Processamento de Linguagens

Trabalho Prático

Relatório de Desenvolvimento

João Paulo Machado Abreu
a91755

Ricardo Cardoso Sousa
a96141

Rui Pedro Guise da Silva
a97133

28 de maio de 2023

Resumo

Este relatório aborda o desenvolvimento de uma extensão funcional, ***FPY***, para a linguagem de programação *Python*, com recurso à ferramenta de análise *ply*.

Conteúdo

Capítulo 1

Introdução

A programação funcional é um paradigma de programação que se foca no uso de funções puras, dados imutáveis e funções de ordem superior. Linguagens de paradigma imperativo e orientado a objetos, como *Python*, têm adotado alguns princípios provenientes deste paradigma de computação mas são ainda bastante limitadas comparativamente ao que é possível fazer numa linguagem puramente funcional. Por este motivo, decidimos aceitar um dos 8 desafios propostos no âmbito do trabalho prático da unidade curricular de *Processamento de Linguagens* e desenvolver uma extensão funcional para *Python*, com o objetivo de simplificar a escrita de funções em *Python*, utilizando o paradigma funcional na escrita das mesmas e ainda implementar algumas funcionalidades inexistentes em *Python*, como a composição de funções.

Ao longo deste relatório, iremos abordar os passos de desenvolvimento desta extensão funcional, nomeadamente, a sintaxe definida para a utilização da mesma, a gramática livre de contexto escrita e os analisadores léxico, sintático e semântico implementados de modo a garantir o funcionamento da extensão, acompanhando sempre as nossas explicações com exemplos.

Capítulo 2

Linguagem Desenvolvida

A linguagem desenvolvida tem como nome *FPY* e é uma extensão funcional para *Python*, escrita em ficheiros *Python*, em comentários *multi-line* iniciados com a tag 'FPY'.

2.1 Estrutura de um ficheiro

Um ficheiro da nossa linguagem é um ficheiro *Python*, que pode conter vários blocos da nossa extensão (comentários *multi-line* identificados por 'FPY'), onde podem ser definidas funções com a nossa sintaxe. De notar que as funções definidas em blocos inferiores, não são reconhecidas nos blocos superiores. Exemplo:

```
1  """FPY
2  deff length
3  {
4      case ([]) = 0;
5      case (x:xs) = 1 + length(xs);
6  }
7  """
8
9  a = f_length_([1,2,3])
10 print(a)
```

2.2 Funções

As funções na nossa extensão são declaradas com a *keyword* 'deff', seguida do nome da função e limitadas por chavetas ('{}'), tendo no meio o corpo da função. Por exemplo:

```
deff sumf
{
    case ([]) = 0;
    case (x:xs) = x + sumf(xs);
}
```

2.2.1 Corpo da função

O corpo da função é um conjunto de *case statements*. Cada *case statement* começa com a keyword '`case`'. De seguida são definidos os argumentos do *case*, seguidos pelo literal '`=`' e a atribuição do *case*, terminando com '`;`'. Exemplo:

```
case (x:xs) = x + sumf(xs);
```

2.2.2 Argumentos do case

Através dos argumentos do *case* efetua-se *pattern matching*. É possível um *case* não ter argumentos, sendo isso representado por '`()`'. É possível definir:

- Uma lista com um número mínimo de n elementos, com a notação '$(x_1 : x_2 : \ldots : x_n)$';
- Uma lista vazia, com a notação '`[]`';
- Inteiros;
- *Float's*;
- Booleanos, com a notação '`True`' ou '`False`';
- Identificadores de variáveis.

Exemplos:

- `(f:s:t)`
- `(h:t)`
- `([])`
- `(a,True)`
- `(2,10)`

2.2.3 Atribuição do case

Na atribuição do *case* é possível utilizar dados do tipo inteiro, *float*, booleanos e listas de um só tipo. Para além disso, é possível realizar operações com identificadores de variáveis, chamadas de funções e composição de funções. Porém, não existe nenhuma verificação de tipos para estes casos, o que faz com que a sua utilização seja válida em todas as operações e, portanto, da plena responsabilidade do utilizador. Para além disso, todas as variáveis utilizadas na atribuição de um *case*, têm de ser declaradas nos argumentos do mesmo, o que garante a imutabilidade dos dados. Quanto às funções, apenas podem ser chamadas se tiverem sido definidas no bloco onde estão a ser chamadas, ou, em blocos mais acima, garantindo que todas as funções são puras.

2.2.3.1 Operações com valores numéricos

Entre valores numéricos, é possível efetuar as seguintes operações aritméticas: adição, subtração, multiplicação, divisão, divisão inteira, resto da divisão inteira e exponenciação. Para multiplicar um valor numérico por 1 ou -1, basta, escrever o mesmo antecedido pelo sinal '+' ou '−', respetivamente. Todas estas operações retornam um valor numérico. É também possível efetuar as seguintes operações relacionais: igual, diferente, menor, maior, menor ou igual e maior ou igual, sendo o resultado destas operações um valor booleano. Exemplos:

- `2+1`
- `3.5*2`
- `4^3`
- `4.2//2`
- `-(1*2)`
- `3 > 2`
- `4 == 2.9`
- `-(a*sqrt(4))`

2.2.3.2 Operações com valores booleanos

Entre valores booleanos, é possível efetuar as seguintes operações lógicas: $\wedge$ e $\vee$. É também possível efetuar operações relacionais. Negar uma expressão booleana é também uma operação válida. O resultado destas operações retorna um valor booleano. Exemplos:

- `!b`
- `!(True || False)`
- `a && isEmpty . genList(2)`
- `False > False`
- `True == False`

2.2.3.3 Operações com listas

Com listas, as operações possíveis de realizar, para além das relacionais, são a operação *cons* e a concatenação. Exemplos:

- `[1,2,3] > [2]`
- `[1,2,3] == [2]`
- `[True] : [[False]]`
- `[3.1] ++ [1] ++ list_names`

2.2.3.4 Instruções condicionais

Nesta linguagem, funcional, a estrutura condicional implementada é um *if-then-else*. O '`if`' tem de ser seguido por uma expressão booleana e o tipo das expressões que seguem o '`then`' e o '`else`' tem que ser igual. O tipo de retorno de uma instrução condicional é o tipo de retorno das expressões que sucedem o '`then`' e o '`else`'. Exemplos:

- `if x > 2 then 1+2 else 5`
- `if a then if b then False else True else True`

2.2.3.5 *Input/Output*

Em *FPY* não é permitido o uso de operações *input/output*, uma vez que se trata de uma extensão funcional. Isso é garantido devido ao facto de, nesta linguagem, as únicas funções possíveis de serem chamadas, serem funções definidas na própria. E, também, por ser proibido utilizar qualquer *keyword* presente nos *built-in's* de *Python*, à exceção de '`True`' e '`False`'.

Capítulo 3

Gramática

Para a nossa extensão funcional, definimos uma Gramática Livre de Contexto. Uma gramática **G** é um objeto com quatro componentes: $G = (T, N, S, P)$, onde **T** é o conjunto dos símbolos terminais e **N** o conjunto de símbolos não terminais. **S** é o símbolo inicial, estando ele, obviamente, contido em **N**. E, por fim, **P**, representa as produções gramaticais.

3.1 Símbolos Terminais

Conjunto de símbolos terminais **T**.

$$T = \left\{ \begin{array}{l} \texttt{INTEGER, FLOAT, BOOLEAN, PLUS, MINUS, MULT, DIV, FLOORDIV, MOD, POWER, LPAREN,} \\ \texttt{RPAREN, LSQUARE, RSQUARE, LBRACE, RBRACE, COMMA, COLON, CONCAT, LE, LT, GE, GT,} \\ \texttt{EQ, NE, IDENTIFIER, AND, OR, NOT, PERIOD, SEMICOLON, ASSIGN, FPYINIT, FPYCLOSE,} \\ \texttt{DEFF, CASE, IF, THEN, ELSE} \end{array} \right\}$$

3.2 Símbolos Não Terminais

Conjunto de símbolos não terminais **N**.

$$N = \left\{ \begin{array}{l} \texttt{bool, flo, id, int, case\_argument, case\_arguments, case\_empty, case\_headtailID,} \\ \texttt{case\_headtail, case\_headtail2, case\_input, case\_list, case\_statement, constant,} \\ \texttt{equality, exponential, expr, factor, sum, term, fpy\_program, function\_arguments,} \\ \texttt{function\_body, function\_call, function\_composition, function\_declaration, unary,} \\ \texttt{function\_declarations, join, list, list\_elements, listop, rel, statement} \end{array} \right\}$$

3.3 Símbolo Inicial

O símbolo inicial da nossa gramática é o `fpy_program`.

$S =$ `fpy_program`

3.4 Produções

3.4.1 Produção Inicial

A produção inicial vai de encontro ao que um bloco da nossa linguagem pode ser. Um terminal FPYINIT, de seguida o conjunto de funções, que pode ser vazio e, por fim, FPYCLOSE.

```
fpy_program -> FPYINIT function_declarations FPYCLOSE
             | FPYINIT FPYCLOSE
```

3.4.2 Funções

A nossa produção de declaração de funções dita que ou podemos ter uma declaração de funções, ou uma lista de declarações de funções.

```
function_declarations -> function_declaration
                       | function_declaration function_declarations
```

Quanto à produção referente à declaração de uma função inicia com o *token* DEFF, de seguida um IDENTIFIER referente ao nome da função, um LBRACE, um não terminal referente ao corpo da função e, por fim, um RBRACE.

```
function_declaration -> DEFF IDENTIFIER LBRACE function_body RBRACE
```

3.4.3 Corpo das Funções

O corpo das funções é composto por *case statements*, sendo que cada *case statement* termina com um terminal SEMICOLON.

```
function_body -> case_statement SEMICOLON
               | case_statement SEMICOLON function_body
```

Cada *case statement* é inicializado com o *token* CASE, seguido de um não terminal referente ao seu *input*. De seguida, um *token* ASSIGN e, por fim, um não terminal referente a uma operação.

```
case_statement -> CASE case_input ASSIGN statement
```

3.4.4 Argumentos do case

Os argumentos do case começam com o *token* LPAREN e terminam com o *token* RPAREN. Se houverem argumentos, tem ainda um não terminal entre estes dois *tokens*, referente à lista de argumentos.

```
case_input -> LPAREN RPAREN
            | LPAREN case_arguments RPAREN
```

A lista de argumentos pode ter um elemento, ou pode ter vários, separados pelo terminal COMMA.

```
case_arguments -> case_argument
                | case_argument COMMA case_arguments
```

Cada argumento pode ser uma constante, ou uma representação de lista. Sendo que esta representação de lista pode ser a lista vazia ou uma lista com cabeça e cauda.

```
case_argument -> constant
               | case_list

constant -> flo
          | int
          | bool
          | id

case_list -> case_empty
           | case_headtail
```

Caso seja a lista vazia, temos um *token* `LSQUARE` seguido de um `RSQUARE`. Caso seja uma representação de lista com cabeça e cauda, temos um *token* `IDENTIFIER` e um `COLON` seguido de um não terminal que representa as 2 formas de recursividade possíveis para este caso. Ou podemos terminar com um `IDENTIFIER`, ou ter um novo `case_headtail`, o que permite que esta representação de lista com cabeça e cauda se expanda para uma lista com elementos entre a cabeça e a cauda.

```
case_empty -> LSQUARE RSQUARE

case_headtail -> IDENTIFIER COLON case_headtail2

case_headtail2 -> case_headtailID
                | case_headtail

case_headtailID -> IDENTIFIER
```

3.4.5 Atribuição do case

A atribuição do case é caracterizada pela produção *statement*. Um *statement* pode ser uma operação condicional ou uma expressão.

```
statement -> IF expr THEN statement ELSE statement
           | expr
```

3.4.5.1 Instrução condicional

A operação condicional começa com um *token* `IF` e, de seguida um não terminal `expr`. De seguida os *tokens* `THEN` e `ELSE`, seguidos por um *statement*, respetivamente.

3.4.5.2 Expressões

Quanto às expressões, foram feitas de forma a respeitar o grau de precedência e associatividade dos operadores.

3.4.5.3 Operações lógicas

```
expr -> expr OR join
      | join

join -> join AND equality
      | equality
```

3.4.5.4 Operações relacionais

```
equality -> equality EQ rel
          | equality NE rel
          | rel

rel -> listop LT listop
     | listop GT listop
     | listop LE listop
     | listop GE listop
     | listop
```

3.4.5.5 Operações com listas

```
listop -> sum COLON listop
        | sum CONCAT listop
        | sum
```

3.4.5.6 Operações aritméticas

```
sum -> sum PLUS term
     | sum MINUS term
     | term

term -> term MULT exponential
      | term DIV exponential
      | term FLOORDIV exponential
      | term MOD exponential
      | exponential

exponential -> exponential POWER unary
             | unary
```

3.4.5.7 Operadores unários

```
unary -> NOT unary
       | MINUS unary
       | PLUS unary
       | factor
```

3.4.5.8 Fatores

```
factor -> LPAREN expr RPAREN
        | id
        | function_call
        | int
        | flo
        | bool
        | list

int -> INTEGER

flo -> FLOAT

bool -> BOOLEAN

id -> IDENTIFIER
```

3.4.5.9 Listas

```
list -> LSQUARE RSQUARE
      | LSQUARE list_elements RSQUARE

list_elements -> expr
               | expr COMMA list_elements
```

3.4.5.10 Chamadas de funções

```
function_call -> function_composition LPAREN function_arguments RPAREN
               | function_composition LPAREN RPAREN

function_composition -> id
                      | id PERIOD function_composition

function_arguments -> expr
                    | expr COMMA function_arguments
```

Capítulo 4

Regras de Tradução

4.1 Análise léxica

O código para o analisador léxico pode ser consultado no apêndice A.

Para realizar a análise léxica, utilizamos o *lex* do módulo *ply*, para *Python*. Denotar que o *lex* guarda meta informações sobre os *tokens* (linha, posição e valor). Com este módulo definimos todos os *tokens* da nossa gramática, com uma expressão regular para cada. Definimos também quais os caracteres a ignorar e uma mensagem de erro, que ocorre quando é encontrada alguma string que não se enquadre em nenhum dos *tokens* definidos. No caso concreto da nossa linguagem, quando é encontrada uma string não definida, a execução do programa para e é mostrada uma mensagem de erro na consola, que contém informações sobre a linha e a coluna do primeiro carácter da string que originou o erro, bem como a própria string. No fim do texto ser analisado pelo nosso analisador sintático, ficamos com uma sequência de *tokens*, que será depois avaliada pelo analisador sintático.

4.2 Análise sintática

O código para o analisador sintático pode ser consultado no apêndice B.

Para realizar a análise sintática, utilizamos o *yacc* do módulo *ply*, para *Python*. Depois de o analisador sintático transformar o texto inicial numa sequência de *tokens*, o analisador sintático, seguindo as produções gramaticais anteriormente descritas, verificou se essa sequência de *tokens* obedecia à sintaxe da nossa gramática. Em caso de erro, a execução do programa para e é mostrada uma mensagem de erro na consola, que contém informações sobre a linha e a coluna do *token* onde o erro foi originado.

4.3 Análise semântica

O código para o analisador semântico também pode ser consultado no apêndice B, devido ao facto de ter sido realizado em conjunto com o analisador sintático.

Durante a análise semântica, realizamos verificações para garantir que o texto analisado respeita certos aspetos fundamentais da linguagem. Todas estas verificações foram possíveis de realizar devido ao facto de, para cada produção, podermos retornar uma estrutura de dados para o nível acima ao dessa produção,

onde pudemos inserir todas as informações necessárias e, também, devido a três variáveis criadas por nós no parser: uma que contém o nome de todas as funções declaradas, outra que contém o nome de todas as funções declaradas no bloco corrente e uma terceira que contém todos os warnings encontrados. Nesta secção também é abordada a forma como traduzimos a nossa linguagem para *Python*.

4.3.1 Atribuição do *case*

Realizamos *type checking* nos *statements* para definir onde é que os tipos de dados poderiam ser usados. Cada produção que possa vir a ser reduzida num *statement*, retorna uma estrutura fixa com várias informações sobre a produção. A estrutura pode ser consultada no apêndice D.
Como a nossa extensão é para a linguagem *Python*, não existe um tipo definido nem para as variáveis, nem para as funções e, portanto, o seu uso é permitido em cada operação. Desta vez, ao contrário do que foi feito na secção das produções gramaticais, iremos apresentar as operações com maior precedência em primeiro lugar, de forma a facilitar o raciocínio.

4.3.1.1 Fatores

4.3.1.1.1 Chamadas de funções

Na chamada de funções não existe verificação de tipos, apenas preenchimento da estrutura.

- ***type***: *any*;
- ***lineno***: número da linha do identificador da primeira função chamada;
- ***lexpos***: posição do identificador da primeira função chamada;
- ***lastpos***: posição do último parênteses;
- ***vars***: lista das variáveis chamadas na produção *function_arguments*;
- ***func_called***: lista de identificadores de funções utilizados na produção *function_composition*;
- ***python***: a composição de funções é transformado numa função a chamar outra. Os argumentos são colocados entre parênteses, separados por virgula. Exemplo: `p . l(a,b) = p(l(a,b))`

4.3.1.1.2 Inteiros

Na produção dos inteiros não existe verificação de tipos, apenas preenchimento da estrutura.

- ***type***: *num*;
- ***lineno***: número da linha do primeiro carácter do inteiro;
- ***lexpos***: posição do primeiro carácter do inteiro;
- ***lastpos***: posição do último carácter do inteiro;
- ***vars***: lista vazia;
- ***func_called***: lista vazia;
- ***python***: não há alterações;

4.3.1.1.3 *Float's*

Na produção dos *float's* não existe verificação de tipos, apenas preenchimento da estrutura.

- ***type***: *num*;
- ***lineno***: número da linha do primeiro carácter do *float*;
- ***lexpos***: posição do primeiro carácter do *float*;
- ***lastpos***: posição do último carácter do *float*;
- ***vars***: lista vazia;
- ***func_called***: lista vazia;
- ***python***: não há alterações.

4.3.1.1.4 Booleanos

Na produção dos booleanos não existe verificação de tipos, apenas preenchimento da estrutura.

- ***type***: *boolean*;
- ***lineno***: número da linha do primeiro carácter do booleano;
- ***lexpos***: posição do primeiro carácter do booleano;
- ***lastpos***: posição do último carácter do booleano;
- ***vars***: lista vazia;
- ***func_called***: lista vazia;
- ***python***: não há alterações.

4.3.1.1.5 Identificador de variáveis

Na produção dos identificadores não existe verificação de tipos, apenas preenchimento da estrutura.

- ***type***: *any*;
- ***lineno***: número da linha do primeiro carácter do identificador;
- ***lexpos***: posição do primeiro carácter do identificador;
- ***lastpos***: posição do último carácter do identificador;
- ***vars***: lista com o identificador;
- ***func_called***: lista vazia;
- ***python***: não há alterações.

4.3.1.1.6 Listas

Nas listas, para além do preenchimento da estrutura, existe verificação sobre o tipo dos elementos da lista, de forma a não permitir listas de vários tipos.

- ***type***: *list_* caso seja lista vazia ou uma lista com elementos do tipo *any* e *list_X* caso seja uma lista com elementos do tipo X;
- ***lineno***: número da linha do primeiro parêntese reto da lista;
- ***lexpos***: posição do primeiro parêntese reto da lista;
- ***lastpos***: posição do último parêntese reto da lista;
- ***vars***: lista das variáveis presentes nos elementos da lista;
- ***func_called***: lista das chamadas de função presentes nos elementos da lista;
- ***python***: cada elemento da lista já tem o seu atributo *python* definido, portanto a tradução para *Python* de uma lista apenas precisa de juntar a tradução de todos os elementos da lista, dentro de parênteses retos, separado por vírgulas. Exemplo: `[1, f . l(2)] == [1, f(l(2))]`

A verificação é efetuada na produção *list_elements*. O tipo do primeiro elemento da lista define o tipo da mesma. Sempre que um novo elemento é reconhecido, verifica-se se o seu tipo é o mesmo do da lista até então. Em caso de falha, é levantada uma exceção e o programa termina.

4.3.1.1.7 Expressões entre parênteses

Na expressão entre parênteses, a estrutura é exatamente igual à da expressão, à exceção que na entrada referente à tradução para *Python* é adicionado um parêntese curvo no inicio e no fim, no *lineno* é usada a linha do primeiro parêntese, no *lexpos* é usada a posição do primeiro parêntese, e, no *lastpos*, é usada a posição do último parêntese.

4.3.1.2 Operadores unários

4.3.1.2.1 Negação

Na negação, para além do preenchimento da estrutura, existe verificação sobre o tipo do elemento unário, de forma a não permitir um valor que não seja booleano.

- ***type***: boolean;
- ***lineno***: número da linha do carácter '`!`';
- ***lexpos***: posição do carácter '`!`';
- ***lastpos***: lastpos do elemento unário ;
- ***vars***: lista das variáveis presentes no elemento unário;
- ***func_called***: lista das chamadas de função presentes no elemento unário;
- ***python***: colocamos '`not`' e a tradução para *Python* do elemento unário.

A verificação é efetuada na produção. Caso o elemento unário seja um boolean, o seu tipo é preenchido. Em caso de falha, é levantada uma exceção e o programa termina

4.3.1.2.2 Menos e mais

Nestes casos, para além do preenchimento da estrutura, existe verificação sobre o tipo do elemento unário, de forma a não permitir um valor que não seja um número.

- ***type***: num;
- ***lineno***: número da linha do carácter '+' ou '-';
- ***lexpos***: posição do carácter '+' ou '-';
- ***lastpos***: lastpos do elemento unário;
- ***vars***: lista das variáveis presentes no elemento unário;
- ***func_called***: lista das chamadas de função presentes no elemento unário;
- ***python***: colocamos '+' ou '-' e a tradução para *Python* do elemento unário.

A verificação é efetuada na produção. Caso o elemento unário seja um número, o seu tipo é preenchido. Em caso de falha, é levantada uma exceção e o programa termina

4.3.1.3 Operações aritméticas

Nas produções referentes a operações aritméticas, para além do preenchimento da estrutura, existe verificação sobre o tipo do elemento à esquerda e do tipo do elemento à direita do operador aritmético, de forma a não permitir um valor que não seja um número.

- ***type***: *num*;
- ***lineno***: *lineno* do elemento à esquerda;
- ***lexpos***: *lexpos* do elemento à esquerda;
- ***lastpos***: *lastpos* do elemento à direita;
- ***vars***: lista das variáveis presentes no elemento à esquerda e no elemento à direita;
- ***func_called***: lista das chamadas de função presentes no elemento à esquerda e no elemento à direita;
- ***python***: colocamos a tradução para *Python* do elemento à esquerda ,('^' ou '%' ou '//' ou '/' ou '*' ou '-' ou '+') e a tradução para *Python* do elemento à direita.

A verificação é efetuada na produção. Caso o elemento à esquerda e à direita tenha a entrada *type* igual a *num*, o seu tipo é preenchido. Em caso de falha, é levantada uma exceção e o programa termina.

4.3.1.4 Operações com listas

Nas produções referentes a operações com listas, para além do preenchimento da estrutura, existe verificação sobre o tipo do elemento à esquerda e do tipo do elemento à direita do operador para listas.

4.3.1.4.1 Operador *cons*

No operador *cons*, é verificado se o tipo do elemento à sua direita é uma lista de elementos do tipo do elemento à sua esquerda.

- ***type***: tipo do elemento à direita;
- ***lineno***: *lineno* do elemento à esquerda;
- ***lexpos***: *lexpos* do elemento à esquerda;
- ***lastpos***: *lastpos* do elemento à direita;
- ***vars***: lista das variáveis presentes no elemento à esquerda e no elemento à direita;
- ***func_called***: lista das chamadas de função presentes no elemento à esquerda e no elemento à direita;
- ***python***: colocamos a tradução para *Python* do elemento à esquerda entre parênteses reto, seguido de um '+' e da tradução para *Python* do elemento à direita. Exemplo: `1:[2] == [1] + [2]`

A verificação é efetuada na produção. Caso o tipo do elemento à direita não seja do tipo "list_X", sendo X o tipo do elemento à esquerda, é levantada uma exceção e o programa termina.

4.3.1.4.2 Operador *concat*

No operador *concat*, é verificado se os tipos dos elementos que o rodeiam (esquerda e direita) são listas e, de seguida, ainda se verifica se são do mesmo tipo.

- ***type***: tipo dos elementos que rodeiam o operador;
- ***lineno***: *lineno* do elemento à esquerda;
- ***lexpos***: *lexpos* do elemento à esquerda;
- ***lastpos***: *lastpos* do elemento à direita;
- ***vars***: lista das variáveis presentes no elemento à esquerda e no elemento à direita;
- ***func_called***: lista das chamadas de função presentes no elemento à esquerda e no elemento à direita;
- ***python***: colocamos a tradução para *Python* do elemento à esquerda, seguido de um '+' e da tradução do elemento à direita. Exemplo: `[1] ++ [2] == [1] + [2]`

A verificação é efetuada na produção. Caso o tipo de um dos elementos que rodeiam o operador não seja uma lista, é levantada uma exceção e o programa termina. Caso o tipo dos dois elementos não seja o mesmo, é levantada uma exceção e o programa termina.

4.3.1.5 Operações relacionais

Nas produções referentes a operações relacionais, para além do preenchimento da estrutura, existe verificação sobre o tipo do elemento à esquerda e do tipo do elemento à direita do operador relacional, de forma a não permitir que se comparem expressões de tipos diferentes.

- ***type***: *boolean*;
- ***lineno***: *lineno* do elemento à esquerda;
- ***lexpos***: *lexpos* do elemento à esquerda;
- ***lastpos***: *lastpos* do elemento à direita;
- ***vars***: lista das variáveis presentes no elemento à esquerda e no elemento à direita;
- ***func_called***: lista das chamadas de função presentes no elemento à esquerda e no elemento à direita;
- ***python***: colocamos a tradução para *Python* do elemento à esquerda ,('==' ou '!=' ou '<' ou '>' ou '<=' ou '>=') e a tradução para *Python* do elemento à direita.

A verificação é efetuada na produção. Caso o elemento à esquerda e à direita não tenham o mesmo tipo, é levantada uma exceção e o programa termina.

4.3.1.6 Operações lógicas

Nas produções referentes a operações lógicas, para além do preenchimento da estrutura, existe verificação sobre o tipo do elemento à esquerda e do tipo do elemento à direita do operador lógico, de forma a não permitir um valor que não seja um booleano.

- ***type***: *boolean*;
- ***lineno***: *lineno* do elemento à esquerda;
- ***lexpos***: *lexpos* do elemento à esquerda;
- ***lastpos***: *lastpos* do elemento à direita;
- ***vars***: lista das variáveis presentes no elemento à esquerda e no elemento à direita;
- ***func_called***: lista das chamadas de função presentes no elemento à esquerda e no elemento à direita;
- ***python***: colocamos a tradução para *Python* do elemento à esquerda ,('and' ou 'or') e a tradução para *Python* do elemento à direita.

A verificação é efetuada na produção. Caso o elemento à esquerda e à direita não tenham a entrada *type* com o valor *boolean*, é levantada uma exceção e o programa termina.

4.3.1.7 Instrução condicional

Na produção referente à instrução condicional, para além do preenchimento da estrutura, existe verificação sobre o tipo do elemento que segue o '`if`', para garantir que é um valor booleano. Existe também uma verificação sobre o tipo das expressões que seguem o '`then`' e o '`else`', para garantir que têm o mesmo tipo.

- ***type***: tipo da expressão que segue o '`then`' ou o '`else`';
- ***lineno***: *lineno* do '`if`';
- ***lexpos***: *lexpos* do '`if`';
- ***lastpos***: *lastpos* do elemento que segue o '`then`';
- ***vars***: lista das variáveis presentes nos elementos que seguem o '`if`', o '`then`' e o '`else`';
- ***func_called***: lista das chamadas de função presentes nos elementos que seguem o '`if`', o '`then`' e o '`else`';
- ***python***: '`if`' mais a tradução do elemento que o segue, seguido de '`:`', uma mudança de linha e um *tab*. De seguida, é escrita a tradução do elemento que segue o '`then`', porém, todos os `'\n'` encontrados são substituídos por `'\n\t'`, de forma a garantir a indentação. De seguida, outro carácter de mudança de linha, um '`else:`', nova mudança de linha e *tab*, e a tradução do elemento que segue o '`else`', de novo com a mesma estratégia que garante que a indentação é respeitada.

A verificação é efetuada na produção. Caso o elemento que segue o *if* não tenha a entrada *type* a *boolean* é levantada uma exceção e o programa termina. Caso o tipo das expressões que seguem o '`then`' e o '`else`' não sejam o mesmo, é levantada uma exceção e o programa termina.

4.3.2 Argumentos do *case*

Os tipos permitidos num argumento do case são as constantes(*float*, inteiro, booleano), identificadores de variáveis, e, a representação da lista vazia e de listas com um número mínimo de elementos. A estrutura utilizada nestas produções é semelhante à utilizada anteriormente, no entanto, os tipos permitidos agora sofrem uma pequena alteração. O tipo *any* apenas pode representar identificadores de variáveis, os tipos com listas deixam de ser utilizados da mesma forma. A lista vazia passa a ser representada por *list_empty* e as listas com um número mínimo de elementos passam a ser chamadas de *list_ht*. Para além disso, a entrada "vars"neste contexto é apenas uma lista com todos os identificadores de variáveis utilizados no argumento e é criada uma nova entrada, denominada de "infoVars", que tem o mesmo papel que a entrada "vars"tinha na estrutura definida na atribuição do case. A entrada 'python' apenas faz sentido nas constantes e identificadores de variáveis, e a entrada "func_called"deixa de ser utilizada.

4.3.2.1 Constantes e identificadores

Para as constantes e para os identificadores, o processo de preenchimento da estrutura é exatamente o mesmo em relação ao processo realizado na atribuição.

4.3.2.2 Lista Vazia

Não existe nenhuma verificação de erros, apenas preenchimento da estrutura.

- ***type***: *list_empty*;
- ***lineno***: número da linha do primeiro parêntese reto da lista;
- ***lexpos***: posição do primeiro parêntese reto da lista;
- ***lastpos***: posição do último parêntese reto da lista;
- ***vars***: lista vazia;
- ***infoVars***: lista vazia;
- ***python***: Irrelevante no contexto dos argumentos.

4.3.2.3 Lista com número mínimo de elementos

Para além do preenchimento da estrutura, verifica-se se há a uma repetição no nome das variáveis usadas.

- ***type***: *list_ht*;
- ***lineno***: número da linha do primeiro parêntese reto da lista;
- ***lexpos***: posição do primeiro parêntese reto da lista;
- ***lastpos***: posição do último parêntese reto da lista;
- ***vars***: lista com os identificadores utilizados;
- ***infoVars***: lista com os identificadores utilizados e informações sobre linha e posição dos mesmos;
- ***python***: Irrelevante no contexto dos argumentos.

É efetuada uma verificação sobre se o nome da variável já está a ser utilizado nesta lista. Em caso afirmativo, é levantada uma exceção e o programa termina.

4.3.2.4 Verificação na produção dos argumentos

Na produção dos argumentos é verificado se existem variáveis repetidas no conjunto de todos os argumentos. Em caso afirmativo, é levantada uma exceção e o programa termina.

4.3.2.5 Estrutura de retorno dos argumentos

A estrutura de retorno dos argumentos é uma lista que contém as estruturas de retorno de cada argumento.

4.3.3 Corpo das Funções

4.3.3.1 *Case statement*

No *case statement* é verificado se as variáveis usadas na atribuição do case foram declaradas nos argumentos. Caso não tenham sido, é levantada uma exceção e o programa termina. Cada *case statement* retorna uma estrutura que pode ser consultada no apêndice D.

4.3.3.2 Estrutura de retorno do corpo das funções

A estrutura de retorno do corpo é uma lista que contém as estruturas de cada um dos seus *case statement*.

4.3.4 Funções

4.3.4.1 Declaração de função

Devido ao facto de ser preciso comparar os argumentos de cada *case_statement* da função, como estas estruturas são muito complexas, criamos duas classes. Uma onde pudemos atribuir uma ordem a cada argumento, e outra onde pudemos atribuir uma ordem a cada conjunto de argumentos. De seguida, fizemos algumas verificações sobre os argumentos e, por fim, traduzimos a função para *Python*.

4.3.4.1.1 Número de argumentos diferente

Verifica-se se todos os *case statement* têm o mesmo número de argumentos, e, em caso negativo é lançada uma execeção que termina com a execução do programa.

4.3.4.1.2 Argumentos iguais

Verifica-se se existem *case statements* cujos argumentos são iguais. Caso existam, é adicionado um warning à variável do parser responsável por armazenar os warnings. De seguida, remove-se um dos elementos repetidos(o último a ter sido escrito).

4.3.4.1.3 Tradução da função para *Python*

Em primeiro lugar, organizamos a função por árvore, de forma a termos todos os elementos comuns organizados no mesmo nível. Cada ramo da árvore representa um argumento, e cada folha representa um *statement*, ou seja, um *return*. Cada nível de profundidade da árvore representa um argumento específico, onde o primeiro nível (nível 0) corresponde ao primeiro argumento da função (arg0), o segundo nível (nível 1) corresponde ao segundo argumento (arg1), e assim por diante, sendo que o último nível é onde se encontram os *statements*. De seguida criamos uma função recursiva, que faz uma análise em profundidade da árvore e faz a tradução para *Python*. De seguida, apenas adicionamos ao inicio da tradução a tag '`def`', o nome da função e todos os argumentos, com base na quantidade de argumentos(numa função com 3 argumentos, ficaria: '`def nome_funcao(arg0,arg1,arg2):`'). De seguida, mostra-se um exemplo.

Em *FPY*:

```
deff example
{
    case ([],a,2) = 2*3;
    case ([],b,3) = 10;
}
```

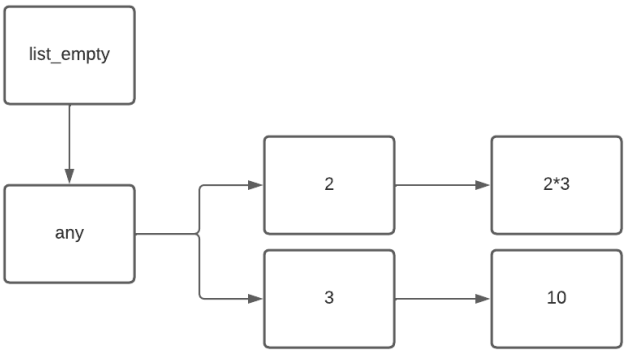

Figura 4.1: Árvore gerada

Em *Python*:

```
def example(arg0,arg1,arg2):
    if len(arg0) == 0:
        if arg2 == 2:
            a = arg1
            return 2*3
        elif arg2 == 3
            b = arg1
            return 10
        else raise ValueError
    else raise ValueError
```

4.3.4.1.4 Estrutura de retorno de uma função

Consultar o apêndice D que fala sobre as estruturas.

4.3.4.2 Funções

Na produção que contém a declaração de todas as funções, verifica-se, para cada uma, se já estava definida. Caso já esteja, adiciona-se um warning à variável do parser responsável por armazená-los. Caso não esteja, adiciona-se a função à variável do parser que contém todas as funções definidas e, também, à variável do parser que contém todas as funções definidas no bloco corrente.

4.3.4.2.1 Estrutura de retorno

A estrutura contém apenas uma entrada com todas as chamadas de funções, de cada função.

4.3.5 Produção Inicial

Na produção inicial, quanto a erros, apenas se verifica se as funções que foram chamadas estão presentes na variável do parser que contém todas as funções declaradas. Em caso negativo, é lançada uma exceção e o programa termina. De seguida, a tradução para *Python* de todas as funções é junta numa só variável. Por fim, é efetuada uma pesquisa nessa variável que procura por chamadas de função e, para cada uma,

acrescenta o prefixo "f_"e o sufixo "_". A estrutura de retorno da produção inicial é a string que contém o código todo traduzido.

4.4 Compilador

O código para o compilador pode ler-se no apêndice C. O ficheiro *fpyCompiler.py* do projeto é o programa referente ao nosso compilador. Pode-se executá-lo da seguinte forma:

```
$ python3 fpyCompiler.py {nome_ficheiro}.py
```

O programa lê todas as linhas do ficheiro de input e guarda numa string. De seguida, utilizando o módulo re, encontramos nessa string todos os blocos FPY, e aplicamos uma função de substituição a cada bloco. Essa função de substituição tem acesso ao parser e ao lexer. Inicialmente, atualiza a linha inicial do lexer, atualiza a variável do parser que tem todas as funções declaradas no bloco atual para a lista vazia e, de seguida, aplica o parser ao texto a substituir. O resultado dessa operação substitui o bloco FPY capturado. De seguida, cria um ficheiro com o mesmo nome do ficheiro de input, mas com o sufixo "FPY"e escreve lá a string resultante. Por fim, ordena os warnings presentes na variável do parser por linha e imprime-os no terminal.

Capítulo 5

Exemplos de utilização

Neste capítulo damos um exemplo da aplicação do compilador a um ficheiro *Python*.

Ficheiro original

```
"""FPY

deff ex{
    case (x:xs,c,a,True) = x * 1;
    case (y:ys,d,b,False) = d * 2;
}

deff maisum{
    case(x) = x+1;
}

deff sumf
{
    case ([]) = 0;
    case (x:xs) = x + sumf(xs);
}

deff soma_impares
{
    case ([]) = 0;
    case (x:xs) = if x%2 == 1 then x + soma_impares(xs) else soma_impares(xs);
}

deff filtra_impares
{
    case ([],2,a) = [];
    case ([],2,1) = [];
    case ([],3,1) = [];
    case (x:y:xs,2,5) = if ! (x % 2 == 0) then filtra_impares(xs) else x :
    ↪ filtra_impares(xs);
```

```
30      case (x:xs,3,5) = if ! (x % 2 == 0) then filtra_impares(xs) else x ++
    ↪   filtra_impares(xs);
31      case (x:xs,3,a) = if ! (x % 2 == 0) then filtra_impares(xs) else x ++
    ↪   filtra_impares(xs);
32  }
33
34  deff soma_impares_2 {
35      case(x) = sumf . filtra_impares(x);
36  }
37
38  deff idF
39  {
40      case (a) = a;
41  }
42
43  deff func_const
44  {
45      case() = [1,3,4,6];
46  }
47
48  deff mult_list_Num
49  {
50      case ([],i,2) = [];
51      case ([],a,x) = a*x : mult_list_Num([],a,x-1);
52      case ([],a,v) = a*v : mult_list_Num([],a,v-1);
53      case ([],a,t) = a*t : mult_list_Num([],a,t-1);
54  }
55
56  deff nzp
57  {
58      case (a) = if a > 0 then 1 else if a == 0 then 0 else a;
59  }
60
61  deff fib
62  {
63      case (0) = 1;
64      case (1) = 1;
65      case (n) = fib(n-1) + fib(n-2);
66  }
67
68  deff maximo
69  {
70      case([],a) = -3 + 5 - 2 ;
71      case(x:xs,b) = maximo (x,maximo (xs));
72  }
73
74  deff ordF
75  {
```

```
    case([])=True;
    case(x:xs) = True && False;
    case(x:y:xs) = x <= y && ordF(y:xs);
}

deff concatena
{
    case(x:xs,ys) = mult(ys);
    case(x,ys) = x : concatena([],ys);
}

deff mult
{
    case (a,b) = a*b;
    case (b,g) = [1,2,3];
}

deff mult
{
    case () = 2;
}
"""

x = 4
y = f_idF_(x)
print(y)
l = [1, 2, 3, 4, 5]
sum_l = f_sumf_(l)
print(sum_l)

"""FPY
"""
```

Ficheiro originado

```
def f_ex_(arg0, arg1, arg2, arg3):
    if len(arg0) >= 1:
        if arg3 == True:
            x = arg0[0]
            xs = arg0[1:]
            c = arg1
            a = arg2
            return x * 1

        elif arg3 == False:
            y = arg0[0]
            ys = arg0[1:]
```

```
            d = arg1
            b = arg2
            return d * 2

        else:
            raise ValueError

    else:
        raise ValueError


def f_maisum_(arg0):
    x = arg0
    return x + 1


def f_sumf_(arg0):
    if len(arg0) >= 1:
        x = arg0[0]
        xs = arg0[1:]
        return x + f_sumf_(xs)

    elif len(arg0) == 0:
        return 0

    else:
        raise ValueError


def f_soma_impares_(arg0):
    if len(arg0) >= 1:
        x = arg0[0]
        xs = arg0[1:]
        if x % 2 == 1:
            return x + f_soma_impares_(xs)
        else:
            return f_soma_impares_(xs)

    elif len(arg0) == 0:
        return 0

    else:
        raise ValueError


def f_filtra_impares_(arg0, arg1, arg2):
    if len(arg0) >= 2:
        if arg1 == 2:
```

```
            if arg2 == 5:
                x = arg0[0]
                y = arg0[1]
                xs = arg0[2:]
                if not (x % 2 == 0):
                    return f_filtra_impares_(xs)
                else:
                    return [x] + f_filtra_impares_(xs)

            else:
                raise ValueError

        else:
            raise ValueError

    elif len(arg0) >= 1:
        if arg1 == 3:
            if arg2 == 5:
                x = arg0[0]
                xs = arg0[1:]
                if not (x % 2 == 0):
                    return f_filtra_impares_(xs)
                else:
                    return x + f_filtra_impares_(xs)

            x = arg0[0]
            xs = arg0[1:]
            a = arg2
            if not (x % 2 == 0):
                return f_filtra_impares_(xs)
            else:
                return x + f_filtra_impares_(xs)

        else:
            raise ValueError

    elif len(arg0) == 0:
        if arg1 == 2:
            if arg2 == 1:
                return []

            a = arg2
            return []

        elif arg1 == 3:
            if arg2 == 1:
                return []

```

```
            else:
                raise ValueError

        else:
            raise ValueError

    else:
        raise ValueError


def f_soma_impares_2_(arg0):
    x = arg0
    return f_sumf_(f_filtra_impares_(x))


def f_idF_(arg0):
    a = arg0
    return a


def f_func_const_():
    return [1, 3, 4, 6]

def f_mult_list_Num_(arg0, arg1, arg2):
    if len(arg0) == 0:
        if arg2 == 2:
            i = arg1
            return []

        a = arg1
        x = arg2
        return [a * x] + f_mult_list_Num_([], a, x - 1)

    else:
        raise ValueError


def f_nzp_(arg0):
    a = arg0
    if a > 0:
        return 1
    else:
        if a == 0:
            return 0
        else:
            return a
```

```
def f_fib_(arg0):
    if arg0 == 0:
        return 1

    elif arg0 == 1:
        return 1

    n = arg0
    return f_fib_(n - 1) + f_fib_(n - 2)


def f_maximo_(arg0, arg1):
    if len(arg0) >= 1:
        x = arg0[0]
        xs = arg0[1:]
        b = arg1
        return f_maximo_(x, f_maximo_(xs))

    elif len(arg0) == 0:
        a = arg1
        return - 3 + 5 - 2

    else:
        raise ValueError


def f_ordF_(arg0):
    if len(arg0) >= 2:
        x = arg0[0]
        y = arg0[1]
        xs = arg0[2:]
        return x <= y and f_ordF_([y] + xs)

    elif len(arg0) >= 1:
        x = arg0[0]
        xs = arg0[1:]
        return True and False

    elif len(arg0) == 0:
        return True

    else:
        raise ValueError


def f_concatena_(arg0, arg1):
    if len(arg0) >= 1:
        x = arg0[0]
```

```
205         xs = arg0[1:]
206         ys = arg1
207         return f_mult_(ys)
208 
209     x = arg0
210     ys = arg1
211     return [x] + f_concatena_([], ys)
212 
213 
214 def f_mult_(arg0, arg1):
215     a = arg0
216     b = arg1
217     return a * b
218 
219 
220 
221 
222 x = 4
223 y = f_idF_(x)
224 print(y)
225 l = [1, 2, 3, 4, 5]
226 sum_l = f_sumf_(l)
227 print(sum_l)
228 
229 
```

Mensagens no *standard output*

```
56:5: <Warning> Redundant input in pattern matching for function 'mult_list_Num'
57:5: <Warning> Redundant input in pattern matching for function 'mult_list_Num'
95:5: <Warning> Redundant input in pattern matching for function 'mult'
98:1: <Warning> Function 'mult' is already defined
```

Capítulo 6

Conclusão

O projeto desenvolvido ajudou o grupo a aperfeiçoar os conhecimentos obtidos, durante o semestre, em Processamento de Linguagens. O tema escolhido permitiu-nos ainda relembrar o funcionamento de uma linguagem de programação funcional.

Tendo em conta o tema escolhido consideramos ainda que cumprimos com os requisitos propostos e ainda implementamos algumas funcionalidades extra, como a composição de funções, por exemplo. Como trabalho futuro, consideramos que podemos aumentar o nosso leque de tipo de dados permitidos e acrescentar mais funcionalidades típicas de linguagens funcionais.

Apêndice A

Analisador Léxico

```
import ply.lex as lex
import builtins

forbidden_names = dir(builtins)

tokens = [
    'INTEGER',
    'FLOAT',
    'PLUS',
    'MINUS',
    'MULT',
    'DIV',
    'FLOORDIV',
    'MOD',
    'POWER',
    'LPAREN',
    'RPAREN',
    'LSQUARE',
    'RSQUARE',
    'LBRACE',
    'RBRACE',
    'COMMA',
    'COLON',
    'CONCAT',
    'LE',
    'LT',
    'GE',
    'GT',
    'EQ',
    'NE',
    'IDENTIFIER',
    'AND',
    'OR',
    'NOT',
```

```
    'PERIOD',
    'SEMICOLON',
    'ASSIGN',
    "FPYINIT",
    "FPYCLOSE"
]

reserved = {
    'True': 'BOOLEAN',
    'False': 'BOOLEAN',
    'if': 'IF',
    'then': 'THEN',
    'else': 'ELSE',
    'deff': 'DEFF',
    'case': 'CASE'
}

tokens += list(set(reserved.values()))


def t_FPYINIT(t):
    r'"""FPY'
    return t


def t_FPYCLOSE(t):
    r'"""'
    t.lexer.lineno += 1
    return t


def t_PERIOD(t):
    r'\.'
    return t


def t_COLON(t):
    r':'
    return t


def t_SEMICOLON(t):
    r';'
    return t


def t_CONCAT(t):
    r'\+\+'
```

```
83      return t
84
85
86  def t_PLUS(t):
87      r'\+'
88      return t
89
90
91  def t_MINUS(t):
92      r'-'
93      return t
94
95
96  def t_MULT(t):
97      r'\*'
98      return t
99
100
101 def t_FLOORDIV(t):
102     r'//'
103     return t
104
105
106 def t_DIV(t):
107     r'/'
108     return t
109
110
111 def t_MOD(t):
112     r'%'
113     return t
114
115
116 def t_POWER(t):
117     r'\^'
118     return t
119
120
121 def t_LPAREN(t):
122     r'\('
123     return t
124
125
126 def t_RPAREN(t):
127     r'\)'
128     return t
129
130
```

```
131 def t_LSQUARE(t):
132     r'\['
133     return t
134
135
136 def t_RSQUARE(t):
137     r'\]'
138     return t
139
140
141 def t_LBRACE(t):
142     r'\{'
143     return t
144
145
146 def t_RBRACE(t):
147     r'\}'
148     return t
149
150
151 def t_COMMA(t):
152     r','
153     return t
154
155
156 def t_LE(t):
157     r'<='
158     return t
159
160
161 def t_LT(t):
162     r'<'
163     return t
164
165
166 def t_GE(t):
167     r'>='
168     return t
169
170
171 def t_GT(t):
172     r'>'
173     return t
174
175
176 def t_EQ(t):
177     r'=='
178     return t
```

```
179
180
181 def t_NE(t):
182     r'!='
183     return t
184
185
186 def t_ASSIGN(t):
187     r'='
188     return t
189
190
191 def t_AND(t):
192     r'&&'
193     return t
194
195
196 def t_OR(t):
197     r'\|\|'
198     return t
199
200
201 def t_NOT(t):
202     r'!'
203     return t
204
205
206 def t_FLOAT(t):
207     r'\d+\.\d+'
208     return t
209
210
211 def t_INTEGER(t):
212     r'\d+'
213     return t
214
215
216 def t_IDENTIFIER(t):
217     r'[a-zA-Z_][a-zA-Z0-9_]*'
218     if t.value in reserved:
219         t.type = reserved[t.value]
220     elif t.value in forbidden_names:
221         line = t.lexer.lineno
222         col = find_column(t.lexer.lexdata, t)
223         raise Exception(f"{line}:{col}: <lexer error> Reserved python token
        ↪  '{t.value}'")
224     else:
225         t.type = 'IDENTIFIER'
```

```
226     return t
227
228
229 def t_newline(t):
230     r'\n+'
231     t.lexer.lineno += len(t.value)
232
233
234 t_ignore = ' \t'
235
236
237 def t_error(t):
238     line = t.lexer.lineno
239     col = find_column(t.lexer.lexdata, t)
240     raise Exception(f"{line}:{col}: <lexer error> Illegal character '{t.value[0]}'")
241
242
243 def find_column(input, token=None, lexpos=None):
244     if token is not None:
245         line_start = input.rfind('\n', 0, token.lexpos) + 1
246         return token.lexpos - line_start + 1
247     elif lexpos is not None:
248         line_start = input.rfind('\n', 0, lexpos) + 1
249         return lexpos - line_start + 1
250     else:
251         raise ValueError("Either token or lexpos must be provided")
252
253
254 lexer = lex.lex()
```

Apêndice B

Analisadores Sintático e Semântico

```
import ply.yacc as yacc
import lexer
import verify
import re
from caseInput import CaseInput

tokens = lexer.tokens

start = 'fpy_program'


def substitute_func_name(match, functions):
    word = match.group(1)
    fm = match.group(0)
    if word in functions:
        return "f_" + word + "_"+"("
    return fm


def p_fpy_program(p):
    """
    fpy_program : FPYINIT function_declarations FPYCLOSE
                | FPYINIT FPYCLOSE
    """
    if len(p) == 3:
        p[0] = ""
    else:
        for func in p[2]["func_called"]:
            if func[2] not in p.parser.functions:
                inputText = p.lexer.lexdata
                line = func[0]
                col = lexer.find_column(inputText, lexpos=func[1])
                raise Exception(f"{line}:{col}: <scope error> Function '{func[2]}'
                ↪  not in scope")
```

```
34
35          code = ""
36          for func in sorted(p.parser.newFunctions, key=lambda x:
            ↪  (p.parser.functions[x]["lineno"], p.parser.functions[x]["col"])):
37              code += p.parser.functions[func]["python"]
38
39          pattern = r'(\b\w+\b)\s*\('
40          finalCode = re.sub(pattern, lambda match: substitute_func_name(match,
            ↪  p.parser.functions), code)
41          p[0] = finalCode
42
43
44  def p_function_declarations(p):
45      """
46      function_declarations : function_declaration
47                            | function_declaration function_declarations
48      """
49      p[0] = {}
50      if len(p) == 2:
51          p[0]["func_called"] = p[1]["func_called"]
52      else:
53          p[0]["func_called"] = p[1]["func_called"] + p[2]["func_called"]
54      func_name = p[1]["func_name"]
55      line = p[1]["lineno"]
56      col = lexer.find_column(p.lexer.lexdata, lexpos=p[1]["lexpos"])
57      if func_name in p.parser.functions:
58          if line < p.parser.functions[func_name]["lineno"]:
59              oldline = p.parser.functions[func_name]["lineno"]
60              oldcol = p.parser.functions[func_name]["col"]
61              p.parser.warnings.append((oldline, oldcol, f"{oldline}:{oldcol}:
                ↪  <Warning> Function '{func_name}' is already defined"))
62              p.parser.functions[func_name] = {"lineno": line, "col": col, "python":
                ↪  p[1]["python"]}
63          elif line == p.parser.functions[func_name]["lineno"]:
64              if col < p.parser.functions[func_name]["col"]:
65                  oldline = p.parser.functions[func_name]["lineno"]
66                  oldcol = p.parser.functions[func_name]["col"]
67                  p.parser.warnings.append((oldline, oldcol, f"{oldline}:{oldcol}:
                    ↪  <Warning> Function '{func_name}' is already defined"))
68                  p.parser.functions[func_name] = {"lineno": line, "col": col,
                    ↪  "python": p[1]["python"]}
69              else:
70                  p.parser.warnings.append((line, col, f"{line}:{col}: <Warning>
                    ↪  Function '{func_name}' is already defined"))
71          else:
72              p.parser.warnings.append((line, col, f"{line}:{col}: <Warning> Function
                ↪  '{func_name}' is already defined"))
73      else:
```

```
74          p.parser.functions[func_name] = {"lineno": line, "col": col, "python":
            ↪  p[1]["python"]}
75          p.parser.newFunctions += [func_name]
76
77
78  def p_function_declaration(p):
79      """
80      function_declaration : DEFF IDENTIFIER LBRACE function_body RBRACE
81      """
82      line = p.lineno(1)
83      col = lexer.find_column(p.lexer.lexdata, lexpos=p.lexpos(1))
84      setLen = set()
85      setInput = set()
86      toBeRemoved = []
87      for l in p[4]:
88          lineL = l["lineno"]
89          colL = lexer.find_column(p.lexer.lexdata, lexpos=l["lexpos"])
90          setLen.add(len(l["input"]))
91          entry = CaseInput(l["input"])
92          if entry in setInput:
93              toBeRemoved.append(l)
94              p.parser.warnings.append((lineL, colL, f"{lineL}:{colL}: <Warning>
                ↪  Redundant input in pattern matching for function '{p[2]}'")),
95          else:
96              setInput.add(entry)
97
98      for l in toBeRemoved:
99          p[4].remove(l)
100
101     if len(setLen) > 1:
102         raise Exception(f"{line}:{col}: <Error> Equations for function '{p[2]}' have
            ↪  different number of arguments")
103
104     lenArgs = setLen.pop()
105     if lenArgs > 0:
106         sortedIn = sorted(list(setInput), reverse=True)
107         lista = map(lambda x: x.inputCase, sortedIn)
108         listaL = list(lista)
109         tree = verify.verify_group_by_level(listaL)
110         tree = verify.verify_fill(p[4], tree)
111         pythonString = verify.str_tree(tree, 0, lenArgs, "")
112     else:
113         pythonString = p[4][0]["statement"]
114
115     func_declare_string = "def " + p[2] + "("
116     for i in range(lenArgs):
117         if i != lenArgs - 1:
118             func_declare_string += "arg" + str(i) + ", "
```

```
119             else:
120                 func_declare_string += "arg" + str(i)
121         func_declare_string += "):"
122         full_function = func_declare_string + "\n\t" + re.sub("\n", "\n\t", pythonString)
        ↪  + "\n\n"
123         p[0] = {}
124         p[0]["func_name"] = p[2]
125         p[0]["python"] = full_function
126         p[0]["lineno"] = p.lineno(1)
127         p[0]["lexpos"] = p.lexpos(1)
128         p[0]["func_called"] = [item for d in p[4] for item in d["func_called"]]
129 
130 
131 def p_function_body(p):
132     """
133     function_body : case_statement SEMICOLON
134                   | case_statement SEMICOLON function_body
135     """
136     if len(p) == 3:
137         p[0] = [p[1]]
138     else:
139         p[0] = [p[1]] + p[3]
140 
141 
142 def p_case_statement(p):
143     """
144     case_statement : CASE case_input ASSIGN statement
145     """
146     varsUsed = p[4]["vars"]
147     for var in sorted(varsUsed, key=lambda x: x[1]):
148         varsInfo = [item for d in p[2] for item in d["infoVars"]]
149         varsIn = [t[-1] for t in varsInfo]
150         if var[2] not in varsIn:
151             inputText = p.lexer.lexdata
152             line = var[0]
153             col = lexer.find_column(inputText, lexpos=var[1])
154             raise Exception(f"{line}:{col}: <scope error> Variable '{var[2]}' not in
                ↪  scope")
155 
156     p[0] = {}
157     p[0]["statement"] = p[4]["python"]
158     p[0]["input"] = p[2]
159     p[0]["lineno"] = p.lineno(1)
160     p[0]["lexpos"] = p.lexpos(1)
161     p[0]["func_called"] = p[4]["func_called"]
162 
163 
164 def p_case_input(p):
```

```
    """
    case_input : LPAREN RPAREN
               | LPAREN case_arguments RPAREN
    """
    if len(p) == 3:
        p[0] = []
    else:
        p[0] = p[2]


def p_case_arguments(p):
    """
    case_arguments : case_argument
                   | case_argument COMMA case_arguments
    """
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        varsInfo = [item for d in p[3] for item in d["infoVars"]]
        varsIn = [t[-1] for t in varsInfo]
        for v in p[1]["infoVars"]:
            if v[2] in varsIn:
                for tup in sorted(varsInfo, key=lambda x: x[1]):
                    if tup[2] == v[2]:
                        result = tup
                        break
                inputText = p.lexer.lexdata
                line = result[0]
                col = lexer.find_column(inputText, lexpos=result[1])
                raise Exception(f"{line}:{col}: <scope error> Variable '{result[2]}'
                ↪  already in scope")
        p[0] = [p[1]] + p[3]


def p_case_argument(p):
    """
    case_argument : constant
                  | case_list
    """
    p[0] = p[1]


def p_constant(p):
    """
    constant : flo
             | int
             | bool
             | id
```

```
212     """
213     p[0] = p[1]
214     p[0]["infoVars"] = p[1]["vars"]
215
216
217 def p_case_list(p):
218     """
219     case_list : case_empty
220               | case_headtail
221     """
222     p[0] = p[1]
223
224
225 def p_case_empty(p):
226     """
227     case_empty : LSQUARE RSQUARE
228     """
229     p[0] = {}
230     p[0]["type"] = "list_empty"
231     p[0]["vars"] = []
232     p[0]["infoVars"] = []
233     p[0]["lineno"] = p.lineno(1)
234     p[0]["lexpos"] = p.lexpos(1)
235     p[0]["lastpos"] = p.lexpos(2)
236
237
238 def p_case_headtail(p):
239     """
240     case_headtail : IDENTIFIER COLON case_headtail2
241     """
242     varsIn = [t[-1] for t in p[3]["infoVars"]]
243     if p[1] in varsIn:
244         for tup in sorted(p[3]["infoVars"], key=lambda x: x[1]):
245             if tup[2] == p[1]:
246                 result = tup
247                 break
248         inputText = p.lexer.lexdata
249         line = result[0]
250         col = lexer.find_column(inputText, lexpos=result[1])
251         raise Exception(f"{line}:{col}: <scope error> Variable '{result[2]}' already
            ↪  in scope")
252
253     p[0] = {}
254     p[0]["type"] = "list_ht"
255     p[0]["vars"] = [p[1]] + p[3]["vars"]
256     p[0]["infoVars"] = [(p.lineno(1), p.lexpos(1), p[1])] + p[3]["infoVars"]
257
258
```

```
259 def p_case_headtail2(p):
260     """
261     case_headtail2 : case_headtailID
262                    | case_headtail
263     """
264     p[0] = p[1]
265
266
267 def p_case_headtailID(p):
268     """
269     case_headtailID : IDENTIFIER
270     """
271     p[0] = {}
272     p[0]["vars"] = [p[1]]
273     p[0]["infoVars"] = [(p.lineno(1), p.lexpos(1), p[1])]
274
275
276 def p_statement(p):
277     """
278     statement : IF expr THEN statement ELSE statement
279               | expr
280     """
281     if len(p) == 2:
282         p[0] = p[1]
283         p[0]["python"] = "return " + p[1]["python"]
284     else:
285         t = verify.verify_UNARY_BOOL_OP(p[2]["type"])
286         verify.verify_ERROR(t, p.lineno(1) + p.lineno(2),
            ↪  lexer.find_column(p.lexer.lexdata, lexpos=p[2]["lexpos"]),
287                             "boolean", p[2]["type"],
                                ↪  p.lexer.lexdata[p[2]["lexpos"]:p[2]["lastpos"]])
288         t = verify.verify_EQUALTYPE(p[4]["type"], p[6]["type"])
289         verify.verify_ERROR(t, p.lineno(1) + p.lineno(6),
            ↪  lexer.find_column(p.lexer.lexdata, lexpos=p[6]["lexpos"]),
290                             p[4]["type"], p[6]["type"],
                                ↪  p.lexer.lexdata[p[6]["lexpos"]:p[6]["lastpos"]])
291
292         p[0] = {}
293         p[0]["type"] = t
294         p[0]["python"] = "if " + re.sub("return ", "", p[2]["python"]) + ":\n\t" +
            ↪  re.sub(r'\n', '\n\t', p[4][
295             "python"]) + "\nelse:\n\t" + re.sub(r'\n', '\n\t', p[6]["python"])
296         p[0]["lexpos"] = p.lexpos(1)
297         p[0]["lineno"] = p.lineno(1)
298         p[0]["lastpos"] = p[6]["lastpos"]
299         p[0]["vars"] = p[2]["vars"] + p[4]["vars"] + p[6]["vars"]
300         p[0]["func_called"] =
            ↪  p[2]["func_called"]+p[4]["func_called"]+p[6]["func_called"]
```

```
301
302
303 def p_list(p):
304     """
305     list : LSQUARE RSQUARE
306          | LSQUARE list_elements RSQUARE
307     """
308     if len(p) == 3:
309         p[0] = {}
310         p[0]["type"] = "list_"
311         p[0]["python"] = "[]"
312         p[0]["lastpos"] = p.lexpos(2) + 1
313         p[0]["func_called"] = []
314         p[0]["vars"] = []
315     else:
316         p[0] = {}
317         p[0]["type"] = "list_" if p[2]["type"] == "any" else "list_" + p[2]["type"]
318         p[0]["python"] = "[" + p[2]["python"] + "]"
319         p[0]["lastpos"] = p.lexpos(3) + 1
320         p[0]["func_called"] = p[2]["func_called"]
321         p[0]["vars"] = p[2]["vars"]
322     p[0]["lexpos"] = p.lexpos(1)
323     p[0]["lineno"] = p.lineno(1)
324
325
326 def p_list_elements(p):
327     """
328     list_elements : expr
329                   | expr COMMA list_elements
330     """
331
332     if len(p) == 2:
333         p[0] = p[1]
334     else:
335         t = verify.verify_EQUALTYPE(p[1]["type"], p[3]["type"])
336         verify.verify_ERROR(t, p[1]["lineno"], lexer.find_column(p.lexer.lexdata,
            ↪  lexpos=p[1]["lexpos"]), p[3]["type"],
337                             p[1]["type"],
                                ↪  p.lexer.lexdata[p[1]["lexpos"]:p[1]["lastpos"]])
338
339         p[0] = {}
340         p[0]["type"] = t
341         p[0]["python"] = p[1]["python"] + ", " + p[3]["python"]
342         p[0]["lexpos"] = p[1]["lexpos"]
343         p[0]["lineno"] = p[1]["lineno"]
344         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
345         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
346
```

```
347
348 def p_expr(p):
349     """
350     expr : expr OR join
351          | join
352     """
353     if len(p) == 2:
354         p[0] = p[1]
355     else:
356         t = verify.verify_UNARY_BOOL_OP(p[1]["type"])
357         verify.verify_ERROR(t, p[1]["lineno"], lexer.find_column(p.lexer.lexdata,
        ↪  lexpos=p[1]["lexpos"]), "boolean",
358                             p[1]["type"],
                             ↪  p.lexer.lexdata[p[1]["lexpos"]:p[1]["lastpos"]])
359         t = verify.verify_UNARY_BOOL_OP(p[3]["type"])
360         verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
        ↪  lexpos=p[3]["lexpos"]), "boolean",
361                             p[3]["type"],
                             ↪  p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
362
363         p[0] = {}
364         p[0]["type"] = t
365         p[0]["python"] = p[1]["python"] + " or " + p[3]["python"]
366         p[0]["lexpos"] = p[1]["lexpos"]
367         p[0]["lineno"] = p[1]["lineno"]
368         p[0]["lastpos"] = p[3]["lastpos"]
369         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
370         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
371
372
373 def p_join(p):
374     """
375     join : join AND equality
376          | equality
377     """
378     if len(p) == 2:
379         p[0] = p[1]
380     else:
381         t = verify.verify_UNARY_BOOL_OP(p[1]["type"])
382         verify.verify_ERROR(t, p[1]["lineno"], lexer.find_column(p.lexer.lexdata,
        ↪  lexpos=p[1]["lexpos"]), "boolean",
383                             p[1]["type"],
                             ↪  p.lexer.lexdata[p[1]["lexpos"]:p[1]["lastpos"]])
384         t = verify.verify_UNARY_BOOL_OP(p[3]["type"])
385         verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
        ↪  lexpos=p[3]["lexpos"]), "boolean",
386                             p[3]["type"],
                             ↪  p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
```

```
387
388         p[0] = {}
389         p[0]["type"] = t
390         p[0]["python"] = p[1]["python"] + " and " + p[3]["python"]
391         p[0]["lexpos"] = p[1]["lexpos"]
392         p[0]["lineno"] = p[1]["lineno"]
393         p[0]["lastpos"] = p[3]["lastpos"]
394         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
395         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
396
397
398 def p_equality(p):
399     """
400     equality : equality EQ rel
401              | equality NE rel
402              | rel
403     """
404     if len(p) == 2:
405         p[0] = p[1]
406     else:
407         t = verify.verify_BIN_COMPARE_OP(p[1]["type"], p[3]["type"])
408         verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
            ↪  lexpos=p[3]["lexpos"]), p[1]["type"],
409                             p[3]["type"], p.lexer.lexdata[p[3]["lexpos"]:
                                ↪  p[3]["lastpos"]])
410
411         p[0] = {}
412         p[0]["type"] = t
413         p[0]["python"] = p[1]["python"] + " " + p[2] + " " + p[3]["python"]
414         p[0]["lexpos"] = p[1]["lexpos"]
415         p[0]["lineno"] = p[1]["lineno"]
416         p[0]["lastpos"] = p[3]["lastpos"]
417         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
418         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
419
420
421 def p_rel(p):
422     """
423     rel : listop LT listop
424         | listop GT listop
425         | listop LE listop
426         | listop GE listop
427         | listop
428     """
429     if len(p) == 2:
430         p[0] = p[1]
431     else:
432         t = verify.verify_BIN_COMPARE_OP(p[1]["type"], p[3]["type"])
```

```
433         verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
            ↪ lexpos=p[3]["lexpos"]), p[1]["type"],
434                             p[3]["type"], p.lexer.lexdata[p[3]["lexpos"]:
                                ↪ p[3]["lastpos"]])
435         p[0] = {}
436         p[0]["type"] = t
437         p[0]["python"] = p[1]["python"] + " " + p[2] + " " + p[3]["python"]
438         p[0]["lexpos"] = p[1]["lexpos"]
439         p[0]["lineno"] = p[1]["lineno"]
440         p[0]["lastpos"] = p[3]["lastpos"]
441         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
442         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
443
444
445 def p_listop(p):
446     """
447     listop : sum COLON listop
448            | sum CONCAT listop
449            | sum
450     """
451     if len(p) == 2:
452         p[0] = p[1]
453     else:
454         p[0] = {}
455         if p[2] == ':':
456             t = verify.verify_LIST(p[3]["type"])
457             verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
                ↪ lexpos=p[3]["lexpos"]), "list",
458                                 p[3]["type"],
                                    ↪ p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
459             t = verify.verify_COLON(p[1]["type"], p[3]["type"])
460             verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
                ↪ lexpos=p[3]["lexpos"]),
461                                 "list_" + p[1]["type"], p[3]["type"],
                                    ↪ p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
462             p[0]["python"] = "[" + p[1]["python"] + "]" + " + " + p[3]["python"]
463         else:
464             t = verify.verify_LIST(p[1]["type"])
465             verify.verify_ERROR(t, p[1]["lineno"], lexer.find_column(p.lexer.lexdata,
                ↪ lexpos=p[1]["lexpos"]), "list",
466                                 p[1]["type"],
                                    ↪ p.lexer.lexdata[p[1]["lexpos"]:p[1]["lastpos"]])
467             t = verify.verify_LIST(p[3]["type"])
468             verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
                ↪ lexpos=p[3]["lexpos"]), "list",
469                                 p[3]["type"],
                                    ↪ p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
470             t = verify.verify_CONCAT(p[1]["type"], p[3]["type"])
```

```
471             verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
                ↪  lexpos=p[3]["lexpos"]),
472                                 p[1]["type"], p[3]["type"],
                                    ↪  p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
473             p[0]["python"] = p[1]["python"] + " + " + p[3]["python"]
474
475         p[0]["type"] = t
476         p[0]["lexpos"] = p[1]["lexpos"]
477         p[0]["lineno"] = p[1]["lineno"]
478         p[0]["lastpos"] = p[3]["lastpos"]
479         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
480         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
481
482
483 def p_sum(p):
484     """
485     sum : sum PLUS term
486          | sum MINUS term
487          | term
488     """
489     if len(p) == 2:
490         p[0] = p[1]
491     else:
492         t = verify.verify_UNARY_NUM_OP(p[1]["type"])
493         verify.verify_ERROR(t, p[1]["lineno"], lexer.find_column(p.lexer.lexdata,
            ↪  lexpos=p[1]["lexpos"]), "num",
494                             p[1]["type"],
                                ↪  p.lexer.lexdata[p[1]["lexpos"]:p[1]["lastpos"]])
495         t = verify.verify_UNARY_NUM_OP(p[3]["type"])
496         verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
            ↪  lexpos=p[3]["lexpos"]), "num",
497                             p[3]["type"],
                                ↪  p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
498
499         p[0] = {}
500         p[0]["type"] = t
501         p[0]["python"] = p[1]["python"] + " " + p[2] + " " + p[3]["python"]
502         p[0]["lexpos"] = p[1]["lexpos"]
503         p[0]["lineno"] = p[1]["lineno"]
504         p[0]["lastpos"] = p[3]["lastpos"]
505         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
506         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
507
508
509 def p_term(p):
510     """
511     term : term MULT exponential
512          | term DIV exponential
```

```
513           | term FLOORDIV exponential
514           | term MOD exponential
515           | exponential
516       """
517       if len(p) == 2:
518           p[0] = p[1]
519       else:
520           t = verify.verify_UNARY_NUM_OP(p[1]["type"])
521           verify.verify_ERROR(t, p[1]["lineno"], lexer.find_column(p.lexer.lexdata,
          ↪  lexpos=p[1]["lexpos"]), "num",
522                               p[1]["type"],
                                ↪  p.lexer.lexdata[p[1]["lexpos"]:p[1]["lastpos"]])
523           t = verify.verify_UNARY_NUM_OP(p[3]["type"])
524           verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
          ↪  lexpos=p[3]["lexpos"]), "num",
525                               p[3]["type"],
                                ↪  p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
526
527           p[0] = {}
528           p[0]["type"] = t
529           p[0]["python"] = p[1]["python"] + " " + p[2] + " " + p[3]["python"]
530           p[0]["lexpos"] = p[1]["lexpos"]
531           p[0]["lineno"] = p[1]["lineno"]
532           p[0]["lastpos"] = p[3]["lastpos"]
533           p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
534           p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
535
536
537   def p_exponential(p):
538       """
539       exponential : exponential POWER unary
540                   | unary
541       """
542       if len(p) == 2:
543           p[0] = p[1]
544       else:
545           t = verify.verify_UNARY_NUM_OP(p[1]["type"])
546           verify.verify_ERROR(t, p[1]["lineno"], lexer.find_column(p.lexer.lexdata,
          ↪  lexpos=p[1]["lexpos"]), "num",
547                               p[1]["type"],
                                ↪  p.lexer.lexdata[p[1]["lexpos"]:p[1]["lastpos"]])
548           t = verify.verify_UNARY_NUM_OP(p[3]["type"])
549           verify.verify_ERROR(t, p[3]["lineno"], lexer.find_column(p.lexer.lexdata,
          ↪  lexpos=p[3]["lexpos"]), "num",
550                               p[3]["type"],
                                ↪  p.lexer.lexdata[p[3]["lexpos"]:p[3]["lastpos"]])
551
552           p[0] = {}
```

```
553         p[0]["type"] = t
554         p[0]["python"] = p[1]["python"] + " " + p[2] + " " + p[3]["python"]
555         p[0]["lexpos"] = p[1]["lexpos"]
556         p[0]["lineno"] = p[1]["lineno"]
557         p[0]["lastpos"] = p[3]["lastpos"]
558         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
559         p[0]["func_called"] = p[1]["func_called"]+p[3]["func_called"]
560
561
562 def p_unary(p):
563     """
564     unary : NOT unary
565           | MINUS unary
566           | PLUS unary
567           | factor
568     """
569     if len(p) == 2:
570         p[0] = p[1]
571     else:
572         if p[1] == '!':
573             t = verify.verify_UNARY_BOOL_OP(p[2]["type"])
574             p[1] = 'not'
575             verify.verify_ERROR(t, p[2]["lineno"], lexer.find_column(p.lexer.lexdata,
                ↪  lexpos=p[2]["lexpos"]), "boolean",
576                                 p[2]["type"],
                                    ↪  p.lexer.lexdata[p.lexpos(1):p[2]["lastpos"]])
577         else:
578             t = verify.verify_UNARY_NUM_OP(p[2]["type"])
579             verify.verify_ERROR(t, p[2]["lineno"], lexer.find_column(p.lexer.lexdata,
                ↪  lexpos=p[2]["lexpos"]), "num",
580                                 p[2]["type"],
                                    ↪  p.lexer.lexdata[p.lexpos(1):p[2]["lastpos"]])
581         p[0] = {}
582         p[0]["type"] = t
583         p[0]["python"] = p[1] + " " + p[2]["python"]
584         p[0]["lexpos"] = p.lexpos(1)
585         p[0]["lineno"] = p.lineno(1)
586         p[0]["lastpos"] = p[2]["lastpos"]
587         p[0]["vars"] = p[2]["vars"]
588         p[0]["func_called"] = p[2]["func_called"]
589
590
591 def p_factor(p):
592     """
593     factor : LPAREN expr RPAREN
594            | id
595            | function_call
596            | int
```

```
597             | flo
598             | bool
599             | list
600     """
601     if len(p) > 2:
602         p[0] = p[2]
603         p[0]["python"] = "(" + p[2]["python"] + ")"
604         p[0]["lexpos"] = p.lexpos(1)
605         p[0]["lineno"] = p.lineno(1)
606         p[0]["lastpos"] = p.lexpos(3)
607     else:
608         p[0] = p[1]
609 
610 
611 def p_int(p):
612     """
613     int : INTEGER
614     """
615     p[0] = {}
616     p[0]["type"] = "num"
617     p[0]["python"] = p[1]
618     p[0]["lexpos"] = p.lexpos(1)
619     p[0]["lineno"] = p.lineno(1)
620     p[0]["lastpos"] = p.lexpos(1) + len(p[1])
621     p[0]["func_called"] = []
622     p[0]["vars"] = []
623 
624 
625 def p_flo(p):
626     """
627     flo : FLOAT
628     """
629     p[0] = {}
630     p[0]["type"] = "num"
631     p[0]["python"] = p[1]
632     p[0]["lexpos"] = p.lexpos(1)
633     p[0]["lineno"] = p.lineno(1)
634     p[0]["lastpos"] = p.lexpos(1) + len(p[1])
635     p[0]["func_called"] = []
636     p[0]["vars"] = []
637 
638 
639 def p_bool(p):
640     """
641     bool : BOOLEAN
642     """
643     p[0] = {}
644     p[0]["type"] = "boolean"
```

```
645     p[0]["python"] = p[1]
646     p[0]["lexpos"] = p.lexpos(1)
647     p[0]["lineno"] = p.lineno(1)
648     p[0]["lastpos"] = p.lexpos(1) + len(p[1])
649     p[0]["func_called"] = []
650     p[0]["vars"] = []
651
652
653 def p_id(p):
654     """
655     id : IDENTIFIER
656     """
657     p[0] = {}
658     p[0]["type"] = "any"
659     p[0]["python"] = p[1]
660     p[0]["lexpos"] = p.lexpos(1)
661     p[0]["lineno"] = p.lineno(1)
662     p[0]["lastpos"] = p.lexpos(1) + len(p[1])
663     p[0]["func_called"] = []
664     p[0]["vars"] = [(p[0]["lineno"], p[0]["lexpos"], p[1])]
665
666
667 def p_function_composition(p):
668     """
669     function_composition : id
670                          | id PERIOD function_composition
671     """
672     p[0] = {}
673     p[0]["lexpos"] = p[1]["lexpos"]
674     p[0]["lineno"] = p[1]["lineno"]
675     p[0]["func_called"] = [(p[1]["lineno"], p[1]["lexpos"], p[1]["python"])]
676     if len(p) == 2:
677         p[0]["python"] = p[1]["python"]
678     else:
679         p[0]["python"] = p[1]["python"] + "(" + p[3]["python"] + ")"
680         p[0]["func_called"] += p[3]["func_called"]
681
682
683 def p_function_call(p):
684     """
685     function_call : function_composition LPAREN function_arguments RPAREN
686                   | function_composition LPAREN RPAREN
687     """
688     p[0] = {}
689     p[0]["type"] = "any"
690     if len(p) == 4:
691         p[0]["python"] = p[1]["python"] + "()"
692         p[0]["lastpos"] = p.lexpos(3) + 1
```

```
693         p[0]["vars"] = []
694     else:
695         p[0]["python"] = p[1]["python"] + "(" + p[3]["python"] + ")" if
            ↪  p[1]["python"][-1] != ")" else p[1]["python"][:-1] + "(" + p[3]["python"]
            ↪  + "))"
696         p[0]["lastpos"] = p.lexpos(4) + 1
697         p[0]["vars"] = p[3]["vars"]
698     p[0]["lexpos"] = p[1]["lexpos"]
699     p[0]["lineno"] = p[1]["lineno"]
700     p[0]["func_called"] = p[1]["func_called"]
701
702
703 def p_function_arguments(p):
704     """
705     function_arguments : expr
706                        | expr COMMA function_arguments
707     """
708     if len(p) == 2:
709         p[0] = p[1]
710     else:
711         p[0] = {}
712         p[0]["python"] = p[1]["python"] + ", " + p[3]["python"]
713         p[0]["lexpos"] = p[1]["lexpos"]
714         p[0]["lineno"] = p[1]["lineno"]
715         p[0]["vars"] = p[1]["vars"] + p[3]["vars"]
716         p[0]["func_called"] = p[1]["func_called"] + p[3]["func_called"]
717
718
719 def p_error(p):
720     column_number = lexer.find_column(p.lexer.lexdata, p)
721     if p:
722         raise Exception(f"{p.lineno}:{column_number}: <parse error> Unexpected token
            ↪  '{p.value}'")
723     else:
724         raise Exception(f"{p.lineno}:{column_number}: <parse error> Unexpected end of
            ↪  input")
725
726
727 parser = yacc.yacc()
728 parser.functions = {}
729 parser.warnings = []
730 parser.newFunctions = []
```

Apêndice C

Compilador FPY

```
from parserGrammar import parser
from parserGrammar import lexer
import re
import sys


def repl_func(match, data):
    matched_str = match.group(0)
    start_index = match.start()
    line_num = data.count('\n', 0, start_index) + 1
    lexer.lexer.lineno = line_num
    parser.newFunctions = []
    try:
        ret_str = parser.parse(matched_str)
    except Exception as e:
        print(e, file=sys.stderr)
        sys.exit(1)

    return ret_str


erFPY = re.compile(r'"""FPY.+?"""', re.DOTALL)

if len(sys.argv) > 1:
    filename = sys.argv[1]
    if re.search(r'\.py$', filename):
        try:
            file = open(filename, 'r')
        except Exception as e:
            print(e, file=sys.stderr)
            sys.exit(1)
        lines = file.readlines()
        data = ''.join(lines)
        texto = erFPY.sub(lambda match: repl_func(match, data), data)
```

```
35          outputFile = open(filename[:-3] + "FPY.py", 'w')
36          outputFile.write(texto)
37          warnings = sorted_list = sorted(parser.warnings, key=lambda x: (x[0], x[1]))
38          for w in warnings:
39              print(w[2], file=sys.stderr)
40      else:
41          print(f"File '{filename}' is not valid", file=sys.stderr)
42  else:
43      print(f"File not specificated", file=sys.stderr)
```

Apêndice D

Estruturas Yacc

D.1 Statements

Campos do statement:

- ***type***: tipo da produção;
- ***python***: tradução da produção para *Python*;
- ***lexpos***: posição do primeiro carácter da produção;
- ***lineno***: linha onde começa a produção;
- ***lastpos***: posição do ultimo carácter da produção;
- ***func_called***: lista de funções que foram chamadas na produção. A lista contém tuplos no formato (número da linha,posição do primeiro carácter,nome da função);
- ***vars***: lista de variáveis utilizadas na produção. Esta lista contém tuplos no formato (número da linha,posição do primeiro carácter,identificador da variável).

Tipos de dados possiveis num *statement*:

- ***num***: inteiros e floats;
- ***boolean***: booleanos;
- ***any***: identificadores de variáveis e chamadas de funções;
- ***list_(X)***: lista de elementos do tipo X;
- ***list_***: lista vazia ou lista com elementos do tipo *any*.

D.2 Case statement

Campos do case statement:

- ***statement***: tradução da sua atribuição para *Python*;

- ***lexpos***: posição do primeiro carácter da produção;
- ***lineno***: linha onde começa a produção;
- ***lastpos***: posição do ultimo carácter da produção;
- ***func_called***: lista de funções que foram chamadas na sua atribuição;
- ***input***: Estrutura proveniente dos seus argumentos.

D.3 Função

Campos da função:

- ***python***: tradução da função para *Python*;
- ***func_name***: nome da função;
- ***lineno***: linha onde começa a produção;
- ***lexpos***: posição do primeiro carácter da produção;
- ***func_called***: lista de funções que foram chamadas nas suas atribuições.

Apêndice E

Erros

E.1 Formato

```
numero_linha:numero_coluna: <tipo_erro> mensagem_erro
```

E.2 Possíveis Erros

E.2.1 Erros Léxicos

- Token reservado em *Python*:
  `f"{linha}:{coluna}: <lexer error> Reserved python token '{caracter de erro}'"`
- Caracter ilegal:
  `f"{linha}:{coluna}: <lexer error> Illegal character '{caracter de erro}'"`

E.2.2 Erros Sintáticos

- Token inesperado:
  `f"{linha}:{coluna}: <parse error> Unexpected token '{token}'"`
- Fim de input inesperado:
  `f"{linha}:{coluna}: <parse error> Unexpected end of input"`

E.2.3 Erros Semânticos

- Função não definida:
  `f"{linha}:{coluna}: <scope error> Function '{nome da função}' not in scope"`
- `f"{linha}:{Python}: <Error> Equations for function '{nome da função}' have different number of arguments"`
- Variável não definida:
  `f"{linha}:{coluna}: <scope error> Variable '{nome da variável}' not in scope"`
- Variável já definida:
  `f"{linha}:{coluna}: <scope error> Variable '{nome da variável}' already in scope"`

- Incompatibilidade do tipo da expressão com o tipo esperado:
  `f"{linha}:{coluna}: <type error> Couldn't match expected type '{tipo esperado}' with actual type '{tipo recebido}', in expression '{expressão}'"`

Apêndice F

Warning's

- Função já definida:
  `f"{linha}:{coluna}: <Warning> Function '{nome da função}' is already defined"`
- Input redundante:
  `f"{linha}:{coluna}: <Warning> Redundant input in pattern matching for function '{nome da função}'"`

Bibliografia

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Pearson Education, Inc, 2006.

[BA20] José Bernardo Barros and José João Almeida. *Reconhecedores Sintáticos*. 2020. `https://natura.di.uminho.pt/~jj/pl-20/aulas/linguagens.pdf`.

[Bea] David M. Beazley. PLY (Python Lex-Yacc). `https://www.dabeaz.com/ply/ply.html`.