Relatório do projeto de PLC 2025-26

Gonçalo Monteiro, Gonçalo Soares, and José Novais

University of Minho, Department of Informatics, 4710-057 Braga, Portugal
e-mail: {a108659,a108393,a108395}@alunos.uminho.pt

Abstract Este trabalho apresenta a implementação de um compilador de Pascal. Neste trabalho temos presente as seguintes etapas: análise léxica (`lexer_pascal.py`), análise sintática (`parserPascal.py`, `automato/lr_automaton.dot`), análise semântica com verificação de tipos e gestão de erros (`analiseSemantica.py`, `erros.py`) e geração de código para a máquina-alvo (`codigoMaquina.py`). O desenvolvimento da gramática está presente em `gramatica.txt`; Tudo isto encontra-se no (`main.py`) usado para testes e visualização.

Índice

1 Introdução

No âmbito da unidade curricular de Processamento de Linguagens e Compiladores (PLC), foi proposto o desenvolvimento de um compilador de Pascal Standard. O objetivo principal deste projeto é aplicar os conhecimentos teóricos adquiridos sobre as diversas fases de compilação, implementando um sistema capaz de transformar código Pascal em código executável na máquina virtual EWVM.

O Pascal é uma linguagem de programação imperativa, estruturada e fortemente tipada. O compilador desenvolvido suporta as principais construções da linguagem Pascal, incluindo:

- Declaração de variáveis com tipos básicos (integer, real, boolean, string)

– Estruturas de controlo (if-then-else, while, for, repeat-until)
– Definição e invocação de funções
– Expressões aritméticas e lógicas
– Arrays e manipulação de strings

A arquitetura do compilador segue o modelo clássico de fases de compilação: análise léxica para identificação de tokens, análise sintática com construção de uma árvore sintática abstrata (AST), análise semântica para verificação de tipos e deteção de erros, e finalmente geração de código para a máquina virtual EWVM.

Este relatório descreve a arquitetura do compilador, as decisões de implementação adotadas em cada fase, a gramática desenvolvida para a linguagem, e os testes realizados para validação do sistema.

2 Arquitetura do Compilador

A arquitetura do compilador segue uma estrutura modular, organizada de forma a distinguir claramente cada etapa do processo de tradução: as análises léxica, sintática e semântica.

2.1 Análise Léxica

O analisador léxico, implementado no módulo `lexer_pascal.py`, constitui a primeira fase do compilador. Este componente é responsável pela leitura do código fonte e pela sua decomposição em *tokens*.

O lexer utiliza expressões regulares para definir os padrões de reconhecimento de cada categoria de tokens.

2.2 Gramática

A gramática da linguagem encontra-se formalizada no ficheiro `gramatica.txt` e segue a notação de gramáticas livres de contexto (BNF). Esta especificação define a estrutura sintática válida dos programas, estabelecendo as regras de produção que governam a composição das diversas construções da linguagem.

A gramática contempla as principais estruturas do Pascal Standard, incluindo declarações de variáveis, definições de funções, estruturas de controlo de fluxo e expressões aritméticas e lógicas. A sua definição formal permite a geração automática das tabelas de parsing necessárias para a análise sintática.

2.3 Análise Sintática

O analisador sintático, implementado em `parserPascal.py`, processa a sequência de tokens produzida pelo lexer e verifica a sua conformidade com a gramática especificada. Este componente emprega um parser LR.

O resultado da análise sintática é uma estrutura de dados hierárquica que representa a árvore sintática abstrata (AST) do programa. Esta estrutura intermédia preserva a semântica do input original enquanto abstrai detalhes sintáticos irrelevantes para as fases seguintes.

3 Representação Intermédia

3.1 Estrutura de Representação Intermédia

A representação intermédia adotada consiste numa estrutura denominada *ASTree* (Abstract Syntax Tree), implementada como uma lista de listas anotadas com etiquetas identificadoras (TAGS). Esta organização permite uma separação clara entre os diferentes componentes do programa.

A ASTree organiza-se em três segmentos principais:

- **Definições de Funções**: Contém as declarações e corpos das funções definidas no programa, incluindo os seus parâmetros e tipos de retorno.
- **Variáveis de Escopo**: Agrupa as declarações de variáveis, distinguindo entre variáveis globais e locais a cada função.
- **Corpo Principal**: Representa o bloco `begin...end` do programa, processado linha a linha para geração de código.

As TAGS associadas a cada nó da estrutura permitem identificar univocamente o tipo de construção representada, facilitando o processamento recursivo durante a geração de código.

3.2 Gestão de Dicionários de Símbolos

O sistema de gestão de escopo baseia-se em dois dicionários distintos: o dicionário global e o dicionário local. Esta separação é fundamental para a resolução correta de identificadores e para a geração de instruções de acesso.

Dicionário Global: O dicionário global armazena informação sobre todas as funções definidas no programa e sobre as variáveis declaradas no escopo global. Para cada entrada, são registados o tipo, a posição de memória e, no caso das funções, a assinatura dos parâmetros. Este dicionário é construído numa fase inicial de análise e permanece inalterado durante a execução.

Dicionário Local: O dicionário local é preenchido dinamicamente em tempo de processamento de cada função. Contém as variáveis locais e os parâmetros formais da função em análise. A gestão do escopo local segue o princípio de que as referências a identificadores são primeiro resolvidas no escopo local e, apenas em caso de insucesso, no escopo global.

Caso Especial: Função MAIN. No contexto da função principal do programa (correspondente ao bloco `begin...end.` exterior), o dicionário local encontra-se vazio. Todas as variáveis acedidas neste contexto são necessariamente globais, simplificando a geração de código para este caso particular.

4 Análise Semântica

A análise semântica, implementada no módulo `analiseSemantica.py`, constitui a fase de validação da correção semântica do programa. Esta etapa complementa a análise sintática ao verificar propriedades que não são expressáveis através de gramáticas livres de contexto.

4.1 Verificação de Declarações

O analisador semântico percorre a árvore sintática para verificar que todos os identificadores utilizados foram previamente declarados no escopo apropriado. Esta verificação previne erros de referência a variáveis ou funções inexistentes.

4.2 Validação de Tipos em Expressões

A verificação de tipos em expressões segue um algoritmo de inferência ascendente na árvore sintática. Para cada operador, são validados os tipos dos operandos e é determinado o tipo resultante da operação.

O módulo `erros.py` centraliza a gestão de mensagens de erro semântico, proporcionando diagnósticos informativos que auxiliam na identificação e correção de problemas no código.

5 Geração de Código

A fase de geração de código, implementada em `codigoMaquina.py`, transforma a representação intermédia em código executável para a máquina virtual EWVM. Esta etapa é caracterizada pelo uso de recursividade como mecanismo fundamental de travessia da estrutura intermédia.

5.1 Alocação de Memória

O modelo de memória da máquina virtual distingue entre o segmento global e a stack. A alocação de espaço para variáveis é determinada durante a análise da estrutura intermédia, sendo atribuídos endereços consecutivos às variáveis.

As variáveis globais são alocadas em posições fixas do segmento global, enquanto as variáveis locais e parâmetros de função residem na stack, com endereços relativos ao *frame pointer* corrente.

5.2 Implementação de Estruturas de Controlo

As estruturas de controlo de fluxo são implementadas através de um sistema de etiquetas (*labels*) e instruções de salto. Cada estrutura de controlo é traduzida num padrão específico de código:

Estruturas Condicionais. As instruções `if-then-else` são implementadas através de saltos condicionais. A condição é avaliada e, com base no resultado, o fluxo de execução é direcionado para o bloco apropriado através de instruções `JZ` (jump if zero) e `JUMP`.

Ciclos. Os ciclos `while`, `for` e `repeat-until` são implementados através de etiquetas de início e fim de ciclo. A geração de etiquetas únicas é assegurada por um contador global, garantindo a ausência de colisões entre diferentes instâncias de estruturas de controlo.

5.3 Geração Recursiva de Expressões

A geração de código para expressões é realizada por uma função recursiva que percorre a sub-árvore correspondente à expressão. Esta função distingue entre diferentes categorias de nós:

- **Constantes**: Geram instruções de carregamento imediato (`PUSHI` para inteiros, `PUSHS` para strings, etc. No caso dos inteiros, gera-se `PUSHI RandomInt` em vez de `PUSHI 0`, de forma a mimetizar o acesso a memória não inicializada (resíduos de memória).").
- **Variáveis Locais**: Geram instruções `PUSHL` com o deslocamento relativo ao frame pointer.
- **Variáveis Globais**: Geram instruções `PUSHG` com o endereço absoluto no segmento global.
- **Operadores**: Geram recursivamente o código dos operandos, seguido da instrução correspondente à operação.

A distinção entre `PUSHL` e `PUSHG` é determinada pela consulta aos dicionários de escopo, verificando primeiro o dicionário local e, em caso de ausência, o dicionário global.

6 Funcionalidades Avançadas

Para além das funcionalidades básicas de um compilador Pascal, o sistema implementa extensões que ampliam a sua aplicabilidade.

6.1 Arrays Multidimensionais

O compilador suporta a declaração e manipulação de arrays com múltiplas dimensões. O cálculo de endereços para acesso a elementos de arrays multidimensionais é realizado através da linearização dos índices.

6.2 Automação Web

O sistema inclui um módulo de automação web, implementado em `webAutomation/`, que permite a submissão automática do código gerado para a plataforma EWVM online. Esta funcionalidade facilita o ciclo de desenvolvimento e teste, automatizando o processo de validação do código gerado.

7 Testes

Esta secção apresenta os casos de teste utilizados para validar o compilador, mostrando para cada um: o código fonte Pascal, a estrutura intermédia (AST) gerada e o código da máquina virtual produzido.

7.1 Teste 1: Hello World

Código Pascal:

```
program HelloWorld;
begin
  writeln('Ola, Mundo!');
end.
```

Estrutura Intermédia:

```
['START', [
  ['FUNÇOES', []],
  ['VARIAVEIS', []],
  ['CorpoMain', [
    ['BEGIN', [
      ['fun', 'writeln(', [
        ("'Ola, Mundo!'",
         'string')
      ]]
    ]]
  ]]
]]
```

Código VM:

```
START:
PUSHI 1
PUSHS "Ola, Mundo!"
WRITES
STOP
```

7.2 Teste 2: Fatorial Iterativo

Código Pascal:

```
program Fatorial;
var
  n, i, fat: integer;
begin
  writeln('Introduza um
    numero inteiro
    positivo:');
  readln(n);
  fat := 1;
  for i := 1 to n do
    fat := fat * i;
  writeln('Fatorial de ',
    n, ': ', fat);
end.
```

Estrutura Intermédia:

```
['START', [
 ['FUNÇOES', []],
 ['VARIAVEIS', [
  (['n','i','fat'],
   'integer')]],
 ['CorpoMain', [
  ['BEGIN', [
   ['fun','writeln(',[
    ("'Introduza...'",
     'string')]],
   ['fun','readln(',[
    ('n','var')]],
   ['Atrib','fat',
    ('1','integer')],
   ['forto',0,1,
    ['Atrib','i',
     ('1','integer')],
    ('n','var'),
    ['Atrib','fat',
     ['*',('fat','var'),
      ('i','var')]]],
   ['fun','writeln(',[
    ("'Fatorial de '",
     'string'),
    ('n','var'),
    ("': '",'string'),
    ('fat','var')]]
  ]]
 ]]
]]
```

Código VM:

```
START:
PUSHI 4
PUSHI 364003341
PUSHI 1171003364
PUSHI 4148601796
PUSHS "Introduza..."
WRITES
READ
ATOI
STOREL 1
PUSHI 1
STOREL 3
PUSHI 1
STOREL 2
label0:
PUSHL 1
PUSHL 2
SUPEQ
jz label1
PUSHL 2
PUSHL 3
MUL
STOREL 3
PUSHI 1
PUSHL 2
ADD
STOREL 2
JUMP label0
label1:
PUSHS "Fatorial de "
WRITES
PUSHL 1
WRITEI
PUSHS ": "
WRITES
PUSHL 3
WRITEI
STOP
```

7.3 Teste 3: Número Primo

Código Pascal:

```
program NumeroPrimo;
var
  num, i: integer;
  primo: boolean;
begin
  writeln('Introduza um
    numero positivo:');
  readln(num);
  primo := true;
  i := 2;
  while (i <= (num div 2))
    and primo do
  begin
    if (num mod i)=0 then
      primo := false;
    i := i + 1;
  end;
  if primo then
    writeln(num,
      ' e primo')
  else
    writeln(num,
      ' nao e primo')
end.
```

Estrutura Intermédia:

```
['START', [
 ['FUNÇOES', []],
 ['VARIAVEIS', [
  (['num','i'],
   'integer'),
  (['primo'],
   'boolean')]],
 ['CorpoMain', [
  ['BEGIN', [
   ['fun','writeln(',[
    ("'Introduza...'"
     ,'string')]],
   ['fun','readln(',[
    ('num','var')]],
   ['Atrib','primo',
    ('true','boolean')],
   ['Atrib','i',
    ('2','integer')],
   ['while',1,2,
    ['and',
     ['<=',...],
     ('primo','var')],
    [['BEGIN',[
     ['if',0,...],
     ['Atrib','i',
      ['+',...]]
    ]]]],
   ['ifelse',3,4,
    ('primo','var'),
    ['fun',...],
    ['fun',...]]
  ]]
 ]]
]]
```

Código VM:

```
START:
PUSHI 4
PUSHI 1987753468
PUSHI 2263874586
PUSHI 515656542
PUSHS "Introduza..."
WRITES
READ
ATOI
STOREL 1
PUSHI 1
STOREL 3
PUSHI 2
STOREL 2
label1:
PUSHL 3
PUSHL 1
PUSHI 2
DIV
PUSHL 2
SUPEQ
AND
jz label2
PUSHI 0
PUSHL 1
PUSHL 2
MOD
EQUAL
jz label0
PUSHI 0
STOREL 3
label0:
PUSHL 2
PUSHI 1
ADD
STOREL 2
JUMP label1
label2:
PUSHL 3
jz label3
PUSHL 1
WRITEI
PUSHS " e primo"
WRITES
JUMP label4
label3:
PUSHL 1
WRITEI
PUSHS " nao e primo"
WRITES
label4:
STOP
```

7.4 Teste 4: Soma de Array

Código Pascal:

```
program SomaArray;
var
  numeros: array[1..5]
    of integer;
  i, soma: integer;
begin
  soma := 0;
  writeln('Introduza 5
    numeros inteiros:');
  for i := 1 to 5 do
  begin
    readln(numeros[i]);
    soma := soma +
      numeros[i];
  end;
  writeln('A soma e: ',
    soma);
end.
```

Estrutura Intermédia:

```
['START', [
 ['FUNÇOES', []],
 ['VARIAVEIS', [
  (['numeros'],
   ([(1,5)],'integer',
    'arr')),
  (['i','soma'],
   'integer')]],
 ['CorpoMain', [
  ['BEGIN', [
   ['Atrib','soma',
    ('0','integer')],
   ['fun','writeln(',[
    ("'Introduza...'"
     ,'string')]],
   ['forto',0,1,
    ['Atrib','i',
     ('1','integer')],
    ('5','integer'),
    [['BEGIN',[
     ['fun','readln(',[
      ('numeros',
       'arr_var',
       [('i','var')])]],
     ['Atrib','soma',
      ['+',...]]
    ]]]],
   ['fun','writeln(',[
    ("'A soma...'"
     ,'string'),
    ('soma','var')]]
  ]]
 ]]
]]
```

Código VM:

```
START:
PUSHI 8
PUSHN 5
PUSHI 1196612151
PUSHI 3603559758
PUSHI 0
STOREL 7
PUSHS "Introduza 5..."
WRITES
PUSHI 1
STOREL 6
label0:
PUSHI 5
PUSHL 6
SUPEQ
jz label1
PUSHFP
PUSHI 1
PADD
PUSHL 6
PUSHI 1
SUB
READ
ATOI
STOREN
PUSHL 7
PUSHFP
PUSHI 1
PADD
PUSHL 6
PUSHI 1
SUB
LOADN
ADD
STOREL 7
PUSHI 1
PUSHL 6
ADD
STOREL 6
JUMP label0
label1:
PUSHS "A soma e: "
WRITES
PUSHL 7
WRITEI
STOP
```

7.5 Teste 5: Conversão Binário para Inteiro

Código Pascal:

```
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 binaria:');
  readln(bin);
  valor := BinToInt(bin);
  writeln('O valor e: ',
    valor);
end.
```

Estrutura Intermédia:

```
['START', [
 ['FUNÇOES', [
  ['BinToInt(',
   ['ARGUMENTOS',[
    (['bin'],'string')]],
   ['VARIAVEIS',[
    (['i','valor',
      'potencia'],
     'integer')]],
   ['CorpoB',[
    ['BEGIN',[
     ['Atrib','valor',
      ('0','integer')],
     ['Atrib','potencia',
      ('1','integer')],
     ['fordownto',1,2,
      ['Atrib','i',
       (['fun',
        'length(',...],
        'fun')],
      ('1','integer'),
      [['BEGIN',[...]]]],
     ['Atrib','BinToInt',
      ('valor','var')]
    ]]]],
   ['TipoRetorno',
    'integer']]]],
 ['VARIAVEIS',[
  (['bin'],'string'),
  (['valor'],'integer')]],
 ['CorpoMain',[...]]
]]
```

Código VM:

```
START:
PUSHI 3
PUSHS ""
PUSHI 2140798298
PUSHS "Introduza..."
WRITES
READ
STOREL 1
PUSHI 0
PUSHL 1
pusha BinToInt
call
pop 1
STOREL 2
PUSHS "O valor e: "
WRITES
PUSHL 2
WRITEI
STOP

BinToInt:
PUSHI 4
PUSHI 2452350365
PUSHI 2925555451
PUSHI 1187894645
PUSHI 0
STOREL 2
PUSHI 1
STOREL 3
PUSHL -1
STRLEN
STOREL 1
label1:
PUSHI 1
PUSHL 1
INFEQ
jz label2
PUSHS "1"
CHRCODE
PUSHL -1
PUSHL 1
PUSHI 1
SUB
CHARAT
EQUAL
jz label0
PUSHL 2
PUSHL 3
ADD
STOREL 2
label0:
PUSHI 2
PUSHL 3
MUL
STOREL 3
PUSHL 1
PUSHI 1
SUB
STOREL 1
JUMP label1
label2:
PUSHL 2
STOREL -2
PUSHL 0
POPN
RETURN
```

7.6 Teste 6: Fatorial Recursivo

Código Pascal:

```
program factorial;
function Factorial(
  x: integer): integer;
begin
  if x = 0 then
    Factorial := 1
  else
    Factorial :=
      Factorial(x-1) * x;
end;

var
  resultado: integer;
begin
  resultado := 99;
  resultado :=
    Factorial(9);
end.
```

Estrutura Intermédia:

```
['START', [
 ['FUNÇOES', [
  ['Factorial(',
   ['ARGUMENTOS',[
    (['x'],'integer')]],
   ['VARIAVEIS',[]],
   ['CorpoB',[
    ['BEGIN',[
     ['ifelse',0,1,
      ['=',('x','var'),
       ('0','integer')],
      ['Atrib',
       'Factorial',
       ('1','integer')],
      ['Atrib',
       'Factorial',
       ['*',
        (['fun',
         'Factorial(',[
          ['-',
           ('1','integer'),
           ('x','var')]]],
         'fun'),
        ('x','var')]]
     ]]]
   ]],
   ['TipoRetorno',
    'integer']]]],
 ['VARIAVEIS',[
  (['resultado'],
   'integer')]],
 ['CorpoMain',[
  ['BEGIN',[
   ['Atrib','resultado',
    ('99','integer')],
   ['Atrib','resultado',
    (['fun',
     'Factorial(',[
      ('9','integer')]],
     'fun')]
  ]]
 ]]
]]
```

Código VM:

```
START:
PUSHI 2
PUSHI 509405757
PUSHI 99
STOREL 1
PUSHI 0
PUSHI 9
pusha Factorial
call
pop 1
STOREL 1
STOP

Factorial:
PUSHI 1
PUSHI 0
PUSHL -1
EQUAL
jz label0
PUSHI 1
STOREL -2
JUMP label1
label0:
PUSHL -1
PUSHI 0
PUSHL -1
PUSHI 1
SUB
pusha Factorial
call
pop 1
MUL
STOREL -2
label1:
PUSHL 0
POPN
RETURN
```

8 Conclusão

O projeto atingiu os objetivos propostos, resultando num compilador funcional capaz de traduzir programas Pascal para a máquina virtual EWVM. A arquitetura modular adotada facilitou o desenvolvimento incremental e a deteção de erros. A integração com a plataforma EWVM através de automação web acelerou o ciclo de testes e validação do código gerado.