Processamento de Linguagens

Equipa 2

6 de maio de 2024

Daniel Pereira	**Francisco Ferreira**	**Rui Lopes**
A100545	A100660	A100643

Introdução

Este relatório tem como objetivo apresentar o trabalho prático desenvolvido na unidade curricular de Processamento de Linguagens. O trabalho consiste no desenvolvimento de um compilador de *Forth*[1] para a EWVM[2], uma máquina virtual capaz de interpretar instruções estilo *assembly*. Deste modo, iremos apresentar a arquitetura do sistema, a sua implementação e as decisões tomadas pelo grupo durante o desenvolvimento do mesmo.

1. Arquitetura

Na implementação do compilador, o grupo optou por uma arquitetura simples, porém altamente eficaz e modular. Nos próximos subcapítulos, iremos detalhar os componentes que constituem este compilador.

1.1. Lexer

O primeiro componente a entrar em ação durante o processo de compilação é, naturalmente, o *lexer*. Para *tokenizar* o código *Forth* submetido ao nosso programa, fazemos uso do módulo `lex` fornecido pelo Ply[3], o que nos permitiu implementar um *lexer* de forma bastante rápida e eficaz.

De uma forma geral, a implementação do *lexer* não envolveu grandes decisões de implementação, dada a simplicidade da linguagem de programação *Forth*. Desta maneira, seguimos a estrutura genérica exemplificada pela documentação do próprio *Ply*, sem qualquer necessidade de recorrer a *lexing states*. Mesmo sendo simples, essa abordagem permitiu-nos desenvolver perfeitamente o processo de *tokenização* deste projeto.

Os *tokens* do nosso *lexer* são: `NUMBER`, `PLUS`, `MINUS`, `TIMES`, `DIVIDE`, `MOD`, `SLASH_MOD`, `EQUALS`, `NOT_EQUALS`, `LESS_THAN`, `LESS_THAN_OR_EQUAL_TO`, `GREATER_THAN`, `GREATER_THAN_OR_EQUAL_TO`, `ZERO_EQUALS`, `ZERO_LESS_THAN`, `ZERO_LESS_THAN_OR_EQUAL_TO`, `ZERO_GREATER_THAN`, `ZERO_GREATER_THAN_OR_EQUAL_TO, COLON`, `SEMI_COLON`, `LITERAL`, `PRINT_STRING`, `CHAR_WORD`, `IF`, `ELSE`, `THEN`, `DO`, `LOOP`, `PLUS_LOOP`, `BEGIN`, `UNTIL`, `AGAIN`, `VARIABLE_DECLARATION`, `CONSTANT_DECLARATION`, `STORE`, `FETCH`.

Vale relembrar que, apesar do *Ply* providenciar formas de definir *literals*, o grupo decidiu optar por não utilizar esses mecanismos. Isso deve-se ao facto de alguns operadores, em *Forth*, serem definidos por mais de um caracter, como é o caso do `/mod`. Assim, por uma questão de uniformização, todos os *tokens* são definidos por regras.

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

1.2. Parser

Durante o processo de implementação do *parser* do nosso projeto, adotamos uma abordagem iterativa. Conforme desenvolvíamos a gramática, simultaneamente trabalhávamos na estrutura do *parser*, dada a forte interligação entre ambos.

Para tal, utilizamos o módulo `yacc`, novamente da biblioteca *Ply*, para construir o *parser*. Desta forma, a interligação entre o *parser* e o *lexer* foi direta. De ressaltar que o tipo de parsing utilizado pelo `yacc` é LALR (Look-Ahead Left-to-Right), o que oferece vantagens significativas em termos de eficiência e desempenho.

1.2.1. Gramática

A gramática desenvolvida apresenta uma estrutura clara e concisa que reflete as principais construções da linguagem *Forth*.

É recomendado a consulta da gramática na Secção 6.1 para um melhor enquadramento durante os próximos capítulos.

1.2.2. Estrutura intermédia (Árvore sintática)

Para simplificar o subsequente processo de tradução do código *Forth* para a EWVM, o grupo optou por adotar a construção de uma *Abstract Syntax Tree* (AST) durante o *parsing*. Nesse sentido, em vez das produções retornarem diretamente o código traduzido para a EWVM, elas retornam classes que representam os nós da nossa AST.

Todas as ASTs produzidas têm como raiz a classe `AbstractSyntaxTree`, que armazena todos os seus nós filhos numa lista. Esta classe é composta por métodos de mostrar a sua representação, de comparação e de tradução. Este último será explorado em maior detalhe posteriormente neste relatório. Todos os nós filhos seguem uma estrutura semelhante, podendo ter mais ou menos atributos conforme a necessidade, e representam uma variedade de expressões que podem ocorrer no código *Forth*.

Tal como se pode ver no excerto de código que se segue, a produção `p_ast()` do nosso *parser*, devolve o nodo `AbstractSyntaxTree` passando como argumento toda a gramática que foi reconhecida.

```
def p_ast(self, p):
  """ast : grammar"""
  p[0] = ast.AbstractSyntaxTree(p[1])
```

Tomando por exemplo o código `5 5 + 0 DO I . LOOP`, a respetiva AST seria:

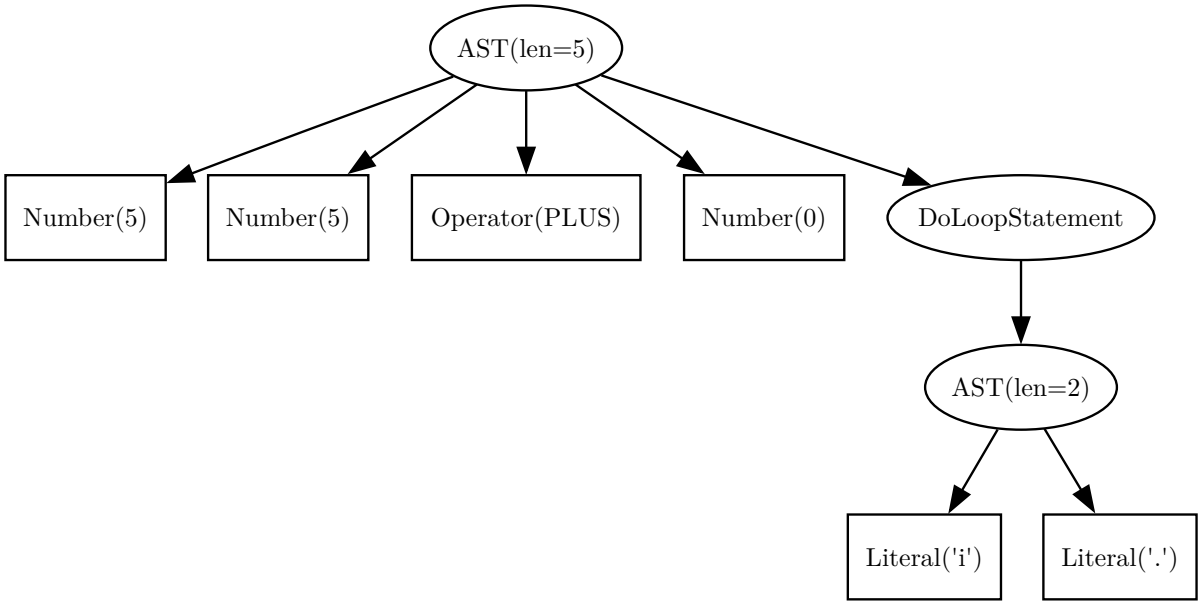

Figura 1: Exemplo de AST de código *Forth*

1.3. Tradutor

Conforme discutido no subcapítulo anterior, a estrutura resultante do *parser* servirá como base neste componente do programa para efetuar as traduções para a EWVM.

1.3.1. *Visitor Pattern*

A simplicidade da implementação do nosso projeto é amplamente atribuída à adoção de um mecanismo de travessia da AST, como o *Visitor Pattern*.

A adoção deste padrão possibilitou-nos separar toda a lógica de tradução num módulo distinto das classes dos nós da AST. Isso resultou numa base de projeto muito mais limpa, organizada e escalável. Além disso, a implementação desse padrão facilita a adição de travessias adicionais no futuro, mantendo o processo simples e direto.

A implementação deste padrão consiste numa classe apenas, denominada de `EWVMTranslator` que herda da classe abstrata `Translator` onde estão definidos todos os métodos visitantes que terão de ser implementados para a AST definida, ou seja, um método visitante por cada tipo de nó da AST.

Desta maneira, tendo o resultado do *parser*, que é do tipo `AbstractSyntaxTree`, e querendo reduzir a AST para código da EWVM bastaria fazer:

```
result: AbstractSyntaxTree = parser.parse(s)
result.evaluate(EWVMTranslator())
```

Onde temos de passar como argumento a implementação do *Visitor* que pretendemos que seja usada na travessia.

2. Implementação do tradutor

Finalizada a análise geral da arquitetura do nosso projeto, este capítulo propõe-se a examinar de que modo as diferentes funcionalidades de *Forth* foram implementadas no nosso compilador. Para tal, foram exploradas diferentes estratégias tanto no código gerado para a EWVM como na tradução da estrutura intermédia explorada anteriormente.

Nesta análise mais detalhada, vamos examinar a classe `EWVMTranslator` mais detalhadamente, que, como mencionado anteriormente, é a implementação do *visitor* de tradução para a EWVM.

2.1. Operações matemáticas

O *Forth* disponibiliza uma série de operações matemáticas básicas que foi imperativo implementar no projeto. Estas operações incluem os operadores aritméticos fundamentais: adição (`+`), subtração (`-`), multiplicação (`*`), divisão (`/`), resto da divisão inteira (`/MOD`), e operadores de comparação: igual (`=`), diferente (`<>`), menor (`<`), maior (`>`), menor ou igual (`<=`), maior ou igual (`>=`), igual a zero (`0=`), menor que zero (`0<`), menor ou igual a zero (`0<=`), maior que zero (`0>`), maior ou igual a zero (`0>=`). Estes operadores são reconhecidos pela nossa gramática como símbolos terminais e são transformados em nodos `Operator` e `ComparisonOperator` pelo nosso parser.

A tradução destes operadores para a EWVM é praticamente direta, visto que muitos deles são também primitivas da EWVM:

Operadores matemáticos

As traduções de quase todos os operadores são feitas recorrendo a uma instrução: `-` → `sub` , `+` → `add` , `*` → `mul` , `/` → `div` , `\` → `mod` .

No entanto, o `/MOD` , que coloca o resto e o quociente no topo da *stack* a partir de dois números já é mais complicada. O que este faz é duplicar os dois primeiros elementos da *stack* (primeiras quatro instruções), fazer a divisão, guardar o seu resultado na *struct heap*, fazer o resto da divisão e carregar a divisão guardada na *struct heap*.

```
pushsp
load -1
pushsp
load -1
div
alloc 1
swap
store 0
mod
pushst 0
load 0
popst
```

Código 1: Produção para `/MOD`

Operadores lógicos

De forma semelhante, instruções para operadores lógicos do *Forth* já estavam implementados na EWVM nativamente: `=` → `equal` , `<>` → `equal not` , `<` → `inf` , `>` → `sup` , `<=` → `infeq` , `>=` → `supeq` , `0=` → `not` , `0<=` → `pushi 0` e `infeq` , `0<` → `pushi 0` e `inf` , `0>=` → `pushi 0` e `supeq` , `0>` → `pushi 0` e `sup` .

2.2. Palavras

Uma das principais funcionalidades do *Forth* é a capacidade de mapear pedaços de código executável a palavras, tornando os programas bastante mais modulares e simpáticos de ler. Para a implementação desta funcionalidade, o grupo decidiu seguir uma estratégia bastante simplista.

2.2.1. Palavras definidas pelo utilizador

Para registar as palavras definidas pelo utilizador, o `EWVMTranslator` emprega um dicionário interno. Nele, o nome da palavra é utilizado como chave, enquanto o código traduzido correspondente é atribuído como valor.

Quando uma nova palavra é definida, o método `visit_word()` primeiro verifica se a palavra já está definida no dicionário. Se estiver, ele lança uma exceção `TranslationError`. Caso contrário, ele avalia a AST definida dentro da palavra e armazena-la no dicionário interno.

Posteriormente, quando uma palavra é chamada, o compilador verifica novamente se esta está definida. Se estiver, traduz a palavra para o código correspondente da EWVM. Caso contrário, uma exceção `TranslationError` é lançada.

Uma característica notável do *Forth* é que as suas funções não recebem argumentos, utilizando apenas os valores presentes na pilha de dados. Como resultado, não foi necessário realizar chamadas explícitas na EWVM ou armazenar valores em memória auxiliar. A substituição direta da palavra pelo código correspondente da EWVM foi suficiente para manter a compatibilidade e eficiência do compilador.

Tomando como exemplo a seguinte definição:

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

Este código irá produzir a seguinte AST:

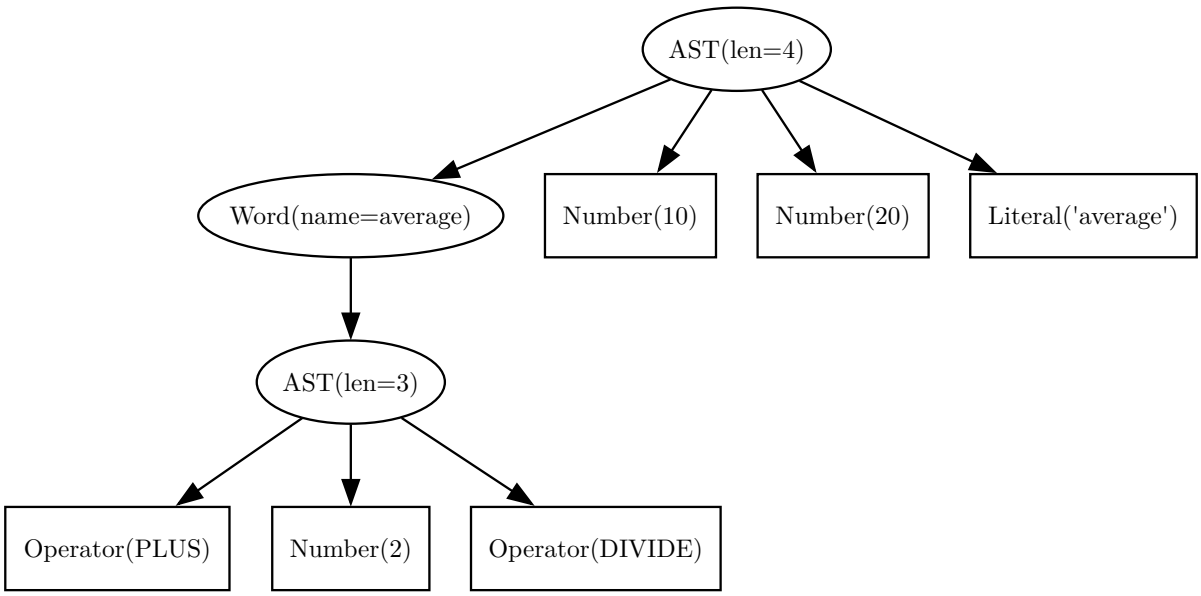

Figura 2: Exemplo de AST da definição de uma palavra

Que por sua vez, produz o seguinte código da EWVM:

```
start
pushi 10
pushi 20
add
pushi 2
div
stop
```

Código 2: Produção para uma palavra

2.2.2. Palavras pré-definidas

Como é comum em linguagens de programação, *Forth* já possui algumas palavras pré-definidas que fornecem funcionalidades básicas da linguagem. Para dar suporte a essas palavras, o nosso compilador mantém um dicionário interno de palavras pré-definidas, semelhante ao dicionário de palavras definidas pelo utilizador. A diferença fundamental é que essas palavras não são adicionadas dinamicamente durante a compilação, mas são programadas diretamente no compilador.

Naturalmente, existem outras estratégias para alcançar esse objetivo, como definir as palavras pré-definidas diretamente na gramática. No entanto, essa abordagem resultaria em mais sobrecarga ao adicionar novas palavras, já que seria necessário modificar tanto o *lexer* quanto o *parser*, além do tradutor, cada vez que uma nova palavra fosse necessária. Diante dessas considerações, o grupo concluiu que a abordagem adotada, com o uso de um dicionário interno, era a mais eficiente e minimalista para gerir as palavras pré-definidas.

Alguns exemplos destas palavras pré-definidas são:

`emit` → `writechr`
`.` → `writei`
`cr` → `pushi 10 writechr`

No entanto, também adicionamos uma biblioteca padrão ao projeto, composta por palavras pré-definidas pelo *Forth*. Essas palavras são implementadas em código *Forth* e disponibilizadas para uso. Esta biblioteca padrão é passada como argumento na criação do objeto `EWVMTranslator` que por sua vez avalia estas palavras em código da EWVM e guarda-as no dicionário `predefined_words`.

Um exemplo disto é a `SPACES` que foi implementada da seguinte forma:

```
: SPACES 0 DO SPACE LOOP ;
```

Quando a tradução da função `SPACES` é chamada, a função acima é avaliada como se fosse uma função definida pelo utilizador.

Desta forma, será possível, futuramente, incluir toda a *standard lib* do *Forth* no nosso transpilador facilmente.

2.3. Condicionais

Vamos agora discutir um bocado sobre a implementação das condicionais.

Em *Forth*, as condicionais são bastante simples, apresentando apenas as seguintes variantes:

```
1 2 = IF ." IGUAIS" THEN
```

```
1 2 = IF ." IGUAIS" ELSE ." DIFERENTES" THEN
```

A AST produzida para este último caso seria:

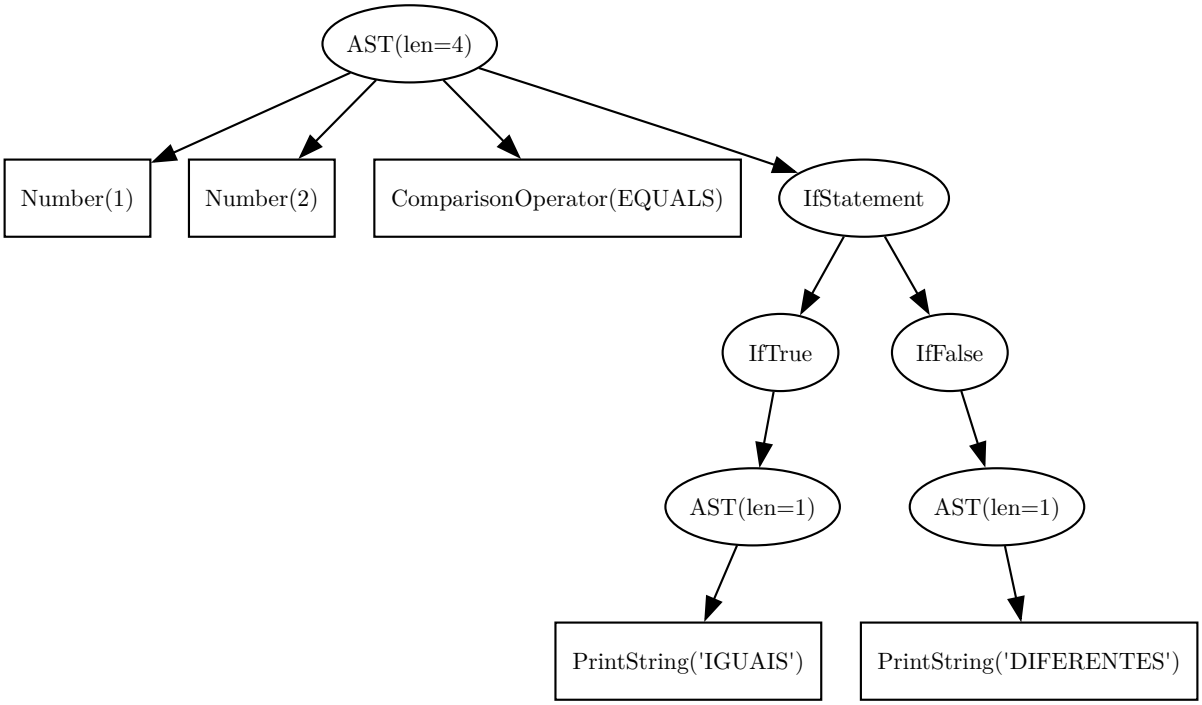

Figura 3: Exemplo de AST de uma condicional

Como podemos observar, o nodo `IfStatement` produz duas ramificações, uma para caso de verdade e outra para caso de falsidade. O resultado de tradução desta AST seria:

```
start
pushi 1
pushi 2
equal
jz else0
pushs "IGUAIS"
writes
jump endif0
else0:
  pushs "DIFERENTES"
  writes
endif0:
stop
```

Código 3: Produção para um condicional

Para traduzir as estruturas condicionais para a EWVM, o grupo utilizou *labels* em conjunto com *jumps* para realizar saltos condicionais no código gerado. Uma consideração importante sobre esta estratégia é que as *labels* produzidas precisam ser numeradas (por exemplo, `else0`) para evitar conflitos entre diferentes estruturas condicionais. Portanto, foi necessário manter um contador de condicionais no estado do nosso tradutor, de modo que sempre que um nodo

condicional fosse visitado utilizando o método `visit_if_statement()`, esse contador fosse incrementado.

2.4. Ciclos

Um dos maiores desafios na tradução de código *Forth* para a EWVM foi na implementação de ciclos devido à variedade de ciclos existentes em *Forth* e ao modo de operação da EWVM.

Durante a implementação, uma das principais dificuldades enfrentadas foi a necessidade de criar ciclos que pudessem aceder tanto à pilha global quanto ao contador do ciclo, além de suportar ciclos aninhados. No entanto, o grupo conseguiu superar esses desafios e garantir que os ciclos funcionassem corretamente para todos os cenários previstos pelo *Forth*. Vamos explorar em mais detalhes cada um dos tipos de ciclos implementados.

2.4.1. `DO ... LOOP`

O formato mais comum de ciclos em *Forth* é o seguinte:

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

Neste exemplo em específico estamos a fazer *print* do valor do iterador atual a cada iteração. Para este caso, a AST produzida é a seguinte:

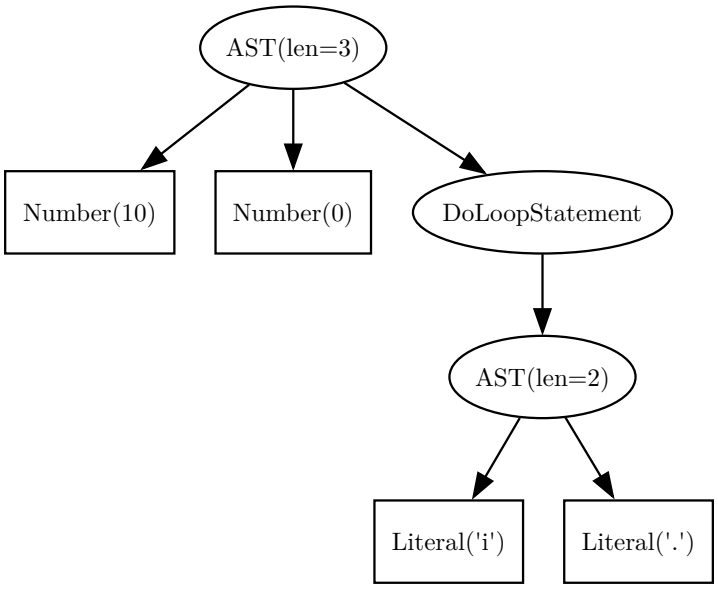

Figura 4: Exemplo de AST de um loop

E o código da EWVM produzido:

```
start
pushi 10
pushi 0

// início do ciclo
alloc 2
swap
store 1
pushst 0
swap
store 0
startloop1:
  pushst 0
  load 0
  pushst 0
  load 1
  sup
  jz endloop1

  // início do corpo do ciclo
  pushst 0
  load 1
  writei
  //fim do corpo do ciclo

  pushst 0
  load 1
  pushi 1
  add
  pushst 0
  swap
  store 1
  jump startloop1
endloop1:
  popst
stop
```

Código 4: Produção para um loop simples

A estratégia geral adotada para este tipo de ciclos envolve a utilização da *struct heap* da EWVM como pilha de memória auxiliar. Como podemos observar, é feita uma alocação (`alloc 2`) logo no início do ciclo. Nesta memória auxiliar, são armazenados os dois valores no topo da pilha global, que representam o valor inicial e final do ciclo. Essa abordagem é necessária devido à característica do *Forth* que permite alterações na pilha global durante a execução do corpo do ciclo, o que nos impede de guardar esses valores diretamente na pilha global.

Para além disso, assim como nas condicionais discutidas anteriormente, o grupo utilizou novamente *labels* e *jumps*, o que nos levou a manter um contador de ciclos no estado do nosso tradutor para todos os ciclos implementados, pelos mesmos motivos explicados anteriormente. Ademais, foi necessário monitorizar o número de elementos na *struct heap* para garantir a correta manipulação da memória auxiliar durante a execução dos ciclos.

2.4.2. `DO ... +LOOP`

Este tipo de ciclo é bastante similar ao anteriormente descrito, quer nas AST's produzidas como no código da EWVM a que dá resultado. A única diferença está no valor de incrementação a

cada iteração que ao invés de ser 1, passa a ser o valor presente no topo da pilha de dados. De resto, a solução é idêntica à do `DO ... LOOP`.

2.4.3. `BEGIN ... UNTIL`

Em contraste com os dois ciclos mencionados anteriormente, em termos de código EWVM gerado, este é consideravelmente mais simples, pois não requer o armazenamento de valores em memória auxiliar. Uma vez que, ao final do corpo de cada ciclo, é realizada uma verificação no topo da pilha de dados para determinar se há um valor 0 ou 1, apenas um *jump* condicional é necessário. Tomemos como exemplo o seguinte código:

```
0 BEGIN 1 + DUP . DUP 10 >= UNTIL
```

Este código Forth imprime o valor no topo da pilha de dados até que seja igual a 10, aumentando em 1 a cada iteração. A AST produzida por este pedaço de código *Forth* é muito similar à estrutura do primeiro exemplo, substituindo apenas o nodo `DoLoopStatement` por `BeginUntilStatement` e o código da EWVM produzido a partir desta expressão é simplesmente:

```
start
pushi 0
startloop1:
  pushi 1
  add
  dup 1
  writei
  dup 1
  pushi 10
  supeq
  jz startloop1
stop
```

Código 5: Produção para um loop *begin until*

Dentro da *label* startloop1, encontra-se apenas o código contido no corpo do ciclo, seguido por um *jump* condicional que retorna ao início desse mesmo ciclo caso no topo da pilha de dados esteja um valor diferente de 0.

2.4.4. `BEGIN ... AGAIN`

Para terminar, o último tipo de ciclo implementado pelo grupo foram os ciclos infinitos. Neste tipo de ciclos, apesar de muito similares aos ciclos `BEGIN ... UNTIL`, não há qualquer tipo de verificação lógica no fim de cada iteração havendo simplesmente um *jump* para a *label* inicial do ciclo.

2.5. Variáveis e Constantes

Para concluir, a última funcionalidade de *Forth* adicionada ao nosso projeto foi a capacidade de definir variáveis e constantes.

Para implementar essas funcionalidades, optamos por uma abordagem um pouco diferente das anteriores. Tanto as variáveis quanto as constantes permitem armazenar valores fora da pilha de dados principal, além de possibilitar a inserção e, no caso das variáveis, a alteração do valor armazenado. Portanto, foi crucial encontrar uma maneira eficiente de aceder a essas variáveis durante a execução do programa.

Durante a travessia da nossa AST, mantemos um contador de variáveis como estado. Cada vez que o *visitor* responsável pela declaração de variáveis ou constantes é chamado, esse contador é incrementado em 1. Além disso, os nomes das variáveis são armazenados como chaves em dicionários internos (`user_declared_variables` e `user_declared_constants`), enquanto os valores correspondentes são os números do contador de variáveis quando cada variável é declarada. Ao final da travessia da AST, são inseridos na pilha global tantos `pushi 0` quantas variáveis e constantes foram declaradas, reservando espaço necessário antes do início do programa.

2.5.1. Variáveis

Vamos analisar mais a fundo o exemplo da declaração de uma variável. Vejamos o seguinte exemplo:

```
VARIABLE FOO 10 FOO ! FOO @ .
```

Neste exemplo, estamos a declarar a variável `FOO`, a armazenar o valor 10 dentro da variável e, de seguida, a dar *push* do valor armazenado em `FOO` para a pilha de dados e imprimir o valor do topo da pilha.

A AST produzida por este programa é:

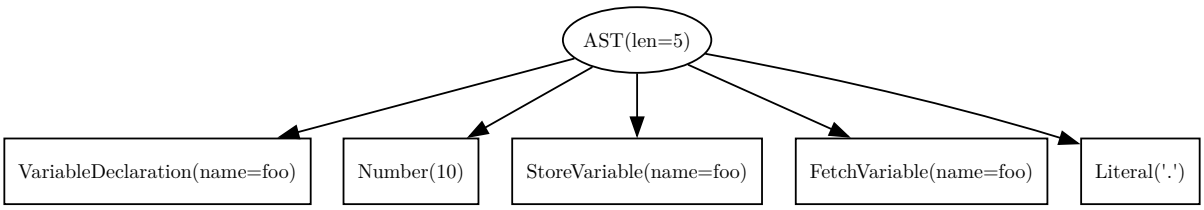

Figura 5: Exemplo de AST de declaração de uma variável

E o código da EWVM produzido pela tradução desta AST é:

```
pushi 0
start
pushi 10
storeg 0
pushg 0
writei
stop
```

Código 6: Produção para variáveis

Como podemos observar, dado que temos apenas uma variável, há apenas um comando `pushi 0` inserido antes do início do programa (start). Quando desejamos armazenar um valor em `FOO`, utilizamos o comando `storeg 0`, onde 0 representa o índice da posição da variável em relação ao apontador geral da pilha de dados e quando pretendemos aceder ao valor de uma variável simplesmente fazemos `pushg 0`, onde 0 é novamente o índice da variável que pretendemos aceder.

2.5.2. Constantes

A implementação das constantes é extremamente semelhante à das variáveis, diferenciando-se apenas no facto de que não podemos alterar o valor de uma constante após a sua declaração. Esta restrição é feita na tradução do nosso programa. No entanto, a maneira como armazenamos valores nas constantes e é feito o acesso a eles permanece exatamente a mesma.

3. Interpretador

No esforço de aprimorar a experiência do utilizador com o nosso compilador *Forth*, desenvolvemos um interpretador com uma gama diversificada de modos de operação. Esses modos proporcionam a capacidade de compilar e analisar o quer o código *Forth* traduzido quer as estruturas intermédias do nosso compilador de maneira eficiente, oferecendo uma experiência de programação mais flexível e interativa. Neste capítulo, vamos explorar em detalhes os diferentes modos de operação disponíveis, destacando as suas características.

Para iniciar a nossa interface de utilização via terminal, basta correr `bin/forthpiler` dentro do nosso projeto e nosso programa inicia a seguinte interface:

```
Starting in TRANSLATE.
Change to other interpreter modes with /parse, /run, /translate, /visualize
translate >>
```

Para mudar de modo de interpretação basta inserir o nome do modo que pretendemos seguido de um `/`. Vamos agora explorar cada um dos modos de interpretação desenvolvidos.

3.1. *Translate*

Este é o principal modo de operação onde dado um pedaço de código *Forth* retorna como resultado o código da EWVM gerado:

```
Starting in TRANSLATE.
Change to other interpreter modes with /parse, /run, /translate, /visualize
translate >> 1 2 + 3 *
start
pushi 1
pushi 2
add
pushi 3
mul
stop
```

3.2. *Parse*

Neste modo de operação, o interpretador não traduz a AST para o código da EWVM, em vez disso, retorna a AST gerada a partir do código *Forth* fornecido. Esta funcionalidade mostrou-se extremamente útil para o grupo, pois permitiu depurar o código desenvolvido ao longo do projeto de forma eficaz. Ao fornecer a AST como saída, os membros do grupo puderam examinar a estrutura interna do código, identificar possíveis erros e realizar ajustes conforme necessário, agilizando assim o processo de desenvolvimento.

A implementação desta funcionalidade cabe à chamada do método de mostrar representação da árvore principal. Esta, irá chamar, recursivamente, os métodos de representação dos seus filhos até chegar aos nodos terminais, formando a *string* completa.

```
Starting in TRANSLATE.
Change to other interpreter modes with /parse, /run, /translate, /visualize
translate >> /parse
Mode changed to PARSE
parse >> 1 2 + 3 *
AST(expressions=[Number(1), Number(2), Operator(PLUS), Number(3), Operator(TIMES)])
```

3.3. *Run*

Outro modo de operação que se revelou essencial durante o desenvolvimento do projeto foi o modo de execução. Neste modo, o interpretador retorna o resultado do código da EWVM gerado pelo programa *Forth* inserido no interpretador. Este modo de operação foi fundamental para facilitar o processo de testagem do código da EWVM produzido pelo compilador. Ao fornecer o resultado da execução do código EWVM, o grupo pôde verificar se o compilador estava a gerar o código desejado e se este estava a funcionar conforme o esperado, permitindo assim a deteção e correção eficiente de possíveis erros ou problemas de implementação, sem ter o trabalho de abrir o site da EWVM.

A implementação desta funcionalidade recorre ao endpoint https://ewvm.epl.di.uminho.pt/run enviando um JSON com contéudo `{"code": <código>}`. A resposta, apesar de ser um HTML pesado e verboso, é interpretada (recorrendo ao pacote `beautifulsoup4` [4]) para retirar apenas o resultado da execução do código e assim o mostrar no terminal.

```
Starting in TRANSLATE.
Change to other interpreter modes with /parse, /run, /translate, /visualize
translate >> /run
Mode changed to RUN
run >> 1 2 + 3 *
'9'
```

3.4. *Visualizer*

Por último, temos o modo de visualização, que, embora semelhante ao modo *parse*, retorna a Árvore Sintática Abstrata (AST) produzida no formato de grafo numa imagem. Todas as imagens das ASTs neste relatório foram geradas utilizando este modo de operação. Este modo permite uma representação visual clara da estrutura do código *Forth* analisado, tornando mais acessível a compreensão da organização e dos relacionamentos entre os elementos da AST.

Esta visualização é criada usando o pacote *graphviz*[5], que disponibiliza métodos desde a definição de grafos, até à sua criação e abertura automática do `.svg` gerado.

A sua implementação tirou proveito da forte modularidade do nosso projeto, ao recorrermos ao *visitor pattern*. Desta forma, foi implementado outra classe que estende o `Translator`, apelidado de `GraphvizTranslator`, que define que nodos criar no grafo conforme o tipo de nó que é visitado da AST.

A tradução, de modo geral, é feita semelhantemente para todos os nodos. Cria-se o nodo do nó da AST e caso seja um nó com filhos, cria os filhos recursivamente com uma ligação até eles.

[4] https://pypi.org/project/beautifulsoup4/
[5] https://pypi.org/project/graphviz/

Se o nó for terminal (por exemplo, um número) a representação dele é feita numa caixa em vez de uma figura circular.

De forma semelhante de como a tradução para a EWVM é chamada para a AST, a chamada para tradução para nodos do *graphviz* é semelhante:

```
def visualize(ast: syntax.AbstractSyntaxTree):
    translator = GraphvizTranslator()
    result.evaluate(translator)

    translator.graph.render("visualize/result", view=True)
```

4. Considerações adicionais

4.1. Contribuições para EWVM

Dada a complexidade de algumas funcionalidades da linguagem *Forth*, à medida que o projeto ia sendo desenvolvido, o grupo sentiu que o conjunto de instruções originalmente fornecido pela EWVM não era necessário para a implementação deste projeto. Desta forma, o grupo tomou a iniciativa de fazer algumas contribuições para o repositório da EWVM[6] de forma a possibilitar um maior conjunto de funcionalidades.

Algumas dessas contribuições foram:

- Conserto da divisão por 0[7]
- Adição de comandos de manipulação com código ASCII `chrcode` e `writechr`[8]
- Adição de comandos de manipulação na *struct heap* `pushst` e `popst`[9]

Para além destas contribuições nas funcionalidades da EWVM, o grupo também submeteu algumas contribuições para o bom funcionamento da plataforma web.

4.2. Definição de condicionais e ciclos fora de palavras

De acordo com a documentação oficial de *Forth*, a definição de condicionais e ciclos é tradicionalmente permitida apenas dentro da definição de novas palavras. No entanto, o grupo optou por uma abordagem diferente, permitindo a definição dessas primitivas fora da definição de palavras. Para garantir a conformidade com a prática convencional, uma versão alternativa do projeto foi mantida numa *branch* separada, na qual tal permissão não é concedida, seguindo estritamente as diretrizes da documentação oficial. Esta decisão foi tomada para explorar a flexibilidade e as possíveis vantagens de permitir a definição de condicionais e ciclos em diferentes contextos, além de facilitar a comparação entre as duas abordagens.

A única distinção entre ambas as *branches* reside exclusivamente na gramática, que é configurada para reconhecer a definição de condicionais e ciclos apenas dentro da definição de palavras.

[6] https://github.com/jcramalho/ewvm
[7] https://github.com/jcramalho/EWVM/pull/1
[8] https://github.com/jcramalho/EWVM/pull/2
[9] https://github.com/jcramalho/EWVM/pull/9

5. Trabalho Futuro

Após a conclusão deste projeto, ficou claro que há áreas significativas para melhorar o compilador de *Forth* desenvolvido. Uma lacuna notável é a falta de uma análise semântica abrangente. Embora o compilador possa identificar erros de compilação, não realiza análises semânticas mais complexas, como verificação de tipos ou deteção de erros de lógica. A implementação de uma análise semântica completa foi considerada, mas devido à complexidade e tempo limitado, não foi possível incluí-la. Uma das principais dificuldades é simular a pilha de execução do *Forth*, o que adicionaria complexidade ao compilador. No entanto, com mais tempo e recursos, explorar e desenvolver essa funcionalidade seria benéfico para melhorar a robustez do compilador.

Outra área de melhoria identificada está relacionada à implementação de chamadas recursivas. Devido à forma como arquitetamos os resultados produzidos para a EWVM, atualmente não é possível implementar chamadas recursivas no compilador. No entanto, com ajustes na estrutura e na organização do projeto, seria viável explorar a possibilidade de adicionar suporte a chamadas recursivas. Isso abriria novas possibilidades e funcionalidades para o compilador, permitindo a implementação de algoritmos e estruturas de dados mais complexos em *Forth*.

6. Conclusão

Concluindo este projeto com sucesso, o grupo sentiu que o desenvolvimento do mesmo serviu de uma excelente oportunidade para explorar os processos envolvidos na construção de um simples compilador aplicando conceitos aprendidos dentro e fora das aulas.

Anexos

6.1. Gramática

```
grammar : expression grammar
        | ε

expression : NUMBER
           | operator
           | comparison_operator
           | word
           | if_statement
           | loop_statement
           | LITERAL
           | PRINT_STRING
           | CHAR_FUNC
           | VARIABLE_DECLARATION LITERAL
           | CONSTANT_DECLARATION LITERAL
           | LITERAL STORE
           | LITERAL FETCH

operator : PLUS
         | MINUS
         | TIMES
         | DIVIDE
         | EXP
         | MOD
         | SLASH_MOD

comparison_operator : EQUALS
                    | NOT_EQUALS
                    | LESS_THAN
                    | LESS_THAN_OR_EQUAL_TO
                    | GREATER_THAN
                    | GREATER_THAN_OR_EQUAL_TO
                    | ZERO_EQUALS
                    | ZERO_LESS_THAN
                    | ZERO_LESS_THAN_OR_EQUAL_TO
                    | ZERO_GREATER_THAN
                    | ZERO_GREATER_THAN_OR_EQUAL_TO

word : COLON LITERAL grammar SEMI_COLON

if_statement : IF grammar THEN
             | IF grammar ELSE grammar THEN

loop_statement : DO grammar LOOP
               | DO grammar PLUS_LOOP
```