Processamento de Linguagens

Trabalho Prático

Relatório de Desenvolvimento

José Pereira
(A 89596)

Sérgio Costa
(A 81215)

8 de maio de 2024

Resumo

O documento visa à documentação sucinta do desenvolvimento de um compilador.
O compilador foi preparado para processar a linguagem de programação Forth e posteriormente traduzi-la para código máquina (assembly).
O código máquina diz respeito à máquina virtual desenvolvida pela casa, de nome EWVM.
O documento contempla ainda uma secção que respeita às aprendizagens adquiridas no curso do desenvolvimento de um compilador.
Tais aprendizagens poupar-nos-hão tempo -futuro- no que ao trabalho nesta área diz respeito.

Conteúdo

1 Introdução **3**

1.1 FORTH . . . 3
1.2 Expressões aritmáticas . . . 4
1.3 Strings e Suporte à emissão de Caracteres . . . 5
1.4 Funções . . . 5
1.5 Condições . . . 5
1.6 Ciclos . . . 5
1.7 Variáveis . . . 6

2 Análise e Especificação **7**

2.1 Descrição informal do problema . . . 7
2.2 Especificação do Requisitos . . . 7
2.2.1 Dados . . . 7
2.2.2 Pedidos . . . 7
2.2.3 Relações . . . 8

3 Concepção/desenho da Resolução **9**

3.1 Estruturas de Dados . . . 9
3.2 Algoritmos . . . 10
3.2.1 Aritmética . . . 13
3.2.2 Funções . . . 13
3.2.3 Saltos Condicionais . . . 13
3.2.4 Ciclos . . . 14
3.2.5 Variáveis . . . 16
3.2.6 Arrays . . . 16

4 Codificação e Testes **17**

4.1 Alternativas, Decisões e Problemas de Implementação . . . 17
4.1.1 Stack Frame . . . 17
4.1.2 Ciclos . . . 18
4.1.3 Expressões Regulares . . . 18
4.2 Testes realizados e Resultados . . . 18
4.2.1 N Push . . . 18

4.2.2 Tabuada (Erro) 19
4.2.3 Tabuada 20
4.2.4 Congruência Módulo 2 22
4.2.5 Divisão por zero? 24
4.2.6 Divisão por zero? (Pro Forth) 25
4.2.7 Eggsize 25

5 Conclusão **27**

Capítulo 1

Introdução

O grupo é constituído por dois alunos frequentadores do 3° ano da Licenciatura em Engenharia Informática, da Universidade do Minho, campus de Gualtar.
O tema que nos foi proposto advém do âmbito da unidade curricular de Processamento de Linguagens, unidade curricular incorporada no plano de estudos do último semestre do último ano da Licenciatura em Engenharia Informática.
A unidade curricular contempla a leitura, validação, interpretação e tradução de texto.
Inúmeras áreas recorrem aos fundamentos desta unidade curricular para o desenvolvimento de vários tipos de ferramentas. Os princípios estudados nesta unidade curricular contemplam vastos propósitos, desde o processamento de linguagem natural, ao próprio processamento de linguagens máquina, tudo, sob a alçada dos mesmos princípios teóricos.
No âmbito da unidade curricular, foi-nos então pedido que desenvolvêssemos um compilador de uma linguagem de programação, a FORTH. Ao longo deste documento, abordamos a conceção de uma ferramenta capaz de converter a linguagem (de alto nível) da programação via FORTH, numa linguagem de baixo nível (assembly), direcionada à máquina virtual desenvolvida na casa [1].
Com este relatório, o grupo visa contemplar a elaboração de uma solução suportada através dos princípios teórcos lecionados a par na unidade curricular de Processamento de Linguagens. Assim, o nosso principal objetivo com este relatório é o de colocar o leitor a par dos princípios subjacentes às definições adotadas (no que toca à linguagem *per se* respetivamente), bem como, às ferramentas utilizadas e encadeamentos lógicos adotados para produzir o resultado final.
Os resultados em causa visam a geração de instruções assembly capazes de produzir as computações especificadas através da linugagem (de programação) FORTH.
O documento é estruturado mediante a resolução de cada um dos problemas resultantes da divisão do problema de interpretação e tradução como um todo. Assim, encontram-se subsequentes os capítulos relativos à resolução da leitura, interpretação e tradução de: (1) expressões aritméticas, (2) definição de funções, (3) impressão de caracteres e strings, (4) fluxos lógicos derivados de condições, (5) ciclos iterativos e (6) declaração e uso de variáveis.

1.1 FORTH

O Forth é uma ferramenta que se divide em dois âmbitos: (i) interpretação e (ii) compilação. Neste projeto trataremos de (ii).
A proposta de desenvolvimento do compilador pela equipa docente segue a seguinte divisão de trabalho

[1] `https://ewvm.epl.di.uminho.pt/`

e esforços: (1) suporte à compilação de expressões aritméticas, (2) suporte à compilação de funções e uso destas, (3) suporte à compilação de strings, (4) suporte à compilação de condicionais, (5) suporte à compilação de ciclos e (6) suporte à compilação e uso de variáveis.
O resultado da compilação de um programa escrito em Forth contempla a geração de código assembly da Máquina Virtual desenvolvida no âmbito de um projeto da casa[2].
Nas próximas secções enumeraremos nuances a ter em conta num programa Forth relativas a cada uma das funcionalidades supra mencionadas.

1.2 Expressões aritmáticas

Um programa FORTH pode consisitir exclusivamente na declaração de expressões aritméticas.
As expressões aritméticas são trabalhadas através de uma notação pós fixa[3]. Assim, tanto podemos representar $5+5+5+5$ como 5 5 5 5 + + + e como 5 5 + 5 + 5 +. O importante é que, conscientes daquilo que estamos a fazer, existam dois operandos em memória para cada uma das operações de soma a executar.

Existem ainda duas nuances no que à aritmética em Forth diz respeito: (1) os números tratam-se exclusivamente de inteiros, o que implica a truncação, ou arredondamento[4], no caso de um número fraccionário originado pela divisão de dois números, p.e. 1 4 /, onde se lê 1/4, que diz respeito a 0.25, donde se transforma num inteiro 0 face à truncação da sua composição decimal e (2) o primeiro operando da operação diz respeito aquele que foi introduzido em primeiro lugar.

Stack A aritmética em Forth é apoiada através de uma stack. Infra ilustramos o seu conteúdo no cálculo de 1 4 −.

1

Tabela 1.1: Leitura do inteiro 1 e armazenamento na stack

4
1

Tabela 1.2: Leitura do inteiro 4 e armazenamento na stack

-3

Tabela 1.3: Retirada dos inteiros, operação de $1-4$ e armazenamento do resultado na stack

[2] `https://ewvm.epl.di.uminho.pt/`

[3] `https://www.forth.com/starting-forth/2-stack-manipulation-operators-arithmetic/`

[4] No ambiente de desenvolvimento do projeto, a nossa máquina virtual está munida com um comando de conversão de um real para um inteiro através da truncação da sua parte decimal. Por questões de simplicidade faremos uso desta estratégia aquando da necessidade de conversão (fruto de uma operação de divisão) de um resultado real para um resultado inteiro.

1.3 Strings e Suporte à emissão de Caracteres

Em Forth, é possível a utilização de caracteres de duas maneiras: (1) guardar um-a-um ou (2) emitir diretamente a sua sequência.

Para proceder com (1) existem duas possibilidades: (a) gravar diretamente o código ASCII do caracter ou (b) realizar uma conversão, dado um caracter, converter e guardar o seu respetivo código inteiro. O armazenamento é direcionado à Stack. Para proceder a (a) realizamos a sequência habitual para o tratamento de um inteiro, para procedermos a (b) realizamos uma conversão explícita (`[CHAR]`), donde no caso de "`[CHAR] *`"será então colocado em stack o respetivo código ASCII (42).

Para realizar a emissão de um caracter recorre-se ao comando `EMIT`, o mesmo pega no operando do topo da pilha - stack - e realzia a sua impressão caracteriana. Não obstante, existe a operação de POP (`.`) onde é tido o mesmo procedimento, mas o resultado é a sua impressão inteira.

Ambas as operações necessitam de um elemento - operando - em stack, e ao utilizá-lo, este é descartado.

No caso de se tratar de uma string *per se*, em Forth trabalhamos a mesma dentro do protocolo sintático `."TEXTO"`, e, sempre que esta sequência aparecer, é imediatamente emitida em ecrã.

1.4 Funções

Em Forth, uma definição é feita através da inicialização (`:`). Em seguida, vem o nome que queremos dar à função (`: STAR`). Posteriormente vem o corpo da função e encerramos a sua definição com (`;`), donde resultaria: `: STAR 42 EMIT ;`.

Não podem existir definições de funções dentro de definições.

Aquando de uma função definida, esta entra num glossário próprio, onde posteriormente é recorrido aquando da sua chamada.

1.5 Condições

As condições em Forth utilizam-se, exclusivamente, dentro da definição de uma função.

A sequência condicional diz-nos que, no momento de decisão, a stack necessita de pelo menos 1 operando. Antes disso é um jogo de `0` e diferente de zero (`!0`) que determina a veracidade ou a falsidade lógica, respetivamente. Para atingir esses efeitos, existe a possibilidade da combinação relacional dos operandos, onde, são permitidos os seguintes testes relacionais: igualdade (`=`), diferença (`<>`), inferioridade (`<`), superioridade (`>`), igualdade ao zero (`0=`), negatividade (`0<`) e positividade (`0>`).

Os saltos são realizados de duas maneiras: `IF ... THEN` ou `IF ... ELSE ... THEN`.

No primeiro salto, é testado se o operando no topo da stack é diferente de 0, ao sê-lo, é considerada uma veracidade lógica e são executadas as instruções imediatamente a seguir ao `IF`. Por fim, é executado o `THEN`. Este é executado sempre, independentemente dos resultados lógicos do salto condicional.

No segundo salto, existe a alternativa das instruções a executar. Não obstante, o "THEN"executa-se sempre.

1.6 Ciclos

Os ciclos em Forth respeitam 4 variantes: (1) DO ... LOOP, (2) DO ... +LOOP, (3) BEGIN ... UNTIL e (4) BEGIN ... WHILE ... REPEAT. E, tal como as condições, só podem ser definidas dentro de uma função.

Em (1), é esperado no topo da stack o limite superior e o índice inicial, por esta ordem. A condição testada é se o índice inicial é inferior ao limite superior, e, ao sê-lo é executado o corpo do ciclo (instruções posteriores

ao DO). No fim, incrementa-se o índice uma unidade, e volta-se para o início do ciclo. Quando este termina, são executadas as intruções imediatamente a seguir ao LOOP.

Em (2), é tudo igual ao (1), com a nuance de que este espera 3 operandos em stack, sendo que, o último dos operandos deve indicar o valor a ser incrementado ao índice no final de cada ciclo.

Em (3), o ciclo é executado até se verificar que o operando indicia a falsidade lógica da condição. A condição é imposta antes do bloco de código UNTIL, e, ao verificar-se a sua veracidade lógica, as instruções imediatamente a seguir ao BEGIN voltam a ser executadas.

Em (4), as instruções subsequentes ao BEGIN são executadas pelo menos uma vez, onde, posteriormente se testa a veracidade lógica presente no topo da stack. Ao verificar-se, é executado o código a seguir ao WHILE, e re-executadas as intruções a seguir ao BEGIN. Em caso de falsidade, executam-se as intruções imediatamente a seguir ao REPEAT.

1.7 Variáveis

Em Forth as variáveis, antes de usadas precisam de ser declaras. Para declarar uma variável instruímos o compilador da seguinte maneira: "VARIABLE DATE". E vemos acrescentada uma nova variável ao programa (a variável DATE). Posto isto, podemos realizar 4 tipos de operações com a mesma: guardar um valor numérico, consultar o valor numérico, imprimir o valor numérico e somar um valor numérico.

Suponhamos então que temos a variável DATE definida.

Queremos guardar nela o valor do dia de hoje (4), então fazemos: 4 DATE !.

Queremos utilizar o seu valor numa operação (p.e.), fazemos: DATE @.

Queremos imprimir o seu conteúdo: DATE ?.

Queremos somar-lhe um valor: 1 DATE +!.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Existe uma linguagem de programação, o Forth, onde nos vamos basear para gerar - traduzir - programas para a máquina virtual da casa[1].

O objetivo consiste na interpretação e reconhecimento de alto nível das instruções escritas em Forth, para a sua tradução em código máquina para uma arquitetura muito específica.

2.2 Especificação do Requisitos

A tradução da linguagem deve conseguir incorporar cálculos aritméticos, impressão de strings e caracteres, definição de funções, suporte a saltos condicionais, suporte a ciclos e suporte a variáveis.

2.2.1 Dados

Forth O Forth é uma linguagem que trabalha sobretudo com a manipulação direta da stack computacionária em mente. É por isso uma linguagem que requere um certo cuidado na forma como manuseia a stack principal do programa.

À parte disso, contamos com uma documentação sucinta da mesma em
`https://www.forth.com/starting-forth/`.

Máquina Virtual A máquina virtual trabalha com instruções assembly próprias, criadas e definidas no contexto específico da virtualização. O grupo considera um ótimo ponto de partida para os trabalhos compilatórios inerentes à arquitetura processual de cada um dos processadores no mercado.

A mesma conta com uma vasta e pertinente documentação online, que pode ser consultada em:
`https://ewvm.epl.di.uminho.pt/manual`.

2.2.2 Pedidos

É-nos pedido que utilizemos a ferramenta PLY [2] para suportar o processo de análise e tradução da linguagem Forth para a linguagem máquina da EWMV.

[1] `https://ewvm.epl.di.uminho.pt/`
[2] `https://www.dabeaz.com/ply/ply.html`

Através da ferramenta devemos definir uma gramática capaz de reconhecer o conteúdo da linguagem (independente de contexto), e que, através da programação em Pyhton, lhe atribuamos o seu próprio contexto.

2.2.3 Relações

A primeira relação inerente ao processo é o de reconhecimento de tokens (através de expressçoes regulares). Estes são definidos num módulo específico, onde, posteriormente, serão passados ao analisador gramatical que os vai analisar num contexto gramatical.

A gramática, por si só, é independente de contexto, donde, o mesmo deve ser suportado através de variáveis (e fluxos) lógicas.

Capítulo 3

Concepção/desenho da Resolução

3.1 Estruturas de Dados

Para este trabalho utilizamos duas estruturas de dados: uma árvore inerente ao parser (PLY) e um conjunto de variáveis que contemplam a contextualização do conteúdo em parsing.
Assim, o grupo fará uma breve descrição do intuito das mesmas. Infra tabeladas.

Variáveis de Contexto		
Nome	Tipo	Descrição
parser.exito	Bool	Tokens esperados.
parser.erro	Bool	Erro Forth.
parser.stack_construtor	[ ]	Controlo das definições em vigor.
parser.stack_else_labels	[ ]	Controlo dos saltos em vigor.
parser.stack_then_labels	[ ]	Controlo dos saltos/fluxos em vigor.
parser.stack_loop_labels	[ ]	Controlo dos saltos/ciclos em vigor.
parser.erros	[ ]	Acumulador de mensagens de erro/warnings.
parser.banco	Int	Controlo de custo de operações.
parser.divida	Int	Controlo da falta de operandos.
parser.banco_construtor	Int	Controlo no contexto de definição de função.
parser.divida_construtor	Int	Controlo no contexto de definição de função.
parser.label_else_n	Int	Identificador único a cada branch/label.
parser.label_then_n	Int	Identificador único a cada branch/label.
parser.label_loop_n	Int	Identificador único a cada branch/label.
parser.heap_alloc	Int	Controlo de alocações na heap (blocos) pela função.
parser.my_heap	Int	Controlo de blocos em vigor.
parser.if_flag	Bool	Flag de IF statement.
parser.do_flag	Bool	Flag de DO statement.
parser.untreaceble_keywords	[ ]	Definições não verificadas do n° de operandos em dívida/resultantes.
parser.custos	{ }	Custo das operações.
parser.dicionario	{ }	Resultado assembly das definições definidas.
parser.variaveis	{ }	Controlo das variáveis em memória: nome - posição.
parser.arrays	{ }	Controlo dos arrays em memória: nome - (posição, tamanho).
parser.compilacao	String	Assembly EWVM.

Tabela 3.1: Descrição das variáveis de contexto.

3.2 Algoritmos

O algoritmo subjacente à ferramenta respeita uma travessia pre-order relativa à construção bottom-up de uma árvore N-Ária que respeita a expanção e redução das regras gramaticais infra definidas.

```
Programa : Instrucoes

Instrucoes : Instrucoes Instrucao
Instrucoes : Instrucao

Instrucao : Operando
          | Operador
          | Keyword
          | Funcao
          | Condicao
          | Loop
          | Variavel
```

```
          | Store
          | Fetch
          | Show
          | Inc
          | Array
          | ArrayStore
          | ArrayFetch
          | ArrayShow
          | Comment

Operando : Int
         | Char
         | String
         | i
         | j
         | Cr

Int : INT
Char : CHAR
String : STRING
i : I
j : J
Cr : CR

Operador : Add
         | Sub
         | Mul
         | Div
         | Mod
         | Igual
         | Diferente
         | Menor
         | Superior
         | Zero_Igual
         | Negativo
         | Positivo
         | Ponto
         | Dup
         | Drop
         | Emit

Add : ADD
Sub : SUB
Mul : MUL
Div : DIV
Mod : MOD

Igual : IGUAL
```

```
Diferente : DIFERENTE
Menor : MENOR
Superior : SUPERIOR
Zero_Igual : ZERO_IGUAL
Negativo : NEGATIVO
Positivo : POSITIVO

Ponto : PONTO

Dup : DUP
Drop : DROP
Emit : EMIT

Keyword : KEYWORD

Funcao : Start_Func Instrucoes End_Func

Start_Func : START_FUNC KEYWORD

End_Func : END_FUNC

Condicao : If Instrucoes Then
If : IF
Then : THEN

Condicao : If Instrucoes Else Instrucoes Then
Else : ELSE

Loop : Do Instrucoes LOOP
Do : DO

Loop : Do Instrucoes INT_MAIS_LOOP
Loop : Do Instrucoes MAIS_LOOP

Loop : Begin Instrucoes UNTIL
Begin : BEGIN

Loop : Begin Instrucoes WHILE Instrucoes REPEAT

Variavel : VARIABLE
Store : STORE
Fetch : FETCH
Show : SHOW
Inc : INC

Array : ARRAY
ArrayStore : ARRAYSTORE
ArrayFetch : ARRAYFETCH
```

`ArrayShow : ARRAYSHOW`
`Comment : COMMENT`

3.2.1 Aritmética

A aritmética é suportada essencialmente pelas produções que dizem respeito ao operando Int, bem como a todas as derivações de Operador (essencialmente as aritméticas).

Na produção do token INT (`Int : INT`), existe o cuidado de criar um comando de push com a representação textual do inteiro em questão. Este comando segue a documentação da EWVM. O comando é passado via argumento `p[0]` de uma árvore de derivação gerida pelo Yac.

Para controlar futuros erros de código (e avisos), o grupo corre ainda uma função dentro desta produção que calcula os operandos atuais em Stack mediante o token em eminência. Esta função consegue distinguir se o push foi efetuado fora de uma função, ou dentro. O cálculo é efectuado através da consulta de uma tabela via dicionário com o nome do token a mapear o custo e produção de operandos pela operação em destaque, respetivamente.

Nas produções subsequentes às operações aritméticas, existe o cuidado de as transcrever - traduzir - para as suas homólogas na EWVM.

O grupo destaca o cuidado a ter na operação de divisão, que, a seguir a uma, acrescentamos a instrução de conversão para um inteiro (prevenindo assim o futuro rebentamento da Stack de operandos).

3.2.2 Funções

No momento em que o lexer distingue o token de início de definição ':', o parser fica a aguardar pelo nome da mesma (uma Keyword). Assim, assume uma redução na produção respetiva (`Start_Func : START_FUNC KEYWORD`).

Nesta redução é verificado o contexto desta definição. Ao inicializar-se dentro de uma já em ocorrência, é assinalada uma flag de erro, o erro é reportado e conduzido ao stock em `parser.erros`.

Ao correr bem, isto é, ao tratar-se da primeira definição em causa, são inicializados alguns parâmetros de contexto. Nomeadamente, o custo da operação, quantos operandos vão ficar em stack e quantos blocos serão alocados em heap.

Por questões de tratamento de erros, são ainda afixados 3 dados relativos à mesma: (1) nome, (2) linha e (3) coluna.

A partir daqui, existem vários fluxos de captura que levam a que existam várias reduções à produção de `Instrucoes : Instrucoes Instrucao` via `Funcao : Start_Func Instrucoes End_Func`.

Na produção de `End_Func : END_FUNC` faz-se a contextualização do token na sequência: em cima de um erro ou por cima de uma definição.

Em `Funcao : Start_Func Instrucoes End_Func` culmina-se a definição da função. A mesma é guardada num glossário de definições. O glossário é mapeado pelo nome da função.

3.2.3 Saltos Condicionais

Existem dois tipos de saltos condicionais: IF ... THEN e IF ... ELSE ... THEN.

As produções das respetivas reduções condicionais são "`Condicao : If Instrucoes Then`"e "`Condicao : If Instrucoes Else Instrucoes Then`", donde são processadas por duas produções comuns: "`If : IF`"e "`Then : THEN`".

Na produção If, é verificado o uso condicional dentro de uma função (exclusividade do seu uso em Forth), ativada a flag condicional e calculados os operandos necessários em stack para proceder com a operação

condicional.

Na produção Then, existe o cuidado da criação de uma label única inerente à condição. A label é colocada numa stack para que possam existir condições dentro de condições. A produção é responsável (ainda) pela etiquetação das instruções subsequentes (relativas aos tokens que venham ainda a ser capturados). Assim, é deixada a marca, nesta mesma produção, de onde começa o bloco de código respetivo à sequÇencia do bloco THEN.

Posteriormente, aquando da redução da produção da condicional em causa, é retirado do topo da pilha a respetiva label, donde, provirá a instrução de salto condicional (ou obrigatória) para a respetiva etiqueta.

A produção Else é análoga à produção Then.

IF ... THEN

A redução da condição respetiva é dada pela produção `Condicao : If Instrucoes Then`.

Aqui é retirada a etiqueta - pop - que diz respeito ao bloco the instruções a seguir ao THEN - instrução -, donde, ao verificar-se a falsidade da condição lógica inerente, é escrito - traduzido - um salto condicional para a etiqueta em questão.

Ao verificar-se a vericidade, este salto -isntrução EWVM- é ignorado e executa o bloco a seguir ao IF. No fim da execução é feito um salto (obrigatório) para a etiqueta de respeito às instruções a seguir ao THEN[1].

IF ... ELSE ... THEN

A redução da sequência dos tokens respetivos a este tipo de condição é dada pela produção
`Condicao : If Instrucoes Else Instrucoes Then`.

Aqui retiram-se duas etiquetas do topo de duas pilhas distintas: etiquetas then e etiquetas else, respetivamente. Assim, em caso de falsidade da condição, é efetuado um salto -jz- para a etiqueta de else, ao passo que, no fim da execução do bloco if, para saltar o bloco else, é realizado um salto -jump- obrigatório para a label do then[2].

3.2.4 Ciclos

Em Forth existem 4 tipos de ciclos: (1) DO ... LOOP, (2) DO ... +LOOP, (3) BEGIN ... UNTIL e (4) BEGIN ... WHILE ... REPEAT.

Para (1) e (2) existe a partilha de uma produção relativa ao tratamento do token DO. Ao passo que em (3) e (4), existe a partilha de uma produção relativa ao tratamento do token (comum) BEGIN.

O token DO é tratado pela produção `Do : DO` que contextualiza a gramática do seguinte: está eminente o tratamento de tokens respetivos a um ciclo -flag-, está eminente a alocação de um bloco em heap[3], cria-se e armazena-se a label (única) respetiva ao ciclo e ainda se calcula o custo (operandos necessários em stack) à instrução DO (2 no caso).

Em relação ao token BEGIN, o mesmo é tratado pela produção `Begin : BEGIN`, onde a mesma se encarrega (única e exclusivamente) de criar uma label única relativa ao ciclo em causa.

[1] De notar que em Forth, o THEN é sempre executado, independentemente da veracidade lógica da condição.

[2] Notem que, o bloco de instruções a seguir ao else é o bloco de instruções adjacente ao bloco then, pelo que o trabalho de correr as instruções subsequentes ao then fica (deste modo) realizado -tratado (de forma automática)-.

[3] Fruto da estratégia abordada para o tratamento de ciclos: opta-se por discernir em heap o índice inicial, o limite e o step -salto iterativo-.

DO ... LOOP

A redução de um ciclo DO ... LOOP é dada pela produção `Loop : Do Instrucoes LOOP`. A mesma encarrega-se de retirar da stack a label respetiva, e com ela criar 5 tipos de etiquetas diferentes, uma para representar as isntruções respetivas a cada componente do ciclo: inicialização, teste, corpo, incrementação -iteração- e fim.

Deste modo, as labels serão distribuídas conforme e entre cada produção. A inicialização (por motivos de inteligibilidade no assembly) denota o carregamento dos operandos que dizem respeito ao índice, ao limite e ao incremento, por esta ordem. A condição respeita o teste de se o índice é inferior ao limite (ao não se verificar, é dado o salto -jz- para o fim do ciclo). O corpo do ciclo denota (mais uma vez -etiquetado- por questões de inteligibilidade) as instruções do ciclo. A incrementação, também ela etiquetada por questões de inteligibilidade no assembly dizem respeito às operações de incremento do índice pelo valor definido (neste tipo de ciclos, 1). No fim da incrementação, é colocada uma instrução de salto obrigatório -jump- para a etiqueta que respeita as operações -instruções- de teste -verificação- de execução do ciclo. Por fim, deixa-se uma etiqueta a marcar o fim do ciclo como a última instrução assembly da produção. Assim, código subsequente pode ser montado posteriormente a esta etiqueta, donde resutlará o goto da não verificação da condição de execução do cilo (corpo e incrementação).

Relativamente à estratégia abordada pelo grupo, o grupo decidiu carregar (e trabalhar com eles) em heap os operandos que controlam o ciclo: índice, limite e salto.

Assim, acresce uma estratégia de contextualização ao bloco em heap no momento de montagem assembly (caso se chame o mesmo ciclo mais que uma vez -e em ordens diferentes-).

Deste modo, o grupo opta por criar uma linguagem intermédia que dirá respeito à posição relativa do bloco alocado em heap no momento de execução. Aquando da montagem assembly, é corrida uma função de substituição que realiza um parsing intermédio donde calculará (mediante as chamadas à heap contabilizadas até ao momento) o bloco da heap respetivo ao ciclo.

Relativamente ao facto de poderem existir ciclos dentro de ciclos, esse fator é contornado na adoção de uma sintáxe que capture essa mesma semântica (que é atribuída através do cálculo do comprimento da stack das labels correntes aos ciclos em tratamento), garantido por isso a respetiva profundidade (posição) relativa ao aninhamento -inclusão- de um ciclo dentro do outro: o ciclo exterior será tido no bloco x, o ciclo interior a esse no bloco $x+1$, e assim em diante...

DO ... LOOP ... +LOOP

A redução deste tipo de ciclos é análoga ao ciclo descrito na secção anterior, com a exceção de que, se o inteiro relativo ao incremento for dado dentro da definição, esta é carregada no momento da sua definição. Se o inteiro relativo ao incremento não for descriminado na definição, existe o cuidado do carregamento do mesmo através da sua posição relativa na stack.

Para tratar estes dois casos (descriminação do step *vs* não descriminação do step) o grupo opta por tratar os dois casos em diferentes produções: numa das reduções o tamanho do incremento é carregado através do operando no topo da stack, ao passo que na outra é descriminado através da instrução explícita e assim então carregado.

BEGIN ... UNTIL

Neste tipo de ciclos, a redução trata de montar as instruções do seguinte modo: etiquetar as intruções com a label respetiva (previamente retirada -pop- da stack de labels referentes a ciclos), colar as produções previas que dizem respeito ao corpo do ciclo, e testar se o operando (dita flag) é logicamente verdadeira (!=0) ou

logicamente falsa (=0). Ao tratar-se de um valor relativo à veracidade lógica, é encaminhada -jump- através da negação da igualdade a zero -jz- para o início do cíclo.

BEGIN ... WHILE ... REPEAT

Neste tipo de instruções a produção deve ter cuidado no encaminhamento de instruções relativas a partes do ciclo. Ao retirar a label da stack respetiva o grupo adota a criação posterior de 3 labels que dizem respeito às instruções a seguir ao BEGIN, às instruções a seguir ao WHILE e às instruções a seguir ao REPEAT. Assim, ao executar o primeiro bloco, é testada a veracidade lógica do operando no topo da stack e encaminhado -jz- para o fim do REPEAT -label-. Se o operando correpsonder à veracidade lógica, então segue o curso normal, donde, no fim do segundo statement existe um reencaminhamento (obrigatório) -jump- para o início do ciclo (BEGIN).
Por fim etiqueta-se (no fim da produção) com a etiqueta que diz respeito ao fim do ciclo. Desse modo, dizendo respeito às instruções subsequentes à produção em causa.

3.2.5 Variáveis

Em relação ao tratamento de variáveis existem um conjunto de produção que se encarregam do efeito, tais como: declaração, armazenamento na mesma, uso do conteúdo da mesma, amostragem do conteúdo da mesma e incremento de um inteiro arbitrário à própria.

Relativamente à declaração de uma variável, o grupo opta por definir uma produção que se encarrega de remover (caso exista) a definição do glossário de funções definidas pelo utilizador. Obrigatoriamente, nesta produção, realiza-se a tradução da alocação respetiva e contextualização via código python (armazenamento numa tabela que enquadra o seu nome como chave de acesso e o seu bloco respetivo, em heap). A produção contextualiza ainda a utilzação da heap como tendo sido utilizado um bloco da mesma.

Relativamente às operações restantes, estas seguem o mesmo procedimento antes da montagem do respetivo assembly EWVM: verificação da mesma na entrada -tabela- de variáveis, montagem do código -operações- de acesso à mesma, e derivação desta -amostragem, push para a stack, push de um operando em stack para o endereço em heap respetivo à variável ou incrementação do conteúdo da variável-.

3.2.6 Arrays

Relativamente às produções relativas à definição e manuseamento de um array, as mesmas são homólogas às produções relativas às Variáveis, com a nuance de que estes são armazenados numa tabela diferente (própria) que incute a informação do tamanho destes. Assim, aquando do acesso a uma posição destes, o compilador verifica se o offset indicada se encontra dentro dos respetivos limites.

Capítulo 4

Codificação e Testes

A codificação foi realizada através da ferramenta PLY, pelo que adotamos uma abordagem Python para a construção de toda a contextualização e posterior tradução.

Foram desenvolvidos 3 módulos que dizem respeito à análise lexicográfica -lexer-, à análise semântica -yacc- e à utilização (run) do compilador -main-.

Os testes foram desenvolvidos num editor de texto e posteriormente corridos via ferramenta -compilador- desenvolvido pelo grupo.

O compilador gera um ficheiro com o código assembly da EWVM, donde o transcrevemos (Ctrl-a, Ctrl-c e Ctrl-v) para a plataforma. Os pedidos de execução do código resultante foram realizados através da plataforma web.

4.1 Alternativas, Decisões e Problemas de Implementação

O grupo deparou-se com várias decisões, ainda numa parte embrionária do projeto, que levou a cabo até ao fim. Não obstante, de terem surgido alternativas, donde, por falta de tempo, não poderam ser postas em prática. Não obstante, deixamos aqui essas mesmas observações para que no futuro se possam medir as suas consequências desde cedo, permitindo uma melhor gestão de tempo e esforços (intelectuais, de experimentação, etc...).

4.1.1 Stack Frame

A primeira decisão que o grupo se viu envolvido a tomar foi à cerca do manusemanento das frames em stack: (1) se utilizar uma única, (2) se utilizar uma para cada chamada a cada uma das funções. E, apesar de acharmos mais adequado a segunda alternativa (2), envergamos por (1), assim, vimos desde logo facilitado o manuseamento de operações condicionais que dariam fruto a várias possibilidades de manusementos na stack prinicapl. Assim, tabalhando na mesma frame, os pops e pushes relativos aos slatos condicionais poderiam ocorrer sem qualquer tipo de esforço adicional.

Para suportar a medida, o grupo passa a carregar o código (assembly) relativo a uma definição *per se* ao invés de realizar uma operação de call à função (bem como a definir esta num qualquer lugar no assembly). Deste modo, o assembly pode vir a ter mais instruções do que aquelas que seriam necessárias via abordagem (2)[1].

A alternativa a esta abordagem seria incorrer via (1) ao uso de uma manipulação heapónica que nos permitisse controlar a reestruturação da stack antes da call à função (sem ou com novos elementos).

[1] Se pelo menos existir uma única que seja, chamada repetida a uma determinada função.

4.1.2 Ciclos

Em relação aos ciclos surgiram algumas dificuldades inerentes ao processo de trabalhar numa única frame. Assim, os mesmos só aninham se explicitamente definidos. O processo de os definir numa função e os chamar faz com que os cálculos relativos ao bloco alocado sejam adulterados (provocando um erro de acesso ao bloco em questão).
Para resolver este problema teríamos de voltar atrás na abordagem da chamada de funções ou investir mais tempo na conceção de um processo/mecanismo que nos permitisse realizar a previsão correta do bloco utilizado no aninhamento de ciclos através da chamada de funções.

4.1.3 Expressões Regulares

Não nas expressões em si, mas devido à ferramenta, à ordem pela qual estas foram definidas.
Na sequência destas restrições resulta na coneção de um mecanismo de segurança que antevê a adulteração do tipo da captura em eminência aquando da captura de uma palavra restrita (IF, ELSE, DO, etc...) dentro de uma captura de um Identificador (p.e.).
No nosso caso utilizámos então um dicionário cujos nomes dos tokens mapeavam o tipo destes, fazendo com que o yacc -a gramática- os encaminha-se às respetivas produções.

4.2 Testes realizados e Resultados

Os testes que o grupo realizou à ferramenta proveem de um ficheiro de instruções Forth.
Posteriormente, em caso de compilação bem sucedida é gerado um código EWVM para um fihceiro. O nome do ficheiro pode ser alterado (por omissão é gerado um "a.vm"). Não obstante, existe a possibilidade do surgimento de warnings no ecrã, nomeadamente no caso de poderem existir operandos em stack ainda no final do programa.
No caso da compilação não ser bem sucedida, existem algumas mensagens de erro que o grupo preparou com o intuito de conduzir o utilizador à recuperação do(s) mesmo(s).

4.2.1 N Push

FORTH

```
: PUSHN DO I LOOP ;

10 0 PUSHN
```

EWVM

```
Start

    pushi 10
    pushi 0
loop1start:
    alloc 3
    pop 1

    pushst 0
```

```
    pushsp
    load -1
    store 0
    pop 1

    pushst 0
    pushsp
    load -1
    store 1
    pop 1

    pushst 0
    pushi 1
    store 2

loop1cond:
    pushst 0
    load 0
    pushst 0
    load 1
    inf
    jz loop1fim

loop1body:
    pushst 0
    load 0

loop1inc:
    pushst 0

    pushst 0
    load 0
    pushst 0
    load 2
    add

    store 0
    jump loop1cond

loop1fim:
frees:
    popst
Stop
```

4.2.2 Tabuada (Erro)

FORTH

```
: TABUADA DO 11 1 DO I J * LOOP LOOP ;

    TABUADA
```

Terminal

```
Erro: 3:4: Falta(m) 2 operando(s)!
   3| TABUADA

STOP: Ficheiro não compilado!
```

4.2.3 Tabuada

FORTH

```
: TABUADA DO 11 1 DO I J * LOOP LOOP ;

3 2 TABUADA
```

EWVM

```
Start

    pushi 3
    pushi 2
loop1start:
    alloc 3
    pop 1

    pushst 0
    pushsp
    load -1
    store 0
    pop 1

    pushst 0
    pushsp
    load -1
    store 1
    pop 1

    pushst 0
    pushi 1
    store 2

loop1cond:
    pushst 0
    load 0
    pushst 0
```

```
    load 1
    inf
    jz loop1fim

loop1body:
    pushi 11
    pushi 1
loop2start:
    alloc 3
    pop 1

    pushst 1
    pushsp
    load -1
    store 0
    pop 1

    pushst 1
    pushsp
    load -1
    store 1
    pop 1

    pushst 1
    pushi 1
    store 2

loop2cond:
    pushst 1
    load 0
    pushst 1
    load 1
    inf
    jz loop2fim

loop2body:
    pushst 1
    load 0
    pushst 0
    load 0
    mul

loop2inc:
    pushst 1

    pushst 1
    load 0
    pushst 1
```

```
    load 2
    add

    store 0
    jump loop2cond

loop2fim:

loop1inc:
    pushst 0

    pushst 0
    load 0
    pushst 0
    load 2
    add

    store 0
    jump loop1cond

loop1fim:
frees:
    popst
    popst
Stop
```

4.2.4 Congruência Módulo 2

FORTH

```
: CONGRUENCIA  (n n --) DO I 2 % ( - - % n )
                            IF I . ." : Não congruente!" CR ( n : String CR )
                            ELSE I . ." : Congruente." CR ( n : String CR )
                            THEN
                        LOOP ;

10 0 CONGRUENCIA
```

EWVM

```
Start

    pushi 10
    pushi 0

loop1start:
    alloc 3
    pop 1
```

```
    pushst 0
    pushsp
    load -1
    store 0
    pop 1

    pushst 0
    pushsp
    load -1
    store 1
    pop 1

    pushst 0
    pushi 1
    store 2

loop1cond:
    pushst 0
    load 0
    pushst 0
    load 1
    inf
    jz loop1fim

loop1body:
    pushst 0
    load 0
    pushi 2
    mod

    jz else1
    pushst 0
    load 0
    writei
    pushi 32
    writechr
    pushs  " : Não congruente!"
    writes
    writeln

    jump then1
else1:
    pushst 0
    load 0
    writei
    pushi 32
    writechr
    pushs  " : Congruente."
```

```
    writes
    writeln

then1:

loop1inc:
    pushst 0

    pushst 0
    load 0
    pushst 0
    load 2
    add

    store 0
    jump loop1cond

loop1fim:
frees:
    popst
Stop
```

4.2.5 Divisão por zero?

FORTH

```
: DIVCHECK   DUP 0= IF  ." invalid " DROP
                    ELSE  /
                    THEN ;

2 2 DIVCHECK
```

EWVM

```
Start

    pushi 2
    pushi 2
    dup 1
    pushi 0
    equal
    jz else1
    pushs  " invalid "
    writes
    pop 1
    jump then1
else1:
    div
    ftoi
```

```
then1:
Stop
```

4.2.6 Divisão por zero? (Pro Forth)

FORTH

```
: DIVCHECK  DUP IF  /  THEN DROP;

2 0 DIVCHECK
```

EWVM

```
Start

    pushi 2
    pushi 0
    dup 1
    jz then1
    div
    ftoi
    jump then1
then1:
    pop 1
Stop
```

4.2.7 Eggsize

FORTH

```
: EGGSIZE
    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 ;

25 EGGSIZE
```

EWVM

```
Start

    pushi 25
    dup 1
    pushi 18
    inf
    jz else1
```

```
    pushs  " reject "
    writes
    jump then5
else1:
    dup 1
    pushi 21
    inf
    jz else2
    pushs  " small "
    writes
    jump then4
else2:
    dup 1
    pushi 24
    inf
    jz else3
    pushs  " medium "
    writes
    jump then3
else3:
    dup 1
    pushi 27
    inf
    jz else4
    pushs  " large "
    writes
    jump then2
else4:
    dup 1
    pushi 30
    inf
    jz else5
    pushs  " extra large "
    writes
    jump then1
else5:
    pushs  " error "
    writes
then1:
then2:
then3:
then4:
then5:
    pop 1
Stop
```

Capítulo 5

Conclusão

O documento investe em tornar sucinto o trabalho desenvolvido pelo grupo, sobretudo as partes fulcrais (tomadas de decisão que levam à aquisição de experiência para trabalhos futuros na área).

Em suma, queremos deixar claro que achamos ter adquirido os princípios inerentes ao trabalho: conhecimento e manipulação da ferramenta PLY, uso de expressões regulares, definição de gramáticas independentes de contexto, bem como linguagens arquiteturais de processamento máquina e os seus desafios.

Saímos mais experientes e maduros deste trabalho, e agradecemos a toda a equipa docente por isso.

How-to

```
$ tree
.
├── lex.py
├── main.py
└── yac.py

0 directories, 3 files
```

```
$ tree
.
├── congruencia.forth
├── lex.py
├── main.py
└── yac.py

0 directories, 4 files
```

```
$ python3 main.py congruencia.forth
$ tree
.
├── a.vm
...
```

```
$ python3 main.py congruencia.forth -o congruencia
$ tree
.
├── congruencia.vm
...
```