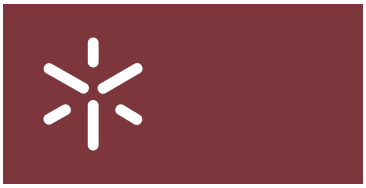

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2024/2025
Grupo 16

Projeto PL

Luís Cunha a104613
Tomás Barbosa a104532
Rafael Pereira a104095

Junho, 2025

PL

Data da Receção	
Responsável	
Avaliação	
Observações	

Projeto PL

Luís Cunha a104613
Tomás Barbosa a104532
Rafael Pereira a104095

Junho, 2025

Índice

Lista de Figuras

1. Introdução

1.1. Objetivo do Projeto

O objetivo deste projeto foi desenvolver um compilador para a linguagem Pascal standard. O compilador é capaz de analisar, interpretar e traduzir código Pascal para um formato intermediário e deste para código máquina.

1.2. Etapas do Projeto

O desenvolvimento do compilador foi estruturado nas seguintes fases principais:

1. **Análise Léxica**

Implementação de um analisador léxico (lexer) com recurso à biblioteca ply.lex. Conversão do código Pascal numa lista de tokens. Reconhecimento de palavras-chave, identificadores, números, operadores e símbolos especiais.

2. **Análise Sintática**

Construção de um analisador sintático (parser) com ply.yacc. Validação da estrutura gramatical do código-fonte segundo a gramática da linguagem Pascal.

3. **Análise Semântica**

Verificação de tipos de dados, declarações de variáveis e coerência semântica do programa.

4. **Geração de Código**

Traduzir o código para a linguagem da máquina virtual (VM).

5. **Testes**

Execução de programas de exemplo em Pascal para validação completa.

2. Desenvolvimento

O desenvolvimento deste projeto seguiu o princípio da modularidade, com as responsabilidades organizadas e distribuídas por diferentes ficheiros.

2.1. Análise Lexical

2.1.1. Tecnologia Utilizada

Nesta fase do projeto, foi implementado um analisador léxico utilizando a biblioteca ply.lex, que nos permite decompor o código-fonte em tokens válidos. Nesta biblioteca as regras léxicas são definidas via expressões regulares e funções, permitindo reconhecer palavras-chave, operadores, literais e delimitadores.

O analisador lexical tem como objetivo converter o código-fonte em unidades léxicas significativas que serão posteriormente utilizadas pelas fases de análise sintática e semântica.

2.1.2. Tokens Definidos

Foram definidos os seguintes grupos de tokens:

Palavras Reservadas: Incluem comandos e estruturas típicas da linguagem:

```
'PROGRAM', 'VAR', 'BEGIN', 'END', 'IF', 'THEN', 'ELSE', 'WHILE', 'DO',
'FOR', 'TO', 'DOWNTO', 'OF', 'ARRAY', 'FUNCTION', 'PROCEDURE', 'CONST',
'TYPE', 'RECORD', 'FORWARD', 'RETURN'
```

Figura 1: Palavras Reservadas

Tipos de Dados: Identificação dos tipos primitivos:

```
'STRING', 'INTEGER', 'CHAR', 'REAL', 'BOOLEAN', 'SET', 'FILE'
```

Figura 2: Tipos

Operadores: Abrangem operadores aritméticos, relacionais e lógicos:

```
'ADD', 'SUB', 'MUL', 'DIVIDE', 'EQUAL', 'NOTEQUAL', 'LESS', 'GREATER',
'LESSEQUAL', 'GREATEREQUAL', 'AND', 'OR', 'NOT', 'DIV', 'MOD', 'POW', 'AT'
```

Figura 3: Operadores

Delimitadores: Símbolos de pontuação e agrupamento:

```
'COMMA', 'ASSIGN', 'COLON', 'SEMICOLON', 'LPAREN', 'RPAREN',
'LBRACKET', 'RBRACKET', 'RANGE', 'DOT'
```

Figura 4: Delimitadores

Literais: Valores constantes e identificadores:

```
'TRUE', 'FALSE', 'INTEGER_CONST', 'REAL_CONST', 'STRING_CONST', 'HEX_CONST', 'CHAR_CONST', 'ID'
```

Figura 5: Palavras Reservadas

Funções de Entrada/Saída: Tokens reservados para funções básicas:

```
'WRITELN', 'READLN'
```

Figura 6: Tokens reservados para funções básicas

Controlo de Fluxo: Comandos adicionais para controlo:

```
'BREAK', 'EXIT', 'CONTINUE'
```

Figura 7: Controlo de fluxo

Depois disto foi necessário definir as expressões regulares que definiriam cada um dos tokens.

Exemplos:

```
t_RANGE = r'\.\.'
t_DOT = r'\.'

def t_PROGRAM(t):
    r'\b(?i)program\b'
    t.type = 'PROGRAM'
    t.value = t.value.lower()
    return t

def t_INTEGER(t):
    r'\b(?i)integer\b'
    t.type = 'INTEGER'
    t.value = t.value.lower()
    return t
```

Figura 8: Exemplo de expressões regulares utilizadas

Desta forma, o lexer permite-nos identificar corretamente as palavras-chave (tokens) da linguagem, assim como detetar eventuais erros léxicos no código fonte (expressões que não são reconhecidas pelo analisador léxico). De notar que este lexer é adaptado para processar apenas programas Pascal standard que incluem declaração de variáveis, expressões aritméticas, comandos de controlo de fluxo (if, while, for), e subprogramas (procedure e function).

2.2. Análise Sintática

Para o desenvolvimento desta fase do trabalho foi usado o ply.yacc que é uma biblioteca em Python usada para criar analisadores sintáticos (ou parsers).

O ply.yacc trabalha em conjunto com o ply.lex, que é usado para fazer a análise léxica, dado que o primeiro usa os tokens para fazer a análise sintática.

O analisador sintático usa regras de gramática para reconhecer a estrutura do código, ou seja, verifica se a sequência de tokens segue uma linguagem válida definida por nós.

Cada regra de gramática é definida como uma função, e o PLY usa estas regras para construir uma árvore sintática.

2.2.1. Gramática

Uma parte fundamental na definição de qualquer linguagem é a especificação da sua gramática. Após análise, a gramática final definida para este projeto foi a seguinte:

- **programa** -> PROGRAM ID SEMI bloco DOT
- **bloco** -> declaracoes_opcionais comando_composto
- **declaracoes_opcionais** -> declaracoes
  | ε
- **declaracoes** -> secao_declaracao | declaracoes secao_declaracao
- **secao_declaracao** -> VAR lista_var SEMI
  | CONST lista_const SEMI
  | TYPE lista_tipo SEMI
  | FUNCTION declaracao_funcao SEMI | PROCEDURE declaracao_procedimento SEMI
- **lista_var** -> declaracao_var
  | lista_var SEMI declaracao_var
- **lista_const** -> declaracao_const
  | lista_const SEMI declaracao_const
- **lista_tipo** -> declaracao_tipo
  | lista_tipo SEMI declaracao_tipo
- **declaracao_funcao** -> ID parametros COLON tipo SEMI bloco
  | ID parametros COLON tipo SEMI FORWARD SEMI
- **declaracao_procedimento** -> ID parametros SEMI bloco
  | ID parametros SEMI FORWARD SEMI
- **parametros** -> LPAREN lista_parametro RPAREN | ε
- **lista_parametro** -> parametro
  | lista_parametro SEMI parametro
- **parametro** -> ID_lista COLON tipo
  | VAR ID_lista COLON tipo
- **ID_lista** -> ID
  | ID_lista COMMA ID
- **declaracao_var** -> ID_lista COLON tipo
- **declaracao_const** -> ID EQUAL expressao
- **declaracao_tipo** -> ID EQUAL tipo
- **tipo** -> INTEGER
  | REAL
  | BOOLEAN
  | CHAR
  | STRING
  | SET OF tipo
  | FILE OF tipo
  | ARRAY LBRACKET INTEGER_CONST RANGE INTEGER_CONST RBRACKET OF tipo
  | RECORD lista_campos END
  | ID
- **lista_campos** -> declaracao_var SEMI
  | lista_campos declaracao_var SEMI
- **comando_composto** -> BEGIN lista_comandos END

- **lista_comandos** -> comando
  | lista_comandos SEMI comando
- **comando** -> comando_sem_if
  | comando_if
- **comando_sem_if** -> atribuicao
  | comando_while
  | comando_for
  | comando_composto
  | comando_writeln
  | comando_readln
  | chamada_funcao
  | BREAK
  | CONTINUE
  | EXIT
  | RETURN expressao
  | ε
- **comando_if** -> IF expressao THEN comando
  | IF expressao THEN comando ELSE comando
- **atribuicao** -> variavel ASSIGN expressao
- **comando_while** -> WHILE expressao DO comando
- **comando_for** -> FOR ID ASSIGN expressao TO expressao DO comando
  | FOR ID ASSIGN expressao DOWNTO expressao DO comando
- **comando_writeln** -> WRITELN LPAREN lista_expressao RPAREN
  | WRITELN LPAREN RPAREN
- **comando_readln** -> READLN LPAREN lista_variavel RPAREN
  | READLN LPAREN RPAREN
- **chamada_funcao** -> ID LPAREN lista_argumentos RPAREN
  | ID LPAREN RPAREN
- **lista_expressao** -> expressao
  | lista_expressao COMMA expressao
- **lista_variavel** -> variavel
  | lista_variavel COMMA variavel
- **variavel** -> ID
  | ID LBRACKET expressao RBRACKET
  | AT ID
  | ID LPAREN lista_argumentos RPAREN
  | ID LPAREN RPAREN
- **lista_argumentos** -> expressao
  | lista_argumentos COMMA expressao
- **expressao** -> expressao_simples
  | expressao_simples operador_relacional expressao_simples
- **expressao_simples** -> termo
  | ADD termo
  | SUB termo
  | expressao_simples operador_adicao termo
- **termo** -> fator
  | termo operador_multiplicacao fator
  | termo POW fator

- **fator** -> variavel
  | INTEGER_CONST
  | REAL_CONST
  | STRING_CONST
  | LPAREN expressao RPAREN
  | CHAR_CONST
  | HEX_CONST
  | TRUE
  | FALSE
  | NOT fator
- **operador_relacional** -> EQUAL
  | NOTEQUAL
  | LESS
  | GREATER
  | LESSEQUAL | GREATEREQUAL
- **operador_adicao** -> ADD
  | SUB
  | OR
- **operador_multiplicacao** -> MUL
  | DIVIDE
  | DIV
  | MOD
  | AND

As regras gramaticais foram então implementadas da seguinte forma:

```python
def p_argument_list(p):
    '''argument_list : expression
                     | argument_list COMMA expression'''
    if len(p) == 2:
        p[0] = [p[1]] if p[1] is not None else []
    else:
        p[0] = p[1] + [p[3]]


def p_expression(p):
    '''expression : simple_expression
                  | simple_expression relational_operator simple_expression'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ('binary_op', p[2], p[1], p[3])
```

Figura 9: Regras gramticais em python

2.3. Análise Semântica

A análise semântica é realizada pelo ficheiro converter.py,que ao gerar deteta os erros semânticos do código. Ao converter o código intermédio para código máquina, podem ser detetadas situações erradas que não permitem uma geração de código máquina consistente, como por exemplo: acessar a variáveis que não estão definidas, acessar uma posição da lista que não existe.

2.4. Geração de Código

O converter.py também é a parte responsável por traduzir código Pascal em instruções executáveis para a máquina virtual. Ele analisa o código intermédio, regista e gerencia variáveis através de uma tabela de símbolos, cria funções ou procedures e no final converte cada comando em Pascal em operações de baixo nível como atribuições, cálculos e controlo de fluxo, permitindo a execução do programa na máquina virtual.

Inicialização:
- Criação das estruturas de dados necessárias

Compilação do Programa Principal:
- Processamento de declarações (variáveis, funções, procedimentos)
- Geração de código para alocação de arrays globais
- Compilação do bloco de instruções principal

Compilação de Funções/Procedimentos:
- Processamento de parâmetros e variáveis locais
- Geração de código para o corpo da função
- Adicionar instruções de retorno

2.5. Funcionamento Geral do Projeto

Como se pode ver, o projeto pode ser comparado a uma linha de montagem, onde cada componente tem uma função específica e sequencial:

Lexer – Responsável por converter o código fonte em tokens, que representam as unidades básicas da linguagem.

Parser – Recebe os tokens gerados e constrói a Árvore de Sintaxe Abstrata (AST), analisando a estrutura gramatical do programa e convertendo para uma representação intermédia.

Conversor – A partir da AST, gera o código correspondente em EWVM e realiza a verificação de erros semânticos.

2.6. Testes

2.6.1. Material Utilizado

O programa foi testado utilizando os exemplos de código fornecidos no enunciado do projeto e criando outros novos.

2.6.2. Metodologia utilizada

As funções escritas em Pascal foram progressivamente convertidas para código da linguagem máquina da EWVM. Em seguida, esses programas foram testados no simulador disponibilizado online. Através da execução dos testes, foi possível verificar se cada função se comportava corretamente e produzia os resultados esperados. Este processo permitiu validar o funcionamento do compilador e identificar possíveis ajustes necessários.

Os testes encontram-se no ficheiro testes.md.

2.7. Dificuldades e pontos a melhorar

As principais dificuldades foram sentidas ao implementar a conversão da representação intermédia para o código máquina, uma vez que foi algo não muito explorado nas aulas.

O programa poderia ser melhorado, tornando-o mais abrangente, ou seja, tendo a capacidade para processar um vocabulário maior.

3. Conclusão

Este projeto teve como principal objetivo proporcionar-nos a experiência prática de desenvolver um compilador funcional para a linguagem Pascal standard. Ao longo do trabalho, foram implementadas as várias fases de um compilador de forma bem-sucedida — desde a análise léxica e sintática até à geração de código para a máquina virtual fornecida.

Para além do aprofundamento dos conceitos teóricos abordados nas aulas, este projeto permitiu desenvolver competências práticas essenciais na área da construção de compiladores, como a definição de gramáticas, a gestão de tokens, e a transformação de código em diferentes representações. O resultado final foi um compilador capaz de interpretar e traduzir programas escritos em Pascal, que cumpre com os requisitos definidos.