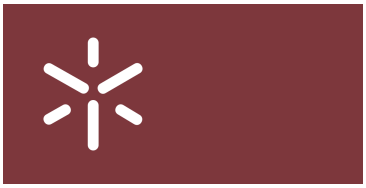

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2023/2024

Projeto Final

Bruno Silva a100828
Manuel Serrano a100825
Marta Gonçalves a100593

May 12, 2024

PL

Índice

1. Introdução

Neste projeto de Processamento de Linguagens, tivemos como objetivo a tradução da linguagem *Forth* para a linguagem da máquina virtual. O *Forth* utiliza notação pós-fixa, a qual não estávamos habituados a trabalhar e que nos obrigou a reformular completamente o nosso raciocínio. Para desenvolver este trabalho, recorremos à ferramenta *PLY* do *Python*, tanto para a análise léxica como para a análise sintática. Cumprimos todos os pontos propostos pelo enunciado e desenvolvemos alguns tópicos extras, como variáveis locais, comentários e as funções pré-definidas pelo *Forth*. No último caso, embora não tenhamos conseguido implementar todas, definimos as mais utilizadas.

O relatório deste projeto abordará os seguintes tópicos: análise léxica, onde se discute a análise dos tokens da linguagem *Forth*; análise sintática, onde se descreve a estrutura gramatical da linguagem e a sua interpretação; análise semântica, onde se avalia o significado das expressões e instruções em *Forth*; e, por fim, serão apresentados os testes realizados para validar a nossa gramática.

2. Análise Léxica

Para a análise léxica, fomos analisando as componentes da linguagem *Forth* que o nosso compilador suporta e fomos identificando os diferentes *tokens* e literais (no caso de ser preciso identificar um único carater) que era necessário o nosso analisador léxico encontrar.

Os literais encontrados, assim como alguns dos *tokens* correspondem aos **símbolos** que podemos encontrar na linguagem *Forth*. Alguns dos *tokens* representam ainda **palavras reservadas**, isto é, representam a palavra que lhes dá o nome, seja em maiúsculas, minúsculas ou uma combinação de ambas, e correspondem a comandos específicos na linguagem, não lhes podendo ser atribuido outro significado. Por fim, temos ainda alguns *tokens* que são **terminais variáveis** e que nos permitem obter conjuntos de carateres que seguem uma determinada expressão regular e que se inserem numa determinada categoria, para serem tratados da mesma forma dependendo do contexto em que aparecem.

É ainda de notar que tivemos de ter em conta a ordem pela qual fomos definindo os diferentes tokens, visto que, por exemplo, o *token* NOME engloba todas as palavras e, caso aparecesse antes das definições de palavras reservadas, impediria que estas fossem reconhecidas como tal.

2.1. Operações aritméticas

- *Tokens*:
  - INT -> engloba as sequências não vazias de dígitos e devolve o valor inteiro correspondente.
  - MOD -> palavra reservada
- Literais:
  - ‘+’
  - ‘-’
  - ‘/’
  - ‘*’

2.2. Comparação de valores

- *Tokens*:
  - MAIORIGUAL -> símbolo ‘>=’
  - MENORIGUAL -> símbolo ‘<=’
- Literais:

- ‘<’
- ‘>’
- ‘=’

2.3. Comentários

- *Token* COMMENT -> Reconhece comentários de linha (começados por \) e multilinha (entre parênteses) e ignora o seu conteúdo pois não é relevante para um compilador

2.4. Prints

- *Tokens*:
  - PRINT -> encontra um ponto seguido da abertura de parênteses ou de aspas e lê o conteúdo até ao fecho de parênteses ou aspas, devolvendo apenas o conteúdo que será impresso posteriormente
  - CHAR -> palavra reservada
  - EMIT -> palavra reservada
- Literais:
  - ‘.’

2.5. Funções

- *Tokens*:
  - NOME -> encontra sequências de carateres alfanuméricos ou ‘-’ que podem representar o nome das funções ou de variáveis
- Literais:
  - ‘:’
  - ‘;’

2.6. Condicionais

- *Tokens*:
  - IF -> palavra reservada
  - THEN -> palavra reservada
  - ELSE -> palavra reservada

2.7. Ciclos

- *Tokens*:
  - BEGIN -> palavra reservada
  - UNTIL -> palavra reservada
  - WHILE -> palavra reservada
  - REPEAT -> palavra reservada
  - DO -> palavra reservada
  - LOOP -> palavra reservada
  - LOOPN -> palavra reservada (+LOOP)
  - I -> palavra reservada

2.8. Variáveis

- *Tokens*:
  - VARIABLE -> palavra reservada
  - LOCALS -> palavra reservada
  - VALUE -> palavra reservada
  - TO -> palavra reservada
- Literais:
  - '@'
  - '|'
  - '!'

2.9. Funções predefinidas

- *Token* SPACES -> palavra reservada

3. Análise Sintática (Gramática)

3.1. Gramática

De seguida, apresenta-se a gramática que elaboramos, seguida de uma explicação de algumas das decisões tomadas.

T = {'INT', 'MOD', 'COMMENT', 'PRINT', 'CHAR', 'EMIT', 'NOME', 'VARIABLE', 'MAIORIGUAL', 'MENORIGUAL', 'IF', 'THEN', 'ELSE', 'BEGIN', 'UNTIL', 'WHILE', 'REPEAT', 'DO', 'LOOP', 'LOOPN', 'VALUE', 'TO', 'LOCALS', 'SPACES', 'I', '+', ':', '-', '/', '*', '.', ';', '!', '>', '<', '@', '|', '='}

N = {Forth, Funcs, Funcao, Ciclos, Is, Operacoes, Fator, Nomefunc, PontoVirgula, Vars, Var, Nomes}

S = Forth

P = {

```
Forth -> Funcs

Funcs -> Funcs Funcao
Funcs -> Funcao

Ciclos -> Ciclos Funcs Is
Ciclos -> Is

Is -> I
Is -> &

Funcao -> ':' Nomefunc Funcs PontoVirgula
Funcao -> ':' Nomefunc Vars Funcs PontoVirgula
Funcao -> Operacoes
Funcao -> IF Funcs THEN
Funcao -> IF Funcs ELSE Funcs THEN
Funcao -> BEGIN Funcs UNTIL
Funcao -> BEGIN Funcs WHILE Funcs REPEAT
Funcao -> DO Ciclos LOOP
Funcao -> DO Ciclos LOOPN
```

```
    Nomefunc -> NOME

    PontoVirgula -> ';'

    Vars -> Vars Var
    Vars -> Var

    Var -> LOCALS '|' Nomes '|'

    Nomes -> NOME
    Nomes -> Nomes NOME

    Operacoes -> Fator
    Operacoes -> Operacoes Fator

    Fator -> INT
    Fator -> '+'
    Fator ->  '-'
    Fator -> '\*'
    Fator -> '/'
    Fator -> '='
    Fator -> '>'
    Fator -> '<'
    Fator -> MAIORIGUAL
    Fator -> MENORIGUAL
    Fator -> MOD
    Fator ->  '.'
    Fator ->  PRINT
    Fator ->  EMIT
    Fator ->  COMMENT
    Fator ->   CHAR
    Fator ->   VARIABLE NOME
    Fator ->   NOME '!'
    Fator ->  NOME
    Fator ->  NOME '\@'
    Fator ->   VALUE NOME
    Fator ->   TO NOME
    Fator ->   INT   SPACES
}
```

A nossa gramática está organizada em sete níveis distintos. Estes são os seguintes: `Forth`, `Funcs`, `Funcao`, `Ciclos`, `Is`, `Operacoes` e `Fator`. De seguida, abordaremos cada um desses níveis separadamente, explicando as produções mais relevantes. Optamos por utilizar recursividade à esquerda nas produções para melhor deteção de erros.

3.1.1. Forth

Este é o nível inicial da nossa gramática, e como o próprio enunciado sugere, a linguagem *Forth* é composta por várias funções. A produção neste nível reflete precisamente essa composição.

3.1.2. Funcs

Por sua vez este nível revela-nos que um programa em *Forth* pode ser composto por uma única função ou por múltiplas funções.

3.1.3. Função

É neste nível que encontramos as produções mais importantes do nosso projeto. Aqui estão definidas as produções para identificar funções, condicionais, diferentes tipos de ciclos existentes e ainda as operações mais elementares.

Começando pela produção mais básica, as funções podem ser compostas por **operações**. A `operacoes` é um símbolo não-terminal e, portanto, originará novas produções, que serão abordadas mais adiante neste tópico.

Para as **funções**, temos duas produções: uma sem variáveis locais e outra com variáveis locais. Em ambos os casos, a função é definida da mesma forma, exceto pela presença ou ausência de `vars`. `Vars` consiste no *token* `Locals`, seguido de uma lista com os nomes das variáveis. É relevante mencionar que a definição das variáveis locais só pode ocorrer no início da função, ou seja, após o nome da função. Optamos por ter uma produção para `nomefunc`, pois é necessário para a análise semântica sabermos em que função estamos. Decidimos também incluir uma produção para `pontoVirgula`, pois é onde identificamos o final da função. Quanto ao corpo da função, este pode ser composto por operações, novas funções, condicionais, e, portanto, definimos o corpo da função como `funcs`.

Quanto aos **condicionais**, implementamos os dois tipos existentes em *Forth*. A primeira produção representa o caso em que o condicional é apenas composto por `if` e `then`. À esquerda do `if` temos a condição. De seguida, temos o corpo do condicional, que mais uma vez pode ser composto por operações, novas funções, ciclos e novos condicionais. Portanto, o corpo de execução dos condicionais corresponde a `funcs` A segunda produção para o condicional é bastante semelhante à primeira, com a diferença de que é adicionado o `else` e o respetivo corpo de execução.

Passando agora para os **ciclos**, que são talvez as produções mais complexas de definir. Começando pelo ciclo `Begin-Until`, iniciamos identificando o *token* `Begin`. De seguida, as operações a serem realizadas e a condição do ciclo são capturadas pela produção `funcs`. Assim que o *token* `Until` é encontrado, o ciclo é concluído e reconhecido pela nossa gramática.

Avançando agora para o ciclo `Begin-While-Repeat`, mais uma vez o ciclo é iniciado com o *token* `Begin`. De seguida, o primeiro corpo do ciclo é composto por operações e por uma condição, identificados pelo símbolo não terminal `funcs`. A palavra `While` é reconhecida, e o segundo corpo do ciclo é composto apenas por operações, também identificadas pelo símbolo não terminal `funcs`. O ciclo é reconhecido e concluído assim que o *token* `Repeat` é encontrado.

3.1.3.1. Ciclos, “is”

Para definir a produção do ciclo `Do-Loop`, foi necessário introduzir uma produção auxiliar denominada `ciclos`. O ciclo é iniciado com a identificação do *token* `Do`. De seguida, o símbolo não terminal `ciclos` é chamada, que pode incluir novos ciclos, operações, condicionais ou a definição de índices, representados pelo simbolo não terminal `is`. O `is` pode ser apenas a letra “i”, indicando a presença de um índice, ou pode ser vazio. A identificação do ciclo é concluída com o reconhecimento do token `Loop`. Como se pode constatar, a definição deste ciclo é um pouco complexa.

3.1.4. Operações

A definição de operações em termos gramaticais é bastante simples. As operações podem ser compostas por vários **fatores** ou por apenas um **fator**.

3.1.5. Fator

Na nossa gramática, o símbolo não terminal `fator` representa as operações mais elementares do *Forth*, abrangendo desde a identificação de inteiros até operações aritméticas, *prints*, definição de variáveis e

chamadas de funções pré-definidas ou declaradas pelo utilizador. É uma lista de produções destinada a identificar todas estas operações de forma abrangente.

4. Análise Semântica (Transformação para código da VM)

Para a análise semântica, fomos percorrendo as diferentes produções e analisando como poderíamos tornar o código *Forth* capturado em cada uma em código para a Máquina Virtual.

A primeira produção, que define o símbolo não terminal Forth, engloba todo o código que foi sendo gerado pelas outras produções, sendo utilizada para alocar espaço para as variáveis necessários, inicializar o programa após esta alocação com um "START" e terminar o programa com um "STOP".

De seguida, vamos explicar como obtivemos o código VM a partir das produções de diferentes tipos e algumas estruturas adicionais que utilizamos para nos ajudar.

4.1. Expressões aritméticas e comparação de valores

Sendo *Forth* uma linguagem pós-fixa, assim como a linguagem da Máquina Virtual, a passagem das operações aritméticas e das operações de comparação de valores para código VM, são bastante diretas até porque já existem os comandos específicos para estas operações. Assim, a conversão é a seguinte:

- INT -> pushi (valor do INT)
- '+' -> ADD
- '-' -> SUB
- '*' -> MUL
- '/' -> DIV
- MOD -> MOD
- '<' -> INF
- '>' -> SUP
- '=' ->EQUAL
- MAIORIGUAL -> SUPEQ
- MENORIGUAL -> INFEQ

4.2. Variáveis Globais

No tratamento das variáveis, encontramos dois casos diferentes: as variáveis que são alocadas previamente e aquelas que aparecem quando lhes é atribuído um valor através da palavra reservada "to". Estas variáveis são guardadas em dicionários diferentes, no "variable_dict" e no "value_dict", respetivamente. O nome da variável é então a chave do dicionário, sendo o seu valor a posição de memória em que ficará guardada na VM. Para obter a posição de memória, utilizamos uma variável chamada "posicao_memoria" e, sempre que criávamos uma variável nova, utilizávamos o seu valor incrementávamo-la.

Assim, era possível, de acordo com as produções, ao encontrar um token NOME, verificar se ele existia em algum destes dicionários e, existindo, saber a posição de memória em que essa variável vai estar guardada na Máquina Virtual. Desta forma, sabendo que uma determinada variável se encontra numa posição n, para obter o seu valor basta utilizar o comando "pushg n" e para guardar um valor nessa variável utilizar "storeg n".

As funções auxiliares utilizadas para a criação de variáveis verificam primeiro a sua existência, encontrando erros sempre que o nome que se pretende utilizar já está em uso. Já as funções que procuram as variáveis nos dicionários para devolver a sua posição de memória encontram erros como a não existência de uma variável com o nome pretendido.

4.3. Funções pré-definidas e criação de funções

O nosso grupo implementou algumas funções pré-definidas do *Forth* utilizando o código da máquina virtual. Estas, juntamente com todas as funções implementadas pelo utilizador, são inseridas no dicionário `func_dict`. Assim que uma função é reconhecida pela nossa gramática, a função `func_def_unica` é chamada para verificar se o nome da função já existe no dicionário. Se existir, é gerado um erro em tempo de compilação e o utilizador é alertado sobre a repetição de nomes de funções, sendo também indicada a função que está a ser repetida. Para cada função presente no dicionário `func_dict`, o seu código correspondente na máquina virtual é associado como valor.

Nas funções com variáveis locais, é relevante ter uma produção para o nome da função, pois isso indica se estamos dentro de uma função específica ou se estamos na função `main`. Para armazenar esta informação, criamos uma variável global, `func_atual`. Além disso, é necessário definir uma produção para o ponto e vírgula, pois este indica o fim da definição da função, permitindo assim a atualização do valor da variável `func_atual`. Nas funções que envolvem variáveis locais, o processo é semelhante ao das funções "normais", com a diferença de que é necessário chamar a função `adicionarVarsLocais`. Esta função tem como objetivo alocar posições de memória para as variáveis locais e adicionar esses valores e respetivas variáveis ao dicionário `locals_dict`, onde os nomes das funções servem como chaves. Nesta função caso seja detetado dois nomes iguais nas variáveis locais é emitido um erro para o utilizador.

A chamada de funções ocorre quando a produção "fator : NOME" é identificada. De seguida, a função `verificar_tipo_variavel` é invocada para determinar o tipo do nome identificado, isto é, se corresponde a uma variável ou a uma função. Nesta secção, focamo-nos apenas na identificação de funções e variáveis locais.

A função começa por verificar o valor da variável `func_atual`. Se este valor for diferente de "main" e pertencer ao dicionário `locals_dict`, estamos perante a possibilidade de uma variável local. Neste caso, é necessário recuperar a sua posição de memória e efetuar um `pushg` dessa posição na máquina virtual. Se `func_atual` for igual a "main", trata-se da chamada de uma função ou de uma variável global. No caso de ser uma função, é necessário aceder ao dicionário `func_dict` e obter o código correspondente na máquina virtual.

Relativamente ao código devolvido pelo dicionário `func_dict`, é necessário aplicar a função `change_labels`. Se isso não for feito, com múltiplas chamadas de funções, as *labels* utilizadas para os condicionais e ciclos seriam repetidas e o código não teria o comportamento desejado. Por conseguinte, aplicamos a função `change_labels`, que encontra essas *labels* e altera os seus nomes para garantir que são diferentes em cada chamada.

4.4. Print de caracteres e strings

4.4.1. Print

Para imprimir o último valor da pilha em *Forth*, é necessário utilizar o comando ".". Assim que a nossa gramática identifica este símbolo, é gerado o seguinte código para a máquina virtual: `writei`. Como todos os valores na pilha são inteiros, apenas esta instrução é necessária.

4.4.2. Print de strings

Para imprimir strings em *Forth*, é necessária a seguinte sintaxe: `." string a imprimir"` ou `.( string a imprimir)`. Este padrão é identificado na análise léxica pela função `t_PRINT`. Na análise sintática, através da produção "fator : PRINT", é retornado o código correspondente na máquina virtual: `pushs {t[1]} writes`, em que `t[1]` representa a string capturada no analisador léxico.

4.4.3. Emit e Char

A função `Emit` em *Forth* recebe um código ASCII e imprime no terminal o caractere correspondente. Para realizar esta operação na máquina virtual, utilizamos as funções já predefinidas nela, como é o caso da função `writechr`.

A função `Char` opera de forma oposta à função `Emit`, ou seja, recebe um caractere e tenta convertê-lo no código ASCII, colocando-o no topo da *stack*. No entanto, diferentemente da maioria das funções em *Forth*, a chamada desta função não segue a notação pré-fixa. Isto ocorre porque a função recebe um caractere enquanto a *stack* armazena apenas inteiros. Para realizar esta passagem para a máquina virtual, precisamos realizar um `pushs t[1]`, onde `t[1]` corresponde o caractere a ser transformado, seguido da execução da função pré-definida na máquina virtual, `Chrcode`.

4.5. Condicionais

Tal como foi referido anteriormente, as condições dividem-se em dois casos, aquele em que se pretende apenas executar algum código quando se verifica uma determinada condição (IF … THEN) e aquele em que existem trechos de código alternativos consoante a condição seja verdadeira ou falsa (IF … ELSE … THEN), sendo ambos explicados de seguida.

4.5.1. IF … THEN

Em Forth este tipo de condicional tem o seguinte formato, em que cond é a condição a ser verificada e codn é um trecho de código:

cond IF cod1 THEN cod2

O código a gerar para a Maquina Virtual segue o seguinte algoritmo:

- cond em formato VM
- jz then **(Salta para a label then caso não seja verdade)**
- cod1 em formato VM
- then:
- cod2 em formato VM

4.5.2. IF … ELSE … THEN

Este tipo de condicional tem uma estrutura idêntica mas permite ter código alternativo para executar:

cond IF cod1 ELSE cod2 THEN cod3

Isto é obtido da seguinte forma:

- cond em formato VM
- jz else **(Salta para a label else caso não seja verdade)**
- cod1 em formato VM
- jump then **(quando acaba o código do if salta para o fim da condição para não executar a alternativa)**
- else:
- cod2 em formato VM
- then:
- cod3 em formato VM

4.5.3. Estruturas auxiliares

Como foi visto anteriormente, estas estruturas recorrem a labels para realizar saltos. No entanto, estas labels ficando com os mesmo nomes, iriam dar origem a alguma confusão quando aparecessem mais do que uma vez, pelo que os nomes das labels se seguem por um valor inteiro que permite a sua distinção. Este valor inteiro é otido a partir de uma variável global chamada "ifs" que vai sendo incrementada à medida que aparecem novas condições.

4.6. Ciclos

Seguindo o mesmo formato que os condicionais, apresentamos de seguida a estrutura de 4 ciclos diferentes em Forth e a forma como é possível realizar a sua transformação para código VM. Esse ciclos são: BEGIN … UNTIL, BEGIN … WHILE … REPEAT, DO … LOOP e DO .. +LOOP.

4.6.1. BEGIN … UNTIL

BEGIN cod1 cond UNTIL cod2

O ciclo BEGIN … UNTIL permite a execução de código até que uma condição seja verdadeira.

- begin:
- cod1 em formato VM
- cond em formato VM
- jz begin **(caso a condição não se verifique, volta para o início do ciclo)**
- cod2 em formato VM

4.6.2. BEGIN … WHILE … REPEAT

BEGIN cod1 cond WHILE cod2 REPEAT cod3

O ciclo BEGIN … WHILE … REPEAT permite a execução de código enquanto uma condição for verdadeira, saltando-se para o fim quando isto não se realizar.

- begin:
- cod1 em formato VM
- cond em formato VM
- jz end **(caso a condição não se verifique, salta-se para o fim do ciclo)**
- cod2 em formato VM
- jump begin **(se continuamos a executar, volta-se novamente ao início)**
- end:
- cod3 em formato VM

4.6.3. LOOPS

Os loops têm algumas particularidades, como o facto de possuírem um index que define o valor inicial e um limit que indica até que valor do index o ciclo deve ser executado. Estes dois valores vão sendo necessários ao longo do ciclo para averiguar a condição de paragem, pelo que optamos por alocar memória para guardar os seus valores como se fossem variáveis.

Além disso, possuem também uma variável I que coloca na stack o valor atual do index. A nossa solução foi criar produções para o corpo do ciclo que encontrassem a variável I no início, meio ou fim das restantes operações e separasse os Is do resto do código, sendo obtida uma lista. Essa lista era posteriormente per-

corrida de modo a substituir as posições do I por storeg index, em que o index diz respeito á sua posição em memória.

4.6.3.1. DO … LOOP

limit index DO cod1 LOOP cod2

- guardar espaço para 2 variáveis
- storeg index
- storeg limit
- while:
- cod1 em formato VM com a colocação dos valores do index na posição dos Is
- pushg index
- pushi 1
- add **(incrementar o index)**
- storeg index **(guardar o novo valor do index)**
- pushg index
- pushg limit
- supeq **(verificar se o index já atingiu o valor limit)**
- jz while **(caso ainda não tenha sido atingido salta para o início do ciclo)**
- cod2 em formato VM

4.6.3.2. DO … +LOOP

limit index DO cod1 valor +LOOP cod2

- storeg index
- storeg limit
- while:
- cod1 em formato VM com a colocação dos valores do index na posição dos Is
- dup 1 **(duplica o valor que vai ser adicionado ao index para depois o avaliar)**
- pushg index
- add **(a incrementação é feita de acordo com o valor no topo da stack)**
- storeg index
- **se o valor que vai ser adicionado ao index for > 0:**
- pushg index
- pushg limit
- supeq **(o index deve alcançar ou superar o limit para sair do ciclo)**
- **senão:**
- pushg index
- pushg limit
- infeq **(o index deve ser igual ou inferior ao limit para sair do ciclo)**
- **por fim:**
- jz while **(volta para o início do ciclo caso o limit não tenha sido alcançado)**
- cod2 em formato VM

Este algoritmo simplifica o código na medida em que não apresenta a condição em código VM para facilitar a sua compreensão mas no código gerado a condição está implementada em código da Máquina Virtual.

Esta funcionalidade funciona apenas quando os valores de incrementação do loop apresentam o mesmo sinal, caso contrário pode dar problemas.

4.6.4. Estruturas auxiliares

Tal como acontece nos condicionais, os ciclos também apresentam o problema da identificação das labels, havendo mais 2 variáveis globais chamadas “begins” e “dos” que funcionam da mesma forma que o “ifs” já explicado atrás.

Além disso, tal como foi referido, foram criadas novas produções para lidar com a existência da variável I.

Por fim, tal como se verificou para as variáveis, criamos um dicionário de indexs e limits onde novamente associamos o nome da variável à posição de memória (n) em que se encontra, podendo alterar o seu valor com “storeg n” ou obtê-lo através da expressão “pushg n”.

5. Testes

Nesta secção, apresentaremos alguns testes realizados para validar a nossa gramática. Aqui, serão apresentados apenas testes simples, devido ao grande número de instruções geradas para a máquina virtual. Testes mais completos serão fornecidos em anexo. Os testes serão apresentados em ordem crescente de complexidade, dos mais simples para os mais complexos.

5.1. Operações aritméticas

Código em *Forth* :

```
5 3 + 4 2 - * 5 mod .
```

Código na máquina virtual:

```
START
pushi 5
pushi 3
ADD
pushi 4
pushi 2
SUB
MUL
pushi 5
MOD
writei
pushs " "
writes
STOP
```

Resultado

```
1
```

5.2. Variáveis

5.2.1. Teste 1

Código em *Forth* :

```
VARIABLE num1
VARIABLE num2
```

```
VARIABLE resultado

5 num1 !
3 num2 !

num1 @ num2 @ +
resultado !

resultado @ .
```

Código na máquina virtual:

```
pushi 0
pushi 0
pushi 0
START
pushi 5
storeg 0
pushi 3
storeg 1
pushg 0
pushg 1
ADD
storeg 2
pushg 2
writei
pushs " "
writes
STOP
```

Resultado

```
8
```

5.2.2. Teste 2

Código em *Forth* :

```
0 VALUE cinco
0 VALUE tres

5 TO cinco
3 TO tres

cinco tres + .
```

Código na máquina virtual:

```
pushi 0
pushi 0
START
pushi 0
storeg 0
pushi 0
storeg 1
pushi 5
storeg 0
pushi 3
storeg 1
pushg 0
pushg 1
ADD
writei
```

```
pushs " "
writes
STOP
```

Resultado

```
8
```

5.3. Funções pré-definidas e criação de funções

5.3.1. Teste 1

Código em *Forth* :

```
: calculo  ( a b c -- resultado )
    *       ( Multiplica a e b )
    2 /     ( Divide o resultado pela metade )
    3 +     ( Adiciona 3 ao resultado )
    swap    ( Inverte a ordem dos dois últimos elementos na pilha )
    -       ( Subtrai c do resultado )
    ;

 1 2 3 calculo .
```

Código na máquina virtual:

```
pushi 1
pushi 2
pushi 3
MUL
pushi 2
DIV
pushi 3
ADD
SWAP
SUB
writei
pushs " "
writes
STOP
```

Resultado:

```
5
```

5.3.2. Teste 2

Código *Forth*:

```
: test-locals  ( a b c -- )
           LOCALS| c b a |
           CR  ." Normal order: " a .  b .  c .
           CR  ." Stack order:  " c .  b .  a .
           13 TO a   14 TO b  15 TO c  \ how TO works
           CR ." Changed: " a . b . c
 ;

3 4 5 test-locals
```

Código na máquina virtual:

```
pushi 0
pushi 0
pushi 0
START
pushi 3
pushi 4
pushi 5
storeg 0
storeg 1
storeg 2
WRITELN
pushs "Normal order:  "
writes
pushg 2
writei
pushs " "
writes
pushg 1
writei
pushs " "
writes
pushg 0
writei
pushs " "
writes
WRITELN
pushs "Stack order:   "
writes
pushg 0
writei
pushs " "
writes
pushg 1
writei
pushs " "
writes
pushg 2
writei
pushs " "
writes
pushi 13
storeg 2
pushi 14
storeg 1
pushi 15
storeg 0
WRITELN
pushs "Changed:  "
writes
pushg 2
writei
pushs " "
writes
pushg 1
writei
pushs " "
writes
pushg 0
STOP
```

Resultado:

```
Normal order: 3 4 5
Stack order: 5 4 3
Changed: 13 14
```

5.4. Print de caracteres e strings

5.4.1. Teste 1

Código *Forth*:

```
CHAR A DUP .
CR

: stringsChars  ( --- )
     .( Vou confirmar que o valor impresso é mesmo o valor ASCII de A: )
     EMIT CR
     2 3 *
     ." Imprimir o valor da multiplicação: " .
 ;

stringsChars
```

Código na máquina virtual:

```
START
pushs "A"
CHRCODE
DUP 1
writei
pushs " "
writes
WRITELN
pushs "Vou confirmar que o valor impresso é mesmo o valor ASCII de A:  "
writes
WRITECHR
WRITELN
pushi 2
pushi 3
MUL
pushs "Imprimir o valor da multiplicação:  "
writes
writei
pushs " "
writes
STOP
```

Resultado:

```
65
Vou confirmar que o valor impresso é mesmo o valor ASCII de A:  A
Imprimir o valor da multiplicação:  6
```

5.4.2. Teste 2

Código *Forth*:

```
: imprimir_caracteres  ( -- )
    ." Iniciando impressão de caracteres..." CR
    65 CHAR EMIT CR  ( Imprime o caractere 'A' )
```

```
    DUP 2 + .  ( Utiliza a função pré-definida DUP para duplicar o valor no topo da pilha,
soma 2 e imprime o resultado )
    ." Esta é uma mensagem de exemplo." CR
    67 CHAR EMIT CR  ( Imprime o caractere 'C' )
    ." Finalizando impressão de caracteres." CR ;

imprimir_caracteres
```

Código na máquina virtual:

```
START
pushs "Iniciando impressão de caracteres... "
writes
WRITELN
pushi 65
pushs "EMIT"
CHRCODE
WRITELN
DUP 1
pushi 2
ADD
writei
pushs " "
writes
pushs "Esta é uma mensagem de exemplo. "
writes
WRITELN
pushi 67
pushs "EMIT"
CHRCODE
WRITELN
pushs "Finalizando impressão de caracteres. "
writes
WRITELN
STOP
```

Resultado:

```
Iniciando impressão de caracteres...

71 Esta é uma mensagem de exemplo.

Finalizando impressão de caracteres.
```

5.5. Condicionais

5.5.1. Teste 1

Código em *Forth* :

```
: TOO-HOT  220 > IF ." Danger -- reduce heat " THEN ;
1 . 290 TOO-HOT
CR
2 . 130 TOO-HOT
```

Código na máquina virtual:

```
START
pushi 1
writei
pushs " "
```

```
writes
pushi 290
pushi 220
SUP
jz then1f1
pushs "Danger -- reduce heat  "
writes
then1f1:
WRITELN
pushi 2
writei
pushs " "
writes
pushi 130
pushi 220
SUP
jz then1f3
pushs "Danger -- reduce heat  "
writes
then1f3:
STOP
```

Resultado

```
1 Danger -- reduce heat
2
```

5.5.2. Teste 2

Código em *Forth* :

```
: DAY  DUP 1 <  SWAP 31 > + IF ." No way " ELSE ." Looks good " THEN ;
35 DAY
```

Código na máquina virtual:

```
START
pushi 35
DUP 1
pushi 1
INF
SWAP
pushi 31
SUP
ADD
jz else1f3
pushs "No way  "
writes
jump then1f3
else1f3:
pushs "Looks good  "
writes
then1f3:
STOP
```

Resultado

```
 way
```

5.6. Ciclos

5.6.1. Teste 1

Código em *Forth* :

```
10 BEGIN 1 - DUP DUP . 1 < UNTIL ." DONE"
```

Código na máquina virtual:

```
START
pushi 10
begin1:
pushi 1
SUB
DUP 1
DUP 1
writei
pushs " "
writes
pushi 1
INF
jz begin1
pushs "DONE "
writes
STOP
```

Resultado

```
9 8 7 6 5 4 3 2 1 0 DONE
```

5.6.2. Teste 2

Código em *Forth* :

```
10 BEGIN 1 - DUP DUP . 1 > WHILE ." | " REPEAT
```

Código na máquina virtual:

```
START
pushi 10
begin1:
pushi 1
SUB
DUP 1
DUP 1
writei
pushs " "
writes
pushi 1
SUP
jz end1
pushs "|  "
writes
jump begin1
end1:
STOP
```

Resultado

```
9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1
```

5.6.3. Teste 3

Código em *Forth* :

```
10 0 DO  I .  LOOP
```

Código na máquina virtual:

```
pushi 0
pushi 0
START
pushi 10
pushi 0
storeg 0
storeg 1
while1:
pushg 0
writei
pushs " "
writes
pushg 0
pushi 1
add
storeg 0
pushg 0
pushg 1
supeq
jz while1
STOP
```

Resultado `0 1 2 3 4 5 6 7 8 9`

5.6.4. Teste 4

Código em *Forth* :

```
32767 1 DO  I . I +LOOP
```

Código na máquina virtual:

```
pushi 0
pushi 0
START
pushi 32767
pushi 1
storeg 0
storeg 1
while1:
pushg 0
writei
pushs " "
writes
pushg 0
dup 1
pushg 0
add
storeg 0
pushi 0
inf
jz else1
pushg 0
pushg 1
infeq
jump then1
else1:
pushg 0
pushg 1
```

```
supeq
then1:
jz while1
STOP
```

Resultado

```
 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384
```

5.7. Testes gerais

Estes testes foram criados de forma a intercalarem diferentes tipos de conteúdo para percebermos como o nosso programa se comporta quando aparecem todos os tipos de operações possíveis intercaladas

5.7.1. Teste 1

Código em *Forth* :

```
0 VALUE seven
7 TO seven

: MULTIPLICATIONS  CR 11 1 DO  DUP I * .  LOOP  DROP ;
seven MULTIPLICATIONS CR CR

: RECTANGLE  49 0 DO  I seven MOD 0= IF  CR  THEN ." *" LOOP ;
RECTANGLE
```

Código na máquina virtual:

```
pushi 0
pushi 0
pushi 0
pushi 0
pushi 0
START
pushi 0
storeg 0
pushi 7
storeg 0
pushg 0
WRITELN
pushi 11
pushi 1
storeg 1
storeg 2
while1f4:
DUP 1
pushg 1
MUL
writei
pushs " "
writes
pushg 1
pushi 1
add
storeg 1
pushg 1
pushg 2
supeq
jz while1f4
```

```
POP 1
WRITELN
WRITELN
pushi 49
pushi 0
storeg 3
storeg 4
while2f8:
pushg 3
pushg 0
MOD
pushi 0
EQUAL
jz then1f8
WRITELN
then1f8:
pushs "* "
writes
pushg 3
pushi 1
add
storeg 3
pushg 3
pushg 4
supeq
jz while2f8
STOP
```

Resultado

```
7 14 21 28 35 42 49 56 63 70


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

5.7.2. Teste 2

Código em *Forth* :

```
VARIABLE eggs

23 eggs !

: 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 ;

eggs @ EGGSIZE
CR
```

```
29 eggs !
eggs @ EGGSIZE
CR

40 eggs !
eggs @ EGGSIZE
CR
```

Código na máquina virtual:

```
pushi 0
START
pushi 23
storeg 0
pushg 0
DUP 1
pushi 18
INF
jz else5f7
pushs "reject  "
writes
jump then5f7
else5f7:
DUP 1
pushi 21
INF
jz else4f7
pushs "small  "
writes
jump then4f7
else4f7:
DUP 1
pushi 24
INF
jz else3f7
pushs "medium  "
writes
jump then3f7
else3f7:
DUP 1
pushi 27
INF
jz else2f7
pushs "large  "
writes
jump then2f7
else2f7:
DUP 1
pushi 30
INF
jz else1f7
pushs "extra large  "
writes
jump then1f7
else1f7:
pushs "error  "
writes
then1f7:
then2f7:
then3f7:
then4f7:
then5f7:
POP 1
```

```
WRITELN
pushi 29
storeg 0
pushg 0
DUP 1
pushi 18
INF
jz else5f9
pushs "reject  "
writes
jump then5f9
else5f9:
DUP 1
pushi 21
INF
jz else4f9
pushs "small  "
writes
jump then4f9
else4f9:
DUP 1
pushi 24
INF
jz else3f9
pushs "medium  "
writes
jump then3f9
else3f9:
DUP 1
pushi 27
INF
jz else2f9
pushs "large  "
writes
jump then2f9
else2f9:
DUP 1
pushi 30
INF
jz else1f9
pushs "extra large  "
writes
jump then1f9
else1f9:
pushs "error  "
writes
then1f9:
then2f9:
then3f9:
then4f9:
then5f9:
POP 1
WRITELN
pushi 40
storeg 0
pushg 0
DUP 1
pushi 18
INF
jz else5f11
pushs "reject  "
writes
```

```
jump then5f11
else5f11:
DUP 1
pushi 21
INF
jz else4f11
pushs "small  "
writes
jump then4f11
else4f11:
DUP 1
pushi 24
INF
jz else3f11
pushs "medium  "
writes
jump then3f11
else3f11:
DUP 1
pushi 27
INF
jz else2f11
pushs "large  "
writes
jump then2f11
else2f11:
DUP 1
pushi 30
INF
jz else1f11
pushs "extra large  "
writes
jump then1f11
else1f11:
pushs "error  "
writes
then1f11:
then2f11:
then3f11:
then4f11:
then5f11:
POP 1
WRITELN
STOP
```

Resultado

```
medium
extra large
error
```

6. Conclusão

Ao longo deste trabalho, aprendemos o funcionamento da linguagem *Forth*, assim como da linguagem da Máquina Virtual (VM) e conseguimos identificar os diferentes componentes da primeira, perceber como interagiam entre si e transformá-los em código para a VM. Apesar de termos tido algumas dificuldades em situações específicas que foram sendo expostas ao longo deste relatório, fomos capazes de criar um programa que transforma código *Forth*, já com alguma complexidade, em código para a VM que apresente a mesma funcionalidade.