Dr.

UNIVERSIDADE FEDERAL DE OURO PRETO

decom
departamento de computação

UMA ABORDAGEM BASEADA EM PARSING EXPRESSION GRAMMARS PARA CASAMENTO DE PADRÃO EM CÓDIGO-FONTE MULTI-LINGUAGEM.

Guilherme Augusto Anício Drummond do Nascimento

Orientador: Rodrigo Geraldo Ribeiro

Ouro Preto

Julho de 2025

UNIVERSIDADE FEDERAL DE OURO PRETO

UMA ABORDAGEM BASEADA EM PARSING EXPRESSION GRAMMARS PARA CASAMENTO DE PADRÃO EM CÓDIGO-FONTE MULTI-LINGUAGEM.

Guilherme Augusto Anício Drummond do Nascimento

Exame de Qualificação de Mestrado apresentado ao Programa de Pós-graduação em Ciência da Computação, da Universidade Federal de Ouro Preto, como parte dos requisitos necessários à obtenção do título de Mestre em Ciência da Computação.

Orientador: Rodrigo Geraldo Ribeiro

Ouro Preto

Julho de 2025

Nós somos uma maneira do
cosmos conhecer a si mesmo.
Carl Sagan

Agradecimentos

—

Resumo do Exame de Qualificação apresentado à UFOP como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA ABORDAGEM BASEADA EM PARSING EXPRESSION GRAMMARS PARA CASAMENTO DE PADRÃO EM CÓDIGO-FONTE MULTI-LINGUAGEM.

Guilherme Augusto Anício Drummond do Nascimento

Julho/2025

Orientador: Rodrigo Geraldo Ribeiro

Programa: Ciência da Computação

A PARSING EXPRESSION GRAMMARS-BASED APPROACH FOR PATTERN MATCHING IN MULTI-LANGUAGE SOURCE CODE.

Guilherme Augusto Anício Drummond do Nascimento

July/2025

Advisor: Rodrigo Geraldo Ribeiro

Department: Computer Science

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introduction

Pattern matching is the act of checking a given sequence fo tokens for the presence of the constituents of some pattern. The match usually must be exact: "either it will or will not be a match". It is frequently used to output the locations (if any) of a pattern within a token sequence, output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (i.e., search and replace). Patterns generally have the form of either sequences or tree structures. Often, patterns sequences are described using regular expressions.

1.1   Objectives

The main objective of this work is to formalize the semantics of pattern matching in syntax trees. Specifically, we plan to:

1. Define the semantics for generating a parse tree when executing a parsing expression.

2. Define the semantics for pattern matching on a parse tree.

3. Prove properties of the defined semantics.

4. ...

1.2   Contributions

Our contributions are:

- A type system and operational semantics for generatin a parse tree.

- ...

1.3   Dissertation Structure

The rest of this dissertation is structured as follow: Chapter 2 covers the necessary background knowledge used in this work, Chapter 3 presents the pattern matching and generation of parse tree, Chapter 4 discusses some case studies using the proposed approach, Chapter 5 presents the schedule of next steps, and finally Chapter 6 concludes this work. The code for the parsing and pattern match of parse trees can be found on `https://github.com/guinasc2/ast-pattern-matching`.

Capítulo 2

Background

This chapter presents fundamental concepts for understanding and development of the work. Section 2.1 reviews concepts related to type systems and Section 2.2 reviews concepts related to operational semantics. Section 2.3 formally introduces PEGs and their operational semantics. Related work is discussed in Section 2.4. Section 2.5 concludes the chapter.

2.1 Type systems

2.2 Operational semantics

2.3 An Overview of PEGs

Intuitively, PEGs are a formalism for describing top-down parsers. Formally, a PEG is a 4-tuple (V, Σ, R, e_S), where V is a finite set of variables, Σ is the alphabet, R is the finite set of rules, and e_S is the start expression. Each rule $r \in R$ is a pair (A, e), usually written $A \leftarrow e$, where $A \in V$ and e is a parsing expression. We let the meta-variable a denote an arbitrary alphabet symbol, A a variable and e a parsing expression. Following common practice, all meta-variables can appear primed or sub-scripted. The following context-free grammar defines the syntax of a parsing expression:

$$e \rightarrow \epsilon \mid a \mid A \mid e_1\, e_2 \mid e_1\, /\, e_2 \mid e^{\star} \mid !\, e$$

The execution of parsing expressions is defined by an inductively defined judgment that relates pairs formed by a parsing expression and an input string to pairs formed by the consumed prefix and the remaining string. Notation $(e, s) \Rightarrow_G (s_p, s_r)$ denote that parsing expression e consumes the prefix s_p from the input string s leaving the suffix s_r. The notation $(e, s) \Rightarrow_G \bot$ denote the fact that s cannot be parsed by e. We let meta-variable r denote an arbitrary parsing result, i.e., either r is a pair

3

(s_p, s_r) or $\perp$. We say that an expression e fails if its execution over an input produces $\perp$; otherwise, it succeeds. Figure 2.1 defines the PEG semantics. We comment on

$$\frac{}{(\epsilon, s) \Rightarrow_G (\epsilon, s)} \ {}^{\{Eps\}} \qquad \frac{}{(a, as_r) \Rightarrow_G (a, s_r)} \ {}^{\{ChrS\}} \qquad \frac{a \neq b}{(a, bs_r) \Rightarrow_G \perp} \ {}^{\{ChrF\}} \qquad \frac{A \leftarrow e \in R}{(A, s)}$$

$$\frac{(e_1, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1}, s_{p_2} s_r) \quad (e_2, s_{p_2} s_r) \Rightarrow_G (s_{p_2}, s_r)}{(e_1 e_2, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1} s_{p_2}, s_r)} \ {}^{\{Cat_{S1}\}} \qquad \frac{(e_1, s_p s_r) \Rightarrow_G (s_p, s_r) \quad (e_2, s_r) \Rightarrow}{(e_1 e_2, s_p s_r) \Rightarrow_G \perp}$$

$$\frac{(e_1, s) \Rightarrow_G \perp}{(e_1 e_2, s) \Rightarrow_G \perp} \ {}^{\{Cat_{F1}\}} \qquad \frac{(e_1, s_p s_r) \Rightarrow_G (s_p, s_r)}{(e_1 / e_2, s_p s_r) \Rightarrow_G (s_p, s_r)} \ {}^{\{Alt_{S1}\}} \qquad \frac{(e_1, s_p s_r) \Rightarrow_G \perp \quad (e_2, s_p s_r) \Rightarrow}{(e_1 / e_2, s_p s_r) \Rightarrow_G r}$$

$$\frac{(e, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1}, s_{p_2} s_r) \quad (e^\star, s_{p_2} s_r) \Rightarrow_G (s_{p_2}, s_r)}{(e^\star, s_{p_1} s_{p_2} s_r) \Rightarrow_G (s_{p_1} s_{p_2}, s_r)} \ {}^{\{Star_{rec}\}} \qquad \frac{(e, s) \Rightarrow_G \perp}{(e^\star, s) \Rightarrow_G (\epsilon, s)} \ {}^{\{Star_{er}\}}$$

$$\frac{(e, s_p s_r) \Rightarrow_G (s_p, s_r)}{(!\,e, s_p s_r) \Rightarrow_G \perp} \ {}^{\{Not_F\}} \qquad \frac{(e, s) \Rightarrow_G \perp}{(!\,e, s) \Rightarrow_G (\epsilon, s)} \ {}^{\{Not_S\}} \qquad \frac{}{(a, \epsilon) \Rightarrow}$$

Figura 2.1: Parsing expressions operational semantics.

some rules of the semantics. Rule $_{Eps}$ specifies that expression ϵ will not fail on any input s by leaving it unchanged. Rule $_{ChrS}$ specifies that an expression a consumes the first character when the input string starts with an 'a' and rule $_{ChrF}$ shows that it fails when the input starts with a different symbol. Rule $_{Var}$ parses the input using the expression associated with the variable in the grammar G. When parsing a sequence expression, $e_1 e_2$, the result is formed by e_1 and e_2 parsed prefixes and the remaining input is given by e_2. Rules $_{Cat_{F1}}$ and $_{Cat_{F2}}$ say that if e_1 or e_2 fail, then the whole expression fails. The rules for choice impose that we only try expression e_2 in e_1/e_2 when e_1 fails. Parsing a star expression $e^\star$ consists in repeatedly execute e on the input string. When e fails, $e^\star$ succeeds without consuming any symbol of the input string. Finally, the rules for the not predicate expression, $!\,e$, specify that whenever the expression e succeeds on input s, $!\,e$ fails; and when e fails on s we have that $!\,e$ succeeds without consuming any input.

Example 1. As an example, consider the PEG (Figure 2.2) which recognizes mathematical formulas that apply the basic four operations to non-negative integers:

$$
\begin{aligned}
E &\leftarrow \text{T ('+' T)*} \\
T &\leftarrow \text{F ('*' F)*} \\
F &\leftarrow \text{[0-9]+ / '(' Expr ')'}
\end{aligned}
$$

Figura 2.2: PEG for mathematical formulas.

4

– **Montar árvore de derivação da semântica para 1+2** − − **Eu troco [0-9]+ pra n na PEG e deixo como n na árvore? Ou troco pra [0-9]+ na árvore e continuo a derivação?** −

Consider the string `1+2`. The initial rule `E`, first tries to parse a `T` that will, in turn, first try to consume a `F`, which recognizes the number '1'. Since the `T` rule does not consume a '*', it returns back to `E`. It then finds the '+' operator and tries to parse another `T`, which will consume the 2 as a `F` and goes back to `E`, that does not find another '+' operator and finalizes the parsing process. The derivation tree for the string `1+2` can be seen on Figure 2.3.

2.4   Related work

Atkinson and Griswold [1] presents the matching tool TAWK, which extends extend the pattern syntax of AWK to support matching of abstract syntax trees. In TAWK, pattern syntax is language-independent, based on abstract tree patterns, and each pattern can have associated actions, which are written in C for generality, familirity and performance. An example of extracting a call graph from a given code is presented throughout the paper, providing examples in different pattern matching tools, including TAWK, and a scorecard for each, detailing four characteristics: matching power, programming power, speed and robustness.

Kopell et al. [2] presents an approach for building source-to-source transformation that can run on multiple programming languages, based on a representation called incremental parametric syntax (IPS). In IPS, languages are represented using a mixture of language-specific and generic parts. Transformations deal only with the generic fragments, but the implementer starts with a pre-existing normal syntax definition, and only does enough up-front work to redefine a small fraction of a language in terms of these generic fragments. The IPS was implemented in a Haskell framework called *Cubix*, and currently supports C, Java, JavaScript, Lua, and Python[1]. They also demonstrate a whole-program refactoring for threading variables through chains of function calls and three smaller source-to-source transformations, being a hoisting transformation, a test-coverage transformation and a the three-address code transformation.

Premtoon et al. [3] presents *Yogo*, a tool that uses an approach to semantic code search based on equational reasoning, that considers not only the dataflow graph of a function, but also the dataflow graphs of all equivalents functions reachable via a set of rewrite rules [3]. The tool is capable of recognizing differents variations of the same operation and also when code is an instance of a higher-level concept.

---

[1]See `https://github.com/cubix-framework/cubix`

$$
\cfrac{
\cfrac{
\cfrac{\overline{\langle n,1+2\rangle \Rightarrow_G \langle 1,+2\rangle}}{\langle n/'('E')',1+2\rangle \Rightarrow_G \langle 1,+2\rangle}}{\langle F,1+2\rangle}
\qquad
\cfrac{\cfrac{\overline{\langle '*'F,+2\rangle \Rightarrow_G \langle \perp,+2\rangle}}{\langle ('*'F)^\star,+2\rangle \Rightarrow_G \langle \epsilon,+2\rangle}}{}
}{
\cfrac{\langle F('*'F)^\star,1+2\rangle \Rightarrow_G \langle 1,+2\rangle}{\langle T,1+2\rangle \Rightarrow_G \langle 1,+2\rangle}}
$$

$$
\cfrac{
\overline{\langle '+',+2\rangle \Rightarrow_G \langle +,2\rangle}
\qquad
\cfrac{
\cfrac{
\cfrac{\overline{\langle n,2\rangle \Rightarrow_G \langle 2,\epsilon\rangle}}{\langle n/'('E')',2\rangle \Rightarrow_G \langle 2,\epsilon\rangle}}{\langle F,2\rangle \Rightarrow_G \langle 2,\epsilon\rangle}
\quad
\cfrac{\overline{\langle '*'F,\epsilon\rangle \Rightarrow_G \langle \perp,\epsilon\rangle}}{\langle ('*'F)^\star,\epsilon\rangle \Rightarrow_G \langle \epsilon,\epsilon\rangle}
}{
\cfrac{\langle F('*'F)^\star,2\rangle \Rightarrow_G \langle 2,\epsilon\rangle}{\langle T,2\rangle \Rightarrow_G \langle 2,\epsilon\rangle}}
\qquad
\cfrac{\overline{\langle '+'T,\epsilon\rangle \Rightarrow_G \langle \perp,\epsilon\rangle}}{\langle ('+'T)^\star,\epsilon\rangle \Rightarrow_G \langle \epsilon,\epsilon\rangle}
}{
\cfrac{\langle '+'T,+2\rangle \Rightarrow_G \langle +2,\epsilon\rangle}{\langle ('+'T)^\star,+2\rangle \Rightarrow_G \langle +2,\epsilon\rangle}}
$$

$$
\cfrac{\langle T('+'T)^\star,1+2\rangle \Rightarrow_G \langle 1+2,\epsilon\rangle}{\langle E,1+2\rangle \Rightarrow_G \langle 1+2,\epsilon\rangle}
$$

Figura 2.3: Semantic derivation for expression '1+2*3'

Yogo is built on the *Cubix* infraestructure and can find equivalent code in multiple languages from a single query.

Silva et al. [4] proposes *RefDiff 2.0*, a multi-language refactoring detection tool. Their approach introduces a refactoring detection algorithm that relies on the Code Structure Tree (CST), a representation of the source code that abstract away the specificities of particular programming languages. The tool has results that are on par with state-of-the-art refactoring detection approaches specialized in the Java language. It also has support for two other popular programming languages: JavaScript and C. This demonstrates that the tool can be a viable alternative for multi-language refactoring research and in practical applications of refactoring detection.

van Tonder and Le Goues [5] proposes that the problem of automatically transforming programs can be decomposed by having a common grammar and open extension points. The common grammar expresses the central context-free language (CFL) properties shared by many contemporary languages, while the open extensions points allow customizing syntax and hooks in smaller parsers to handle language-specific syntax, such as comments. The decomposition is made using a Parser Parser combinator (PPC), a mechanism that generates parsers for matching syntactic fragments in source code by parsing declarative user-supplied templates. This allows to detach from translating input programs to any particular abstract syntax tree representation, and lifts syntax rewriting to a modularly-defined parsing problem. They also evaluate *Comby*, an implementation of the approach process using PPC, on a large scale multi-language rewriting across 12 languages, and validated effectiveness of the approach by producing correct and desirable lightweight transformations on popular real-world projects.

Matute et al. [6] proposes a search architecture that relies only on tokenizing a query, introducing a new language and matching algorithm to support tree-aware wildcards by building on tree automata. They also present *stsearch*, a syntactic search tool leveraging their approach, which supports syntactic search even for previously unparsable queries.

Ierusalimschy [7] proposes the use of PEGs as a basis for text pattern-mathing and presents LPEG, a text pattern-matching tool based on PEGs for the Lua scripting language, and a Parsing Machine that allows a small and efficient implementation of PEGs for pattern matching. This allow LPEG to have both the expressive power of PEGs with the ease of use of regular expressions. LPEG also seems specially suited for languages that are too complex for traditional pattern-matching tools but do not need a complex yacc-lex implementation, like domain-specific languages such as SQL and regular expressions, and even XML.

Of the approaches found in the literature, only LPEG uses the concept of PEGs,

trying to combine their expressive capacity with the ease of use of regular expressions, but it focuses on text matching. The approach presented in this work uses PEGs to generate a parse tree and perform the matching (and rewriting) of these trees.

2.5   Conclusion

This chapter presents the theoretical framework for this work: initially, concepts about type systems and operational semantics were reviewed. Next, the concept of PEGs and their operational semantics were presented. Finally, works related to source code analysis and pattern matching were presented.

Capítulo 3

Methodology

This chapter presents the formalisms developed during the work, as well as the current state of the developed library. Section 3.1 details the execution of a PEG for generating a parse tree and how pattern matching works. Section 3.2 discusses some implementation details. Section 3.3 concludes the chapter.

3.1 Parse trees

Let $G = (V, \Sigma, R, e_s)$ be an arbitrary PEG, the meta-variable $a \in \Sigma$ an arbitrary alphabet symbol, $A \in V$ a variable and e a parsing expression. The following context-free grammar defines the syntax of a parse tree:

$$t \quad \rightarrow \quad \hat{\epsilon} \mid \hat{a} \mid \hat{A} \mid \langle t_1, t_2 \rangle \mid L\,t \mid R\,t \mid [t] \mid \eta$$

Where $\hat{\epsilon}$ represents that a parsing expression resulted in success without consuming any symbol of its input, $\hat{a}$ represents that the parsing expression consumed the symbol a from the input, $\hat{A}$ represents that the parsing of the rule $(A, e) \in R$ was succeesful, $\langle t_1, t_2 \rangle$ represents that a sequence of parsing expressions succeeded, $L\,t$ and $R\,t$ both represent that a branch of an ordered choice succeeded, with $L\,t$ for the left one and $R\,t$ for the right one, $[t]$ is a list of trees and η represents that a not predicate was successful.

Executing a parsing expressions produces a parsing tree and is defined by an inductively defined judgment that relates pairs formed by a parsing expression and an input string to pairs formed by the generated tree and the remaining string. Notation $(e, s_p s_r) \Rightarrow_G (t, s_r)$ denote that parsing expression e consumes the prefix s_p and generates the parse tree t from the input string $s_p s_r$ leaving the suffix s_r. The notation $(e, s) \Rightarrow_G \bot$ denote the fact that s cannot be parsed by e. We let meta-variable r denote an arbitrary parsing result, i.e., either r is a pair (t, s_r) or $\bot$. We say that an expression e fails if its execution over an input produces $\bot$;

otherwise, it succeeds. Figure **??** defines the PEG semantics for tree generation.

Definition 1 (Type of a parse tree)**.** We say that a parse tree t has type e, $t : e$, when t is generated by a parsing expression e, i.e., when $(e, s_p s_r) \Rightarrow_G (t, s_r)$.

Definition 2 (Identified pattern)**.** An identified pattern is a pair (i, p) where i is an identifier and p is a pattern.

Let Θ a finite set of identified patterns, U a finite set of variables and $A \in V$ a variable. Each identified pattern $p_i \in \Theta$ is a pair (I, p), where $I \in U$ and p is a pattern. The following context-free grammar defines the syntax of a pattern:

$$p \quad \rightarrow \quad \epsilon \mid a \mid A \mid p_1 \, p_2 \mid p_1 \, / \, p_2 \mid p^\star \mid ! \, p \mid M \mid I$$

Where ϵ is a pattern that matches with the tree of empty string $(\hat{\epsilon})$, a matches only with the tree of the symbol a $(\hat{a})$, A matches with a subtree of type e and $(A, e) \in R$, $p_1 \, p_2$ matches if both p_1 and p_2 matches sequentially. $p_1 \, / \, p_2$ matches if one of p_1 or p_2 matches, $p^\star$ will try to match p sequentially as many times as possible, $!p$ matches only if p does not matches. M is a meta-variable that, given a variable A, matches with any tree $t : e$ where $(A, e) \in R$ and I is a reference to another pattern p' where $(I, p') \in \Theta$.

Figure 3.2 defines the pattern semantics.

$$\frac{}{\Theta, G \vdash \epsilon} \; {}^{\{Eps\}} \qquad\qquad \frac{}{\Theta, G \vdash a} \; {}^{\{ChrS\}} \qquad\qquad \frac{A \in V}{\Theta, G \vdash A} \; {}^{\{Var\}}$$

$$\frac{\Theta, G \vdash p_1 \quad \Theta, G \vdash p_2}{\Theta, G \vdash p_1 \, p_2} \; {}^{\{Sequence\}} \qquad \frac{\Theta, G \vdash p_1 \quad \Theta, G \vdash p_2}{\Theta, G \vdash p_1 \, / \, p_2} \; {}^{\{Choice\}} \qquad \frac{\Theta, G \vdash p}{\Theta, G \vdash p^\star} \; {}^{\{Star\}}$$

$$\frac{\Theta, G \vdash p}{\Theta, G \vdash !p} \; {}^{\{Not\}} \qquad\qquad \frac{\exists e.M : e \wedge A \leftarrow e \in R}{\Theta, G \vdash M} \; {}^{\{MetaVar\}} \qquad \frac{\exists e.\Theta(I) = e}{\Theta, G \vdash I} \; {}^{\{Ref\}}$$

Figura 3.2: Patterns semantics.

Definition 3 (Valid pattern with respect to a tree)**.** We say that a pattern p is valid with respect to a tree t, $p \sim t$, if and only if $\exists e.p : e \wedge t : e$.

We also present a type coercion for parsing expressions.

We present the syntax for terms of subtyping.

$$p \quad \rightarrow \quad \epsilon \mid a \mid A \mid p_1 \, p_2 \mid p_1 / p_2 \mid L \, p \mid R \, p \mid p^\star \mid [p] \mid ! \, p \mid M$$

Of note, are the production rules $L \, p$, $R \, p$ and $[p]$ which represents, respectively, the proof that the left expression in a choice operator is a subtype of the choice, the

$$\frac{}{e <: e} \; \{Reflexive\} \qquad\qquad \frac{e_1 <: e_2 \quad e_2 <: e_3}{e_1 <: e_3} \; \{Transitive\}$$

$$\frac{}{e_1 <: e_1/e_2} \; \{Alt_{Left}\} \quad \frac{}{e_2 <: e_1/e_2} \; \{Alt_{Right}\} \quad \frac{n \geq 1}{e^n <: e^\star} \; \{Star\}$$

Figura 3.3: Subtype relations for parsing expressions

proof that the right expression in a choice operator is a subtype of the choice, and the proof that a list (possibly empty) of patterns is a subtype of the $\star$ operator.

$$\frac{p : e \quad e <: e' \quad \exists p'.p' = C(p, e <: e') \quad \forall t.t : e'}{p' \sim t} \; Pattern$$

Figura 3.4: Pattern coercion

Where C is a resursively defined function that receives a pattern and a proof that the pattern is valid and returns a corrected pattern and is defined as follows:

$$
\begin{aligned}
C(\epsilon, \epsilon) &= \epsilon \\
C(a, a) &= a \\
C(a, a') &= \bot, \text{ if } a \neq a' \\
C((A\,p), (A\,p')) &= A\,C(p, p') \\
C((A\,p), (A'\,p')) &= \bot, \text{ if } A \neq A' \\
C(M, M) &= M \\
C(M, M') &= \bot, \text{ if } M \neq M' \\
C(p_1\,p_2, p'_1\,p'_2) &= C(p_1, p'_1)\,C(p_2, p'_2) \\
C(\epsilon, xs)) &= [] \\
C(p, [x]) &= [C(p, x)] \\
C(p_1\,p_2, x : xs) &= C(p_1, x)\,C(p_2, xs) \\
C(p_1\,/\,p_2, p'_1\,/\,p'_2) &= C(p_1, p'_1)/C(p_2, p'_2) \\
C(p, L\,p') &= C(p, p')/!\,\epsilon \\
C(p, R\,p') &= !\,\epsilon/C(p, p') \\
C(p^\star, p'^\star) &= (C(p, p'))^\star \\
C(!\,p, !\,p') &= !\,C(p, p')
\end{aligned}
$$

– **Escrever regras de casamento** –

Figura 3.5: Matching rules

3.2 Implementation details

After parsing the patterns, we replace references to other patterns with the pattern itself. To do this, we create a dependency graph between the patterns, topologically sort and replace the references so that no resulting pattern contains references to other patterns and can be treated as a single pattern.

We also define two new operators for PEGs: flatten ($^\wedge e$) and indentation ($e_1 > e_2$). The former flattens the parsed tree in one single node that turns into a terminal (and can be matched as such via patterns) and the latter, for the purpose of parsing, acts as the sequence $e_1 e_2^\star$ with the restriction that $e_2^\star$ must be indented with respect to e_1 and matches as if it was a normal sequence.

3.3 Conclusion

—

Capítulo 4

Results

This chapter presents the results obtained so far. Section 4.1 presents some case studies to evaluate the capabilities of the tool (library?) developed. Section 4.2 concludes the chapter.

4.1 Case studies

To evaluate and demonstrate the capabilities of the tool, we present below some case studies: the generation of a call graph, a suggestion for rewriting the code, and a verification of the presence of specific constructions in the code. All case studies use the PEG shown in Appendix A, which accepts a subset of Python.

4.1.1 Call graph generation

In this case study we try to extract a call graph from a given source code and to do so, we will need two different patterns: one that matches with all definitions of functions and one that matches with all functions calls.

```
pattern definition : function\def := ("def" @space #name:identifier "(" @space #p:i
pattern call : function\call := #name:identifier @space "(" @space #v:expr\list? ")
pattern space : space := " "*
```

The syntax *pattern name : type := expression* represents a pattern identified by *name* that matches trees with type *type* and *expression* specifies how it will match. @*name* denotes a reference to another pattern and #*var* : *type* denotes a variable that matches with trees of type *type*. ϵ indicates that there must be nothing following the call.

The pattern *definition* matches with any function definition, storing the function identifier, parameters and function body, respectively, in variables *name*, *p* and *block*. The pattern *function_call* matches with any function call, storing the

13

function name and arguments, respectively, in variables *name* and *v*. Then, by first matching the pattern *definition* in the source code and then matching *function_call* in each function's body using what was stored in each match of variable *block*, it is possible to make a list of pairs (*caller, calee*) and create a call graph. Consider the following Python code as an example:

```python
import math

def delta(a, b, c):
    return b**2 - 4*a*c

def bhaskara(a, b, c):
    d = delta(a, b, c)
    x1 = (-b + math.sqrt(d)) / 2*a
    x2 = (-b - math.sqrt(d)) / 2*a
    return x1, x2

a = float(input("Digite o valor de a: "))
b = float(input("Digite o valor de b: "))
c = float(input("Digite o valor de c: "))

x1, x2 = bhaskara(a, b, c)

print(f"{a}x^2 + {b}x + {c}")
print(f"Raiz 1: {x1}")
print(f"Raiz 2: {x2}")
```

Pattern *definition* will match with functions *delta* and *bhaskara*. In *deltas*'s body, pattern *function_call* won't match with any statement, since there are no calls in its body. In *bhaskara*'s body, *function_body* will match the call to *delta* and both calls to *math.sqrt*. This will make the list [(*bhaskara*, *delta*), (*bhaskara*, *math.sqrt*), (*bhaskara*, *math.sqrt*)] and, removing the duplicates, it is possible to make the call graph.

4.1.2 Source code validation

These examples were designed in the context of an introductory programming course. This course uses an automatic judge that evaluates the results of programs submitted by the student, but does not perform a static analysis of the submitted source code. The following examples present two distinct cases that the judge fails to identify, but that could be identified and penalized (or even rejected) due to the use of

unauthorized constructs.

Checking for specific constructs

Consider a question that asks the student to implement a program that calculates the factorial of an integer n entered by the user. The expected solution is for the student to use a loop, such as *while*, to implement the successive multiplication of the numbers, as in the code presented below:

```
n = int(input("Digite um numero: "))
fatorial = 1
contador = n
while (contador >= 1):
    fatorial = fatorial * contador
    contador = contador - 1
print(f"{n}! = {fatorial}")
```

However, within the Python `math` library, there is the `factorial` function, which, given an integer n, returns the result of $n!$. For this reason, some students end up importing the library and using the ready-made function, circumventing the objective of the exercise, which is to practice the repetition loop, as shown below.

```
import math
n = int(input("Digite um numero: "))
fatorial = math.factorial(n)
print(f"{n}! = {fatorial}")
```

The pattern presented below is capable of identifying the presence of a call to the *factorial* function.

```
pattern factorial_call : function_call := (identifier := "math.factorial") @spa
pattern space : space := " "*
```

Where *(identifier := "math.factorial")* means that the name of the function must be *math.factorial*, *#v2:expr_list?* means that the function's argument list will be stored in the variable *v2*. So, if the pattern matches, it means that the student is using the *factorial* function from the *math* library instead of writing the repetition loop, bypassing the original objective of the exercise.

Blocking disallowed constructs

Imagine that you are evaluating a question whose statement is as follows:

"You have been hired by the Ministry of the Environment to assess the reforestation target of Brazilian regions and will implement a program to help you with your

analyses. To facilitate data collection, each state is divided into microregions. You periodically receive a vector of integer values indicating the minimum number of native trees planted for each state, representing the target for each state, and a matrix of integer values showing the number of trees planted in each state in each microregion. The rows of the matrix represent the microregions and the columns represent the states. The inputs of the vector and matrix are made through the functions inputVector and inputMatrix, respectively (defined in the course textbook).

Your program calculates the total number of trees planted by the states and evaluates whether they have met the target (the number of trees planted is equal to or greater than the state's target), printing on the terminal the states that failed to meet the target (the state numbers start at 1, although the indexes start at 0, so index 0 represents state 1, index 1 represents state 2, and so on). The relationship between the vector and the matrix is given by the indexes of the vector elements and the column indexes of the matrix."

And, when opening a solution submitted by a student, you come across this code:

```python
def inputVetor():
    entrada = input("Informe as metas dos estados: ")
    return list(map(int, entrada.split(',')))


def inputMatriz():
    entrada = input("Informe o plantio de arvores: ")
    linhas = entrada.split(';')
    matriz = [list(map(int, linha.split(','))) for linha
        in linhas]
    return matriz


def main():
    print("Ministerio do Meio Ambiente")
    metas = inputVetor()
    plantio = inputMatriz()

    num_estados = len(metas)
    totais_plantio = [sum(linha[i] for linha in plantio)
        for i in range(num_estados)]

    for i in range(num_estados):
    if totais_plantio[i] < metas[i]:
        print(f"Estado {i+1}, meta = {metas[i]}, plantio
```

```
            -=-{totais_plantio[i]}")
```

```python
if __name__ == "__main__":
    main()
```

Although correct and generating the expected response, it uses Python language resources that ignore the intended learning objectives or were not presented in the course, such as the use of the `map` and `sum` functions, list comprehension and the use of the `__name__` attribute. The PEG presented in Appendix A would immediately reject this solution, as it does not accept constructions such as list comprehension.

4.1.3 Source code rewriting

Now consider the following code snippet:

```python
if not a:
    print("Condition -'a'-is-false")
else:
    print("Condition -'a'-is-true")
```

Although the code does not present any errors, it can be refactored, with the aim of improving the structure and, consequently, understanding of the code, by removing the *not* from the `if` condition and exchanging the command blocks of `if` and `else`, as follows:

```python
if a:
    print("Condition -'a'-is-true")
else:
    print("Condition -'a'-is-false")
```

By identifying this type of construction in the student's code, it is possible to suggest a rewrite to the student, explaining the reason for the suggestion and the improvement it would bring to the code. The patterns presented below represent a way of detecting the construction presented previously and how to rewrite it.

```
pattern if_def : & if_stmt := (("if" @space @expr ":") #ifBlock:statement*) @el
pattern elseBlock : & else_block := ("else" @space ":") #elseBlock:statement* \

pattern subst : & if_stmt := (("if" @space #condition:expression ":") #elseBloc
pattern elseBlock2 : & else_block := ("else" @space ":") #ifBlock:statement* \\

pattern expr    : & expression := @orExpr \epsilon \\
pattern orExpr  : & or_expr    := @andExpr \epsilon \\
```

17

```
pattern andExpr : & and_expr   := "not" @space #condition:comparison \\

pattern space : & space := " "*
```

Where the *if_def* pattern matches when it finds an *if* that has as a condition a negated expression and the *subst* pattern represents the rewrite that will be suggested to the student.

The variables $\#condition : expression$, $\#ifBlock : statement*$ and $\#elseBlock : statement*$ in the *if_def* pattern capture, respectively, the expression in the `if` condition, the entire `if` block of statements and the entire `else` block of statements. The way these variables appear in the *subst* pattern indicates how the rewrite will be performed. In this pattern, you can see that the *not* in the condition no longer appears, while the position of the block variables has been changed. Thus, it is possible to use what was captured by the variables in the *if_def* pattern and place it in the places where the variables appear in the *subst* pattern. Finally, we can present the rewrite to the student, along with an explanation, to make a suggestion for improving their solution.

4.2   Conclusion

—

Capítulo 5

Schedule and Expected results

This chapter presents the next steps for the development of this work and estimated deadlines. They are not precise and can change depending on the results obtained. The remaining activities are enumerated below.

1. Correct and terminate the pattern matching algorithm based on the presented formalization.

2. Testing and proofs of properties on patterns.

3. Submission of a paper.

4. Writing and presentation.

The intended continuation of this is work is summarized on Table 5.1.

Tabela 5.1: Intended schedule of future work

Task	Deadline
—	—

Capítulo 6

Conclusion

Referências Bibliográficas

[1] ATKINSON, D., GRISWOLD, W. "Effective pattern matching of source code using abstract syntax patterns", *Softw., Pract. Exper.*, v. 36, pp. 413–447, 04 2006. doi: 10.1002/spe.704.

[2] KOPPEL, J., PREMTOON, V., SOLAR-LEZAMA, A. "One tool, many languages: language-parametric transformation with incremental parametric syntax", *Proc. ACM Program. Lang.*, v. 2, n. OOPSLA, out. 2018. doi: 10.1145/3276492. Disponível em: <https://doi.org/10.1145/3276492>.

[3] PREMTOON, V., KOPPEL, J., SOLAR-LEZAMA, A. "Semantic code search via equational reasoning". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, p. 1066–1082, New York, NY, USA, 2020. Association for Computing Machinery. ISBN: 9781450376136. doi: 10.1145/3385412.3386001. Disponível em: <https://doi.org/10.1145/3385412.3386001>.

[4] SILVA, D., DA SILVA, J. P., SANTOS, G., et al. "RefDiff 2.0: A Multi-Language Refactoring Detection Tool", *IEEE Transactions on Software Engineering*, v. 47, n. 12, pp. 2786–2802, Dec 2021. ISSN: 1939-3520. doi: 10.1109/TSE.2020.2968072.

[5] VAN TONDER, R., LE GOUES, C. "Lightweight multi-language syntax transformation with parser parser combinators". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, p. 363–378, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450367127. doi: 10.1145/3314221.3314589. Disponível em: <https://doi.org/10.1145/3314221.3314589>.

[6] MATUTE, G., NI, W., BARIK, T., et al. "Syntactic Code Search with Sequence-to-Tree Matching: Supporting Syntactic Search with Incomplete Code Fragments", *Proc. ACM Program. Lang.*, v. 8, n. PLDI, jun. 2024. doi: 10.1145/3656460. Disponível em: <https://doi.org/10.1145/3656460>.

[7] IERUSALIMSCHY, R. "A text pattern-matching tool based on Parsing Expression Grammars", *Softw. Pract. Exper.*, v. 39, n. 3, pp. 221–258, mar. 2009. ISSN: 0038-0644.

Apêndice A

Simplified Python PEG

```
file          <- (blank* newline)* statement+
statement     <- (compound / simple / comment) blank* newline*
compound      <- function_def / if_stmt / while_stmt / for_stmt
function_def  <- ("def" space identifier "(" space id_list? ")" space ":") > statem
if_stmt       <- (("if" space expression ":") > statement) (elif_block / else_block
elif_block    <- (("elif" space expression ":") > statement) (elif_block / else_blo
else_block    <- ("else" space ":") > statement
while_stmt    <- ("while" space expression ":") > statement
for_stmt      <- ("for" space identifier space "in" space expression ":") > stateme
simple        <- import_stmt / assignment / return_stmt / expression
return_stmt   <- "return" space expr_list
import_stmt   <- simple_import / from_import
simple_import <- "import" space identifier
from_import   <- "from" space identifier space "import" space (id_list / "*")
assignment    <- id_list space attr space expression
attr          <- "=" / "+=" / "-=" / "*=" / "/="
expression    <- or_expr ("or" space or_expr)*
or_expr       <- and_expr ("and" space and_expr)*
and_expr      <- "not" space comparison / comparison
comparison    <- sum (op_comp sum)*
sum           <- term (op_term term)*
term          <- factor (op_factor factor)*
factor        <- power (op_power power)*
power         <- "-" neg / neg
neg           <- primary space / "(" space expression ")" space
op_comp       <- ("==" / "!=" / "<=" / ">=" / "<" / ">") space
op_term       <- ("+" / "-") space
op_factor     <- ("*" / "/" / "%") space
```

```
op_power        <- "**" space
primary         <- function_call / array_access / "[" items? "]" / atom
function_call   <- identifier space "(" space expr_list? ")" ("." primary)?
array_access    <- identifier space "[" space expression "]" ("." primary)?
atom            <- "True" / "False" / "None" / number / strings / identifier
expr_list       <- expr1 space (sep expr1 space)*
expr1           <- (single_id space "=" space)? expression
items           <- expression (sep expression space)*
id_list         <- identifier space (sep identifier space)*
sep             <- "," space
^strings        <- fstring / string
fstring         <- "f" string
string          <- ['] (!['] char)* ['] / ["] (!["] char)* ["]
char            <- [a-zA-Z0-9 :{}.,;^+-*/%()_!?áéúçã] / "[" / "]"
^identifier     <- single_id ("." single_id)*
single_id       <- [a-zA-Z] [a-zA-Z0-9_]*
^number         <- [0-9]+
space           <- " "*
^blank          <- comment / " "
^comment        <- "#" char*
^newline        <- "\r\n" / "\r" / "\n"
```