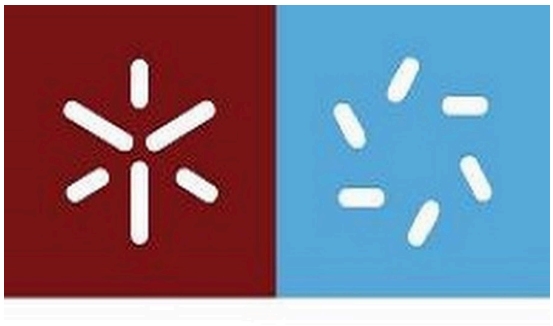

Universidade do Minho
Escola de Ciências

Processamento de Linguagens e Compiladores

2025/2026

Projeto

Construção de um Compilador para Pascal Standard

Grupo 25

Joao Durães n°A109065

João Neiva n° A108579

Ricardo Vilaça n°A102879

Indice

1. Introdução

Este trabalho descreve a conceção e implementação de um compilador de Pascal, cujo objetivo principal é a geração de código para uma Máquina Virtual. O projeto cobre o ciclo completo de compilação: desde a receção do ficheiro .pas, passando pela validação semântica e estruturação em AST, até às instruções assembly finais.

O compilador, desenvolvido em Python utilizando as ferramentas Lex e Yacc, possui as seguintes funcionalidades principais:

- Gestão de Memória: Declaração de variáveis primitivas e estruturadas (arrays unidimensionais).
- Lógica e Aritmética: Suporte completo para expressões algébricas, comparativas e operadores lógicos.
- Fluxo de Controlo: Implementação de condicionais (if-then-else) e ciclos (while e for).
- Interação I/O: Comandos para leitura de dados e escrita formatada de diversos tipos de dados.

Ao longo deste relatório, exploramos a arquitetura do compilador, a definição da gramática utilizada e as estratégias adotadas para converter conceitos de alto nível em operações de baixo nível interpretáveis pela VM.

2. Etapas do Projeto

Organizámos o projeto por etapas. Primeiro tratámos da análise léxica, que lê o código e identifica palavras, símbolos e operadores. Depois passámos para a análise sintática, onde aplicámos as regras gramaticais para dar estrutura ao programa e garantir que tudo fazia sentido. Em seguida, fizemos a análise semântica para validar a coerência do código e evitar combinações incompatíveis.

Por fim, gerámos as instruções para a máquina virtual executar e confirmámos que o compilador estava a funcionar como esperado através de vários testes, com exemplos diferentes de código em Pascal.

3. Análise Léxica (Lexer)

O analisador léxico, implementado no ficheiro lex.py, é o primeiro passo do compilador: lê o código fonte e divide-o em tokens. Nesta fase são identificadas e classificadas coisas como palavras reservadas, identificadores, operadores e números. Para o fazer, usamos a biblioteca PLY, definindo regras (expressões regulares) que transformam o texto original numa sequência de tokens pronta a ser usada na análise sintática.

3.1. Funcionalidades do Lexer

O lexer não se limita a reconhecer tokens também foi preparado para lidar com detalhes práticos do código Pascal e tornar as fases seguintes mais simples. Ao longo da leitura, ignora espaços e tabulações para que a formatação não influencie o resultado, e remove comentários para que estes não interfiram com a análise.

Mantém ainda a contagem correta de linhas, atualizando-a sempre que encontra quebras de linha, o que é fundamental para indicar a localização de erros de forma clara. Sempre que faz sentido, o lexer converte alguns valores para um formato mais adequado (por exemplo, literais numéricos), em vez de os manter apenas como texto.

Por fim, quando encontra caracteres inválidos, reporta o problema de forma controlada e avança na leitura, permitindo continuar a análise e facilitar a depuração do código.

3.2. Implementação do Lexer

Começámos por definir o conjunto de palavras-reservadas da linguagem, através de um dicionário que associa cada palavra ao respetivo tipo de token. Isto garante que a linguagem seja case-insensitive, pois convertemos o identificador para minúsculas antes da verificação:

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

De seguida, definimos a lista de tokens reconhecidos pelo lexer, incluindo tokens de identificadores, literais e operadores compostos:

```
tokens = [
    'ID', 'REAL_NUMBER', 'NUMBER', 'STRING_LITERAL',
    'ASSIGN', 'EQUALS', 'NOT_EQUALS',
    'LESS_THAN', 'LESS_THAN_OR_EQUAL_TO',
    'GREATER_THAN', 'GREATER_THAN_OR_EQUAL_TO',
    'RANGE'
] + list(reserved.values())
```

Para simplificar o lexer, definimos como literals os símbolos de um único carácter (pontuação e operadores simples). Espaços e tabulações são ignorados através da variável t_ignore.

```
literals = [';', ',', '(', ')', '.', ':', '[', ']', '+', '-', '\*', '/']
t_ignore = ' \t'
```

Os operadores compostos (com mais do que um carácter) são reconhecidos com funções de regra próprias, garantindo que sequências como := ou .. não sejam confundidas com símbolos individuais:

```
def t_ASSIGN(t):
    r':='
    return t

def t_NOT_EQUALS(t):
    r'<>|!='
    return t

def t_RANGE(t):
    r'\.\.'
    return t
```

A regra de identificadores (t_ID) trata também o reconhecimento de palavras-reservadas de forma case-insensitive, e converte os valores true e false para booleanos nativos de Python:

```
def t_ID(t):
    r'[a-zA-Z\_][a-zA-Z0-9\_]\*'
    t.type = reserved.get(t.value.lower(), 'ID')
    if t.type == 'TRUE':
        t.value = True
    elif t.type == 'FALSE':
        t.value = False
    return t
```

Por fim, incluímos suporte a comentários nos dois formatos do Pascal ({ ... } e (...)), contagem de linhas através de t_newline e tratamento de caracteres inválidos com t_error, permitindo reportar erros com a linha correta e continuar a análise.

4. Análise Sintática (Parser)

Nesta etapa, o objetivo é validar se a sequência de tokens obtida anteriormente respeita a estrutura hierárquica da linguagem. Para tal, utilizámos a ferramenta `ply.yacc`, que implementa um algoritmo de parsing LALR(1).

4.1. Implementação do Parser

O analisador sintático foi construído através de funções Python que definem a gramática e as ações associadas. O ponto central desta fase é a coordenação entre as regras de produção e a resolução de ambiguidades.

Para garantir que expressões complexas sejam processadas corretamente sem gerar conflitos de shift/reduce, definimos explicitamente a precedência e a associatividade dos operadores. Isto permite que o parser decida corretamente a ordem de operações (por exemplo, garantir que a multiplicação tem prioridade sobre a adição):

```
precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('right', 'NOT'),
    ('left', 'EQUALS', 'NOT_EQUALS', 'LESS_THAN',
            'LESS_THAN_OR_EQUAL_TO',
            'GREATER_THAN',
            'GREATER_THAN_OR_EQUAL_TO'),
    ('left', '+', '-'),
    ('left', '*', '/', 'DIV', 'MOD'),
    ('right', 'UMINUS'),
)
```

Uma funcionalidade essencial do nosso parser é a capacidade de reportar erros de forma informativa sem interromper a execução imediatamente. Através da função p_error, o sistema identifica o token problemático e a linha onde este ocorreu, permitindo ao utilizador corrigir o código fonte com precisão:

```
def p_error(p):
    if p:
        print(f"Erro de sintaxe no token '{p.value}' (tipo: {p.type}) na linha
{p.lineno}")
        parser.errok() # Tenta recuperar e continuar a análise
    else:
        print("Erro de sintaxe: fim de arquivo inesperado")
```

O parser percorre os tokens e, a cada redução de regra bem-sucedida, executa uma ação para construir a estrutura que será detalhada mais à frente. O uso do método parser.errok() dentro do tratamento de erros é vital para que o compilador consiga encontrar múltiplos erros sintáticos numa única passagem, em vez de parar no primeiro problema encontrado.

5. Gramática

A gramática da linguagem desenvolvida baseia-se na sintaxe do Pascal, seguindo uma estrutura formal que separa a definição de dados da lógica de execução. Esta é definida através de uma gramática independente do contexto, utilizando a notação BNF (Backus-Naur Form) para especificar as produções permitidas.

A linguagem está organizada em três blocos principais:
- Bloco de Identificação: Definido pela palavra-reservada `program`, que atribui um nome ao identificador global.
- Bloco Declarativo: Onde são definidas as variáveis globais (`var`), procedimentos (`procedure`) e funções (`function`). A gramática permite o suporte a tipos primitivos (`integer`, `real`, `boolean`, `char`, `string`) e tipos compostos (`array`).
- Bloco Executável: Delimitado pelos termos `begin` e `end`, contendo a lista de instruções que compõem o fluxo do programa.

A gramática foi desenhada para suportar estruturas de controlo comuns na programação imperativa:
- Seleção: Implementada através da estrutura `if-then-else`, permitindo a execução condicional de blocos.
- Iteração: Suporte para ciclos `while-do` (controlo à cabeça) e `for to/downto` (iteração contada).
- Subprogramação: As funções e procedimentos permitem a passagem de parâmetros e a criação de escopos locais, promovendo a modularidade do código.

6. Representação Intermédia (AST)

A Árvore de Sintaxe Abstrata (AST) é a representação interna do programa após a validação gramatical. Ao contrário de uma árvore de derivação completa, a AST simplifica a estrutura, removendo elementos puramente sintáticos (como parênteses de agrupamento ou pontos e vírgulas) e focando-se na hierarquia lógica das operações.

6.1. Estrutura da Árvore

Nesta implementação, a AST é construída utilizando tuplos aninhados em Python. Cada nó da árvore é representado por um tuplo onde o primeiro elemento é uma "etiqueta" (tag) que identifica o tipo de instrução ou expressão, seguida pelos seus componentes.

Esta estrutura foi escolhida pela sua imutabilidade e facilidade de travessia durante as fases de análise semântica e interpretação na máquina virtual.

6.2. Exemplo

Para ilustrar a transformação do código fonte na representação intermédia, considere-se o seguinte trecho de código Pascal:

```
if (x > 0) then
x := x + 1;
```

O parser gera o seguinte nó da AST para esta instrução:

```
(
  'if',
  ('binop', '>', ('var', 'x'), 0),
  ('assign', ('var', 'x'), ('binop', '+', ('var', 'x'), 1)),
  None
)
```

Os nós da AST no projeto dividem-se em quatro categorias principais:

- Declarações: Nós como (‘var_decl’, lista_id, tipo) que definem o ambiente de dados.
- Expressões: Operações binárias (‘binop’, op, esq, dir), unárias ou acessos a variáveis.
- Instruções: Estruturas de controlo como (‘while’, cond, corpo) e (‘if’, cond, then, else).
- Subprogramas: Definições de funções e procedimentos, que incluem a sua própria lista de parâmetros e bloco de instruções local.

Esta representação intermédia permite que o analisador semântico percorra a árvore de forma recursiva para verificar tipos e escopos de variáveis antes da execução final.

7. Análise Semântica

A fase de análise semântica é responsável por verificar a consistência lógica do programa, garantindo que as construções sintaticamente corretas obedecem às regras de tipagem e escopo da linguagem Pascal. Esta etapa foi implementada no módulo `semantica.py` utilizando o padrão *Visitor*, percorrendo a Árvore de Sintaxe Abstrata (AST) gerada anteriormente.

- Tabela de Símbolos e Gestão de Escopo

A gestão de identificadores é realizada através da classe `TabelaSimbolos`. Para suportar o aninhamento de subprogramas (funções e procedimentos) e o escopo global, a tabela utiliza uma estrutura de pilha de dicionários:

- Escopo Global: É a base da pilha, contendo variáveis globais e definições de funções/procedimentos.
- Escopos Locais: Sempre que uma função ou procedimento é visitado (`entrar_escopo`), um novo dicionário é empilhado. Ao sair do subprograma (`sair_escopo`), o dicionário é removido.

A busca de variáveis (`procurar_variavel`) ocorre do topo para a base da pilha, garantindo que variáveis locais ocultem variáveis globais de mesmo nome (shadowing).

- Verificação de Tipos

O `AnalisadorSemantico` implementa uma tipagem forte e estática. As principais validações incluem:

- Compatibilidade de Atribuição: Verifica se o tipo da variável à esquerda é compatível com a expressão à direita. O compilador permite coerção implícita de `INTEGER` para `REAL`, mas proíbe o inverso sem conversão explícita.
- Expressões Aritméticas e Lógicas: Garante que operadores matemáticos (`+`, `-`, `*`) operam sobre números e operadores lógicos (`AND`, `OR`) sobre booleanos. A concatenação de strings é suportada pelo operador `+`.
- Controlo de Fluxo: As expressões condicionais de estruturas `IF` e `WHILE` são obrigatoriamente verificadas para garantir que resultam num tipo `BOOLEAN`.
- Validação de Subprogramas

Durante a análise de chamadas de função (`visit_call`), o analisador verifica:
- Se o identificador foi declarado como função ou procedimento.
- A correspondência do número de argumentos passados versus declarados.
- A compatibilidade de tipo de cada argumento individualmente.

8. Geração de Código

A etapa final do compilador, implementada em `maquina.py`, traduz a AST validada para instruções de uma Máquina Virtual baseada em pilha (Stack-Based VM).

- Modelo de Memória e Endereçamento

O gerador de código distingue explicitamente entre memória global e local, calculando endereços e *offsets* durante a compilação:

- Variáveis Globais: São alocadas sequencialmente. O compilador mantém um contador `endereco_atual` e mapeia cada variável global a um endereço estático. O acesso é feito via instruções `PUSHG` (carregar) e `STOREG` (armazenar).
- Variáveis Locais e Parâmetros: Dentro de funções, o compilador calcula *offsets* relativos ao apontador de registo de ativação (*Frame Pointer*):
  - Parâmetros: Possuem *offsets* negativos (ex: `FP[-1]`, `FP[-2]`).
  - Locais: Possuem *offsets* positivos, iniciados em 0.
  - O acesso utiliza as instruções `PUSHL` e `STOREL`.
- Tradução de Estruturas de Controlo

A linearização do código Pascal requer o uso de *labels* e saltos condicionais. O método `novo_label` gera identificadores únicos para gerir o fluxo:

- Condicionais (IF-THEN-ELSE): Utiliza a instrução `JZ` (*Jump Zero*) para saltar para o bloco `ELSE` ou para o fim da instrução caso a condição seja falsa.
- Ciclos (WHILE/FOR): São implementados com dois *labels*: um no início para reavaliação da condição e outro no final para saída. No caso do ciclo `FOR`, o compilador injeta automaticamente as instruções de incremento ou decremento da variável de controlo.
- Manipulação de Arrays

O suporte a *arrays* unidimensionais envolve o cálculo de endereços em tempo de execução. O compilador gera código para:

- Colocar o endereço base do array na pilha (`PUSHGP` + offset).
- Calcular o índice ajustado (subtraindo o índice mínimo declarado, já que Pascal permite intervalos arbitrários como `1..10` ou `5..15`).
- Utilizar as instruções `LOADN` e `STOREN` para acesso indireto à memória.
- Exemplo de Geração de Instruções

Para uma operação binária simples como `a + b`, a máquina de pilha gera a sequência:

```
PUSHL 0   ; Carrega 'a' (local)
PUSHL 1   ; Carrega 'b' (local)
ADD       ; Soma o topo da pilha
```

9. Exemplos de Testes

9.1. Teste1

9.1.1. Código Pascal

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

9.1.2. Código VM

```
    JUMP main
  main:
    START
    PUSHS "Ola, Mundo!"
    WRITES
    WRITELN
    STOP
```

9.2. Teste2

9.2.1. 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.
```

9.2.2. Código VM

```
    JUMP main
  main:
    START
    PUSHN 3
    PUSHS "Introduza um número inteiro positivo:"
    WRITES
    WRITELN
    READ
    ATOI
    STOREG 0
    PUSHI 1
    STOREG 2
    PUSHI 1
    STOREG 1
```

```
  label1:
    PUSHG 1
    PUSHG 0
    INFEQ
    JZ label2
    PUSHG 2
    PUSHG 1
    MUL
    STOREG 2
    PUSHG 1
    PUSHI 1
    ADD
    STOREG 1
    JUMP label1
  label2:
    PUSHS "Fatorial de "
    WRITES
    PUSHG 0
    WRITEI
    PUSHS ": "
    WRITES
    PUSHG 2
    WRITEI
    WRITELN
    STOP
```

9.3. Teste3

9.3.1. 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.
```

9.3.2. Código VM

```
  JUMP main
main:
  START
  PUSHN 3
  PUSHS "Introduza um número inteiro positivo:"
  WRITES
  WRITELN
  READ
  ATOI
  STOREG 0
  PUSHI 1
  STOREG 2
  PUSHI 2
  STOREG 1
label1:
  PUSHG 1
  PUSHG 0
  PUSHI 2
  DIV
  INFEQ
  PUSHG 2
  AND
  JZ label2
  PUSHG 0
  PUSHG 1
  MOD
  PUSHI 0
  EQUAL
  JZ label4
  PUSHI 0
  STOREG 2
label4:
  PUSHG 1
  PUSHI 1
  ADD
  STOREG 1
  JUMP label1
label2:
  PUSHG 2
  JZ label5
  PUSHG 0
  WRITEI
  PUSHS " é um número primo"
  WRITES
  WRITELN
  JUMP label6
label5:
  PUSHG 0
  WRITEI
  PUSHS " não é um número primo"
  WRITES
  WRITELN
label6:
  STOP
```

9.4. Teste4

9.4.1. 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.
```

9.4.2. Código VM

```
    JUMP main
  main:
    START
    PUSHN 7
    PUSHI 0
    STOREG 6
    PUSHS "Introduza 5 números inteiros:"
    WRITES
    WRITELN
    PUSHI 1
    STOREG 5
  label1:
    PUSHG 5
    PUSHI 5
    INFEQ
    JZ label2
    READ
    ATOI
    STOREG 7
    PUSHGP
    PUSHI 0
    PADD
    PUSHG 5
    PUSHI 1
    SUB
    PUSHG 7
    STOREN
    PUSHG 6
    PUSHGP
    PUSHI 0
    PADD
    PUSHG 5
```

```
    PUSHI 1
    SUB
    LOADN
    ADD
    STOREG 6
    PUSHG 5
    PUSHI 1
    ADD
    STOREG 5
    JUMP label1
  label2:
    PUSHS "A soma dos números é: "
    WRITES
    PUSHG 6
    WRITEI
    WRITELN
    STOP
```

9.5. Teste5

9.5.1. 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.
```

9.5.2. Código VM

```
    JUMP main
BinToInt:
  PUSHN 3
  PUSHI 0
  STOREL 1
  PUSHI 1
  STOREL 2
  PUSHL -1
  STRLEN
  STOREL 0
label1:
  PUSHL 0
  PUSHI 1
  SUPEQ
  JZ label2
  PUSHL -1
  PUSHL 0
  PUSHI 1
  SUB
  CHARAT
  PUSHS "1"
  CHRCODE
  EQUAL
  JZ label4
  PUSHL 1
  PUSHL 2
  ADD
  STOREL 1
label4:
  PUSHL 2
  PUSHI 2
  MUL
  STOREL 2
  PUSHL 0
  PUSHI 1
  SUB
  STOREL 0
  JUMP label1
label2:
  PUSHL 1
  STOREL -2
  PUSHL -2
  RETURN
main:
  START
  PUSHN 2
  PUSHS "Introduza uma string binária:"
  WRITES
  WRITELN
  READ
  STOREG 0
  PUSHI 0
  PUSHG 0
  PUSHA BinToInt
```

```
CALL
STOREG 1
PUSHS "O valor inteiro correspondente é: "
WRITES
PUSHG 1
WRITEI
WRITELN
STOP
```

10. Conclusão

O presente projeto resultou no desenvolvimento de um compilador funcional para a linguagem Pascal Standard, capaz de gerar código para uma Máquina Virtual. Utilizando a biblioteca PLY em Python, implementaram-se com sucesso as quatro fases essenciais: análise léxica, análise sintática (geração da AST), validação semântica e geração de código máquina.

O principal desafio superado foi a correta tradução das estruturas de alto nível para a lógica de baixo nível da VM, especificamente a gestão da stack frame para suporte a funções e o cálculo de offsets para variáveis locais e parâmetros. O compilador final processa corretamente operações aritméticas, controlo de fluxo, arrays e manipulação de strings, cumprindo integralmente os requisitos propostos.

Em suma, este trabalho permitiu consolidar a aplicação prática da teoria de compiladores, demonstrando a transformação completa de código-fonte em instruções executáveis.