Processamento de Linguagens e Compiladores

LCC – 3º Ano – 1º Semestre
Universidade do Minho

Construção de um Compilador para Pascal Standard

Grupo 03

Trabalho realizado por:	*Número:*
Catarina Machado Barbosa	A108558
Idy Saquina Carlos	A105237
José Afonso da Silva Miranda	A102933

Conteúdo

Introdução

O presente relatório descreve o desenvolvimento de um compilador para a linguagem *Pascal Standard*, realizado no âmbito da unidade curricular *Processamento de Linguagens e Compiladores* do 3.º ano da Licenciatura em Ciências da Computação da Universidade do Minho. Este trabalho tem como principal objetivo a aplicação prática dos conceitos teóricos abordados ao longo da unidade curricular, nomeadamente as diferentes fases do processo de compilação.

O compilador desenvolvido suporta um subconjunto representativo da linguagem Pascal, incluindo declarações de variáveis simples e de arrays, expressões aritméticas, relacionais e lógicas, estruturas de controlo condicionais e iterativas (*if*, *while* e *for*), bem como operações básicas de entrada e saída através das instruções *readln* e *writeln*. O processo de compilação termina na geração de código para uma máquina virtual, permitindo a execução dos programas fonte após uma compilação bem-sucedida.

A implementação do compilador foi realizada em Python, recorrendo à biblioteca *PLY (Python Lex-Yacc)* para a construção do analisador léxico e do analisador sintático. A fase de análise semântica foi desenvolvida com base numa tabela de símbolos, permitindo detetar erros como a utilização de variáveis não declaradas, declarações duplicadas, incompatibilidades de tipos e acessos inválidos a arrays. Posteriormente, foi implementado um gerador de código responsável por traduzir a árvore sintática anotada em instruções para a máquina virtual, respeitando a semântica da linguagem.

Este relatório encontra-se organizado por capítulos: no Capítulo 2 é apresentada uma visão geral do compilador e das suas funcionalidades; no Capítulo 3 é descrita a arquitetura do projeto, detalhando as fases de análise léxica, sintática, semântica e geração de código; no Capítulo 4 é realizada uma avaliação experimental do sistema, através de vários exemplos de teste; por fim, no Capítulo 5 são apresentadas as conclusões e possíveis melhorias futuras.

Visão Geral do Compilador

O compilador desenvolvido no âmbito deste trabalho tem como objetivo traduzir programas escritos em *Pascal Standard* para código executável numa máquina virtual, seguindo a arquitetura clássica de um compilador. O processo de compilação encontra-se dividido em várias fases bem definidas, permitindo uma separação clara de responsabilidades e facilitando a deteção e correção de erros.

A primeira fase corresponde à **análise léxica**, na qual o código fonte é processado caractere a caractere, sendo convertido numa sequência de tokens. Estes tokens representam as unidades básicas da linguagem, como identificadores, palavras reservadas, operadores, constantes numéricas e símbolos de pontuação. Esta fase é responsável por eliminar comentários e espaços em branco irrelevantes, bem como por identificar erros lexicais.

De seguida, ocorre a **análise sintática**, cujo objetivo é verificar se a sequência de tokens obtida respeita a gramática da linguagem Pascal suportada pelo compilador. Nesta fase é construída uma *Árvore Sintática Abstrata* (AST), que representa de forma estruturada o programa fonte, preservando apenas a informação relevante para as fases seguintes do processo de compilação.

Após a construção da AST, o compilador realiza a **análise semântica**. Esta fase tem como principal finalidade garantir que o programa, para além de sintaticamente correto, é semanticamente válido. Para tal, é utilizada uma tabela de símbolos onde são registadas informações relativas às variáveis, arrays e respetivos tipos. São detetados erros como variáveis não declaradas, declarações duplicadas, incompatibilidades de tipos, acessos inválidos a arrays e utilização incorreta de estruturas de controlo.

Por fim, na fase de **geração de código**, a AST anotada é percorrida com o objetivo de produzir código para uma máquina virtual. Este código intermédio utiliza uma linguagem de baixo nível baseada em instruções de pilha, permitindo a execução dos programas compilados. O gerador de código respeita a semântica

da linguagem, tratando corretamente expressões, ciclos, instruções condicionais e operações de entrada e saída.

O compilador foi implementado em Python, recorrendo à biblioteca *PLY (Python Lex-Yacc)* para as fases de análise léxica e sintática. A modularização do projeto permite que cada fase seja testada de forma independente, contribuindo para uma maior robustez e facilidade de manutenção do sistema. Na secção seguinte é apresentada em detalhe a arquitetura do projeto, descrevendo o funcionamento de cada um dos seus componentes.

Arquitetura do Projeto

3.1 Lexer

A análise léxica constitui a primeira fase do processo de compilação e tem como principal objetivo converter o código fonte escrito em *Pascal Standard* numa sequência de tokens. Cada token representa uma unidade léxica da linguagem, como palavras reservadas, identificadores, constantes, operadores e símbolos de pontuação, que serão posteriormente utilizados pelo analisador sintático.

O lexer foi implementado em Python utilizando a biblioteca *PLY (Python Lex-Yacc)*, que fornece mecanismos para a definição de tokens através de expressões regulares. Para cada tipo de token suportado pela linguagem, foi definida uma regra léxica correspondente, permitindo identificar corretamente os diferentes elementos do programa fonte.

Entre os tokens reconhecidos encontram-se identificadores, números inteiros, operadores aritméticos e relacionais, símbolos de atribuição, delimitadores e palavras reservadas da linguagem Pascal, como *program*, *var*, *begin*, *end*, *if*, *then*, *else*, *while*, *for*, *to*, *readln* e *writeln*. O lexer distingue corretamente palavras reservadas de identificadores, garantindo que estas não possam ser utilizadas como nomes de variáveis.

Para além da identificação dos tokens, o analisador léxico é responsável por ignorar caracteres irrelevantes, como espaços em branco, tabulações e quebras de linha, bem como comentários presentes no código fonte. Adicionalmente, são detetados e assinalados erros lexicais sempre que surgem caracteres inválidos ou não reconhecidos pela linguagem, contribuindo para uma deteção precoce de erros no processo de compilação.

A utilização do *PLY* permite uma integração direta entre o lexer e o parser, facilitando a passagem da sequência de tokens para a fase de análise sintática. Esta abordagem modular contribui para a clareza do código e para a manutenção do compilador, uma vez que alterações ao léxico da linguagem podem ser efetuadas de

forma isolada, sem impacto direto nas restantes fases do sistema.

3.1.1 Implementação

Inicialmente, começamos por definir um dicionário `reserved` que é responsável por mapear as palavras-chave da linguagem Pascal Standard, atribuindo a cada uma delas o seu token correspondente.

Este acaba por ser importante, uma vez que permite ao compilador distinguir identificadores genéricos de palavras que são reservadas.

Assim, para cada palavra chave, que é apresentada em mínusculas (uma vez que aparece assim no código fonte de Pascal), atribui-se o respetivo identificador, em maiúsculas, sendo este posteriormente usado pelo parser.

As palavras reservadas foram:

1. **Estrutura do Programa:** `program, begin, end;`
2. **Declaração de Variáveis e Tipos:** `var, integer, real, boolean, string;`
3. **Controlo de Fluxo:** `if, then, else, while, do, for, to, downto;`
4. **Operações Built-in:** `readln, writeln, mod, div;`
5. **Valores Booleanos:** `true, false;`
6. **Operadores Lógicos:** `and, or, not;`
7. **Estruturas de Dados:** `array, of.`

De seguida, definimos um tuplo, `tokens`, que representa todos os tipos de tokens que o lexer será capaz de reconhecer. Entre esses, temos:

1. **Tipos de dados e literais:** `INTEGER, REAL,BOOLEAN, STRINGTYPE, NUM (numeros), ID (identificadores), STRING (strings literais);`
2. **Valores booleanos e operadores lógicos:** `TRUE, FALSE, AND, OR, NOT;`
3. **Palavras-chave da linguagem:** `todas as restantes definidas no reserved`
4. **Operadores aritméticos:** `SOMA (+), SUBTR (-), MULT (*), DIVIDE (/), DIV, MOD;`
5. **Operadores relacionais:** `IGUAL (=), DIFF (<>), MAIOR (>), MENOR (<), MAIORIGUAL (>=), MENORIGUAL(<=);`

6. **Símbolos especiais:** `ATRIBUICAO (:=), PONTOEVIRG (;), DOISPONTOS (:), VIRG (,), PONTO (.), LPAREN(), RPAREN ()), LBRACKET ([), RBRACKET (]);`

Posteriormente, cada token simples (soma, subtração, multiplicação, divisão, maior, menor, maior ou igual, menor ou igual, igual, atribuição, intervalo, ponto e vírgula, dois pontos, vírgula, ponto, parênteses curvos e retos abertos e parênteses curvos e retos fechados) é associado a uma expressão regular, responsável por definir a sequência de caracteres que o lexer deve reconhecer.

De seguida, definimos funções responsáveis por interpretar os tokens complexos, cuja estrutura não é representada através de uma expressão simples.

A função `t_STRING` utiliza uma expressão, `\'([^\\']` .)*⌋, para reconhecer sequências de caracteres, sendo essas delimitadas por um ‘ . Ainda assim, essa função remove essas aspas que se encontram nos extremos, o que preserva apenas o conteúdo da string.

A função `t_NUM` é responsável por identificar os números, inteiros ou reais, através da expressão definida: `\d+(\.\d+)?`. É ainda responsável por converter esse valor para o tipo apropriado: caso contenha um “.”, converte-o para um float e, caso não tenha, para um int.

A função `t_ID` reconhece identificadores, que, por exemplo, podem ser nomes de variáveis, de funções, etc. Para isso, utiliza uma expressão regular, `[A-Za-z_][A-Za-z0-9_]*`, e, para além disso, consulta o dicionário das palavras reservadas, de modo a distinguir id’s de palavras chaves.

Decidimos ainda tratar os três tipos de comentários que são admitidos pelo Pascal Standard através de três diferentes funções.

A função `t_COMMENT1` reconhece comentários do tipo “ ... ”, com recurso à expressão regular `\{[\s\S]*?\}` , descartando tudo o que se encontra dentro desse bloco, e, obviamente, atualizando a contagem correta das linhas

A função `t_COMMENT2` reconhece comentários do tipo “(* ... *)”, com recurso à expressão regular `\(\*[\s\S]*?\*\)` , descartando tudo o que se encontra dentro desse bloco, e, obviamente, atualizando a contagem correta das linhas

A função `t_COMMENT3` reconhece comentários do tipo “//”, com recurso à expres-

são regular `//.*` , descartando tudo o que se encontra nessa linha.

A contagem do número de linhas é feita com recurso à função `t_newline`, que incrementa o contador para cada quebra de linha, `\n`, que encontra no código fonte.

Ignoramos ainda os espaços em branco, definidos na componente `t_ignore`.

Quanto aos erros léxicos, estes são tratados pela função `t_error` que emite uma mensagem de erro, com o número da linha onde esse ocorreu e o caracter que levou a tal erro, quando o lexer intercepta um caracter que não reconhece.

Por fim, criamos uma instância do lexer, que vai processar todas as definições anteriores e gerar o analisador léxico.

3.2 Parser

A análise sintática tem como objetivo verificar se a sequência de tokens produzida pelo analisador léxico respeita a gramática da linguagem Pascal suportada pelo compilador. Esta fase garante que a estrutura do programa fonte está correta do ponto de vista sintático, permitindo a identificação de erros como instruções mal formadas ou utilização incorreta de estruturas de controlo.

O parser foi implementado em Python utilizando a biblioteca *PLY (Python Lex-Yacc)*, através da definição de uma gramática livre de contexto. As regras gramaticais descrevem a estrutura dos programas Pascal, incluindo declarações, blocos de instruções, expressões aritméticas e lógicas, bem como as diferentes estruturas de controlo suportadas, tais como instruções condicionais e ciclos.

Durante a análise sintática é construída uma Árvore Sintática Abstrata (AST), que representa a estrutura do programa de forma hierárquica. A AST elimina detalhes irrelevantes da sintaxe concreta, preservando apenas a informação necessária para as fases seguintes do processo de compilação, nomeadamente a análise semântica e a geração de código.

O parser foi desenvolvido de forma a permitir a deteção e sinalização de erros sintáticos, indicando situações em que a sequência de tokens não corresponde a nenhuma regra da gramática definida. Sempre que possível, o parser tenta recuperar de erros simples, permitindo a continuação da análise e a identificação de múltiplos erros num único programa.

A separação clara entre as regras do lexer e do parser, aliada à utilização do PLY, contribui para uma implementação modular e de fácil manutenção. Alterações

à gramática da linguagem podem ser efetuadas de forma localizada, sem impacto direto nas restantes fases do compilador.

3.2.1 Implementação

O ponto principal da implementação do parser foi criar uma gramática que fosse capaz de representar a estrutura dos programas feitos seguindo a linguagem Pascal Standard. No nosso caso, criamos uma gramática independente de contexto.

⟨*programa*⟩ ::= `PROGRAM ID ;` ⟨*declaracoes*⟩ ⟨*bloco_final*⟩

⟨*bloco_final*⟩ ::= `BEGIN` ⟨*comandos*⟩ `END .`

⟨*declaracoes*⟩ ::= `VAR` ⟨*lista_declaracoes*⟩
| ϵ

⟨*lista_declaracoes*⟩ ::= ⟨*lista_declaracoes*⟩ ⟨*declaracao*⟩
| ⟨*declaracao*⟩

⟨*declaracao*⟩ ::= ⟨*lista_ids*⟩ `:` ⟨*tipo*⟩ `;`

⟨*lista_ids*⟩ ::= `ID`
| ⟨*lista_ids*⟩ `, ID`

⟨*tipo*⟩ ::= `INTEGER`
| `REAL`
| `BOOLEAN`
| `STRING_TYPE`
| `ARRAY [` ⟨*intervalo*⟩ `] OF` ⟨*tipo*⟩

⟨*intervalo*⟩ ::= `NUM .. NUM`

⟨*comandos*⟩ ::= ⟨*comandos*⟩ ⟨*comando*⟩
| ⟨*comando*⟩

⟨*comando*⟩ ::= ⟨*atribuicao*⟩
| ⟨*atribuicao_array*⟩
| ⟨*comando_if*⟩
| ⟨*comando_while*⟩
| ⟨*comando_for*⟩
| ⟨*comando_writeln*⟩
| ⟨*comando_readln*⟩
| ⟨*bloco*⟩ `;`

⟨*bloco*⟩ ::= `BEGIN` ⟨*comandos*⟩ `END`

⟨*expr*⟩ ::= ⟨*expr*⟩ `+` ⟨*expr*⟩
| ⟨*expr*⟩ `-` ⟨*expr*⟩
| ⟨*expr*⟩ `*` ⟨*expr*⟩
| ⟨*expr*⟩ `/` ⟨*expr*⟩
| ⟨*expr*⟩ `DIV` ⟨*expr*⟩
| ⟨*expr*⟩ `MOD` ⟨*expr*⟩
| ⟨expr⟩ *relop* ⟨expr⟩
| ⟨*expr*⟩ `AND` ⟨*expr*⟩
| ⟨*expr*⟩ `OR` ⟨*expr*⟩
| `NOT` ⟨*expr*⟩
| `-` ⟨*expr*⟩
| `(` ⟨*expr*⟩ `)`
| `NUM`
| `ID`
| `ID [` ⟨*expr*⟩ `]`
| `STRING`
| `TRUE`
| `FALSE`

Todos os programas nessa linguagem começam com um `PROGRAM` e o nome do programa, fazem declarações e tem um bloco principal, `bloco_final`, que contem um `BEGIN`, um `END` e, por fim um `.`.

É exatamente isso que definimos nas primeiras duas produções: o programa começa com essa palavra reservada, com um `ID` referente ao nome dele, com um ponto e virgula a separar, fazendo de seguida as declarações e tendo um bloco principal.

Ora, tendo essa estrutura como base, vamos definir as restantes produções, que são derivadas dessas principais.

Quando às `declaracoes` que fazemos inicialmente, nós sabemos que em Pascal essas podem de facto ser feitas, tendo uma ou mais, ou então não serem feitas, o que equivale a não ter nenhuma. Assim, dizemos na gramática que podemos não ter nenhuma, através da produção do vazio, ou então ter declarações, sendo essas definidas em pascal com `VAR` e tendo, em seguida, a lista de declarações de variáveis que forem feitas.

Essa `lista_declaracoes` pode então conter apenas uma declaração de variável,

obtida pela produção `declaracao`, ou então ter mais do que uma declaração, obtida pela produção `lista_declaracoes declaracao`.

Assim, uma `declaracao` de uma variável é feita através de uma lista de identificadores, ie, o nome das variáveis que estamos a criar, `:` e o `tipo` dessas variáveis, sendo que esse pode ser: `INTEGER`, `REAL`, `BOOLEAN`, `STRING_TYPE`, ou então um `ARRAY [ intervalo ] OF tipo` (este intervalo é dado por um `NUM .. NUM`).

Quando à `lista_ids`, podemos então definir apenas uma variável, dado pela produção `ID`, ou definir mais do que uma, com recursão à esquerda, `lista_ids , ID`.

Dado isso, o parser já consegue então processar a lista de declarações de variáveis. Vamos agora para os `comandos` efetuados dentro do bloco principal.

Podemos então ter apenas um `comando` dentro desse bloco, ou então uma lista deles, dada pela produção `comandos comando`.

Esse `comando` pode ser uma uma `atribuicao`, uma `atribuicao_array`, um if, `comando_if`, um ciclo, podendo ser `comando_while`, `comando_for`, pode ser um `comando_writeln` ou um `comando_readln`, ou até pode ser outro bloco, `bloco PONTOEVIRG`, sendo este também derivado em um `BEGIN comandos END`.

Por fim, podemos ter expressões, `expr`, que podem ser de vários tipos: soma, subtração, multiplicação, divisão, div, mod, operações relacionais ($>$, $<$, $=$, etc...), and, or, not, - (negação), (), NUM, ID, ID[expr] (atribuição a um elemento de um array), STRING, true ou false).

Assim, passamos então à parte de definir, de facto, as funções que implementam esta gramática.

Primeiro, definimos a precedência dos operados, que serve para dizer quais, em expressões, são usados / processados primeiro, ie, os que têm precedência maior são usados primeiro. Quando temos operadores com a mesma precedência, vamos associá-los à esquerda, `left`, ou à direita `right`.

Por exemplo, no caso de termos soma e subtrações, como nesta expressão `a + b - c`, uma vez que usamos left fica `(a+b) - c`.

Assim, na lista, quanto mais a cima, maior é a precedência. Então temos a seguinte ordem: primeiro fazemos o `NOT`, depois as multiplicações e divisões `MULT, DIVIDE, DIV, MOD`, depois a soma e subtração `SOMA, SUBTR`, depois os operadores relacionais `IGUAL, DIFF, MAIOR, etc...`, depois o `AND`, de seguida o `OR`, após o

NEG e só no fim o ELSE.

O ELSE aparecer no final da precedência serve para resolver um problema que obtivemos inicialmente ao implementar a gramática.

Quando tinhamos if aninhados sem delimitadores explícitos, como por exemplo if expr then if expr then comando else comando, surgia ambiguidade, em que o parser não sabia se o else pertencia ao primeiro if ou ao segundo if.

Assim, ao atribuir o else com precedência right, estamos a dizer ao parser que ele tem que associar esse else ao if mais próximo.

De seguida, vamos implementar de facto as função que executam / definem aquela gramática explicada inicialmente.

Começamos pelas duas produções principais de que falamos, a do inicio do programa e da declaração do bloco principal.

A função p_programa implementa a produção inicial. Esta é invocada pelo parser sempre que encontra uma sequência PROGRAM <id> ; <declaracoes> <bloco_final>.

Aqui, os parâmetros p[1], p[2], p[3], p[4] e p[5] correspondem, respetivamente, aos tokens e produções: PROGRAM, ID, PONTOEVIRG, declaracoes e bloco_final.

Assim, essa função constrói um nó na AST, sendo esse nó representado pela tupla ('programa', nome_programa, declaracoes, bloco_principal), armazenando-o em p[0].

Fazemos algo muito semelhante para a função p_bloco_final, que implementa a produção referente ao bloco principal do código Pascal, BEGIN comandos END PONTO.

Aqui, os parâmetros p[1], p[2], p[3] e p[4] correspondem, respetivamente, aos tokens e produções: BEGIN, comandos, END e PONTO.

Assim, essa função constrói um nó na AST, sendo esse nó representado pela tupla ('bloco_principal', comandos), armazenando-o em p[0].

De seguida, vamos para as declarações das funções que serão responsáveis pela parte das declarações iniciais das variáveis que vão ser usadas pelo código Pascal. Essas correspondem às que falamos inicialmente, desde a `declaracoes` até a `intervalo`.

As duas primeiras funções vão corresponder às produções definidas inicialmente em `declaracoes`.

Ora, nós sabemos que, como tínhamos explicado antes, podemos ter ou não declaração de variáveis.

Caso não tenhamos, dizemos que `declaracoes` deriva em vazio, que é definido na função `p_declaracoes_vazio`, em que dizemos que o nó da AST, em `p[0]` fica igual à lista vazia.

Caso tenhamos declarações, usados a primeira função, `p_declaracoes_var`. Nesta, nós sabemos que `p[1]` corresponde à palavra reservada `VAR` e `p[2]` correspode à lista de declarações feitas. Por isso, dizemos que em `p[0]` vamos guardar essa `lista_declaracoes`.

–

De seguida, vamos para a definição da produção `lista_declaracoes`. Como tinhamos explicado antes, esta ou deriva em uma única `declaracao`, ou temos mais do que uma, derivando em `lista_declaracoes declaracao`. É exatamente isso que implementamos.

Na função `p_lista_declaracoes_single` dizemos que temos apenas uma declaração, dizendo que `p[0]` vai guardar essa `declaracao`, representada por `p[1]`.

Na função `p_lista_declaracoes_multi` dizemos que temos mais do que uma declaração, dizendo que `p[0]` vai guardar a lista composta por essas `lista_declaracoes`, representada por `p[1]`, mais a lista feita pela única `declaracao` feita, representada por `p[2]`.

–

Por fim, vamos definir o que é a produção de uma `declaracao`, representado pela função `p_declaracao`.

Começamos por dizer, tal como tínhamos explicado antes, que esta possui então

uma `lista_ids`, com os identificadores das variáveis que são definidas, `DOISPONTOS`, `tipo`, que é o tipo dessas variáveis e, por fim, `PONTOEVIRG`.

Assim, em `p[0]` guardamos um tuplo composto por `'decl'`, ela `lista_ids`, representada por `p[1]`, e por `tipo`, representado por `p[3]`.

–

Esses `lista_ids`, como tinhamos dito antes, podem ser só uma declaração de variável, representada pela função `p_lista_ids_single`, ou então ser a declaração de várias variáveis, representada pela função `p_lista_ids_multi`.

Caso seja só uma declaração, guardamos em `p[0]` apenas o `ID` dessa declaração, sendo esse representado por `p[1]`.

Caso seja mais do que uma, vamos guardar em `p[0]` a lista dos ids já definidos, `lista_ids`, que é a variável `p[1]`, e o `ID` agora definido, representado por `p[3]`.

–

Por fim, vamos tratar o `tipo`destas variáveis.

Sabemos que estes podem ser `INTEGER`, `REAL`, `BOOLEAN` ou então `STRING_TYPE`.

Por isso, na função `p_tipo_base`, atribuimos a `p[0]` o tipo correspondente, guardado em `p[1]`.

Já a função `p_tipo_array` é responsável por tratar dos tipos, quando estes são arrays.

Assim, quando o tipo de uma variável é da forma `ARRAY [ intervalo ] OF tipo`, guardamos em `p[0]` um tuplo composto por `'array'`, o `intervalo` em que esse array está definido (variável `p[3]`), e o `tipo` dos elementos desse array (variável `p[6]`).

Para terminar, o `intervalo` é tratado pela função `p_intervalo`. Dado um `NUM .. NUM`, ela atribui a `p[0]` um tuplo contendo o primeiro `NUM`, guardado em `p[1]`, e o segundo `NUM`, guardado em `p[3]`.

O próximo ponto então da construção da gramática é definir os comandos que são utilizados dentro do bloco principal. Estes são então definidos nas seguintes funções.

Sabemos que `comandos` pode ser dividido em dois casos: ou temos um único `comando`, ou então temos uma lista de comandos, `comandos comando`.

Na função `p_comandos_single` começamos por tratar o caso de só termos um comandos. Assim, em `p[0]` vamos guardar esse único `comando`, representado pela lista de por `p[1]`.

Ao olhar ainda para exemplos de códigos Pascal, verificamos que era comum permitir um `;` final, antes do operador `END`. Por isso, vamos dividir a verificação de termos uma lista de comandos em duas funções.

Na função `p_comandos_multi` tratamos do caso de termos mais do que um comando, em que estes aparecem na forma `comandos ; comando`. Aqui, atribuímos a `p[0]` a lista dos `comandos`, representada por `p[1]`, junto da lista do `comando`, representada por `p[3]`.

Na função `p_comandos_trailing_semicolon` trata quase o mesmo tipo de caso, mas apenas temos a lista dos `comandos` com um `;` final, sem `comando` a seguir. Assim, atribuímos a `p[0]` a lista dos `comandos`, representada por `p[1]`.

Por fim, na função `p_comando` definimos os tipos de comandos que podemos ter, como tínhamos dito antes.

Sabemos que estes podem ser: uma atribuição, uma atribuição de um array, um if, um ciclo while ou for, um writeln ou um readln, ou então um novo bloco.

Assim, nessa função atribuímos a `p[0]` o comando que foi usado, que está em `p[1]`.

–

Agora vamos nos focar na definição específica de cada um desses tipos de comandos apresentados

–

Como tínhamos explicado inicialmente, podemos ter mais blocos do tipo `BEGIN comandos END` dentro do bloco principal. É exatamente isso que é definido aqui.

Caso o parser encontre algo desse género, em `p[0]` vai guardar um tuplo composto por `'bloco'` e os `comandos` que são definidos dentro dele, representado por `p[2]`

–

Quanto às atribuições, estas podem ser feitas normalmente a `ID`, ou então a `ARRAY`.

Quanto aos normais, esses casos são tratados pela função `p_atribuicao` que, dado um `ID ::= expr`, guarda em `p[0]` um tuplo composto por `'atribuicao'`, o `ID` ao qual fazemos a atribuição, representado por `p[1]`, e a `expr` da atribuição, guardada em `p[3]`.

No caso dos arrays, estes são tratador pela função `p_atribuicao_array` que, dado um `ID [expr] ::= expr`, atribui a um elemento do array uma expressão. Ora, guardamos então em `p[0]` um tuplo composto `'atribuicao_array'`, o `ID` ao qual estamos a atribuir, `p[1]`, a `expr` referente ao elemento do array, `p[3]`, e a `expr` que vamos atribuir a esse elemento, `p[6]`.

–

Quanto aos ifs, vamos ter dois casos: o `if` não têm um `else` associado, ou tem um `else` associado.

Para o primeiro caso, usamos a função `p_comando_if` que dado `IF expr THEN comando`, guarda em `p[0]` um tuplo com o `'if'`, a `expr` a ser verificada, `p[2]`, e o `comando` a ser executado caso essa expressão seja verdadeira, `p[4]`.

No segundo caso, usamos a função `p_comando_if_else` que dado `IF expr THEN comando ELSE comando`, guarda em `p[0]` um tuplo com o `'ifelse'`, a `expr` a ser verificada, `p[2]`, o `comando` a ser executado caso essa expressão seja verdadeira, `p[4]`, e o `comando` a ser executado caso essa expressão seja false, `p[6]`.

–

Quanto ao comando `while`, este é tratado com a função `p_comando_while` que, para quando recebe algo do tipo `WHILE expr DO comando`, guarda em `p[0]` o tuplo composto por `'while'`, pela `expr` que comanda o ciclo, `p[2]`, e pelo `comando` que é executado enquanto essa expressão é válida, `p[4]`.

–

Quando ao tratamento do `for`, vamos dividi-lo em dois casos.

No caso de este estar relacionado com a expressão `TO`, ie, ser do género `FOR id ::= expr TO expr DO comando`, vamos tratar com a função `p_comando_for_to`. Esta guarda em `p[0]` um tuplo composto por `'for_to'`, pelo `ID` que aparece na condição, `p[2]` e a expressão que lhe atribuímos, `p[4]`, a expressão que aparece depois do `TO`, `p[6]`, e, por fim, o `comando` a ser executado enquanto essa expressão é válida, `p[8]`.

No caso de este estar relacionado com a expressão `DOWNTO`, ie, ser do género `FOR id ::= expr DOWNTO expr DO comando`, vamos tratar com a função `p_comando_for_downto`. Esta guarda em `p[0]` um tuplo composto por `'for_down'`, pelo `ID` que aparece na condição, `p[2]` e a expressão que lhe atribuímos, `p[4]`, a expressão que aparece depois do `DOWNTO`, `p[6]`, e, por fim, o `comando` a ser executado enquanto essa expressão é válida, `p[8]`.

–

Quando temos o comando a ser um `writeln`, vamos o tratar com a função `p_comando_writeln`. Esta função, quando lê um `WRITELN` *lista_expr*, guarda em `p[0]` um tuplo contendo o `'writeln'` e a `lista_expr` que passamos a esse comando, que é representada por `p[3]`.

Essa `lista_expr` pode ser apenas uma expressão ou várias.

Caso seja só uma, é tratada na função `p_lista_expr_single` que, dada uma `expr`, guarda em `p[0]` a lista composta por ela, que está representada em `p[1]`.

Caso seja mais do que uma expressão, são tratadas na função `p_lista_expr_multi` que, dada uma `lista_expr , expr`, guarda em `p[0]` a lista composta pelas várias expressões, `p[1]`, junto da lista formada pela única, `p[3]`.

–

Quando temos o comando a ser um `readln`, vamos o tratar com a função `p_comando_readln`. Esta função, quando lê um `READLN` *lista_expr*, guarda em `p[0]` um tuplo contendo o `'readln'` e a `lista_expr` que passamos a esse comando, que é representada por `p[3]`.

Para a `lista_expr`, vão ser usadas, novamente, as mesmas funções que foram explicadas agora, `p_lista_expr_single` e `p_lista_expr_multi`.

——

Agora vamos tratar das expressões que podem ser atribuídas.

A função `p_expr_binop` trata todas as operações binárias, que é o caso das operações de `SOMA`, `SUBTR`, `MULT`, `DIVIDE`, `DIV`, `MOD`, das operações relacionais, como `IGUAL, DIFF, MAIOR, MENOR, MAIORIGUAL, MENORIGUAL`, e das lógicas, `AND, OR`. Quando ele encontra algo do género `expr operador expr` guarda em `p[0]` um tuplo composto por `'op'`, o operador utilizado, `p[2]`, a expressão à esquerda, `p[1]`, e a da direita, `p[3]`.

A função `p_expr_not` trata da negação lógica. Quando recebe um `NOT expr`, guarda em `p[0]` um tuplo contendo o `'not'` e a `expr` a ser negada, `p[2]`.

A função `p_expr_neg` trata da negação unária. Quando recebe uma expressão precida por -, guarda em `p[0]` um tuplo contendo o `'neg'` e a `expr` a ser negada, `p[2]`.

A função `p_expr_paren` trata de expressões entre parênteses. Quando recebe um *expr*, guarda em `p[0]` a `expr`, `p[2]`.

A função `p_expr_num` trata constantes numéricas. Quando recebe um `NUM`, guarda em `p[0]` um tuplo contendo o `'num'` e o valor numério, `p[1]`.

A função `p_expr_id` trata identificadores (nomes de variáveis). Quando recebe um `ID`, guarda em `p[0]` um tuplo contendo o `'id'` e o identificador, `p[1]`.

A função `p_expr_array_acess` trata acessos a elementos de um array. Quando recebe um `ID [expr]`, guarda em `p[0]` um tuplo contendo o `'array_acess'`, o identificador do array, `p[1]`, e o elemento a que estamos a aceder, `p[3]`.

A função `p_expr_string` trata strings. Quando recebe um `STRING`, guarda em `p[0]` um tuplo contendo a `'string'` e o valor desta, `p[1]`.

As funções `p_expr_true` e `p_expr_true` tratam dos valores booleanos. Quando recebe um `TRUE` ou um `FALSE`, guarda em `p[0]` um tuplo contendo apenas o `'true'` ou o `'false`, respetivamente.

Por fim, vamos tratar dos erros sintáticos.

A função `p_error` é invocada pelo parser sempre que este encontra um erro de

sintático, ie, quando recebe um token que não consegue processar de acordo com a gramática definida. Assim, ela primeiro verifica se existe um token problemático, que é representado por `p`.

Caso exista, extrai o número da linha onde esse erro ocorreu, e imprime uma mensagem de erro descritiva, indicando a linha do erro, o token que o gerou e o valor que tinha sido atribuído a este.

Caso este seja `None`, significa que o parser chegou ao fim do ficheiro de código Pascal de forma inesperada, ie, sem conseguir completar uma derivação válida, pelo que imprime uma mensagem de erro contendo essa informação.

Finalmente, define o `succes` do parser como sendo `False`, o que indica que a análise sintática falhou.

Por fim, criamos uma instância do parser, que irá processar todas as produções definidas e gerar uma AST. A variável `parser.succes` é inicializada a `True`, indicando que, por defeito, nenhum erro foi detetado até ao momento.

3.3 Análise semântica

A análise semântica é responsável por verificar se o programa é **correto do ponto de vista do significado** das instruções, garantindo que não existem incoerências que poderiam causar erros durante a execução. Esta fase utiliza a Árvore Sintática Abstrata (AST) gerada pelo parser e aplica regras relativas a tipos, declarações e utilização de variáveis.

Uma das principais estruturas utilizadas nesta fase é a **tabela de símbolos**, onde são registadas informações sobre variáveis, constantes, tipos e funções/procedimentos declarados no programa. Para cada elemento, a tabela armazena o nome, o tipo, o escopo e outras propriedades relevantes.

A análise semântica realiza as seguintes verificações:

- **Declaração prévia**: todas as variáveis e funções/procedimentos devem ser declarados antes de serem utilizados.
- **Compatibilidade de tipos**: operações aritméticas e lógicas são verificadas para garantir que os operandos têm tipos compatíveis.
- **Atribuições**: o valor atribuído a uma variável deve ser compatível com o seu tipo.

Sempre que um erro semântico é detetado, o compilador regista a informação relevante (tipo de erro, linha e contexto) de forma a que o programador possa corrigi-lo. A fase de análise semântica garante que apenas programas válidos prosseguem para a geração de código.

A implementação da análise semântica foi realizada de forma **modular**, permitindo adicionar ou modificar regras facilmente, e integrando-se diretamente com a AST produzida pelo parser.

3.3.1 Implementação

Inicialmente, criamos a classe `SemanticError` que é uma exceção personalizada, utilizada para sinalizar **erros semânticos** relacionados com declarações, como por exemplo declarações duplicadas ou a utilização de um id não identificado. Isto permite que haja um tratamento diferente consoante o tipo de erro, ao longo da análise semântica.

A classe `Symbol` serve para representar um símbolo na tabela de símbolos, podendo ser uma variável normal ou um array. Quando se cria uma instância desta classe, armazena-se informações referentes ao identificador:

- `name`: nome do identificador, como por exemplo, x;
- `kind`: o tipo do símbolo, que pode ser `var` no caso de uma variável normal ou `array` no caso de um array;
- `type_name`: o tipo do símbolo, como `integer`, `real`, `string`, ou `array` no caso de ser um array;
- `offset`: a posição na memória reservada para o símbolo, que vai ser usada na parte da geração de código para a VM;
- Reservamos ainda campos específicos para arrays:
  - `lower`: limite inferior do intervalo do array;
  - `upper`: limite superior do intervalo do array;
  - `elem_type`: tipo dos elementos do array, como `integer`, `real`, `string`;

A classe `SymbolTable` vai gerenciar a tabela de símbolos de um dado programa, armazenando informações sobre todas as variáveis e arrays que foram declarados.

Esta é inicializada como um dicionário vazio, em `_symbols` (mapeia nomes de ids para os respectivos objetos `Symbols`), e com `_next_offset` a 0 (rastreia a próxima posição de memória disponível).

A função `define_var` é responsável por registrar uma variável simples na tal tabela de símbolos.

Começa por verificar se já foi declarada uma variável com esse `name` na tabela de símbolos e, caso exista, lança uma exceção, dizendo que houve um erro semântico derivado do facto de se ter uma declaração duplicada.

Caso contrário, cria um novo símbolo do tipo `'var'` com o offset atual, adicionando-o ao dicionário, incrementando o offset, e retornando o símbolo criado.

A função `define_array` é responsável por registrar um array na tabela de símbolos.

Começa por verificar se já foi declarado algo com esse `name` na tabela de símbolos e, caso exista, lança uma exceção, dizendo que houve um erro semântico derivado ao facto de se ter uma declaração duplicada.

Caso contrário, cria um novo símbolo do tipo `'array'`, associando-lhe os limites superiores e inferiores e o tipo dos elementos fornecidos. Esta ainda reserva múltiplas posições de memória consecutivas, calculando o número de elementos do array, e reservando esse número, e, no fim, incrementando o `_next_offset` com base nesse valor.

A função `lookup` procura um símbolo pelo seu nome no dicionário de símbolos. Caso esse exista, retorna o objeto `Symbolo` correspondente e, caso não exista, retorna `None`. Vai ser usada para verificar se um id já foi previamente declarado.

A função `items` serve para retornar a lista de todos os símbolos, ie, pares `(nome, simbolo)` armazenados da tabela de símbolos.

Assim, definimos a classe principal, `SemanticAnalyser`, que é a componente central desta análise semântica, sendo responśavel por validar a semântica de um dado código em Pascal Standard.

Começamos por, em `__init__` dizer que, quando se cria uma instância desta classe, incializamos a tabela de símbolos, `.symtab`, como sendo uma instância da classe `SymbolTable`, e inicializamos ainda a lista `errors` como vazia, uma vez que, inicialmente, ainda não foram encontrados nenhuns erros semânticos.

De seguida, criamos uma função `_norm_type_token` responsável por converter os tipos vindos de uma AST do parser para uma forma uniforme, sendo essencial para garantir que a comparação de tipos seja feita de forma justa.

Assim, quando recebe um tipo `t` ela vai verificar:

- Se `t` é do tipo `None`, então apenas retorna esse tipo;
- Se `t` é um tuplo, pode representar um array.
  - Se for uma tupla que contém 3 elementos, e o primeiro é `'array'`, então estamos de facto perante um array, pelo que obtemos o intervalo deste, guardado na primeira componente, e os elementos, guardados na segunda componente. De seguida, os tipos dos elementos são normalizados, retornando no final o mesmo tipo de tupla, só que normalizada.
  - Caso não seja esse tipo de tupla, apenas retorna o `t`.
- Se `t` é uma string, então apenas o normalizamos, convertendo-o para minúsculas. Por exemplo, se for `INTEGER` fica `integer`, `REAL` -> `real`, etc.

Definimos a função `error` que quando se obtém um erro semântico ao longo da análise, os métodos podem invocar esta função, sendo que ela adiciona a mensagem de erro à lista `errors` do analisador semântico.

A função `analyze` é a função principal da análise semântica. Esta vai receber a AST produzida pelo parser e verificar a semântica completa.

Inicialmente, ela limpa a lista de erros anterior, garantindo que começa a fazer uma nova análise.

De seguida, vai verificar se o nó raiz é uma tupla que comece com `'programa'` e que contém pelo menos um elemento, uma vez que sabemos que um programa em Pascal começa sempre com uma linha `program ...`.

Se a AST não cumprir esse requisito, vai se registrar um erro informando isso, e retorna-se `False`.

Caso contrário, vai se tentar extrair as três componentes desse tuplo, o nome do progama, `prog_name`, a lista de declarações feitas, `declarations`, e o bloco principal, `block_node`.

Caso a AST tenha uma estrutura que não essa, adicionamos à lista `error` esse erro, e retorna-se `False`.

Asim, com recurso à função `_.process_declarations` vai se processar todas as declarações feitas.

De seguida, extrai-se os comandos feitos no bloco principal, criando uma lista `commands` para os armazenar. O bloco pode estar representado de duas formas:

- Com uma tupla com rótulo `'bloco_principal` ou `'bloco'`, cujo segundo elemento da tupla é uma lista de comandos;
- Ou diretamente como uma lista de comandos.

Se o bloco não corresponder a nenhum destes casos, retorna-se `False` e guarda-se o erro correspondente.

Depois de os extrair, vamos chamar a função `_process_command` para cada um dos comandos, responsável por os processar.

No fim, caso não tenha havido erros, devolve `True`.

Esta função `_process_declarations` é responsável então por processar todas as declarações das variáveis e array acima referidos, sendo que recebe então a lista dessas declarações vindas do parser, sendo que cada elemento tem o formato `('decl', lista_ids, tipo)`.

Ela começa por verificar se não existe nenhuma declaração e, nesse caso, diz que as declarações são válidas (já que no código Pascal não se faz nenhuma).

Caso existam declarações, vai iterar por cada uma delas naquela lista.

Para cada uma, começa por verificar o seu formato: tem de ser um tuplo, com pelo menos 3 elementos, e o primeiro tem de ser `'decl'`. Se isso não for verificado, regista um erro semântico, e verifica as restantes declarações.

Se a declaração é válida, vai extrair os seus elementos:

- `ids`, que correspondem à lista de identificadores declarados (por exemplo, `x, y, z`);
- `vtype`, que correspondem ao tipo, tal como vêm do parser;
- `vtype_norm`, que corresponde ao tipo normalizado, através da função `_norm_type_token`.

A função então agora apresenta dois casos.

Se o tipo normalizado é uma tupla, com o rótulo `'array'`, vamos processar uma **declaração de array**, extraindo-se o intervalo, e o tipo dos elementos desse array.

Primeiro, vamos verificar se o intervalo é uma tupla com exatamente dois elementos e, caso não seja, regista-se um erro, passando à próxima declaração.

Depois, vamos extrair os limites inferior e superior desse intervalo, e verificar se ambos são inteiros e, caso algum não o seja, regista-se outro tipo de erro, passando novamente à próxima declaração.

De seguida, vamos validar a lógica do intervalo, ie, verificar se o extremo inferior não é maior que o extremo superior. Caso seja, regista-se outro tipo de erro, e passa-se à próxima declaração.

Por fim, se todas as validações passarem, para cada id na lista `ids`, invoca-se a função que `define_array`, registrando o array na tabela de símbolos. Caso essa chamada lance uma exceção (por exemplo, id já declarado), captura-se a tal exceção e regista-se a mensagem de erro.

Se o tipo normalizado não é um `'array'`, vamos processar uma **declaração de tipos simples**.

Verifica-se primeiro se é uma string, que indica um tipo primitivo (como `integer, real, boolean, string`).

Se for, invoca-se a função que `define_var`, registrando a variável na tabela de símbolos. Caso essa chamada lance uma exceção, captura-se essa exceção e regista-se a mensagem de erro.

Se não for uma string e nem um array, significa que é um tipo desconhecido, não registado pelo analisador e, nesse caso, regista-se um erro indicando que o tipo da declaração é inválido.

A função `_process_command` é responsável por processar os comandos que se encontram dentro daquele bloco principal ou, futuramente, em outros blocos declarados dentro desse. Assim, vai receber um comando, `cmd`, e analisá-lo semanticamente.

Começa então por verificar se esse comando é `None`. Nesse caso, ele é válido, já que não há nada a processar, pelo que retorna logo.

De seguida, verifica se é uma lista, pelo que se sabe então que temos uma lista de comandos. Para isso, itera sobre cada comando e, para cada um deles, chama a função `_process_command` novamente.

Se não é `None` e nem uma lista, vai validar o formato esperado: tupla não vazia. Caso isso não seja cumprido, regista um novo erro, retornando logo.

Por fim, a função extrai o `kind`, que é o primeiro elemento da tupla e que representa o tipo do comando que estamos a processar (por exemplo, atribuição, writeln, if, etc).

Dependendo do tipo, vamos fazer coisas diferentes.

Atribuição Simples

Quando o comando é uma atribuição simples, vindo do parser como `('atribuicao', var, expr)`, começamos por extrair o identificador da variável, `var_name`, e a expressão que lhe está a ser atribuída, `expr`.

De seguida, vai procurar na tabela de símbolos o símbolo correspondente a `var_name`. Se esse símbolo for `None`, significa que foi utilizada uma variável não declarada, pelo que regista um erro semântico, mas, mesmo assim, avalia o tipo da expressão, com `_eval_expr_type`, para tentar capturar mais erros.

Se a variável foi declarada, vai avaliar o tipo da expressão, com `_eval_expr_type`.

Invoca ainda `_check_assignment_types`, que verifica se o tipo da expressão é compatível com o tipo da variável, aplicando as regras de compatibilidade de tipos do Pascal.

Atribuição a arrays

Quando o comando é uma atribuição a um elemento de um array, recebido pelo parser no formato `('atribuicao_array', nome_array, expr_indice, expr)`, começa-se por extrair o identificador do array, `name`, a expresão que representa o seu índice, `index_expr` e a `expr`.

Começa-se então por procurar o símbolo correspondente a `name` na tabela de símbolos.

Se não existir, regista-se um erro indicando que está a tentar se o usar um array que não foi declarado. Avalia-se ainda o tipo de expressões usadas no índice do array e o tipo de expressões de `expr`.

De seguida, verifica-se se o símbolo encontrado é efetivamente um array. Se não for, regista-se essa informação num erro.

Depois, avalia-se o tipo de expressão de índice, que deve obrigatoriamente ser um `integer`. Se for de outro tipo, regista-se um erro semântico.

Além disso, vai se verificar os limites do intervalo, caso este seja um literal. Para isso, verifica-se se a expressão de índice, `index_expr` é uma tupla com rótulo sendo `'num'`, contendo um inteiro.

Se for, extraímos o valor literal do índice e comparamos com os limites superior e inferior do intervalo. Se o índice a que estamos a tentar aceder do array não estiver dentro desses limites, registramos um erro semântico.

Posteriormente, avalia-se o tipo da expressão a ser atribuída, `expr`. Se o tipo não é `None`, vai se verificar a compatibilidade com o tipo dos elementos do array:

- Se o tipo da expressão coincide exatamente com o tipo dos elementos, a atribuição é válida;
- Se o tipo dos elementos é `real` e o tipo da expressão é `integer`, também se permite a atribuição;
- Em qualquer outro caso, a atribuição não é válida, pelo que se regista um erro de incompatibilidade de tipos.

Writeln

Quando o comando é um `writeln`, recebido pelo parser como `('writeln', lista_expr)`, o analisador semântico começa por obter a lista de expressões a serem impressas, `exprs`.

De seguida, vai verificar se isso é de facto uma lista e, se não for, guarda um erro, dizendo que o `writeln` recebeu argumentos inválidos.

Assim, vai então iterar sobre cada expressão dessa lista e, para cada uma delas, vai avaliar o seu tipo através da função `_eval_expr_type`.

Caso isso retorne `None`, significa que o erro já houve um erro registado durante a avaliação da expressão, pelo que passa à próxima.

Se foi possível determinar o tipo, vai verificar se é possível o imprimir. Uma vez que em Pascal só se pode usar essa função para `integer, real, boolean, string`, se a expressão não for de nenhum desses tipos, regista um erro semântico, dizendo que esse tipo não é imprimível.

Readln

Quando o comando é um `readln`, recebido pelo parser como `('readln', lista_args)`, o analisador semântico começa por obter a lista de argumentos de destino.

De seguida, vai verificar se isso é de facto uma lista e, se não for, guarda-se os `args` numa lista, permitindo que um único argumento seja processado como lista com um elemento.

De seguida, vai se iterar sobre cada elemento dessa lista.

Se o argumento é um tuplo, com o rótulo `id`, então estamos perante uma **variável simples**, pelo que se extrai o seu nome, `vname`, e procura-se tal símbolo na tabela de símbolos.

Se o símbolo não existir, regista-se um erro dizendo que a variável não foi declarada.

Se existir, vamos verificar primeiro se o símbolo é efetivamente uma variável, `var`, e não um array ou outro tipo. Se não for uma `var`, regista-se um erro dizendo isso.

Depois, vai se verificar se o tipo da variável é suportado para leitura, ie, se é um `integer, real, string` e, se não for, regista o devido erro semântico.

Se o argumento é um tuplo, com o rótulo `array_acess`, então estamos perante um **acesso a array**, pelo que se extrai o seu nome, `arr_name` e a expressão referente ao índice que estamos a aceder, `idx_expr` , e procura-se tal símbolo na tabela de símbolos.

Se não existir na tabela, regista-se um erro dizendo isso.

Se existir, vai se verificar se é efetivamente um array e, caso não seja, regista-se outro tipo de erro semântico.

De seguida, vai-se avaliar se o tipo da expressão que se refere ao índice a ser acedido é de facto um `integer`. Se não for, regista-se outro erro.

Vai se verificar também se o índice que foi recebido se encontra dentro dos limites do array. Para isso, se a expressão do índice é um tuplo com rótulo `num`, extraímos o valor literal e vamos comparar os limites do array: caso o índice a ser acedido estiver fora desses limites, registramos um erro dizendo exatamente isso.

Por fim, verifica-se se o tipo de elementos do array é suportado para o `readln`, sendo que este apenas pode ler `integer, real, string`. Caso o tipo seja diferente, regista-se um erro semântico.

Para terminar esta função, caso o argumento não seja nem uma variável simples e nem um acesso a array, regista-se um erro dizendo que o destino é inválido.

If

Quando o comando é um `if` simples (sem else), recebido pelo parser como `('if', expr_condicao, comando)`, o analisador semântico começa por extrair a expressão condicional, `cond` e o comando a executar se essa for verdadeira, `then_cmd`.

De seguida, vai avaliar se o tipo da expressão, obtido com `_eval_expr_type`, é um `boolean`. Se não for, regista-se um erro semântico informando que não o é.

Se for, vai se processar o comando `then_cmd` invocando recursivamente a função `_process_command`.

IfElse

Quando o comando é um `if-else`, recebido pelo parser como `('ifelse', expr_condicao, comando_then, comando_else)`, o analisador semântico começa por extrair a expressão condicional, `cond`, o comando a executar se essa for verdadeira, `then_cmd`, e o comando a executar se ela for falsa, `else_cmd`.

De seguida, vai avaliar se o tipo da expressão, obtido com `_eval_expr_type`, é um `boolean`. Se não for, regista-se um erro semântico informando que não o é.

Se for, vai se processar o comando `then_cmd` e o comando `else_cmd`, invocando recursivamente a função `_process_command`.

While

Quando o comando é um `while`, recebido pelo parser como `('while', expr_condicao, comando_corpo)`, o analisador semântico começa por extrair a expressão condicional, `cond` e o comando do corpo do ciclo, `body`.

De seguida, vai avaliar se o tipo da expressão, obtido com `_eval_expr_type`, é um `boolean`. Se não for, regista-se um erro semântico informando que não o é.

Se for, vai se processar o comando `body`, invocando recursivamente a função `_process_command`.

For

Quando o comando é um `for`, este pode ser de dois tipos

- `for_to`, vindo do parser como `('for_to', var, expr_inicio, expr_fim, comando_corpo)`;
- `for_down`, vindo do parser como `('for_down', var, expr_inicio, expr_fim, comando_corpo)`;

Assim, primeiro vai se extrair a variável de controlo, `var_name`, as expressões de ínicio e fim de ciclo, `start_expr, end_expr`, e o comando do corpo do ciclo, `body`.

De seguida, vai-se primeiro procurar essa variável de controlo na tabela de símbolos e, caso essa não exista, regista-se um erro semântico, indicando que a variável não foi declarada.

Depois, se existir, vai se verificar se é uma `var` simples (não é array) e que o seu tipo é `integer`, uma vez que em Pascal a variável de controlo de um `for` deve ser um inteiro. Se não cumprir essas condições, regista-se um novo erro semântico.

Seguidamente, avaliamos o tipo das expressões de início e de fim de ciclo, através da função `_eval_expr_type`, sendo que ambas deve ser do tipo `integer`. Caso alguma não seja, regista-se um erro semântico.

Por fim, processa-se o corpo do ciclo, `body`.

Bloco de comandos

Quando o comando é um `bloco`, ou um `bloco_principal`, começamos por extrair a lista de comandos contida nesse bloco, `cmds`.

Se isso é de facto uma lista, itera-se sobre cada comando, processando-os recursivamente, com `_process_command`.

Caso seja um único comando, vamos também o processar com `_process_command`.

Comando desconhecido

Se nenhum dos tipos de comandos acima referidos foi identificado, significa que estamos perante um comando desconhecido, pelo que o analisador semântico tem que fazer um tratamento de fallback.

Se esse comando é uma lista, vai iterar por cada comando da lista, chamando recursivamente a funçaĩ `_process_comando`.

Se não for uma lista, vai registar um erro semântico, informando que não é conhecido esse comando.

Esta função `_eval_expr_type` é responsável então por inferir o tipo de uma expressão e retornar esse tipo.

Inicialmente, verifica-se se a `expr` é `None` e, caso seja, regista-se um erro semân-

tico, retornando também `None`,.

Caso a expressão não seja uma tupla (formato esperado para nó da AST), então regista-se um erro semântico com essa informação.

De seguida, vai se extrair `tag`, que corresponde ao primeiro elemento da tupla, correspondente ao tipo de nó da expressão.

Literais

Para literais numéricos, representados no parser por `('num', valor)`, começa-se por extrair o valor, `v`, e determina-se o seu tipo.

Se o valor é inteiro, então retorna-se `integer`. Caso contrário, assume-se que é um número `real`.

Caso seja uma `string`, retorna-se isso mesmo.

Caso a tag seja `true / false`, estamos perante um `boolean`.

Identificadores

Para identificadores, representados por `('id', nome)`, começa-se por extrair esse `name` e por procurar na tabela de símbolos o símbolo correspondente.

Caso esse não exista, regista-se um erro semântico dizendo que essa variável não foi declarada.

Se o símbolo existir, e for do tipo `var`, retorna-se o tipo declarado. Se for um array, retorna-se `array`. Se for qualquer outro tipo desconhecido, regista-se um erro semântico informando isso.

Acesso a Arrays

Para acesso a arrays, representados por `('array_acess', nome_array, expr_indice)`, começa-se por extrair o nome do array, `name`, e a expressão que representa o índice a que estamos a aceder, `idx`.

De seguida, procura-se na tabela de símbolos tal símbolo, pelo que, se ele não existir, regista-se um erro semântico de array não declarado. Vai se ainda avaliar o índice, de modo a capturar mais erros.

Se ele existir na tabela, vamos verificar se é de facto um array e, caso nã seja, registramos um erro informando isso e, ainda, avaliamos novamente o índice.

Se não estiver em nenhum dos casos acima, avalia novamente o índice e se este

não for um `integer`, vai registar um erro.

Além disso, vai verificar os limites: se o índice a que tenta aceder é um literal, e não está dentro dos limites do intervalo do array, regista um erro semântico.

Finalmente, retorna o tipo dos elementos do array.

Operadores binários

Para cada operador binário, representado por `('op', operador, expr_esq, expr_dir)`, extrai-se o operador, `op`, e as expressões relacionadas, `left` e `right`.

De seguida, avalia-se recursivamente o tipo de ambas as expressões, e se nenhuma conseguir ser avaliada, retorna-se `None`.

Normalizamos ainda o operador para minúsculas, armazenando em `lop`.

De seguida temos diferentes casos.

- **Operadores aritméticos (+, -, *)**: estes operadores requerem operandos numéricos (inteiros ou reais). Se ambos são numéricos, o resultado é `real` se qualquer um deles for real e, caso contrário, é `integer`. Se algum deles não é numérico, regista-se um erro;
- **Divisão real (/)**: esta retorna sempre um `real`, mesmo que ambos os operandos sejam inteiros;
- **Div e Mod**: só funcionam com inteiros, retornando sempre um `integer`. Se algum deles não for inteiro, regista-se um erro;
- **= e <>**: comparam dois valores e retornam `boolean`, sendo que só são permitidas comparações entre tipos iguais, ou entre inteiros e reais. Se os tipos são incompatíveis, regista-se um erro semântico;
- **<, <=, >, >=**: comparam números (inteiros ou reais), ou strings, retornando sempre `boolean`. Podemos, em Pascal, comparar entre tipos mistos de números mas, se os tipos não forem compatíveis, regista um erro;
- **and e or**: requerem que ambos os operandos sejam do tipo `boolean`, retornado isso também. Assim, se algum deles não o for, regista um erro;
- **desconhecido**: se o operador não é conhecido, regista ym erro semântico.

Operadores unários

- **Negação (-)**: esta pode ser aplicada apenas a tipos numéricos, pelo que se verifica se a sub-expressão é `integer` ou `real`. Se não for, regista-se um erro;
- **Not (not)**: esta pode ser aplicada apenas a `boolean`, pelo que se a sub-expressão não o for, regista um erro.

Fallback

Se nenhum tipo acima foi identificado, estamos perante uma expressão cujo tipo é desconhecido, pelo que registramos um erro semântico informando isso.

Esta função `_check_assignment_types` é responsável por validar se o tipo de uma expressão é compatível com o tipo de uma variável na atribuição.

Inicialmente, ela verifica se o tipo da expressões é `None` e, caso seja, significa que já houve erros durante a avaliação da expressão, pelo que apenas retorna.

Se seguida, vai verificar se o símbolo é uma `var` simples (não array) e, se não for regista um erro semântico.

Se for, extrai-se o tipo declarado, `var_type`, e compara-se com o tipo da expressão, `expr_type`.

Se estes forem iguais, a atribuição é válida, pelo que se retorna logo.

Se a variável é do tipo `real` e a expressão é do tipo `integer`, a atribuição também é válida, uma vez que no Pascal se permite essas conversões.

Se a variável é do tipo `integer` e a expressão é do tipo `real`, a atribuição não é válida, uma vez que no Pascal tal não é permitido, e, assim, regista-se um erro semântico.

Por fim, se nenhum dos casos acima se aplica, os tipos não são compatíveis, pelo que se regista um erro informando isso.

Esta função `_report_erros` é responsável por apresentar todos os erros semânticos encontrados.

Se a lista `self.erros` se encontra vazia, significa que não foi encontrado nenhum erro semântico, pelo que se imprime essa informação.

Se ela tem elementos, significa que ocorrem erros semânticos, pelo que os imprimimos um a um.

3.4 Geração de código para a máquina virtual

A fase de geração de código tem como objetivo transformar a *AST* construída nas fases anteriores em instruções executáveis pela máquina virtual. Esta fase percorre a árvore de forma recursiva, convertendo cada nó numa sequência de instruções que preserva a semântica do programa fonte.

O gerador de código foi implementado na classe CodeGenerator, que mantém uma lista de instruções e uma tabela de símbolos, permitindo aceder a informações sobre variáveis, arrays e seus tipos. A função principal desta fase, generate, recebe a *AST* do programa e inicia a tradução a partir da função gen_program, que controla a emissão das instruções iniciais e finais do programa.

Cada bloco de instruções é processado pela função gen_bloco, enquanto gen_stmt identifica o tipo de cada comando e delega para funções específicas, como:

Atribuições: Avalia a expressão associada à variável e armazena o resultado na posição correta da memória, realizando conversões de tipo quando necessário (por exemplo, de inteiro para real).

Leitura e escrita de dados: Processa comandos de entrada (readln) e saída (writeln), tratando os tipos de dados adequadamente e convertendo valores quando necessário.

Estruturas de controlo: Ciclos (for, while) e condicionais (if, ifelse) são traduzidos em instruções da máquina virtual com a criação de etiquetas únicas para controlar o fluxo de execução.

Acesso e atribuição a arrays: Calcula endereços baseados na posição inicial e no índice, garantindo que os valores corretos sejam lidos ou armazenados.

As expressões aritméticas, lógicas e de comparação são processadas pela função gen_expr, que identifica os tipos dos operandos e gera instruções correspondentes. Operações envolvendo números reais e inteiros são tratadas com conversão automática para garantir a coerência dos cálculos. Operadores como soma, subtração, multiplicação, divisão, comparações e operações lógicas são traduzidos para instruções adequadas da máquina virtual.

Esta abordagem modular permite que cada tipo de instrução seja tratado de forma independente, facilitando a manutenção e testes do compilador, e garantindo que o programa fonte seja corretamente interpretado e executado na máquina virtual.

3.4.1 Implementação

A classe `CodeGenerator` é responsável por converter a AST criada pelo parser e validada pela análise semântica em código, sendo este usado na Máquina Virtual disponibilizada.

Quando se cria uma instância desta classe, inicializa-se as variáveis:

- `symtab`, que representa a tabela de símbolos;
- `code`, que vai ser uma lista, inicialmente vazia, que vai guardar as instruções da máquina virtual geradas durante esta fase;
- `label_counter`, que é um contador inteiro, inicializado a 0, para regar rótulos no código da VM. Por exemplo, quando temos loops, através de exemplos de códigos da VM, vemos que existem jumps para diferentes labels. Aqui, esse contador serve para gerar o nome dessas labels.

A função `emit` é uma função auxiliar usada para adicionar instruções à lista de código já produzido para a VM. Assim, recebendo uma `instr`, adiciona-a ao final da lista `code`.

A função `new_label` é uma função auxiliar usada para gerar rótulos únicos, aqueles usados para jumps de ciclos, por exemplo. Recebendo um prefixo fixo, que por defeito é `'L'`, retorna uma string combinando esse prefixo com o contador incrementado, garantindo que não se gera labels iguais.

A função `_ensure_float_operands` é uma função auxiliar usada para garantir que dois operandos estão no formato correto para operações aritméticas.

Assim, recebendo dois operandos, `lt` e `rt`, começa por verificar se ambos são `real` e, caso sejam, não é preciso fazer nada.

Caso o `lt` seja `real`, mas o `rt` seja um `integer`, precisamos de o converter para `real`. Pela lógica da pilha apresentada na VM, `rt` estará no topo dela, pelo que só se emite uma instrução `ITOF` para o converter.

Caso seja ao contrário, `lt integer` e `rt real`, nós precisamos converter o primeiro. Mas, no topo da stack encontra-se o `rt`. Então trocamos estes, fazendo `SWAP`, e depois sim fazemos a conversão `ITOF` e voltamos a colocá-los nas posições iniciais com outro `SWAP`.

Caso ambos sejam do tipo `integer`, precisa-se converter ambos. Primeiro convertemos o `rt`, com `ITOF`, trocamos a ordem dois dois com o `SWAP` e convertemos o `lf` com o `ITOF`. No final, volta-se a trocar as posições deles.

A função `_expr_type` é usada para determinar o tipo de uma expressão presente na AST, utilizando uma lógica bastante semelhante à que foi usada na análise semântica. Será útil pois, ao longo da geração de código, precisamos saber qual operação usar, inteira ou real, ou se é necessário converter tipos das variáveis.

Assim, começamos por obter a `tag`, que guarda o tipo da variável que está no `node` da AST.

Caso esse tipo seja um `num`, vamos verificar se o valor guardado nesse nó é um float ou não. Caso seja, então estamos perante um `real`, e se não for é um `integer`.

No caso de termos uma `string`, retornamos exatamente isso, e se tivermos algo como `true / false`, retornamos que é um `boolean`.

Se a tag for um `id`, então consultamos a tabela de símbolos, e obtemos o tipo dessa variável e, caso encontre, retornamos esse tipo.

No caso de termos um `array_acess`, vamos novamente consultar o tipo dos elementos desse array à tabela de símbolos, retornando-o se possível.

Para operadores, `op`, começamos por extrair o operador, e as expressões relacionadas por ele, `l` e `r`, obtendo o tipo dessas expressões através de `_expr_type`. De seguida, aplica-se as regras do Pascal:

- Se o operador é `/`, devolve `real`;
- Se o operador é `div, mod`, devolve `integer`;
- Se o operador é `+,-,*`, devolve `real` se algum operando é desse tipo, ou `integer` caso contrário;
- Se o operador é `and, or`, devolve `boolean`;

Para a `neg`, retornamos o tipo da expressão negada.
Por fim, caso tenhamos `not`, retornamos um `boolean`.

A função `generate` é o ponto principal desta fase, sendo que recebe uma AST e retorna a string com todas as instruções geradas. Esta invoca a função `gen_program`.

Essa função `gen_program` é responsável por processar o nó raiz dessa AST, que possui o formanto `('programa', nome, decls, bloco)`. Assim, extrai o bloco de comandos, envolvendo a sua geração de código dentro de instruções `START` e `STOP`.

Por fim, a função `gen_bloco` vai fazer e geração de instruções para esse bloco principal. Ela vai pegar na lista de comandos, `stmts`, presentes nesse bloco e, para cada comando dentro dessa lista, vai gerar o código correspondente, com a função `gen_stmt`.

A função `gen_stmt` é a função principal para gerar o código da VM para os comandos presentes no bloco principal. Recebendo como argumento a AST, ela começa por extrair a `tag`, que identifica o tipo de comando com que ela está a lidar. Ora, dependendo desse tipo de comando, vai chamar a função correspondente e caos obtenha um tipo não conhecido, lança uma exceção, dizendo que esse tipo ainda não foi implementado no gerador de código.

Atribuição

No caso de se ter um tipo `atribuicao`, chama-se esta função `gen_atribuicao`.

Recebendo um nó do tipo `('atribuicao', nome_var, expr)`, vai extrair essas componentes, guardando em `var` e em `expr`.

De seguida, vai procurar o símbolo correspondente à variável na tabela de símbolos, obtendo o seu tipo.

Assim, vai gerar o código da VM para a expressão através de `gen_expr`.

Por fim, vai se verificar se o tipo da variável é `real` e se o tipo da expressão é `integer`. Nesses casos, é preciso realizar uma conversão, pelo que se emite uma instrução `ITOF`.

Após isso, adiciona-se a instrução `STOREL offset`, que pega no último elemento da pilha e o armazena na variável identificada pelo seu offset.

Readln

No caso de se ter um tipo `readln`, chama-se esta função `gen_readln`.

Recebendo um nó do tipo `('readl', lista_args)`, começa por extrair essa

lista e vai iterar por cada elemento dela.

Caso estejamos perante um argumento que seja `id`, estamos perante uma variável simples, pelo que extraímos o seu nome, `var`, e procuramos-la na tabela de símbolos, emitindo já uma instrução `READ`. De seguida, convertemos a string para o devido tipo: se `var` é do tipo `integer`, usa-se `ATOI`, e caso seja do tipo `real` usa-se `ATOF`. No fim, emite-se `STOREL offset`, que pega no valor convertido e armazena-o na variável.

Caso estejamos perante um argumento que seja `array_acess`, extrai-se o nome do arrar, `var`, e a expressão referente ao índice a que queremos aceder, `index_expr`, procurando na tabela de símbolos o array. De seguida, vai se calcular o endereço de memória do elemento a ser acedido: emite-se um `PUSHFP` que empilha o valor do FP, depois um `PUSHI offset` que empilha o offset base do array e, por fim, um `PADD` que desempilha ambos os valores e guarda a sua soma, obtendo o endereço base do array em memória. Vai-se então gerar o código para a expressão referente ao índice. Se o array tem um limite inferior, normaliza-se o índice subtraindo esse limite- `PUSHI sym.lower` para guardar o limite inferior na stack, e faz-se `SUB` para executar a subtração. Emite-se ainda um `PADD` para calcular o endereço do elemento específico, sendo que, de seguida, se emite um `READ` que vai ler uma linha e guardar uma string. Convertemos tal string para o seu tipo apropriado: se o elemento é `integer`, emite-se `ATOI`, se é `real`, emite-se `ATOF`. Finalmente, adiciona-se a instrução `STORE 0` que pega no valor convertido e guarda-o no topo da pilha.

Se não é nenhum dos casos, lançamos uma exceção informando isso.

Writeln

No caso de se ter um tipo `writeln`, chama-se esta função `gen_writeln`.

Recebendo um nó do tipo `('writeln', lista_exprs)`, vai extrair essa lista e iterar por cada expressão presente nela.

Se a expressão é uma `string`, então emite um `PUSHS valor`, que guarda a string no topo da stack, seguido de um `WRITES`, que escreve a string para o output.

Se a expressão é um `boolean`, converte-se esse valor para inteiro (0 ou 1), guardando-o no topo da stack através de `PUSHI`, e escrevendo-o com `WRITEI` no output.

Para qualquer outro caso, gera-se o código chamando `gen_expr`, que vai guardar na stack o seu valor. De seguida, vai se determinar o tipo da expressão, com `_expr_type`, e se esse tipo for `real`, usa-se `WRITEF` e caso contrário `WRITEI`, para escrever o resultado no output.

Por fim, faz-se um `WRITELN`, que escreve um `barra n`, marcando o final deste

comando.

For

No caso de se ter um tipo `for`, chama-se esta função `gen_for_to`.

Recebendo um nó do tipo `('for_to', va, expr_inicio, expr_fim, corpo)`, vai se extrair a variável de controlo, `var`, a expressão de ínicio e fim, `start, end`, e os comandos a executar dentro do for, `body`, procurando ainda na tabela de símbolos essa `var`.

Vai se gerar aqui dois labels únicos, um para o ínicio do ciclo, `start_label`, e outro para o fim, `end_label`.

De seguida, vamos gerar o código de inicialização, em que se gera o código VM para a expressão de ínicio, guardando no topo da stack o seu valor, emite-se um `STOREL offset`, que pega nesse valor e o guarda na `var`, e emitimos ainda a marcação do ínicio do ciclo.

Com isto, emite-se um `PUSHL offset` que guarda o valor atual de `var`, pelo que a seguir se gera o código VM da expressão de fim, guardando esse valor no topo da stack.

Com `INFEQ` compara os dois valores no topo, deixando lá 1 se for verdadeiro (<=) ou 0 se for falso. Caso a condição seja false, o `JZ end_label` faz com que salte para essa label, terminando assim o ciclo.

Se a condição for verdadeira, executa-se o corpo do ciclo, com `gen_stmt (body)`, em que fazemos `PUSHL offset` para guardar o valor atual, `PUSHI 1` e `ADD`, somando esses dois valores. Fazemos ainda `STOREL offset` e `JUMP start_label`, de modo a voltar ao início do ciclo.

No fim, marca-se `end_label` como ponto para sair deste for.

While

No caso de se ter um tipo `while`, chama-se esta função `gen_while`.

Recebendo um nó do tipo `('while', expr_condicao, corpo)`, vai se extrair a condição, `cond`, e os comandos a executar dentro do ciclo, `body`.

Vai se gerar novamente duas labels- `start_label, end_label`, marcando a de início.

De seguida, geramos o código VM para a expressão condicional, guardando no topo da pilha o valor (verdadeiro ou falso).

Emite-se ainda `JZ end_label` que salta para o fim do ciclo se a condição for falsa.

Se for verdadeira, executa o corpo e, no fim disso, emite-se um `JUMP start_label`, indicando que se volta ao início do ciclo.

No fim, marca-se o `end_label`, como ponto de saída do ciclo.

If

No caso de se ter um tipo `if`, chama-se esta função `gen_if`.

Recebendo um nó do tipo `('if', expr_condicao, cmd_then)` ou `('if', expr_condicao, cmd_then, cmd_else)`, vai extrair o rótulo do nó.

Caso esse seja `if`, extrai-se a condição, `cond`, e os comandos do `then_stmt`, definindo ainda que não há else.

Caso esse seja `ifelse`, extrai-se a condição, `cond`, e os comandos do `then_stmt` e do `else_stmt`.

Geram-se então dois labels. `else_label` para marcar o início do ramo else, e `end_label` para marcar o fim.

Gera-se o código da expressão, guardando no topo da stack o resultado (verdadeiro ou falso), emitindo ainda um `JZ else_label`, que salta para a label else se for falso.

Caso seja verdadeiro, executa-se o comando then. Após esse ramo then, emite-se `JUMP end_label`, saltando para o fim, e marcando a label `else_label`, sendo esse o ponto de entrada do ramo else (ou fim se este não existir).

Se existir tal else, vai se executá-lo, marcando no fim `end_label`, para sair deste if.

Atribuição array

No caso de se ter um tipo `atribuicao_array`, chama-se esta função `gen_atribuicao_array`.

Recebendo um nó do tipo `('atribuicao_array', nome_array, expr_indice, expr_valor)`, vai se extrair o nome dele, `var`, o índice a ser acedido, `index_expr`, e a expressão a ser atribuída, `expr`, procurando na tabela de símbolos o array.

Calcula-se o endereço de memória do elemento que estamos a aceder do array, fazendo `PUSHFP`, metendo no topo da stack o FP, `PUSHI offset`, empilhando o offset base do array, e `PADD` somando esses dois valores, obtendo assim o endereço base do array.

Gera-se então o código para a expressão que indica o índice a ser acedido do array, guardando esse valor no topo da pilha.

Se o array tem limite inferior, normaliza-se o índice: `PUSHI sym.lower`, empilhando o limite inferior, `SUB`, fazendo índice - lower.

Faz-se `PADD` que calcula o endereço final do elemento e, de seguida, gera-se o código para a expressão que se vai atribuir a esse elemento.

Determinamos ainda o tipo da expressão, e se o elemento tiver tipo `real` mas ela é do tipo `integer`, emite-se um `ITOF` para converter.

Por fim, emite-se um `STORE 0` de modo a pegar no valor obtido e o guardar no topo da pilha.

A função `gen_expr` é responsável por gerar código para a VM de expressões. Recebendo um nó da AST que representa uma expressão, ela emite o código referente da VM.

Começa por extrair `tag`m que identifica o tipo de expressão.

Literais Numéricos

Para números, representados por `('num', valor)`, primeiro extrai-se esse valor e vê-se o seu tipo.

Se for um float, fazemos um `PUSHF`, que mete o valor `real` no topo da stack.

Caso contário, fazemos um `PUSHI`, que mete o valor `inteiro` no topo da stack.

Id

Para identificadores, representados por `('id', nome)`, primeiro procura-se o símbolo correspondente na tabela de símbolos, obtendo o seu offset de memória.

Depois, emite-se um `PUSHL offset`, que carrega o valor da variável- armazenada no offset relativo ao FP- e mete-o no topo da stack

Operadores

Para operadores binários, representados por `('op', operador, expr_esq, expr_dir)`, extrai-se o operador, `op`, e ambas as expressões, `left, right`, obtendo ainda o tipo destas e gerando o código VM para ambas.

- **O operador é** `+` :

  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack, e depois usa-se `FADD` para os somar;
  - caso contrário, apenas se usa `ADD` para os somar.

- **O operador é `-` :**
  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack, e depois usa-se `FSUB` para os subtrair;
  - caso contrário, apenas se usa `SUB` para os subtrair.

- **O operador é `*` :**
  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack, e depois usa-se `FMULT` para os multiplicar;
  - caso contrário, apenas se usa `MUL` para os multiplicar.

- **O operador é `/` :** esta em Pascal é sempre `real`, pelo que se transfora os operandos são transformados em tal, e, no fim, emite-se `FDIV` para os dividir;

- **O operador é `div` :** emite-se `DIV` para os dividir;

- **O operador é `mod` :** emite-se `mod`;

- **O operador é `<=` :**
  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack, e depois usa-se `FINFEQ` para verificar esse operador;
  - caso contrário, apenas se usa `INFEQ` para os verificar tal operador.

- **O operador é `<` :**
  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack, e depois usa-se `FINF` para verificar esse operador;
  - caso contrário, apenas se usa `INF` para os verificar tal operador.

- **O operador é `>=` :**

- se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack, e depois usa-se `FSUPEQ` para verificar esse operador;
- caso contrário, apenas se usa `SUPEQ` para os verificar tal operador.

- **O operador é `>`** :
  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack, e depois usa-se `FSUP` para verificar esse operador;
  - caso contrário, apenas se usa `SUP` para os verificar tal operador.
- **O operador é `and`** : emite-se `AND`;
- **O operador é `or`** : emite-se `OR`;
- **O operador é `=`** :
  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack;
  - depois, usa-se a instrução `EQUAL`;
- **O operador é `<>`** :
  - se ambos os operandos são `real`, chama-se a função `_ensure_float_operands` para os converter para reais na stack;
  - depois, usa-se a instrução `EQUAL`, que guarda no topo 1 ou 0. Depois, compara-se o resultado com 0, através de outro `EQUAL` (a<>b é equivalente a not(a=b));

Neg

Para a negação, representada por `('neg', expr)`, começa-se por avaliar o tipo dessa `expr`.

Se é `real`, emite-se um `PUSHF 0.0` para meter no topo da pilha `0.0`, e depois gera-se o código dessa expressão, emitindo ao fim um `FSUB`, para calcular `0.0 - valor da expressão`, realizando a negação.

Se é `integer`, segue a mesma lógica, só que com as instruções `PUSHI 0`, geração do código da expressão, `SUB`.

Not

Para a negação lógica, representada por `('not', expr)`, começa-se por gerar o código para a expressão, guardando no topo 1 ou 0 e, ao fim disso, fazendo `PUSHI 0, EQUAL`, comparando o resultado com 0, inventando o seu valor booleano.

True

Para o valor booleano `true`, emite-se `PUSHI 1`.

False

Para o valor booleano `false`, emite-se `PUSHI 0`.

Acesso a array

Para o acesso a arrays, representado por `('array_acess', nome_array, expr_indice)`, começa-se por extrair o nome, `var`, e o indice, `index_expr`, procurando na tabela de símbolos esse array.

De seguida, calcula-se o endereço de memória do elemento: `PUSHFP` (empilha o FP), `PUSHI offset` (empilha o offset base do array), `PADD` (calcula e empilha o endereço base do array em memória).

Depois, gera-se o código para a expressão do índice, empilhando esse valor.

Se o array tem limite inferior, normaliza-se o índice: `PUSHI lower` (empilha o limite inferior), `SUB` (calcula índice - lower).

Emite-se `PADD`, que adiciona o índice ao endereço base, calculando o endereço final do elemento, e, por fim, `LOAD 0`, que carrega o valor armazenado no endereço que estava no topo da stack, e coloca no topo esse valor.

Fallback

Para terminar, se a expressão é de um tipo desconhecido, lança-se um erro, informando isso.

Avaliação Experimental

Neste capítulo, apresentamos os testes que realizamos para validar a funcionalidade e a correção todas as etapas do compilador que foi desenvolvido.

Desenvolvemos exemplos para as 4 fases de construção do mesmo: análise léxica (lexer), análise sintática (parser), análise semântica e geração de código para a máquina virtual, tendo sido cada uma destas fases testada de forma isolada e integrada.

Os resultados apresentados neste capítulo validam a implementação das várias componentes do compilador e demonstram a sua robustez na deteção e reportagem de erros.

4.1 Testes do Lexer

A análise léxica é a primeira etapa do compilador, responsável por converter o código fonte numa sequência de tokens.

4.1.1 Exemplo 1

Este corresponde a um exemplo mais simples, em que apenas se cria variáveis `x,y` e se atribui o valor `10` à `x`, e, em `y` guardamos o resultado de `x ' 20`, imprimindo esse valor.

Código de entrada:

```
1 program exemplo;
2 var x, y: integer;
3 begin
4     x := 10;
5     y := x + 20;
6     writeln(y);
```

```
end.
```

Elementos testados:

- Palavras chave: `program, var, begin, writeln, end`;
- Identificadores: `exemplo, x, y`;
- Tipos: `atribuicao (:=), soma (+)`
- Delimitadores: `';', ',' , parênteses, '.'`

Tokens esperados: espera-se que o lexer produza uma sequência de tokens começando com `PROGRAM`, seguido de `ID`, `PONTEVIRG, NUM`, e sucessivamente os tokens correspondentes a cada elemento léxico do código, terminando com `END PONTO`.

Resultado:

```
LexToken(PROGRAM,'program',2,1)
LexToken(ID,'exemplo',2,9)
LexToken(PONTOEVIRG,';',2,16)
LexToken(VAR,'var',3,18)
LexToken(ID,'x',3,22)
LexToken(VIRG,',',3,23)
LexToken(ID,'y',3,25)
LexToken(DOISPONTOS,':',3,26)
LexToken(INTEGER,'integer',3,28)
LexToken(PONTOEVIRG,';',3,35)
LexToken(BEGIN,'begin',4,37)
LexToken(ID,'x',5,47)
LexToken(ATRIBUICAO,':=',5,49)
LexToken(NUM,10,5,52)
LexToken(PONTOEVIRG,';',5,54)
LexToken(ID,'y',6,60)
LexToken(ATRIBUICAO,':=',6,62)
LexToken(ID,'x',6,65)
LexToken(SOMA,'+',6,67)
LexToken(NUM,20,6,69)
LexToken(PONTOEVIRG,';',6,71)
LexToken(WRITELN,'writeln',7,77)
LexToken(LPAREN,'(',7,84)
```

```
LexToken(ID,'y',7,85)
LexToken(RPAREN,')',7,86)
LexToken(PONTOEVIRG,';',7,87)
LexToken(END,'end',8,89)
LexToken(PONTO,'.',8,92)
```

Como vemos através desse output obtido, todos os tokens são corretamente identificados e classificados, validando o lexer criado, para este exemplo.

4.1.2 Exemplo 2

Este corresponde ao segundo exemplo apresentado no enunciado do projeto.

Código de entrada:

```
program Fatorial;
var
    n, i, fat: integer;
begin
    writeln('Introduza um numero inteiro positivo:');
    readln(n);
    fat := 1;
    for i := 1 to n do
        fat := fat * i;
    writeln('Fatorial de ', n, ': ', fat);
end.
```

Elementos testados:

- Ciclo `for`, com palavras chave `for, to, do`;
- Strings literais com espaços e pontuação;
- Múltiplos identificadores em lista;
- Operador de multiplicação e comando `readln`;
- Múltiplos argumentos no comando `writeln`;

Tokens esperados: espera-se que o lexer identifique `FOR, TO DO, STRING, READL, MULT, WRITELN`.

Resultado:

```
LexToken(PROGRAM,'program',2,1)
LexToken(ID,'Fatorial',2,9)
LexToken(PONTOEVIRG,';',2,17)
LexToken(VAR,'var',3,19)
LexToken(ID,'n',4,23)
LexToken(VIRG,',',4,24)
LexToken(ID,'i',4,26)
LexToken(VIRG,',',4,27)
LexToken(ID,'fat',4,29)
LexToken(DOISPONTOS,':',4,32)
LexToken(INTEGER,'integer',4,34)
LexToken(PONTOEVIRG,';',4,41)
LexToken(BEGIN,'begin',5,43)
LexToken(WRITELN,'writeln',6,49)
LexToken(LPAREN,'(',6,56)
LexToken(STRING,'Introduza␣um␣numero␣inteiro␣positivo:',6,57)
LexToken(RPAREN,')',6,96)
LexToken(PONTOEVIRG,';',6,97)
LexToken(READLN,'readln',7,99)
LexToken(LPAREN,'(',7,105)
LexToken(ID,'n',7,106)
LexToken(RPAREN,')',7,107)
LexToken(PONTOEVIRG,';',7,108)
LexToken(ID,'fat',8,110)
LexToken(ATRIBUICAO,':=',8,114)
LexToken(NUM,1,8,117)
LexToken(PONTOEVIRG,';',8,118)
LexToken(FOR,'for',9,120)
LexToken(ID,'i',9,124)
LexToken(ATRIBUICAO,':=',9,126)
LexToken(NUM,1,9,129)
LexToken(TO,'to',9,131)
LexToken(ID,'n',9,134)
LexToken(DO,'do',9,136)
LexToken(ID,'fat',10,139)
LexToken(ATRIBUICAO,':=',10,143)
LexToken(ID,'fat',10,146)
LexToken(MULT,'*',10,150)
LexToken(ID,'i',10,152)
```

```
LexToken(PONTOEVIRG,';',10,153)
LexToken(WRITELN,'writeln',11,155)
LexToken(LPAREN,'(',11,162)
LexToken(STRING,'Fatorial de ',11,163)
LexToken(VIRG,',',11,177)
LexToken(ID,'n',11,179)
LexToken(VIRG,',',11,180)
LexToken(STRING,': ',11,182)
LexToken(VIRG,',',11,186)
LexToken(ID,'fat',11,188)
LexToken(RPAREN,')',11,191)
LexToken(PONTOEVIRG,';',11,192)
LexToken(END,'end',12,194)
LexToken(PONTO,'.',12,197)
```

Como vemos através desse output obtido, todos os tokens são corretamente identificados e classificados, validando o lexer criado, para este exemplo.

4.1.3 Exemplo 3

Este corresponde ao terceiro exemplo apresentado no enunciado do projeto.

Código de entrada:

```
program NumeroPrimo;
    var
        num, i: integer;
        primo: boolean;
    begin
        writeln('Introduza um numero inteiro positivo:');
        readln(num);
        primo := true;
        i := 2;
        while (i <= (num div 2)) and primo do
        begin
            if (num mod i) = 0 then
            primo := false;
            i := i + 1;
        end;
        if primo then
        writeln(num, ' e um numero primo')
```

```
18          else
19          writeln(num, ' nao e um numero primo')
20      end.
```

Elementos testados:

- Tipo `boolean` e os valores `true, false`;
- Ciclo `while`;
- Operadores relacionais, `<=`, aritméticos, `div, mod`, e lógicos, `and`;
- Condicionais `if-then, if-then-else`;

Tokens esperados: espera-se que o lexer identifique `BOOLEAN, TRUE, FALSE, WHILE, DIV, MOD, AND, IF, THEN, ELSE, MENORIGUAL`.

Resultado:

```
1  LexToken(PROGRAM,'program',2,1)
2  LexToken(ID,'NumeroPrimo',2,9)
3  LexToken(PONTOEVIRG,';',2,20)
4  LexToken(VAR,'var',3,26)
5  LexToken(ID,'num',4,38)
6  LexToken(VIRG,',',4,41)
7  LexToken(ID,'i',4,43)
8  LexToken(DOISPONTOS,':',4,44)
9  LexToken(INTEGER,'integer',4,46)
10 LexToken(PONTOEVIRG,';',4,53)
11 LexToken(ID,'primo',5,63)
12 LexToken(DOISPONTOS,':',5,68)
13 LexToken(BOOLEAN,'boolean',5,70)
14 LexToken(PONTOEVIRG,';',5,77)
15 LexToken(BEGIN,'begin',6,83)
16 LexToken(WRITELN,'writeln',7,97)
17 LexToken(LPAREN,'(',7,104)
18 LexToken(STRING,'Introduza um numero inteiro positivo:',7,105)
19 LexToken(RPAREN,')',7,144)
20 LexToken(PONTOEVIRG,';',7,145)
21 LexToken(READLN,'readln',8,155)
22 LexToken(LPAREN,'(',8,161)
```

```
LexToken(ID,'num',8,162)
LexToken(RPAREN,')',8,165)
LexToken(PONTOEVIRG,';',8,166)
LexToken(ID,'primo',9,176)
LexToken(ATRIBUICAO,':=',9,182)
LexToken(TRUE,'true',9,185)
LexToken(PONTOEVIRG,';',9,189)
LexToken(ID,'i',10,199)
LexToken(ATRIBUICAO,':=',10,201)
LexToken(NUM,2,10,204)
LexToken(PONTOEVIRG,';',10,205)
LexToken(WHILE,'while',11,215)
LexToken(LPAREN,'(',11,221)
LexToken(ID,'i',11,222)
LexToken(MENORIGUAL,'<=',11,224)
LexToken(LPAREN,'(',11,227)
LexToken(ID,'num',11,228)
LexToken(DIV,'div',11,232)
LexToken(NUM,2,11,236)
LexToken(RPAREN,')',11,237)
LexToken(RPAREN,')',11,238)
LexToken(AND,'and',11,240)
LexToken(ID,'primo',11,244)
LexToken(DO,'do',11,250)
LexToken(BEGIN,'begin',12,261)
LexToken(IF,'if',13,279)
LexToken(LPAREN,'(',13,282)
LexToken(ID,'num',13,283)
LexToken(MOD,'mod',13,287)
LexToken(ID,'i',13,291)
LexToken(RPAREN,')',13,292)
LexToken(IGUAL,'=',13,294)
LexToken(NUM,0,13,296)
LexToken(THEN,'then',13,298)
LexToken(ID,'primo',14,315)
LexToken(ATRIBUICAO,':=',14,321)
LexToken(FALSE,'false',14,324)
LexToken(PONTOEVIRG,';',14,329)
LexToken(ID,'i',15,343)
LexToken(ATRIBUICAO,':=',15,345)
```

```
LexToken(ID,'i',15,348)
LexToken(SOMA,'+',15,350)
LexToken(NUM,1,15,352)
LexToken(PONTOEVIRG,';',15,353)
LexToken(END,'end',16,363)
LexToken(PONTOEVIRG,';',16,366)
LexToken(IF,'if',17,376)
LexToken(ID,'primo',17,379)
LexToken(THEN,'then',17,385)
LexToken(WRITELN,'writeln',18,398)
LexToken(LPAREN,'(',18,405)
LexToken(ID,'num',18,406)
LexToken(VIRG,',',18,409)
LexToken(STRING,'␣e␣um␣numero␣primo',18,411)
LexToken(RPAREN,')',18,431)
LexToken(ELSE,'else',19,441)
LexToken(WRITELN,'writeln',20,454)
LexToken(LPAREN,'(',20,461)
LexToken(ID,'num',20,462)
LexToken(VIRG,',',20,465)
LexToken(STRING,'␣nao␣e␣um␣nemero␣primo',20,467)
LexToken(RPAREN,')',20,491)
LexToken(END,'end',21,497)
LexToken(PONTO,'.',21,500)
```

Como vemos através desse output obtido, todos os tokens são corretamente identificados e classificados, validando o lexer criado, para este exemplo.

4.1.4 Exemplo 4

Este corresponde ao quarto exemplo apresentado no enunciado do projeto.

Código de entrada:

```
program SomaArray;
    var
        numeros: array[1..5] of integer;
        i, soma: integer;
    begin
        soma := 0;
        writeln('Introduza␣5␣numeros␣inteiros:');
```

```
8           for i := 1 to 5 do
9               begin
10                  readln(numeros[i]);
11                  soma := soma + numeros[i];
12              end;
13          writeln('A soma dos numeros e: ', soma);
14      end.
```

Elementos testados:

- Palavras chave: `array`, `of`;
- Intervalo de array `[1..5]`;
- Acesso a elementos do array com `[indice]`;
- Atribuições envolvendo elementos de arrays;
- Operações aritméticas com elementos de arrays;

Tokens esperados: espera-se que o lexer identifique `ARRAY`, `OF`, `LBRACKET`, `INTERVALO`, `RBRACKED` e, para além disso, os índices devem ser corretamente tokenizados.

Resultado:

```
1  LexToken(PROGRAM,'program',2,1)
2  LexToken(ID,'SomaArray',2,9)
3  LexToken(PONTOEVIRG,';',2,18)
4  LexToken(VAR,'var',3,20)
5  LexToken(ID,'numeros',4,24)
6  LexToken(DOISPONTOS,':',4,31)
7  LexToken(ARRAY,'array',4,33)
8  LexToken(LBRACKET,'[',4,38)
9  LexToken(NUM,1,4,39)
10 LexToken(INTERVALO,'..',4,40)
11 LexToken(NUM,5,4,42)
12 LexToken(RBRACKET,']',4,43)
13 LexToken(OF,'of',4,45)
14 LexToken(INTEGER,'integer',4,48)
15 LexToken(PONTOEVIRG,';',4,55)
16 LexToken(ID,'i',5,57)
```

```
LexToken(VIRG,',',5,58)
LexToken(ID,'soma',5,60)
LexToken(DOISPONTOS,':',5,64)
LexToken(INTEGER,'integer',5,66)
LexToken(PONTOEVIRG,';',5,73)
LexToken(BEGIN,'begin',6,75)
LexToken(ID,'soma',7,81)
LexToken(ATRIBUICAO,':=',7,86)
LexToken(NUM,0,7,89)
LexToken(PONTOEVIRG,';',7,90)
LexToken(WRITELN,'writeln',8,92)
LexToken(LPAREN,'(',8,99)
LexToken(STRING,'Introduza␣5␣numeros␣inteiros:',8,100)
LexToken(RPAREN,')',8,131)
LexToken(PONTOEVIRG,';',8,132)
LexToken(FOR,'for',9,134)
LexToken(ID,'i',9,138)
LexToken(ATRIBUICAO,':=',9,140)
LexToken(NUM,1,9,143)
LexToken(TO,'to',9,145)
LexToken(NUM,5,9,148)
LexToken(DO,'do',9,150)
LexToken(BEGIN,'begin',10,153)
LexToken(READLN,'readln',11,159)
LexToken(LPAREN,'(',11,165)
LexToken(ID,'numeros',11,166)
LexToken(LBRACKET,'[',11,173)
LexToken(ID,'i',11,174)
LexToken(RBRACKET,']',11,175)
LexToken(RPAREN,')',11,176)
LexToken(PONTOEVIRG,';',11,177)
LexToken(ID,'soma',12,179)
LexToken(ATRIBUICAO,':=',12,184)
LexToken(ID,'soma',12,187)
LexToken(SOMA,'+',12,192)
LexToken(ID,'numeros',12,194)
LexToken(LBRACKET,'[',12,201)
LexToken(ID,'i',12,202)
LexToken(RBRACKET,']',12,203)
LexToken(PONTOEVIRG,';',12,204)
```

```
LexToken(END,'end',13,206)
LexToken(PONTOEVIRG,';',13,209)
LexToken(WRITELN,'writeln',14,211)
LexToken(LPAREN,'(',14,218)
LexToken(STRING,'A␣soma␣dos␣numeros␣e:␣',14,219)
LexToken(VIRG,',',14,243)
LexToken(ID,'soma',14,245)
LexToken(RPAREN,')',14,249)
LexToken(PONTOEVIRG,';',14,250)
LexToken(END,'end',15,252)
LexToken(PONTO,'.',15,255)
```

Como vemos através desse output obtido, todos os tokens são corretamente identificados e classificados, validando o lexer criado, para este exemplo.

4.1.5 Exemplo 5

Este corresponde ao mesmo exemplo do 1, mas com comentários pelo meio.

Código de entrada:

```
(*
   comentario
*)

program exemplo;
    var x, y: integer;

    //isto aqui tambem e um comentario
    begin
        x := 10;
        { isto e um
        comentario
        }
        y := x + 20;
        writeln(y);
    (*
    comentario
    *)
    end.

```

```
21      (*
22      comentario
23      *)
```

Elementos testados:

- Todos os tipos de comentários: em bloco, ou em linha.

Tokens esperados: espera-se que o lexer ignore completamente todos os comentários, produzindo a mesma sequência que no primeiro exemplo, a menos do número da linha.

Resultado:

```
1  LexToken(PROGRAM,'program',7,23)
2  LexToken(ID,'exemplo',7,31)
3  LexToken(PONTOEVIRG,';',7,38)
4  LexToken(VAR,'var',8,40)
5  LexToken(ID,'x',8,44)
6  LexToken(VIRG,',',8,45)
7  LexToken(ID,'y',8,47)
8  LexToken(DOISPONTOS,':',8,48)
9  LexToken(INTEGER,'integer',8,50)
10 LexToken(PONTOEVIRG,';',8,57)
11 LexToken(BEGIN,'begin',11,95)
12 LexToken(ID,'x',12,105)
13 LexToken(ATRIBUICAO,':=',12,107)
14 LexToken(NUM,10,12,110)
15 LexToken(PONTOEVIRG,';',12,112)
16 LexToken(ID,'y',16,156)
17 LexToken(ATRIBUICAO,':=',16,158)
18 LexToken(ID,'x',16,161)
19 LexToken(SOMA,'+',16,163)
20 LexToken(NUM,20,16,165)
21 LexToken(PONTOEVIRG,';',16,167)
22 LexToken(WRITELN,'writeln',17,173)
23 LexToken(LPAREN,'(',17,180)
24 LexToken(ID,'y',17,181)
25 LexToken(RPAREN,')',17,182)
26 LexToken(PONTOEVIRG,';',17,183)
```

```
LexToken(END,'end',21,205)
LexToken(PONTO,'.',21,208)
```

Como vemos através desse output obtido, todos os tokens são corretamente identificados e classificados, validando o lexer criado, para este exemplo.

4.2 Testes do Parser

A análise semântica é a segunda etapa do compilador, sendo responsável por converter a sequência de tokens obtida na fase anterior numa AST, representando a estrutra gramatical do programa.

Iremos usar os mesmos códigos da fase anterior.

4.2.1 Exemplo 1

Código de entrada:

```
program exemplo;
var x, y: integer;
begin
    x := 10;
    y := x + 20;
    writeln(y);
end.
```

Resultado:

```
('programa', 'exemplo',
    [('decl', ['x', 'y'], 'integer')],
    ('bloco_principal',
    [('atribuicao', 'x', ('num', 10)), ('atribuicao', 'y', ('
       op', '+', ('id', 'x'), ('num', 20))),
    ('writeln', [('id', 'y')])]))
```

Como vemos através desse output obtido, a AST é corretamente construída.

O nó da raiz é de facto `'programa'`, contendo o nome deste.

As declarações de variáveus são agrupadas numa lista de tuplos `'decl'`.

O bloco principal é representado por `'bloco_principal'`, contendo a lista de comandos, em que temos atribuiçõres representadas por tuplos `'atribuicao'`, ope-

rações binárias em tuplos `'op'`, identificadores em tuplos `'id'` e literais numéricos em `'num'`, entre outros aspectos.

4.2.2 Exemplo 2

Código de entrada:

```
program Fatorial;
var
    n, i, fat: integer;
begin
    writeln('Introduza um numero inteiro positivo:');
    readln(n);
    fat := 1;
    for i := 1 to n do
        fat := fat * i;
    writeln('Fatorial de ', n, ': ', fat);
end.
```

Resultado:

```
('programa', 'Fatorial',
[('decl', ['n', 'i', 'fat'], 'integer')],
('bloco_principal',
    [('writeln',
    [('string', 'Introduza um numero inteiro positivo:')]),
    ('readln', [('id', 'n')]),
    ('atribuicao', 'fat', ('num', 1)),
    ('for_to', 'i', ('num', 1), ('id', 'n'),
    ('atribuicao', 'fat', ('op', '*', ('id', 'fat'), ('id', 'i
        '))))),
    ('writeln', [('string', 'Fatorial de '), ('id', 'n'), ('
        string', ': '), ('id', 'fat')])]))
```

Como vemos através desse output obtido, a AST é corretamente construída.

O nó da raiz é de facto `'programa'`, contendo o nome deste.

As declarações de variáveus são agrupadas numa lista de tuplos `'decl'`.

O bloco principal é representado por `'bloco_principal'`, contendo a lista de comandos, em que temos atribuições representadas por tuplos `'atribuicao'`, operações binárias em tuplos `'op'`, identificadores em tuplos `'id'` e literais numéricos em `'num'`, entre outros aspectos.

4.2.3 Exemplo 3

Código de entrada:

```
program NumeroPrimo;
    var
        num, i: integer;
        primo: boolean;
    begin
        writeln('Introduza um numero inteiro positivo:');
        readln(num);
        primo := true;
        i := 2;
        while (i <= (num div 2)) and primo do
        begin
            if (num mod i) = 0 then
            primo := false;
            i := i + 1;
        end;
        if primo then
        writeln(num, ' e um numero primo')
        else
        writeln(num, ' nao e um numero primo')
    end.
```

Resultado:

```
('programa', 'NumeroPrimo',
[('decl', ['num', 'i'], 'integer'),
('decl', ['primo'], 'boolean')],
('bloco_principal',
    [('writeln', [('string', 'Introduza um numero inteiro
       positivo:')]),
    ('readln', [('id', 'num')]),
    ('atribuicao', 'primo', ('true',)),
    ('atribuicao', 'i', ('num', 2)),
    ('while', ('op', 'and', ('op', '<=', ('id', 'i'), ('op', '
       div', ('id', 'num'), ('num', 2))), ('id', 'primo')),
    ('bloco',
        [('if', ('op', '=', ('op', 'mod', ('id', 'num'), ('id'
           , 'i')), ('num', 0)),
```

```
12          ('atribuicao', 'primo', ('false',))),
13          ('atribuicao', 'i', ('op', '+', ('id', 'i'), ('num',
                1)))])),
14      ('ifelse', ('id', 'primo'),
15          ('writeln', [('id', 'num'), ('string', '␣e␣um␣numero␣
                primo')]),
16          ('writeln', [('id', 'num'), ('string', '␣nao␣e␣um␣
                numero␣primo')]))]))
```

Como vemos através desse output obtido, a AST é corretamente construída.

O nó da raiz é de facto `'programa'`, contendo o nome deste.

As declarações de variáveus são agrupadas numa lista de tuplos `'decl'`.

O bloco principal é representado por `'bloco_principal'`, contendo a lista de comandos, em que temos atribuições representadas por tuplos `'atribuicao'`, operações binárias em tuplos `'op'`, identificadores em tuplos `'id'` e literais numéricos em `'num'`, entre outros aspectos.

4.2.4 Exemplo 4

Código de entrada:

```
1  program SomaArray;
2      var
3          numeros: array[1..5] of integer;
4          i, soma: integer;
5      begin
6          soma := 0;
7          writeln('Introduza␣5␣numeros␣inteiros:');
8          for i := 1 to 5 do
9              begin
10                 readln(numeros[i]);
11                 soma := soma + numeros[i];
12             end;
13         writeln('A␣soma␣dos␣numeros␣e:␣', soma);
14     end.
```

Resultado:

```
1  ('programa', 'SomaArray',
2  [('decl', ['numeros'], ('array', (1, 5), 'integer')),
```

```
3  ('decl', ['i', 'soma'], 'integer')],
4  ('bloco_principal',
5      [('atribuicao', 'soma', ('num', 0)),
6      ('writeln', [('string', 'Introduza 5 numeros inteiros:')])
           ,
7      ('for_to', 'i', ('num', 1), ('num', 5),
8      ('bloco',
9      [('readln', [('array_access', 'numeros', ('id', 'i'))]),
10     ('atribuicao', 'soma', ('op', '+', ('id', 'soma'), ('
           array_access', 'numeros', ('id', 'i'))))])),
11     ('writeln', [('string', 'A soma dos numeros e: '), ('id',
           'soma')])]))
```

Como vemos através desse output obtido, a AST é corretamente construída.

O nó da raiz é de facto `'programa'`, contendo o nome deste.

As declarações de variáveus são agrupadas numa lista de tuplos `'decl'`.

O bloco principal é representado por `'bloco_principal'`, contendo a lista de comandos, em que temos atribuições representadas por tuplos `'atribuicao'`, operações binárias em tuplos `'op'`, identificadores em tuplos `'id'` e literais numéricos em `'num'`, entre outros aspectos.

4.2.5 Exemplo 5

Código de entrada:

```
1  (*
2     comentario
3  *)
4
5  program exemplo;
6      var x, y: integer;
7
8      //isto aqui tambem e um comentario
9      begin
10         x := 10;
11         { isto e um
12         comentario
13         }
14         y := x + 20;
15         writeln(y);
```

```
16      (*
17      comentario
18      *)
19      end.
20
21      (*
22      comentario
23      *)
```

Resultado:

```
('programa', 'exemplo',
[('decl', ['x', 'y'], 'integer')],
('bloco_principal',
   [('atribuicao', 'x', ('num', 10)),
   ('atribuicao', 'y', ('op', '+', ('id', 'x'), ('num', 20))),
   ('writeln', [('id', 'y')])]))
```

Como vemos através desse output obtido, a AST é corretamente construída.

O nó da raiz é de facto `'programa'`, contendo o nome deste.

As declarações de variáveus são agrupadas numa lista de tuplos `'decl'`.

O bloco principal é representado por `'bloco_principal'`, contendo a lista de comandos, em que temos atribuições representadas por tuplos `'atribuicao'`, operações binárias em tuplos `'op'`, identificadores em tuplos `'id'` e literais numéricos em `'num'`, entre outros aspectos.

4.3 Testes Semânticos

A análise semântica é a terceira etapa do compilador, sendo responsável por validar a correção semântica do programa após a análise sintática ter confirmado a estrutura gramatical.

Iremos dividir esta em duas fases, em que primeiro, para os mesmos exemplos anteriores, analisamos exemplos cuja semântica está correta. Depois criamos novos exemplos, com semântica errada, de modo a mostrar que, de facto, esta fase reconhece bem vários tipos de erros semânticos.

4.3.1 Semântica Correta

Exemplo 1

Código de entrada:

```
program exemplo;
var x, y: integer;
begin
    x := 10;
    y := x + 20;
    writeln(y);
end.
```

Resultado:

```
    Semantica correta     nenhum erro encontrado.
```

Como vemos através desse output obtido, nenhum erro semântico foi encontrado, pelo que valida que o analisador aceita programas semanticamente corretos.

Exemplo 2

Código de entrada:

```
program Fatorial;
var
    n, i, fat: integer;
begin
    writeln('Introduza um numero inteiro positivo:');
    readln(n);
    fat := 1;
    for i := 1 to n do
        fat := fat * i;
    writeln('Fatorial de ', n, ': ', fat);
end.
```

Resultado:

```
    Semantica correta     nenhum erro encontrado.
```

Como vemos através desse output obtido, nenhum erro semântico foi encontrado, pelo que valida que o analisador aceita programas semanticamente corretos.

Exemplo 3

Código de entrada:

```
program NumeroPrimo;
    var
        num, i: integer;
        primo: boolean;
    begin
        writeln('Introduza um numero inteiro positivo:');
        readln(num);
        primo := true;
        i := 2;
        while (i <= (num div 2)) and primo do
        begin
            if (num mod i) = 0 then
            primo := false;
            i := i + 1;
        end;
        if primo then
        writeln(num, ' e um numero primo')
        else
        writeln(num, ' nao e um numero primo')
    end.
```

Resultado:

```
    Semantica correta     nenhum erro encontrado.
```

Como vemos através desse output obtido, nenhum erro semântico foi encontrado, pelo que valida que o analisador aceita programas semanticamente corretos.

Exemplo 4

Código de entrada:

```
program SomaArray;
    var
        numeros: array[1..5] of integer;
        i, soma: integer;
    begin
```

```
        soma := 0;
        writeln('Introduza 5 numeros inteiros:');
        for i := 1 to 5 do
            begin
                readln(numeros[i]);
                soma := soma + numeros[i];
            end;
        writeln('A soma dos numeros e: ', soma);
    end.
```

Resultado:

```
    Semantica correta     nenhum erro encontrado.
```

Como vemos através desse output obtido, nenhum erro semântico foi encontrado, pelo que valida que o analisador aceita programas semanticamente corretos.

Exemplo 5

Código de entrada:

```
(*
   comentario
*)

program exemplo;
    var x, y: integer;

    //isto aqui tambem e um comentario
    begin
        x := 10;
        { isto e um
        comentario
        }
        y := x + 20;
        writeln(y);
    (*
    comentario
    *)
    end.

```

```
21      (*
22      comentario
23      *)
```

Resultado:

```
1       Semantica correta     nenhum erro encontrado.
```

Como vemos através desse output obtido, nenhum erro semântico foi encontrado, pelo que valida que o analisador aceita programas semanticamente corretos.

4.3.2 Semântica Errada

Aqui apresentamos alguns exemplos de códigos em Pascal cuja semântica não se encontra correta, pelo que o analisador deve retornar erros semânticos.

Exemplo 1

Código de entrada:

```
1  program Ex1;
2  var x: integer;
3  begin
4      x := 10;
5      y := 5;   { ERRO: y n o foi declarada }
6  end.
```

Resultado:

```
1       Erros sem nticos detetados:
2
3  Erros sem nticos encontrados:
4  - Atribui  o a vari vel n o declarada 'y'.
```

Aqui nós devemos ter um erro semântico, uma vez que usamos uma variável que não foi declarada.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 2

```
program Ex2;
var
    x: integer;
    x: real;   { ERRO: redeclara  o }
begin
    x := 3;
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
- Vari vel 'x' j  declarada.
```

Aqui nós devemos ter um erro semântico, uma vez que declaramos duas vezes a mesma variável.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 3

```
program Ex3;
var
    x: integer;
begin
    x := true;   { ERRO: tipos incompat veis }
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
- Incompatibilidade de tipos na atribui  o a 'x': vari vel
      'integer' mas express o    'boolean'.
```

Aqui nós devemos ter um erro semântico, uma vez que atribuimos uma expressão de um tipo, a uma variável que é de outro tipo.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 4

```
program Ex4;
var
    s: string;
    x: integer;
begin
    s := 'ola';
    x := 5 + s;   { ERRO: n o pode somar string com integer }
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
- Operador '+' exige operandos num ricos (integer/real).
   Obteve: 'integer' e 'string'.
```

Aqui nós devemos ter um erro semântico, uma vez que se tenta fazer uma operação aritmética entre tipos que são incompatíveis.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 5

```
program Ex5;
var
    x, y: integer;
begin
    x := 3;
    y := 4;
    if x and y then   { ERRO: and s   aceita boolean }
        writeln('OK');
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
```

```
- Operador l gico 'and' exige operandos boolean (obtido '
   integer' e 'integer').
- Condi   o do IF deve ser boolean, obteve 'None'.
```

Aqui nós devemos ter um erro semântico, uma vez que se tenta usar um operador lógico com operandos que não são booleanos.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 6

```
program Ex6;
var
    x: integer;
begin
    x := 10;
    if x then   { ERRO: IF exige boolean }
        writeln('ok');
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
- Condi   o do IF deve ser boolean, obteve 'integer'.
```

Aqui nós devemos ter um erro semântico, uma vez que a condição do if não é um booleano.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 7

```
program Ex7;
var
    n: integer;
begin
    n := 0;
    while n do      { ERRO }
        n := n + 1;
```

```
8 end.
```

Resultado:

```
1     Erros sem nticos detetados:
2
3 Erros sem nticos encontrados:
4 - Condi   o do WHILE deve ser boolean, obteve 'integer'.
```

Aqui nós devemos ter um erro semântico, uma vez que a condição do while não é um integer.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 8

```
1 program Ex8;
2 var
3     x: integer;
4 begin
5     for i := 1 to 10 do   { ERRO: i n o existe }
6         x := x + 1;
7 end.
```

Resultado:

```
1     Erros sem nticos detetados:
2
3 Erros sem nticos encontrados:
4 - Vari vel de controlo do FOR 'i' n o declarada.
```

Aqui nós devemos ter um erro semântico, uma vez que o for usa uma variável que não foi declarada.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 9

```
1 program Ex9;
2 var
3     b: boolean;
```

```
4 begin
5     for b := true to false do   { ERRO: controle do for deve
          ser integer }
6         writeln('oi');
7 end.
```

Resultado:

```
1     Erros sem nticos detetados:
2
3 Erros sem nticos encontrados:
4 - Vari vel de controlo do FOR 'b' deve ser integer.
5 - Express es de in cio/fim do FOR devem ser integer (obtido:
      boolean, boolean).
```

Aqui nós devemos ter um erro semântico, uma vez que o for usa uma variável que não é inteira, e as expressões de ínicio e fim deste também não são inteiras.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 10

```
1 program Ex10;
2 var
3     i: integer;
4 begin
5     for i := 1.5 to 10 do   { ERRO: limites devem ser integer
          }
6         writeln(i);
7 end.
```

Resultado:

```
1     Erros sem nticos detetados:
2
3 Erros sem nticos encontrados:
4 - Express es de in cio/fim do FOR devem ser integer (obtido:
      real, integer).
```

Aqui nós devemos ter um erro semântico, uma vez que o for usa expressões de ínicio e fim deste também não são inteiras.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 11

```
1 program Ex11;
2 var
3     x: integer;
4 begin
5     x := numeros[3];    { ERRO: numeros n o existe }
6 end.
```

Resultado:

```
1     Erros sem nticos detetados:
2
3 Erros sem nticos encontrados:
4 - Acesso a array n o declarado 'numeros'.
```

Aqui nós devemos ter um erro semântico, uma vez que se tenta aceder a um array não declarado.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 12

```
1 program Ex12;
2 var
3     x: integer;
4 begin
5     x := x[1];  { ERRO: x n o     array }
6 end.
```

Resultado:

```
1     Erros sem nticos detetados:
2
3 Erros sem nticos encontrados:
4 - Identificador 'x' n o     um array.
```

Aqui nós devemos ter um erro semântico, uma vez que tenta aceder a algo que não é um array.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 13

```
program Ex13;
var
    arr: array[1..5] of integer;
begin
    arr['a'] := 3;   { ERRO:  ndice  deve ser integer }
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
-  ndice  do array 'arr' deve ser integer, obteve 'string'.
```

Aqui nós devemos ter um erro semântico, uma vez que o índice do array a que quer se aceder não é um inteiro.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 14

```
program Ex14;
var
    arr: array[2..4] of integer;
begin
    arr[1] := 5;   { ERRO: 1 est  fora de 2..4 }
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
-  ndice  1 fora dos limites do array 'arr' [2..4].
```

Aqui nós devemos ter um erro semântico, uma vez que o índice do array a que quer se aceder está fora dos limites desse array.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 15

```
program Ex15;
var
    arr: array[1..3] of integer;
begin
    arr[2] := 'ola';   { ERRO: array espera integer }
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
- Tipo incompat vel na atribui  o ao array 'arr': elemento
      'integer', express o    'string'.
```

Aqui nós devemos ter um erro semântico, uma vez que se atribui a um elemento do array uma expressão que não equivale ao seu tipo.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 16

```
program Ex16;
var
    x: inteiro;   { ERRO: tipo n o existe }
begin
    x := 5;
end.
```

Resultado:

```
Erro de sintaxe na linha 4: token 'ID' com valor 'inteiro'
    Erros sem nticos detetados:

```

```
Erros sem nticos encontrados:
- AST inv lida: n   raiz n o    'programa'.
```

Aqui nós devemos ter um erro semântico, uma vez que se atribui a uma variável um tipo que não se conhece.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 17

```
program Ex17;
var
    arr: array[1..3] of integer;
begin
    writeln(arr);   { ERRO: n o podes imprimir arrays }
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
- writeln: tipo n o imprim vel 'array' (expr ('id', 'arr')).
```

Aqui nós devemos ter um erro semântico, uma vez que se tenta imprimir algo não imprimível com writeln.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

Exemplo 18

```
program Ex18;
var
    x: real;
    arr: array[1..3] of boolean;
begin
    y := 10;                { y n o declarado }
    x := arr;               { tipos incompat veis }
    arr[1.5] := true;       {  ndice  n o -inteiro }
    if x then writeln('x'); { condi  o n o -boolean }
end.
```

Resultado:

```
    Erros sem nticos detetados:

Erros sem nticos encontrados:
- Atribui  o a vari vel n o declarada 'y'.
- Incompatibilidade de tipos na atribui  o a 'x': vari vel
      'real' mas express o    'array'.
-  ndice  do array 'arr' deve ser integer, obteve 'real'.
- Condi  o do IF deve ser boolean, obteve 'real'.
```

Aqui nós devemos ter vários erros semânticos, uma vez que se: usa uma variável não declarada, atribui-se tipos incompatíveis, tenta-se aceder a um índice não inteiro do array, e usa-se uma condição não booleana.

Este exemplo serve para testar que o analisador consegue identificar vários tipos de erros.

Como vemos através do output, de facto o analisador semântico identifica exatamente esse erro, pelo que podemos afirmar que ele valida bem este tipo de erros.

4.4 Testes do Gerador de Código (MV)

A geração de código para a máquina virtual é a última etapa do compilador, responsável por pegar no código original em pascal, depois deste ter passado pelas outras fases, e transformá-lo em instruções para a VM dada no enunciado do projeto.

Iremos usar os mesmos cinco exemplos usados nas outras fases.

4.4.1 Exemplo 1

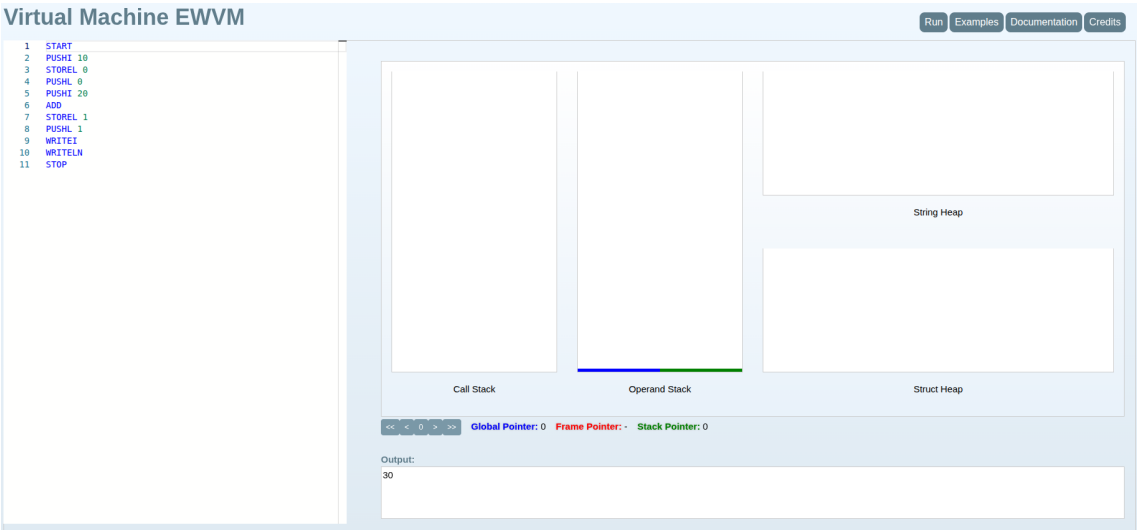
Código de entrada:

```
program exemplo;
var x, y: integer;
begin
    x := 10;
    y := x + 20;
    writeln(y);
end.
```

Resultado:

```
START
PUSHI 10
STOREL 0
PUSHL 0
PUSHI 20
ADD
STOREL 1
PUSHL 1
WRITEI
WRITELN
STOP
```

Aqui vemos que de facto obtemos instruções na linguagem da VM. Se colocarmos esse código resultante na máquina virtual, vemos que obtemos o seguinte, que confirma a correta escolha das instruções por parte desta fase e que, de facto, estas instruções fazem o que o código original em Pascal faz.



4.4.2 Exemplo 2

Código de entrada:

```
program Fatorial;
var
    n, i, fat: integer;
begin
```

```
5       writeln('Introduza um numero inteiro positivo:');
6       readln(n);
7       fat := 1;
8       for i := 1 to n do
9           fat := fat * i;
10      writeln('Fatorial de ', n, ': ', fat);
11  end.
```

Resultado:

```
1   START
2   PUSHS "Introduza um n mero inteiro positivo:"
3   WRITES
4   WRITELN
5   READ
6   ATOI
7   STOREL 0
8   PUSHI 1
9   STOREL 2
10  PUSHI 1
11  STOREL 1
12  LOOP1:
13  PUSHL 1
14  PUSHL 0
15  INFEQ
16  JZ END2
17  PUSHL 2
18  PUSHL 1
19  MUL
20  STOREL 2
21  PUSHL 1
22  PUSHI 1
23  ADD
24  STOREL 1
25  JUMP LOOP1
26  END2:
27  PUSHS "Fatorial de "
28  WRITES
29  PUSHL 0
30  WRITEI
```

```
31 PUSHS ": "
32 WRITES
33 PUSHL 2
34 WRITEI
35 WRITELN
36 STOP
```

Aqui vemos que de facto obtemos instruções na linguagem da VM. Se colocarmos esse código resultante na máquina virtual, vemos que obtemos o seguinte, que confirma a correta escolha das instruções por parte desta fase e que, de facto, estas instruções fazem o que o código original em Pascal faz.

4.4.3 Exemplo 3

Código de entrada:

```
1  program NumeroPrimo;
2      var
3          num, i: integer;
4          primo: boolean;
5      begin
6          writeln('Introduza um numero inteiro positivo:');
7          readln(num);
8          primo := true;
9          i := 2;
10         while (i <= (num div 2)) and primo do
```

```
11          begin
12              if (num mod i) = 0 then
13              primo := false;
14              i := i + 1;
15          end;
16          if primo then
17          writeln(num, ' e um numero primo')
18          else
19          writeln(num, ' nao e um numero primo')
20      end.
```

Resultado:

```
1   START
2   PUSHS "Introduza um n mero inteiro positivo:"
3   WRITES
4   WRITELN
5   READ
6   ATOI
7   STOREL 0
8   PUSHI 1
9   STOREL 2
10  PUSHI 2
11  STOREL 1
12  WHILE1:
13  PUSHL 1
14  PUSHL 0
15  PUSHI 2
16  DIV
17  INFEQ
18  PUSHL 2
19  AND
20  JZ END2
21  PUSHL 0
22  PUSHL 1
23  MOD
24  PUSHI 0
25  EQUAL
26  JZ ELSE3
27  PUSHI 0
```

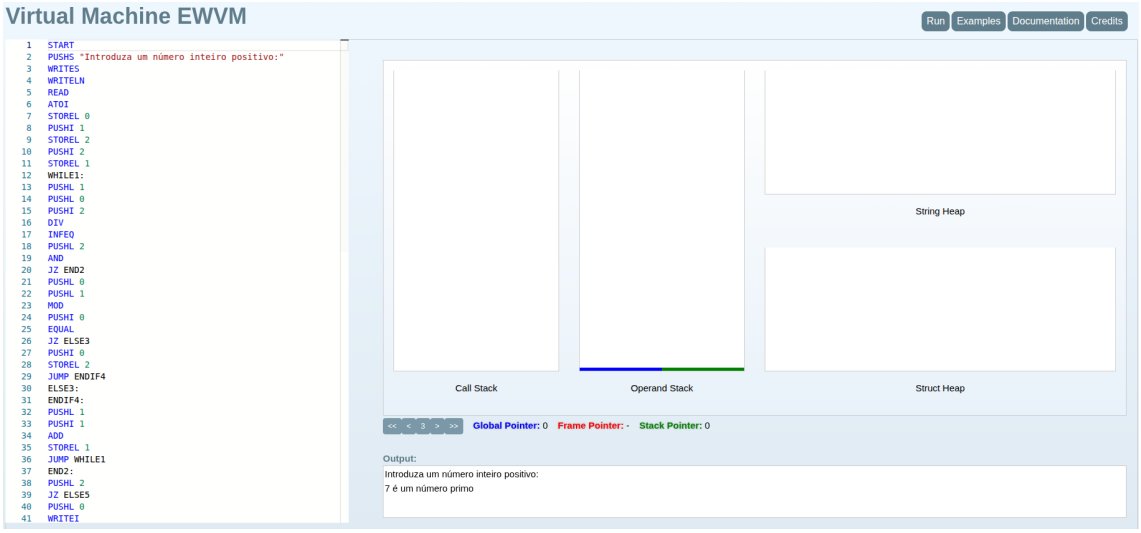
```
28 STOREL 2
29 JUMP ENDIF4
30 ELSE3:
31 ENDIF4:
32 PUSHL 1
33 PUSHI 1
34 ADD
35 STOREL 1
36 JUMP WHILE1
37 END2:
38 PUSHL 2
39 JZ ELSE5
40 PUSHL 0
41 WRITEI
42 PUSHS "␣  ␣um␣n mero␣primo"
43 WRITES
44 WRITELN
45 JUMP ENDIF6
46 ELSE5:
47 PUSHL 0
48 WRITEI
49 PUSHS "␣n o␣  ␣um␣n mero␣primo"
50 WRITES
51 WRITELN
52 ENDIF6:
53 STOP
```

Aqui vemos que de facto obtemos instruções na linguagem da VM. Se colocarmos esse código resultante na máquina virtual, vemos que obtemos o seguinte, que confirma a correta escolha das instruções por parte desta fase e que, de facto, estas instruções fazem o que o código original em Pascal faz.

4.4.4 Exemplo 4

Código de entrada:

```
1 program SomaArray;
2     var
3         numeros: array[1..5] of integer;
4         i, soma: integer;
```




```
5       begin
6           soma := 0;
7           writeln('Introduza 5 numeros inteiros:');
8           for i := 1 to 5 do
9               begin
10                  readln(numeros[i]);
11                  soma := soma + numeros[i];
12              end;
13          writeln('A soma dos numeros e: ', soma);
14      end.
```

Resultado:

```
1   START
2   PUSHI 0
3   STOREL 6
4   PUSHS "Introduza 5 n meros inteiros:"
5   WRITES
6   WRITELN
7   PUSHI 1
8   STOREL 5
9   LOOP1:
10  PUSHL 5
11  PUSHI 5
12  INFEQ
13  JZ END2
```

```
14 PUSHFP
15 PUSHI 0
16 PADD
17 PUSHL 5
18 PUSHI 1
19 SUB
20 PADD
21 READ
22 ATOI
23 STORE 0
24 PUSHL 6
25 PUSHFP
26 PUSHI 0
27 PADD
28 PUSHL 5
29 PUSHI 1
30 SUB
31 PADD
32 LOAD 0
33 ADD
34 STOREL 6
35 PUSHL 5
36 PUSHI 1
37 ADD
38 STOREL 5
39 JUMP LOOP1
40 END2:
41 PUSHS "A␣soma␣dos␣n meros␣  :␣"
42 WRITES
43 PUSHL 6
44 WRITEI
45 WRITELN
46 STOP
```

Aqui vemos que de facto obtemos instruções na linguagem da VM. Se colocarmos esse código resultante na máquina virtual, vemos que obtemos o seguinte, que confirma a correta escolha das instruções por parte desta fase e que, de facto, estas instruções fazem o que o código original em Pascal faz.

4.4.5 Exemplo 5

Código de entrada:

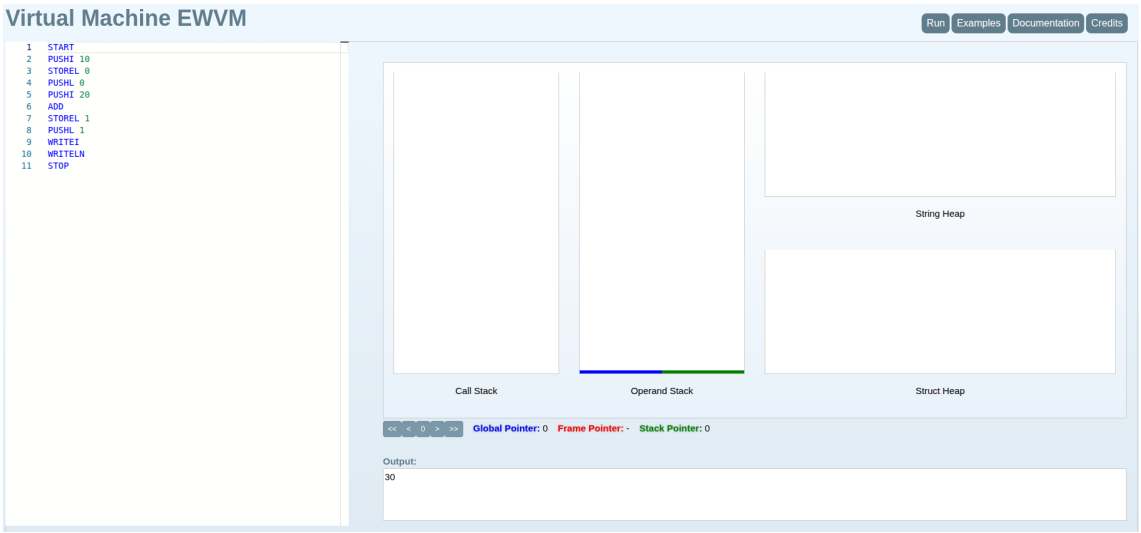
```
(*
    comentario
*)

program exemplo;
    var x, y: integer;

    //isto aqui tambem e um comentario
    begin
        x := 10;
        { isto e um
        comentario
        }
        y := x + 20;
        writeln(y);
    (*
    comentario
    *)
    end.

    (*
    comentario
```

```
23      *)
```

Resultado:

```
1  START
2  PUSHI 10
3  STOREL 0
4  PUSHL 0
5  PUSHI 20
6  ADD
7  STOREL 1
8  PUSHL 1
9  WRITEI
10 WRITELN
11 STOP
```

Aqui vemos que de facto obtemos instruções na linguagem da VM. Se colocarmos esse código resultante na máquina virtual, vemos que obtemos o seguinte, que confirma a correta escolha das instruções por parte desta fase e que, de facto, estas instruções fazem o que o código original em Pascal faz.



Conclusão

O trabalho desenvolvido permitiu a construção de um compilador capaz de traduzir programas escritos em *Pascal Standard* para código executável numa máquina virtual. A implementação seguiu a arquitetura clássica de compiladores, com fases bem definidas de análise léxica, análise sintática, análise semântica e geração de código.

Durante o desenvolvimento, foi possível perceber a importância de cada fase na deteção e prevenção de erros, bem como a necessidade de um planeamento cuidadoso da estrutura do compilador para garantir modularidade e facilidade de manutenção. A utilização da biblioteca *PLY* em Python facilitou a implementação das fases de análise léxica e sintática, permitindo focar nos aspetos semânticos e na geração de código para a máquina virtual.

A fase de geração de código mostrou-se particularmente crítica, exigindo atenção à conversão de tipos, à correta gestão de variáveis e arrays, e ao controlo do fluxo de execução em estruturas condicionais e ciclos. A abordagem modular adotada assegurou que cada componente pudesse ser testado independentemente, aumentando a robustez do compilador.

Em síntese, este trabalho permitiu consolidar conhecimentos de compiladores, reforçando competências em análise de linguagens, estruturas de dados, árvores sintáticas e tradução de programas para uma máquina virtual. Para trabalhos futuros, seria interessante explorar otimizações do código gerado e expandir o suporte a mais funcionalidades da linguagem Pascal.