .
```

Código 6.12: Resultado

```
Push: 14
Stack: [14]
Push: 24
Stack: [14, 24]
Condition: =
Stack: [0]
Topo da Stack: 0
Push: 32
Stack: [0, 32]
Push: 43
Stack: [0, 32, 43]
Condition: <
Stack: [0, 1]
Topo da Stack: 1
```

6.0.7 Not

Código 6.13: Terminal

```
62 (-1) 0
23 4 5
NOT
```

Código 6.14: Resultado

```
Push: 62
Stack: [62]
Push: -1
Stack: [62, -1]
Push: 0
Stack: [62, -1, 0]
Push: 23
Stack: [62, -1, 0, 23]
Push: 4
Stack: [62, -1, 0, 23, 4]
Push: 5
Stack: [62, -1, 0, 23, 4, 5]
Command: NOT
Stack: [62, -1, 0, 23, 4, 0]
```

6.0.8 Dup

Código 6.15: Terminal

```
34 5 63
+
DUP
```

Código 6.16: Resultado

```
Push: 34
Stack: [34]
Push: 5
Stack: [34, 5]
Push: 63
Stack: [34, 5, 63]
Operation: +
Stack: [34, 68]
Command: DUP
Stack: [34, 68, 68]
```

6.0.9 Pop

Código 6.17: Terminal

```
34 5 63
-
POP
```

Código 6.18: Resultado

```
Push: 34
Stack: [34]
Push: 5
Stack: [34, 5]
Push: 63
Stack: [34, 5, 63]
Operation: -
Stack: [34, -58]
Command: POP
Stack: [34]
```

6.0.10 Swap

Código 6.19: Terminal

```
34 5 63
*
SWAP
```

Código 6.20: Resultado

```
Push: 34
Stack: [34]
Push: 5
Stack: [34, 5]
Push: 63
Stack: [34, 5, 63]
Operation: *
Stack: [34, 315]
Command: SWAP
Stack: [315, 34]
```

6.0.11 Negate

Código 6.21: Terminal

```
34 5 63
-
NEGATE
```

Código 6.22: Resultado

```
Push: 34
Stack: [34]
Push: 5
Stack: [34, 5]
Push: 63
Stack: [34, 5, 63]
Operation: -
Stack: [34, -58]
Command: NEGATE
Stack: [34, 58]
```

6.0.12 Depth

Código 6.23: Terminal

```
34 5 63
+
DEPTH
```

Código 6.24: Resultado

```
Push: 34
Stack: [34]
Push: 5
Stack: [34, 5]
Push: 63
Stack: [34, 5, 63]
Operation: +
Stack: [34, 68]
Tamanho da Stack: 2
```

6.0.13 Empty

Código 6.25: Terminal

```
34 5 63
+
45
12
*
EMPTY
```

Código 6.26: Resultado

```
Push: 34
Stack: [34]
Push: 5
Stack: [34, 5]
Push: 63
Stack: [34, 5, 63]
Operation: +
Stack: [34, 68]
Push: 45
Stack: [34, 68, 45]
Push: 12
Stack: [34, 68, 45, 12]
Operation: *
Stack: [34, 68, 540]
Command: EMPTY
Stack: []
```

Capítulo 7

Conclusão

Ao concluir este projeto, pode afirmar-se que os desafios propostos foram superados com sucesso. Embora tenham sido encontradas algumas dificuldades, tais como na interação com a máquina virtual e na implementação dos ciclos, que não foi obtido um resultado positivo na execução de um dos exemplos do enunciado, fomos capazes de superá-los, o que nos permitiu aprofundar nosso conhecimento em processamento de linguagens. Além disso, decidiu-se ir além do solicitado e desenvolveu-se um compilador adicional, demonstrando iniciativa perante o projeto.
Em suma, estamos satisfeitos com o nosso trabalho e com os resultados obtidos.

Capítulo 8

Enunciado do Trabalho Prático

Processamento de Linguagens

Engenharia Informática (3º ano)

Projeto Final

18 de Março de 2024

Objectivos e Organização

Este trabalho prático tem como principais objectivos:

- 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.

Na resolução dos trabalhos práticos desta UC, aprecia-se a imaginação/criatividade dos grupos em todo o processo de desenvolvimento! Deve entregar a sua solução até Domingo dia 12 de Maio.

O ficheiro com o relatório e a solução deve ter o nome 'pl2024- projeto-grNN.zip' de deverá estar no formato ZIP, em que NN corresponderá ao número de grupo. O número de grupo será ou foi atribuído no registo das equipes do projeto.

A submissão deverá ser feita no Black Board da UC.

Na defesa, a realizar na semana de 13 a 17 de Maio, o programa desenvolvido será apresentado aos membros da equipa docente, totalmente pronto e a funcionar (acompanhado do respectivo relatório de desenvolvimento) e será defendido por todos os elementos do grupo, em data e hora a marcar. O relatório a elaborar, deve ser claro e, além do respectivo enunciado, da descrição do problema, das decisões que lideraram o desenho da solução e sua implementação, deverá conter exemplos de utilização (textos fontes diversos e respectivo resultado produzido). Como é de tradição, o relatório será escrito em LaTeX.

Recursos e documentação

Estão disponíveis na internet vários recursos quer de documentação quer de ambientes de programação para Forth que o aluno poderá usar para estudar a linguagem e perceber melhor o problema.

Podemos destacar os seguintes recursos que foram usados na preparação do projeto e para criar os programas exemplo neste enunciado:

- **Documentação**: um dos muito manuais online apadrinhado pelo criador da linguagem: *"...I hope this book is not so easy and enjoyable that it seems trivial. Be warned that there is heavy content here*

and that you can learn much about computers and compilers as well as about programming." — *Charles Moore*;

- **IDE de desenvolvimento e testes**: IDE para escrita e teste de programas, neste IDE os programas são compilados;
- **Máquina Virtual (VM)** a ser usada para a geração de código: Documentação e IDE para testes e execução.

Forth: uma linguagem de programação

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.

Em resumo, Forth é uma linguagem de programação simples, eficiente e flexível, que se destaca pela sua abordagem baseada numa stack e pela capacidade de se adaptar a uma ampla gama de aplicações, desde sistemas embebidos até sistemas operativos.

Existem muitos manuais disponíveis na internet pelo que não descreveremos mais da linguagem neste documento. No entanto, apresentam-se uma série de programas comentados que poderão ajudar a perceber a linguagem e poderão ser usados como teste no compilador.

Enunciado: o problema

Neste projeo, deverás implementar um compilador de Forth que deverá gerar código para a máquina virtual criada no contexto desta UC e disponível online.

Ambas as linguagens, o Forth e o código máquina da máquina virtual serão apresentados e trabalhados nas aulas teóricas.

Na introdução deste documento, tens as localizações da documentação e ambientes de teste e desenvolvimento que terás de usar.

Este projeto será avaliado pelo seu grau de completude e apresentam-se já os seguintes níveis, do mais básico para o mais completo:

1. Suporte a todas as expressões aritméticas: soma, adição, subtração, divisão, resto da divisão inteira;
2. Suporte à criação de funções;
3. Suporte ao `print`de carateres e strings (`.`, `." string"`, `emit`, `char`);
4. Suporte a condicionais;
5. Suporte a ciclos;
6. Suporte a variáveis;
7. ...

Sintaxe

Um programa em FORTH é uma lista de palavras (`words`) separadas por espaço:

```
WAKE.UP EAT.BREAKFAST WORK EAT.DINNER PLAY SLEEP
```

ou

```
." #S SWAP ! @ ACCEPT . *
```

Baseada em Stacks

Todas as operações em FORTH assentam na manipulação de Stacks. A mais simples e mais usada é a Stack de Dados (dataStack).

A Stack de Dados está inicialmente vazia. Para colocarmos valores na stack, basta introduzirmos esses valores como palavras:

```
17 34 23
```

Para imprimir o valor no topo da stack usamos a palavra `.`:

```
< 17 34 23
< .
> 23
```

E agora usa os recursos apontados para estudares a linguagem.

Aritmética

Exemplo:

```
2 3 + .
2 3 + 10 + .
```

As expressões aritméticas escrevem-se no formato RPN ("Reverse Polish Notation").

```
30 5 - . ( 25=30-5 )
30 5 / . ( 6=30/5 )
30 5 * . ( 150=30*5 )
30 5 + 7 / . \ 5=(30+5)/7
```

Atalhos: `1+ 1- 2+ 2- 2* 2/`

Experimentar no interpretador online:

```
10 1- .
7 2* 1+ . ( 15=7*2+1 )
```

Exercício: Conversão de expressões em formato infixo para pós-fixo

Escreve expressões FORTH para as seguintes expressões aritméticas:

```
(12 * ( 20 - 17 ))
(1 - ( 4 * (-18) / 6) )
( 6 * 13 ) - ( 4 * 2 * 7 )
```

Definição de uma nova palavra ou função

Utilizam-se duas novas palavras: `:` e `;` Que marcam o início e o fim de uma definição de uma nova palavra ou função.

Exemplo: `: AVERAGE ( a b -- avg ) + 2/ ;`

```
: AVERAGE ( a b -- avg ) + 2/ ;
10 20 AVERAGE .
```

Input e Output de carateres

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

```
CHAR ( <char> -- char , get ASCII value of a character )
```

Strings

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

```
: SPROUTS ." Miniature vegetables." ;
: MENU
    CR TOFU CR SPROUTS CR
;
MENU
```

Input

```
: TESTKEY ( -- )
    ." Hit a key: " KEY CR
    ." That = " . CR
;
TESTKEY
```

Resumindo:

```
EMIT ( char -- , output character )
KEY ( -- char , input character )
SPACE ( -- , output a space )
SPACES ( n -- , output n spaces )
CHAR ( <char> -- char , convert to ASCII )
CR ( -- , start new line , carriage return )
." ( -- , output " delimited text )
```

Alguns programas exemplo em Forth

A seguir apresentam-se alguns programas em Forth que poderão ajudar a compreender melhor a linguagem e que podem ser usados para testar o compilador desenvolvido.

Hello world!

```
: hello-world  ( -- )
  ." Hello, World!"  cr ;

hello-world  \ Call the defined word
```

Notas:

- `: hello-world` inicia a definição de uma nova palavra designada `hello-world`;
- `( -- )` indica que `hello-world` não recebe argumentos e não deixa nada na stack;
- `." Hello, World!"` coloca na saída (stdout) a string `"Hello, World!"` sem alterar o estado da stack;
- `cr` escreve um caráter de mudança de linha '\n' na saída;
- `;` termina a definição da nova palavra.

Maior de 2 números passados como argumento

```
: maior2 2dup > if swap . ." é o maior " else . ." é o maior " then ;
77 156 maior2
```

Maior de 3 números passados como argumento

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

Maior de N números passados como argumento

```
: 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 .
```

Somatório de 1 até n-1

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

Playing with chars and strings

```
( May the Forth be with you)
: STAR 42 EMIT ;
: STARS   0 DO  STAR  LOOP ;
: MARGIN  CR 30 SPACES ;
: BLIP MARGIN STAR ;
: IOI MARGIN STAR 3 SPACES STAR ;
```

```
: IIO MARGIN STAR STAR 3 SPACES ;
: OIO MARGIN 2 SPACES STAR 2 SPACES ;
: BAR  MARGIN 5 STARS ;
: F    BAR BLIP BAR BLIP BLIP CR ;
: O BAR IOI IOI IOI BAR CR ;
: R BAR IOI BAR IIO IOI CR ;
: T BAR OIO OIO OIO OIO CR ;
: H IOI IOI BAR IOI IOI CR ;
F O R T H
```

Fatorial

```
: factorial  ( n -- n! )
  dup 0 = if
    drop 1
  else
    dup 1 - recurse *
  then ;

\ Example usage:
5 factorial .  \ Calculate factorial of 5 and print result
```

Notas:

Explanation:

- `: factorial` inicia a definição de uma nova palavra designada `factorial`;
- `( n -- n! )` indica que `factorial` recebe um argumento (n) e deixa um resultado (n!) na stack;
- `dup 0 = if` testa se o argumento é 0;
- Se o argumento for 0, este é descartado e o valor 1 é colocado na stack (será o caso de base parao cálculo do fatorial);
- Se o argumento não for 0, este é duplicado, subtrai-se 1 ao duplicado, a função é chamada recursivamente, e o seu resultado é multiplicado pelo argumento original;
- `then` marca o fim do bloco condicional;
- `\ Example usage:` é um comentário que indica como usar a palavra/função nova;
- `5 factorial .` calcula o fatorial de 5 e escreve o resultado na saída.

Este programa mostra a utilização da recursividade em Forth, neste caso, para calcular o fatorial.

Somatório de n números

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

\ Example usage:
5 sum .
```

Notas:

- `: sum` inicia a definição de uma nova palavra designada `sum`;
- `( n -- sum )` indica que `sum` recebe 1 argumento (n) e deixa1 resultado (sum) na stack;
- `0 swap` inicializa `sum` a 0 e troca o argumento com o topo da stack;
- `1 do` inicia o ciclo de 1 até ao valor do argumento;
- `i +` adiciona o índice do ciclo corrente (i) a `sum`;
- `loop` termina o ciclo;
- `\ Example usage:` é um comentário;
- `5 sum .` calcula e imprime o somatório dos números inteiros de 1 a 5.