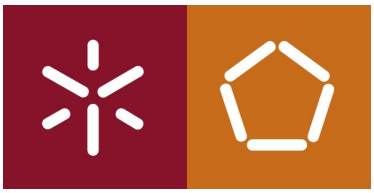

Universidade do Minho
Licenciatura em Engenharia Informática
3ºano - 2º Semestre

Processamento de Linguagens

Projeto final - Extensão funcional para Python

A93192 - André Gil

1 de junho de 2025

Conteúdo

Breve descrição do enunciado

No âmbito da unidade curricular de Processamento de Linguagens foi-nos proposto a realização de um projeto que consiste no desenvolvimento de um compilador simples para uma linguagem baseada em Pascal, utilizando Python e a biblioteca PLY (Python Lex-Yacc). O compilador analisa a sintaxe e traduz o código para uma linguagem de máquina virtual (VM).

Semântica e sintaxe

Nesse sentido o grupo começou por definir as regras de sintaxe e semântica da linguagem a construir.

A análise sintática tem como objetivo verificar se a estrutura do programa está de acordo com as regras da gramática definida. Foi utilizada a biblioteca PLY, que permite definir regras de produção em Python, de forma semelhante ao Yacc/Bison.

A gramática implementada suporta os seguintes elementos:

A análise sintática tem como objetivo verificar se a estrutura do programa está de acordo com as regras da gramática definida. Foi utilizada a biblioteca `PLY`, que permite definir regras de produção em Python, de forma semelhante ao Yacc/Bison.

A gramática implementada suporta os seguintes elementos:

- **Declarações de variáveis**, incluindo arrays:

```
var x, y: integer;
var arr: array[1..5] of integer;
```

- **Tipos**: `integer`, `boolean`, `string`
- **Blocos de comandos** com `begin ... end`:

```
begin
  x := 5;
  writeln(x);
end
```

- **Estruturas de controlo**:
  - `if ... then ... else`
  - `while ... do`
  - `for ... to ... do`
  - `for ... downto ... do`

- **Expressões lógicas e aritméticas**, com precedência:

  ```
  a + b * 2 > 10 and x = y
  ```

- **Comandos de entrada/saída**:

  ```
  readln(x);
  writeln('Mensagem:', x);
  ```

A gramática foi enriquecida com regras de *precedência e associatividade* para evitar ambiguidades.

Análise Semântica e Geração de Código VM

A análise semântica foi feita de forma implícita, durante a geração do código VM. Apesar de não ser feita verificação de tipos ou existência de variáveis, a tradução só ocorre se a estrutura for sintaticamente válida.

Exemplos:

- **Atribuição**:

  ```
  x := 5 + 3;
  ```

  Gera:

  ```
  PUSH 5
  PUSH 3
  ADD
  STORE x
  ```

Gramática

De seguida o grupo definiu uma gramática de acordo com a semântica e sintaxe estabelecidas. Esta gramática cobre todas as ações definidas para os blocos FPYTHON.

```
programa       : 'program' id ';' declaracoes bloco '.'

declaracoes    : 'var' lista_decl_var
               | empty

lista_decl_var : decl_var ';' '{' decl_var ';' '}'
```

```
decl_var      : lista_id ':' tipo
                | id ':' 'array' '[' number '..' number ']' 'of' tipo

lista_id      : id { ',' id }

tipo          : 'integer'
                | 'boolean'
                | 'string'

bloco         : 'begin' lista_comandos 'end'

lista_comandos : comando { ';' comando }

comando       : atribuicao
                | leitura
                | escrita
                | condicional
                | repeticao
                | para
                | bloco
                | empty

atribuicao    : id ':=' expr
                | id '[' id ']' ':=' expr

leitura       : 'readln' '(' id ')'
                | 'readln' '(' id '[' id ']' ')'

escrita       : 'writeln' '(' argumentos ')'

argumentos    : argumento { ',' argumento }

argumento     : expr | string

condicional   : 'if' expr_bool 'then' comando
                | 'if' expr_bool 'then' comando 'else' comando

repeticao     : 'while' expr_bool 'do' comando

para          : 'for' id ':=' expr 'to' expr 'do' comando
                | 'for' id ':=' expr 'downto' expr 'do' <comando>

expr_bool     : expr
                | expr op_rel expr
                | expr_bool 'and' expr_bool
```

```
                | expr_bool 'or' expr_bool

op_rel          : '='
                | '<>'
                | '<'
                | '<='
                | '>'
                | '>='

expr            : termo { ('+' | '-') termo }

termo           : fator { ('*' | 'div' | 'mod') fator }

fator           : number
                | string
                | 'true'
                | 'false'
                | id
                | id '[' id ']'
                | '(' expr_bool ')'
                | 'not' fator
                | 'length' '(' id ')'

id              : str
number          : int
string          : str
```

A gramática definida para este projeto foi elaborada com base nas regras sintáticas da linguagem Pascal, adaptadas para serem processadas por um parser LR gerado com a biblioteca `PLY` (Python Lex-Yacc). Esta gramática permite reconhecer e interpretar programas escritos numa linguagem imperativa estruturada, cobrindo as principais construções da linguagem, tais como:

- **Declarações de variáveis**, incluindo variáveis escalares e vetores (arrays).
- **Comandos de entrada/saída**, como `readln` e `writeln`, com suporte para múltiplos argumentos.
- **Estruturas de controlo**, incluindo:
  - Condicionais (`if ... then ... else`)
  - Ciclos `while`
  - Ciclos `for`, com suporte a incremento e também a `downto` (decremento)
- **Expressões aritméticas e lógicas**, com operadores como `+`, `-`, `*`, `div`, `mod`, `and`, `or`, entre outros.

- **Acesso a vetores** com notação `[]`, e uso de funções simples como `length`.

A gramática foi cuidadosamente modularizada para separar a parte **lexical** (realizada com o ficheiro `lexer.py`) da parte **sintática**, garantindo clareza e flexibilidade na evolução do projeto.

Do ponto de vista técnico:

- É uma **gramática livre de contexto (CFG)**.
- Foi projetada para ser compatível com um **analisador sintático LALR(1)**, conforme exigido pela ferramenta PLY.
- Foram resolvidos manualmente conflitos do tipo `shift/reduce` e `reduce/reduce`, usando regras de precedência e associatividade.

Arquitetura da solução

De modo a atender às necessidades do projeto o grupo decidiu dividir o código em dois ficheiros: "lexer.py"e "parser_FP.py". O primeiro é responsável por ler o ficheiro de input e identificar todos os tokens lidos de modo a que o segundo possa traduzir todos os blocos de FPYTHON e produzir um novo ficheiro de Python puro.

Lexer

O lexer reconhece os seguintes **tokens** principais:

- **Identificadores e literais**: `ID`, `NUMBER`, `STRING`
- **Operadores aritméticos**: `PLUS (+)`, `MINUS (-)`, `TIMES (*)`, `DIVIDE (/)`
- **Operadores relacionais**: `EQUAL (=)`, `NEQ (<>)` , `LT (<)`, `LE (<=)`, `GT (>)`, `GE (>=)`
- **Operadores lógicos**: `AND`, `OR`, `NOT`
- **Delimitadores**: `LPAREN`, `RPAREN`, `LBRACKET`, `RBRACKET`, `SEMI`, `COLON`, `COMMA`, `DOT`, `DOTDOT`, `ASSIGN`

As seguintes **palavras reservadas** são também reconhecidas:
`program, begin, end, var, integer, boolean, string, readln, writeln, write, if, then, else, while, do, for, to, downto, array, of, true, false, div, mod`

Parser

O lexer reconhece os seguintes **tokens** principais:

- **Identificadores e literais**: `ID`, `NUMBER`, `STRING`
- **Operadores aritméticos**: `PLUS (+)`, `MINUS (-)`, `TIMES (*)`, `DIVIDE (/)`
- **Operadores relacionais**: `EQUAL (=)`, `NEQ (<>)` , `LT (<)`, `LE (<=)`, `GT (>)`, `GE (>=)`
- **Operadores lógicos**: `AND`, `OR`, `NOT`
- **Delimitadores**: `LPAREN`, `RPAREN`, `LBRACKET`, `RBRACKET`, `SEMI`, `COLON`, `COMMA`, `DOT`, `DOTDOT`, `ASSIGN`

As seguintes **palavras reservadas** são também reconhecidas:
`program, begin, end, var, integer, boolean, string, readln, writeln, write, if, then, else, while, do, for, to, downto, array, of, true, false, div, mod`

Conclusão

Em síntese, acredito que conseguiu resolver maior parte dos pontos abrangidos pelo enunciado, sendo que alguns pontos não foram concluidos com sucesso. Reconheço que poderiam ser feitas muitas melhorias. Na generalidade o grupo aprendeu a criar gramáticas e parsers que respondem às necessidades das mesmas com a biblioteca "ply.lex"e desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora.