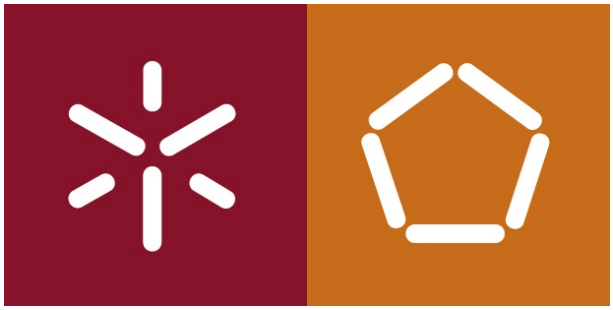

Processamento de Linguagens: Relatório Técnico

Grupo 53

Bruno Campos (A98639)

Fábio Leite (A100902)

Paulo Cruz (A80949)

June 1, 2025

Índice

Introdução

Este relatório é uma ferramenta complementar ao projeto proposto da unidade curricular de Processamento de Linguagens.

O objetivo deste projeto é o desenvolvimento de um compilador para a linguagem Pascal standard.

Este compilador deve ser capaz de analisar e traduzir código Pascal para um formato intermédio, passando deste último para código máquina, ou, caso possível, diretamente para código máquina.

Ao longo do relatório vamos abordar as diferentes etapas que realizamos de modo a cumprir com o que foi proposto, tal como o analisador léxico, sintático e semântico.

Lexer

Para a análise léxica foi-nos proposto a implementação de um analisador léxico para converter código Pascal numa lista de tokens, bem como usar a libraria ply, nomeadamente a ferramenta **ply.lex** para esta implementação e também a identificação de palavras-chave, identificadores, números, operadores e símbolos especiais.

2.1 Implementação do Lexer

Tal como foi dito anteriormente, o lexer do projeto foi implementado utilizando a biblioteca PLY que permite a definição de regras léxicas através de expressões regulares. O objetivo do lexer é transformar o código fonte Pascal numa sequência de tokens que serão posteriormente analisados pelo parser.

2.1.1 Palavras Reservadas e Tokens

No início do ficheiro, é definido umas regras para a deteção de comentários, sendo ,logo de seguida, apresentado um dicionário `reserved` que associa as palavras reservadas da linguagem Pascal aos respetivos nomes de tokens. Isto permite ao lexer distinguir facilmente entre identificadores normais e palavras-chave da linguagem.

Temos também uma lista `tokens` que inclui todos os tipos de tokens reconhecidos pelo lexer, tal como:

1. Números Reais (NUM_REAL).
2. Números inteiros (NUM).
3. Strings (STR).
4. Identificadores (VARNAME).
5. Operadores de Atribuição (ATRIB).
6. Operadores Relacionais (EQUALS, NE, LT, LE, GT, GE)

Bem como todos os outros tokens correspondentes a palvaras reservadas definidas anteriormente.

Além disso, a lista `literals` define os caracteres individuais que são tratados como tokens literais, como operadores aritméticos (+, -, ...), parênteses, ponto e vírgula, entre outros.

2.1.2 Regras Léxicas

Cada token é definido por uma função Python cujo nome começa por t_ seguido do nome do token para fácil interpretação. Estas funções utilizam expressões regulares para identificar padrões no texto de entrada seguindo a mesma estrutura:

```
def t_Token(t):
    r'RE equivalente'
    t.type = 'Token'
    return t
```

1. **Operadores de atribuição e relacionais**: São definidos tokens para :=, =, <>, <=, >=, <, >, cada um com a sua expressão regular.
2. **Números**:
   (a) t_NUM_REAL reconhece números reais (com ponto decimal).
   (b) t_NUM reconhece números inteiros.
3. **Strings**: O token t_STRING reconhece sequências de caracteres entre aspas.
4. **Identificadores e Palavras Reservadas**: O token t_VARNAME reconhece identificadores válidos em Pascal. Caso o identificador corresponda a uma palavra reservada, o tipo do token é ajustado para o valor correto usando o dicionário reserved.
5. **Novas linhas**: A função t_newline atualiza o número da linha do lexer sempre que encontra um ou mais caracteres de nova linha.
6. **Espaços e tabs**: São ignorados pelo lexer através da variável t_ignore.
7. **Tratamento de erros**: A função t_error é chamada sempre que um caracte inválido é encontrado, imprimindo uma mensagem de erro com o caractere e a linha correspondente, avançando o lexer para o próximo caractere.

2.1.3 Inicialização

No final do ficheiro, o lexer é criado com lex.lex() e o número da linha é inicializado a 1 de modo a começar logo na primeira linha.

Parser

O parser foi implementado em Python, recorrendo à biblioteca `PLY` (Python Lex-Yacc), que disponibiliza funcionalidades semelhantes às do `lex` e `yacc` em ambiente C.

A gramática baseia-se fortemente na utilizada pelo compilador Free Pascal, mas ajustada para um subconjunto que exclui *functions* e *procedures*, bem como alguns recursos avançados (por exemplo, *downto*, *case*, *repeat*, *const*, etc.). O parser aceita declarações de variáveis, estruturas de controlo (`if/then/else`, `for/to`, `while`), leitores/escritores (`readln`, `write`, `writeln`), literais de números inteiros e reais, strings e arrays unidimensionais.

Este documento organiza-se da seguinte forma: na Secção 3.1 definem-se os objetivos específicos do parser; na Secção 3.2 apresenta-se a gramática adoptada em notação BNF; na Secção 3.3 detalha-se a implementação em PLY, explicando cada componente; na Secção 3.4 discute-se a abordagem às verificações semânticas básicas; a Secção 3.5 descreve o tratamento de erros sintáticos; na Secção **??** apontam-se limitações atuais e, por fim, na Secção **??** apresentam-se conclusões e perspetivas para trabalho futuro.

3.1 Objetivos do Parser

O parser `pascal_sin.py` cumpre as seguintes metas:

1. **Definir a gramática de um subconjunto de Pascal**: Formalizar as regras sintáticas que abrangem declarações, expressões aritméticas e relacionais, controlo de fluxo, declarações de arrays e comandos de I/O.

2. **Implementar a análise sintática em PLY**: Traduzir as regras gramaticais para funções `p_<nome_produção>` adequadas, definindo precedências e associatividades quando necessário.

3. **Construir uma Árvore de Sintaxe Abstrata (AST)**: Cada produção deve gerar um nó AST (geralmente representado por tuplos Python) que capture a estrutura do programa, permitindo a travessia posterior pelo `translator.py`.

4. **Efetuar verificações semânticas básicas durante o parsing**:

    - Assegurar que variáveis são declaradas antes de serem usadas (*undeclared variable*).

- Verificar, caso o índice seja literal, se o acesso a posições de array respeita os limites definidos na declaração.

5. **Gerir o problema do dangling else**: Implementar as produções `MatchedStatement` e `UnmatchedStatement` para garantir a correta associação dos comandos `else`.

6. **Acumular e reportar erros sintáticos**: Não interromper a análise logo ao primeiro erro, mas armazenar mensagens numa lista `syntax_errors`, indicando linha e coluna do erro.

3.2 Gramática Definida

Nesta secção apresenta-se a gramática formal adoptada para o parser, numa notação próxima de BNF (Backus–Naur Form). As produções encontram-se agrupadas por categorias: definição do programa, declarações, tipos, expressões e statements.

3.2.1 Produção Inicial: Programa

```
Program ::= PROGRAM VARNAME ';' Code '.'
```

Comentários:

- `PROGRAM` e identificador são sensíveis a palavras reservadas; `VARNAME` é reconhecido em `pascal_lex.py`.
- A produção obriga a terminação com `'.'` (ponto final).

3.2.2 Declarações e Bloco Principal

```
Code ::= Declarations CompoundStatement


Declarations ::=                \# Declarações opcionais
               | VAR VariableList


VariableList ::= VariableDeclaration
               | VariableList VariableDeclaration


VariableDeclaration ::= IdentifierList ':' DataType ';'
```

```
IdentifierList ::= VARNAME
                 | IdentifierList ',' VARNAME

DataType ::= INTEGER
           | REAL
           | BOOLEAN
           | STRING
           | ARRAY '[' NUM '..' NUM ']' OF DataType
```

Comentários:

- O bloco `Declarations` pode ser vazio (produção implícita `Declarations ::=`).
- Em `DataType`, o tipo `ARRAY` aceita apenas índices literais inteiros (`NUM` corresponde a inteiro).
- As produções definem arrays unidimensionais com limite inferior e superior provenientes de literais.
- `IdentifierList` permite declarações do tipo `a, b, c: integer;`.

3.2.3 Statements e Dangling Else

Para evitar ambiguidades no emparelhamento de `else`, foram definidas duas categorias de statements: `MatchedStatement` (com else completamente emparelhado) e `UnmatchedStatement` (if sem else ou else emparelhado parcialmente).

```
Statement ::= MatchedStatement
            | UnmatchedStatement

MatchedStatement ::= IF Exp THEN MatchedStatement ELSE MatchedStatement
                   | NonIfStatement

UnmatchedStatement ::= IF Exp THEN MatchedStatement ELSE UnmatchedStatement
                     | IF Exp THEN Statement

NonIfStatement ::= CompoundStatement
```

```
                | RepetitiveStatement
                | SingleStatement
```

Comentários:

- `MatchedStatement` reconhece `if` com `else` ou qualquer outro statement que não comece por `if`.
- `UnmatchedStatement` cobre dois casos:
  - `if ... then <MatchedStatement> else <UnmatchedStatement>` (caso em que o `else` continua a procurar correspondência mais interna)
  - `if ... then <Statement>` (caso sem `else`).

3.2.4 Statements Simples e Compostos

```
CompoundStatement ::= BEGIN StatementList END OptionalSemicolon

StatementList ::= Statement
                | StatementList Statement

OptionalSemicolon ::= ';'
                    |

RepetitiveStatement ::= FOR VARNAME ATRIB Exp TO Exp DO Statement
                      | WHILE Exp DO Statement

SingleStatement ::= VARNAME ATRIB Exp ';'
                  | WRITELN '(' ArgumentList ')' OptionalSemicolon
                  | WRITELN OptionalSemicolon
                  | WRITE '(' ArgumentList ')' OptionalSemicolon
                  | READLN '(' VARNAME ')' ';'
                  | READLN '(' VARNAME '[' Exp ']' ')' ';'
```

Comentários:

- `CompoundStatement` agrupa statements entre `BEGIN` e `END`, opcionalmente seguidos de `';'` antes do `END`.
- Em `SingleStatement`, o símbolo `ATRIB` corresponde a `":="`.
- As produções de `READLN` aceitam leitura de variáveis simples ou de elementos de um array.
- `WRITELN` pode ser usado sem argumentos.

3.2.5 Argumentos de I/O (WRITE/Writeln)

```
ArgumentList ::= Argument
               | ArgumentList ',' Argument

Argument ::= STR
           | Exp
           | Exp ':' FORMAT

FORMAT ::= NUM
         | NUM ':' NUM
```

Comentários:

- `STR` corresponde a literais string entre apóstrofos (`'texto'`).
- A cláusula `FORMAT` aceita formato padrão de Pascal, por exemplo `5` (largura) ou `5:2` (largura e precisão). Atualmente, o parser regista o valor, mas a implementação no `translator.py` não efetua formatação detalhada.

3.2.6 Expressões Aritméticas e Relacionais

```
Exp ::= SimpleExpression RelOp SimpleExpression
      | SimpleExpression

RelOp ::= EQUALS    # '='
        | NE        # '<>'
        | LT        # '<'
```

```
        | LE        # '<='
        | GT        # '>'
        | GE        # '>='
        | AND
        | OR

SimpleExpression ::= '+' AdditiveExpression
                   | '-' AdditiveExpression %prec UMINUS
                   | AdditiveExpression

AdditiveExpression ::= AdditiveExpression '+' Term
                     | AdditiveExpression '-' Term
                     | Term

Term ::= Term '*' Factor
       | Term '/' Factor
       | Term '%' Factor
       | Factor

Factor ::= NUM
         | NUM_REAL
         | STRING
         | VARNAME
         | '(' Exp ')'
         | VARNAME '[' Exp ']'
         | NOT Factor %prec NOT
```

Comentários:

- Os literais `NUM` (inteiros), `NUM_REAL` (reais) e `STRING` são reconhecidos pelo *lexer*.
- Acesso a arrays via `VARNAME '[' Exp ']'`. Caso o índice seja literal, verifica-se limite durante o parsing.
- `NOT`, `AND` e `OR` são tratados como operadores lógicos, com precedência ajustada no bloco `precedence`.

- A precedência define que operações aritméticas de multiplicação têm prioridade sobre adição/subtração, e que `UMINUS` e `NOT` têm precedência maior (unários).

3.2.7 Precedência e Associatividade

No início do ficheiro `pascal_sin.py` define-se o tuple `precedence` para resolver ambiguidades:

```
precedence = (
    ('nonassoc', 'EQUALS', 'NE', 'LT', 'LE', 'GT', 'GE', 'AND', 'OR'),
    ('left', '+', '-'),
    ('left', '*', '/', '%'),
    ('right', 'UMINUS'),
    ('right', 'NOT'),
)
```

Comentários:

- Operadores relacionais e lógicos (`AND`, `OR`) são não-associativos (`nonassoc`).
- Soma e subtração (+, −) e multiplicação/divisão/módulo (*, /, %) têm associatividade à esquerda (`left`).
- Operadores unários − (negativo) e `NOT` são tratados com precedência maior, associativos à direita (`right`).

3.3 Implementação em PLY

Esta secção descreve os principais aspetos da implementação do parser usando PLY, abordando a declaração de tokens (importados de `pascal_lex.py`), definição das funções de produção, construção da AST e integração com o lexer.

3.3.1 Declaração de Tokens e Literais

No início de `pascal_sin.py`, importa-se:

```
import ply.yacc as yacc
from pascal_lex import tokens, literals, lexer, precedence
```

`Tokens` e `literals` são listas definidas em `pascal_lex.py`, que incluem todos os símbolos terminais necessários (palavras reservadas, operadores e pontuação). O objeto `lexer` é usado para fornecer tokens ao parser.

3.3.2 Construção da AST

Cada função de produção em PLY segue o padrão:

```
def p_NomeDaProducao(p):
    "Producao : simbolo1 simbolo2 ..."
    # Aceder a p[1], p[2], ...
    p[0] = <estrutura de nó AST>
```

O parser constrói a AST usando tuples Python. A convenção adotada é:

- O primeiro elemento do tuplo identifica o tipo de nó (por exemplo, `'program'`, `'decl'`, `'assign'`, `'if'`, `'for'`, `'array_access'` etc.).
- Os elementos subsequentes contêm os campos relevantes do nó (nomes de variáveis, expressões filhas, listas de declarações, etc.).

3.3.3 Produção Inicial: `Program`

```
def p_Program(p):
    "Program : PROGRAM VARNAME ';' Code '.'"
    p[0] = ('program', p[2], p[4])
```

Descrição:

- Verifica a sintaxe obrigatória: a palavra reservada `PROGRAM`, seguida de um identificador (`VARNAME`), ponto-e-vírgula, bloco `Code` e ponto final.
- A atribuição `p[0] = ('program', p[2], p[4])` cria o nó de raiz da AST, com etiqueta `'program'`, armazenando o nome do programa (`p[2]`) e o subnó `Code`.

3.3.4 Declarações de Variáveis (`Declarations`)

```
def p_Declarations_empty(p):
    "Declarations :"
    p[0] = None


def p_Declarations_var(p):
    "Declarations : VAR VariableList"
    p[0] = ('var_decls', p[2])
```

- Quando não existem declarações (`Declarations :`), define-se `p[0] = None`.
- Se existir a palavra reservada `VAR`, a produção `VariableList` gera uma lista de declarações individuais. O nó `('var_decls', p[2])` armazena essa lista.

3.3.5 Declaração de Tipos e Variáveis

```
def p_VariableDeclaration(p):
    "VariableDeclaration : IdentifierList ':' DataType ';'"
    for var in p[1]:
        if p[3] == "integer":
            dic[var] = (0, "integer")
        if p[3] == "boolean":
            dic[var] = ('true', "boolean")
        if p[3] == "string":
            dic[var] = ('', "string")
        if p[3] == "real":
            dic[var] = (0.0, "real")
        elif isinstance(p[3], tuple) and p[3][0] == 'array':
            low, high = p[3][1]
            size = high - low + 1
            dic[var] = ([0] * size, p[3])
        else:
            dic[var] = (None, p[3])
```

```
    p[0] = ('decl', p[1], p[3])
```

Explicação:

- Recebe `IdentifierList` (lista de nomes), `DataType` (tipo de dados) e inicializa o dicionário global `dic` com valores padrão:
  - Inteiros iniciados a `0`, reais a `0.0`, booleanos a `'true'` (string), strings a `''`.
  - Arrays: calcula tamanho (em `size`) e aloca lista de zeros.
- O nó AST `('decl', [vars], tipo)` armazena a lista de variáveis e o respectivo tipo.
- A existência de `dic` aqui permite verificações semânticas posteriores (por exemplo, verificação de existência na atribuição e limites de array).

3.3.6 Expressões e Operadores

3.3.6.1 Expressões Relacionais e Lógicas

```
def p_Expression_relop(p):
    "Exp : SimpleExpression RelOp SimpleExpression"
    p[0] = ('rel', p[2], p[1], p[3])

def p_Expression_and(p):
    "Exp : Exp AND Exp"
    p[0] = ('and', p[1], p[3])

def p_Expression_or(p):
    "Exp : Exp OR Exp"
    p[0] = ('or', p[1], p[3])

def p_Expression_simple(p):
    "Exp : SimpleExpression"
    p[0] = p[1]

def p_RelOp(p):
```

```
    """RelOp : EQUALS
             | NE
             | LT
             | LE
             | GT
             | GE
    """
    p[0] = p[1]
```

- Exp pode corresponder a expressões comparativas: `SimpleExpression RelOp SimpleExpression`, devolvendo nó `('rel', operador, esquerda, direita)`.
- `RelOp` inclui operadores relacionais (`=, <>, <, <=, >, >=`) e lógicos (`AND, OR`). Embora em Pascal `AND/OR` sejam operadores booleanos, aqui enquadram-se no mesmo nível dos relacionais.
- Exemplo: para `a < b`, gera-se `('rel', '<', ('var','a'), ('var','b'))`.

3.3.6.2 Expressões Aritméticas

```
def p_SimpleExpression_sign(p):
    "SimpleExpression : '+' AdditiveExpression"
    p[0] = p[2]

def p_SimpleExpression_sign_neg(p):
    "SimpleExpression : '-' AdditiveExpression %prec UMINUS"
    p[0] = ('uminus', p[2])

def p_SimpleExpression(p):
    "SimpleExpression : AdditiveExpression"
    p[0] = p[1]

def p_AdditiveExpression_plus(p):
    "AdditiveExpression : AdditiveExpression '+' Term"
```

```
    p[0] = ('+', p[1], p[3])

def p_AdditiveExpression_minus(p):
    "AdditiveExpression : AdditiveExpression '-' Term"
    p[0] = ('-', p[1], p[3])

def p_AdditiveExpression_term(p):
    "AdditiveExpression : Term"
    p[0] = p[1]

def p_Term_mul(p):
    "Term : Term '*' Factor"
    p[0] = ('*', p[1], p[3])

def p_Term_div(p):
    "Term : Term '/' Factor"
    p[0] = ('/', p[1], p[3])

def p_Term_mod(p):
    "Term : Term '%' Factor"
    p[0] = ('%', p[1], p[3])

def p_Term_factor(p):
    "Term : Factor"
    p[0] = p[1]
```

- A gramática segue a hierarquia de precedência: multiplicação/divisão/mod tem precedência sobre adição/subtração.
- O operador unário negativo – (precedência `UMINUS`) gera nó `('uminus', expr)`.
- Operadores binários produzem nós `('+', esquerda, direita)`, `('-', esquerda, direita)`, `('*', esquerda, direita)`, etc.

3.3.6.3 Fatores

```
def p_Factor_num(p):
    "Factor : NUM"
    p[0] = p[1]

def p_Factor_real(p):
    "Factor : NUM_REAL"
    p[0] = p[1]

def p_Factor_string(p):
    "Factor : STRING"
    p[0] = p[1]

def p_Factor_true(p):
    "Factor : TRUE"
    p[0] = 1

def p_Factor_false(p):
    "Factor : FALSE"
    p[0] = 0

def p_Factor_var(p):
    "Factor : VARNAME"
    p[0] = ('var', p[1])

def p_Factor_paren(p):
    "Factor : '(' Exp ')'"
    p[0] = p[2]

def p_Factor_array(p):
    "Factor : VARNAME '[' Exp ']'"
    # Verificação de limites de array em tempo de parsing, se índice literal.
```

```
    p[0] = ('array_access', p[1], p[3])

def p_Factor_not(p):
    "Factor : NOT Factor %prec NOT"
    p[0] = ('not', p[2])
```

- Literais numéricos (`NUM`, `NUM_REAL`) e strings (`STRING`) propagam o valor Python (int, float ou str).
- Variável gera nó `('var', nome_variável)`.
- Acesso a arrays (por ex., `arr[5]`) produz nó `('array_access', nome_array, índice)`. Caso o índice seja constante, executa-se verificação de limites, fazendo o ajuste do `parser.success = False` caso falhd.
- `NOT` gera nó `('not', expr)`.

3.3.7 Exemplo de Produção Completa

Para ilustrar, segue a produção completa de um statement `if-then-else` totalmente emparelhado:

```
def p_MatchedStatement_if(p):
    "MatchedStatement : IF Exp THEN MatchedStatement ELSE MatchedStatement"
    p[0] = ('if', p[2], ('then', p[4]), ('else', p[6]))
```

Explicação do AST gerado:

- `('if', condição, ('then', nó_then), ('else', nó_else))`.
- `p[2]` é a expressão lógica ou relacional.
- `p[4]` e `p[6]` são nós AST recursivos para os ramos `then` e `else`.

3.3.8 Tratamento do Dangling Else

O *dangling else* é resolvido pelas produções:

```
MatchedStatement ::= IF Exp THEN MatchedStatement ELSE MatchedStatement
                   | NonIfStatement

UnmatchedStatement ::= IF Exp THEN MatchedStatement ELSE UnmatchedStatement
                     | IF Exp THEN Statement
```

Desta forma, assegura-se que cada `else` corresponde ao `if` mais próximo sem `else`, evitando ambiguidade. A inclusão de `MatchedStatement` como não-terminal no `UnmatchedStatement` garante que blocos `then` completos podem conter outros `if/else` devidamente emparelhados antes de encontrar `else` externo.

3.3.9 Verificação de Erros Sintáticos

A função de tratamento de errosé definida como:

```
def p_error(p):
    if p:
        line = p.lexer.lexdata[:p.lexpos].count('\n') + 1
        col = find_column(p.lexer.lexdata, p)
        syntax_errors.append(
            f"Syntax error at line {line}, column {col}: unexpected token '{p.value}'
        )
    else:
        syntax_errors.append("Syntax error: unexpected end of input!")
```

- Obtém a linha e coluna aproximada em que o token inesperado ocorreu, usando `lexdata` e `lexpos`.
- Se `p` for `None`, significa fim de ficheiro inesperado.
- As mensagens são acumuladas em `syntax_errors`, sem interromper imediatamente o parsing.

3.4 Verificações Semânticas Básicas

Embora o parser tenha como missão principal verificar a conformidade sintática, foram incorporadas algumas verificações semânticas simples:

3.4.1 Existência de Variáveis

Na produção de atribuição:

```
def p_SingleStatement_assign(p):
    "SingleStatement : VARNAME ATRIB Exp OptionalSemicolon"
    if p[1] in dic:
        var_type = dic[p[1]][1]
        dic[p[1]] = (p[3], var_type)
    else:
        print(f"Error: variable {p[1]} not declared")
        dic[p[1]] = (p[3], None)
    p[0] = ('assign', p[1], p[3])
```

- Se `VARNAME` não estiver em `dic`, imprime mensagem de erro de variável não declarada e regista `(valor, None)` no `dic`. Isto evita que a tradução prossiga silenciosamente num programa que usa variáveis indefinidas.
- Caso exista, atualiza temporariamente o valor da variável em `dic` (útil para verificar, por exemplo, índices literais de arrays ou formatação de `write`).

3.4.2 Verificação de Limites de Arrays

Na produção de acesso a elemento de array:

```
def p_Factor_array(p):
    "Factor : VARNAME '[' Exp ']'"
    limit = dic[p[1]][1][1]
    a, b = limit
    # Caso p[3] seja inteiro literal
    if isinstance(p[3], int):
        value = p[3]
    elif isinstance(p[3][1], int) or isinstance(p[3][1], float):
        value = -p[3][1]
    else:
```

```
        value = p[3]
    if isinstance(value, int) and not value in range(a, b + 1):
        print("Warning: range check error ...")
        parser.success = False
    p[0] = ('array_access', p[1], p[3])
```

- Obtém o tuplo `limit = (lower, upper)` da declaração do array (armazenada em `dic`).
- Se o índice for literal (inteiro), verifica se `lower índice upper`. Caso contrário, gera um aviso e define `parser.success = False`, impedindo tradução posterior.
- Se o índice for expressão mais complexa, tenta extrair valor de `p[3] [1]`, mas esta abordagem só funciona em certos casos de expressões unárias. Para expressões arbitrárias, não há verificação estática.
- O nó AST resultante é `('array_access', nome_array, Exp)`.

3.4.3 Verificação em `readln` de Array

De forma análoga, em `p_SingleStatement_readln_array`, verifica-se índice constante em tempo de parsing, havendo um report `parser.success = False` casjo esteja fora dos limites.

3.5 Tratamento de Erros Sintáticos

As principais estratégias adotadas para detetar e reportar erros sintáticos são:

- **Registar em lista**: Todas as mensagens de erro acumulam-se em `syntax_errors`. Ao final do parsing, se a lista não estiver vazia, imprime-se cada erro ao utilizador e aborta-se a tradução.
- **Linha e coluna aproximadas**: A função auxiliar `find_column` (não exibida aqui) calcula a coluna exata do token inválido, baseada no número de caracteres desde o início da linha corrente.
- **Não interrupção imediata**: Ao contrário de "fail-fast", o PLY tenta fazer *error recovery* básico (ignorando tokens inválidos). O parser armazena cada ocorrência para posterior análise, apresentando ao final todos os erros detetados numa única execução.

3.6 Limitações e Trabalhos Futuros

3.6.1 Verificação de Índices de Arrays para Expressões Complexas

Atualmente, a checagem estática de índices apenas funciona se o índice for literal ou expressão unária simples. Para expressões como `arr[i+1]`, não se verifica o valor em tempo de parsing. Possíveis melhorias:

- Implementar um pequeno interpretador estático que avalie expressões compostas se as variáveis envolvidas tiverem valores constantes conhecidos em tempo de compilação.
- Caso não seja possível análise estática, adotar verificação dinâmica no código traduzido (inserir instruções EWVM que validem índice em tempo de execução, gerando erro ou terminando programa caso fora do limite).

3.6.2 Precedência de `AND` e `OR`

Na gramática, `AND` e `OR` foram colocados no mesmo nível de não-associativo que operadores relacionais. Em Pascal standard, `AND` e `OR` possuem precedência inferior às comparações. Caso se deseje maior conformidade semântica:

- Ajustar a tabela `precedence` para:

```
precedence = (
    ('nonassoc', 'EQUALS', 'NE', 'LT', 'LE', 'GT', 'GE'),
    ('left', 'AND', 'OR'),
    ('left', '+', '-'),
    ('left', '*', '/', '%'),
    ('right', 'UMINUS'),
    ('right', 'NOT'),
)
```

- Rever as produções que encaixam `AND`/`OR` em `RelOp`.

3.6.3 Integração de Tabelas de Símbolos Locais

Atualmente, a tabela de símbolos (`dic`) é global, o que impede suporte a scopes internos (por ex., variáveis locais em `for` ou em subprogramas). Futuras versões podem:

- Substituir `dic` por classe `SymbolTable` hierárquica, mantendo pilha de escopos.
- Garantir que declarações dentro de blocos `begin/end` sejam restritas a esse escopo.

Translator

Por fim criamos o ficheiro **"translator.py"** que, como o nome indica, traduz a árvore sintática resultante da execução do parser, passando então o conjunto dos vários tokens representativos do código **Pascal** para instruções compativeis com o funcionamento da EWVM e devolvendo um ficheiro Output.txt com o código resultante. Estas instruções quando colocadas na EWVM produzem o mesmo resultado que o código passado como input.

Na construção do **translator** começamos por dividi-lo em diferentes classes, sendo que a cada classe é atribuído um papel diferente. Desta forma o código tornou-se mais organizado, da mesma forma que a procura e correção de erros tornou-se mais eficiente. O programa encontra-se então dividido nas seguintes classes:

- SymbolTable
- LabelGenerator
- ExpressionTranslator
- StatmentTranslator
- Translator (classe principal)

4.1 SymbolTable

A classe **SymbolTable** permite a alocação das diferentes variáveis, colocando-as dentro de um dicionário. Quando colocada dentro do dicionário é atribuída à variável também um índice, valor o qual é obtido através da contagem total das variáveis. Da mesma forma que guarda o endereço, é guardado também o tipo do mesmo, o que facilitará o trabalho com as variáveis quando as mesmas forem necessárias.

4.1.1 Métodos principais:

- **allocate_var()**: Aloca uma nova variável e retorna seu endereço
- **get_var_info()**: Recupera informações sobre uma variável específica

4.2 LabelGenerator

A classe **LabelGenerator** tem como única função a geração das labels necessárias para rotular elementos como if/else, while, for loops. Dentro da classe foi criado um contador interno de forma a garantir a unicidade dos labels gerados.

Métodos principais:

- **generate()**: Produz um novo rótulo com prefixo personalizado

4.3 ExpressionTranslator

A classe **ExpressionTranslator** é o responsável pela tradução das expressões Pascal (constantes, variáveis, operações aritméticas, relacionais) para instruções da máquina virtual. É através desta classe que operações como "a + b" são transformadas em sequências de instruções PUSH e ADD.

Métodos principais:

- **translate_numeric_constant()**: Aplica um PUSHI a um inteiro passado como input.
- **translate_real_constant()**: Aplica um PUSHF a um número real passado como input.
- **translate_string_constant()**: Aplica um PUSHS a uma string passada como input.
- **translate_variable_ref()**: Aplica um PUSHG ao endereço de uma variável, recebendo o nome desta mesma como input.
- **translate_array_access()**: Função que permite o acesso a um array, recebendo o nome do array e a posição em que vai ser acessada.
- **translate_arithmetic_op()**: Funçao responsável pela tradução das operações aritmétricas.
- **translate_unary_op()**: Função responsável pela tradução de números negativos(uminus) e a operação NOT.
- **translate_relational_op()**: Funçao responsável pela tradução das operações relacionais.
- **translate()**: Tanto nesta classe como na classe **StatmentTranslator**, existe esta função (**translate()**) que serve como coordenador e, recebendo uma dada expressão, utiliza o **isinstance()** ou atributos da expressão fornecida de forma a redirecionar o "trabalho" para outra função.

4.4 StatmentTranslator

Esta classe é responsável pela criação das instruções VM equivalentes aos comandos Pascal (atribuições, if/else, loops, read/write). Utiliza o auxilio da classe ExpressionTranslator para processar expressões dentro dos comandos e gera o controle de fluxo adequado.

Métodos principais:

- **translate_assignment():** Função responsável pela tradução das atribuições de valores a variáveis.
- **translate_write() / translate_writeln():** Funções responsáveis pela tradução das funções de print. O **translate_writeln()** aproveita-se do **translate_write()** uma vez que a lógica é exatamente a mesma, acrescentando apenas um WRITELN no fim (que acrescenta newline no final da string.
- **translate_readln()/ translate_readln_array():** Função responsável por criar as instruções que permitiram o input de dados.
- **translate_if():** Função responsável pela criação das instruções que permitiram o funcionamento das estruturas condicionais. Esta função utiliza o **LabelGenerator** para criar as labels que tornaram os saltos condicionais possíveis.
- **translate_while() / translate_for():** Funções responsáveis pela criação das instruções que permitiram a tradução dos ciclos. Estas funções utilizam também o **LabelGenerator** para criar as labels que tornaram os saltos condicionais dentro dos ciclos possíveis.
- **get_expression_type()**: Função auxiliar que permite a obtenção do tipo de uma expressão.

Nota: Na execução do translate_write() foi implementada a receção dos parâmetros para formatar a string como passado no input (exemplo disso, ”write(x:8:2);”), mas o grupo não encontrou solução para implementar essa funcionalidade com instruções VM, sendo esses valores apenas armazenados dentro de variáveis.

4.5 Translator

Esta é a classe coordena todo o processo de tradução, processando declarações de variáveis e integrando todos os tradutores especializados. Recebe a AST do programa Pascal (resultante do funcionamento do parser) e trata de unir as várias peças que irão formar o código completo da máquina virtual, introduzindo as instruções START/STOP.

Métodos principais:

- **translate_declarations()**: Função responsável por declarar as variáveis, verificando o tipo das mesmas e introduzindo instruções de PUSH conforme o tipo.
- **translate_program()**: Função coordenadora. Esta função é responsável por chamar todas as classes anteriores de forma a ser feita a tradução completa da AST para instruções da máquina virtual.

Conclusão

Agora que findado o desenvolvimento do projeto, concluimos que este foi um método muito bom de consolidar os conhecimentos adquiridos nesta UC.

Através deste projeto fomos postos à prova em vários pontos. Fomos desafiados a desenvolver uma gramática que conseguisse responder as necessidades de uma linguagem relativamente "nova" para nós.

Em conjunto com o desenvolvimento dessa gramática veio também a atribuição das ações sintáticas presentes no parser, sendo um passo bastante importante para preparar a tradução do código Pascal para código máquina.

Por fim, também fomos postos à prova na tradução, onde através de diversos tokens (árvore sintática) resultantes do funcionamento do parser, criamos as instruções VM que permitem a obtenção do mesmo resultado final que o código Pascal que é recebido no input.

Como grupo consideramos que fizemos um bom trabalho. Apesar das diversas dificuldades que tivemos, principalmente com a criação do código VM resultante, desenvolvemos grande parte das funcionalidades pedidas, mantendo um código organizado e eficiente.