

Construção de um Compilador para Pascal Standard

Processamento de Linguagens e Compiladores

Ricardo Barbosa	Guilherme Duarte	Simão Pereira
A97317	A101122	A102521



Departamento de Informática
Universidade do Minho
janeiro 2026

Resumo

Este relatório descreve o desenvolvimento de um compilador para uma linguagem do tipo Pascal, realizado no âmbito da unidade curricular de Processamento de Linguagens. O compilador foi implementado em Python, recorrendo à biblioteca PLY (Python Lex-Yacc), e integra as fases clássicas da construção de compiladores: análise léxica, análise sintática, análise semântica e geração de código para uma máquina virtual. O sistema desenvolvido suporta declarações de variáveis, expressões aritméticas e booleanas, estruturas de controlo de fluxo, manipulação de arrays e instruções de entrada e saída. A análise semântica assegura a coerência do programa através da verificação de tipos, da validação de declarações e da gestão de escopos. O trabalho demonstra a aplicação prática dos conceitos teóricos abordados na unidade curricular, resultando num compilador funcional e extensível.

Índice

1	Introdução	1
2	Objetivos do Projeto	2
3	Como executar o compilador	3
4	Arquitetura Geral Do Projeto	4
5	Análise Léxica (Lexer)	5
5.1	Descrição geral	5
5.2	Palavras reservadas	5
5.3	Tokens e símbolos	5
5.4	Regras principais	6
6	Análise Sintática	7
6.1	Descrição Geral	7
6.2	Gramática	7
6.3	Estruturas Internas de Suporte ao Parser	9
7	Análise semântica	11
7.1	Declarações e uso de variáveis	11
7.2	Compatibilidade de tipos	11
7.3	Condições em <code>if</code> e <code>while</code>	11
7.4	Arrays e indexação	11
7.5	Strings e indexação	11
7.6	Chamadas de função	12
8	Geração de código para a VM	13
8.1	Abordagem geral	13
8.2	Inicialização de variáveis	13
8.3	Atribuições	13
8.4	Leitura (READLN)	13
8.5	Escrita (WRITELN)	14
8.6	Controlo de fluxo	14
8.7	Funções	14
9	Testes	15
10	Considerações adicionais	16
10.1	Decisão de não usar AST	16
10.2	Tratamento de erros	16
11	Trabalho futuro	17
12	Conclusão	18

1 Introdução

Este relatório apresenta o desenvolvimento de um compilador para uma linguagem do tipo Pascal, realizado no âmbito da unidade curricular de Processamento de Linguagens. O objetivo do trabalho foi construir um compilador funcional capaz de analisar e traduzir programas para código executável numa máquina virtual. O compilador segue a arquitetura clássica de compiladores, integrando as fases de análise léxica, sintática e semântica, bem como a geração de código. A implementação foi realizada em Python, recorrendo à biblioteca PLY (Python Lex-Yacc). O sistema suporta declarações de variáveis, expressões, estruturas de controlo, arrays e instruções de entrada e saída, assegurando a coerência do código através de verificações semânticas. O presente relatório descreve, ainda, a organização do compilador, as principais decisões de implementação e os testes realizados para validação do seu funcionamento.

2 Objetivos do Projeto

Os principais objetivos deste projeto são:

- Implementar um analisador léxico para reconhecimento de tokens da linguagem Pascal;
- Desenvolver um analisador sintático baseado numa gramática livre de contexto;
- Efetuar verificação semântica, nomeadamente tipos, declarações e compatibilidade de operações;
- Gerar código intermédio para uma máquina virtual baseada em pilha;
- Suportar estruturas de controlo, funções, arrays e operações de entrada/saída.

3 Como executar o compilador

O compilador foi desenvolvido em Python e utiliza a biblioteca `PLY` (Python Lex-Yacc). Com a introdução de um *automatizador*, o processo de execução foi simplificado, permitindo que o utilizador compile programas Pascal sem recorrer explicitamente a redirecionamento de entrada padrão. O script `programPASCAL.py` funciona como ponto de entrada do sistema, sendo responsável por inicializar o analisador léxico, o analisador sintático e todo o processo de geração de código para a máquina virtual. Para executar o compilador, basta utilizar o seguinte comando a partir da raiz do projeto:

```
python .\Projeto\programPASCAL.py
```

O automatizador gere internamente a leitura do ficheiro de entrada, invoca os módulos responsáveis pela análise léxica e sintática e apresenta como saída o código intermédio gerado para a máquina virtual. Esta abordagem torna a utilização do compilador mais simples e reduz a probabilidade de erros por parte do utilizador.

Em alternativa, a estrutura modular do projeto permite que os componentes do compilador sejam reutilizados ou integrados noutros contextos, mantendo a flexibilidade da implementação.

O analisador léxico encontra-se num módulo separado e é importado pelo parser. A arquitetura modular permite evoluir o compilador mantendo as responsabilidades bem definidas.

4 Arquitetura Geral Do Projeto

O sistema desenvolvido reconhece um subconjunto expressivo de Pascal, incluindo:

- Estrutura de programa: `program ...; var ... begin ... end.`
- Tipos: `integer`, `real`, `boolean`, `char`, `string`
- Arrays: `array[i..j] of T`
- Expressões aritméticas, relacionais e booleanas
- Entrada/saída: `readln` e `writeln`
- Controlo de fluxo: `if/then/else`, `while`, `for to/downto`
- Funções definidas pelo utilizador com parâmetros e valor de retorno

O compilador inclui verificações semânticas essenciais (variáveis não declaradas, duplicações, compatibilidade de tipos em operações e atribuições, validação de índices) e produz código para uma máquina virtual baseada em pilha, utilizando instruções como `PUSHI`, `PUSHS`, `ADD`, `JZ`, `JUMP`, `CALL`, `RETURN`, entre outras. (Implementação baseada no código fornecido: `lexer` e `parser`.)

5 Análise Léxica (Lexer)

5.1 Descrição geral

A etapa de análise léxica tem como objetivo converter o código Pascal numa sequência de tokens, identificando palavras reservadas, identificadores, números, strings, operadores e símbolos. A implementação foi feita em PLY através do módulo `ply.lex`.

A estratégia adotada baseia-se em:

- Um dicionário de **palavras reservadas** (`reserved`) que mapeia lexemas para tipos de token;
- Uma lista `tokens` que inclui tokens gerais e todos os valores do dicionário `reserved`;
- Regras simples (`t_...`) para símbolos/operadores;
- Funções para `STRING`, `REAL`, `INTEGER` e `IDENTIFICADOR`;
- Ignorar comentários, espaços e gerir contagem de linhas;
- Tratamento de erros lexicais com `t_error`.

5.2 Palavras reservadas

As palavras reservadas incluem a estrutura do programa, controlo de fluxo, operadores lógicos, tipos e I/O. Exemplos:

- Estrutura: `program`, `var`, `begin`, `end`
- Estruturas: `array`, `of`
- Controlo: `if`, `then`, `else`, `while`, `for`, `to`, `downto`, `do`
- Funções: `function`, `length`
- Lógicos: `not`, `and`, `or`, `div`, `mod`
- I/O: `readln`, `writeln`
- Booleanos: `true`, `false`
- Tipos: `integer`, `real`, `boolean`, `char`, `string`

A regra de identificadores consulta `reserved` para decidir se o lexema é palavra reservada ou apenas um identificador.

5.3 Tokens e símbolos

Para além das palavras reservadas, foram definidos tokens para:

- **Identificadores:** `IDENTIFICADOR`
- **Literais:** `INTEGER`, `REAL`, `STRING`
- **Operadores:** `+` `-` `*` `/`, `=` `<>` `<` `>` `<=` `>=`, `:=`
- **Delimitadores e pontuação:** `(` `)` `[` `]` `;` `:` `,` `.` `..`

5.4 Regras principais

Strings Strings Pascal são reconhecidas entre aspas simples e armazenadas sem as aspas:

Listing 1: Regra de strings (simplificada)

```
def t_STRING(t):  
    r"\ '([^\']*)\ '"  
    t.value = t.value[1:-1]  
    return t
```

Números inteiros e reais Inteiros e reais são distinguidos por regex dedicada. O valor do token é convertido para `int` ou `float`.

Identificadores e palavras reservadas A regra de identificadores retorna `IDENTIFICADOR` ou uma palavra reservada, via `reserved.get(...)`.

Comentários O lexer ignora comentários em dois formatos: `{ ... }` e `(* ... *)`. Isto permite que o programador comente livremente sem afetar a análise.

Erros lexicais Quando um caractere não corresponde a nenhuma regra, é impresso um erro e o lexer avança 1 caractere, permitindo continuar a análise.

6 Análise Sintática

6.1 Descrição Geral

A análise sintática tem como objetivo validar a estrutura do programa a partir da sequência de tokens produzida pelo analisador léxico, garantindo a conformidade com a gramática definida para a linguagem. Esta fase foi implementada em Python com recurso ao módulo `ply.yacc` da biblioteca `PLY`, permitindo a especificação formal das regras sintáticas e a verificação da correção estrutural do código. O resultado desta etapa é uma representação estruturada do programa, que serve de base para as fases seguintes do processo de compilação.

6.2 Gramática

A gramática foi implementada com o objetivo de reconhecer as construções suportadas pelo compilador e garantir a correta análise estrutural dos programas de entrada. As regras definidas permitem processar declarações, instruções e expressões de forma consistente com a linguagem alvo, assegurando a validação sintática do código. Segue abaixo um excerto da implementação da gramática, evidenciando a organização das regras e a cobertura das principais estruturas da linguagem:

```
start_ -> PROGRAM IDENTIFICADOR ; declaracoes BEGIN bloco_comandos END .
```

```
declaracoes -> declaracoes declaracao_tipo  
            | e
```

```
declaracao_tipo -> dec_variaveis  
                | dec_funcao
```

```
dec_variaveis -> VAR lista_vars
```

```
lista_vars -> lista_vars lista_ids : tipo_variaveis ;  
            | e
```

```
lista_ids -> IDENTIFICADOR  
           | lista_ids , IDENTIFICADOR
```

```
tipo_variaveis -> INTEGER_TYPE  
                | REAL_TYPE  
                | BOOLEAN_TYPE  
                | CHAR_TYPE  
                | STRING_TYPE  
                | ARRAY [ INTEGER .. INTEGER ] OF tipo_variaveis
```

```
dec_funcao -> FUNCTION IDENTIFICADOR ( parametros ) :  
tipo_variaveis ; dec_variaveis_f BEGIN bloco_comandos_f END ;
```

```

parametros -> parametro
    | parametro ; parametros
    | e

parametro -> parametro lista_ids : tipo_variaveis
    | e

bloco_comandos -> lista_comandos_pv comando
    | lista_comandos_pv
    | comando
    | e

lista_comandos_pv -> comando_pv
    | lista_comandos_pv comando_pv

comando_pv -> comando ;

comando -> atribuicao
    | leitura
    | escrita
    | if_h
    | while_h
    | for_h
    | return
    | BEGIN bloco_comandos END

atribuicao -> IDENTIFICADOR := expressao
    | IDENTIFICADOR := chamada_funcao

leitura -> READLN ( IDENTIFICADOR )
    | READLN ( IDENTIFICADOR [ expressao ] )

escrita -> WRITELN ( argumentos_escrita )

argumentos_escrita -> argumento
    | argumentos_escrita , argumento
    | e

argumento -> STRING
    | IDENTIFICADOR
    | IDENTIFICADOR [ expressao ]

if_h -> IF expressao THEN comando
    | IF expressao THEN comando ELSE comando

while_h -> WHILE expressao DO comando

```

```

for_h -> FOR IDENTIFICADOR := expressao TO expressao DO comando
      | FOR IDENTIFICADOR := expressao DOWNTO expressao DO comando

chamada_funcao -> IDENTIFICADOR ( lista_ids )
               | IDENTIFICADOR ( )

expressao -> expressao + expressao
          | expressao - expressao
          | expressao * expressao
          | expressao / expressao
          | expressao DIV expressao
          | expressao MOD expressao
          | expressao AND expressao
          | expressao OR expressao
          | expressao = expressao
          | expressao <> expressao
          | expressao < expressao
          | expressao > expressao
          | expressao <= expressao
          | expressao >= expressao
          | NOT expressao
          | ( expressao )
          | IDENTIFICADOR
          | IDENTIFICADOR [ expressao ]
          | INTEGER
          | REAL
          | STRING
          | TRUE
          | FALSE
          | LENGTH ( IDENTIFICADOR )

return -> IDENTIFICADOR := IDENTIFICADOR ;

```

6.3 Estruturas Internas de Suporte ao Parser

Durante a análise sintática, o parser utiliza diversas variáveis internas que permitem controlar o estado da análise e armazenar informações intermédias, essenciais para a validação semântica e a posterior geração de código para a máquina virtual. Estas variáveis não fazem parte da gramática do Pascal, mas suportam o funcionamento do compilador. Posto isto, as variáveis auxiliares deste programa são:

- **parser.index**

Mantém a posição atual de memória disponível. Cada nova variável declarada recebe o endereço correspondente, incrementando automaticamente o conta-

dor. Para arrays, o avanço ocorre em múltiplas posições, uma para cada elemento, garantindo a alocação correta.

- **parser.regists**

Um dicionário que mapeia cada nome de variável para o respetivo endereço na memória da máquina virtual. Permite localizar rapidamente onde cada variável está armazenada, por exemplo, `parser.regists["x"] = 0` indica que x ocupa a primeira posição da stack global.

- **parser.var_types**

Armazena os tipos das variáveis, incluindo detalhes específicos para arrays (como início, fim e tipo de elemento). Exemplos de tipos: integer, string, boolean ou estruturas de arrays descritas como ('array', início, fim, tipo_elemento). Esta informação é usada para inicialização adequada e para gerar instruções corretas na EWVM.

- **parser.labels** Contém identificadores únicos utilizados para controlar fluxos como IF, FOR e WHILE. Cada novo label é gerado com base no tamanho atual da lista, evitando repetições e conflitos, produzindo sequências como L0, L1, L2...

- **parser.success**

Indicador lógico que sinaliza se o parsing decorreu corretamente. Se False, o processo é interrompido e o código não é produzido. Esta variável é essencial para detetar erros como tipos inconsistentes ou variáveis não declaradas.

7 Análise semântica

A componente semântica está embutida nas ações das regras do parser. As verificações mais relevantes incluem:

7.1 Declarações e uso de variáveis

- Variáveis usadas antes de serem declaradas geram erro e anulam a compilação.
- Duplicações na mesma declaração ou redeclarações globais são sinalizadas.

7.2 Compatibilidade de tipos

Atribuições Ao compilar `x := expr`, o compilador obtém:

- tipo de `x` em `parser.var_types`
- tipo inferido de `expr`

e valida que são compatíveis. Caso contrário, é emitida uma mensagem de erro.

Operações em expressões As expressões retornam um par (`codigo`, `tipo`). Para operadores:

- Aritméticos (`+` `-` `*` `/` `div` `mod`): exigem inteiros (na implementação atual)
- Booleanos (`and` `or` `not`): exigem booleanos
- Relacionais (`=` `<>` `<` `>` `<=` `>=`): exigem tipos compatíveis e produzem booleano

Nota: na implementação atual, operações aritméticas são restringidas a `integer`. Existe token e tipo `real`, mas o suporte aritmético a reais não está totalmente generalizado (p.ex., a emissão de instruções VM específicas para real depende da VM).

7.3 Condições em `if` e `while`

Expressões condicionais devem ser do tipo `boolean`. Se a expressão não for booleana, o compilador assinala erro e marca o processo como falhado.

7.4 Arrays e indexação

- Acesso a `a[i]` exige que `a` seja array e que `i` seja inteiro.
- A implementação calcula o endereço efetivo com base no índice inicial do array.

7.5 Strings e indexação

A indexação em strings é suportada através de `CHARAT`. O compilador traduz `s[i]` em código que obtém o carácter na posição (ajustando indexação 1-based do Pascal).

7.6 Chamadas de função

Numa chamada $f(x, y, \dots)$, o compilador:

- verifica se cada argumento é variável declarada
- recolhe os tipos dos argumentos
- compara com a assinatura registada em `parameters_f_static`

Em caso de erro de aridade ou tipos, a compilação falha com mensagem explicativa.

8 Geração de código para a VM

8.1 Abordagem geral

A geração de código é **direta** (sem construção de AST). Cada regra do parser devolve imediatamente:

- uma lista de instruções da VM
- e, no caso de expressões, também o tipo inferido

Esta estratégia simplifica o compilador e permite depuração rápida, mas limita otimizações globais e separação clara entre *frontend* e *backend*.

8.2 Inicialização de variáveis

No bloco `var`, cada variável é inicializada:

- `integer/boolean`: `PUSHI 0`
- `string`: `PUSHS`
- `array`: empilha um valor por elemento, de acordo com o tipo do elemento

8.3 Atribuições

Atribuição simples `x := expr` compila para:

- código de `expr` (que deixa o valor no topo da pilha)
- `STOREG addr(x)`

Atribuição com função `x := f(...)` gera:

- `PUSHA f`
- `CALL`
- `STOREG addr(x)`

8.4 Leitura (READLN)

Variáveis A leitura usa `READ` e converte para o tipo da variável:

- `atoi` para `integer/boolean`
- `atof` para `real`
- strings são usadas diretamente

Leitura para elemento de array A leitura em `a[i]` calcula o endereço e usa `STOREN`.

8.5 Escrita (WRITELN)

A escrita suporta mistura de:

- Strings literais (PUSHS ...; WRITES)
- Variáveis simples (seleciona WRITEI/WRITEF/WRITES)
- Elementos de array (gera acesso e depois escreve de acordo com tipo)

Ao final, emite sempre WRITELN para nova linha.

8.6 Controlo de fluxo

If/Then/Else O compilador gera labels e usa JZ para saltar quando a condição é falsa, emitindo JUMP quando existe ramo `else`.

While Um ciclo `while` é traduzido com:

- label de início
- avaliação da condição
- salto para label de fim com JZ
- corpo do ciclo
- salto de volta ao início

For To / Down To O `for` inicializa a variável de controlo, compara com o limite e atualiza por +1 ou -1, usando INFEQ ou SUPEQ conforme o sentido.

8.7 Funções

As funções são emitidas como blocos identificados por label (ex.: `nomeFuncao:`). A implementação inclui:

- gestão de parâmetros (alocação em contexto de função)
- corpo com comandos e controlo de fluxo (versões `_f` das regras)
- retorno via mecanismo RETURN

O compilador valida coerência do identificador de retorno e do tipo retornado antes de aceitar a função.

9 Testes

Os testes devem cobrir progressivamente:

- **I/O e estrutura mínima:** `writeln('Olá');`
- **Aritmética:** atribuições e expressões com precedência
- **Condições:** `if` simples e `if-else`
- **Ciclos:** `while` e `for` `to/downto`
- **Arrays:** declaração, leitura/escrita e indexação
- **Strings:** `length(s)` e `s[i]`
- **Funções:** assinatura, chamada e retorno

```
=== A correr ex2.pas ===
START
PUSHI 0
PUSHI 0
PUSHI 0
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
READ
ATOI
STOREG 2
PUSHG 0
PUSHG 1
INF
JZ L2
PUSHG 1
PUSHG 2
INF
JZ L0
PUSHS "a < b < c"
WRITES
WRITELN
JUMP L1
L0:
PUSHS "a < b mas b >= c"
WRITES
WRITELN
L1:
L2:
PUSHS "Fim do programa"
WRITES
WRITELN
STOP
```

```
Programa:
program TesteIfAninhado;
var
  a, b, c: integer;
begin
  readln(a);
  readln(b);
  readln(c);

  if a < b then
    if b < c then
      writeln('a < b < c')
    else
      writeln('a < b mas b >= c');

  writeln('Fim do programa')
end.
```

10 Considerações adicionais

10.1 Decisão de não usar AST

Foi adotada a estratégia de gerar código durante o parsing, evitando uma fase intermediária de AST. Esta decisão:

- reduz complexidade e acelera o desenvolvimento;
- facilita ver resultados imediatos;
- dificulta otimizações e reescritas globais;
- concentra análise semântica nas ações das produções.

10.2 Tratamento de erros

O tratamento de erros ocorre em vários níveis:

- **Léxico:** caracteres inválidos são reportados e ignorados.
- **Sintático:** `p_error` reporta o token problemático e a linha.
- **Semântico:** regras verificam declarações e tipos, marcando `parser.success = False`.

11 Trabalho futuro

Existem extensões naturais para evoluir o compilador:

- **AST**: separar parsing e geração de código; permitir otimizações.
- **Otimizações**: folding de constantes, eliminação de código morto, melhorias locais.
- **Tipos numéricos**: suporte completo a **real** em operações e instruções VM específicas.
- **Mais Pascal**: procedimentos, escopos aninhados, passagem por referência, etc.
- **Erros mais ricos**: coluna exata, recuperação de erro, múltiplos erros por execução.

12 Conclusão

O compilador desenvolvido cumpre os objetivos principais do projeto: processa programas Pascal Standard com variáveis, expressões, controlo de fluxo, arrays, strings, I/O e funções, realizando validação sintática e semântica e gerando código para uma máquina virtual baseada em pilha.

A abordagem de geração direta durante o parsing mostrou-se eficaz e prática para o contexto académico, resultando num sistema funcional, extensível e adequado para demonstrar os conceitos fundamentais de compilação: lexing, parsing, semântica e geração de código.

A Anexo A — Tabela resumo de tokens

- **Tokens gerais:** IDENTIFICADOR, INTEGER, REAL, STRING
- **Operadores:** MAIS, MENOS, VEZES, DIVIDIR, IGUAL, DIFERENTE, MENOR, MAIOR, MENOR_IGUAL, MAIOR_IGUAL, ATRIBUICAO
- **Símbolos:** PA, PF, PRA, PRF, PONTO_VIRGULA, DOIS_PONTO, VIRGULA, PONTO, PONTO_PONTO
- **Reservadas:** PROGRAM, VAR, ARRAY, OF, BEGIN, END, IF, THEN, ELSE, WHILE, DO, FOR, TO, DOWNT, FUNCTION
- **Lógicos:** NOT, AND, OR, DIV, MOD
- **I/O:** READLN, WRITELN
- **Constantes:** TRUE, FALSE
- **Tipos:** INTEGER_TYPE, REAL_TYPE, BOOLEAN_TYPE, CHAR_TYPE, STRING_TYPE
- **Função nativa:** LENGTH