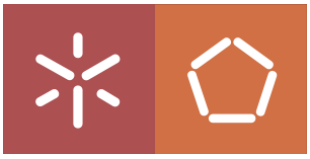
Universidade do Minho

Departamento de Informática



LICENCIATURA EM ENGENHARIA INFORMÁTICA

Processamento de Linguagens

Gonçalo Brandão

Maya Gomes

Henrique Pereira

Gonçalo Brandão A100663
Maya GomesA100822
Henrique Pereira A100831

Maio de 2024

Conteúdo

1 Descrição do Problema

O problema apresentada na Unidade Curricular de Processamento de Linguagens em 2024 foi a implementação de um compilador para a linguagem FORTH que irá gerar código para a maquina Virtual EWVM. FORTH é uma linguagem de baixo nivel que opta por uma abordagem baseada em stack e com notação pós-fixa (Reverse Polish Notation - RPN), onde os operadores são colocados após os operandos.

Para tal, o compilador deverá ser capaz de gerar código para várias características do Forth como, expressões aritméticas, suporte de funções, strings, condicionais, ciclos e variáveis. Sendo necessário compreender tanto a documentação do FORTH como da própria EWVM garantindo sim o desenvolvimento e execução de programas eficientes e funcionais.

2 Introdução

Assim neste relatório dividimos a resolução deste problema por várias etapas, começando pelo analisar lexico, onde procuramos interpretar cada token de forma precisa, facilitando a análise posterior do programa. A uma analise dos fatores mais importantes da nossa gramática que permitem a expressão dos conceitos fundamentais da linguagem FORTH.

Sendo que de seguida, analisamos o que implementamos no nosso projeto final, os motivos por de trás das nossas decisões e o porque destas nos parecerem as decisões mais corretas para a resolução deste problema.

3 Léxico

O analisador léxico que desenvolvemos é feito especificamente para interpretar a linguagem de programação Forth. Cada "token"no código-fonte está associado a uma expressão regular que define como é reconhecido no código.

Os tokens STRING e FUNCAO_RAW são tratados de forma especial, usando a função re.match para transmitir apenas o conteúdo essencial ao yacc. Esse processo é crucial, pois divide o código-fonte em unidades significativas, facilitando a análise do programa. O analisador léxico é uma parte essencial do processo de compilação ou interpretação no nosso projeto.

```
def t_STRING(t):
    r'\."(.*?)"'
    match = re.match(r'."(.*?)"', t.value)
    if match:
        t.value = match.group(1)
    return t

def t_FUNCAO_RAW(t):
    r':\s+?(\w+)\s+?(\([^()]*\))?\s*(.*?)(?=;);'
    match = re.match(r':\s+?(\w+)\s+?(\([^()]*\))?\s*(.*?)(?=;);', t.value)
    if match:
        t.value = ": " + match.group(1) + " " + match.group(3) + ";"
        return t
```

Figura 1: Tokens STRING e FUNCAO_RAW

4 Gramática

$$
\begin{aligned}
Z &: \text{exp}' \ \$' \\
\\
\text{exp} &: \text{termo} \\
&\mid \text{termo exp} \\
\\
\text{termo} &: \text{NUM} \\
&\mid \text{expressao\_aritmetica} \\
&\mid \text{FUNCAO} \\
&\mid \text{FUNCAO\_RAW} \\
&\mid \text{condicional} \\
&\mid \text{condicional expressao\_condicional} \\
&\mid \text{str} \\
&\mid \text{PONTO} \\
\\
\text{expressao\_aritmetica} &: \text{SOMA} \\
&\mid \text{SUBTRACAO} \\
&\mid \text{MULT} \\
&\mid \text{DIV} \\
&\mid \text{MOD} \\
&\mid \text{SWAP} \\
&\mid \text{2DROP} \\
&\mid \text{DIVMOD} \\
&\mid \text{2DUP} \\
&\mid \text{OVER} \\
&\mid \text{DUP} \\
&\mid \text{DROP} \\
&\mid \text{2SWAP} \\
&\mid \text{ROT} \\
\\
\text{condicional} &: \text{MAIOR} \\
&\mid \text{MENOR} \\
\\
\text{expressao\_condicional} &: \text{IF exp THEN} \\
&\mid \text{IF exp THEN exp} \\
&\mid \text{IF exp ELSE exp THEN} \\
&\mid \text{IF exp ELSE exp THEN exp} \\
\\
\text{str} &: \text{CHAR} \\
&\mid \text{SPACE} \\
&\mid \text{CR} \\
&\mid \text{STRING} \\
&\mid \text{EMIT}
\end{aligned}
$$

Na formulação da gramática tomamos algumas decisões de modo que acrescentar funcionalidades ao projeto fosse o mais simples possível.

Definimos exp : termo | term exp , este regra permite-nos a geração de expressões que consistem em um ou mais termo, permitindo a recursividade na gramática, também garante que é possível o push de um elemento isolado para a stack e que os termos seguintes terão acesso aos termos anteriores.

O símbolo não-terminal "termo"desempenha um papel crucial na gramática, servindo como um ponto de expansão fundamental. Ele representa uma unidade que pode ser substituída por uma variedade de expressões terminais, como números, funções, bem como por expressões não-terminais, como expressões aritméticas, strings e condicionais. Essas expressões não-terminais, por sua vez, consistem em agrupamentos de outras expressões terminais ou não-terminais, permitindo a formação de estruturas mais complexas. Assim, a expansão da gramática envolve principalmente a organização de expressões terminais semelhantes em expressões não-terminais, que, por sua vez, podem ser utilizadas na construção de estruturas mais elaboradas através da regra associada ao símbolo não-terminal "termo".

5 Operações Aritméticas

5.1 Operações Aritméticas Básicas

Para implementar as operações Aritméticas descritas no enunciado, percebemos que teríamos de garantir de alguma forma que estas operações teriam elementos suficientes na stack para realizar as mesmas. Primeiro, tentamos que a gramática garantisse isto, contudo, apercebemo-nos que estávamos a aplicar demasiada semântica à nossa gramática. Portanto, decidimos implementar uma variável global que nos garantia que a nossa stack tinha sempre elementos suficientes para garantir que as operações eram possíveis, no final adicionamos um elemento à stack do resultado da operação.

```
def p_4soma (p):
    '''expressao_artimetica : SOMA '''
    global stack_size
    stack_size -=2
    if stack_size < 0:
        p[0] = "Error: Stack Vazia"
    stack_size +=1
    p[0] = f'add\n'
```

Figura 2: Exemplo da Operação Aritmética SOMA

5.2 Operações Aritméticas Avançadas

Ao analisar a documentação de forth fornecida no enunciado, apercebemo-nos que havia mais operações aritméticas do que as que eram pedidas no enunciado e tivemos curiosidade em como implementá-las.

Algumas operações tinham traduções literais para a EWVM como o *swap*; *drop*, que na EWVM é *pop* mas tem o mesmo efeito ou por exemplo *2drop* que tem a mesma funcionalidade que o *pop 2* na EWVM.

Outros exemplos mais complexos em forth como 2SWAP e 2OVER, que utilizam *double digit operators* seriam mais complicados de aplicar na nossa estrutura em stack.

OVER	(n1 n2 -- n1 n2 n1)	Makes a copy of the second item and pushes it on top.
ROT	(n1 n2 n3 -- n2 n3 n1)	Rotates the third item to the top.
DROP	(n --)	Discards the top stack item.
2SWAP	(d1 d2 -- d2 d1)	Reverses the top two pairs of numbers.
2DUP	(d -- d d)	Duplicates the top pairs of numbers.
2OVER	(d1 d2 -- d1 d2 d1)	Makes a copy of the second pair of numbers and pushes it on top.
2DROP	(d --)	Discards the top pair of numbers.

Figura 3: Excerto da documentação

Analisando a documentação da EWVM decidimos que a melhor abordagem seria utilizar as funções ***storeg n*** e ***pushg n*** para realizar estas operações. Para isto ser possível teríamos de alocar espaço para as variáveis antes de fazer as operações, por isso, temos uma variável global que conta o número de alocações que vão ser necessárias escrever antes de gerar o código para a EWVM.

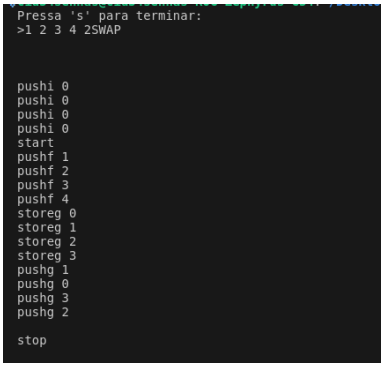


Figura 4: Geração do Código da VM para 2SWAP

No final do projeto, aplicamos as seguintes operações aritméticas com as estratégias e garantias descritas em cima:

SOMA	*SUBTRACAO*
MULT	*DIV*
MOD	*DIVMOD*
DROP	*2DROP*
OVER	*2OVER*
SWAP	*2SWAP*
DUP	*2DUP*
ROT	

6 Implementação de Funções

A nossa estratégia para a implementação de funções foi muito influenciada pela documentação da EWVM, inicialmente, pretendíamos definir *labels* na EWVM e o utilizar o método *call* para usar as funções. Contudo, a alteração dos apontadores da EWVM quando isto acontecia era um obstáculo, tornando esta implementação mais complexa do que era suposto. Assim, decidimos não utilizar este método, mas sim sempre que um utilizador chamava uma função gerávamos todo o código da VM dessa função.

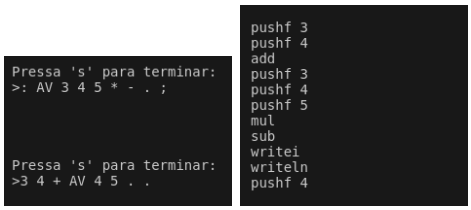


Figura 5: Geração do Código da VM para Funções

Para esta implementação percebemos que necessitaríamos de 2 tokens, um para inicialização de uma função, caracterizado por : <NOME DA FUNÇÃO> <CONTEÚDO> ; e outro token para quando o nome da função aparecer gerar o código da função.

Assim, decidimos que o token para inicializar uma função deveria ser uma expressão regular que capturasse os dois grupos alvo entre : e ;. E que na hora que este token terminal aparecesse na gramática devia guardar o <Conteúdo> da função num dicionário onde o <Nome da Função> seria a *Key* do mesmo.

O *token* de quando o nome da função aparece, fornece o conteúdo da key ao *parser*, obtendo assim o código da VM da função.

7 Strings

Na implementação das strings decidimos implementar com a mesma lógica que nas operações aritméticas. Começando pelo o ponto decidimos traduzi-lo para a EWVM como *writei writeln* para ser mais fácil a perceção dos outputs na máquina virtual.

Para o CHAR e ."string"decidimos, no analisador lexico, devolver no token obtido o conteúdo do mesmo, de modo a ser mais fácil a sua utilização. Para outros como o EMIT e CR aplicamos o equivalente ao código da EWVM.

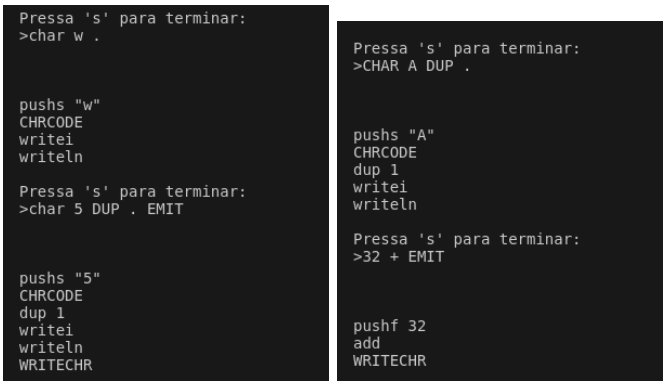


Figura 6: Exmplos do Enunciado

No final das contas aplicamos os seguintes operadores:

.	*CHAR*
SPACE	*CR*
."string"	*EMIT*

8 Condicionais

De modo a suportar condicionais decidimos dividir em 2 subgrupos, condicionais, por exemplo *‘2 3 >‘* e expressões condicionais *‘IF 3 ELSE 4 THEN 5‘* .

No primeiro grupo, os condicionais, realizamos as condições de maior ou menor, sendo que aplicamos a mesma lógica que definimos no inicio do projeto para as operações aritméticas e apenas traduzindo as instruções para o código da EWVM, sup e inf, respetivamente.

Já para as expressões condicionais, decidimos aplicar esta gramática:

```
'''expressao_condicional    : IF exp THEN
                            | IF exp THEN exp
                            | IF exp ELSE exp THEN
                            | IF exp ELSE exp THEN exp '''
```

Figura 7: Gramática Expressões condicionais

Esta gramática é a razão de todos os warnings de shift/reduce que temos presentes no trabalho, sendo que, o yacc dá sempre shift como nós pretendemos. Mesmo com estes warnings achamos que esta era a solução mais apropriada pois permite facilmente o aninhamento de expressões condicionais. Para isto é apenas preciso trocar o nome das labels dos vários IF's, ELSE's e THEN's, e para isto fizemos uma variável que incrementa a cada utilização.

```
Pressa 's' para terminar:
>3 4 < IF 4 ELSE 3 4 > IF 100 THEN 3 THEN 66



start
pushf 3
pushf 4
inf
jz else2
pushf 4

jump endif2
else2:
pushf 3
pushf 4
sup
jz endif1
pushf 100
endif1:
pushf 3
endif2:
pushf 66

stop
```

Figura 8: Expressões Condicionais aninhadas

9 Conclusão

Equilibrar as funções de diferentes partes de um projeto foi a nossa maior dificuldade. Começamos criação uma gramática que incutia muita semântica, limitando o número de operadores antes e depois de uma operação, e garantindo a validade em todas as etapas. Contudo, percebemos que esse caminho não estava correto. Então decidimos, reestruturar o trabalho, adicionando um layer ao Yacc. Porém, apercebemo-nos que isto era redundante, pois efetivamente, estávamos a fazer o parsing do projeto duas vezes, o que não era a solução pretendida. Assim, com estas duas soluções, aproveitamos o melhor de ambas e fomos produzindo uma nova gramática que conseguisse aplicar tudo o que pretendíamos da forma correta, alcançando este resultado final.

Também gostamos de analisar a documentação tanto do Forth como da EWVM e da definição de estratégias para aplicar o que o Forth fazia a algo com uma estrutura e instruções próprias. Sendo que o que nos deu mais gosto foi a definição das operações aritméticas avançadas que nos fez deferir uma estratégia diferente das outras propostas de implementação. Por fim, acabamos por não implementar os loops o que teria sido uma mais valia para este trabalho.