Universidade do Minho
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2023/2024

Compilador Forth

Projeto final

Grupo 42
Leonardo Barroso, a100894
Miguel Guimarães, a100837
Pedro Carneiro, a100652

1 Preparação e problema

1.1 Enunciado do projeto

O enunciado do projeto expõe a estrutura do projeto e os objetivos deste, sendo que neste projeto final deverá ser criado um compilador de forth.

O compilador de forth deverá traduzir o código forth em código que possa ser executado na maquina virtual dedicada ao projeto (EWVM). O funcionamento do código final na EWVM deverá possuir o mesmo funcionamento do código original em forth.

O projeto é avaliado pelo seu grau de completude, Abaixo segue-se uma lista das adições sugeridas no enunciado:
. Aritmética
. Criação de funções
. Print de carateres e strings
. Condicionais
. Ciclos
. Variáveis

Forth é uma linguagem que essencialmente traduz as várias operações em ações que são realizadas sobre uma stack de dados.

1.2 Preparação para o projeto

Antes de iniciar a programação de todo o projeto foi realizado um profundo estudo sobre o funcionamento da linguagem e da máquina virtual.

Foi estudado o funcionamento das operações aritméticas do forth, através do manual fornecido no enunciado do projeto. Foram retiradas duvidas em aula sobre a notação utilizada pelo forth e a forma como este manipulava a stack.

Em relação à máquina virtual foram realizadas inúmeras leituras da documentação e dos exemplos fornecidos no site que a hospeda, foram retiradas várias duvidas com a equipa docente ao longo do semestre.

Seguidamente foi prosseguido o desenvolvimento com criação de um analisador léxico que conseguisse em identificar todos os tokens de aritmética.

As informações recolhidas durante esta semana encontram-se na pasta 'Documentação' do projeto.

2 Programas desenvolvidos

2.1 Analisador Léxico (anaLex.py)

Para a criação do analisador léxico foi utilizada a linguagem python com recurso à biblioteca PLY.

Com recurso aos vários manuais de forth foram reconhecidos e adicionados os seguintes tokens:

Nome do símbolo	Exemplo	Utilização
MAIS	+	Soma os dois valores no topo da stack.
MENOS	-	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	MOD	Insere na stack o resto da divisão inteira dos dois valores no topo.
ABS	ABS	Insere no topo da stack o valor absoluto do valor atual no topo da stack.
1SUM	1+	Soma 1 ao valor no topo da stack.
1SUB	1-	Subtrai 1 ao valor no topo da stack.
2DIV	2/	Divide o valor no topo da stack por 2.

Tabela 1: Tokens aritmética

Nome do símbolo	Exemplo	Utilização
COMMENT	(texto exemplo)	O texto entre parêntesis é ignorado.
ENDCOMMENT	\\ texto exemplo	O texto após as barras é ignorado.

Tabela 2: Tokens comentário

Nome do símbolo	Exemplo	Utilização
COLON	:	Utilizado para reconhecer o inicio de uma função.
SEMICOLON	;	Utilizado para reconhecer o fim de uma função.

Tabela 3: Tokens de Função

Nome do símbolo	Exemplo	Utilização
NUM	+19 -1 12	Inserção de um número na stack.
CHAR	CHAR A	Inserção do valor ascii de um caráter na stack.
DROP	DROP	Remoção do primeiro elemento da stack.
DUP	DUP	Duplica o valor no topo da stack.
SWAP	SWAP	Troca a posição dos 1^{o} e 2^{o} elementos da stack.
NIP	NIP	Remove o 2^{o} elemento da stack
2DROP	2DROP	Remove os dois elementos no topo da stack.
ROT	ROT	Coloca o terceiro elemento da stack no topo da mesma.
OVER	OVER	Cria uma copia do 2^{o} elemento da stack e insere no topo da mesma.
2DUP	2DUP	Duplica o par no topo da stack e insere no topo da mesma.
2SWAP	2SWAP	Troca a posição dos dois pares no topo da stack e insere no topo da mesma.
2OVER	2OVER	Cria uma copia do 2^{o} par e insere no topo da stack.
TUCK	TUCK	Insere uma copia do 1^{o} elemento abaixo do 2^{o} elemento da stack.

Tabela 4: Tokens de Stack

Nome do símbolo	Exemplo	Utilização
KEY	KEY	Lê um carater do input do utilizador.

Tabela 5: Tokens de input

Nome do símbolo	Exemplo	Utilização
EQ	=	Insere 1 no topo da stack se os dois valores no topo forem iguais e 0 caso contrario.
NEQ	<>	Insere 1 no topo da stack se os dois valores no topo forem diferentes e 0 caso contrario.
MENOR	<	Insere 1 no topo da stack se o 2º valor no topo da stack for menor que o 1º, inserindo 0 caso contrario.
MAIOR	>	Insere 1 no topo da stack se o 2º valor no topo da stack for maior que o 1º, inserindo 0 caso contrario.
MENOREQ	<=	Insere 1 no topo da stack se o 2º valor no topo da stack for menor ou igual que o 1º, inserindo 0 caso contrario.
MAIOREQ	>=	Insere 1 no topo da stack se o 2º valor no topo da stack for maior ou igual que o 1º, inserindo 0 caso contrario.
ZEROEQ	0=	Insere 1 no topo da stack se o valor no topo da stack for igual a 0, inserindo 0 caso contrario.
ZERONEQ	0<>	Insere 1 no topo da stack se o valor no topo da stack for diferente de 0, inserindo 0 caso contrario.
ZEROMENOR	0<	Insere 1 no topo da stack se o valor no topo da stack for menor que 0, inserindo 0 caso contrario.
ZEROMAIOR	0>	Insere 1 no topo da stack se o valor no topo da stack for maior que 0, inserindo 0 caso contrario.
NEGATE	NEGATE	Inverte o valor no topo da stack.
MIN	MIN	Insere no topo o menor dos dois valores no topo da stack.
MAX	MAX	Insere no topo o maior dos dois valores no topo da stack.

Tabela 6: Tokens de comparação

Nota: Os tokens de comparação inserem um valor (0 ou 1) no topo da stack como resultado, mas removem os valores utilizados na comparação.

Nome do símbolo	Exemplo	Utilização
PONTO	.	Dá print ao valor no topo da stack.
EMIT	EMIT	Dá print ao caráter no topo da stack (que se encontra representado em ascii).
STRPRINT	."texto exemplo"	Dá print ao texto entre aspas.
STRPRINT2	.(texto exemplo)	Dá print ao texto entre parêntesis. Remove os espaços consecutivos no texto entre parentesis.
SPACES	SPACES	Dá print a uma determinada quantidade de espaços. O número de espaços é o valor no topo da stack.
SPACE	SPACE	Dá print a um único espaço.
CR	CR	Dá print a um \n.

Tabela 7: Tokens de print

Nome do símbolo	Exemplo	Utilização
IF	IF	Utilizado para compor condições. Inicio de uma condição, o código que o sucede é executado caso o valor no topo da stack seja diferente de 0.
ELSE	ELSE	Utilizado para compor condições. Sinaliza o local onde se encontra o código a ser executado caso o valor no topo da stack seja igual a 0.
THEN	THEN	Utilizado para compor condições. O código que o sucede é sempre executado.

Tabela 8: Tokens dos condicionais

Nome do símbolo	Exemplo	Utilização
DO	DO	Utilizado para compor ciclos. Indica o inicio de um ciclo.
LOOP	LOOP	Utilizado para compor ciclos. Indica o final de um ciclo.
I	I	Extrai o valor do contador interno do ciclo e insere na stack.

Tabela 9: Tokens dos ciclos

Nome do símbolo	Exemplo	Utilização
VARIABLE	VARIABLE	Utilizado para criar variáveis.
FETCH	@	Utilizado para obter o valor de uma variável.
STORE	!	Utilizado para inserir o valor no topo da stack em uma variável.

Tabela 10: Tokens dos condicionais

Nome do símbolo	Exemplo	Utilização
ID	exemplo1234	Utilizado na chamada de funções e nome de variáveis.

Tabela 11: Outros tokens

Para a identificação dos vários tokens foram utilizados lookaheads e lookbehinds, isto porque surgiram vários conflitos na identificação dos tokens. Tokens como ZEROEQ e NUM possuíam conflitos, isto porque o 0 utilizado na representação do ZEROEQ (0=) era reconhecido pelo NUM, sendo que existiam vários outros casos em que os tokens eram indevidamente reconhecidos.

Também ocorreu o conflito com o token ID que reconhecia vários dos tokens que eram compostos apenas por letras, tal como MOD por exemplo. Para resolver este problema foi criado um set de tokens reservados cuja chave seria o próprio token. A função responsável pelo reconhecimento do token ID verifica o texto reconhecido no set e atribui ao token o tipo apropriado.

Há tokens que possuem um valor atribuído, um deles é o token NUM que assim que é reconhecido é atribuído ao mesmo o seu valor sobre forma de inteiro ao invés de string. O valor do token CHARLETRA por exemplo possui como valor o carácter inserido. O token STRPRINT possui como valor o texto entre aspas e o STRPRINT2 possui como valor o texto entre parêntesis, removendo espaços repetidos.

Em relação à regra `t_ignore`, foi inserido o carácter `\r`, pois este carácter causa problemas de parsing em sistemas Windows.

Foi também criado um testador para o analisador léxico, o lexTester.py. Este programa possui um funcionamento simples, o testador lê do stdin o input do utilizador e utiliza o lexer criado pelo anaLex.py para reconhecer os vários tokens.

2.2 Analisador sintático (anaSin.py)

2.2.1 Introdução

Em relação ao analisador sintático utilizamos a biblioteca PLY com recurso ao yacc. Para o reconhecimento dos vários tokens utilizamos o lexer criado pelo anaLex.py.

Numa fase inicial foi decidido criar uma gramática simples que permitia o reconhecimento dos vários tokens de forma sequencial.

```
cont -> op cont
     |

op -> MAIS
    | MENOS
    (Restantes tokens...)
```

Para a implementação da gramática foi decidido que a forma mais eficiente de gerar o resultado final seria associar cada regra gramatical ao seu valor na máquina virtual, isto sobre forma de string. Por exemplo: para a regra gramatical do token MAIS iria possuir como valor "add". Para as regras mais gerais como o 'cont' o seu valor seria o resultado das restantes regras que o compõe, no caso seria p[0] = p[1] + p[2].

As várias regras gramaticais também contêm um print que identifica a mesma, este print apenas é realizado quando o modo debug se encontra ativo, permitindo assim a visualização da travessia pelas várias regras gramaticais.

Assim que os vários tokens são reconhecidos vai sendo criada uma string que representa o resultado final a ser introduzido na EWVM. Para facilitar a inserção do resultado na EWVM sempre que o resultado final é gerado este é guardado automaticamente em uma pasta '/results', isto porque a EWVM permite a inserção de ficheiros de texto como anexo.

O armazenamento do resultado na pasta '/results' é garantido pois sempre que o processo de armazenamento é iniciado é realizada uma verificação à existência da mesma, caso não exista esta será gerada.

2.2.2 Tradução dos comandos para EWVM

O valor da generalidade das regras gramaticais que identificavam um único token foram traduzidas de forma relativamente direta pois o seu funcionamento correspondente já era espelhado na máquina virtual por um determinado comando, no entanto houve certos tokens que exigiram um pouco mais de trabalho. O valor da regra gramatical que permitia identificar o token 1SUB, por exemplo, não poderia ser traduzido em um único comando na máquina virtual, sendo que a sua funcionalidade foi replicada com recurso aos comandos "pushi 1"e "sub".

Este processo de tentar replicar o funcionamento de um comando em forth numa sequência de comandos da EWVM foi um processo longo que abrangeu uma quantidade razoável de regras, no entanto houve determinadas regras que exigiram um esforço extra para a sua tradução. Utilizando o comando MIN do forth por exemplo possuía o desafio do resultado poder ser 2 valores diferentes, dependendo dos valores que se encontram na stack, isto porque o comando MIN devolve o menor elemento de entre os 2 valores que se encontram no topo da stack.

Para a resolução deste problema decidimos utilizar labels e os saltos existentes na máquina virtual. O funcionamento do comando MIN na máquina virtual foi definido: Chamar uma label que irá realizar a comparação dos dois valores no topo da stack, dependendo do resultado obtido na comparação a label irá inserir na stack o valor da 2ª posição a partir do topo da stack caso este seja o menor valor ou irá saltar para uma outra label (else) que irá inserir na stack o valor da 1ª posição a partir do topo da stack. Finalmente após a inserção do menor valor no topo da stack é necessário remover os dois valores originais da stack, utilizando swap e pop foi possível realizar esse processo.

Assim o resultado final do comando MIN possui 3 componentes: Código entre o start/stop de chamada, label if e label else.

Para armazenar as labels if e else do MIN foi criado um ficheiro na pasta recursos chamado funções.txt que irá conter todas as labels utilizadas na tradução dos comandos forth. O conteúdo deste ficheiro é sempre carregado após a criação do resultado final por parte do parser e o seu conteúdo é inserido no final do resultado gerado. Nota: a estas funções decidimos apelidar de "funções do sistema".

Existe também um outro conjunto de comandos que exigiu um pouco de trabalho extra na sua tradução, estes apresentavam um desafio diferente do comando MIN. O comando ROT do forth permite colocar o 3^{o} elemento da stack no topo. Não existindo uma operação na máquina virtual que realiza esta ação foi necessário utilizar os comandos existentes para replicar o seu funcionamento. Para replicar o funcionamento deste comando foram utilizadas variáveis globais na máquina virtual para armazenar os 3 valores na stack utilizado o comando storeg. O comando storeg dá pop ao elemento no topo da stack, assim que os 3 valores são armazenados estes são removidos da stack, permitindo a sua reorganização com pushg posteriormente.

Como foram utilizadas variáveis globais, estas precisam de ser declaradas no inicio do programa com o comando "pushi 0"antes do start do programa. Então para este efeito foi criada a função criarMemoria que gera automaticamente uma string com os 'pushi 0' necessários para o funcionamento de todas as operações, sendo que o resultado desta função está sempre presente no resultado final do programa. A variável memSys decide a quantidade de pushi 0 a ser realizados.

2.2.3 Funções

Em uma fase inicial decidimos utilizar a seguinte regra gramatical para representar funções:

```
op : COLON ID cont SEMICOLON
```

Esta regra é composta pelo token que representa o inicio da função 'COLON', o token que representa o nome da função 'ID', a regra que contêm o conteúdo da função 'cont' e o token que representa o final da função 'SEMICOLON'.

A regra criada para as funções possuía o funcionamento desejado sem gerar qualquer tipo de conflitos, no entanto permitia a definição de funções dentro de funções. Foram realizadas inúmeras alterações à gramática em uma tentativa de resolver este problema, sendo que na maioria delas tentamos manter a gramática o mais pequena e simples possível, no entanto, em cada tentativa surgiam inúmeros conflitos, até que foi decidido que a melhor maneira de resolver o problema seria criar uma regra 'cont' (conteúdo) que seria utilizada apenas pela regra que define a criação de funções, este 'cont' foi apelidado de 'contfunc' e contêm exatamente as mesmas operações que a regra 'cont' mas sem a operação que permite criar funções.

A regra final para a criação de funções passou a ser a seguinte:

```
'createfunc : COLON ID contfunc SEMICOLON'
```

Como se pode verificar, a regra para a criação de funções deixou de pertencer ao 'op' e passou a pertencer a 'createfunc', pois era necessário distinguir a criação de funções das restantes operações para permitir a sua remoção no 'contfunc'. O funcionamento de toda a gramática final e as alterações realizadas com as funções será explicada posteriormente.

Um outro problema que surgiu com as funções foi que na tentativa de utilizar labels para a definição de funções, enfrentou-se o problema das operações do

conteúdo da função não funcionarem como esperado, isto porque ao utilizar labels não é possível aceder aos valores que se encontram na stack principal.

Chegou se à conclusão que uma forma de resolver o problema era definir uma label para cada uma das operações do forth, sendo que essa label iria ser responsável por extrair da stack principal (com load) os valores necessários para realizar a operação. No entanto esta solução possuía o problema de os valores utilizados pela operação não serem removidos da stack principal. Uma solução ponderada para resolver o problema era a utilização de pops após a chamada da label, no entanto não era possível realizar estes pops sem haver o perigo de remover valores que eram resultado de outras operações.

Então decidiu-se criar um dicionário em que cada chave seria o nome da função e o valor seria o conteúdo dessa mesma função, assim que essa função fosse utilizada apenas era necessário realizar uma procura pelo nome da função e injetar o seu valor no resultado final.

Finalmente com as funções a funcionar como esperado apenas restava impedir a criação de funções com nomes duplicados. Para isso foram criadas duas listas, uma lista contém o nome de todas as labels utilizadas pelo sistema, já a outra contém o nome de todas as funções definidas pelo utilizador. Quando é definida uma função é verificado se o nome dessa mesma função em ambas as listas, sendo emitido um alerta caso se encontre repetida.

2.2.4 Ciclos

Os ciclos em forth funcionam lendo os dois valores no topo da stack, esses valores representam o valor inicial do contador e o limite superior do mesmo. A regra gramatical criada para reconhecer ciclos foi a seguinte:

```
'cicle : DO cont LOOP'
```

O 'DO' representa o inicio do ciclo, 'cont' representa o conteúdo do ciclo já o 'LOOP' delimita o fim do conteúdo do ciclo.

Para replicar o funcionamento dos ciclos do forth na máquina virtual foram utilizadas labels e o comando ALLOC. As labels são obrigatórias na EWVM para a criação de ciclos, pois apenas é possível repetir uma secção de código N vezes utilizando as labels e os saltos condicionais.

Em termos de funcionamento a estrutura é a seguinte: chamada para a label DO, na label DO é realizado o alocamento de 2 espaços para os dois valores no topo da stack, já que estes representam o contador e o limite superior do ciclo iremos necessitar destes valores várias vezes. Após este armazenamento é realizado um salto para a label doLoop.

Dentro do ciclo doLoop é realizado o carregamento do contador e do limite superior a partir do espaço alocado, depois é realizada uma comparação de inferioridade entre o contador e o limite superior, isto é, se o contador superar o limite superior então será realizado um salto para a label endLoop que irá terminar o ciclo. Se o contador for inferior ao limite superior então serão executadas as várias operações no interior do ciclo, e sucessivamente será realizada

a atualização do contador e um salto novamente para a label doLoop criando assim um ciclo que irá ser executado N vezes.

No entanto surgiu um problema, a operação I que obtém o valor do contador interno do ciclo parecia ser relativamente simples de implementar utilizando um PUSHST seguido de um LOAD, no entanto era necessário como argumento no PUSHST o número da frame utilizada pelo 'DO' atual. No entanto como o I encontra-se na regra 'cont' da gramática isso significa que o I não consegue saber qual é a frame do 'DO' atual. Para resolver este problema poderia-se ter utilizado arvores abstratas mas foi possivel contornar o problema utilizando uma variável global (posição 6) que iria possuir sempre o valor do contador interno do ciclo atual, assim sempre que uma operação I é realizada apenas é necessário um pushg 6 para obter o valor do contador.

2.2.5 Variáveis

Em relação as variáveis criamos a seguinte regra gramatical para o seu reconhecimento:

```
'vars : VARIABLE ID'
```

Para as variáveis decidimos utilizar variáveis globais, pois utilizando ALLOC iríamos ter de evitar possíveis conflitos com o funcionamento dos ciclos. É utilizado um map para saber a posição da variável global, a chave do map é o nome da variável e o seu valor é a sua posição.

Assim que o resultado final do parser é gerado é criada a quantidade de pushi 0 necessários para representar todas as variáveis que foram criadas ao longo do código.

Assim que é necessário dar fetch ao valor de uma variável global apenas é realizado um PUSHG da posição da variável, e quando se deseja guardar um valor nessa mesma variável realiza-se um STOREG na posição adequada.

2.2.6 Gramática final

Na primeira secção foi apresentada a primeira versão da gramática, e nas secções que sucederam esta foi explicado todo o processo de desenvolvimento do analisador sintático. No entanto ao longo do desenvolvimento do projeto foram realizadas algumas modificações á gramática original. Uma dessas modificações já foi explicada na secção referente as funções, sendo que foi necessário criar uma regra especifica para o conteúdo de uma função e a própria sendo que a criação de funções iria possuir a sua própria regra chamada createfunc.

No entanto ao longo do projeto decidiu-se que seria boa ideia alterar o funcionamento da regra 'cont' e fazer com que esta funcionasse com recursividade a esquerda. Então surgiu a seguinte alteração:

Antes

```
cont -> op cont
      | E
```

Depois

```
cont -> cont op
      | E
```

Uma outra alteração realizada na gramática foi a divisão das várias operações por tipos. Na gramática original possuíamos uma sequência de operações em que todas se encontravam aglomeradas sobre 'op' (operações). A divisão por tipos foi realizada para facilitar a leitura da gramática e fazer com que esta ficasse mais organizada.

Gramática:

```
cont -> cont type  (Regra para conteúdo)
      | E          (Regra para vazio)

type -> arit       (Regra para operações aritméticas)
      | comment    (Regra para comentários)
      | createfunc (Regra para criação de funções)
      | callfunc   (Regra para chamada de funções)
      | stack      (Regra para operações de stack)
      | input      (Regra para operações de input)
      | compare    (Regra para operações de comparação)
      | print      (Regra para operações de print)
      | cond       (Regra para condicionais)
      | cicle      (Regra para ciclos)
      | vars       (Regra para variáveis)

-------------------------------------------------------------------------------

contfunc -> contfunc cftype  (Regra para conteúdo de função)
        | E                  (Regra para vazio)

cftype -> arit       (Regra para operações aritméticas)
        | comment    (Regra para comentários)
        | callfunc   (Regra para chamada de funções)
        | stack      (Regra para operações de stack)
        | input      (Regra para operações de input)
        | compare    (Regra para operações de comparação)
        | print      (Regra para operações de print)
        | cond       (Regra para condicionais)
        | cicle      (Regra para ciclos)
        | vars       (Regra para variáveis)

-------------------------------------------------------------------------------

arit -> MAIS    (Soma os dois valores no topo)
      | MENOS   (Subtrai os dois valores no topo)
      | MUL     (Multiplica os dois valores no topo)
      | DIV     (Divide os dois valores no topo)
      | MOD     (Resto da divisão inteira entre os dois valores no topo)
```

```
      | ABS     (Insere o absoluto do valor do topo da stack)
      | 1SUM    (Soma 1 ao valor no topo da stack)
      | 1SUB    (Subtrai 1 ao valor no topo da stack)
      | 2DIV    (Divide o valor no topo da stack por 2)

comment -> COMMENT     (Comentário geral)
         | ENDCOMMENT  (Comentário de linha)

createfunc -> COLON ID contfunc SEMICOLON  (Criar função)

callfunc -> ID  (Chamar função)

stack -> NUM    (Insere um número na stack)
       | CHAR   (Inserir letra na stack sobre forma ASCII)
       | DROP   (Retira o primeiro elemento da stack)
       | DUP    (Duplicar valor no topo da stack)
       | SWAP   (Troca a posição dois valores no topo)
       | NIP    (Remove o segundo item da stack)
       | 2DROP  (Remove os dois elementos no topo da stack)
       | ROT    (Coloca o terceiro valor no topo)
       | OVER   (Copia o segundo valor e coloca no topo)
       | 2DUP   (Duplica o par no topo da stack)
       | 2SWAP  (Troca os dois pares no topo da stack)
       | 2OVER  (Copia o 2o par no topo da stack e cola no topo)
       | TUCK   (Insere uma copia do primeiro elemento debaixo do segundo)

input -> KEY  Recebe como input um caracter/tecla e coloca no topo da stack.

compare -> EQ         (Comparação de igualdade entre as 2 primeira no topo da stack)
         | NEQ        (Comparação de desigualdade entre os 2 valores no topo da stack)
         | MENOR      (Comparação menor entre os 2 valores no topo da stack)
         | MAIOR      (Comparação maior entre os 2 valores no topo da stack)
         | MENOREQ    (Comparação menor ou igual entre os 2 valores no topo da stack)
         | MAIOREQ    (Comparação maior entre os 2 valores no topo da stack)
         | ZEROEQ     (Comparação de igualdade entre 0 e o topo da stack)
         | ZERONEQ    (Comparação de desigualdade entre 0 e o topo da stack)
         | ZEROMENOR  (Comparação de menor ou igual entre 0 e o valor no topo da stack)
         | ZEROMAIOR  (Comparação de maior ou igual entre 0 e o valor no topo da stack)
         | NEGATE     (Nega o valor no topo da stack)
         | MIN        (Menor dos dois valores do topo da stack)
         | MAX        (Maior dos dois valores do topo da stack)

print -> PONTO      (Print do valor no topo da stack)
       | EMIT       (Print do caracter ASCII no topo da stack)
       | STRPRINT   (Print de uma string)
       | STRPRINT2  (Print de uma string removendo espaços consecutivos)
```

```
       | SPACES     (Print a N quantidade de espaços pelo valor no topo da stack)
       | SPACE      (Print de um espaço)
       | CR         (Print de um new-line \n)

cond -> IF cont ELSE cont THEN  (Condicional com if else e then)
      | IF cont THEN cont       (Condicional com if e then)

cicle -> I             (Obter valor do contador do ciclo)
       | DO cont LOOP  (Ciclo)

vars -> VARIABLE ID    (Criação de variáveis)
      | ID FETCH       (Obter valor de variável)
      | ID STORE       (Guardar o valor de variável)
```

2.3 Utilização dos programas criados

O analisador léxico criado pode ser utilizado através da aplicação (anaTester.py). Assim que iniciado é possível inserir input para o programa através do teclado, mas caso necessário, o utilizador pode inserir input a partir de um ficheiro tal como demonstrado a seguir:

`Inserção de input através do teclado: python3 lexTester.py`

`Inserção de input através de um ficheiro: python3 lexTester.py < teste.fs`

Em relação ao analisador sintático pode ser utilizado através da aplicação (anaSin.py). Assim que iniciado, é possível inserir input para o programa através do teclado e de um ficheiro à semelhança do (anaTester.py), a seguir encontram-se alguns exemplos da sua utilização:

`Inserção de input através do teclado: python3 anaSin.py`

`Inserção de input através de um ficheiro: python3 anaSin.py < ficheiro.fs`

O analisador sintático possui um modo debug que revela várias informações ao longo da execução do programa:

- Travessia realizada através das regras gramaticais
- Indicação dos nomes reservados para as funções do sistema
- Indicação da quantidade de memória a alocar para o sistema (pushi 0 antes do start)
- Indicação da pasta onde o resultado será armazenado
- Indicação do ficheiro que contém as funções do sistema

O modo debug pode ser ativado utilizando uma flag na chamada do programa:

`Modo debug: python3 anaSin.py debug`

`Modo debug:  python3 anaSin.py debug < ficheiro.fs`

Embora nestes exemplos se tenha utilizado ficheiros com extensão .fs são também suportados ficheiros com extensão .txt.

2.4 Exemplos de utilização

Para além dos exemplos que serão demonstrados a seguir existe uma pasta (`/inputs_teste`) na diretoria principal do projeto que contém vários outros testes que poderão ser utilizados para testar os programas criados.

2.5 Exemplos de utilização do analisador léxico:

2.5.1 Exemplo 1

Input:

```
1 2 +
: func ." teste" ;
do 1 loop
```

Output:

```
LexToken(NUM,1,1,0)
LexToken(NUM,2,1,2)
LexToken(MAIS,'+',1,4)
LexToken(COLON,':',1,0)
LexToken(ID,'func',1,2)
LexToken(STRPRINT,'teste',1,7)
LexToken(SEMICOLON,';',1,17)
LexToken(DO,'do',1,0)
LexToken(NUM,1,1,3)
LexToken(LOOP,'loop',1,5)
```

2.6 Exemplos de utilização do analisador sintático:

Nota: O texto (...) acima do start representa os vários pushi 0 dados pelo programa para alocar espaço para as as funções do sistema. Já o texto abaixo representa as várias funções do sistema. Foram retiradas dos exemplos para simplicidade de leitura já que não influenciam estes resultados em particular.

2.6.1 Exemplo 1

Input anaSin:

```
1 2 +
```

Output anaSin:

```
(...)
start
PUSHI 1
PUSHI 2
ADD
WRITEI
PUSHS " "
WRITES
stop
(...)
```

Output EWVM:

```
3
```

2.6.2 Exemplo 2

Input anaSin:

```
: teste 1 ;
: teste2 teste 2 + ;
: teste3 teste teste2 + ;
: teste4 teste2 teste3 + ;
teste4 .
```

Output anaSin:

```
(...)
start
// Função teste4
// Função teste2
// Função teste
PUSHI 1
// Fim função teste
PUSHI 2
ADD
// Fim função teste2
// Função teste3
// Função teste
PUSHI 1
// Fim função teste
// Função teste2
// Função teste
PUSHI 1
// Fim função teste
PUSHI 2
ADD
// Fim função teste2
ADD
// Fim função teste3
ADD
// Fim função teste4
WRITEI
PUSHS " "
WRITES
stop
(...)
```

Output EWVM:

```
7
```

2.6.3 Exemplo 3

Input anaSin:

```
10 0 do ." ola" loop
```

Output anaSin:

```
(...)
start
PUSHI 10
PUSHI 0
PUSHA do0
CALL

do0:
ALLOC 2
PUSHFP
LOAD -1 // Guardar counter
STORE 0
PUSHST 0
PUSHFP
LOAD -2 // Guardar limite
STORE 1
PUSHA doLoop0
CALL
RETURN

doLoop0:
PUSHST 0
LOAD 0 // Carregar counter
DUP 1
STOREG 6 // Valor para I
PUSHST 0
LOAD 1 // Carregar limite
INF
JZ endLoop0
// Conteudo
PUSHS "ola"
WRITES
// Fim conteudo
PUSHST 0
PUSHST 0
LOAD 0 // Carregar counter
PUSHI 1
ADD
STORE 0 // Store Counter++
JUMP doLoop0
endLoop0:
stop
(...)
```

Output EWVM:

```
olaolaolaolaolaolaolaolaola
```

2.6.4 Exemplo 4

Input anaSin:

```
1 0 =
IF ." É zero! (Falha)" CR
ELSE ." Não é zero! (Sucesso)" CR
THEN ." Finish" CR
```

Output anaSin:

```
(...)
start
PUSHI 1
PUSHI 0
EQUAL
// Condicional 0
PUSHI 0
JZ if0

if0:
JZ else0
PUSHS "É zero! (Falha)"
WRITES
WRITELN
JUMP then0

else0:
PUSHS "Não é zero! (Sucesso)"
WRITES
WRITELN
JUMP then0

then0:

// Fim do condicional 0
PUSHS "Finish"
WRITES
WRITELN
stop
(...)
```

Output EWVM:

```
Não é zero! (Sucesso)
Finish
```

3 Conclusão

Neste projeto foi criado um compilador de forth em python com recurso a biblioteca PLY, foi explicado todo o processo de desenvolvimento realizado pelo grupo incluindo todas as decisões e desafios enfrentados.

Durante o desenvolvimento do projeto conseguimos colocar à prova os conhecimentos aprendidos durante as aulas práticas e teóricas.

Aprendemos todo o processo de criação de um compilador desde a criação de um analisador léxico que reconhece os vários tokens utilizados por uma linguagem de programação até a criação de uma gramática sofisticada para um analisador LALR(1). O projeto colocou a prova a nosso raciocínio lógico enfrentando vários problemas cuja resolução envolveu criatividade e muita pesquisa.

De um modo geral completamos todas as tarefas esperadas para este projeto sendo o compilador criado é rico em funcionalidades e permite traduzir uma boa percentagem de toda a linguagem forth para a maquina virtual.

Referências

[1] Brodie, L. (1987). *Starting Forth*. Prentice-Hall. `https://www.forth.com/starting-forth/`

[2] Free Software Foundation, Inc *Gforth Manual*. GNU Project. `https://gforth.org/manual/`

[3] Stephen Pelc *Programming Forth*. MicroProcessor Engineering Limited `https://vfxforth.com/arena/ProgramForth.pdf`

[4] Educational Web Virtual Machine. `https://github.com/jcramalho/EWVM/`