Grupo 23: Dinis Peixoto A108566, João Taveira A108559, Miguel Gonçalves A108478

Índice

1. Introdução

O presente relatório descreve o desenvolvimento do projeto proposto pelos docentes **José Carlos Leite Ramalho** e **Nuno Filipe Moreira Macedo** no âmbito da unidade curricular de **Processamento de Linguagens e Compiladores**.

Este projeto visa a criação de um compilador para a linguagem *Pascal*, aplicando os conhecimentos adquiridos durante as aulas teóricas e teórico-práticas. O compilador tem como objetivos principais a análise léxica (tokenização), a análise sintática (*parsing*), a análise semântica e a geração de código-máquina. A implementação foi realizada na linguagem **Python**, recorrendo a bibliotecas abordadas em aula, como *re* e *ply.yacc*, bem como a ferramentas auxiliares de automação para otimizar o fluxo de trabalho.

2. Arquitetura Geral do Compilador

2.1. Lexer (Analisador Léxico)

Inicialmente, procedemos à importação da biblioteca *re*, utilizada para a construção de expressões regulares responsáveis pela identificação dos *tokens*. Baseando-nos nos exemplos lecionados, definimos primeiramente as **palavras reservadas** da linguagem (tokens fixos que não podem ser utilizados como identificadores). Posteriormente, definimos os restantes tokens, tais como operadores aritméticos, delimitadores (parêntesis), números e *strings*.

2.2. Gramática

Após a construção do Lexer, demos início à definição da gramática. O processo foi iterativo: começámos por implementar gramáticas baseadas nos exemplos da aula e, gradualmente, unificámo-las numa estrutura única. Durante este processo, tivemos o cuidado de estruturar as regras de modo a tirar partido da capacidade do parser de lidar com a **recursividade à esquerda**, garantindo a correta precedência de operadores e a estrutura da linguagem.

2.3. Parser (Analisador Sintático)

No âmbito do parser, e seguindo a indicação do professor, optamos pelo método **Bottom-Up**, que consiste na construção da árvore sintática partindo dos tokens de entrada e aplicando reduções sucessivas até atingir o símbolo inicial da gramática. A implementação consistiu na transcrição das regras gramaticais para o *ply.yacc*, estruturando as funções de modo a que o parser retornasse uma **Árvore de Sintaxe Abstrata (AST)**. Nesta estrutura, cada nó é representado por um tuplo que inicia com o tipo da operação, seguido pelos seus operandos, facilitando assim a análise e o processamento nas fases futuras.

2.4. Analise semântica

Na análise semântica seguimos a metodologia apresentada na aula, baseada numa abordagem orientada a classes. Começamos por definir o método ***init***, cuja função é inicializar os arrays (ou listas) que guardam toda a informação necessária ao analisador. Utilizamos ainda *scopes* (escopos) para permitir operações como *append* e *pop*, de forma a gerir corretamente as variáveis declaradas em cada nível.

Antes de detalharmos os métodos auxiliares, é fundamental descrever o mecanismo central de travessia da Árvore de Sintaxe Abstrata . A implementação baseia-se no método *visit*, que atua como despachante da análise. Este método utiliza uma estrutura condicional abrangente *if-elif* para verificar o tipo de cada nó encontrado por exemplo, *if*, *while*, *assign*. Com base nessa identificação, o fluxo de execução é encaminhado para o bloco de código responsável por validar especificamente esse tipo de instrução, garantindo assim que todos os ramos da árvore são percorridos recursivamente e devidamente analisados.

De seguida, implementamos o método *declare*, cujo objetivo é verificar se as variáveis já se encontram declaradas e garantir que são declaradas corretamente. Este método também valida a declaração de arrays, assegurando que os índices são válidos e que o tipo associado está bem definido. Depois criámos a função *lookup*, que procura um identificador na pilha de escopos, começando pelo mais interno (local) e avançando até ao mais externo (global). Esta função permite obter o tipo associado a uma variável ou indicar que a variável não foi declarada.

Também implementámos o método auxiliar *get_constant_value*, usado para validar estruturas de dados cujas dimensões precisam de ser conhecidas em tempo de compilação. Este método aplica uma lógica de avaliação estática, permitindo calcular resultados de expressões aritméticas compostas apenas por literais (por exemplo, `10 + 5` ou `2 * 3`). O uso deste método é essencial para registar corretamente arrays na tabela de símbolos, garantindo que o compilador consegue determinar o tamanho da memória a alocar, mesmo quando os limites são definidos através de operações matemáticas simples. Implementámos também o método *get_string_size* , responsável por determinar o tamanho de uma *string*. Este método funciona tanto para literais simples como para expressões que envolvem o operador de concatenação +.

Por fim, implementámos o método *get_type*, que desempenha um papel central na verificação da consistência semântica. Este método valida a **compatibilidade de tipos** em expressões aritméticas, lógicas e relacionais. Para isso, analisa recursivamente os operandos, garantindo que cumprem os requisitos de cada operador (por exemplo, assegurar que uma operação de soma recebe dois valores numéricos). Quando a validação é bem-sucedida, o método infere e devolve o tipo resultante da operação. Caso contrário, regista a incompatibilidade na lista de erros do analisador, permitindo reportar ao utilizador todas as falhas detetadas.

No que diz respeito à alteração de variáveis, o método *visit_assign* assegura a validade das atribuições. Este método verifica se o tipo do valor atribuído é compatível com o tipo da variável de destino, impedindo, por exemplo, que uma *string* seja atribuída a uma variável declarada como inteira.Para a invocação de subprogramas, utilizámos o método *visit_call*. Esta função valida se o identificador corresponde a uma função ou procedimento declarado e verifica rigorosamente a assinatura da chamada: confirma se o número de argumentos e os seus tipos coincidem com os parâmetros definidos na respetiva declaração.

Por fim, o nosso analisador distingue entre erros fatais e avisos (*warnings*). Estes avisos são emitidos em situações que, embora sintaticamente válidas, podem causar problemas em tempo de execução, como o acesso a índices de *arrays* que não puderam ser totalmente validados de forma estática.

2.5. Geração de código

A etapa final do compilador é responsável por traduzir a AST validada numa sequência de instruções para a Máquina Virtual disponibilizada na unidade curricular. Nesta fase, a árvore é percorrida de forma sistemática, e para cada operação é emitida a instrução correspondente, é exemplo disso as instruções de, *PUSHI*, *ADD*, *SUB*, *MUL*, entre outras seguindo o modelo de execução baseado em pilha.

O gerador de código distingue explicitamente entre variáveis globais e variáveis locais: no caso das globais, utiliza as instruções *STOREG* e *PUSHG*, enquanto para variáveis declaradas no interior de funções recorre às instruções *STOREL* e *PUSHL*. Ambos os casos tiram partido dos endereços previamente registados na tabela de símbolos durante a análise semântica, assegurando um acesso coerente e eficiente à memória da Maquina Virtual.

Outro ponto de destaque é o tratamento do acesso a arrays. Como a linguagem *Pascal* permite limites inferiores diferentes de zero (por exemplo, `array[5..10]`), o gerador de código deve normalizar o índice antes de aceder à memória real. Para isso, o compilador insere automaticamente instruções aritméticas que subtraem o limite inferior ao índice utilizado (*OFFSET*). Este ajuste garante que a posição calculada corresponde à posição física correta na memória da Maquina Virtual, independentemente dos limites declarados no código fonte.

3. Testes

Teste 1:

Input:

```
program HelloWorld;

begin

    writeln('Ola, Mundo!');

end.
```

Output:

```
START


PUSHS "Ola, Mundo!"

WRITES

WRITELN

STOP
```

Teste 2:

Input:

```
program Fatorial;

var

n, i, fat: integer;

begin

writeln('Introduza um número inteiro positivo:');

readln(n);

fat := 1;

for i := 1 to n do

fat := fat * i;

writeln('Fatorial de ', n, ': ', fat);

end.
```

Output:

```
START


PUSHN 3

PUSHS "Introduza um número inteiro positivo:"

WRITES

WRITELN

READ

ATOI

STOREG 0

PUSHI 1
```

```
STOREG 2
PUSHI 1
STOREG 1
label0:
PUSHG 1
PUSHG 0
INFEQ
JZ label1
PUSHG 2
PUSHG 1
MUL
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP label0
label1:
PUSHS "Fatorial de "
WRITES
PUSHG 0
WRITEI
PUSHS ": "
WRITES
PUSHG 2
WRITEI
WRITELN
```

```
STOP
```

Teste 3:

Input:

```
program NumeroPrimo;
var
num, i: integer;
primo: boolean;
begin
writeln('Introduza um número 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, ' é um número primo')
else
writeln(num, ' não é um número primo')
end.
```

Output:

```
START


PUSHN 3
PUSHS "Introduza um número inteiro positivo:"
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 2
STOREG 1
label0:
PUSHG 1
PUSHG 0
PUSHI 2
DIV
INFEQ
PUSHG 2
AND
JZ label1
PUSHG 0
PUSHG 1
MOD
PUSHI 0
EQUAL
```

```
JZ label2

PUSHI 0

STOREG 2

label2:

PUSHG 1

PUSHI 1

ADD

STOREG 1

JUMP label0

label1:

PUSHG 2

JZ label3

PUSHG 0

WRITEI

PUSHS " é um número primo"

WRITES

WRITELN

JUMP label4

label3:

PUSHG 0

WRITEI

PUSHS " não é um número primo"

WRITES

WRITELN

label4:

STOP
```

Teste 4:

Input:

```
program SomaArray;

var

numeros: array[1..5] of integer;

i, soma: integer;

begin

soma := 0;

writeln('Introduza 5 números inteiros:');

for i := 1 to 5 do

begin

readln(numeros[i]);

soma := soma + numeros[i];

end;

writeln('A soma dos números é: ', soma);

end.
```

Output:

```
START


PUSHN 3

PUSHI 5

ALLOCN

STOREG 0

PUSHI 0
```

```
STOREG 2

PUSHS "Introduza 5 números inteiros:"

WRITES

WRITELN

PUSHI 1

STOREG 1

label0:

PUSHG 1

PUSHI 5

INFEQ

JZ label1

PUSHG 0

PUSHG 1

PUSHI 1

SUB

READ

ATOI

STOREN

PUSHG 2

PUSHG 0

PUSHG 1

PUSHI 1

SUB

LOADN

ADD

STOREG 2

PUSHG 1
```

```
PUSHI 1

ADD

STOREG 1

JUMP label0

label1:

PUSHS "A soma dos números é: "

WRITES

PUSHG 2

WRITEI

WRITELN

STOP
```

Teste 5:

Input:

```
program BinarioParaInteiro;

function BinToInt(bin: string): integer;

var

i, valor, potencia: integer;

begin

valor := 0;

potencia := 1;

for i := length(bin) downto 1 do

begin

if bin[i] = '1' then

valor := valor + potencia;

potencia := potencia * 2;
```

```
end;

BinToInt := valor;

end;

var

bin: string;

valor: integer;

begin

writeln('Introduza uma string binária:');

readln(bin);

valor := BinToInt(bin);

writeln('O valor inteiro correspondente é: ', valor);

end.
```

Output:

```
START

PUSHN 2

PUSHS ""

STOREG 0

PUSHS "Introduza uma string binária:"

WRITES

WRITELN

READ

STOREG 0

PUSHG 0

PUSHA bintoint

CALL
```

```
STOREG 1

PUSHS "O valor inteiro correspondente é: "

WRITES

PUSHG 1

WRITEI

WRITELN

STOP


bintoint:

PUSHN 4

PUSHI 0

STOREL 2

PUSHI 1

STOREL 3

PUSHL -3

STRLEN

STOREL 1

label0:

PUSHL 1

PUSHI 1

SUPEQ

JZ label1

PUSHL -3

PUSHL 1

PUSHI 1

SUB

CHARAT
```

```
PUSHI 49

EQUAL

JZ label2

PUSHL 2

PUSHL 3

ADD

STOREL 2

label2:

PUSHL 3

PUSHI 2

MUL

STOREL 3

PUSHL 1

PUSHI 1

SUB

STOREL 1

JUMP label0

label1:

PUSHL 2

STOREL 0

PUSHL 0

RETURN
```

4. Extras

Para facilitar a fase de testes, criámos um script extra chamado *webauto.py* usando a biblioteca *Selenium*. Como o código gerado tem de ser corrido numa Máquina Virtual num site, o processo de "copiar e colar" o código manualmente era muito repetitivo e lento. Este script resolve isso: ele abre o navegador sozinho, vai ao site da VM e escreve lá o código automaticamente. Isto poupou-nos imenso tempo a testar o programa.

5. Conclusão

Este projeto permitiu-nos aplicar de forma prática os principais conceitos da unidade curricular de **Processamento de Linguagens e Compiladores**, culminando na criação de um compilador funcional para uma sub-linguagem de Pascal. Ao longo do trabalho implementámos todas as fases do processo de compilação análise léxica, sintática, semântica e geração de código ganhando uma compreensão mais profunda de como cada etapa contribui para transformar o código fonte em instruções executáveis pela Máquina Virtual.

A construção do lexer e do parser consolidou a importância de definir corretamente tokens e regras gramaticais. A análise semântica revelou-se a parte mais exigente, obrigando-nos a implementar gestão de escopos, verificação de tipos, validação de chamadas a funções e tratamento de arrays e strings. Já na geração de código, foi essencial traduzir a AST para instruções da máquina virtual, garantindo o funcionamento correto de variáveis globais e locais, ciclos, expressões e operações sobre dados.

Os testes realizados demonstram que o compilador lida adequadamente com diferentes estruturas da linguagem e comportamentos esperados. O desenvolvimento do script *webauto.py* também facilitou bastante a fase de testes, tornando o processo mais rápido e eficiente.

Em suma, o projeto permitiu-nos consolidar conhecimentos teóricos e desenvolver competências práticas na implementação de um compilador completo. Consideramos que alcançámos os objetivos propostos e criámos uma base sólida que pode ser expandida ou melhorada no futuro.