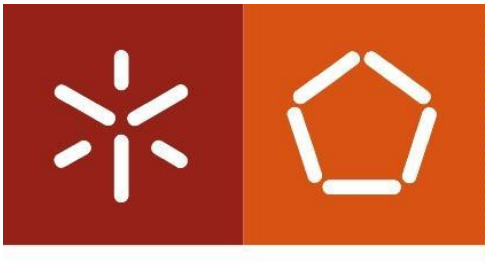


Processamento de Linguagens

Relatório Trabalho Prático
Compilador Pascal Standard
Grupo 54 LEI - 3º Ano - 2º Semestre

Ano Letivo 2024/2025

Diogo Gonçalves
A101919

Tiago Carneiro
A93207

Tiago Guedes
A97369

Braga,
1 de junho de 2025

Conteúdo

1 Introdução **3**

2 Arquitetura do Compilador **3**
2.1 Análise Léxica 3
2.2 Análise Sintática 4
2.3 Análise Semântica 9
2.3.1 Tabela de Símbolos 9
2.3.2 Principais Verificações Semânticas 9
2.3.3 Fluxo da Análise Semântica 9
2.3.4 Funções e Procedimentos Incorporados 10
2.4 Geração de Código 10

3 Testes **11**
3.1 Exemplo 7 — Conversão de Binário para Inteiro 11
3.2 Exemplo Extra — Função de Soma 12

4 Otimizações e Extras **13**
4.1 Otimização do Length 13
4.2 Eliminação de Operações Neutras 13
4.3 Instruções Específicas por Tipo 13
4.4 Pré-Cálculo dos Limites de Ciclos 13
4.5 Retorno Eficiente de Funções 13
4.6 Extras 13

5 Dificuldades na Implementação **14**

6 Conclusão **14**

Lista de Figuras

1 Arquitetura do Compilador 3
2 Classe do nodo da AST de ForStatement 8
3 Código Pascal a transformar para AST 8
4 Árvore AST simplificada para o programa `HelloWorld`. 8
5 Verificações efetuadas para *VariableDeclaration* 10
6 Função *visitor* para o nó da AST correspondente a um *IfStatement* 11
7 Código em Pascal para o Exemplo 7 e respetivo código VM gerado. 12
8 Código em Pascal de uma função de soma e respetivo código VM gerado. 12

1 Introdução

O presente relatório documenta o processo de conceção e desenvolvimento de um compilador para a linguagem Pascal Standard, realizado no contexto da unidade curricular de Processamento de Linguagens. Este projeto teve como principal objetivo implementar um sistema capaz de analisar programas escritos em Pascal, assegurando a sua correção tanto do ponto de vista sintático como semântico. Para além da análise, o compilador é também responsável pela geração de código intermédio compatível com uma máquina virtual disponibilizada no âmbito da unidade curricular.

2 Arquitetura do Compilador

O compilador foi estruturado segundo as fases clássicas da construção de compiladores, conforme abordado nesta UC. Estas fases são:

- **Análise Léxica**
- **Análise Sintática**
- **Análise Semântica**
- **Geração de Código**

Cada uma destas etapas é responsável por transformar e validar progressivamente a entrada do programa, desde o código-fonte original até à geração de código para a máquina virtual. O processo culmina na construção da Árvore de Sintaxe Abstrata (AST) e da Tabela de Símbolos, sendo a AST essencial para a geração de código posterior para a VM, e a Tabela de Símbolos crucial para a analise semântica e deteção de erros.

Em baixo conseguimos ver uma representação visual da arquitetura do nosso compilador:

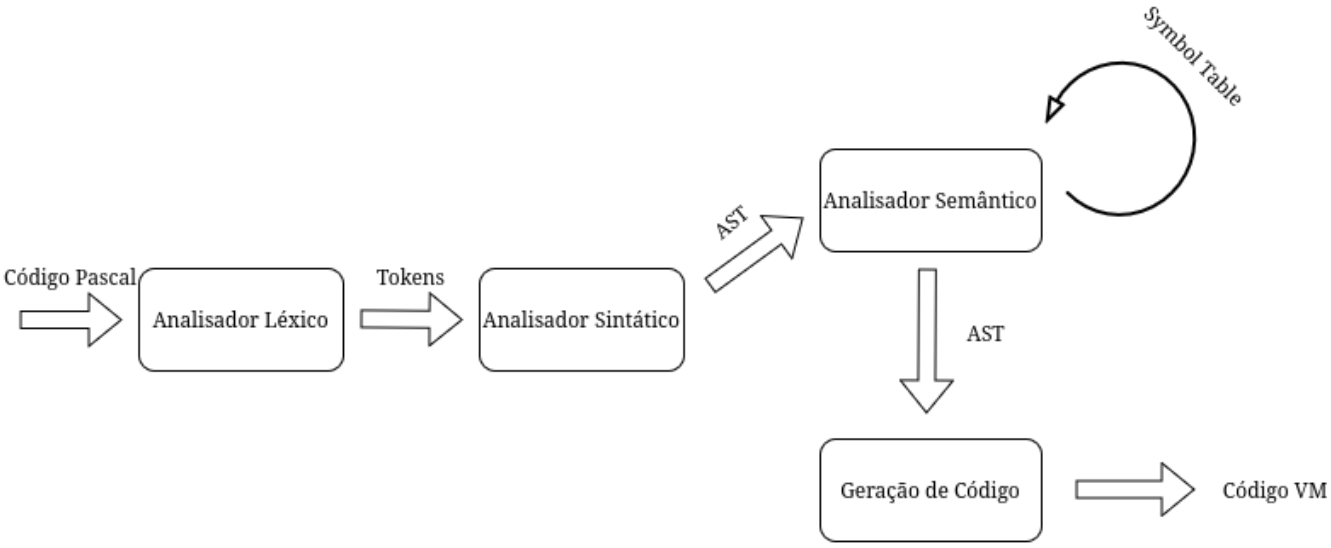


Figura 1: Arquitetura do Compilador

2.1 Análise Léxica

A análise léxica foi implementada utilizando a biblioteca `PLY` em Python, que oferece suporte para a construção de analisadores léxicos baseados em expressões regulares. O analisador reconhece os tokens fundamentais do Pascal Standard, incluindo palavras reservadas, identificadores, literais, operadores aritméticos e relacionais, delimitadores e comentários.

A definição dos tokens foi feita por meio de uma lista explícita, contemplando palavras reservadas como `BEGIN`, `END`, `IF`, `WHILE`, entre outras, além dos operadores básicos (`+`, `-`, `*`, `/`), operadores relacionais (`=`, `<>`, `>=`, etc.) e símbolos de pontuação. Os comentários foram tratados por regras específicas, sendo ignorados durante a análise e sem gerar tokens.

O analisador também gerencia a precedência e associatividade dos operadores, para uso posterior na análise sintática. Identificadores são diferenciados das palavras reservadas por meio de uma verificação interna. Os tokens literais abrangem números inteiros e reais, que são convertidos para seus tipos nativos, além de strings, capturadas sem as aspas.

Por fim, o analisador é capaz de tratar erros léxicos, reportando carateres ilegais e prosseguindo a análise para garantir a robustez do processo.

Essa implementação assegura a correta segmentação do código-fonte em unidades léxicas válidas para a etapa seguinte da compilação.

2.2 Análise Sintática

A análise sintática foi implementada utilizando a biblioteca `PLY` em Python, que permite a definição de uma gramática livre de contexto por meio de regras de produção associadas a funções. O parser constrói a árvore sintática abstrata (AST) correspondente ao programa, facilitando as etapas seguintes da compilação.

A gramática adotada é inspirada no Pascal Standard, contemplando as principais construções da linguagem, tais como declaração de programas, blocos, declarações de variáveis, funções e procedimentos, tipos, comandos compostos, controle de fluxo e expressões aritméticas e lógicas.

O parser gerado pela `PLY` utiliza a técnica *LALR(1)*, uma variação do algoritmo LR que equilibra a capacidade de reconhecimento de linguagens mais complexas com a eficiência na construção das tabelas de análise sintática. Esta abordagem permite um processamento robusto e eficiente, tratando a maioria das ambiguidades e conflitos comuns em gramáticas deste tipo, e é amplamente adotada em compiladores reais.

A seguir, apresenta-se a gramática LALR(1) utilizada para o compilador, especificando as várias produções pensadas para uma gramática para Pascal Standard :

Programa

$$
\begin{aligned}
\text{program} &\rightarrow \texttt{PROGRAM}\ \textit{header}\ \textit{block}\ . \\
\text{header} &\rightarrow \texttt{PROGRAM}\ \textit{id}\ \texttt{;} \\
&\quad \mid \texttt{PROGRAM}\ \textit{id}\ \texttt{(}\ \textit{id\_list}\ \texttt{)}\ \texttt{;} \\
\text{id\_list} &\rightarrow \textit{id} \\
&\quad \mid \textit{id\_list}\ \texttt{,}\ \textit{id} \\
\text{block} &\rightarrow \textit{declarations}\ \textit{compound\_statement}
\end{aligned}
$$

Declarações e Tipos

declarations → ϵ
| *declarations variable_declaration*
| *declarations function_declaration*
| *declarations procedure_declaration*

variable_declaration → `VAR` *variable_list* `;`

variable_list → *variable*
| *variable_list* `;` *variable*

variable → *id_list* `:` *type*

type → *simple_type*
| `ARRAY [` *number* `..` *number* `] OF` *type*

simple_type → `INTEGER`
| `REAL`
| `BOOLEAN`
| `CHAR`
| `STRING`

Parâmetros e Campos

function_declaration → `FUNCTION` *id parameter_list* `:` *type* `;` *block* `;`

procedure_declaration → `PROCEDURE` *id parameter_list* `;` *block* `;`

parameter_list → ϵ
| `(` *parameter_section_list* `)`

parameter_section_list → *parameter_section*
| *parameter_section_list* `;` *parameter_section*

parameter_section → *id_list* `:` *type*
| `VAR` *id_list* `:` *type*

field_list → *field*
| *field_list* `;` *field*

field → *id_list* `:` *type*

Comandos (Statements)

compound_statement → `BEGIN` *statement_list* `END`

statement_list → *statement*
| *statement_list* `;` *statement*

statement → ϵ
| *assignment*
| *expression*
| *compound_statement*
| *if_statement*
| *while_statement*
| *for_statement*
| *io_statement*

assignment → *id* `:=` *expression*

if_statement → `IF` *expression* `THEN` *statement*
| `IF` *expression* `THEN` *statement* `ELSE` *statement*

while_statement → `WHILE` *expression* `DO` *statement*

for_statement → `FOR` *id* `:=` *expression* `TO` *expression* `DO` *statement*
| `FOR` *id* `:=` *expression* `DOWNTO` *expression* `DO` *statement*

io_statement → `WRITE (` *expression_list* `)`
| `WRITELN (` *expression_list* `)`
| `READ (` *expression_list* `)`
| `READLN (` *expression_list* `)`

Expressões

expression → *additive_expression*
| *expression relational_operator additive_expression*

additive_expression → *multiplicative_expression*
| *additive_expression additive_operator multiplicative_expression*

multiplicative_expression → *factor*
| *multiplicative_expression multiplicative_operator factor*

expression_list → ϵ
| *expression*
| *expression_list* `,` *expression*

Fatores

$$
\begin{aligned}
\text{factor} \rightarrow\ & \textit{number} \\
& \mid \textit{string} \\
& \mid \textit{id} \\
& \mid \texttt{TRUE} \\
& \mid \texttt{FALSE} \\
& \mid \texttt{(}\ \textit{expression}\ \texttt{)} \\
& \mid \textit{factor}\ \texttt{[}\ \textit{expression}\ \texttt{]} \\
& \mid \textit{id}\ \texttt{(}\ \textit{expression\_list}\ \texttt{)} \\
& \mid \texttt{-}\ \textit{factor} \\
& \mid \texttt{NOT}\ \textit{factor}
\end{aligned}
$$

Operadores

$$
\begin{aligned}
\text{relational\_operator} &\rightarrow = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \texttt{IN} \\
\text{additive\_operator} &\rightarrow + \mid - \mid \texttt{OR} \mid \texttt{ORELSE} \\
\text{multiplicative\_operator} &\rightarrow * \mid / \mid \texttt{DIV} \mid \texttt{MOD} \mid \texttt{AND} \mid \texttt{ANDTHEN}
\end{aligned}
$$

A análise sintática também trata erros de forma a identificar e reportar situações inválidas durante o parsing, indicando o token problemático e sua posição no código-fonte dado como input, o que contribui para uma melhor experiência de identificação de erros e posterior resolução.

Essa implementação assegura a validação estrutural do código-fonte e gera a árvore sintática abstrata(AST) que será utilizada nas fases seguintes do compilador, como análise semântica e geração de código.

A construção da árvore sintática abstrata (AST) é realizada através da criação de instâncias de classes específicas, desenvolvidas para representar cada tipo de nó correspondente às diversas estruturas da linguagem Pascal Standard. Estas classes foram organizadas segundo uma hierarquia simples baseada numa superclasse comum, denominada `ASTNode`, a qual fornece a funcionalidade partilhada de armazenamento do número da linha de código (`lineno`), essencial para posterior tratamento de erros durante a fase da analise semântica.

Cada nó da AST representa uma construção concreta da linguagem e contém os atributos estritamente necessários para descrever a sua estrutura interna. Temos em baixo um exemplo de uma das classes criadas para um nodo da AST, neste caso de `ForStatement`:

```python
class ForStatement(ASTNode):
    def __init__(self, control_variable, start_expression, end_expression, statement, downto=False, lineno=None):
        """Represents a for loop."""
        super().__init__(lineno)
        self.control_variable = control_variable  # Identifier node
        self.start_expression = start_expression  # Expression node
        self.end_expression = end_expression    # Expression node
        self.statement = statement                # Statement node
        self.downto = downto                      # Boolean

    def __repr__(self):
        return f"ForStatement(var={self.control_variable}, start={self.start_expression}, end={self.end_expression}, downto={self.downto}, statement={self.statement})"
```

Figura 2: Classe do nodo da AST de ForStatement

Foram ainda definidas outras classes para representar elementos fundamentais como variáveis (`Variable`), tipos (`ArrayType`, `RecordType`), expressões (`BinaryOperation`, `UnaryOperation`, `Literal`) e estruturas de controlo de fluxo (`IfStatement`, `WhileStatement` entre outras). Cada uma destas estruturas é modelada de forma especifica a cada produção e extensível, permitindo uma manipulação clara e organizada da AST.

```pascal
program HelloWorld;
var
    sum: Integer;
begin
    writeln('Ola, Mundo!');
    sum := 2 + 3;
end.
```

Figura 3: Código Pascal a transformar para AST

De seguida podemos observar uma representação visual do código anterior quando transformado em AST.

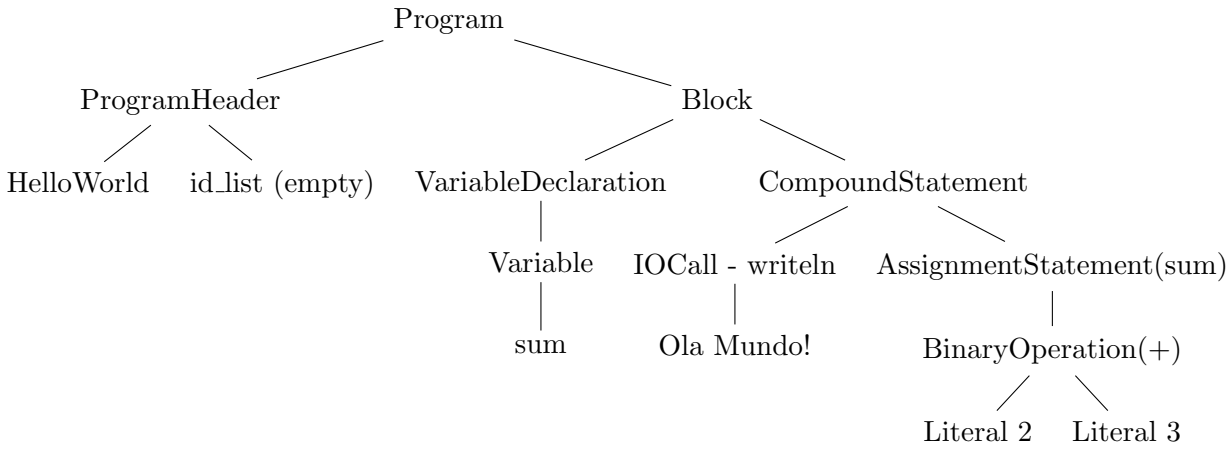


Figura 4: Árvore AST simplificada para o programa `HelloWorld`.

2.3 Análise Semântica

A análise semântica é responsável por garantir que o programa faz sentido do ponto de vista dos tipos, dos scopes e do uso dos identificadores, indo para além da verificação sintática. Nesta implementação, a análise semântica é realizada sobre a árvore sintática abstrata (AST), recorrendo a uma Tabela de Símbolos para acompanhar os identificadores declarados e os seus atributos.

2.3.1 Tabela de Símbolos

A tabela de símbolos (`SymbolTable`) armazena informações sobre variáveis, constantes, funções, procedimentos e parâmetros. Cada símbolo (`Symbol`) contém atributos como o nome, o tipo, a classe (variável, constante, função, etc.), o scope, o tipo de retorno, se é array, entre outros.

A tabela de símbolos suporta scopes aninhados (por exemplo, funções dentro de outras funções), permitindo a pesquisa recursiva de identificadores.

2.3.2 Principais Verificações Semânticas

Durante a análise semântica, são efetuadas as seguintes validações:

- **Declaração de Identificadores:** Todo identificador deve ser declarado antes de ser utilizado. Identificadores duplicados no mesmo scope originam erro.
- **Compatibilidade de Tipos:** Expressões e atribuições são verificadas para assegurar que os tipos são compatíveis (por exemplo, não é permitido atribuir uma `string` a uma variável do tipo inteiro).
- **Verificação de Parâmetros:** O número e os tipos dos argumentos nas chamadas a funções/procedimentos devem coincidir com as declarações dos parâmetros.
- **Acesso a Arrays:** Apenas arrays podem ser indexados e o índice deve ser do tipo inteiro.
- **Atribuição:** Apenas variáveis, parâmetros `VAR` e o nome da função dentro do seu próprio scope podem aparecer no lado esquerdo de uma atribuição.
- **Operações:** Operadores aritméticos, lógicos e relacionais só podem ser usados com operandos compatíveis.

2.3.3 Fluxo da Análise Semântica

A função principal `semantic_check(node, symbol_table)` percorre recursivamente a AST, realizando as validações apropriadas para cada tipo de nó (declarações, comandos, expressões, entre outros). Para cada novo scope (por exemplo, função), é criada uma nova tabela de símbolos, que fica encadeada à tabela do scope pai.
Exemplos de verificações incluem:

- Ao declarar uma variável, verificar se já existe no scope e registar o seu tipo.
- Ao encontrar uma atribuição, validar se o identificador é válido e se o tipo da expressão à direita é compatível com o do lado esquerdo.
- Ao processar chamadas a funções/procedimentos, confirmar se o nome existe, se é função ou procedimento, e se os argumentos são compatíveis.

Em baixo apresentam-se alguns exemplos de ações semânticas implementadas para a declaração de variáveis, bem como as verificações necessárias associadas:

```
elif isinstance(node, VariableDeclaration): # for variable declaration node
    var_decl_group_lineno = getattr(node, 'lineno', None)
    for var_ast_node in node.variable_list: # check each variable in the declaration
        var_decl_specific_lineno = getattr(var_ast_node, 'lineno', var_decl_group_lineno)
        line_info_decl = format_line_info(var_decl_specific_lineno)

        is_an_array_decl, symbol_type_str, symbol_element_type_str = extract_type_info_from_ast(var_ast_node.var_type, line_info_decl, "Variable")

        for var_name_original in var_ast_node.id_list: # check each identifier in the variable declaration
            var_name_lower = var_name_original.lower()
            if symbol_table.resolve(var_name_lower): # if the variable already exists
                raise Exception(f"{line_info_decl}Variable '{var_name_original}' already declared.")

            symbol = create_variable_or_param_symbol(var_name_lower, symbol_type_str, 'variable', symbol_table, is_an_array_decl, symbol_element_type_str)
            symbol_table.define(symbol)
```

Figura 5: Verificações efetuadas para *VariableDeclaration*

Como se pode observar, neste caso específico é realizada uma verificação para todas as variáveis incluídas na declaração (caso existam várias), seguida da verificação individual de cada identificador associado. Posteriormente, é criada uma instância de **Symbol** correspondente na **Symbol_Table**, de forma a registar que uma variável com aquele nome e tipo já foi declarada no respetivo escopo, prevenindo assim erros de duplicação ou conflitos de nome.

Todas as outras funções seguem a mesma estrutura, adaptando especificamente para cada caso conforme necessário.

2.3.4 Funções e Procedimentos Incorporados

Funções padrão (tais como `length`, `abs`, `sqr`, entre outras) são registadas automaticamente na tabela de símbolos global, permitindo o seu uso em qualquer parte do programa.

Esta etapa previne erros que não podem ser detetados apenas pela análise sintática, tornando o programa mais seguro e robusto antes da geração de código.

2.4 Geração de Código

Em relação à geração de código, temos um fluxo com a seguinte estrutura:

- *generator*: É chamado inicialmente com a AST principal, proveniente do analisador semântico, e percorre recursivamente os seus respetivos nós, de forma a gerar o código que será posteriormente utilizado pela máquina virtual.
- *generation_context*: Armazena o estado global da geração de código naquele momento específico. Este estado difere daquele utilizado na análise semântica, onde a tabela de símbolos serve para deteção de erros. Na geração de código, a tabela de símbolos é fundamental para determinar, por exemplo, em que *scope* deve ser emitida uma determinada instrução.
- *node_visitors*: Contém a implementação da lógica de visita a cada tipo de nó definido na AST. Existem funções distintas para visitar, por exemplo, nós do tipo *Program* e do tipo *Block*, uma vez que a geração de código e a respetiva sintaxe são diferentes para cada caso.
- *type_helpers*: Inclui funções auxiliares que apoiam a análise e inferência de tipos, sendo utilizadas pelas funções definidas em *node_visitors*.

```python
@register_visitor("IfStatement")
def visit_IfStatement(node):
    visit(node.condition)
    else_label = ctx.new_label("else")
    endif_label = ctx.new_label("endif")
    if node.else_statement:
        ctx.emit(f"JZ {else_label}", "If condition is false, jump to else")
    else:
        ctx.emit(f"JZ {endif_label}", "If condition is false (no else), jump to endif")
    visit(node.then_statement)
    if node.else_statement:
        ctx.emit(f"JUMP {endif_label}", "Skip else block")
        ctx.emit_label(else_label)
        visit(node.else_statement)
    ctx.emit_label(endif_label)

```

Figura 6: Função *visitor* para o nó da AST correspondente a um *IfStatement*

Sendo o *node_visitors* o módulo que efetivamente gera o código, detalha-se de seguida, para um caso específico, como está a ser realizada a geração de código para a máquina virtual a partir de um nó AST do tipo *IfStatement*:

Para um nó do tipo *IfStatement*, é inicialmente gerado o código correspondente à avaliação da condição. Em seguida, são criadas duas labels auxiliares: uma para marcar o início do bloco *else* (caso exista) e outra para o fim da instrução *if*. Se a condição for falsa, a execução será desviada para o bloco *else* (ou diretamente para o final, caso este não exista). Após o bloco *then*, é ainda emitido um salto para evitar a execução do bloco *else*, garantindo que apenas um dos blocos é executado. Por fim, é emitida a etiqueta que assinala o término da estrutura condicional.

Para os restantes *visitors*, a lógica aplicada é semelhante, sendo adaptada a cada tipo de nó individualmente, de forma a garantir que o código gerado é robusto e livre de erros.

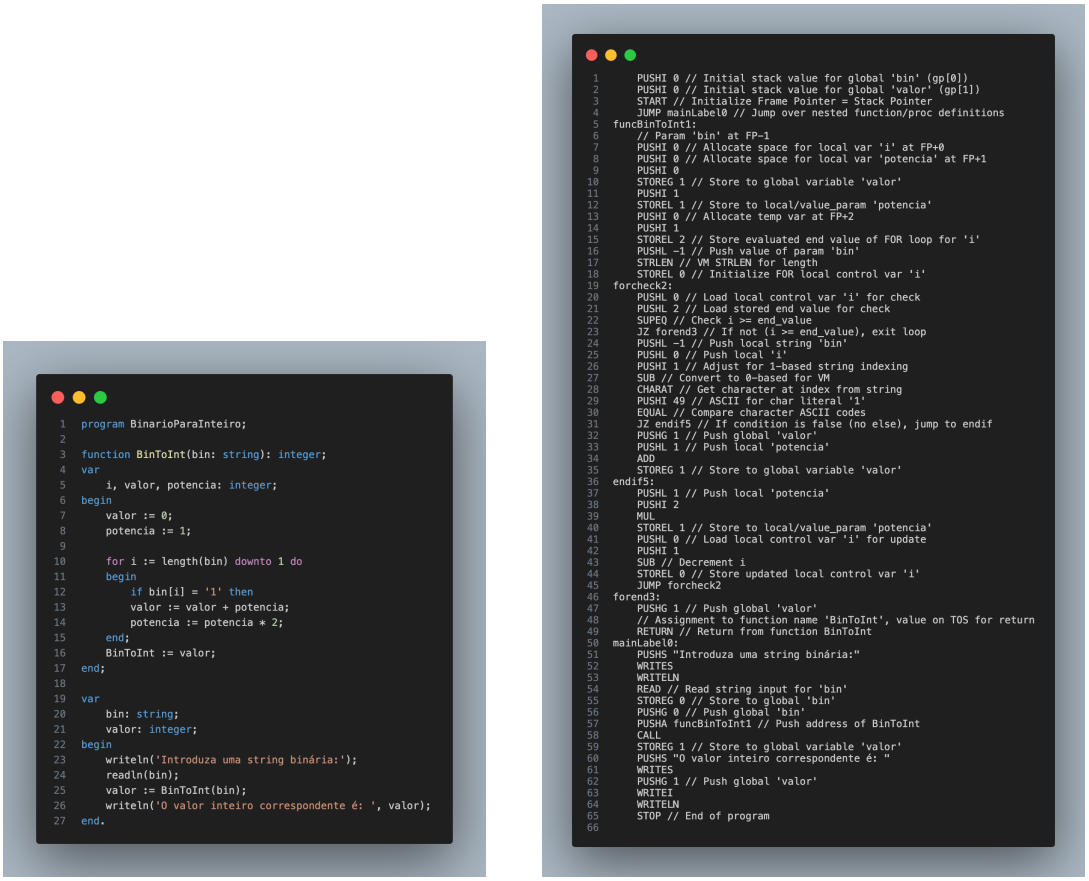
3 Testes

Para garantir o correto funcionamento do gerador de código, foram realizados vários testes com programas escritos em Pascal. Cada teste teve como objetivo validar diferentes estruturas da linguagem e confirmar que a geração de código para a máquina virtual (VM) corresponde às expectativas semânticas e sintáticas.

É de salientar que todos os exemplos apresentados no enunciado foram corretamente transcritos de Pascal Standard para código VM pelo nosso compilador. No entanto, destacamos abaixo os exemplos que considerámos mais relevantes para demonstrar o seu funcionamento, apresentando o respetivo *input* em Pascal e o *output* de código gerado.

3.1 Exemplo 7 — Conversão de Binário para Inteiro

Este exemplo implementa uma função que converte uma cadeia de carateres representando um número binário no seu valor inteiro equivalente. A função percorre a cadeia de forma inversa, somando potências de dois sempre que encontra o carater `'1'`.



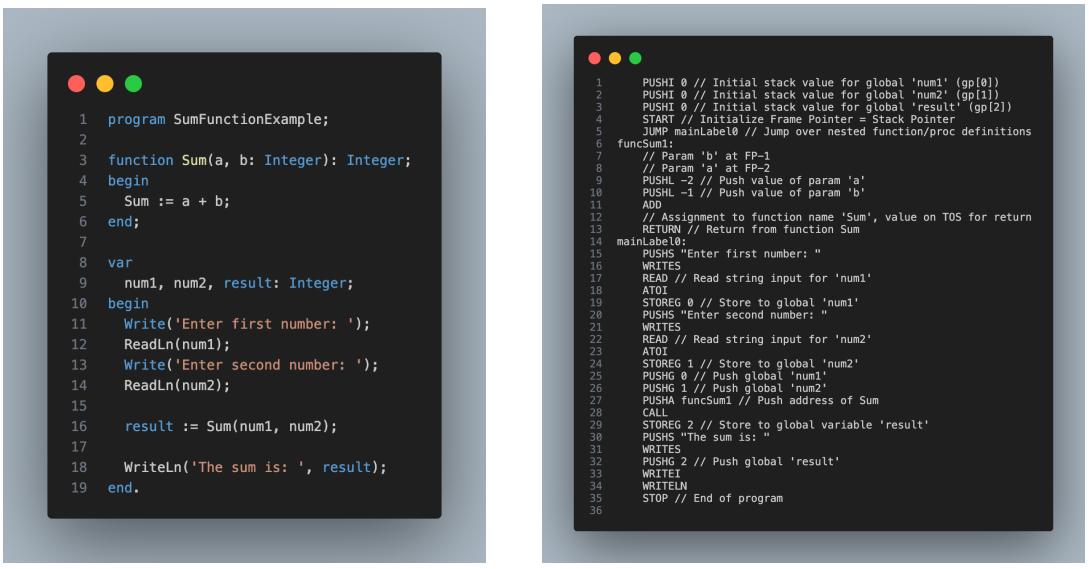
```
program BinarioParaInteiro;

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.
```

```
    PUSHI 0 // Initial stack value for global 'bin' (gp[0])
    PUSHI 0 // Initial stack value for global 'valor' (gp[1])
    START // Initialize Frame Pointer = Stack Pointer
    JUMP mainLabel0 // Jump over nested function/proc definitions
funcBinToInt1:
    // Param 'bin' at FP-1
    PUSHI 0 // Allocate space for local var 'i' at FP+0
    PUSHI 0 // Allocate space for local var 'potencia' at FP+1
    PUSHI 0
    STOREG 1 // Store to global variable 'valor'
    PUSHI 1
    STOREL 1 // Store to local/value_param 'potencia'
    PUSHI 0 // Allocate temp var at FP+2
    PUSHI 1
    STOREL 2 // Store evaluated end value of FOR loop for 'i'
    PUSHL -1 // Push value of param 'bin'
    STRLEN // VM STRLEN for length
    STOREL 0 // Initialize FOR local control var 'i'
forcheck2:
    PUSHL 0 // Load local control var 'i' for check
    PUSHL 2 // Load stored end value for check
    SUPEQ // Check i >= end_value
    JZ forend3 // If not (i >= end_value), exit loop
    PUSHL -1 // Push local string 'bin'
    PUSHL 0 // Push local 'i'
    PUSHI 1 // Adjust for 1-based string indexing
    SUB // Convert to 0-based for VM
    CHARAT // Get character at index from string
    PUSHI 49 // ASCII for char literal '1'
    EQUAL // Compare character ASCII codes
    JZ endif5 // If condition is false (no else), jump to endif
    PUSHG 1 // Push global 'valor'
    PUSHL 1 // Push local 'potencia'
    ADD
    STOREG 1 // Store to global variable 'valor'
endif5:
    PUSHL 1 // Push local 'potencia'
    PUSHI 2
    MUL
    STOREL 1 // Store to local/value_param 'potencia'
    PUSHL 0 // Load local control var 'i' for update
    PUSHI 1
    SUB // Decrement i
    STOREL 0 // Store updated local control var 'i'
    JUMP forcheck2
forend3:
    PUSHG 1 // Push global 'valor'
    // Assignment to function name 'BinToInt', value on TOS for return
    RETURN // Return from function BinToInt
mainLabel0:
    PUSHS "Introduza uma string binária:"
    WRITES
    WRITELN
    READ // Read string input for 'bin'
    STOREG 0 // Store to global 'bin'
    PUSHG 0 // Push global 'bin'
    PUSHA funcBinToInt1 // Push address of BinToInt
    CALL
    STOREG 1 // Store to global variable 'valor'
    PUSHS "O valor inteiro correspondente é: "
    WRITES
    PUSHG 1 // Push global 'valor'
    WRITEI
    WRITELN
    STOP // End of program
```

Figura 7: Código em Pascal para o Exemplo 7 e respetivo código VM gerado.

3.2 Exemplo Extra — Função de Soma

Neste exemplo é definida uma função auxiliar de soma antes do bloco principal, que é posteriormente invocada no mesmo para somar dois números introduzidos pelo utilizador através de `Readln`.



```
program SumFunctionExample;

function Sum(a, b: Integer): Integer;
begin
  Sum := a + b;
end;

var
  num1, num2, result: Integer;
begin
  Write('Enter first number: ');
  ReadLn(num1);
  Write('Enter second number: ');
  ReadLn(num2);

  result := Sum(num1, num2);

  WriteLn('The sum is: ', result);
end.
```

```
    PUSHI 0 // Initial stack value for global 'num1' (gp[0])
    PUSHI 0 // Initial stack value for global 'num2' (gp[1])
    PUSHI 0 // Initial stack value for global 'result' (gp[2])
    START // Initialize Frame Pointer = Stack Pointer
    JUMP mainLabel0 // Jump over nested function/proc definitions
funcSum1:
    // Param 'b' at FP-1
    // Param 'a' at FP-2
    PUSHL -2 // Push value of param 'a'
    PUSHL -1 // Push value of param 'b'
    ADD
    // Assignment to function name 'Sum', value on TOS for return
    RETURN // Return from function Sum
mainLabel0:
    PUSHS "Enter first number: "
    WRITES
    READ // Read string input for 'num1'
    ATOI
    STOREG 0 // Store to global 'num1'
    PUSHS "Enter second number: "
    WRITES
    READ // Read string input for 'num2'
    ATOI
    STOREG 1 // Store to global 'num2'
    PUSHG 0 // Push global 'num1'
    PUSHG 1 // Push global 'num2'
    PUSHA funcSum1 // Push address of Sum
    CALL
    STOREG 2 // Store to global variable 'result'
    PUSHS "The sum is: "
    WRITES
    PUSHG 2 // Push global 'result'
    WRITEI
    WRITELN
    STOP // End of program
```

Figura 8: Código em Pascal de uma função de soma e respetivo código VM gerado.

4 Otimizações e Extras

Durante a fase de geração de código, foram implementadas diversas otimizações com o objetivo de tornar o código gerado mais eficiente e compacto. Abaixo descrevem-se as otimizações consideradas mais relevantes neste contexto.

4.1 Otimização do Length

Sempre que uma função como `LENGTH` é aplicada a um literal *string*, o seu valor é calculado em tempo de compilação, evitando assim qualquer instrução adicional na VM em tempo de execução. Por exemplo, a expressão `LENGTH('Pascal')` origina diretamente a instrução que coloca na stack o valor `6`. Esta otimização está presente no *visitor* para `FunctionCall` e evita cálculos desnecessários durante a execução.

4.2 Eliminação de Operações Neutras

Operações como o operador unário `+` (ex.: `+x`) são reconhecidas como neutras e, por isso, não originam qualquer instrução na VM. Esta decisão permite reduzir o número de instruções geradas, sem afetar o comportamento semântico do programa. A otimização é aplicada no `visitor` de `UnaryOperation`.

4.3 Instruções Específicas por Tipo

O gerador de código escolhe dinamicamente instruções diferentes consoante o tipo dos operandos. Por exemplo, utiliza-se `ADD` para inteiros e `FADD` para reais, `DIV` para divisão inteira e `FDIV` para divisão real. Este comportamento evita verificações de tipo em tempo de execução e torna o código mais eficiente. Esta lógica é implementada no `visit_BinaryOperation`, onde se analisa o tipo de cada operando antes de emitir a instrução correspondente.

4.4 Pré-Cálculo dos Limites de Ciclos

Nos ciclos `FOR`, a expressão que define o valor final do contador é avaliada apenas uma vez antes de iniciar o ciclo e armazenada numa variável temporária. Com isto, evita-se a reavaliação de expressões potencialmente complexas a cada iteração. Esta abordagem pode ser observada na implementação de geração de código, onde é efetuado um `STOREL` antes de cada verificação de condição.

4.5 Retorno Eficiente de Funções

Em Pascal, o valor de retorno de uma função é atribuído ao identificador com o nome da função. O gerador de código trata esta atribuição de forma eficiente, deixando o valor no topo da stack, pronto a ser utilizado pela instrução `RETURN`, sem necessidade de `PUSH`'s adicionais. Esta otimização é visível tanto na visita a atribuições (`visit_AssignmentStatement`) como na definição de funções (`visit_FunctionDeclaration`).

Estas otimizações, embora simples, contribuem significativamente para a qualidade e desempenho do código gerado, demonstrando uma preocupação com a eficiência desde a fase de compilação.

4.6 Extras

Para além das funcionalidades base, foram também implementadas algumas funções *builtin* comuns na linguagem Pascal Standard, de forma a enriquecer a experiência de utilização e a

permitir um conjunto mais alargado de operações em tempo de execução. Entre as funções integradas encontram-se, por exemplo, `Length`, `Sqr`, `Abs`, entre outras. Estas funções foram devidamente reconhecidas e tratadas na fase de geração de código, sendo traduzidas diretamente em instruções apropriadas para a VM. Todas estas funcionalidades encontram-se operacionais e foram validadas com casos de teste adequados, permitindo ao utilizador recorrer a operações matemáticas e de manipulação de *strings* de forma simples e eficaz.

5 Dificuldades na Implementação

Durante o desenvolvimento do compilador, deparámo-nos com algumas dificuldades que exigiram análise e reformulação cuidadosa de certas partes do projeto.

- A principal dificuldade encontrada durante a construção da análise sintática foi garantir a consistência da gramática, evitando conflitos do tipo *shift-reduce*, comuns em gramáticas ambíguas. Um caso particularmente desafiante foi a definição da estrutura `if-then-else`, cuja ambiguidade levou à geração de conflitos internos na ferramenta `PLY`. A resolução deste problema exigiu uma reformulação atenta das regras de produção, e principalmente do controlo de precedência, o que implicou vários testes até realmente ficar consistente.

- Outra dificuldade relevante surgiu na fase de geração de código para a máquina virtual. A complexidade desta etapa reside no facto de existir muitas abordagens possíveis, o que gerou alguma indecisão inicial sobre a melhor forma de estruturar o processo. Além disso, foi necessário garantir que o código gerado fosse robusto, eficiente e corretamente alinhado com os requisitos semânticos da linguagem, o que implicou um trabalho detalhado na definição dos *visitors* e no controlo do contexto de geração.

Apesar destes desafios, consideramos que a abordagem adotada nos permitiu adquirir uma compreensão mais aprofundada dos mecanismos internos de um compilador completo para uma linguagem complexa como o Pascal.

6 Conclusão

A implementação deste compilador para uma linguagem baseada no Pascal Standard permitiu-nos consolidar conhecimentos essenciais lecionados nas aulas desta UC, sobre os vários componentes envolvidos no processo de compilação, desde a análise léxica, sintática e semântica, até à geração de código a partir de representações intermédias.

Em suma, este trabalho permitiu-nos aprofundar a compreensão teórica e prática do funcionamento de um compilador, reforçando a nossa capacidade de analisar, planear e implementar soluções robustas para problemas complexos.