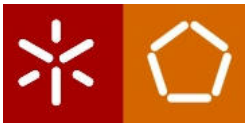

Universidade do Minho
Departamento de Informática

Processamento de Linguagens

Trabalho Prático

Grupo 28

Gonçalo Costa - A100824

Marta Rodrigues - A100743

Diogo Mstos - A100741

Índice

1. Introdução

No âmbito da cadeira de Processamento de Linguagens, foi nos proposta a implementação de um compilador da linguagens de programação **Forth**, gerando código que a máquina virtual,facultada pelos Docentes, fosse capaz de ler.

- Funcionalidades objetivas:
  - Suporte a todas as expressões aritméticas: soma, adição, subtração, divisão, resto da divisão inteira;
  - Suporte à criação de funções;
  - Suporte ao print de carateres e strings (., ." string", emit, char);
  - Suporte a condicionais;
  - Suporte a ciclos;
  - Suporte a variáveis;

2. Forth

O Grupo começou o preparo deste projeto estudando a linguagem **Forth** .

Forth é uma *stack based language*, ou seja, é baseada em *stack* e também é pós fixa. **Forth** foi uma linguagem bem interessante de ser estudada, devido à sua simplicidade, isto deve se ao facto da mesma ser baseada em *Words*, cada *Word* tem uma funcionalidade especifica, podendo ser uma função, uma variável, um ciclo, etc...

Esta linguagem é uma ótima entrada para o mundo da programação, pois apresenta todos os conceitos básicos de programação assim como um bom entendimento de como funciona uma *stack*

Exemplo de código em **Forth** :

```
>> 10 18 + .

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

3. Virtual Machine

A Virtual Machine funciona também à volta de uma *stack*, têm uma sintaxe bastante parecida com *Assembly* que é uma linguagem da baixo nível bastante reconhecida, dado isso, a compreensão de como a mesma funciona não foi tão simples como **Forth**, mas foi bem mais fácil e intuitivo que *Assembly*. Como **Forth** e a **VM** funcionam ambas à volta de *stack*, a tradução do código de um lado para o outro é bastante simplificado, apesar de conter os seus desafios.

4. Arquitetura

4.1. Gramática

Inicialmente foi criada uma gramática *Bottom - UP* que fosse capaz de reconhecer todos os *tokens* e construir o *parser* devidamente.

```
Line  : Line Elem
      | &
Exec : Exec Elem
```

```
      | Elem

Elem : NUMB
      | Operador
      | '.'
      | '.'STRING
      | CHAR LETTER
      | IF Exec THEN
      | IF Exec ELSE Exec THEN
      | Loop
      | Word
      | Var
      | COMENT
      | EMIT
      | CR
      | SWAP
      | DUP
      | DROP
      | KEY
      | SPACE
      | SPACES
      | ATOI
      | Function
Operador : ADD
          | SUB
          | DIV
          | MUL
          | MOD
          | NOT
          | AND
          | OR
          | EQUAL
          | BIGGERTHEN
          | SMALLERTHEN
          | BIGOREQ
          | SMALOREQ
 Var : VARIABLE WORD
      | LOOPI
      | WORD '!'
      | WORD '@'
      | WORD '?
      | WORD MAIS_EXCLAMACAO"
 Loop : Do Exec LOOP
       | Begin Exec UNTIL
       | Begin Exec WHILE Exec REPEAT
 Do : DO
 Begin : BEGIN
 Word : NAME
       | SPACE
 Function : Dots WORD Exec ';'"
 Dots : ':' "
```

O primeiro símbolo não terminal, **Line**, tem como objetivo reconhecer uma expressão da linguagem **Forth**, dando a opção da linha ser vazia, o que gera um output sem nenhum código nenhum e funciona como caso de paragem.

O segundo símbolo, **Exec**, possui recursividade à esquerda, respeitando assim a estratégia *Bottom-UP* e representa toda uma expressão ou parcela de código que será executada.

De seguida, temos **Elem** tem como objetivo listar todos os elementos e conteúdos implementados no trabalho que se quer reconhecer. Estes elementos podem ser números, strings, funções/words, variáveis, funções pré definidas entre outros.

Após esses temos o símbolo **operador** que contém todos os possiveis sinais e operadores, tanto matemáticos como lógicos implementados.

Em seguida temos o símbolo **var** que serve de auxiliar para todas as funcionalidades implementadas mediante as variáveis.

Finalmente temos o símbolo **loop** que apresenta as diferentes representações de ciclos em **Forth** e o símbolo **Function** que representa como se define uma nova *Word* em Forth.

Vale realçar que há dois símbolos separados sendo o **Do** e o **Begin**, estes auxiliam nas verificações necessárias que precisam de ser efetuadas antes de entrar em ciclos e condições, respetivamente.

4.2. Variáveis do Parser

De forma a auxiliar o armazenamento e verificações extras (flags), foram criadas algumas estruturas auxiliares.

- **parser.success**: Um valor booleano que indica se houve algum problema durante o parsing do programa e que ajuda no apanho de erro de compilação.
- **parser.elsecount**: Contador auxiliar do número de *elses*, isto permite ter elses aninhados.
- **parser.thencount**: Contador que segue a mesma lógica do anterior permitindo ter varios *then* aninhados com os *elses*.
- **parser.varpointer**: Dicionário que armazena as variáveis e os seu endereços na *stack*.
- **parser.pushncount**: Contador que vai sendo incrementado à medida que variáveis globais e funções são encontradas com o intuito de alocar espaço para as mesmas no início do programa.
- **parser.looporder**: Lista que armazena os ciclos por ordem, permitindo a existência de vários ciclos.
- **parser.dicfunc**: As novas funções definidas são armazenadas num dicionário, cuja *key* é o nome da função e associada às instruções.
- **parser.funcorder**: Lista que armazena as funções por ordem que elas são chamadas.
- **parser.funcflag**: *Flag* booleana que indica se o programa se encontra dentro de uma definição de uma função.
- **parser.nf**: Contador que armazena o número de funções pré definidas

4.3. Ficheiros

4.3.1. Lexer

Módulo responsável pela divisão do input em **tokens**, com auxilio da biblioteca *ply.lex*, através de expressões regulares.

```
states = (
    ('inloop','inclusive'),
    ('charrec','exclusive'),
)

literals= ['.','"',':',';','!','?','@','(',')']

tokens = (
    'NUMB',
    'STRING',
    'WORD',
    'ADD',
    'SUB',
    'DIV',
    'MUL',
    'MOD',
    'NOT',
    'AND',
    'OR',
    'BIGGERTHEN',
    'SMALLERTHEN',
    'EQUAL',
    'BIGOREQ',
    'SMALOREQ',
    'CHAR',
    'LETTER',
    'IF',
    'THEN',
    'ELSE',
    'BEGIN',
    'UNTIL',
    'WHILE',
    'REPEAT',
    'DO',
    'LOOP',
    'LOOPI',
    'VARIABLE',
    'COMENT',
    'EMIT',
    'CR',
    'SWAP',
    'DUP',
    'DROP',
    'KEY',
    'SPACE',
    'SPACES',
    'ATOI',
    'MAIS_EXCLAMACAO',
)
```

Graças ao auxílio do Professor Pedro Rangel Henriques, definimos dois estados possíveis, que nos foram aconselhados pelo mesmo.

O estado `inloop` permite que a variável I possa ser reconhecida dentro dos ciclos.

O estado `charrec`, permite que após a palavra CHAR o que for encontrado será, definitivamente um carácter, permitindo que esse não seja confundido com uma word ou variável, isso deve ao facto de ele ser **exclusive**.

Vale realçar que os operadores não foram adicionados à lista de `literals` pois esses possuem menor prioridade o que trouxe problemas ao longo do desenvolvimento, podendo esses ser resolvidos definindo os como **tokens**.

4.3.2. Yaccer

Responsável, com o auxilio da biblioteca *ply.yacc*, por definir e construir a gramática e transformar o código **Forth** para código da **VM** .

5. Implementação

5.1. Funções

As funções aproveitam de um símbolo não terminal `Dots` que verifica, antes de entrar na definição da função, que de facto está numa função, isto quando encontra o token ":". A flag auxiliar é "acionada", o que permite ao resto do programa saber que está dentro da definição de um função.

Após isso o nome, assim como o código da função, são armazenados no dicionário de funções.

É importante denotar que nós desenvolvemos duas formas para as funções serem "chamadas", uma que o código é dado imediatamente e outra que usufrui da operação "call", em vez do código ser injetado, é passada para o output o nome da função seguido de um call, e o código das funções é só dado no final do output por ordem que foram chamadas. Para saber quantos argumentos o mesmo precisa, uma variável inicial é definida a −1, se uma das operações da função precisar de um elemento, a variável é incrementado, e se retornar algo é decrementado, sendo o resultado final o número de argumentos necessários.

Exemplo de uma função que é chamada imprime "Hello, World":

Input:

```
: OLA ." Hello, World" ;
OLA
```

Output:

```
pushn 0
pushs " Hello, World"
writes
```

5.2. Condicionais

As condições são bastante simples e diretas, elas são reconhecidas logo no início da gramática, havendo duas possibilidades de definir condições em **Forth**:

- IF Exec THEN
- IF Exec ELSE Exec THEN

Cada vez que uma condição é encontrada é armazenado o valor do seu índice atual, para gerar o seu código com o índice correto e após isso o contador do **else** e do **then** são incrementados(depedente do IF, apenas o **then** é incrementado). Desta forma é possiveis ter várias condições consecutivas.

Exemplo:

Input:

```
1 2 = IF ." Equal " ELSE ." Not Equal " THEN
```

Output:

```
start
pushi 1
pushi 2
equal
jz else0
pushs " Equal "
writes
jump endif0
else0:
pushs " Not Equal "
writes
endif0:
stop
```

5.3. Ciclos

Foram implementados 3 tipos de ciclos.

- Do Exec LOOP
- Begin Exec UNTIL
- Begin Exec WHILE Exec REPEAT

Cada um deles funciona de forma diferente. Seguindo a lógica das funções foram usados dois símbolos não terminais que são reconhecidos no inicio de cada ciclo(**Do** e **Begin**), permitindo fazer as inicializações e verificações necessárias antes de entrar no ciclo.

Caso o **Do** seja verificado inicialmente, é reservada incrementada o contador de reservador de memória e o contador de loops. Após isso é definido o código máquina para a lógica do ciclo, isto é verificado antes de forma a garantir o aninhamento do ciclo corretamente e que o número de iterações é também guardado. Após isso o código do ciclo é adicionado. De forma a garantir que o ciclo cumpre as iterações supostas, o valor da iteração e o limite são sempre verificados através de uma subtração e se o resultado for menor ou igual a 0, sai do ciclo.

Caso seja o **Begin** identificado, a lógica facilita, não sendo preciso definir número de iterações e reservar memória, apenas é acrescentado um *jump* diretamente para o ciclo em si. Após isso o código máquina do ciclo é definido, e quando o valor na *stack* for verdadeiro, saímos do ciclo.

Se for verificada a existência de um **While**, a expressão sucessora do *token* irá ser acrescentada no final no final do ciclo.

Exemplo:

Input:

```
70 0 DO CR ." Hello, World " LOOP
```

Output:

```
pushn 2
Start
pushi 70
pushi 0
storeg 0
storeg 1
jump loop0
loop0:
pushg 0
pushg 1
 sub
 jz endloop0
 writeln
 pushs " Hello, World "
 writes
 pushg 0
 pushi 1
 add
 storeg 0
 jump loop0
 endloop0:
Stop
```

5.4. Variáveis

As variáveis foram desafiadoras no sentido em que havia muitas operações com as mesmas:

```
- VARIABLE WORD -> Define uma nova variável
- LOOPI         -> Variável interior do ciclo
- WORD '!'      -> Guarda um valor na variável
- WORD '@'      -> Substitui o endereço pelo conteudo
- WORD '?'      -> Printa o conteudo do endereço
- WORD "+!"     -> Incrementa a variável
```

Assim como nas variáveis dos ciclos, é guardado no dicionário auxiliar de variáveis, onde lá é armazenada o nome da variável juntamente com a sua posição na *stack*.

Quanto à lógica da implementação, essa é bastante direta, pois todos os símbolos reconhecidos são terminais, sendo devolvido o código da **VM** corresponde às instruções indicadas em cima.

(Devido à **VM** estar em baixo durante a produção desta secção deste relatório, não foi possível gerar código de exemplo para variáveis, mas irá um exemplo seu no ficheiro “output” do código)

5.5. Funções pré definidas implementadas

- DUP
- DROP
- CR
- SWAP
- KEY
- SPACE
- SPACES
- ATOI
- EMIT

Também foram implementadas todas as operações referentes a operações lógicas e matemáticas (ADD, SUB, MOD, DIV, MUL, AND, OR, SUP, EQUAL, ...).

Além disso falta acresentar que, o reconhecimento de números, strings e operações é direto por serem todos símbolos terminais, sendo só preciso adicionar um comando em código máquina.

6. Conclusão Final

Concluindo o desenvolvimento deste projeto, o grupo sente-se bastante satisfeito tendo implementado todas as funcionalidades básicas e importantes para uma linguagem de programação e pedidas pelo enunciado, assim como algumas funções pré definidas. O desenvolvimento deste trabalho prático foi bastante desafiante, trazendo diversas adversidades, mas sentimos que conseguimos supera-las!

Ficamos a sentir que a nossa aprendizagem quanto a gramáticas, expressões regulares e até mesmo como compiladores funcionam, foi bastante produtiva e importante para a nossa formação.

Forth foi uma linguagem bastante bonita e interessante de se aprender, ao longo do seu estudo sentimos que até mesmo o nosso conhecimento de *stacks* e programação num todo, aumentou! Uma linguagem simples mas que têm muito que se diga que só precisa de uma *stack* para ser executada.

Quanto a trabalho futuro, o grupo tem interesse em implementar mais tipos de ciclos como o `Begin Exec AGAIN`, outras features como estruturas de dados e mais funções pré definidas, também é importante polir a gramática que pode ser bastante melhorada e mais complexa de forma a resolver problemas que foram executados, por exemplo por agora, ainda não é possível a implementação de funções recursivas, apesar do grupo ter estado perto da sua implementação e não ter terminado devida à falta de tempo e mudanças extremas na arquitetura da gramática!