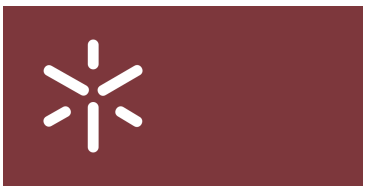

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Processamento de Linguagens

Ano Letivo de 2024/2025

Construção de um Compilador para Pascal Sandard Grupo 23

Bento João Concieiro Guimarães (a96296)

Délio Miguel Lopes Alves (a94557)

João Miguel da Silva Pinto Ferreita (a89497)

June 01, 2025

PL

Índice

1. Introdução

1.1. Enquadramento e Contexto

A linguagem Pascal apresenta uma estrutura clara e rigorosa, sendo fortemente tipada e organizada em blocos bem definidos, o que a torna ideal para a aprendizagem de conceitos fundamentais da programação estruturada. Programas em Pascal seguem uma sequência lógica composta por declarações de variáveis, comandos de controle de fluxo (como if, while, for)

Neste projeto, será desenvolvido um processo de conversão de código fonte escrito em Pascal standard para código executável por uma máquina virtual (VM). O objetivo principal é aprender a traduzir código de uma linguagem de alto nível para um formato mais próximo do nível da máquina. Através desta experiência, pretende-se compreender as etapas envolvidas na construção de um compilador — desde a análise léxica e sintática até à geração de código para a VM — e, assim, adquirir uma compreensão sólida dos princípios que sustentam os compiladores.

Ao realizar esta tradução, temos a oportunidade de explorar os mecanismos internos de interpretação e execução de programas, aprofundando nosso entendimento sobre a linguagem Pascal e sobre o funcionamento de máquinas virtuais, ao mesmo tempo que desenvolvem competências práticas essenciais no campo da engenharia de linguagens.

Este processo de tradução serve como base prática para aprender a construir um compilador para Pascal, permite aplicar os conceitos teóricos na criação de uma ferramenta funcional que transforma código fonte em instruções interpretáveis pela máquina virtual disponibilizada.

1.2. Problema e Objetivos

Neste projeto, o problema consistiu em desenvolver um compilador capaz de traduzir código escrito em Pascal para um formato executável em uma máquina virtual disponibilizada.

Os principais objetivos deste projeto foram:

- **Aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical)**, reforçando a capacidade de escrever gramáticas, tanto independentes de contexto (GIC) quanto tradutoras (GT).
-
- **Desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe**, a partir de uma gramática tradutora.
-
- **Utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc**, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

2. Analisador Léxico

Um analisador léxico, ou lexer, é a primeira fase de um processo mais amplo de análise sintática de linguagens de programação. A sua função é ler uma sequência de caracteres (código fonte) e transformá-la em uma sequência de tokens, que são unidades léxicas com significado na linguagem.

As principais etapas do processo de análise léxica incluem:

- Definição das palavras reservadas da linguagem;
- Criação de expressões regulares para identificar os diferentes tipos de tokens;
- Tratamento de espaços em branco e comentários;
- Deteção e tratamento de erros léxicos.

2.1. Palavras reservadas

A linguagem Pascal possui uma estrutura bem definida, composta por palavras reservadas *(program, begin, end, if, while)* que não podem ser utilizadas como nomes de variáveis ou identificadores. Por isso, essas palavras foram devidamente identificadas e tratadas no nosso analisador léxico.

2.2. Definição de Tokens e Literais

- Número inteiro: NUMBER
- Identificadores: ID, STRING
- Delimitadores: parênteses retos
- Literais e operadores: `'+', '-', '*', '/', '(', ')', '=', '<', '>', ':', ';', '.', ',', ':=', '<>', '<=', '>='`

2.3. Expressões Regulares

As expressões regulares são utilizadas para definir os padrões dos tokens, como por exemplo:

`r'\d+'` : Identifica número inteiro

`r'\'([^\\\n]|(\\.))*?\''` : Identifica strings entre aspas simples (‘)

2.4. Regras de Tokenização

1. **Números:**

   Expressão regular: `r'\d+'`

   Descrição: Identifica números inteiros

2. **Strings**

   Expressão regular: `r'\'([^\\\n]|(\\.))*?\''`

   Descrição: Identifica texto entre aspas simples

3. **Identificadores (ID)**

Expressão regular: `r'[a-zA-Z_][a-zA-Z0-9_]*'`

Descrição: Identifica identificadores válidos que começam com o primeiro caractere por uma letra (a a z ou A a Z) ou um sublinhado (_) e os caracteres seguintes (zero ou mais) podem ser letras, dígitos (0–9) ou sublinhados.

4. **Palavras-chave reservadas**

Expressão regular:

```
reserved = {
  'program': 'PROGRAM',
  'begin': 'BEGIN',
  'end': 'END',
  'var': 'VAR',
  'integer': 'INTEGER',
  'boolean': 'BOOLEAN',
  'true': 'TRUE',
  'false': 'FALSE',
  'if': 'IF',
  'then': 'THEN',
  'else': 'ELSE',
  'while': 'WHILE',
  'do': 'DO',
  'for': 'FOR',
  'to': 'TO',
  'readln': 'READLN',
  'writeln': 'WRITELN',
  'function': 'FUNCTION',
  'procedure': 'PROCEDURE',
  'array': 'ARRAY',
  'of': 'OF',
  'not': 'NOT',
  'and': 'AND',
  'or': 'OR',
  'div': 'DIV',
  'mod': 'MOD',
  'downto': 'DOWNTO',
  'string': 'STRING_TYPE',
  'write': 'WRITE'
}
```

Descrição: Captura palavras-chave reservadas específicas da linguagem.

5. **Operadores e Delimitadores:**

Expressão Regular:

```
  '+', '-', '*', '/', '(', ')', '=', '<', '>', ':', ';', '.', ',', ':=', '<>', '<=',
'>=', '..', '[', ']'
```

2.5. Exemplos de Utilização

```
program Teste;
var
  x: integer;
begin
  x := 10;
  if x > 5 then
    x := x - 1
  else
    x := x + 1;
end.
```

```
LexToken(PROGRAM,'program',1,0)
LexToken(ID,'Teste',1,8)
LexToken(;,';',1,13)
LexToken(VAR,'var',2,15)
LexToken(ID,'x',3,21)
LexToken(:,':',3,22)
LexToken(INTEGER,'integer',3,24)
LexToken(;,';',3,31)
LexToken(BEGIN,'begin',4,33)
LexToken(ID,'x',5,41)
LexToken(ASSIGN,':=',5,43)
LexToken(NUMBER,10,5,46)
LexToken(;,';',5,48)
LexToken(IF,'if',6,52)
LexToken(ID,'x',6,55)
LexToken(>,'>',6,57)
LexToken(NUMBER,5,6,59)
LexToken(THEN,'then',6,61)
LexToken(ID,'x',7,70)
LexToken(ASSIGN,':=',7,72)
LexToken(ID,'x',7,75)
LexToken(-,'-',7,77)
LexToken(NUMBER,1,7,79)
LexToken(ELSE,'else',8,83)
LexToken(ID,'x',9,92)
LexToken(ASSIGN,':=',9,94)
LexToken(ID,'x',9,97)
LexToken(+,'+',9,99)
LexToken(NUMBER,1,9,101)
LexToken(;,';',9,102)
LexToken(END,'end',10,104)
LexToken(.,'.',10,107)
```

3. Analisador Sintático

Após definirmos os tokens que vão ser usados no analisador sintático, nesta segunda fase, passamos para a definição das regras da nossa gramática tradutora capaz de interpretar diferentes estruturas e comandos da linguagem Pascal, traduzir em instruções que podem ser executadas pela máquina virtual.

3.1. Precedência

No processo de análise sintática de uma linguagem de programação, expressões como $a + b * c$ ou $x > y$ *AND* $z = w$ podem ser ambíguas se o analisador não souber em que ordem avaliar os operadores. Para resolver a ambiguidade, definimos uma tabela de precedência.

```
('left', 'OR'),                                        // menor precedência
('left', 'AND'),
('left', 'NOT'),
('left', '=', '<', '>', 'LE', 'GE', 'NE'),
('left', '+', '-'),
('left', '*', '/', 'DIV', 'MOD'),                     // maior precedência
```

3.2. Definição de Regras

1. **Programa:**

   ```
   program : PROGRAM ID ";" block "."
   ```

   Define a estrutura do programa Pascal, o *ID* identifica o nome programa, o *block* é composto declarações e comandos e *"."* delimita o fim do programa.

2. **Bloco código:**

   ```
   block : declarations BEGIN optional_statements END
   ```

   Representa o bloco de código de um programa Pascal. O *BEGIN ... END* delimita o bloco de comandos e *optional_statements* representa os comandos.

3. **Declarações de Variáveis:**

   3.1. **Bloco de Declarações:**

   ```
   declarations : VAR var_declarations
                | empty
   ```

   Um programa pode ou não conter uma secção de declarações de variáveis. O caso *empty* garante que o programa ainda seja válido na ausência da palavra-chave *var*.

   3.2. **Lista de Declarações:**

   ```
   var_declarations : var_declarations var_declaration
                    | var_declaration
   ```

   Permite declarar múltiplas variáveis em linhas diferentes graças a recursividade.

   3.3. **Declaração Individual:**

   ```
   var_declaration : id_list ":" type ";"
   ```

   Numa só linha é possível declarar 1 ou várias variáveis do mesmo tipo.

3.4. **Lista de Identificadores:**

```
id_list : ID
        | id_list ',' ID
```

Permite declarar várias variáveis do mesmo tipo numa linha, separadas por vírgulas.

3.5. **Tipos:**

```
type : INTEGER
     | BOOLEAN
     | STRING_TYPE
     | ARRAY LBRACKET NUMBER DOTDOT NUMBER RBRACKET OF type
```

Uma variável é declarada num tipo inteiro, Booleano, string ou array.

4. **Expressões:**

As expressões são usadas em atribuições, condições, ciclos e chamadas de função. Podem combinar valores, variáveis, operadores e parêntesis.

4.1. **Operações Aritméticas, Relacionais e Lógicas:**

```
expression : expression '+' expression
           | expression '-' expression
           | expression '*' expression
           | expression '/' expression
           | expression DIV expression
           | expression MOD expression
           | expression AND expression
           | expression OR expression
           | expression '=' expression
           | expression '<' expression
           | expression '>' expression
           | expression LE expression
           | expression GE expression
           | expression NE expression
```

As expressões são avaliadas com base na precedência definida referida anteriormente.

4.2. **Operação Negação:**

```
expression : NOT expression
```

Permite aplicar a negação lógica a uma expressão booleana.

4.3. **Acesso a Array:**

```
expression : ID LBRACKET expression RBRACKET
         | variable LBRACKET expression RBRACKET
```

Permite aceder a um elemento de um vetor, a partir um índice.

4.4. **Outras Expressões Suportadas:**

```
expression : NUMBER
         | STRING
         | TRUE
         | FALSE
         | ID
         | variable
         | "(" expression ")"
         | ID "(" expression_list_opt ")"
```

5. **Comandos (Statements):**

As produções que definem os statements representam as instruções executáveis do programa, ou seja, o comportamento do código em tempo de execução. No Pascal, comandos podem ser atribuições, ciclos, condições, entradas/saídas, entre outros.

```
statement : assignment
          | if_statement        // Ciclo if
          | while_loop          // Ciclo while
          | for_loop            // Ciclo for
          | compound_statement
          | writeln
          | write
          | readln
          | empty
```

Esta regra agrupa todos os tipos de comandos válidos. Cada um deles é definido em regras separadas, explicadas a seguir.

5.1. **Função if:**

```
if_statement : IF expression THEN statement ELSE statement
             | IF expression THEN statement
```

Esta regra implementa a estrutura condicional do Pascal. Permite executar um comando se uma condição for verdadeira (then) e, opcionalmente, outro comando se a condição for falsa (else).

5.2. **Função while:**

```
while_loop : WHILE expression DO statement
```

Define um ciclo de repetição enquanto a expressão booleana for verdadeira. A expressão é avaliada antes de cada iteração.

5.3. **Função for:**

```
for_loop : FOR ID ASSIGN expression TO expression DO statement
         | FOR ID ASSIGN expression DOWNTO expression DO statement
```

Define um ciclo com valor inicial e final, podendo incrementar (to) ou decrementar (downto) o contador.

5.4. **compound_statement:**

```
compound_statement : BEGIN statements END
```

Permite agrupar múltiplos comandos como um único bloco. Isso é útil, por exemplo, após if, while, for.

4. Geração de Código para Máquina Virtual

A geração de código segue o modelo de tradução dirigida pela sintaxe, utilizamos a árvore sintática abstrata (AST) construída durante a análise sintática. Em vez de gerar diretamente o código VM linha a linha durante o parsing, o programa primeiro constrói a AST e só depois percorre essa estrutura para produzir o código final.

1. **Construção da AST**

   Após a análise léxica e sintática, o programa fonte em Pascal é transformado numa árvore sintática abstrata. Esta árvore reflete a estrutura lógica do código, representa elementos como declarações, expressões, ciclos e comandos de decisão.

2. **Percurso da AST**

   Um módulo de geração de código percorre recursivamente a AST e, para cada tipo de nó, emite instruções correspondentes da máquina virtual.

3. **Geração do Código VM**

   Durante a travessia, cada construção de alto nível da linguagem é convertida num conjunto de instruções da máquina virtual. Estas instruções simulam o comportamento pretendido em Pascal.

5. Testes

5.1. Teste 1

5.1.1. Código Pascal

```
program HelloWorld;
begin
  writeln('Ola, Mundo!');
end.
```

5.1.2. Código VM

```
START
PUSHS "Ola, Mundo!"
WRITES
WRITELN
STOP
```

5.2. Teste 2

5.2.1. Código Pascal

```
program Maior3;
var
 num1, num2, num3, maior: Integer;
begin
 { Ler 3 números }
 Write('Introduza o primeiro número: ');
 ReadLn(num1);

 Write('Introduza o segundo número: ');
 ReadLn(num2);

 Write('Introduza o terceiro número: ');
 ReadLn(num3);

 { Calcular o maior }
 if num1 > num2 then
  if num1 > num3 then maior := num1
  else maior := num3
 else
  if num2 > num3 then maior := num2
  else maior := num3;

 { Escrever o resultado }
 WriteLn('O maior é: ', maior)
end.
```

5.2.2. Código VM

```
START
PUSHS "Introduza o primeiro número: "
WRITES
READ
ATOI
STOREG 0
PUSHS "Introduza o segundo número: "
WRITES
READ
ATOI
STOREG 1
PUSHS "Introduza o terceiro número: "
WRITES
READ
ATOI
STOREG 2
PUSHG 0
PUSHG 1
SUP
JZ L0
PUSHG 0
PUSHG 2
SUP
JZ L2
PUSHG 0
STOREG 3
JUMP L3
L2:
PUSHG 2
STOREG 3
L3:
JUMP L1
L0:
PUSHG 1
PUSHG 2
SUP
JZ L4
PUSHG 1
STOREG 3
JUMP L5
L4:
PUSHG 2
STOREG 3
L5:
L1:
PUSHS "O maior é: "
WRITES
PUSHG 3
STRI
WRITES
WRITELN
STOP
```

6. Conclusão

Concluído o trabalho prático, podemos afirmar que o projeto foi fundamental para o desenvolvimento e aprimoramento dos nossos conhecimentos na área de processamento de linguagens, a partir da aplicação de expressões regulares até a escrita de gramáticas.

Durante o desenvolvimento do projeto, enfrentamos várias dificuldades que requerem destaque. Nas regras de tokenização, tivemos determinar se seria mais eficaz dividir o analisador léxico em partes específicas. Outra questão central foi a utilização de literais e palavras-chave reservadas. Avaliámos até que ponto seria eficiente incluir símbolos como +, := ou ; diretamente como *literais*, em vez de tokens nomeados, e como distinguir corretamente palavras-chave de identificadores comuns. Também considerámos a implementação de estados no analisador léxico, entre outras dúvidas estruturais relacionadas à tokenização.

Na construção da gramática, a principal dificuldade foi descobrir a melhor abordagem para realizar o parsing do compilador. Identificar a forma ideal para escrever as regras gramaticais que permitissem um parsing adequado ao problema apresentado. Este processo envolveu ajustes contínuos e a resolução de conflitos entre regras gramaticais para garantir a precisão e a robustez do parser desenvolvido. Acreditamos que a arquitetura implementada foi a mais adequada ao problema apresentado.

7. Anexos

7.1. Teste 3

7.1.1. Código Pascal

```
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.
```

7.1.2. Código VM

```
START
PUSHS "Introduza um número inteiro positivo:"
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 1
STOREG 1
L0:
PUSHG 1
PUSHG 0
INFEQ
JZ L1
PUSHG 2
PUSHG 1
MUL
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L0
L1:
PUSHS "Fatorial de "
WRITES
PUSHG 0
STRI
WRITES
PUSHS ": "
WRITES
PUSHG 2
```

```
STRI
WRITES
WRITELN
STOP
```

7.2. Teste 4

7.2.1. Código Pascal

```
 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.
```

7.2.2. Código VM

```
START
PUSHS "Introduza um número inteiro positivo:"
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 2
STOREG 1
L0:
PUSHG 1
PUSHG 0
PUSHI 2
DIV
INFEQ
PUSHG 2
AND
JZ L1
PUSHG 0
PUSHG 1
MOD
PUSHI 0
EQUAL
JZ L2
```

```
PUSHI 0
STOREG 2
JUMP L3
L2:
L3:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L0
L1:
PUSHG 2
JZ L4
PUSHG 0
STRI
WRITES
PUSHS " é um número primo"
WRITES
WRITELN
JUMP L5
L4:
PUSHG 0
STRI
WRITES
PUSHS " não é um número primo"
WRITES
WRITELN
L5:
STOP
```

7.3. Teste 5

7.3.1. Código Pascal

```
 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.
```

7.3.2. Código VM

```
START
PUSHI 5
ALLOCN
STOREG 0
PUSHI 0
STOREG 2
PUSHS "Introduza 5 números inteiros:"
```

```
WRITES
WRITELN
PUSHI 1
STOREG 1
L0:
PUSHG 1
PUSHI 5
INFEQ
JZ L1
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 L0
L1:
PUSHS "A soma dos números é: "
WRITES
PUSHG 2
STRI
WRITES
WRITELN
STOP
```

7.4. Teste 6

7.4.1. Código Pascal

```
program BinarioParaInteiro;
var
  bin: string;
  i, valor, potencia: integer;
begin
  writeln('Introduza uma string binária:');
  readln(bin);
  valor := 0;
  potencia := 1;

  for i := length(bin) downto 1 do
  begin
  if bin[i] = '1' then
        valor := valor + potencia;
      potencia := potencia * 2;
  end;
```

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

7.4.2. Código VM

```
START
PUSHS "Introduza uma string binária:"
WRITES
WRITELN
READ
STOREG 0
PUSHI 0
STOREG 2
PUSHI 1
STOREG 3
PUSHG 0
STRLEN
STOREG 1
L0:
PUSHG 1
PUSHI 1
SUPEQ
JZ L1
PUSHG 0
PUSHG 1
PUSHI 1
SUB
CHARAT
PUSHI 49
EQUAL
JZ L2
PUSHG 2
PUSHG 3
ADD
STOREG 2
JUMP L3
L2:
L3:
PUSHG 3
PUSHI 2
MUL
STOREG 3
PUSHG 1
PUSHI 1
SUB
STOREG 1
JUMP L0
L1:
PUSHS "O valor inteiro correspondente é: "
WRITES
PUSHG 2
STRI
WRITES
WRITELN
STOP
```

7.5. Teste 7

7.5.1. Código Pascal

```
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.
```

7.5.2. Código VM

```
START
PUSHS "Introduza uma string binária:"
WRITES
WRITELN
READ
STOREG 1
PUSHG 1
STOREG 1
PUSHI 0
STOREG 2
PUSHI 1
STOREG 4
PUSHG 1
STRLEN
STOREG 3
L0:
PUSHG 3
PUSHI 1
SUPEQ
JZ L1
PUSHG 1
PUSHG 3
PUSHI 1
SUB
CHARAT
PUSHI 49
EQUAL
JZ L2
PUSHG 2
```

```
PUSHG 4
ADD
STOREG 2
JUMP L3
L2:
L3:
PUSHG 4
PUSHI 2
MUL
STOREG 4
PUSHG 3
PUSHI 1
SUB
STOREG 3
JUMP L0
L1:
PUSHG 2
STOREG 0
PUSHG 0
STOREG 2
PUSHS "O valor inteiro correspondente é: "
WRITES
PUSHG 2
STRI
WRITES
WRITELN
STOP
```