Universidade do Minho
Escola de Ciências

Processamento de Linguagens e Compiladores (3ª ano de LCC)

Trabalho Prático

Relatório

Grupo 4

Marco António Vieira da Silva (A97554)
Pedro Domingues Viana (A98979)
João Pedro da Silva Faria (A100062)

12 de janeiro de 2025

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

O desenvolvimento de compiladores é uma área essencial no campo da ciência da computação, pois permite a tradução de linguagens de alto nível para formatos executáveis por máquinas. Este relatório documenta o progresso e os resultados obtidos no âmbito do trabalho prático da unidade curricular de Processamento de Linguagens e Compiladores (PLC).
O projeto consistiu na implementação de um analisador léxico e sintático utilizando a biblioteca *Python Lex-Yacc* (PLY). Estes componentes formam a base de um compilador, permitindo a análise e validação de programas escritos em uma linguagem específica, cuja gramática foi previamente definida.

Capítulo 2

Gramática do Projeto

Neste capítulo, é apresentada a gramática utilizada no projeto para definir a linguagem de programação alvo. A gramática foi construída com base em uma abordagem livre de contexto e implementada no analisador sintático, permitindo o reconhecimento de estruturas de código e sua validação semântica.

2.1 Definição da Gramática

A gramática do projeto foi projetada para suportar elementos comuns de linguagens de programação, como:

- Estruturas de controle (`if`, `if-else`, `while`, `for`).
- Declaração e manipulação de variáveis.
- Operações aritméticas e condicionais.
- Entrada e saída de dados.

A gramática é composta pelas seguintes produções principais:

2.1.1 Produções Principais

- **Init:** O ponto de entrada da gramática, representando o início do programa.
- **Cmds:** Uma sequência de comandos.
- **Cmd:** Define os diferentes tipos de comandos, como condicionais, laços, atribuições e operações de entrada/saída.
- **Exp:** Representa expressões aritméticas e condicionais.
- **Cond:** Define condições booleanas usadas em estruturas de controle.

2.2 Produções Detalhadas

Abaixo, estão as produções gramaticais detalhadas:

2.2.1 Produção `Init`

A produção inicial:

```
Init : Cmds
```

Ela define que um programa é composto por uma sequência de comandos (`Cmds`).

2.2.2 Produção `Cmds`

```
Cmds : Cmd Cmds
     | /* vazio */
```

Essa produção permite a definição de múltiplos comandos consecutivos ou um programa vazio.

2.2.3 Produção `Cmd`

```
Cmd : Cmd_If
    | Cmd_If_Else
    | Cmd_While
    | Cmd_For
    | Cmd_Write
    | Cmd_Read
    | Atrib
    | VARS
```

A produção `Cmd` representa os diferentes tipos de comandos suportados pela linguagem.

2.2.4 Produção `Exp`

```
Exp : Exp '+' Exp
    | Exp '-' Exp
    | Exp '*' Exp
    | Exp '/' Exp
    | Exp '%' Exp
    | Factor
```

Essa produção descreve operações aritméticas, com suporte a adição, subtração, multiplicação, divisão e módulo.

2.2.5 Produção `Cond`

```
Cond : NOT Cond
     | Cond AND Cond
     | Cond OR Cond
     | Exp '>' Exp
     | Exp '<' Exp
     | Exp '=' '=' Exp
     | Exp '!' '=' Exp
```

As condições booleanas permitem construir expressões lógicas para controle de fluxo.

2.3 Exemplo de Programa

Abaixo está um exemplo de programa que pode ser reconhecido pela gramática:

```
var x;
if (x > 0) then {
    write(x);
} else {
    write(-x);
}
```

Neste exemplo:

- A variável `x` é declarada.
- Uma estrutura `if-else` verifica o valor de `x`.
- Dependendo da condição, escreve-se `x` ou seu valor negativo.

2.4 Conclusão

A gramática desenvolvida neste projeto é suficiente para representar uma linguagem de programação básica, com suporte a estruturas fundamentais de controle, operações e entrada/saída. A sua implementação no analisador sintático garante que o código escrito pelos utilizadores siga as regras gramaticais definidas.

- **Analisador Léxico:** Responsável pela identificação e categorização dos elementos básicos da linguagem (tokens), como palavras reservadas, identificadores e operadores.
- **Analisador Sintático:** Encarreggado de verificar a conformidade do código fonte com as regras gramaticais, traduzindo as estruturas da linguagem para representações mais próximas do código intermediário ou máquina.

Capítulo 3

Analisador Léxico

Este capítulo descreve a implementação do analisador léxico utilizando a biblioteca PLY. Abaixo, encontra-se o código completo:

3.1 Código do Analisador Léxico

```
literals = ('{', '}', ',', '[', ']', '(', ')','=',';','+','-','*','>','<','!')

tokens = [
    'ID',
    'NUM'
]

p_reservadas = {
    'if': 'IF',
    'then': 'THEN',
    'else': 'ELSE',
    'while': 'WHILE',
    'do': 'DO',
    'for': 'FOR',
    'write': 'WRITE',
    'read': 'READ',
    'not': 'NOT',
    'and': 'AND',
    'or': 'OR',
    'var': 'VAR',
    'arr': 'ARR',
    'input': 'INPUT',
    'print': 'PRINT'
}

tokens += list(p_reservadas.values())  #Atualiza os tokens com palavras reservadas


def t_ID(t):
    r'[A-Za-z_][A-Za-z_0-9]*'
    t.type = p_reservadas.get(t.value, 'ID') # Checka as palavras reservadas
    return t

def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_COMMENT(t):
    r'\#.*'
    pass
```

Figura 3.1: Código do Analisador Léxico

3.2 Descrição

O analisador léxico identifica os tokens da linguagem, como palavras reservadas, identificadores e números. A função principal utiliza expressões regulares para definir as regras de reconhecimento de cada token.

3.3 Tokens Implementados

Os tokens implementados incluem:

- **Palavras Reservadas:** `if`, `while`, `for`, `write`, `read`, entre outras.
- **Identificadores:** Variáveis e arrays definidos pelo utilizador.
- **Operadores:** Soma, subtração, multiplicação, divisão, etc.

Capítulo 4

Analisador Sintático

Este capítulo detalha a implementação do analisador sintático, também baseado na biblioteca PLY. Ele é responsável por verificar a estrutura gramatical do código. Foi adcionado tambem ao codigo a opção de escolher o nome do ficheiro de entrada e do ficheiro de saida.

4.1 Código do Analisador Sintático

```python
import ply.yacc as yacc

def p_init(p):
    "Init : Cmds"
    p[0] = p[1]

def p_cmds(p):
    "Cmds : Cmd Cmds"
    p[0] = p[1] + p[2]

def p_cmds2(p):
    "Cmds : "
    p[0] = ""

def p_cmd1(p):
    "Cmd : Cmd_If"
    p[0] = p[1]

def p_cmd2(p):
    "Cmd : Cmd_If_Else"
    p[0] = p[1]

def p_cmd3(p):
    "Cmd : Cmd_While"
    p[0] = p[1]

def p_cmd4(p):
    "Cmd : Cmd_For"
    p[0] = p[1]

def p_cmd5(p):
    "Cmd : Cmd_Write"
    p[0] = p[1]

def p_cmd_write(p):
    "Cmd_Write : PRINT '(' Exp ')'"
    p[0] = p[3] + "WRITEI\n"

def p_cmd_read(p):
    "Cmd_Read : INPUT '(' ID ')'"
    if p[3] in parser.vars:
        p[0] = f"READ\nATOI\nSTOREG {parser.vars[p[3]][0]}"
```

Figura 4.1: Parte do codigo

4.2 Estrutura do Analisador Sintático

A implementação está estruturada da seguinte forma:

- **Definição da Gramática:** Produções gramaticais que definem as regras sintáticas da linguagem.
- **Funções Semânticas:** Associadas às produções, para realizar a geração de código intermediário.
- **Controle de Variáveis:** Um mapeamento que rastreia variáveis e arrays declarados.

4.3 Produções Gramaticais

A gramática cobre diversas estruturas comuns, como ciclos for, condições e atribuições.

4.3.1 Estrutura Principal

A entrada do programa é definida pela produção `Init`, que chama uma sequência de comandos:

```
Init : Cmds
Cmds : Cmd Cmds
     | /* vazio */
```

4.3.2 Comandos

Os comandos incluem estruturas de controle, atribuições, declarações e operações de entrada/saída:

```
Cmd : Cmd_If
    | Cmd_If_Else
    | Cmd_While
    | Cmd_For
    | Cmd_Write
    | Cmd_Read
    | Atrib
    | VARS
```

4.3.3 Expressões e Condições

A gramática suporta expressões aritméticas e condições lógicas:

```
Exp : Exp '+' Exp
    | Exp '-' Exp
    | Exp '*' Exp
    | Exp '/' Exp
    | Factor

Cond : NOT Cond
     | Cond AND Cond
     | Exp '>' Exp
     | Exp '=' '=' Exp
```

4.4 Geração de Código Intermediário

Cada produção gramatical é associada a uma função semântica para traduzir a entrada em código intermediário. Exemplos incluem:

4.4.1 Estruturas de Controle

Para estruturas condicionais `if` e `if-else`, o código intermediário utiliza pontos de salto (`ponto`):

```
Cmd_If : IF '(' Cond ')' THEN '{' Cmds '}'
p[0] = p[3] + f"JZ ponto{parser.pts}\n" + p[7] + f"ponto{parser.pts}:\n"
parser.pts += 1
```

4.4.2 Operações Aritméticas

As expressões aritméticas são traduzidas para instruções como `ADD`, `SUB`, e `MUL`:

```
Exp : Exp '+' Exp
p[0] = p[1] + p[3] + "ADD\n"
```

4.5 Execução e Entrada de Dados

O programa lê um arquivo com extensão `.sus` contendo o código de entrada. O fluxo principal é o seguinte:

1. Ler o arquivo fornecido na linha de comando.
2. Processar o conteúdo com o analisador léxico e sintático.
3. Gerar código intermedio em um arquivo de saída.

```
if "-h" in sys.argv:
    print(f"Usage: python {sys.argv[0]} filename(.sus) (-o savefile [optional])")
    sys.exit()
try:
    filename = sys.argv[1]
    output = filename [:-4] + ".out"
    if "-o" in sys.argv:
        output = sys.argv[3]
    if not filename.endswith(".sus"):
        print("Filename has to end with .sus")
        sys.exit()
except (ValueError, IndexError):
    print("Use -h to get instruction")
    sys.exit()

try:
    with open(filename,"r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: File not found.")
    sys.exit()


res = parser.parse(content)
allvars = {}
allvars.update(parser.vars)
allvars.update(parser.arrs)

allvars = {k: v for k, v in sorted(allvars.items(), key=lambda item: item[1][0])}

variavies = ""
for v in allvars.keys():
    if allvars[v][1] == 0:
        variavies += "PUSHI 0\n"
    else:
        variavies += f"ALLOC {allvars[v][1]}\n"

res = variavies + "START\n" + res

if (parser.sucesso):
    with open(output,"w") as f:
```

Figura 4.2: Parte do codigo responsavel pela alteração do nome

Capítulo 5

Conclusão

Concluindo o projeto, destacamos a implementação dos analisadores léxico e sintático, que formam a base para um compilador funcional. Ambos foram desenvolvidos utilizando boas práticas de programação e bibliotecas robustas, garantindo precisão e extensibilidade. Achámos a realização deste trabalho interessante e até mesmo "divertida". Graças ao trabalho foi nos possível, não só saber como uma linguagem de programação funciona, mas também ganhar experiência com a linguagem pyhton e desafiarmo-nos a nós mesmos.