Universidade do Minho

Licenciatura em Engenharia Informática

Unidade Curricular: Processamento de Linguagens

Relatório do Projeto

Elementos do Grupo 61:

Gonçalo Rocha Sousa Freitas (a104350)
Vasco João Timóteo Gonçalves (a104527)
Pedro Manuel Macedo Rebelo (a104091)

1 de junho de 2025

Conteúdo

Capítulo 1

Introdução

1.1 Contexto e Objetivo

Este projeto teve como objetivo desenvolver um compilador para Pascal Standard, capaz de gerar código para a máquina virtual EWVM fornecida, abrangendo desde a análise léxica até a geração de código.

1.2 Estrutura do Relatório

Este relatório está organizado em seções que cobrem a análise léxica, sintática, semântica, geração de código, testes e validação, seguidas por uma conclusão.

Capítulo 2

Análise Léxica

2.1 Introdução à Análise Léxica

A análise léxica é a primeira etapa do processo de compilação e é responsável por converter o código-fonte numa sequência de tokens, como palavras-chave, identificadores e símbolos.

2.2 Implementação

O analisador léxico foi implementado no arquivo `lexer_pascal.py` utilizando a biblioteca `ply.lex`, que permite definir regras de análise léxica baseadas em expressões regulares. A implementação segue as especificações do Pascal Standard e foi projetada para reconhecer todos os tokens necessários para os programas de exemplo fornecidos no enunciado, bem como para suportar a gramática completa da linguagem.

Os tokens definidos abrangem as seguintes categorias:

- **Palavras-chave**: `PROGRAM`, `BEGIN`, `END`, `VAR`, `INTEGER`, `BOOLEAN`, `IF`, `THEN`, `ELSE`, `WHILE`, `DO`, `FOR`, `TO`, `FUNCTION`, `PROCEDURE`, `READLN`, `WRITELN`, `WRITE`, `DIV`, `MOD`, `AND`, `OR`, `NOT`.
- **Operadores aritméticos**: `PLUS` (+), `MINUS` (-), `TIMES` (*), `DIVIDE` (/).
- **Operadores relacionais**: `EQUAL` (=), `NEQUAL` (<>), `LT` (<), `LE` (<=), `GT` (>), `GE` (>=).
- **Símbolos**: `ASSIGN` (:=), `LPAREN` (), `RPAREN` ()), `SEMI` (;), `COLON` (:), `DOT` (.), `COMMA` (,), `LBRACKET` ([), `RBRACKET` (]).
- **Tipos de dados**: `ID` (identificadores), `NUMBER` (números inteiros), `STRING` (cadeias de caracteres entre aspas simples).

Cada token foi definido com uma expressão regular correspondente:

- **Palavras-chave e identificadores**: A função `t_ID` utiliza a expressão regular `[a-zA-Z_][a-zA-Z0-9_]*` para capturar identificadores e verifica, de forma case-insensitive, se o texto corresponde a uma palavra-chave. Por exemplo, se o texto for `program`, o token `PROGRAM` é retornado; caso contrário, retorna `ID`.

- **Números**: O token `NUMBER` é definido pela expressão `\d+` e convertido para um inteiro (`t.value = int(t.value)`).
- **Strings**: O token `STRING` usa a expressão `'[^']*'` para capturar cadeias entre aspas simples, removendo as aspas do valor final.
- **Operadores e símbolos**: Cada operador e símbolo possui uma expressão regular específica, como `t_PLUS = r'\+'` para o operador de adição.
- **Comentários**: Comentários entre chaves são ignorados pela regra `t_COMMENT`.
- **Espacos e quebras de linha**: Espaços e utilização de tabs são ignorados com `t_ignore = ' \t'`, enquanto quebras de linha (`\n+`) incrementam o contador de linhas (`t.lexer.lineno`).

A função `t_error` trata caracteres inválidos, exibindo uma mensagem de erro com a linha correspondente e avançando o lexer para o próximo caractere. Essa abordagem garante robustez ao processar entradas malformadas.

2.2.1 Desafios e Soluções

Um desafio significativo foi garantir que palavras-chave fossem diferenciadas de identificadores sem duplicar regras. A solução adotada foi centralizar a verificação na função `t_ID`, comparando o texto capturado (em letras minúsculas) com uma lista de palavras-chave, atribuindo o tipo de token correto. Essa estratégia simplificou a manutenção do código e reduziu redundâncias.

Outro desafio foi lidar com strings e comentários, que podem conter caracteres especiais. A expressão regular para `STRING` foi cuidadosamente projetada para capturar apenas o conteúdo entre aspas, enquanto a regra para comentários ignora todo o texto entre chavetas, conforme a sintaxe do Pascal.(`t_ID`).

2.3 Exemplo de Funcionamento

Para ilustrar, considere o excerto do código em Pascal:

```
1 program Exemplo;
2 var x: integer;
3 begin
4     x := 5;
5 end.
```

O analisador léxico produz os seguintes tokens:

- `PROGRAM`, `ID` ("Exemplo"), `SEMI`, `VAR`, `ID` ("x"), `COLON`, `INTEGER`, `SEMI`, `BEGIN`, `ID` ("x"), `ASSIGN`, `NUMBER` (5), `SEMI`, `END`, `DOT`.

Essa sequência de tokens é então passada ao analisador sintático, demonstrando a correta tokenização do código-fonte.

Capítulo 3

Análise Sintática

3.1 Introdução à Análise Sintática

A análise sintática é a segunda etapa do processo de compilação, responsável por verificar se a sequência de tokens gerada pelo analisador léxico segue as regras da gramática da linguagem Pascal Standard.

3.2 Gramática da Linguagem

A gramática do Pascal Standard foi definida no arquivo `parser_pascal.py` utilizando a biblioteca `ply.yacc`. A gramática abrange as principais construções da linguagem, incluindo programas, blocos, declarações de variáveis, comandos de controle de fluxo (`if`, `while`, `for`), entrada/saída (`readln`, `write`, `writeln`), expressões aritméticas e booleanas, e acesso a arrays.

3.3 Implementação

O parser foi implementado utilizando `ply.yacc`, que processa a gramática definida por meio de funções de produção (e.x., `p_programa`, `p_bloco`). Cada produção gera uma tupla representando um nó da AST, com a estrutura `('tipo', argumentos)`, onde `tipo` identifica a construção (e.g., `programa`, `bloco`, `if`) e `argumentos` contém os subnós correspondentes. Por exemplo, a produção para `programa` gera `('programa', ID, bloco)`, enquanto a produção para `comando_if` pode gerar `('if', expr_bool, comando_then, comando_else)` ou `('if', expr_bool, comando_then, None)` se não houver `ELSE`.

A precedência de operadores foi configurada na variável `precedence`, garantindo que expressões como `a + b * c` sejam avaliadas corretamente (multiplicação antes da adição) e que operadores booleanos (`AND`, `OR`) tenham a precedência adequada. A associatividade à esquerda foi definida para operadores binários, enquanto `NOT` e o operador `MINUS` possuem associatividade à direita.

O parser também lida com erros sintáticos por meio da função `p_error`, que exibe mensagens detalhando o token inválido e a linha correspondente, facilitando a depuração de códigos malformados.

3.4 Exemplo de Funcionamento

Considere o programa Pascal para calcular o maior de três números (Exemplo 2 do enunciado):

```
program Maior3;
var num1, num2, num3, maior: Integer;
begin
    Write('Introduza o primeiro n mero: ');
    ReadLn(num1);
    Write('Introduza o segundo n mero: ');
    ReadLn(num2);
    Write('Introduza o terceiro n mero: ');
    ReadLn(num3);
    if num1 > num2 then
        if num1 > num3 then
            maior := num1
        else
            maior := num3
    else
        if num2 > num3 then
            maior := num2
        else
            maior := num3;
    WriteLn('O maior   : ', maior);
end.
```

O parser processa este código e gera uma AST com a seguinte estrutura (simplificada):

- `('programa', 'Maior3', ('bloco', [('declaracao_vars', ['num1', 'num2', 'num3', 'maior'], 'INTEGER')], [comandos]))`
- Comandos incluem `('write', [('valor', 'Introduza o primeiro número: ')])` para `Write`, `('readln', 'num1')` para `ReadLn`, e `('if', ('rel', '>', ('valor', 'num1'), ('valor', 'num2')), ...)` para as estruturas condicionais.

Essa AST é passada para as fases de análise semântica e geração de código, demonstrando a correta análise sintática do programa.

Capítulo 4

Gerar código EWVM

4.1 Introdução à Geração de Código

A geração de código é a etapa final do compilador, responsável por traduzir a Árvore Sintática Abstrata (AST) gerada nas fases anteriores em instruções executáveis para a máquina virtual EWVM. O objetivo foi produzir código que execute corretamente os programas Pascal fornecidos no enunciado, como o cálculo de fatorial e a verificação de números primos.

4.2 Representação Intermediária

A AST, definida em `ast_semantica.py` e construída em `parser_pascal.py`, serve como representação intermediária. Cada nó da AST é uma tupla ou objeto representando construções como programas, blocos, comandos (`if`, `while`, `for`), atribuições, expressões binárias (`BinOp`), unárias (`UnOp`), variáveis (`Var`) e constantes (`Const`). Essa estrutura hierárquica permite a travessia recursiva para gerar instruções EWVM correspondentes a cada construção do programa.

4.3 Geração de Código para EWVM

A geração de código foi implementada na classe `GeradorEWVM` do arquivo `gerador_ewvm.py`. Esta classe mantém uma lista de instruções (`self.codigo`), uma tabela de símbolos (`self.tabela_simbolos`) para mapear variáveis a endereços de memória, e um contador de endereços (`self.proximo_endereco`) para alocação dinâmica. A geração é realizada por uma travessia recursiva da AST, com métodos específicos para cada tipo de nó:

- **Programa**: O método `gerar_Programa` inicia com `START`, gera o código para o bloco principal e termina com `STOP`.
- **Bloco**: O método `gerar_Bloco` processa todas as declarações de variáveis (`decls`) e comandos (`comandos`), gerando instruções para cada um.
- **Declaração de Variáveis**: Para cada `DeclaracaoVar`, o método `gerar_DeclaracaoVar` empilha um valor inicial (`PUSHI 0`) e armazena-o no endereço da variável (`STOREG endereco`), usando o endereço atribuído na tabela de símbolos.

- **Atribuição**: O método `gerar_ComandoAtribuicao` gera o código para a expressão (`expr`), seguido de `STOREG endereco` para armazenar o resultado na variável correspondente.
- **Expressões Constantes e Variáveis**: Para `Const`, `gerar_Const` empilha o valor com `PUSHI valor`. Para `Var`, `gerar_Var` empilha o valor da variável com `PUSHG endereço`.
- **Expressões Binárias**: O método `gerar_BinOp` gera código para os operandos (`esq` e `dir`) e adiciona a instrução correspondente ao operador: `ADD` (+), `SUB` (-), `MUL` (*), `DIV` (div), `MOD` (mod), `EQUAL` (=), `INF` (<), `INFEQ` (<=), `SUP` (>), `SUPEQ` (>=), `AND` (and), `OR` (or).
- **Expressões Unárias**: O método `gerar_UnOp` gera código para a expressão e adiciona `PUSHI -1; MUL` para negação (-) ou `NOT` para negação lógica (`not`).
- **Condicional (if)**: O método `gerar_tuple` para `('if', ...)` gera código para a condição, seguido de `JZ rotulo_else` para saltar ao bloco `else` (ou fim, se não houver `else`). O bloco `then` é gerado, seguido de `JUMP rotulo_fim` (se houver `else`), e os rótulos são definidos com `rotulo:`.
- **Laço (while)**: Gera um rótulo de início (`rotulo_inicio:`), o código da condição, `JZ rotulo_fim` para sair se falso, o corpo do laço, `JUMP rotulo_inicio` para repetir, e `rotulo_fim:`.
- **Laço (for)**: Inicializa a variável de controle com `STOREG endereco`, define um rótulo de início, testa a condição com `PUSHG endereco; PUSHI fim; INFEQ; JZ rotulo_fim`, gera o corpo, incrementa a variável (`PUSHG endereco; PUSHI 1; ADD; STOREG endereco`), e salta ao início com `JUMP rotulo_inicio`.
- **Entrada/Saída**: Para `readln`, gera `READ; ATOI; STOREG endereco`. Para `write` e `writeln`, gera código para cada argumento (números com `WRITEI`, strings com `WRITES`), com `WRITELN` adicionando uma quebra de linha.

Rótulos são gerados dinamicamente com `novo_rotulo` (e.g., `L0`, `L1`), e a tabela de símbolos associa cada variável a um endereço único, incrementando `proximo_endereco`.

Exemplo de Funcionamento Considere o programa Pascal para calcular o fatorial (Exemplo 3 do enunciado):

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

O código EWVM gerado (simplificado) é:

```
START
PUSHI 0
PUSHI 0
PUSHI 0
PUSHS "Introduza um n mero inteiro positivo: "
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 1
STOREG 1
L0:
PUSHG 1
PUSHG 0
INFEQ
JZ L1
PUSHG 2
PUSHG 1
MUL
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L0
L1:
PUSHS "Fatorial de "
WRITES
PUSHG 0
WRITEI
PUSHS ": "
WRITES
PUSHG 2
WRITEI
WRITELN
STOP
```

Este código inicializa variáveis (`n`, `i`, `fat`), executa a entrada/saída e o laço `for`, e produz a saída correta no simulador EWVM.

Capítulo 5

Testes e Validação

5.1 Metodologia de Testes

Os testes foram realizados com os programas de exemplo do enunciado, compilando-os e executando o código EWVM no simulador.

5.2 Exemplos de Programas Pascal

Testamos programas como "Olá, Mundo!", cálculo de fatorial e verificação de números primos.

5.3 Resultados dos Testes

Todos os exemplos foram compilados e executados com sucesso, validando as etapas do compilador.

5.4 Limitações e Possíveis Melhorias

O compilador não suporta otimizações ou funções complexas, que poderiam ser adicionadas em versões futuras.

Capítulo 6

Conclusão

6.1 Resumo do Trabalho

Foi desenvolvido um compilador funcional para Pascal Standard, atendendo aos requisitos do projeto.

6.2 Aprendizagem e Reflexões

O projeto proporcionou a aprendizagem sobre as fases de compilação e o uso de ferramentas como PLY, com potencial aplicação em projetos de tradução de linguagens.