# PL2025-Projeto

**OBS.: Relatório feito e pensado em formato Markdown. O conteúdo que se segue é idêntico ao do README.md do projeto**

## Introdução

Este relatório descreve o processo de desenvolvimento de um compilador para a linguagem Pascal, proposto no âmbito da unidade curricular de Processamento de Linguagens (2024/2025). Irão ser descritas em detalhe as principais fases do projeto: **análise léxica**, **análise sintática**, **análise semântica** e **geração de código** para máquina virtual (VM) fornecida pela equipa docente. A implementação foi feita com recurso aos módulos `lex` e `yacc` da linguagem python.

## Análise Léxica

A primeira etapa deste projeto foi identificar todos os tokens existentes na linguagem Pascal, de forma a que, ao fornecermos um ficheiro de código, este seja analisado e convertido numa lista de tokens.

Para tal foi necessário seguir uma organização na declaração destes padrões, pois quando um token pode ser apanhado por mais do que um padrão, o que se encontra declarado primeiro será o que vai definir o token.

Devido a este problema, o padrão que apanha os identificadores `r'\b[a-zA-Z_]{1}[a-zA-Z0-9_]*\b'` teve de ser o último a ser definido pois este vai ao encontro de keywords pré existentes na linguagem Pascal.

Exemplo de possível problema:

Input:

```
program funcaoTeste
```

Parser:

```
tokens = (
  PROGRAM,
  ID
)

## Keywords

def t_PROGRAM(t):
    r'\bprogram\b'
    return t

## Identifiers

def t_ID(t):
    r'\b[a-zA-Z_]{1}[a-zA-Z0-9_]*\b'
    return t
```

Visto que o padrão do *program* se encontra definido antes do *id* a palavra 'program' será corretamente identificada, evitando assim o conflito que existiria com uma má definição do lexer.

Output:

```
 Tokens:
PROGRAM(program) at line 1, position 5
ID(funcaoTeste) at line 1, position 13
```

O lexer que obtivemos encontra-se localizado no ficheiro *lexer.py.*

## Análise Sintática

Após a definição dos tokens, foi necessário definir as regras que a nossa gramática iria seguir.

### Estrutura definida

Começamos por definir uma estrutura principal com os seguintes **tokens não terminais = {const_decls; func_decls; var_decl; begin_progr}** que consistem nos blocos existentes na liguagem pascal. Cada uma destas produções contém uma lista de tokens não terminais que definem a parte de código correspondente, por exemplo: `const_decls → CONST const_decl_list` , que irá identificar todas as constantes definidas pelo código.

### Statements definidos

De forma a identificar o corpo principal da função, definimos todos os tipos de statements que podem aparecer numa função pascal:

```
P18: statement → simple_statement ';'
               | compound_statement ';'
               | selection_statement
               | while_statement
               | for_statement
               | COMMENT
```

Os *simple_statement* correspondem a todas as linhas de código pascal que acabam com ';' e têm apenas 1 linha.

Os *compound_statement* correspondem a uma lista de statements que se encontram entre um `BEGIN ... END`. Foi realizada uma decisão de dividir a chamada do *compound_statement* em duas produções diferentes sendo uma delas a `P18: statement` que ocorre quando o *compound_statement* se encontra dentro de uma função, e o segundo caso, na produção `P15: begin_progr → compound_statement '.'` que corresponde ao `BEGIN ... END` principal que acaba com um '.' para sinalizar o fim da função.

O *selection_statement* corresponde aos casos `if ... then` ou `if ... then ... else`.

O *while_statement* e o *for_statement* correspondem aos loops existentes em pascal, while loop e for loop respetivamente.

Por fim o statement *COMMENT* corresponde ao token terminal correspondente aos comentários escritos pelo código.

## Gramática completa

A gramática por nós definida foi a seguinte:

```
P1: init → PROGRAM program_name

P2: program_name → ID ';' program_body

P3: program_body → const_decls func_decls var_decl begin_progr
                 | func_decls var_decl begin_progr
                 | const_decls var_decl begin_progr
                 | const_decls func_decls begin_progr
                 | const_decls begin_progr
                 | func_decls begin_progr
                 | var_decl begin_progr
                 | begin_progr

P4: const_decls → CONST const_decl_list

P5: const_decl_list → const_decl
                    | const_decl_list const_decl

P6: const_decl → ID '=' expression ';'

P7: func_decls → func_decl
               | func_decls func_decl

P8: func_decl → FUNCTION ID '(' var_decl_lines ')' ':' type ';' begin_func

P9: begin_func → var_decl compound_statement ';'
               | compound_statement ';'

P10: var_decl → VAR var_decl_lines

P11: var_decl_lines → var_decl_line
                    | var_decl_lines var_decl_line

P12: var_decl_line → id_list ':' type ';'
                   | id_list ':' type

P13: id_list → ID
             | id_list ',' ID

P14: type → INTEGERTYPE
          | FLOATTYPE
          | BOOLEANTYPE
          | STRINGTYPE
```

```
          | STRINGTYPE
          | ARRAY '[' NUMBER '..' NUMBER ']' OF type

P15: begin_progr → compound_statement '.'

P16: compound_statement → BEGIN statement_list END

P17: statement_list → statement
                    | statement_list statement

P18: statement → COMMENT
               | simple_statement ';'
               | compound_statement ';'
               | selection_statement
               | while_statement
               | for_statement

P19: simple_statement → ID ASSIGN expression
                      | WRITEFUNC '(' expression_list ')'
                      | WRITEFUNCLN '(' expression_list ')'
                      | READFUNC expression
                      | READFUNCLN expression
                      | BREAK

P20: selection_statement →
     IF expression THEN statement
   | IF expression THEN inside_statement ELSE statement

P21: inside_statement → simple_statement
                      | compound_statement
                      | inside_selection_statement

P22: inside_selection_statement → IF expression THEN inside_statement
                                | IF expression THEN inside_statement ELSE inside_statement

P23: while_statement → WHILE expression DO statement

P24: for_statement → FOR ID ASSIGN expression TO expression DO statement
                   | FOR ID ASSIGN expression DOWNTO expression DO statement

P25: expression → expression '+' expression
                | expression '-' expression
                | expression '*' expression
                | expression '/' expression
                | expression MOD expression
                | expression AND expression
                | expression GT expression
                | expression LT expression
                | expression GE expression
                | expression LE expression
                | expression EQ expression
                | expression NE expression
                | expression OR expression
                | ORDFUNC expression
                | PREDFUNC expression
                | SUCCFUNC expression
                | LENGTHFUNC expression
                | ID '[' expression ']'
                | ID '(' expression_list ')'
                | '(' expression ')'
                | ID '(' ')'
                | NOT expression
                | STRING
                | NUMBER
                | FLOAT
                | FALSE
```

```
                | TRUE
                | CHAR
                | ID
                | <empty>

P26: expression_list → expression
                     | expression_list ',' expression
```

O parser que obtivemos encontra-se localizado no ficheiro *parser.py*.

# Análise Semântica

Na análise semântica começamos por definir duas tabelas:

- Tabela de Símbolos Global: Esta tabela serve para o compilador ter sempre conhecimento das variáveis declaradas, do seu tipo(int, string, etc), do seu pointer global na stack e do seu tipo na função pascal(constante, variável ou função);
- Tabela de Funções: Esta tabela guarda o nome da função, o tipo a que esta dá return (int, string, etc) e os parâmetros que recebe (lista de parâmetros com o seu nome, frame point e o tipo).

Com estas duas tabelas podemos referenciar qualquer tipo de variável ou chamar qualquer tipo de função que teremos sempre o conhecimento do seu tipo, de onde se encontra o seu valor na stack e conseguimos evitar a atribuição de valores com o tipo errado a uma variável. Também nos permite chamar corretamente uma função, com o número de parâmetros corretos e a devolver o valor esperado.

A análise semântica foi realizada no ficheiro *semantics.py*.

# Geração de Código

Nesta etapa optamos por converter o código Passcal em código da máquina virtual seguindo uma tradução dirigida pela sintaxe.

Após ter o código verificado pela análise semântica, é realizada a conversão do mesmo para a linguagem VM, tentámos converter qualquer tipo de comando que encontrássemos mas acabámos por não definir *arrays* nem *procedures* em código da VM.

Apesar destes serem reconhecidos pelo parser e lexer não é realizado nada com estes comandos na conversão semântica. De resto qualquer coisa escrita em código pascal será convertida em código da VM **(loops while, loops for, operações lógicas, operações aritméticas, operações relacionais, operações if else, declaração de variáveis e definição de funções e chamadas de funções)**.

Também optámos por implementar algumas funções pré-definidas do Pascal, como os dois tipos de write e read, e ainda as funções ord, length, succ e pred. A função ord devolve o código ASCII de um caractere. A length devolve o tamanho de uma string, já que os arrays ainda não foram suportados na geração de código. As funções succ e pred devolvem o valor seguinte e anterior de um dado valor, respetivamente.

Exemplo de um teste (**teste 14***):

Pascal *teste14.pp*

```
program TesteFuncao;

function Dobro(x: integer): integer;
begin
    Dobro := x * 2;
end;

var
    valor, counter: Integer;
begin
    Write('Introduza um número menor que 10: ');
    ReadLn(valor);
    counter := 1;
    while valor < 10 do
    begin
        valor := Dobro(valor);
        writeln('O ciclo ', counter, ' têm o valor: ', valor);
        if valor < 10 then
            counter := counter + 1;
    end;
    writeln('Foram preciso ', counter, ' ciclos e o valor final foi ', valor);
end.
```

Conversão em código da VM

```
JUMP main

dobro:
    // binop *
```

```
    // binop
    PUSHL -1
    PUSHI 2
    MUL

    RETURN


main:
    START

    // write
    PUSHS "Introduza um número menor que 10: "
    WRITES
    // readln
    READ
    ATOI
    STOREL 0
    WRITELN

    // assign 1 to counter
    PUSHI 1
    STOREL 1

    // While loop: {'condition': ('binop', {'type': '<', 'left': 'valor', 'right': 10}), 'body': ('compound', [('assign', 'valor', (
    labelWhileBegin4:
    // binop <
    PUSHL 0
    PUSHI 10
    INF

    JZ labelWhileEnd4
    // Compound statement(lista de comandos dentro de um begin ... end)
    // assign ('Function_call', {'name': 'Dobro', 'args': ['valor']}) to valor
    // Call da dobro com os parametros ['valor']
    PUSHL 0
    PUSHA dobro
    CALL
    STOREL 0

    // writeln
    PUSHS "O ciclo "
    WRITES
    PUSHL 1
    WRITEI
    PUSHS " têm o valor: "
    WRITES
    PUSHL 0
    WRITEI
    WRITELN

    // If case: {'case': ('binop', {'type': '<', 'left': 'valor', 'right': 10}),
     //'do': ('assign', 'counter', ('binop', {'type': '+', 'left': 'counter', 'right': 1}))}
    // binop <
    PUSHL 0
    PUSHI 10
    INF

    JZ labelEndIF1
    // assign ('binop', {'type': '+', 'left': 'counter', 'right': 1}) to counter
    // binop +
    PUSHL 1
    PUSHI 1
    ADD

    STOREL 1
```

```
labelEndIF1:
JUMP labelWhileBegin4
labelWhileEnd4:
// writeln
PUSHS "Foram preciso "
WRITES
PUSHL 1
WRITEI
PUSHS " ciclos e o valor final foi "
WRITES
PUSHL 0
WRITEI
WRITELN

STOP
```

Tal como a análise semântica, a geração de código foi realizada no ficheiro *semantics.py*.

## Correr testes

De forma a facilitar o processo de teste do programa foi realizado um script (*runTests.py*) que mostra os testes existentes e irá corrê-los de forma a que o ficheiro Pascal chegue até ao ficheiro formato final, um output em formato txt com os comandos que podem ser corridos na máquina virtual.

Os ficheiros teste encontram-se na pasta *testes* e os outputs encontram-se divididos por 3 pastas diferentes( *outputsParser*, *outputsLexer* e *outputsSemantics*) com o nome do teste que estão a realizar.

Todos os testes produzem 3 outputs, a conversão em tokens, a conversão na *tree ast* realizada pelo ply.yacc e por fim o output esperado para colocar na VM disponibilizada pelos docentes.

## Limitações

Como mencionado anteriormente, na parte de geração de código acabámos por não desenvolver suporte para arrays e para procedures no código da VM. Apesar de estes elementos serem reconhecidos pelo léxico e pelo parser, a sua utilização não irá resultar em qualquer ação durante a geração de código. São funcionalidades que gostaríamos de ter implementado no futuro.