Compilador Pascal - Grupo 60

João Miguel Mendes Moura - A100615

Maio 2025

Conteúdo

Introdução

Este projeto foi realizado no contexto da unidade curricular de Processamento de Linguagens e visa consolidar conhecimentos em engenharia de linguagens e desenvolvimento de processadores de linguagens segundo o método da tradução dirigida pela sintaxe.

O projeto consiste no desenvolvimento de um compilador para a linguagem *Pascal* standard. O compilador, para além de ser capaz de analisar e interpretar código *Pascal*, deve também produzir código para ser utilizado na máquina virtual fornecida.

Arquitetura

A arquitetura de um compilador *Pascal* é composta por várias partes, incluindo o analisador léxico (*lexer*) e o *parser*. Estas etapas são fundamentais para interpretar o código em linguagem *Pascal* para que possa ser processado e traduzido para código da maquina virtual fornecida.

2.1 Analisador Léxico

O analisador léxico é a primeira etapa do processo de compilação. Ele recebe o código *Pascal* como entrada e divide-o em *tokens*.

Cada *token* representa um elemento individual do código, como palavras, números, símbolos ou operadores. O analisador léxico identifica e classifica esses *tokens*, removendo os espaços em branco.

O resultado é uma sequência de *tokens* organizados, como podemos ver na imagem, que são passados para a próxima etapa do processo de compilação.

```
LexToken(PROGRAM,'program',1,0)
LexToken(ID,'ParOuImpar',1,8)
LexToken(SEMICOLON,';',1,18)
LexToken(VAR,'var',1,20)
LexToken(ID,'Numero',1,28)
LexToken(COLON,':',1,34)
LexToken(INTEGER,'integer',1,36)
LexToken(SEMICOLON,';',1,43)
LexToken(BEGIN,'begin',1,45)
LexToken(WRITELN,'Write',1,55)
LexToken(LPAREN,'(',1,60)
LexToken(STRING_LITERAL,'Digite um número: ',1,61)
LexToken(RPAREN,')',1,81)
LexToken(SEMICOLON,';',1,82)
LexToken(READLN,'ReadLn',1,88)
LexToken(LPAREN,'(',1,94)
LexToken(ID,'Numero',1,95)
LexToken(RPAREN,')',1,101)
LexToken(SEMICOLON,';',1,102)
LexToken(IF,'if',1,108)
LexToken(LPAREN,'(',1,111)
LexToken(ID,'Numero',1,112)
LexToken(MOD,'mod',1,119)
LexToken(NUMBER,2,1,123)
LexToken(RPAREN,')',1,124)
LexToken(EQUAL,'=',1,126)
LexToken(NUMBER,0,1,128)
LexToken(THEN,'then',1,130)
LexToken(WRITELN,'WriteLn',1,143)
LexToken(LPAREN,'(',1,150)
LexToken(STRING_LITERAL,'O número é par.',1,151)
LexToken(RPAREN,')',1,168)
LexToken(ELSE,'else',1,174)
LexToken(WRITELN,'WriteLn',1,187)
LexToken(LPAREN,'(',1,194)
LexToken(STRING_LITERAL,'O número é ímpar.',1,195)
LexToken(RPAREN,')',1,214)
LexToken(SEMICOLON,';',1,215)
LexToken(END,'end',1,217)
LexToken(DOT,'.',1,220)
```

Figura 2.1: Exemplo de *return* de um código *Pascal* separado por tokens

2.2 *Parser*

Após o processo de análise léxica, a sequência de *tokens* é enviada para o *parser*.

O *parser* é responsável por interpretar a estrutura sintática do código *Pascal*.

Durante a análise sintática, o *parser* verifica se a sequência de *tokens* segue as regras gramaticais da linguagem *Pascal* e identifica a estrutura hierárquica do programa, incluindo palavras-chave, operadores e expressões. Com base nessas informações, o *parser* gera o código a ser usado na máquina virtual.

O *input* dado é o *parser* com o teste que se pretende usar, e o resultado é impresso num ficheiro *txt*.

```
joaomoura03@joaomoura03-HP-Pavilion-Laptop-14-dv0xxx:~/3ano2semestre/PL/2avez/PL2425$ python3 parser.py ./tests/pparouimpar.txt
Parsing finalizado
Código VM gerado
```

Figura 2.2: Exemplo de *input* para o *parser*

```
PUSHN 1
PUSHS "Digite um número: "
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHG 0
PUSHI 2
MOD
PUSHI 0
EQUAL
JZ else0
PUSHS "O número é par."
WRITES
WRITELN
JUMP endif1
else0:
PUSHS "O número é ímpar."
WRITES
WRITELN
endif1:
STOP
```

Figura 2.3: Exemplo código da máquina virtual gerado

Implementação

Após a análise léxica e sintática do código, o compilador precisa implementar métodos eficientes para a leitura e execução das operações especificadas no programa.

3.1 Geração de código VM

No que toca à implementação do que faz a geração do código da VM, temos uma lista global que armazena as instruções da máquina virtual.

3.2 Declaração de variáveis

No que toca à declaração de variáveis o *parser* guarda informações sobre cada variável e tem um contador que indica o próximo endereço de memória disponível.

O *parser* começa por identificar a lista de variáveis e identifica o tipo. Depois calcula o espaço necessário para cada variável. No que toca a *arrays*, efetua-se o cálculo do limite superior menos o limite inferior mais um. Após estes cálculos, reserva-se na memória usando o *PUSHN n*.

3.3 Expressões aritméticas

Na interpretação das expressões aritméticas, o parser converte as expressões aritméticas escritas como

$$a + b * c$$

em notação no formato que a máquina virtual entende que é

$$abc * +$$

O *parser* usa regras gramaticais que respeitam a precedência de operadores e com base nestas regras cria uma árvore que depois é processada pela função *process_expression* que percorre essa estrutura de baixo para cima.

3.4 Comandos de controlo de fluxo

3.4.1 Comando *If*

O *parser* converte comandos *if* em saltos condicionais. Em vez de "executar ou não executar"código (como pensamos naturalmente), a máquina virtual "salta sobre"pedaços de código baseado em condições.

O que o *parser* faz é, ele parte o *if* em partes que são a condição, o *then_statement* e o *else_statement*. Depois disto, estas partes são processadas pela *process_expression*.

3.4.2 Comando *while*

O *parser* converte os comandos *while* em um *loop* com teste no início. Cria-se um ciclo ondeo programa volta sempre ao mesmo ponto para testar a condição antes de executar o corpo do *loop*.

A lógica por trás da forma como o *parser* trata o *while* é que o *parser* parte o *while* em partes, sendo estas a condição e o corpo do *loop*. Depois cria *labels* de controlo. Após marcar o início do *loop*, o *parser* processa a condição na *process_expression* emitindo o salto de que se for falso salta para fora do *loop*. De seguida processa o corpo do *loop* com a função *process_statement*. No fim emite o salto para o início do *loop* e marca o fim com a *end_label*.

3.4.3 Comando *for*

No ciclo *for* o *parser* converte o comando *for* em um *loop* controlado automaticamente, isto é, ele inicializa a variável de controlo, testa automaticamente se deve continuar e incrementa/decrementa a variável.

O *parser* à semelhança do ciclo *while* e da condição *if*, divide o ciclo em partes sendo estas, a variável do *loop*, o valor inicial da expressão e o valor final, o corpo do *loop* e a direção (*TO* ou *DOWNTO)*.

TO VS DOWNTO

No ciclo *for* na linguagem *pascal*, podemos ter *TO* que significa que vai de um valor inferior até um valor superior na condição, ou podemos ter também o *DOWNTO* que significa o oposto, isto é, vai de um valor superior até um valor inferior na condição.

No caso do *TO* a instrução da máquina virtual que é emitida é *INFEQ*. No caso do *DOWNTO* a instrução emitida é *SUPEQ*.

3.5 *Procedure*

Em *pascal*, uma *procedure* é quase como se fosse um programa dentro de um programa que pode ser chamado em qualquer lugar, executa o seu código e depois volta para onde foi chamada.

O *parser* primeiro regista a *procedure*: `proc_label = new_label(f"proc{proc_name}")` e adiciona a nova *procedure* à tabela de *procedures* que serve para mapear as *procedures*. De seguida o *parser*, como o código da *procedure* não deve ser executado quando o programa principal roda, o *parser* cria um desvio do código principal.

```
# 1. Marca o início do loop
emit(f"{start_label}:")

# 2. Testa a condição
process_expression(condition)
emit(f"JZ {end_label}")

# 3. Executa o corpo
process_statement(body)

# 4. Volta ao início
emit(f"JUMP {start_label}")

# 5. Marca o fim
emit(f"{end_label}:")
```

Figura 3.1: Lógica do ciclo *while*

Conclusão

Neste projeto adquirimos um largo espetro de conhecimentos. Consolidámos aprendizagens inerentes à UC de Processamento de Linguagens como o uso de expressões regulares e o desenvolvimento de analisadores léxicos e sintáticos criados com o `Lex` e o `Yacc` da versão `PLY` do `Python`.

Implementámos com sucesso um sistema capaz de interpretar várias operações propostas como declaração de variáveis, expressões aritméticas, comandos de controlo de fluxo e *procedures*.

Acreditamos ter atingido os objetivos colocados para a realização do projeto. Pensamos ter implementado um sistema com a capacidade de ler e interpretar códigos com alguma complexidade.

4.1 Trabalho futuro

Por último, apresentamos algumas melhorias e implementações.

A implementação de suporte a *functions* e a integração da máquina virtual para facilitar a execução do sistema.