Dr.

UNIVERSIDADE FEDERAL DE OURO PRETO

decom
departamento de computação

—

Guilherme Augusto Anício Drummond do Nascimento

Orientador: Rodrigo Geraldo Ribeiro

Ouro Preto
Junho de 2025

UNIVERSIDADE FEDERAL DE OURO PRETO

decom
departamento de computação

—

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
Junho de 2025

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

Agradecimentos

—

O autor gostaria de agradecer à FAPEMIG, CAPES, CNPq e UFOP pelo fomento ao projeto de pesquisa apresentado. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

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

—

Guilherme Augusto Anício Drummond do Nascimento

Junho/2025

Orientador: Rodrigo Geraldo Ribeiro

Programa: Ciência da Computação

Abstract of Qualifying Exam presented to UFOP as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

—

Guilherme Augusto Anício Drummond do Nascimento

June/2025

Advisor: Rodrigo Geraldo Ribeiro

Department: Computer Science

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introduction

—

Capítulo 2

Related Literature

Colocar parágrafo introduzindo a seção.

2.1 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 $(e, s) \Rightarrow_G \perp$ 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 (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 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

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

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

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

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

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

Figura 2.1: Parsing expressions operational semantics.

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.

– TODO: Colocar exemplos, mostrando como é o processamento de uma palavra –

The following PEG (Figure 2.2) recognizes mathematical formulas that apply the basic four operations to non-negative integers:

Expr	←	Sum
Sum	←	Prod ('+' Prod)*
Prod	←	Value ('*' Value)*
Value	←	[0-9]+ / '(' Expr ')'

Figura 2.2: PEG for mathematical formulas.

Consider the string `1 + 2 * 3`. The initial rule `Expr` delegates to the rule `Sum`, which will first try to parse a `Prod` that will, in turn, first try to consume a `Value`, which recognizes the number `'1'`. Since the `Prod` rule does not consume a `'*'`, it returns back to `Sum`. It then finds the `'+'` operator and tries to parse another `Prod`, which will consume the 2 as a `Value` and, this time, finds the `'*'` operator and

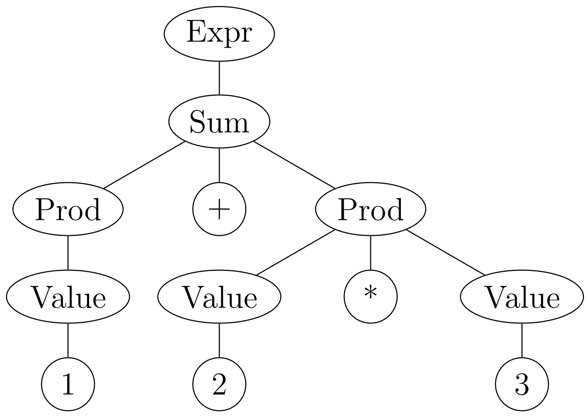

Figura 2.3: Generated abstract syntax tree for the expression 1+2*3

consumes the 3 as another `Value`. It does not find another `'*'` and goes back to `Sum`, that also does not find another `'+'` operator, it goes back to `Expr` and finalizes the parsing process, resulting in the syntactic structure corresponding to the expression: `1+(2*3)`. The abstract syntactic tree generated can be seen on Figure 2.3.

2.2 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. Throughout the paper, a prototypical example of extracting a call-graph from a given code, giving examples in differents tools for pattern matching. At a later section, we also present an extraction of a call-graph using the tool developed in this paper.

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 currentl supports C, Java, JavaScript, Lua, and Python. 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 a tool called *Yogo*, 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. The tool is capable of recognizing differents variations of the same operation and also when code is an instance of a higher-level concept. *Yogo* is built on the *Cubix* multi-language 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 and has support for two other popular programming languages: JavaScript and C, demonstrating 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 such that a common grammar expresses the central context-free language (CLF) properties shared by many contemporary languages and open extensions points in the grammar 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 evaluated *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 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.

2.3 Conclusão

Fazer um parágrafo finalizando a seção.

Capítulo 3

Methodology

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:

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

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.

The generation of tress by 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 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 3.1 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 $(e, s_p s_r) \Rightarrow_G (t, s_r)$.*

Let $G = (V, \Sigma, R, e_s)$ be an arbitrary PEG, Θ a finite set of identified patterns, U a finite set of variables, the meta-variable $a \in \Sigma$ an arbitrary alphabet symbol, $A \in V$ a variable and e a parsing expression. Each identified pattern $p_i \in \Theta$ is a

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

$$\frac{(e_1, s_{p_1} s_{p_2} s_r) \Rightarrow_G (t_1, s_{p_2} s_r) \quad (e_2, s_{p_2} s_r) \Rightarrow_G (t_2, s_r)}{(e_1\, e_2, s_{p_1} s_{p_2} s_r) \Rightarrow_G (\langle t_1, t_2 \rangle, s_r)}\ \{Cat_{S1}\} \qquad \frac{(e_1, s_p s_r) \Rightarrow_G (t_1, s_r) \quad (e_2, s_r) \Rightarrow_G}{(e_1\, e_2, s_p s_r) \Rightarrow_G \bot}$$

$$\frac{(e_1, s) \Rightarrow_G \bot}{(e_1\, e_2, s) \Rightarrow_G \bot}\ \{Cat_{F1}\} \qquad \frac{(e_1, s_p\, s_r) \Rightarrow_G (t, s_r)}{(e_1 \,/\, e_2, s_p\, s_r) \Rightarrow_G (L\, t, s_r)}\ \{Alt_{S1}\} \qquad \frac{(e_1, s_p\, s_r) \Rightarrow_G \bot \quad (e_2, s_p\, s_r) \Rightarrow_G}{(e_1 \,/\, e_2, s_p\, s_r) \Rightarrow_G R\, t}$$

$$\frac{(e, s_{p_1} s_{p_2} s_r) \Rightarrow_G (t_1, s_{p_2} s_r) \quad (e^\star, s_{p_2} s_r) \Rightarrow_G (t_2, s_r)}{(e^\star, s_{p_1} s_{p_2} s_r) \Rightarrow_G ([t_1, t_2], s_r)}\ \{Star_{rec}\} \qquad \frac{(e, s) \Rightarrow_G \bot}{(e^\star, s) \Rightarrow_G (\hat{\epsilon}, s)}\ \{Star_{end}\}$$

$$\frac{(e, s_p\, s_r) \Rightarrow_G (t, s_r)}{(!\, e, s_p\, s_r) \Rightarrow_G \bot}\ \{Not_F\} \qquad \frac{(e, s) \Rightarrow_G \bot}{(!\, e, s) \Rightarrow_G (\eta), s)}\ \{Not_S\} \qquad \frac{}{(a, \epsilon) \Rightarrow_G}$$

Figura 3.1: Parsing expressions operational semantics that produces a tree.

pair (I, p), where $I \in U$ and p is a pattern expression. The following context-free grammar defines the syntax of a pattern expression:

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

Where ϵ is a pattern that matches with the empty string and is always successful, a matches only with the symbol 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 expression p' where $(I, p') \in \Theta$.

Figure 3.2 defines the pattern semantics.

$$\frac{}{\Theta, G \vdash \epsilon}\ \{Eps\} \qquad \frac{}{\Theta, G \vdash a}\ \{ChrS\} \qquad \frac{G = (V, \Sigma, R,}{\Theta, G}$$

$$\frac{\Theta, G \vdash p_1 \quad \Theta, G \vdash p_2}{\Theta, G \vdash p_1\, p_2}\ \{Sequence\} \qquad \frac{\Theta, G \vdash p_1 \quad \Theta, G \vdash p_2}{\Theta, G \vdash p_1 \,/\, p_2}\ \{Choice\} \qquad \frac{\Theta, G}{\Theta, G \vdash}$$

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

Figura 3.2: Pattern expressions semantics.

Definition 2 (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.

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

$$\frac{}{e_1 <: e_1/e_2}\ \{Alt_Left\} \quad \frac{}{e_2 <: e_1/e_2}\ \{Alt_Right\} \quad \frac{n \geq 1}{e^n <: e^\star}\ \{Star\}$$

Figura 3.3: Type coercions

$$\frac{p : e \quad e <: e' \quad \exists p'.p' = C(p, e <: e') \quad \forall t.t : e1}{p' \sim t}\ Pattern$$

Figura 3.4: Pattern coercion

Where $C :: Pattern \rightarrow TypeCoercion \rightarrow Pattern$ and is defined as follows:

$$\begin{array}{lcl}
C(p : e_1, e_1 <: e_1 \,/\, e_2) & = & p \,/\, !\epsilon \\
C(p : e_1, e_2 <: e_1 \,/\, e_2) & = & !\epsilon \,/\, p \\
C(p : e, e <: e^\star) & = & p \,/\, \epsilon \\
C((p_1 : e)\,(p_2 : e), e <: e^\star) & = & C(p_1 : e, e <: e^\star)\, C(p_2 : e, e <: e^\star)
\end{array}$$

$$\frac{}{\epsilon \sim \epsilon}\ \{Eps\} \qquad \frac{}{a \sim a}\ \{ChrS\} \qquad \frac{}{A \sim A}\ \{Var\}$$

$$\frac{p_1 \sim t_1 \quad p_2 \sim t_2}{p_1\, p_2 \sim t_1\, t_2}\ \{Seq\} \quad \frac{p_1 \sim t_1 \quad p_2 \sim t_2}{p_1/p_2 \sim t_1/t_2}\ \{Choice\} \quad \frac{p \sim t}{p^\star \sim [t]}\ \{Star\}$$

$$\frac{p \sim t}{!\,p \sim !\,t}\ \{Not\} \qquad \frac{\exists e.M : e \wedge t : e}{M\ t}\ \{MetaVar\}$$

Figura 3.5: Matching rules

Capítulo 4

Results

Capítulo 5

Conclusion and Future Works

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.