Relatório: Construção de um compilador para Pascal Standard

Unidade Curricular: Processamento de Linguagens e Compiladores (PLC)
Ano Letivo: 2025/2026
Grupo: 19
Autores:

- **Rodrigo da Silva** (A108661)
- **Tomás Viana Lima** (A108488)

Índice

---

1. Introdução

Este projeto visa o desenvolvimento de um compilador completo para um subconjunto da linguagem **Pascal (Standard Pascal)**, capaz de traduzir código de alto nível para linguagem *Assembly* compatível com uma Máquina Virtual (VM) baseada em pilha (*Stack Machine*).

O sistema foi implementado em **Python**, utilizando a biblioteca **PLY (Python Lex-Yacc)** para a análise léxica e sintática. O objetivo central deste projeto não foi apenas a tradução de código, mas a criação de um compilador **robusto, modular e seguro**, com um foco especial na **Análise Semântica** e **Verificação de Tipos** (*Type Safety*) antes da geração de código.

2. Arquitetura do Sistema

Para garantir a escalabilidade e a manutenção do código, adotou-se uma arquitetura estritamente modular, separando a lógica de *parsing*, a gestão de memória e a definição de estruturas de dados em ficheiros distintos.

O processo de compilação foi desenhado como uma *pipeline* linear, onde a saída de um módulo alimenta o seguinte, garantindo que a geração de código só ocorre após a validação semântica:

`Lexer` → `Parser` → `Análise Semântica` → `Gerador de Código`

2.1. Organização dos Módulos

O projeto encontra-se dividido nos seguintes componentes:

1. **`src/lexer.py` (Frontend):** Responsável pela tokenização, normalização de input (*case-insensitivity*) e filtragem de comentários.
2. **`src/ast_nodes.py` (Estrutura de Dados):** Define as classes da Árvore de Sintaxe Abstrata (AST), permitindo uma representação hierárquica do programa em memória.
3. **`src/semantics.py` (Motor Semântico):** Módulo dedicado exclusivamente à lógica de negócio. Contém a classe `SymbolTable`, responsável por controlar escopos (Global vs Local), calcular *offsets* de memória e gerir assinaturas de funções.
4. **`src/parser.py` (Backend e Orquestração):** O núcleo do compilador. Contém a gramática formal (BNF), a lógica de construção da AST, o sistema de inferência de tipos e o gerador de código final.

---

3. Análise Léxica e Sintática

A análise sintática valida a estrutura gramatical do código fonte. A gramática foi desenhada para resolver ambiguidades comuns através da definição explícita de precedência de operadores.

- **Precedência:** Operadores multiplicativos (`*`, `/`, `DIV`, `MOD`) têm prioridade sobre aditivos (`+`, `-`), e operadores lógicos (`NOT`) têm a prioridade máxima.

Esta hierarquia é definida diretamente na configuração do *Parser* (`parser.py`), garantindo que expressões matemáticas complexas são avaliadas na ordem correta:

```
# src/parser.py
precedence = (
    ('left', 'OR'), ('left', 'AND'),                    # Lógicos
(Baixa)
    ('nonassoc', 'EQ', 'NEQ', 'LT', 'GT', 'LE', 'GE'),  # Relacionais
    ('left', 'PLUS', 'MINUS'),                          # Aditivos
    ('left', 'TIMES', 'DIV', 'MOD', 'SLASH'),           #
Multiplicativos
    ('right', 'NOT'),                                   # Unário (Alta)
)
```

- **Distinção de Operadores:** O *Lexer* distingue semanticamente a divisão real (`/`) da divisão inteira (`div`). Enquanto `/` gera sempre um resultado real, o `div` exige operandos inteiros. Esta distinção é crucial para garantir que a instrução correta é enviada para a VM.

---

4. Análise Semântica e Verificação de Tipos (Type Safety)

O diferencial deste compilador reside na sua capacidade de **validação semântica**. Ao contrário de tradutores simples que delegam os erros para o tempo de execução, este sistema verifica a consistência dos tipos de dados durante a compilação.

4.1. Tabela de Símbolos e Escopos

A classe `SymbolTable` gere o ciclo de vida das variáveis e a alocação de memória. O método `get` implementa a lógica de procura, dando prioridade ao escopo local (*shadowing*) antes de consultar o global:

```
# src/semantics.py
def get(self, name):
    n = self.normalize(name)
    # 1. Procura no Escopo Local (se não estivermos no global)
    if self.scope != 'global':
        if n in self.locals: return self.locals[n]

    # 2. Procura no Escopo Global
    if n in self.globals: return self.globals[n]

    # 3. Erro (Fail-Fast)
    print(f"Erro Semântico: Variável '{name}' não definida.")
    sys.exit(1)
```

- **Escopo Global:** Variáveis acessíveis em todo o programa (instruções `pushg`/`storeg`).
- **Escopo Local e Argumentos:** Ao entrar numa função, cria-se um novo contexto. Os argumentos recebem *offsets* negativos (relativos ao *Frame Pointer*), enquanto as variáveis locais recebem *offsets* positivos sequenciais.

4.2. Lógica de Inferência de Tipos e "Fail-Fast"

Implementou-se um sistema de inferência (`infer_type`) que percorre a AST para validar operações aritméticas e de atribuição. A integridade dos tipos é verificada *antes* de qualquer instrução ser emitida.

O excerto abaixo demonstra a implementação da estratégia **Fail-Fast** no momento de uma atribuição:

```
# src/parser.py (dentro da função gen - Assign)
if expr_type != 'UNKNOWN' and var_type != 'UNKNOWN':
     # Permite promoção implícita de Inteiro para Real
     if var_type == 'REAL' and expr_type == 'INTEGER': pass

     # Se os tipos forem incompatíveis, aborta a compilação imediatamente
     elif var_type != expr_type:
```

```
            print(f"⚠  ERRO SEMÂNTICO: Tentativa de atribuir {expr_type} a
{var_type}")
            sys.exit(1)
```

4.3. Tratamento do tipo `ANY`

Para suportar a flexibilidade do retorno de funções em Pascal, implementou-se uma lógica especial que atribui o tipo interno `ANY` à variável de retorno da função. Isto permite a atribuição de valores sem gerar falsos positivos na verificação de tipos, mantendo a segurança nas restantes operações.

Esta lógica é gerida na **Tabela de Símbolos**, detetando quando o utilizador tenta atribuir um valor ao próprio nome da função:

```
# src/semantics.py
if n == self.scope: # Variável de retorno da função
    ret_off = -(self.curr_func_args + 1)
    # 'any' permite atribuir qualquer valor validado pelo parser sem erro
de tipo
    return {'scope': 'return', 'offset': ret_off, 'type': 'any'}
```

---

5. Geração de Código

A geração de código segue o padrão *Visitor*, percorrendo a AST validada e emitindo instruções para a Stack Machine.

5.1. Estruturas de Controlo

As estruturas `If`, `While` e `Repeat` são traduzidas utilizando *labels* e saltos condicionais (`JZ`, `JUMP`). O compilador gera etiquetas únicas dinamicamente para gerir o fluxo de execução:

```
# src/parser.py (Exemplo de Tradução do IF-THEN-ELSE)
elif isinstance(node, If):
    l1, l2 = new_label(), new_label()
    gen(node.cond)     # Gera código da condição
    emit(f"jz {l1}")   # Se falso, salta para o Else (L1)
    gen(node.then_b)   # Corpo do Then
    emit(f"jump {l2}") # Salta o Else (vai para o fim L2)
    emit(f"{l1}:")     # Label do Else
    if node.else_b: gen(node.else_b)
    emit(f"{l2}:")     # Label de Fim
```

5.2. Gestão de Memória (Heap e Arrays)

A implementação de Arrays utiliza alocação dinâmica na *Heap*:

1. Na declaração, é emitida a instrução `alloc N`.

2. No acesso (`arr[i]`), o compilador calcula o endereço da célula em tempo de execução: soma o endereço base do *pointer* ao índice desejado e utiliza `storen` (guardar) ou `loadn` (ler) para manipulação direta da memória.

```
# src/parser.py (Gestão de Arrays)

# 1. Alocação (Declaração)
if isinstance(kind, dict) and kind['kind']=='array':
    emit(f"alloc {kind['size']}")

# 2. Acesso (Leitura v[i])
gen(node.index_expr) # Calcula índice
emit("pushi 1")
emit("sub")          # Ajuste de índice (Pascal é 1-based, VM é 0-based)
emit("loadn")        # Leitura da Heap (Pointer + Offset)
```

5.3. Mapeamento de Instruções (AST -> VM)

A tradução das operações da árvore sintática para a *Stack Machine* é direta e eficiente. A tabela abaixo ilustra o mapeamento entre os nós da AST e as instruções de *assembly* geradas:

Operador Pascal	Nó da AST (Python)	Instrução VM Gerada
`+` (Soma)	`BinOp(left, '+', right)`	`add`
`*` (Multiplicação)	`BinOp(left, '*', right)`	`mul`
`div` (Divisão Int)	`BinOp(left, 'DIV', right)`	`div`
`/` (Divisão Real)	`BinOp(left, '/', right)`	`div`
`and` (Lógico)	`BinOp(left, 'AND', right)`	`mul`
`or` (Lógico)	`BinOp(left, 'OR', right)`	`add`
`=` (Igualdade)	`BinOp(left, '=', right)`	`equal`

6. Validação e Testes

O compilador foi validado através de um conjunto abrangente de testes, divididos em três categorias para garantir a conformidade funcional e a robustez.

6.1. Testes Base (Requisitos do Guião)

Teste	Funcionalidade Validada	Resultado
`ola.pas`	I/O básico e Strings literais	✅ Sucesso
`fatorial.pas`	Recursividade e gestão da pilha de funções	✅ Sucesso
`primo.pas`	Operadores lógicos e aritmética (`mod`)	✅ Sucesso

Teste	Funcionalidade Validada	Resultado
`array.pas`	Alocação dinâmica e indexação de vetores	✅ Sucesso
`binario.pas`	Manipulação de Strings (`charat`) e conversão	✅ Sucesso

6.2. Testes Adicionais (Funcionalidades Extra)

Teste	Objetivo	Resultado
`fibonacci.pas`	Lógica sequencial e atualização de variáveis	✅ Sucesso
`soma.pas`	Leitura de input (`readln`) e acumulação	✅ Sucesso
`temperatura.pas`	Precedência de operadores aritméticos	✅ Sucesso
`div.pas`	**Distinção semântica entre divisão inteira e real**	✅ Sucesso

6.3. Testes de Robustez e Filosofia "Fail-Fast"

Estes testes foram criados para **falhar propositadamente**, provando a eficácia das guardas semânticas.

Princípio *Fail-Fast*: Adotámos uma política de terminação imediata em caso de erro semântico. Ao detetar uma operação inválida (ex: incompatibilidade de tipos), o compilador interrompe a execução (`sys.exit`), garantindo que nenhum ficheiro de saída `.vm` é gerado. Isto assegura que apenas programas logicamente corretos avançam para a fase de execução.

Teste	Cenário de Erro	Comportamento do Compilador
`erro_soma.pas`	Tentativa de somar Inteiro + String	🛡️ **Bloqueado:** `Erro Semântico` detetado
`erro_atribuicao.pas`	Atribuir String a variável Inteira	🛡️ **Bloqueado:** `Erro Semântico` detetado
`erro_div.pas`	Usar `div` com resultado Real	🛡️ **Bloqueado:** `Erro Semântico` detetado

7. Conclusão

O projeto resultou num compilador funcional e seguro. A arquitetura modular e a introdução da **Análise Semântica com Verificação de Tipos** elevam a qualidade da solução, garantindo que o código gerado para a VM é não só sintaticamente válido, mas também logicamente coerente. Todos os requisitos propostos foram cumpridos e superados com a implementação de validações de robustez.

---

8. Como Executar

Pré-requisitos: Python 3 e biblioteca PLY.

Compilação: Para gerar o código máquina (`.vm`) a partir de um ficheiro Pascal:

```
./run.sh testes/nome_do_teste.pas
```