Automatic evaluation of top-down predictive parsing

Carles Creus
ccreus@cs.upc.edu

Pau Fernández
pfernandez@cs.upc.edu

Guillem Godoy
ggodoy@cs.upc.edu

Nil Mamano
nil.mamano@gmail.com

Universitat Politècnica de Catalunya, Computer Science Department

Abstract

We develop efficient methods to check whether two given Context-Free Grammars (CFGs) are transformed into parsers that recognize the same language and construct the same Abstract Syntax Trees (ASTs) for each input. In this setting, we consider a model of top-down predictive parser generator with directives for AST construction that is a simplified variant of PCCTS/ANTLR3. As an application, we implement an evaluator for an online judge with educational purposes in the context of a Compilers course.

1 Introduction

For the last few years, we have developed a specialized judge for the Theory of Computation course, publicly accessible at https://racso.cs.upc.edu with no registration required. It offers users a list of exercises on deterministic finite automata, context-free grammars (CFGs) [3], push-down automata, reductions between undecidable problems, and reductions between NP-complete problems [2]. Users can submit their solution proposals, the judge evaluates them, and provides a counterexample when the submission is wrong. In our experience, the judge has had a positive effect on the motivation and involvement of the students of the course, and showing them counterexamples has proven crucial to help them correct their mistakes.

In order to make the judge also useful in a Compilers course, we have recently added exercises on CFGs for top-down predictive parsing [1, 4]. To evaluate this kind of exercises, we have developed new methods to, on the one hand, check whether the solution proposals submitted by users parse the same language as the reference solution provided by the problem setter and, on the other hand, whether the proposals and the reference solution construct the same abstract syntax trees (ASTs) for each input. These methods are not complete, but behave well in practice, free the problem setter from the tedious task of creating comprehensive test sets, and return the verdict in just a few seconds (at worst). The latter is important in our context, since providing instant feedback to students helps in keeping them motivated. Moreover, the methods are also able to produce a counterexample when the submissions are wrong.

We have chosen a specific model of top-down predictive parser generator that is a simplified variant of PCCTS/ANTLR3 [5, 6]. It accepts CFG descriptions like the following:

```
instruction : IDENTIFIER '='^ expression
            | 'if'^ expression instruction ( 'else'! instruction | ) ;
expression  : term ( '+'^ term | '-'^ term )* ;
term        : basic ( '*'^ basic | '/'^ basic )* ;
basic       : INTCONSTANT | IDENTIFIER | '('! expression ')'! ;
IDENTIFIER  : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')* ;
INTCONSTANT : ('0'..'9')+ ;
```

This work has been partially supported by funds from the Spanish Ministry of Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R), and by funds from the Spanish Ministry of Science and Innovation (MICINN) under grant FORMALISM (ref. TIN2007-66523). Additionally, the first author has been supported by an FPU-MECD 2010 grant from the Spanish Ministry of Education.

Lowercase identifiers denote variables (non-terminals) and quoted strings as well as uppercase identifiers denote tokens (terminals). In addition, some tokens are followed by directives for AST construction, like `^` to indicate that the token must become the root of the currently constructed AST, and `!` to mark that such token must not be included in the AST. The previous CFG is ambiguous because a sequence of the form `'if' expression 'if' expression IDENTIFIER '=' expression 'else'` ... has at least two different syntax trees, since the `'else'` can be associated to either the first or the second `'if'`. This is the classical "dangling else" ambiguity, and it cannot be tackled by a (free-of-conflicts) LL(k) CFG. Nevertheless, our model gives priority to the left branch of each alternative `|` when a conflict occurs, and hence, in the previous example the `'else'` is associated to the last seen `'if'`. In general, prioritizing rules according to their definition order may cause the parsed language to be different from the language generated by the given CFG. The interpretation as a parser of the above CFG corresponds to the following program:

```
function instruction() returns Forest
  Forest  ast := empty
  Bool rooted := false
  if lookahead() = IDENTIFIER
    t := consumeToken(IDENTIFIER)
    ast.addToForest(t,rooted)
    t := consumeToken('=')
    ast.addAsRoot(t)
    rooted := true
    f := expression()
    ast.addToForest(f,rooted)
  else if lookahead() = 'if'
    t := consumeToken('if')
    ast.addAsRoot(t)
    rooted := true
    f := expression()
    ast.addToForest(f,rooted)
    f := instruction()
    ast.addToForest(f,rooted)
    if lookahead() = 'else'
      consumeToken('else')
      f := instruction()
      ast.addToForest(f,rooted)
    else if lookahead() = $
      // Nothing to do
    else SYNTAXERROR
  else SYNTAXERROR
  return ast
```

```
function expression() returns Forest
  Forest  ast := empty
  Bool rooted := false
  f := term()
  ast.addToForest(f,rooted)
  while lookahead() ∈ {'+','-'}
    if lookahead() = '+'
      t := consumeToken('+')
      ast.addAsRoot(t)
      rooted := true
      f := term()
      ast.addToForest(f,rooted)
    else if lookahead() = '-'
      t := consumeToken('-')
      ast.addAsRoot(t)
      rooted := true
      f := term()
      ast.addToForest(f,rooted)
  return ast

function term() returns Forest
  ...

function basic() returns Forest
  ...
```

In the code above, the `lookahead` function returns the next token of the input without consuming it, the `consumeToken` function consumes and returns the next token of the input while verifying its lexical class, and \$ represents the end-of-input mark. Also, the `ast` variable keeps the current constructed AST as a forest, i.e., a list of trees. The instruction `ast.addAsRoot(t)` sets `ast` to be the tree whose root is `t` and whose list of children is the previous forest kept in `ast`. When `rooted` is false, `ast.addToForest(f,rooted)` sets `ast` to be the forest resulting of concatenating the previous value of `ast` with `f`. Otherwise, when `rooted` is true, the current value of `ast` is a tree and `ast.addToForest(f,rooted)` modifies it by concatenating its list of children with `f`.

Note that the ASTs constructed by the above program correspond to an interpretation of the input where operators $*$ and $/$ have precedence over $+$ and $-$, and all of them are left-associative. For example, with input $1*2/3-4+5/6$, the AST returned by the `expression` function is $+(-(/(*(1,2),3),4),/(5,6))$, i.e., the expression is interpreted with the implicit parenthesization $(((1*2)/3)-4)+(5/6)$. A different CFG parsing the same language may produce different ASTs.

1.1 Approach

Given two CFGs G_1 and G_2, we have to check whether they parse the same language, i.e., whether $\mathcal{L}_{\mathcal{P}}(G_1) = \mathcal{L}_{\mathcal{P}}(G_2)$, and whether the corresponding constructed ASTs for each input coincide. The first part is the easiest one and, in fact, it is decidable: note that the parsing process can be simulated by a deterministic push-down automaton (DPDA), and equivalence of DPDAs is decidable in non-elementary time [7, 8]. Nevertheless, such time complexity is excessive, and we prefer to design (perhaps incomplete) methods that behave well in practice. In [3] we developed an efficient hashing method for testing whether two CFGs G_1 and G_2 generate the same language, i.e., whether $\mathcal{L}(G_1) = \mathcal{L}(G_2)$. This method is based on a hash function H that maps multisets of words to natural numbers. To compare G_1 and G_2, we fix a length L and compute $N_1 := H([w \mid |w| \leq L \ \wedge \ w \in \mathcal{L}[G_1]])$ and $N_2 := H([w \mid |w| \leq L \ \wedge \ w \in \mathcal{L}[G_2]])$, where $\mathcal{L}[G]$ denotes $\mathcal{L}(G)$ as multiset by considering each word as many times as it is generated by G. If $N_1 = N_2$, we conclude $\mathcal{L}[G_1] = \mathcal{L}[G_2]$, and in particular $\mathcal{L}(G_1) = \mathcal{L}(G_2)$. Note that this conclusion might be wrong, either because the smallest $w \in (\mathcal{L}(G_1) \triangle \mathcal{L}(G_2))$ has length greater than L, or in the event of hash collisions, although practice shows that this is unlikely. If $N_1 \neq N_2$, then $\mathcal{L}[G_1] \neq \mathcal{L}[G_2]$ necessarily holds, and if G_1 is unambiguous, we can conclude either that $\mathcal{L}(G_1) \neq \mathcal{L}(G_2)$ or that G_2 is ambiguous. One of the advantages of this approach is that the function H can be computed efficiently over the CFG structure without generating the multisets. Moreover, the method efficiently produces a counterexample to $\mathcal{L}[G_1] = \mathcal{L}[G_2]$ when $N_1 \neq N_2$.

Recall that we need to check $\mathcal{L}_{\mathcal{P}}(G_1) = \mathcal{L}_{\mathcal{P}}(G_2)$ instead of $\mathcal{L}(G_1) = \mathcal{L}(G_2)$. Our approach to check $\mathcal{L}_{\mathcal{P}}(G_1) = \mathcal{L}_{\mathcal{P}}(G_2)$ consists in reducing this problem to check $\mathcal{L}(G'_1) = \mathcal{L}(G'_2)$ using the hashing method, where G'_1 and G'_2 are unambiguous CFGs obtained from G_1 and G_2, respectively, such that $\mathcal{L}_{\mathcal{P}}(G_1) = \mathcal{L}(G'_1)$ and $\mathcal{L}_{\mathcal{P}}(G_2) = \mathcal{L}(G'_2)$. Intuitively, G'_1 and G'_2 simulate the deterministic (conflict-resolved) behaviour of the parsers corresponding to G_1 and G_2, respectively. Obtaining such CFGs is definitely possible: the languages parsed by CFGs can be recognized by DPDAs, and it is well known that DPDAs can be transformed into equivalent unambiguous CFGs. Nevertheless, these transformations are involved and we present a direct and simpler transformation T such that, given a CFG G, $\mathcal{L}_{\mathcal{P}}(G) = \mathcal{L}(T(G))$. Moreover, T can be composed with another transformation A such that $A(T(G))$ generates the set of words that represent (in an appropriate formalism) the ASTs constructed by G. This way, we can test if G_1 and G_2 construct the same ASTs by checking $\mathcal{L}(A(T(G_1))) = \mathcal{L}(A(T(G_2)))$. This requires to modify the hashing method to compute a more general function $H([\sigma(w) \mid |w| \leq L \ \wedge \ w \in \mathcal{L}[G]])$ that depends on a morphism σ. The morphism allows to represent the fact that tokens followed by ! must be removed from the AST by mapping them to the empty word, and to make the method more robust against collisions.

According to our experiments, this method is accurate in practice and runs in a few seconds with big CFGs when looking for counterexamples of size at most $L \approx 12$. This is because the complexity of this method has a factor L^4. Since this limitation can be excessive in some cases, we propose an alternative transformation P that allows to compare *partial* executions of the parsers. More precisely, $P(G)$ generates the set of words w such that there is a partial execution of G that consumes exactly w without producing a syntax error. Hence, a counterexample to $\mathcal{L}(P(G_1)) = \mathcal{L}(P(G_2))$ is a word w that produces a syntax error on a partial execution of just one of G_1, G_2. The advantage of using P instead of T is that, in many occasions, the size of such w will be significantly smaller than the size of a counterexample to $\mathcal{L}_{\mathcal{P}}(G_1) = \mathcal{L}_{\mathcal{P}}(G_2)$. In addition, the transformation P can also be composed with A to check equivalence of partial AST construction.

1.2 Outline

In Section 2 we recall the basic concepts of words, languages, morphisms, CFGs and their transformation to CNF. In Section 2.1 we describe in detail the original hashing method that was briefly presented in [3]. In Section 3, we define the model of parser generator. In Section 4 we present the transformations T and P. In Section 5, we discuss the way to represent ASTs as words, present the transformation A, and adapt the hashing method to check equivalence of AST construction. In Section 6 we analyse the empirical performance of the method. In Section 7 we conclude.

2 Preliminaries

Words are finite-length lists of symbols chosen over an underlying (finite) alphabet Σ. The length of a word w is denoted by $|w|$, its i'th symbol by $w[i]$, and its subword between i and j, inclusive, by $w[i \dots j]$, for $1 \leq i \leq j \leq |w|$. The empty word is denoted by ε. The concatenation of two words x, y is denoted by $x \cdot y$, or just xy. It is extended to languages (sets of words) as $L_1 L_2 = L_1 \cdot L_2 = \{xy \mid x \in L_1 \ \wedge \ y \in L_2\}$. By $\mathsf{Prefixes}(L)$ we denote $\{u \mid \exists v : uv \in L\}$. By $\mathsf{match}(u, v)$ we denote that either u is a prefix of v (i.e., $v = uv'$ for some v') or v is a prefix of u. A morphism σ is a mapping from words to words satisfying $\sigma(xy) = \sigma(x)\sigma(y)$. Thus, it suffices to define σ for symbols, since then it is generalized to arbitrary words as $\sigma(a_1 \cdots a_n) = \sigma(a_1) \cdots \sigma(a_n)$.

We assume that the reader is familiar with the concept of context-free grammar (CFG) as a structure $G = \langle V, \Sigma, \delta, S \rangle$, where V is the (finite) set of variable symbols, Σ is the (finite) alphabet of terminal symbols, $\delta \subset V \times (V \cup \Sigma)^*$ is the (finite) set of production rules, and $S \in V$ is the initial symbol. We will usually denote variable symbols with uppercase letters $X, Y, Z, \dots$, with possible subscripts, and terminal symbols with lowercase letters $a, b, c, \dots$, with possible subscripts. Often, grammars are represented by a list of rules, where the variable at the left-hand side of the first rule is considered the initial symbol. Also, rules with common left-hand side are usually described together, in compact form, e.g., the two rules $X \to u$ and $X \to v$ are represented by $X \to u \mid v$.

The notation $u \to_G^* w$ represents the fact that the word u can be transformed into the word w by applying the rules of G, as well as a specific derivation from u into w, depending on the context. A derivation is leftmost if, at each step of the derivation, the variable being rewritten is the one occurring leftmost. The language generated by G is $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \to_G^* w\}$, and G is ambiguous if there exist two different leftmost derivations from S to a word $w \in \mathcal{L}(G)$ (recall that this definition is equivalent to say that there exist two different derivation trees of the same word). For any word w, $D(G, X, w)$ denotes the number of leftmost derivations of G from X to w. We may omit the parameter G when it is clear from the context, and X when it is the start symbol of G. A variable is useless if it does not appear in any derivation of the form $S \to_G^* w \in \Sigma^*$. Unit rules and ε-rules are rules of the form $X \to Y$ and $X \to \varepsilon$, respectively. A CFG is in Chomsky Normal Form (CNF) if all its rules are of the form $X \to YZ$ or $X \to a$. We assume that the reader knows the classical transformations on grammars removing useless variables, ε-rules and unit rules, and the conversion to CNF.

We recall some basic definitions of top-down parsing [1, 4]. We fix a specific symbol \$ out from Σ as an end-of-input mark. For $u \in (V \cup \Sigma \cup \{\$\})^+$, by $\mathsf{First}(G, u)$ we denote $\{a \in (\Sigma \cup \{\$\}) \mid \exists w : u \to_G^* aw\}$. This is generalized to sets of words $U \subseteq (V \cup \Sigma \cup \{\$\})^+$ as $\mathsf{First}(G, U) = \bigcup_{u \in U} \mathsf{First}(G, u)$. For each $X \in V$, by $\mathsf{Follow}(G, X)$ we denote $\{a \in (\Sigma \cup \{\$\}) \mid \exists w_1 \in (V \cup \Sigma)^*, w_2 \in (V \cup \Sigma \cup \{\$\})^* : S\$ \to_G^* w_1 X a w_2\}$. When G is clear from the context, we may omit the parameter G and just write $\mathsf{First}(u)$, $\mathsf{First}(U)$, or $\mathsf{Follow}(X)$.

To denote abstract syntax trees (ASTs) we use the concept of forests, that are finite structures defined recursively as follows, in combination with the definition of trees. A forest F is a (possibly empty) list of trees $t_1 t_2 \cdots t_n$. A tree is an element of the form $a(F)$, where a is an alphabet symbol and F is a forest. The tree $a()$ (i.e., a tree without children) is simply denoted as a. Concatenation of forests F_1 and F_2 is simply denoted $F_1 F_2$. A forest F with just one tree t is identified with t, i.e., F and t are considered identical elements (a list of one element and the element itself are considered to represent the same concept).

When explicitly writing a tree t, we usually separate the trees of the forests occurring inside t with commas. For example, we will prefer to write $a(b, c(d, e), f)$ rather than $a(bc(de)f)$.

2.1 The hashing method

We describe here in detail the hashing method that was briefly presented in [3] for testing equivalence of two given CFGs G_1, G_2. We illustrate the constructions with the CFGs of the following running example.

Example 2.1 *Consider the exercise asking for an unambiguous CFG generating the language $\{a^i b^j \mid i \geq j\}$. Assume that the problem setter has prepared the following unambiguous CFG G_{ref} as reference solution:*

$$\begin{array}{l} S \to aS \mid X \\ X \to aXb \mid \varepsilon \end{array}$$

and that a student submits the following CFG G_{sub} as proposal of solution:

$$T \to aT \mid aTb \mid \varepsilon$$

Note that $\mathcal{L}(G_{\mathsf{ref}}) = \mathcal{L}(G_{\mathsf{sub}}) = \{a^i b^j \mid i \geq j\}$, but G_{sub} is ambiguous since $D(G_{\mathsf{sub}}, aab) = 2$. Thus, the judge should produce a rejection verdict together with a counterexample, preferably one of minimum size like aab.

As a first step, the method transforms the two given CFGs G_1, G_2 to CNF. This transformation preserves the generated language, except for the empty word ε that is no longer generated. Thus, for the particular case of ε we must check whether it is generated by G_1 or by G_2, and how many times, before continuing. The transformation to CNF also preserves ambiguity and unambiguity, except for certain ill cases of ambiguity that can be detected. For example, the CFG

$$\begin{array}{l} S \to XY \mid a \\ X \to S \\ Y \to \varepsilon \end{array}$$

generates a in infinitely many different ways, but the process removing ε-rules and unit rules produces the unambiguous CFG $S \to a$.

Example 2.2 *We illustrate the transformation process to CNF of the CFGs G_{ref} and G_{sub} of Example 2.1 as follows:*

$$G_{\mathsf{ref}} : \left\{ \begin{array}{l} S \to aS \mid X \\ X \to aXb \mid \varepsilon \end{array} \right\} \xrightarrow{\textit{bound}} \left\{ \begin{array}{l} S \to aS \mid X \\ X \to aY \mid \varepsilon \\ Y \to Xb \end{array} \right\} \xrightarrow{\varepsilon\textit{-delete}} \left\{ \begin{array}{l} S \to aS \mid X \mid a \\ X \to aY \\ Y \to Xb \mid b \end{array} \right\}$$

$$\xrightarrow{\textit{unit-delete}} \left\{ \begin{array}{l} S \to aS \mid aY \mid a \\ X \to aY \\ Y \to Xb \mid b \end{array} \right\} \xrightarrow{\textit{rename}} \left\{ \begin{array}{l} S \to AS \mid AY \mid a \\ X \to AY \\ Y \to XB \mid b \\ A \to a \\ B \to b \end{array} \right\}$$

$$G_{\mathsf{sub}} : \left\{ \begin{array}{l} T \to aT \mid aTb \mid \varepsilon \end{array} \right\} \xrightarrow{\textit{bound}} \left\{ \begin{array}{l} T \to aT \mid aZ \mid \varepsilon \\ Z \to Tb \end{array} \right\} \xrightarrow{\varepsilon\textit{-delete}} \left\{ \begin{array}{l} T \to aT \mid aZ \mid a \\ Z \to Tb \mid b \end{array} \right\}$$

$$\xrightarrow{\textit{unit-delete}} \left\{ \begin{array}{l} T \to aT \mid aZ \mid a \\ Z \to Tb \mid b \end{array} \right\} \xrightarrow{\textit{rename}} \left\{ \begin{array}{l} T \to AT \mid AZ \mid a \\ Z \to TB \mid b \\ A \to a \\ B \to b \end{array} \right\}$$

We also call G_{ref} and G_{sub} to the corresponding CNF CFGs to avoid introducing new names. Note that now $\mathcal{L}(G_{\mathsf{ref}}) = \mathcal{L}(G_{\mathsf{sub}}) = \{a^i b^j \mid i \geq j\} \setminus \{\varepsilon\} = \{a^i b^j \mid i \geq j \ \wedge \ i > 0\}$, and that

G_{ref} and G_{sub} are still unambiguous and ambiguous, respectively, as the same counterexample aab shows.

The hashing method fixes a hashing function h from words to natural numbers defined in a usual way over a natural number $\mathcal{B}$ (the base) and a prime natural number $\mathcal{M}$ (the modulus). To simplify the notation, we assume that all arithmetic operations in this section are done modulo $\mathcal{M}$. The hashing function h is then defined as $h(a_0 \dots a_n) = a_0\mathcal{B}^0 + a_1\mathcal{B}^1 + \dots + a_n\mathcal{B}^n$, where each symbol in the alphabet is implicitly identified with a fixed positive natural number.

The method looks for a word w such that $D(G_1, w) \neq D(G_2, w)$, i.e., a word generated a different number of times (with leftmost derivations) with G_1 and G_2. To this end, it fixes a maximum length L for the size of w, and for each $\ell \leq L$ and each $G \in \{G_1, G_2\}$, it computes $H(G, \ell) = \sum_{|u|=\ell \,\wedge\, u\in\mathcal{L}(G)} D(u)\cdot h(u)$. Note that this corresponds to the addition of all $h(u)$, for words $u \in \mathcal{L}(G)$ of length ℓ, by considering each u as many times as it is generated.

Example 2.3 *Consider the resulting CFGs in CNF from Example 2.2, that is:*

$$G_{\mathsf{ref}} : \left\{ \begin{array}{l} S \rightarrow AS \,|\, AY \,|\, a \\ X \rightarrow AY \\ Y \rightarrow XB \,|\, b \\ A \rightarrow a \\ B \rightarrow b \end{array} \right\} \qquad G_{\mathsf{sub}} : \left\{ \begin{array}{l} T \rightarrow AT \,|\, AZ \,|\, a \\ Z \rightarrow TB \,|\, b \\ A \rightarrow a \\ B \rightarrow b \end{array} \right\}$$

The hashing method obtains the following values of $H(G_{\mathsf{ref}}, \ell)$ and $H(G_{\mathsf{sub}}, \ell)$:

	$\ell = 1$	$\ell = 2$	$\ell = 3$
$H(G_{\mathsf{ref}}, \ell)$	a	$(a + a\mathcal{B}) + (a + b\mathcal{B})$	$(a + a\mathcal{B} + a\mathcal{B}^2) + (a + a\mathcal{B} + b\mathcal{B}^2)$
$H(G_{\mathsf{sub}}, \ell)$	a	$(a + a\mathcal{B}) + (a + b\mathcal{B})$	$(a + a\mathcal{B} + a\mathcal{B}^2) + 2(a + a\mathcal{B} + b\mathcal{B}^2)$

It stops for $\ell = 3$ since $H(G_{\mathsf{ref}}, 3) \neq H(G_{\mathsf{sub}}, 3)$ (unless an unfortunate selection of $\mathcal{B}, \mathcal{M}$ and the values corresponding to a and b produces a hash collision).

The values $H(G, \ell)$ are obtained by computing the following values for each variable X of G (where the parameter G is left implicit to ease the presentation):

$$\begin{aligned} C(X, \ell) &= \textstyle\sum_{|u|=\ell \,\wedge\, X \rightarrow^*_G u \in \Sigma^*} D(X, u) \\ H(X, \ell) &= \textstyle\sum_{|u|=\ell \,\wedge\, X \rightarrow^*_G u \in \Sigma^*} D(X, u) \cdot h(u) \end{aligned}$$

These values can be computed as follows, where δ is the set of rules of G:

$$\begin{aligned} C(X, \ell) = &\textstyle\sum_{(X \rightarrow YZ)\in\delta \,\wedge\, 1 \leq \ell' \leq \ell - 1} (C(Y, \ell') \cdot C(Z, \ell - \ell'))+ \\ &\textstyle\sum_{X \rightarrow a \in \delta \,\wedge\, \ell = 1} 1 \\ H(X, \ell) = &\textstyle\sum_{(X \rightarrow YZ)\in\delta \,\wedge\, 1 \leq \ell' \leq \ell - 1} (H(Y, \ell') \cdot C(Z, \ell - \ell')+ \\ &\qquad\qquad\qquad\qquad \mathcal{B}^{\ell'} \cdot C(Y, \ell') \cdot H(Z, \ell - \ell'))+ \\ &\textstyle\sum_{X \rightarrow a \in \delta \,\wedge\, \ell = 1} h(a) \end{aligned}$$

Using memoization, the above values can be computed for all $\ell \leq L$ in $\mathcal{O}(|G| \cdot L^2)$, where $|G|$ denotes the number of rules of G. Note that $H(G, \ell)$ coincides with $H(S, \ell)$, where S is the start symbol of G. Thus, we can compute and compare the pairs $H(G_1, \ell), H(G_2, \ell)$ in $\mathcal{O}((|G_1| + |G_2|) \cdot L^2)$. If $H(G_1, \ell) \neq H(G_2, \ell)$ for some ℓ, we conclude that there exists a word w of length ℓ such that $D(G_1, w) \neq D(G_2, w)$. In the particular case where G_1 is unambiguous, we conclude that either $\mathcal{L}(G_1) \neq \mathcal{L}(G_2)$ or that G_2 is ambiguous, and call any such word w a *counterexample* to the correctness of G_2.

Example 2.4 *The hashing method obtains the following values of $C(W, \ell)$ for each variable W of the CFGs G_{ref} and G_{sub} of Example 2.3. Note that, before, we intentionally used distinct variable names for each CFG in order to avoid the need to specify the corresponding CFG names here (except for the variables A and B, that have identical rules in both CFGs).*

	$\ell = 1$	$\ell = 2$	$\ell = 3$
$C(A, \ell)$	1	0	0
$C(B, \ell)$	1	0	0
$C(Z, \ell)$	1	1	2
$C(T, \ell)$	1	2	3
$C(X, \ell)$	0	1	0
$C(Y, \ell)$	1	0	1
$C(S, \ell)$	1	2	2

Note that $C(S, 3) \neq C(T, 3)$. This already implies that $D(G_{\mathsf{ref}}, w) \neq D(G_{\mathsf{sub}}, w)$ for some word w holding $|w| = 3$. Thus, for this case, it is not necessary to compute the values H in order to realize that; nevertheless, we show them to make the example more complete:

	$\ell = 1$	$\ell = 2$	$\ell = 3$
$H(A, \ell)$	a	0	0
$H(B, \ell)$	b	0	0
$H(Z, \ell)$	b	$a + b\mathcal{B}$	$(a + a\mathcal{B} + b\mathcal{B}^2) + (a + b\mathcal{B} + b\mathcal{B}^2)$
$H(T, \ell)$	a	$(a + a\mathcal{B}) + (a + b\mathcal{B})$	$(a + a\mathcal{B} + a\mathcal{B}^2) + 2(a + a\mathcal{B} + b\mathcal{B}^2)$
$H(X, \ell)$	0	$a + b\mathcal{B}$	0
$H(Y, \ell)$	b	0	$a + b\mathcal{B} + b\mathcal{B}^2$
$H(S, \ell)$	a	$(a + a\mathcal{B}) + (a + b\mathcal{B})$	$(a + a\mathcal{B} + a\mathcal{B}^2) + (a + a\mathcal{B} + b\mathcal{B}^2)$

Thus, we obtain $H(S, 3) \neq H(T, 3)$ (in the absence of hash collisions), and this again proves $D(G_{\mathsf{ref}}, w) \neq D(G_{\mathsf{sub}}, w)$ for some word w holding $|w| = 3$.

The hashing method also allows to efficiently construct such counterexample w. This is done iteratively by first constructing $w[1]$, then $w[2]$, then $w[3]$, and so on, as follows. Suppose that we have already constructed $w[1], w[2], \ldots, w[i-1]$ for $1 \leq i \leq \ell$. Now, we consider each symbol $a \in \Sigma$ and check whether $p = w[1] \cdots w[i-1] \cdot a$ is a prefix of some word w' of length ℓ such that $D(G_1, w') \neq D(G_2, w')$. To this end, for each $G \in \{G_1, G_2\}$ we compute $H(G, p, \ell) = \sum_{|u|=\ell \,\wedge\, u[1\ldots|p|]=p \,\wedge\, u \in \mathcal{L}(G)} D(u) \cdot h(u)$. When for some $a \in \Sigma$ the corresponding p satisfies $H(G_1, p, \ell) \neq H(G_2, p, \ell)$, then this a is a valid choice for $w[i]$, and we can proceed to construct $w[i+1]$ analogously. In order to compute $H(G, p, \ell)$, we need to generalize the above computation to obtain the following values, where $1 \leq j \leq \ell$ and $1 \leq \ell' \leq \ell - j + 1$:

$$
\begin{aligned}
C(X, p, j, \ell') &= \textstyle\sum_{|u|=\ell' \,\wedge\, X \to^*_G u \in \Sigma^* \,\wedge\, (j > |p| \,\vee\, \mathsf{match}(u, p[j \ldots |p|]))} D(X, u) \\
H(X, p, j, \ell') &= \textstyle\sum_{|u|=\ell' \,\wedge\, X \to^*_G u \in \Sigma^* \,\wedge\, (j > |p| \,\vee\, \mathsf{match}(u, p[j \ldots |p|]))} D(X, u) \cdot h(u)
\end{aligned}
$$

Note that for $j > |p|$ the values $C(X, p, j, \ell')$ and $H(X, p, j, \ell')$ coincide with the previously computed values $C(X, \ell')$ and $H(X, \ell')$, respectively. For the cases $j \leq |p|$ we can proceed as follows:

$$
\begin{aligned}
C(X, p, j, \ell') &= \textstyle\sum_{(X \to YZ) \in \delta \,\wedge\, 1 \leq \ell'' \leq \ell' - 1} (C(Y, p, j, \ell'') \cdot C(Z, p, j + \ell'', \ell' - \ell''))+ \\
&\quad \textstyle\sum_{X \to a \in \delta \,\wedge\, \ell' = 1 \,\wedge\, p[j] = a} 1 \\
H(X, p, j, \ell') &= \textstyle\sum_{(X \to YZ) \in \delta \,\wedge\, 1 \leq \ell'' \leq \ell' - 1} (H(Y, p, j, \ell'') \cdot C(Z, p, j + \ell'', \ell' - \ell'')+ \\
&\qquad\qquad \mathcal{B}^{\ell''} \cdot C(Y, p, j, \ell'') \cdot H(Z, p, j + \ell'', \ell' - \ell''))+ \\
&\quad \textstyle\sum_{X \to a \in \delta \,\wedge\, \ell' = 1 \,\wedge\, p[j] = a} h(a)
\end{aligned}
$$

For a fixed p, the above values can be computed for all such j, ℓ' in $\mathcal{O}(|G| \cdot \ell^3)$. Since this is done for each $G \in \{G_1, G_2\}$ and repeated to obtain each of the symbols of w, the total time to construct w is in $\mathcal{O}((|G_1| + |G_2|) \cdot |\Sigma| \cdot \ell^4)$.

Example 2.5 *Continuing from Example 2.4, we have to generate a counterexample w of size 3. We start by considering the case where w starts with an a. Thus, we fix $p = a$ and compute the values $C(S, p, 1, 3)$ and $C(T, p, 1, 3)$:*

$$\begin{aligned}C(S,p,1,3) &= C(A,p,1,1)\cdot C(S,p,2,2) + C(A,p,1,2)\cdot C(S,p,3,1) + \\ &\quad C(A,p,1,1)\cdot C(Y,p,2,2) + C(A,p,1,2)\cdot C(Y,p,3,1) \\ &= 1\cdot C(S,2) + 0\cdot C(S,1) + 1\cdot C(Y,2) + 0\cdot C(Y,1) = 2\end{aligned}$$

$$\begin{aligned}C(T,p,1,3) &= C(A,p,1,1)\cdot C(T,p,2,2) + C(A,p,1,2)\cdot C(T,p,3,1) + \\ &\quad C(A,p,1,1)\cdot C(Z,p,2,2) + C(A,p,1,2)\cdot C(Z,p,3,1) \\ &= 1\cdot C(T,2) + 0\cdot C(T,1) + 1\cdot C(Z,2) + 0\cdot C(Z,1) = 3\end{aligned}$$

Since these values differ, we know that there exists a word w of size 3 starting with an a and holding $D(G_{\mathsf{ref}}, w) \neq D(G_{\mathsf{sub}}, w)$. We do not need to compute the values of H here or in the remaining of the example to realize that. Thus, for simplification purposes, we do not show their computation.

Now, we consider the case where the second symbol of w is an a. Thus, we fix $p = aa$ and compute the values $C(S, p, 1, 3)$ and $C(T, p, 1, 3)$:

$$\begin{aligned}C(S,p,1,3) &= C(A,p,1,1)\cdot C(S,p,2,2) + C(A,p,1,2)\cdot C(S,p,3,1) + \\ &\quad C(A,p,1,1)\cdot C(Y,p,2,2) + C(A,p,1,2)\cdot C(Y,p,3,1) \\ &= 1\cdot C(S,p,2,2) + 0\cdot C(S,1) + 1\cdot C(Y,p,2,2) + 0\cdot C(Y,1) \\ &= C(A,p,2,1)\cdot C(S,p,3,1) + C(A,p,2,1)\cdot C(Y,p,3,1) + \\ &\quad C(X,p,2,1)\cdot C(B,p,3,1) \\ &= 1\cdot C(S,1) + 1\cdot C(Y,1) + 0\cdot C(B,1) = 2\end{aligned}$$

$$\begin{aligned}C(T,p,1,3) &= C(A,p,1,1)\cdot C(T,p,2,2) + C(A,p,1,2)\cdot C(T,p,3,1) + \\ &\quad C(A,p,1,1)\cdot C(Z,p,2,2) + C(A,p,1,2)\cdot C(Z,p,3,1) \\ &= 1\cdot C(T,p,2,2) + 0\cdot C(T,1) + 1\cdot C(Z,p,2,2) + 0\cdot C(Z,1) \\ &= C(A,p,2,1)\cdot C(T,p,3,1) + C(A,p,2,1)\cdot C(Z,p,3,1) + \\ &\quad C(T,p,2,1)\cdot C(B,p,3,1) \\ &= 1\cdot C(T,1) + 1\cdot C(Z,1) + 1\cdot C(B,1) = 3\end{aligned}$$

Since these values differ, we know that there exists a word w of size 3 starting with aa and holding $D(G_{\mathsf{ref}}, w) \neq D(G_{\mathsf{sub}}, w)$. Now, we consider the case where the third symbol of w is also an a. Thus, we fix $p = aaa$ and compute the values $C(S, p, 1, 3)$ and $C(T, p, 1, 3)$:

$$\begin{aligned}C(S,p,1,3) &= C(A,p,1,1)\cdot C(S,p,2,2) + C(A,p,1,2)\cdot C(S,p,3,1) + \\ &\quad C(A,p,1,1)\cdot C(Y,p,2,2) + C(A,p,1,2)\cdot C(Y,p,3,1) \\ &= 1\cdot C(S,p,2,2) + 0\cdot C(S,p,3,1) + 1\cdot C(Y,p,2,2) + 0\cdot C(Y,p,3,1) \\ &= C(A,p,2,1)\cdot C(S,p,3,1) + C(A,p,2,1)\cdot C(Y,p,3,1) + \\ &\quad C(X,p,2,1)\cdot C(B,p,3,1) \\ &= 1\cdot 1 + 1\cdot 0 + 0\cdot 0 = 1\end{aligned}$$

$$\begin{aligned}C(T,p,1,3) &= C(A,p,1,1)\cdot C(T,p,2,2) + C(A,p,1,2)\cdot C(T,p,3,1) + \\ &\quad C(A,p,1,1)\cdot C(Z,p,2,2) + C(A,p,1,2)\cdot C(Z,p,3,1) \\ &= 1\cdot C(T,p,2,2) + 0\cdot C(T,p,3,1) + 1\cdot C(Z,p,2,2) + 0\cdot C(Z,p,3,1) \\ &= C(A,p,2,1)\cdot C(T,p,3,1) + C(A,p,2,1)\cdot C(Z,p,3,1) + \\ &\quad C(T,p,2,1)\cdot C(B,p,3,1) \\ &= 1\cdot 1 + 1\cdot 0 + 1\cdot 0 = 1\end{aligned}$$

Since this values coincide, the third symbol of the counterexample is not an a. Hence, we consider now the case where the third symbol of w is a b. Thus, we fix $p = aab$ and compute *the values $C(S,p,1,3)$ and $C(T,p,1,3)$ again:*

$$\begin{aligned}
C(S,p,1,3) &= C(A,p,1,1)\cdot C(S,p,2,2) + C(A,p,1,2)\cdot C(S,p,3,1)+\\
&\quad C(A,p,1,1)\cdot C(Y,p,2,2) + C(A,p,1,2)\cdot C(Y,p,3,1)\\
&= 1\cdot C(S,p,2,2) + 0\cdot C(S,p,3,1) + 1\cdot C(Y,p,2,2) + 0\cdot C(Y,p,3,1)\\
&= C(A,p,2,1)\cdot C(S,p,3,1) + C(A,p,2,1)\cdot C(Y,p,3,1)+\\
&\quad C(X,p,2,1)\cdot C(B,p,3,1)\\
&= 1\cdot 0 + 1\cdot 1 + 0\cdot 1 = 1\\
C(T,p,1,3) &= C(A,p,1,1)\cdot C(T,p,2,2) + C(A,p,1,2)\cdot C(T,p,3,1)+\\
&\quad C(A,p,1,1)\cdot C(Z,p,2,2) + C(A,p,1,2)\cdot C(Z,p,3,1)\\
&= 1\cdot C(T,p,2,2) + 0\cdot C(T,p,3,1) + 1\cdot C(Z,p,2,2) + 0\cdot C(Z,p,3,1)\\
&= C(A,p,2,1)\cdot C(T,p,3,1) + C(A,p,2,1)\cdot C(Z,p,3,1)+\\
&\quad C(T,p,2,1)\cdot C(B,p,3,1)\\
&= 1\cdot 0 + 1\cdot 1 + 1\cdot 1 = 2
\end{aligned}$$

Since these values differ, we conclude that $w = aab$ satisfies $D(G_{\mathsf{ref}}, w) \neq D(G_{\mathsf{sub}}, w)$. We have obtained the expected counterexample mentioned at the beginning.

Although the hashing method behaves well in many practical situations, it fails in some cases independently of the chosen $\mathcal{B}, \mathcal{M}$, like in this example presented in [3]:

Example 2.6 *Consider any two unambiguous CFGs G_1, G_2 in CNF generating the languages $L_1 = \{a^n b^n \mid n \geq 0\} \cup \{c^n d^n \mid n \geq 0\}$ and $L_2 = \{a^n d^n \mid n \geq 0\} \cup \{c^n b^n \mid n \geq 0\}$, respectively. For all $\ell \geq 0$, the pairs $H(G_1, \ell), H(G_2, \ell)$ coincide, although $L_1 \neq L_2$. This is because the contribution of each symbol s at each position depends only on how many times s occurs at such position, regardless of which symbols appear before and after each of the occurrences. For example, $H(G_1, 2) = h(ab) + h(cd) = (a + b\mathcal{B}) + (c + d\mathcal{B}) = (a + d\mathcal{B}) + (c + b\mathcal{B}) = h(ad) + h(cb) = H(G_2, 2)$.*

In Section 5.4 we show how to solve some of these anomalies by extending the hashing method with an adequate use of a morphism.

3 The model of parser generator

Our model of parser generator admits grammars described with a special syntax borrowed from the PCCTS/ANTLR3 tools. We have chosen it due to its simplicity and ability to describe the AST construction process in a natural way. We call such grammars PCFGs. Each variable X of a PCFG is at the left-hand side of exactly one rule, denoted $\mathsf{Rule}(X)$, whose right-hand side is a parenthesized expression over variables, terminals, and the operators $|, \cdot, *$ (listed in increasing order of precedence). Alternative $|$ and concatenation $\cdot$ (sometimes omitted) are binary infix operators, whereas Kleene star $*$ is postfix and unary. In contrast to the introduction and to ease the presentation, we use uppercase letters $X, Y, Z, A, B, C, \ldots$ instead of arbitrary lowercase identifiers to denote variables, as well as lowercase letters $a, b, c, \ldots$ instead of uppercase identifiers and quoted strings to denote tokens/terminals. In some examples, we may also use $=, +, \bullet, \triangle, \square, \#$ as tokens. Each token can be followed by either ˆ (to indicate that it must become the root of the AST) or ! (to denote that it must not be included in the AST), and variables in left-hand sides can be followed by ˆ (to indicate that the variable must become the root of the constructed AST for each part of the input parsed with that variable). The latter use of ˆ is not an option of the PCCTS/ANTLR3 tools, but we have incorporated it for its usefulness and simplicity.

Example 3.1 *The following PCFG describes lists of instructions, where each instruction is an assignment (token $=$) of a number (n) to an identifier (i):*

$$\begin{aligned}
L\hat{} &\rightarrow I*\\
I &\rightarrow i{=}\hat{}n
\end{aligned}$$

By using $L\hat{}$ as left-hand side of the first rule, the ASTs produced for any input will be of the form $L(=(i,n), =(i,n), \ldots)$. Note that the variable symbol L is in the AST although it is not a token.

To ease the presentation, we consider a normalized form for PCFGs where each subexpression of the right-hand side is replaced by a new variable. Moreover, we mark original variables X as $\bar{X}$, since, in order to describe the AST construction process, the parser generator needs to distinguish which variables are original from which ones have been generated by the transformation. This way, rules of a normalized PCFG are of the form $X \to Y \mid Z$, $X \to YZ$, $X \to Y*$, $X \to a$, $X \to a\hat{}$, $X \to a!$, $X \to \varepsilon$, $X \to \bar{Y}$, $\bar{X} \to Y$, or $\bar{X}\hat{} \to Y$. Note that original variables in a normalized PCFG (i.e., those variables marked like $\bar{X}$) only appear in unit productions. This fact allows to simplify later transformations.

Example 3.2 *The normalized form for the PCFG of Example 3.1 is:*

$$\begin{array}{lll} \bar{L}\hat{} \to L_\lambda & \bar{I} \to I_\lambda & I_{11} \to i \\ L_\lambda \to L_1 * & I_\lambda \to I_1 I_2 & I_{12} \to =\hat{} \\ L_1 \to \bar{I} & I_1 \to I_{11} I_{12} & I_2 \to n \end{array}$$

The notions of generated language, First and Follow of CFGs are adapted to PCFGs by implicitly assuming that each rule of the form $X \to Y*$ can be replaced by $X \to YX \mid \varepsilon$. The interpretation of a normalized PCFG as a parser is given by Definition 3.3.

Definition 3.3 *Let G be a normalized PCFG. For each variable X of G, we define* Code(X) *depending on the form of* Rule(X) *as detailed in Table 1.*

The parsed language of G, *denoted $\mathcal{L}_\mathcal{P}(G)$, is the set of words $w \in \Sigma^*$ such that $w\$$ is entirely consumed without error when calling* Code(S) *and then* `consumeToken($)`*, where S is the start symbol of G. Similarly, the* partially parsed language of G, *denoted $\mathcal{L}_{\mathcal{PP}}(G)$, is the set of non-empty words $w \in (\Sigma \cup \{\$\})^+$ such that w is entirely consumed by the previous calls before producing any error. We say that G does* eager detection of syntax errors *if it produces a syntax error as soon as the currently consumed word cannot be extended to a word of the parsed language, i.e., if $\mathcal{L}_{\mathcal{PP}}(G) =$ Prefixes$(\mathcal{L}_\mathcal{P}(G)\$) \setminus \{\varepsilon\}$.*

Note that non-eager detection of syntax errors requires PCFGs with unnatural constructions like, e.g., $S \to a{*}a$. This example produces a syntax error on any word of the form $aa \cdots a$, but the error is only detected after consuming the whole input. Usually, such constructions do not produce a correct parser for the intended language. Also note that $\mathcal{L}_{\mathcal{PP}}(G_1) \neq \mathcal{L}_{\mathcal{PP}}(G_2)$ implies that either $\mathcal{L}_\mathcal{P}(G_1) \neq \mathcal{L}_\mathcal{P}(G_2)$ or one of G_1, G_2 does not produce syntax errors eagerly.

Example 3.4 *The normalized PCFG of Example 3.2 gives rise to the following code:*

```
function $\bar{L}$() returns Forest
  Forest  ast := empty
  Bool rooted := false
  while lookahead()$\in \{i\}$
    f := $\bar{I}$()
    ast.addToForest(f,rooted)
  ast.addAsRoot($L$)
  return ast
```

```
function $\bar{I}$() returns Forest
  Forest  ast := empty
  Bool rooted := false
  t := consumeToken($i$)
  ast.addToForest(t,rooted)
  t := consumeToken($=$)
  ast.addAsRoot(t)
  rooted := true
  t := consumeToken($n$)
  ast.addToForest(t,rooted)
  return ast
```

Note that with input $i{=}ni{=}ni{=}n$ the resulting AST is $L(=^2(i^1,n^3), =^5(i^4,n^6), =^8(i^7,n^9))$, where the superindices have been added to make it easier to identify the original position of each token in the input.

Table 1: Definition of $\mathsf{Code}(X)$ depending on the form of $\mathsf{Rule}(X)$, where X is a variable of a given normalized PCFG. The `lookahead` function returns the next token of the input without consuming it, and `consumeToken` consumes and returns the next token of the input after checking its lexical class. If F is the current forest kept in `ast`, the instruction `ast.addAsRoot(t)` sets `ast` := `t`(F). If, in addition, `rooted` is false, the instruction `ast.addToForest(f,rooted)` sets `ast` := F`f`, and, otherwise, F is a tree of the form $a(F')$ and it sets `ast` := $a(F'$`f`$)$.

- If $\mathsf{Rule}(X) = \bar{X} \to Y$:

```
function $\bar{X}$() returns Forest
  Forest  ast := empty
  Bool rooted := false
  $\mathsf{Code}(Y)$
  return ast
```

- If $\mathsf{Rule}(X) = \bar{X}\hat{} \to Y$:

```
function $\bar{X}$() returns Forest
  Forest  ast := empty
  Bool rooted := false
  $\mathsf{Code}(Y)$
  ast.addAsRoot($X$)
  return ast
```

- If $\mathsf{Rule}(X) = X \to \bar{Y}$:

```
f := $\bar{Y}$()
ast.addToForest(f,rooted)
```

- If $\mathsf{Rule}(X) = X \to Y \mid Z$:

```
if lookahead() $\in \mathsf{First}(Y\mathsf{Follow}(X))$
  $\mathsf{Code}(Y)$
else if lookahead() $\in \mathsf{First}(Z\mathsf{Follow}(X))$
  $\mathsf{Code}(Z)$
else SYNTAXERROR
```

- If $\mathsf{Rule}(X) = X \to YZ$:

```
$\mathsf{Code}(Y)$
$\mathsf{Code}(Z)$
```

- If $\mathsf{Rule}(X) = X \to Y*$:

```
while lookahead() $\in \mathsf{First}(Y)$
  $\mathsf{Code}(Y)$
```

- If $\mathsf{Rule}(X) = X \to a!$:

```
consumeToken($a$)
```

- If $\mathsf{Rule}(X) = X \to a$:

```
t := consumeToken($a$)
ast.addToForest(t,rooted)
```

- If $\mathsf{Rule}(X) = X \to a\hat{}$:

```
t := consumeToken($a$)
ast.addAsRoot(t)
rooted := true
```

- If $\mathsf{Rule}(X) = X \to \varepsilon$:

```
// Nothing to do
```

4 Transformations T and P

We define a transformation T that, given a normalized PCFG G, gives rise to an unambiguous CFG $T(G)$ such that $\mathcal{L}(T(G)) = \mathcal{L}_{\mathcal{P}}(G)$. The transformation T creates new variables of the form X_{ab} that are intended to generate the set of words w such that $a \in \mathsf{First}(wb)$ and $\mathsf{Code}(X)$ runs correctly with input wb having consumed exactly w. The specific production rules of X_{ab} introduced by T depend on $\mathsf{Rule}(X)$. Consider for instance $\mathsf{Rule}(X) = X \to YZ$, and note that any correct execution of $\mathsf{Code}(X)$ on wb consuming w first executes $\mathsf{Code}(Y)$, which consumes a prefix of w, say α, and then $\mathsf{Code}(Z)$, which consumes the remaining part of w, say β. Inductively, the execution of $\mathsf{Code}(Y)$ on $\alpha\beta b$ consuming α and the execution of $\mathsf{Code}(Z)$ on βb consuming β are simulated by variables of the form Y_{ac} and Z_{cb}, where c is the first symbol of βb (and hence $c \in \mathsf{First}(\beta b)$). Since β could be any suffix of w, the transformation T introduces a rule of the form $X_{ab} \to Y_{ac}Z_{cb}$ for each possible symbol c. Note that α, as well as β, could be empty words. Now consider $\mathsf{Rule}(X) = X \to Y \mid Z$, and note that any correct execution of $\mathsf{Code}(X)$ on wb consuming w executes only one of the branches, giving more priority to the first one: $\mathsf{Code}(Y)$ is executed when $a \in \mathsf{First}(Y\mathsf{Follow}(X))$, and $\mathsf{Code}(Z)$ is executed when the previous case is not possible and $a \in \mathsf{First}(Z\mathsf{Follow}(X))$. These executions are simulated by variables of the form Y_{ab} and Z_{ab}, respectively, and hence, the transformation T introduces a single rule for X_{ab} of the form $X_{ab} \to Y_{ab}$ or $X_{ab} \to Z_{ab}$, depending on which branch has to be taken. The following definition formalizes the previous reasoning, extending it to the remaining cases of $\mathsf{Rule}(X)$.

Definition 4.1 *Given a normalized PCFG $G = \langle V, \Sigma, \delta, S \rangle$, the CFG $T(G)$ is defined over Σ, with set of variables $\mathcal{V} = \{S'\} \cup \{X_{ab} \mid X \in V \ \wedge\ b \in \mathsf{Follow}(X) \ \wedge\ a \in \mathsf{First}(Xb)\}$, start symbol S', and the following set of rules:*

$$
\begin{array}{ll}
\{S' \to S_{a\$} & \mid a \in \mathsf{First}(S\$)\} \cup \\
\{X_{ab} \to Y_{ab} & \mid (X \to Y|Z) \in \delta \ \wedge\ b \in \mathsf{Follow}(X) \ \wedge\ a \in \mathsf{First}(Y\mathsf{Follow}(X))\} \cup \\
\{X_{ab} \to Z_{ab} & \mid (X \to Y|Z) \in \delta \ \wedge\ b \in \mathsf{Follow}(X) \ \wedge\ a \notin \mathsf{First}(Y\mathsf{Follow}(X)) \ \wedge \\
 & \ \ a \in \mathsf{First}(Z\mathsf{Follow}(X))\} \cup \\
\{X_{ab} \to Y_{ac}Z_{cb} & \mid (X \to YZ) \in \delta \ \wedge\ b \in \mathsf{Follow}(X) \ \wedge\ c \in \mathsf{First}(Zb) \ \wedge\ a \in \mathsf{First}(Yc)\} \cup \\
\{X_{ab} \to Y_{ac}X_{cb} & \mid (X \to Y*) \in \delta \ \wedge\ b \in \mathsf{Follow}(X) \ \wedge\ c \in \mathsf{First}(Xb) \ \wedge\ a \in \mathsf{First}(Y)\} \cup \\
\{X_{bb} \to \varepsilon & \mid (X \to Y*) \in \delta \ \wedge\ b \in \mathsf{Follow}(X) \wedge\ b \notin \mathsf{First}(Y)\} \cup \\
\{X_{bb} \to \varepsilon & \mid (X \to \varepsilon) \in \delta \ \wedge\ b \in \mathsf{Follow}(X)\} \cup \\
\{X_{ab} \to a & \mid (X \to a) \in \delta \ \wedge\ b \in \mathsf{Follow}(X)\} \cup \\
\{X_{ab} \to Y_{ab} & \mid (X \to Y) \in \delta \ \wedge\ b \in \mathsf{Follow}(X) \ \wedge\ a \in \mathsf{First}(Yb)\}
\end{array}
$$

For clarity, we have omitted the directives for AST construction and we do not distinguish original variables from the ones introduced in the normalization. Nevertheless, it is implicitly assumed that these decorations are preserved by T.

Example 4.2 *Applying T to the normalized PCFG of Example 3.2, and after removing all useless rules and variables, we obtain:*

$$
\begin{array}{rlcrlcrl}
S' & \to \bar{L}_{\$\$} \mid \bar{L}_{i\$} & & & & & & \\
\bar{L}_{\$\$}{}^{\wedge} & \to L_{\lambda,\$\$} & & \bar{I}_{i\$} & \to I_{\lambda,i\$} & & & \\
\bar{L}_{i\$}{}^{\wedge} & \to L_{\lambda,i\$} & & \bar{I}_{ii} & \to I_{\lambda,ii} & & I_{11,i=} & \to i \\
L_{\lambda,\$\$} & \to \varepsilon & & I_{\lambda,i\$} & \to I_{1,in}I_{2,n\$} & & I_{12,=n} & \to ={}^{\wedge} \\
L_{\lambda,i\$} & \to L_{1,i\$}L_{\lambda,\$\$} \mid L_{1,ii}L_{\lambda,i\$} & & I_{\lambda,ii} & \to I_{1,in}I_{2,ni} & & I_{2,n\$} & \to n \\
L_{1,i\$} & \to \bar{I}_{i\$} & & I_{1,in} & \to I_{11,i=}I_{12,=n} & & I_{2,ni} & \to n \\
L_{1,ii} & \to \bar{I}_{ii} & & & & & &
\end{array}
$$

Lemma 4.3 *Let $G = \langle V, \Sigma, \delta, S \rangle$ be a normalized PCFG, and let $G' = T(G)$. Then, G' is an unambiguous CFG such that $\mathcal{L}(G') = \mathcal{L}_{\mathcal{P}}(G)$.*

Proof sketch. By definition, G' is a CFG that simulates the behaviour of the parser generated from G by conjecturing the first unread symbol at each step. There is a bijection between the executions of the parser and the terminal derivations with G'. This concludes $\mathcal{L}(G') = \mathcal{L}_{\mathcal{P}}(G)$. Unambiguity follows from the fact that the parser is a deterministic program: a parsed word w has only one execution, and hence, only one leftmost derivation with G'. □

As a consequence of Lemma 4.3, given two normalized PCFGs G_1, G_2, in order to test $\mathcal{L}_{\mathcal{P}}(G_1) = \mathcal{L}_{\mathcal{P}}(G_2)$, we can equivalently test $\mathcal{L}(T(G_1)) = \mathcal{L}(T(G_2))$ using the hashing method. This method is expected to produce a counterexample w to $\mathcal{L}(T(G_1)) = \mathcal{L}(T(G_2))$, when one exists satisfying $|w| \leq L$. Since the cost of the method has a factor L^4, only small counterexamples can be considered. For this reason, we focus on an alternative approach that does not need to produce the entire w, but only a prefix of w that suffices to determine $\mathcal{L}(T(G_1)) \neq \mathcal{L}(T(G_2))$. One possibility would be to transform $T(G_1)$ and $T(G_2)$ into new unambiguous CFGs generating the corresponding sets of prefixes, and test equivalence again. There are simple standard procedures on CFGs to this end, but, unfortunately, they do not preserve unambiguity. We present a direct transformation P on the initial normalized PCFG G that produces an unambiguous CFG $P(G)$ satisfying $\mathcal{L}(P(G)) = \mathcal{L}_{\mathcal{PP}}(G)$.

Thus, in particular, $\mathcal{L}(P(G)) = \mathsf{Prefixes}(\mathcal{L}(T(G))\$) \setminus \{\varepsilon\}$ if G does eager detection of syntax errors. The transformation P creates variables of the form X_{ab} with the same interpretation as transformation T, and also variables of the form X_a that are intended to generate the set of non-empty words w such that a is the first symbol of w (and hence $a \in \mathsf{First}(w)$) and there is a partial execution of $\mathsf{Code}(X)$ that consumes exactly w and does not produce syntax error. The specific production rules of X_a introduced by P depend on $\mathsf{Rule}(X)$. Consider for instance $\mathsf{Rule}(X) = X \to YZ$, and note that there are two possible kinds of partial executions: either the whole w is consumed while executing $\mathsf{Code}(Y)$, or $\mathsf{Code}(Y)$ consumes a proper prefix of w, say α, and then $\mathsf{Code}(Z)$ consumes the remaining part of w, say β. For the first case, the transformation P introduces a rule of the form $X_a \to Y_a$, and for the second case it introduces a rule of the form $X_a \to Y_{ac}Z_c$ for each symbol c in $\mathsf{First}(\beta)$. The following definition formalizes the previous reasoning, extending it to the remaining cases of $\mathsf{Rule}(X)$.

Definition 4.4 *Given a normalized PCFG $G = \langle V, \Sigma, \delta, S\rangle$, the CFG $P(G)$ is defined over $\Sigma \cup \{\$\}$, with set of variables $\mathcal{V} = \{S'\} \cup \{X_{ab} \mid X \in V \ \wedge\ b \in \mathsf{Follow}(X) \ \wedge\ a \in \mathsf{First}(Xb)\} \cup \{X_a \mid X \in V \ \wedge\ a \in \mathsf{First}(X)\}$, start symbol S', and where the set of rules contains exactly the ones of Definition 4.1 except $\{S' \to S_{a\$} \mid a \in \mathsf{First}(S\$)\}$, and also the following rules:*

$$
\begin{array}{ll}
\{S' \to S_{a\$}\$ & \mid a \in \mathsf{First}(S\$)\} \cup \\
\{S' \to S_a & \mid a \in \mathsf{First}(S)\} \cup \\
\{X_a \to Y_a & \mid (X \to Y|Z) \in \delta \ \wedge\ a \in \mathsf{First}(Y)\} \cup \\
\{X_a \to Z_a & \mid (X \to Y|Z) \in \delta \ \wedge\ a \notin \mathsf{First}(Y\mathsf{Follow}(X)) \ \wedge\ a \in \mathsf{First}(Z)\} \cup \\
\{X_a \to Y_a & \mid (X \to YZ) \in \delta \ \wedge\ a \in \mathsf{First}(Y)\} \cup \\
\{X_a \to Y_{ac}Z_c & \mid (X \to YZ) \in \delta \ \wedge\ c \in \mathsf{First}(Z) \ \wedge\ a \in \mathsf{First}(Yc)\} \cup \\
\{X_a \to Y_{ac}X_c & \mid (X \to Y*) \in \delta \ \wedge\ c \in \mathsf{First}(X) \ \wedge\ a \in \mathsf{First}(Y)\} \cup \\
\{X_a \to Y_a & \mid (X \to Y*) \in \delta \ \wedge\ a \in \mathsf{First}(Y)\} \cup \\
\{X_a \to a & \mid (X \to a) \in \delta\} \cup \\
\{X_a \to Y_a & \mid (X \to Y) \in \delta \ \wedge\ a \in \mathsf{First}(Y)\}
\end{array}
$$

As in Definition 4.1, AST directives are omitted and original variables are not distinguished, but these decorations are implicitly assumed to be preserved by P.

Example 4.5 *Applying P to the normalized PCFG of Example 3.2, and after removing all useless rules and variables, we obtain all the rules of Example 4.2 except $S' \to \bar{L}_{\$\$} \mid \bar{L}_{i\$}$, and the following rules:*

$$
\begin{array}{rcr}
S' \to \bar{L}_{\$\$}\$ \mid \bar{L}_{i\$}\$ \mid \bar{L}_i & & \\
\bar{L}_i\hat{} \to L_{\lambda,i} & \bar{I}_i \to I_{\lambda,i} & I_{11,i} \to i \\
L_{\lambda,i} \to L_{1,i} \mid L_{1,ii}L_{\lambda,i} & I_{\lambda,i} \to I_{1,i} \mid I_{1,in}I_{2,n} & I_{12,=} \to =\hat{} \\
L_{1,i} \to \bar{I}_i & I_{1,i} \to I_{11,i} \mid I_{11,i=}I_{12,=} & I_{2,n} \to n
\end{array}
$$

Lemma 4.6 *Let $G = \langle V, \Sigma, \delta, S\rangle$ be a normalized PCFG, and let $G' = P(G)$. Then, G' is an unambiguous CFG such that $\mathcal{L}(G') = \mathcal{L}_{\mathcal{PP}}(G)$.*

5 Checking equivalence of AST construction

Once we have tested that two PCFGs G_1, G_2 parse the same language, we want to check that they construct the same AST for each parsed word. In Section 5.1 we present a first (incorrect) approach to tackle this problem, and in Section 5.2 we discuss its limitations. We solve its flaws in Sections 5.3 and 5.4 with an additional transformation on CFGs and improvements of the hashing method, respectively.

5.1 A first approach to check equivalence of AST construction

Our goal is to define an adequate formalism representing ASTs as words, and a transformation A' from PCFGs into CFGs such that, given a PCFG G, $A'(G)$ generates the set of words that represent (in this new formalism) the ASTs constructed for the words parsed by G. Next, given two PCFGs G_1, G_2, we can compare $\mathcal{L}(A'(G_1)), \mathcal{L}(A'(G_2))$ using the hashing method to determine if G_1, G_2 construct the same ASTs.

In order to discuss which representation for ASTs is adequate, consider the PCFG of expressions $G : \{E \to T(+\hat{}T)*,\ T \to n(\bullet\hat{}n)*\}$ over addition (token $+$) and product (token $\bullet$) with the usual precedence and left associativity. With the input $n{\bullet}n{\bullet}n{+}n{\bullet}n{+}n{\bullet}n$, the constructed AST is $+^{10}(+^{6}(\bullet^{4}(\bullet^{2}(n^1, n^3), n^5), \bullet^{8}(n^7, n^9)), \bullet^{12}(n^{11}, n^{13}))$, where the superindices have been added to make it easier to identify the original position of each token in the input. Note that this is already a word representing an AST. However, this formalism is not adequate because the symbols are reordered in a way that is difficult to describe by A'. In general, one of the difficulties is that, for each rule $X \to YZ$, the root of a constructed AST could be inside Y or Z, depending on the input word.

To find a better form to represent ASTs, we make the following observation: to deduce how the AST is constructed from an input word w, we just need to know how each token of w has been inserted into the AST (with `addToForest`, with `addAsRoot`, or not inserted), which subwords of w have been entirely consumed by an execution of a procedure of the parser generated from G, and which procedure has inserted a variable symbol into the AST. All this information can be represented by, e.g., appending a symbol $\wedge$ to each symbol inserted with `addAsRoot`, appending a symbol **!** to each symbol consumed but not inserted, enclosing between brackets each subword entirely consumed by a procedure, and adding a variable name just before the closing brackets, if necessary. We call $\mathsf{Tree}(G, w)$, or just $\mathsf{Tree}(w)$ by omitting G, the result of modifying a word w in this way according to G. For our previous example, we have $\mathsf{Tree}(n{\bullet}n{\bullet}n{+}n{\bullet}n{+}n{\bullet}n) = [[n{\bullet}{\wedge}n{\bullet}{\wedge}n]{+}{\wedge}[n{\bullet}{\wedge}n]{+}{\wedge}[n{\bullet}{\wedge}n]]$. Note that the tokens preserve their ordering, and that the constructed AST is implicitly represented.

We can easily describe the transformation A' from PCFGs to CFGs such that $\mathcal{L}(A'(G)) = \{\mathsf{Tree}(G, w) \mid w \in \mathcal{L}_{\mathcal{P}}(G)\}$. It suffices to alter the transformation T by replacing each directive $\hat{}$ by the new terminal symbol $\wedge$, each directive ! by the new terminal symbol **!**, each rule of the form $\bar{X} \to Y$ by $\bar{X} \to [Y]$, and each rule of the form $\bar{X}\hat{} \to Y$ by $\bar{X} \to [Y\boldsymbol{X}\wedge]$ for a new terminal symbol $\boldsymbol{X}$.

Example 5.1 *Applying A' to the normalized PCFG of Example 3.2 and removing all useless rules and variables gives rise to the rules of Example 4.2, except for some of them that are modified as follows:*

$$\begin{array}{lll} \bar{L}_{\$\$}\hat{} \to [L_{\lambda,\$\$}\boldsymbol{L}\wedge] & \bar{I}_{i\$} \to [I_{\lambda,i\$}] & \\ \bar{L}_{i\$}\hat{} \to [L_{\lambda,i\$}\boldsymbol{L}\wedge] & \bar{I}_{ii} \to [I_{\lambda,ii}] & I_{12,=n} \to {=}\wedge \end{array}$$

Unfortunately, A' is not suitable to check that two PCFGs construct the same ASTs due to some anomalies that we describe in the next section.

5.2 Anomalies

First, note that the same AST can be represented by several different words of our formalism. This can be an advantage in some cases. For instance, consider the PCFGs $G_1 : S \to a\hat{}b \mid b\hat{}a$ and $G_2 : S \to ab\hat{} \mid ba\hat{}$. Both parse the same language $\{ab, ba\}$ and construct the same set of ASTs $\{a(b), b(a)\}$, but they are not equivalent since each parsed word gives rise to a different AST. In this case, A' works properly, since $\mathcal{L}(A'(G_1)) = \{[a{\wedge}b], [b{\wedge}a]\} \neq \{[ab\wedge], [ba\wedge]\} = \mathcal{L}(A'(G_2))$. Now, consider a more dubious case with $G_1 : S \to i\hat{}i$ and $G_2 : S \to ii\hat{}$. Again, both PCFGs parse the same word ii and produce the same AST $i(i)$, but $\mathcal{L}(A'(G_1)) = \{[i{\wedge}i]\} \neq \{[ii\wedge]\} = \mathcal{L}(A'(G_2))$. Thus, we will conclude that the constructed ASTs differ.

But this is not a wrong conclusion: in general, the AST is the mechanism to give a structured interpretation to the input, and putting the tokens in different places of the AST corresponds to the representation of a different concept, even if the swapped tokens are of the same class.

The previous examples are not actually anomalies of A', but its desirable behavior. Nevertheless, the fact that different words can represent the same AST can also cause some problems. On the one hand, in some cases a $\hat{}$ is superfluous from the point of view of AST construction, and thus, a $\wedge$ should not be inserted by the transformation. For example, the PCFGs $G_1 : S \rightarrow a\hat{}$ and $G_2 : S \rightarrow a$ produce the same AST for the same word, but $[a\wedge] \neq [a]$. Here, the problem is that for a variable consuming a single token it does not matter whether this token is followed by $\hat{}$ or not. Also, $G_1 : S \rightarrow a\hat{}b\hat{}$ and $G_2 : S \rightarrow ab\hat{}$ produce the same AST for the same word, but $[a\wedge b\wedge] \neq [ab\wedge]$. In general, the $\hat{}$ of the first token read inside a procedure is useless if it is immediately followed by another token with $\hat{}$. On the other hand, not all subwords consumed by procedures should be enclosed between brackets. Consider for instance the PCFGs $G_1 : S \rightarrow abc$ and $G_2 : \{S \rightarrow aXc,\ X \rightarrow b\}$, which produce the same AST for the same word, but $[abc] \neq [a[b]c]$. To solve this problem, it is necessary to have brackets only when the corresponding subword contains a token followed by $\hat{}$. This has to be combined with the solution to the previous anomalies to work properly. For example, in $G : \{S \rightarrow aXc,\ X \rightarrow b\hat{}\}$ the $\hat{}$ is superfluous and should be ignored, and thus, the inner brackets should not be inserted. Another case where brackets are useless is, e.g., the PCFGs $G_1 : S \rightarrow ab\hat{}c\hat{}$ and $G_2 : \{S \rightarrow Xc\hat{},\ X \rightarrow ab\hat{}\}$, where the ASTs are equal but $[ab\wedge c\wedge] \neq [[ab\wedge]c\wedge]$. In general, the inner brackets of an AST of the form $[[w]c\wedge\ldots]$ should be removed. In Section 5.3 we give a transformation A on the CFG resulting from T or P that solves all the previous anomalies.

Nevertheless, the most severe problems arise with symbols followed by !. For instance, the PCFGs $G_1 : \{S \rightarrow a\hat{}b!X,\ X \rightarrow c\hat{}d\}$ and $G_2 : \{S \rightarrow a\hat{}X,\ X \rightarrow b!c\hat{}d\}$ generate the same AST for the same input word, but $[a\wedge b\mathbf{!}[c\wedge d]] \neq [a\wedge[b\mathbf{!}c\wedge d]]$. To solve these cases in general, we would need to be able to move a symbol followed by ! any number of brackets inside or outside. However, this cannot be easily managed by a transformation on CFGs. Another option would be to modify A' so that those symbols are removed. This seems to make sense because these symbols do not take part in the AST, which is what we are just trying to represent. But this may have undesirable side effects. On the one hand, since the formalism would not keep the original w, false positives may occur. For example $G_1 : S \rightarrow a!c\hat{}d \mid b!cd\hat{}$ and $G_2 : S \rightarrow a!cd\hat{} \mid b!c\hat{}d$ construct different ASTs for each word, but the set of represented ASTs after removing the symbols followed by ! is $\{[c\wedge d], [cd\wedge]\}$ for both PCFGs. On the other hand, unambiguity may be lost. For instance, after applying the transformation that removes the symbols followed by ! to the PCFG $G : S \rightarrow a!b \mid b$, it holds that the AST b is generated twice. This particular example is not problematic since the hashing method counts each word as many times as it is generated, so it will distinguish such G from, e.g., the PCFG $G' : S \rightarrow b$. But there are cases where the loss of ambiguity may cause the hashing method to be inapplicable. For instance, with $G : S \rightarrow (b!)*$ the resulting CFG would generate the empty word infinitely many times. Summarizing, we need to discard symbols followed by ! without removing them from the CFG. In Section 5.4 we face this problem by generalizing the hashing method to compute $H([\sigma(w) \mid |w| \leq L \ \wedge \ w \in \mathcal{L}[G]])$ depending on a morphism σ. This way, unambiguity is preserved, and by defining σ so that these symbols followed by ! are mapped to ε, all the false negatives disappear.

5.3 Transformation A

As justified in the previous discussion, the transformation A has two main goals. On the one hand, it has to identify those $\hat{}$ that lead to degenerate ASTs of the form $[a\wedge]$ or $[a\wedge b\wedge\ldots]$, and erase them. On the other hand, and after having erased the appropriate $\hat{}$, it has to detect which of the original variables need to introduce brackets. Recall that brackets are only needed when there is some occurrence of $\wedge$, except in an AST of the form $[[w]a\wedge\ldots]$, where the inner brackets surrounding w are unnecessary even if w has occurrences of $\wedge$. To simplify the presentation, we define A as the composition of two transformations, $A_2 \circ A_1$. Intuitively, the goal of A_1 is to erase the superfluous ˆ, and to add information to the variables of the grammar to easily distinguish when an original variable generates a non-empty AST, and moreover, when such an AST has a defined root (i.e., it has generated a $\wedge$). In the case of A_2, it further adds information to the variables of the grammar to detect when an original variable generates an AST x that is always placed inside an AST of the form $[xa\wedge\ldots]$, and thus, x should not be bracketed even if it has occurrences of $\wedge$.

We start describing A_1. To this end, we need to introduce notation to easily distinguish which kind of tokens are generated by a variable. In particular, we are interested in distinguishing whether the generated tokens have a decorator, and in the case they have a decorator, whether it is a !, or a superfluous ˆ, or a useful ˆ. We use the following symbols:

$*$ denotes that anything is generated (maybe nothing).

$!$ denotes that only !-tokens are generated (if any).

$\vee$ denotes that any number of !-tokens and exactly one non-!-token are generated, and the latter has a ˆ that is superfluous, and thus, must be erased.

$\wedge$ analogous to $\vee$ but the ˆ is useful and must be preserved.

Σ denotes that any number of !-tokens and either a non-!-token without a ˆ or a non-empty sub-AST are generated.

Σ^+ analogous to Σ but with possible repetitions.

Note that in all the cases there might be !-tokens, and thus, from now on we do not explicitly mention such tokens. We combine the previous symbols to denote more complex cases, where the order in which the symbols are combined is relevant, for instance: $\vee\wedge*$ means that the generated AST is of the form $a\wedge b\wedge\ldots$, i.e., it starts with a token with a superfluous ˆ to erase, followed by a token with a useful ˆ to preserve, followed by anything (maybe nothing).

The transformation A_1 will split each variable X into several variables X_r, where r is a label composed of the previous symbols $\{*, !, \vee, \wedge, \Sigma, \Sigma^+\}$. Moreover, it will guarantee that X_r generates the specific subset of the ASTs generated by X that conform to the label r. Fortunately, the number of distinct labels to consider for our goals is very small. Consider for instance an original variable $\bar{X}$, and note that any possible AST that it generates can be classified into one of the following six disjoint cases. First, if the AST has no defined root, then it corresponds to one of the two following classes:

$!$ it is empty.

Σ^+ there are some tokens without ˆ or non-empty sub-ASTs.

The third case corresponds to ASTs that have a defined root but nothing else, i.e., the class:

$\vee$ there is an isolated token with a superfluous ˆ.

The last three cases correspond to when there is a defined root and something else:

$\vee\wedge*$ the first two tokens have ˆ, and thus the first is superfluous, then there is anything.

$\wedge\Sigma*$ the first token has a ˆ, it is followed by a token without ˆ or a non-empty sub-AST, and then anything.

$\Sigma^+\wedge*$ it starts with some tokens without ˆ or non-empty sub-ASTs, then a token with ˆ, and then anything.

In summary, the variable $\bar{X}$ can be distinguished into six disjoint variables $\bar{X}_!$, $\bar{X}_{\Sigma^+}$, $\bar{X}_\vee$, $\bar{X}_{\vee\wedge*}$, $\bar{X}_{\wedge\Sigma*}$, $\bar{X}_{\Sigma^+\wedge*}$ depending on the form of the generated AST. Note that $\bar{X}_!$, $\bar{X}_{\Sigma^+}$, $\bar{X}_\vee$ generate ASTs that do not require brackets since there is no root defined (in the last case there is a root, but the ˆ is superfluous), whereas $\bar{X}_{\vee\wedge*}$, $\bar{X}_{\wedge\Sigma*}$, $\bar{X}_{\Sigma^+\wedge*}$ generate ASTs that (a priori) require brackets due to the presence of at least one useful ˆ. Moreover, by considering these six cases, we have precisely identified when a ˆ is useless: $\bar{X}_{\vee}$ generates a useless ˆ, and the first ˆ generated by $\bar{X}_{\vee\wedge*}$ is useless; any other ˆ is useful.

Intuitively, A_1 can be seen as a recursive transformation, which forwards the class of ASTs specified at the left-hand side of a rule to the corresponding right-hand sides. Initially, it starts with $S' \to S_! \mid S_{\Sigma+}$, where S' is a new variable and S is the starting variable of the input CFG (by construction, S is not an original variable of the grammar, but an auxiliary variable introduced by the transformation T, and its productions are of the form $S \to \bar{X}$), and then proceeds expanding the rules for $S_!$ and $S_{\Sigma+}$. Note that, according to our notation, the variable $S_!$ is intended to generate empty ASTs, whereas $S_{\Sigma+}$ is intended to generate non-empty ASTs (actually, it will generate non-empty sub-ASTs, since S immediately generates an original variable of the grammar). Consider the case where the input CFG has the rule $S \to \bar{X}$, and $\bar{X}$ is a variable that was not decorated with ˆ in the original grammar. In such case, A_1 must introduce for $S_!$ the rule $S_! \to \bar{X}_!$, since only $\bar{X}_!$ guarantees that the generated AST is empty. On the other hand, $S_{\Sigma+}$ must capture the remaining cases of ASTs described above, i.e., A_1 introduces the rules $S_{\Sigma+} \to \bar{X}_{\Sigma+} \mid \bar{X}_{\vee} \mid \bar{X}_{\vee\wedge*} \mid \bar{X}_{\wedge\Sigma*} \mid \bar{X}_{\Sigma^+\wedge*}$. Afterwards, the process continues by introducing rules for each of the new variables $\bar{X}_!$, $\bar{X}_{\Sigma+}$, $\bar{X}_{\vee}$, $\bar{X}_{\vee\wedge*}$, $\bar{X}_{\wedge\Sigma*}$, $\bar{X}_{\Sigma^+\wedge*}$. The main difficulty arises when we consider a variable W_r, where r is any of the possible labels for the classes of ASTs, and the input CFG has a rule of the form $W \to YZ$. In such case, part of the AST of class r must be conjectured to be generated by Y, and the remaining part by Z. This forces us to split the previous classes of ASTs and introduce the following extra classes: $*$, $\wedge$, $\wedge*$, and $\Sigma*$. Overall, all the classes of ASTs that we need to consider correspond to the following set of labels:

$$\mathcal{R} = \{*,\ !,\ \wedge,\ \vee,\ \wedge*,\ \vee\wedge*,\ \Sigma*,\ \wedge\Sigma*,\ \Sigma^+,\ \Sigma^+\wedge*\}$$

The transformation A_1 can be formalized as follows. Given the unambiguous CFG $G = \langle V, \Sigma, \delta, S\rangle$ resulting from the transformation T, we define the unambiguous CFG $A_1(G) = \langle V', \Sigma, \delta', S'\rangle$ where $V' = \{S'\} \cup \{X_r \mid X \in V \wedge r \in \mathcal{R}\}$ and δ' has the rules $S' \to S_! \mid S_{\Sigma+}$ and also the rules described in Table 2.

For A_2 it suffices to identify whether an original variable occurs leftmost inside brackets, and whether it is followed by a token with ˆ. To this end, we extend the variables to distinguish 3 cases: it is leftmost and the next token has ˆ, it is leftmost and the next token does not have ˆ, and any other case. This transformation is straightforward: it roughly consists in forwarding the information of the left-hand side of a rule to its right-hand sides, leveraging the information already introduced by A_1. Only left-hand sides of the form $\bar{X}_r$ and $\bar{X}_r$ˆ must be handled with care: if their label r implies that they must be enclosed within brackets, then the information of leftmost must be set to true, and the information of being followed by ˆ to false for $\bar{X}_r$ and to true for $\bar{X}_r$ˆ.

We can also compose A with P. In this case, however, we have to modify A_1 to erase any rule of the form $X_a \to a!$. This is done to guarantee that variables of the form X_a generate at least one non-!-token, and hence, that their ASTs are not empty. If such condition was not ensured, there might be anomalies with original variables with ˆ: for example, with input a, there is a partial execution of $G : \{S \to a!X,\ X\hat{} \to b\}$ that produces an empty AST, whereas any partial execution of the equivalent $G' : \{S \to X,\ X\hat{} \to a!b\}$ produces the AST $\boldsymbol{X}$. By removing such rules we force that the input a is not considered for checking partial executions, and instead we start by considering the input ab. With such input, partial (and complete) executions of G and G' produce the same AST $\boldsymbol{X}(b)$.

5.4 Generalization of the hashing method

In this section we generalize the method presented in [3] and outlined in Section 2.1. This generalization is based on a given morphism σ, and the idea consists in replacing the sum of hashes of the generated words by the sum of hashes of the images through σ of the generated words. The original hashing method corresponds to the particular case where σ is the identity.

Table 2: Definition of the rules of $A_1(G)$ produced from the rules δ of G.

- for each $X \to \bar{Y} \in \delta$ such that $\bar{Y}$ has rules of the form $\bar{Y} \to Z$ (i.e., no $\hat{}$ at the left-hand side), $r_1 \in \{*,\ !\}$, $r_2 \in \{*,\ \Sigma*,\ \Sigma^+\}$:

$$\begin{aligned} &X_{r_1} \to \bar{Y}_! \\ &X_{r_2} \to \bar{Y}_{\Sigma^+} \mid \bar{Y}_\vee \mid \bar{Y}_{\vee\wedge*} \mid \bar{Y}_{\wedge\Sigma*} \mid \bar{Y}_{\Sigma^+\wedge*} \end{aligned}$$

- for each $X \to \bar{Y} \in \delta$ such that $\bar{Y}$ has rules of the form $\bar{Y}\hat{} \to Z$ (i.e., $\hat{}$ at the left-hand side), $r \in \{*,\ \Sigma*,\ \Sigma^+\}$:

$$X_r \to \bar{Y}_! \mid \bar{Y}_{\Sigma^+} \mid \bar{Y}_\vee \mid \bar{Y}_{\vee\wedge*} \mid \bar{Y}_{\wedge\Sigma*} \mid \bar{Y}_{\Sigma^+\wedge*}$$

- for each $X \to YZ \in \delta$:

$$\begin{aligned} X_* &\to Y_* Z_* \\ X_! &\to Y_! Z_! \\ X_\wedge &\to Y_\wedge Z_! \mid Y_! Z_\wedge \\ X_\vee &\to Y_\vee Z_! \mid Y_! Z_\vee \\ X_{\wedge*} &\to Y_{\wedge*} Z_* \mid Y_! Z_{\wedge*} \\ X_{\vee\wedge*} &\to Y_{\vee\wedge*} Z_* \mid Y_\vee Z_{\wedge*} \mid Y_! Z_{\vee\wedge*} \\ X_{\Sigma*} &\to Y_{\Sigma*} Z_* \mid Y_! Z_{\Sigma*} \\ X_{\wedge\Sigma*} &\to Y_{\wedge\Sigma*} Z_* \mid Y_\wedge Z_{\Sigma*} \mid Y_! Z_{\wedge\Sigma*} \\ X_{\Sigma^+} &\to Y_{\Sigma^+} Z_! \mid Y_{\Sigma^+} Z_{\Sigma^+} \mid Y_! Z_{\Sigma^+} \\ X_{\Sigma^+\wedge*} &\to Y_{\Sigma^+\wedge*} Z_* \mid Y_{\Sigma^+} Z_{\wedge*} \mid \\ &\quad\ Y_{\Sigma^+} Z_{\Sigma^+\wedge*} \mid Y_! Z_{\Sigma^+\wedge*} \end{aligned}$$

- for each $X \to a \in \delta$, $r \in \{*,\ \Sigma*,\ \Sigma^+\}$:

$$X_r \to a$$

- for each $X \to a\hat{} \in \delta$, $r \in \{*,\ \wedge,\ \wedge*\}$:

$$\begin{aligned} X_r &\to a\hat{} \\ X_\vee &\to a \end{aligned}$$

- for each $X \to a! \in \delta$, $r \in \{*,\ !\}$:

$$X_r \to a!$$

- for each $X \to \varepsilon \in \delta$, $r \in \{*,\ !\}$:

$$X_r \to \varepsilon$$

- for each $\bar{X} \to Y \in \delta$, $r \in \{!,\ \vee,\ \vee\wedge*,\ \wedge\Sigma*,\ \Sigma^+,\ \Sigma^+\wedge*\}$:

$$\bar{X}_r \to Y_r$$

- for each $\bar{X}\hat{} \to Y \in \delta$, $r \in \{!,\ \vee,\ \vee\wedge*,\ \wedge\Sigma*,\ \Sigma^+,\ \Sigma^+\wedge*\}$:

$$\bar{X}_r\hat{} \to Y_r$$

- for each $X \to Y \in \delta$, $r \in \mathcal{R}$:

$$X_r \to Y_r$$

The function $H(G, \ell)$ is generalized to $H(G, \ell, \sigma) = \sum_{|u|=\ell \,\wedge\, u \in \mathcal{L}(G)} D(u) \cdot h(\sigma(u))$. This value is obtained by computing the following values for each variable X of G (where the parameter G is left implicit):

$$\begin{aligned} E(X, \ell, \sigma) &= \textstyle\sum_{|u|=\ell \,\wedge\, X \to_G^* u \in \Sigma^*} D(X, u) \cdot \mathcal{B}^{|\sigma(u)|} \\ H(X, \ell, \sigma) &= \textstyle\sum_{|u|=\ell \,\wedge\, X \to_G^* u \in \Sigma^*} D(X, u) \cdot h(\sigma(u)) \end{aligned}$$

These values can be computed as follows (recall function C of Section 2.1):

$$\begin{aligned} E(X, \ell, \sigma) &= \textstyle\sum_{(X \to YZ) \in \delta \,\wedge\, 1 \le \ell' \le \ell-1} (E(Y, \ell', \sigma) \cdot E(Z, \ell - \ell', \sigma)) + \\ &\quad \textstyle\sum_{X \to a \in \delta \,\wedge\, \ell = 1} \mathcal{B}^{|\sigma(a)|} \\ H(X, \ell, \sigma) &= \textstyle\sum_{(X \to YZ) \in \delta \,\wedge\, 1 \le \ell' \le \ell-1} (H(Y, \ell', \sigma) \cdot C(Z, \ell - \ell') + \\ &\qquad\qquad E(Y, \ell', \sigma) \cdot H(Z, \ell - \ell', \sigma)) + \\ &\quad \textstyle\sum_{X \to a \in \delta \,\wedge\, \ell = 1} h(\sigma(a)) \end{aligned}$$

This can be done in $\mathcal{O}(|G| \cdot L^2 + \|\sigma\|)$ for all $\ell \le L$, where $\|\sigma\| = \sum_{a \in \Sigma} |\sigma(a)|$. Note that $H(G, \ell, \sigma)$ coincides with $H(S, \ell, \sigma)$, where S is the start symbol of G. Thus, we can compute and compare the pairs $H(G_1, \ell, \sigma), H(G_2, \ell, \sigma)$ in $\mathcal{O}((|G_1| + |G_2|) \cdot L^2 + \|\sigma\|)$. If $H(G_1, \ell, \sigma) \ne H(G_2, \ell, \sigma)$ for some ℓ, we conclude that there exists a word w of length ℓ such that $D(G_1, w) \ne D(G_2, w)$. We generalize the function $H(G, p, \ell)$ used while constructing the counterexample w as $H(G,p,\ell,\sigma) = \sum_{|u|=\ell \,\wedge\, u[1\ldots|p|]=p \,\wedge\, u\in\mathcal{L}(G)} D(u)\cdot h(\sigma(u))$. In order to compute it, we need to generalize the above computation to obtain the following values, where $1 \leq j \leq \ell$ and $1 \leq \ell' \leq \ell - j + 1$:

$$\begin{aligned} E(X,p,j,\ell',\sigma) &= \textstyle\sum_{|u|=\ell' \,\wedge\, X\to^*_G u\in\Sigma^* \,\wedge\, (j>|p| \,\vee\, \mathsf{match}(u,p[j\ldots|p|]))} D(X,u)\cdot \mathcal{B}^{|\sigma(u)|} \\ H(X,p,j,\ell',\sigma) &= \textstyle\sum_{|u|=\ell' \,\wedge\, X\to^*_G u\in\Sigma^* \,\wedge\, (j>|p| \,\vee\, \mathsf{match}(u,p[j\ldots|p|]))} D(X,u)\cdot h(\sigma(u)) \end{aligned}$$

Note that for $j > |p|$ the values $E(X,p,j,\ell',\sigma)$ and $H(X,p,j,\ell',\sigma)$ coincide with the previously computed values $E(X,\ell',\sigma)$ and $H(X,\ell',\sigma)$, respectively. For the cases $j \leq |p|$ we can proceed as follows:

$$\begin{aligned} E(X,p,j,\ell',\sigma) = &\textstyle\sum_{(X\to YZ)\in\delta \,\wedge\, 1\leq\ell''\leq\ell'-1}(E(Y,p,j,\ell'',\sigma)\cdot E(Z,p,j+\ell'',\ell'-\ell'',\sigma))+ \\ &\textstyle\sum_{X\to a\in\delta \,\wedge\, \ell'=1 \,\wedge\, p[j]=a} \mathcal{B}^{|\sigma(a)|} \\ H(X,p,j,\ell',\sigma) = &\textstyle\sum_{(X\to YZ)\in\delta \,\wedge\, 1\leq\ell''\leq\ell'-1}(H(Y,p,j,\ell'',\sigma)\cdot C(Z,p,j+\ell'',\ell'-\ell'')+ \\ &\qquad\qquad\qquad E(Y,p,j,\ell'',\sigma)\cdot H(Z,p,j+\ell'',\ell'-\ell'',\sigma))+ \\ &\textstyle\sum_{X\to a\in\delta \,\wedge\, \ell'=1 \,\wedge\, p[j]=a} h(\sigma(a)) \end{aligned}$$

For a fixed p, these values can be computed for all such j, ℓ' in $\mathcal{O}(|G|\cdot\ell^3 + \|\sigma\|)$. Since this is done for each $G \in \{G_1, G_2\}$ and repeated to obtain each symbol of w, the time to construct w is in $\mathcal{O}((|G_1|+|G_2|)\cdot|\Sigma|\cdot\ell^4+\|\sigma\|)$. This can be reduced to $\mathcal{O}((|G_1|+|G_2|)\cdot\log(|\Sigma|)\cdot\ell^4+\|\sigma\|)$ by doing binary search on Σ when trying to identify each symbol of the counterexample.

Our generalization of the hashing method with a morphism σ avoids some cases where the properties of the hashing for words are degraded when one considers its generalization to sets of words:

Example 5.2 *Consider the two unambiguous CFGs G_1, G_2 from Example 2.6. Now, by defining σ as $\sigma(a) = aa$ and as the identity for the rest of symbols, we obtain $H(G_1, 2, \sigma) \neq H(G_2, 2, \sigma)$ (unless there is a collision), since $H(G_1, 2, \sigma) = h(\sigma(ab)) + h(\sigma(cd)) = h(aab) + h(cd) = (a + a\mathcal{B} + b\mathcal{B}^2) + (c + d\mathcal{B})$ whereas $H(G_2, 2, \sigma) = h(\sigma(ad)) + h(\sigma(cb)) = h(aad) + h(cb) = (a + a\mathcal{B} + d\mathcal{B}^2) + (c + b\mathcal{B})$. Thus, the false positive is avoided.*

5.4.1 Application to check equivalence of AST construction

As we mention in Section 5.2, the generalization of the hashing method by allowing the use of a morphism σ is essential to tackle the !-tokens: instead of removing them from the CFG, we just define $\sigma(a!) = \varepsilon$ for each symbol a. This way, such symbols do not contribute to the computation of the hash, and hence, the corresponding anomalies do not take place. The use of the morphism has other additional advantages. Note that in the transformation A we do not transform each $a\hat{}$ into $a\wedge$ as is done with A'. Instead, we can simply define $\sigma(a\hat{}) = a\wedge$. Note that with A' the size of the counterexamples increases as new symbols $\wedge$ are inserted, whereas by using σ the size is preserved. Since the size of the counterexample has a strong influence on the performance of the method, the approach using σ is preferable.

A similar argument can be made with the insertion of the brackets for the original variables. The transformation A does not insert such brackets to avoid increasing the size of the counterexamples, and instead, we simulate their insertion with further changes of the hashing method. On the one hand, the process removing ε-rules and unit rules must be adapted, so that the particular unit rules with an original variable as left-hand side are preserved. Note that, according to the normalization process of PCFGs and the definitions of the transformations T, P, A, such variables appear always in unit rules. The computations can be extended to these rules in a straightforward way:

$$\begin{aligned} C(X,\ell) &= \textstyle\sum_{(X\to Y)\in\delta} C(Y,\ell) \\ E(X,\ell,\sigma) &= \textstyle\sum_{(X\to Y)\in\delta} E(Y,\ell,\sigma) \\ H(X,\ell,\sigma) &= \textstyle\sum_{(X\to Y)\in\delta} H(Y,\ell,\sigma) \end{aligned}$$

On the other hand, for those variables X that should insert brackets, we have to modify the computation of the values $E(X, \ell, \sigma)$, $H(X, \ell, \sigma)$ as follows:

$$\begin{aligned} E(X,\ell,\sigma) &= \textstyle\sum_{(X\to Y)\in\delta} \mathcal{B}^{|\sigma([)|} \cdot E(Y,\ell,\sigma) \cdot \mathcal{B}^{|\sigma(\alpha_1)\sigma(\alpha_2)|} \\ H(X,\ell,\sigma) &= \textstyle\sum_{(X\to Y)\in\delta} h(\sigma([)) \cdot C(Y,\ell) + \\ &\qquad \mathcal{B}^{|\sigma([)|} \cdot (H(Y,\ell,\sigma) + E(Y,\ell,\sigma) \cdot h(\sigma(\alpha_1)\sigma(\alpha_2))) \end{aligned}$$

where the expression $\sigma([)$ represents the application of the morphism σ to the symbol [, the word α_1 is $\bar{\boldsymbol{X}}\wedge$ when the original variable $\bar{X}$ that gives rise to X is followed by ˆ, and ε otherwise, and the word α_2 is] when X is a variable of the form $\bar{X}_{ab}$, and ε when it is a variable of the form $\bar{X}_a$. The remaining functions $C(X, p, j, \ell')$, $E(X, p, j, \ell', \sigma)$, $H(X, p, j, \ell', \sigma)$ are adapted analogously.

The following theorem states that the method does not produce false negatives. Of course, it may produce false positives, either because it is used with ℓ's up to a too small L or because there is a hash collision. However, collisions are very unlikely if the execution is iterated several times with different σ's.

Theorem 5.3 *Let σ be a morphism, and let ℓ be a natural number. Let G_1, G_2 be two normalized PCFGs such that $\mathcal{L}_{\mathcal{P}}(G_1) = \mathcal{L}_{\mathcal{P}}(G_2)$ and each word parsed by them generates the same AST. Then, $H(A(T(G_1)), \ell, \sigma) = H(A(T(G_2)), \ell, \sigma)$. Moreover, if G_1 and G_2 do eager detection of syntax errors, then $H(A(P(G_1)), \ell, \sigma) = H(A(P(G_2)), \ell, \sigma)$.*

6 Performance

Given a normalized PCFG $G = \langle V, \Sigma, \delta, S\rangle$, $T(G)$ and $P(G)$ have $\mathcal{O}(|\delta| \cdot |\Sigma|^2)$ variables and $\mathcal{O}(|\delta| \cdot |\Sigma|^3)$ rules, and the same holds for $(A \circ T)(G)$ and $(A \circ P)(G)$ but with a bigger constant factor. To appreciate this growth, we consider a PCFG $G_{n,m}$ for expressions similar to those found in most programming languages, parameterized by the number n of literals (tokens a_i) and the number m of levels of precedence of operators (tokens $\triangle_i$, $\square_i$), with two operators at each precedence level and allowing to parenthesize subexpressions (tokens o, c represent the opening and closing parentheses, respectively):

$$\begin{aligned} X_m &\to X_{m-1}\ ((\triangle_m\hat{}\mid \square_m\hat{})\ X_{m-1})* \\ X_{m-1} &\to X_{m-2}\ ((\triangle_{m-1}\hat{}\mid \square_{m-1}\hat{})\ X_{m-2})* \\ X_{m-2} &\to X_{m-3}\ ((\triangle_{m-2}\hat{}\mid \square_{m-2}\hat{})\ X_{m-3})* \\ &\vdots \\ X_1 &\to X_0\ ((\triangle_1\hat{}\mid \square_1\hat{})\ X_0)* \\ X_0 &\to a_1 \mid a_2 \mid \ldots \mid a_n \mid o!\ X_m\ c! \end{aligned}$$

Figure 1 shows the number of variables of the CFGs resulting from applying the transformations to $G_{n,m}$. The increase in the number of variables is notorious since the Follow sets are big for most of the X_i's, as they include all the operators with higher subindex (i.e., lower precedence). Thus, these experimental results can be considered pessimistic with respect to a practical setting. Also note that, since the variables generated by T are a subset of those generated by P, the latter generates more variables, but the overhead is not significant. Finally, notice that the process of cleaning useless variables and unit rules significantly reduces the size of the CFGs, and that the constant factor increase due to A is approximately 10.

We also evaluate the performance of constructing a counterexample with the hashing method. Recall that this corresponds to the most expensive step: the transformations T and P produce an increase in the number of rules proportional to $|\Sigma|^3$, and thus, the cost of generating a counterexample of length ℓ for two normalized PCFGs G_1, G_2 is in $\mathcal{O}((|G_1| + |G_2|) \cdot |\Sigma|^3 \cdot \log(|\Sigma|) \cdot \ell^4 + \|\sigma\|)$. We define the PCFGs $G_{n,m,r}$ and $G'_{n,m,r}$ as variations of $G_{n,m}$, where $G_{n,m,r}$ is obtained by erasing the operators $\square_i$ and by replacing the rule of X_m by $X_m \to X_{m-1}\ (\triangle_m\hat{}\ X_{m-1})* \mid \# \cdots \#$, with exactly r occurrences of

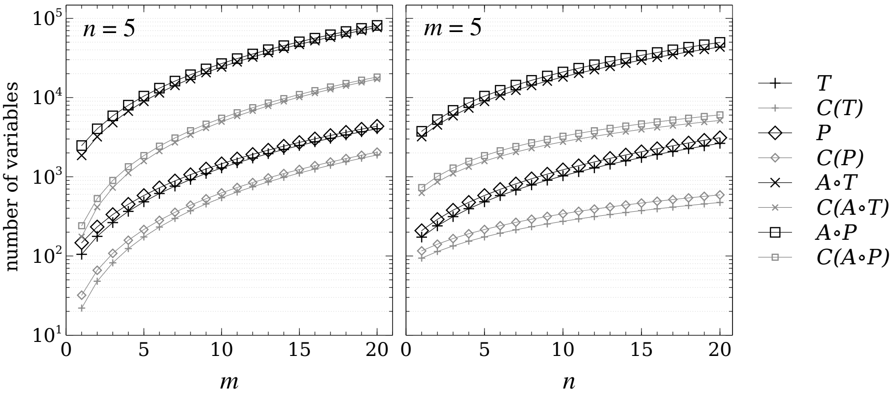

Figure 1: Amount of variables generated by each transformation on $G_{n,m}$, as a function of the number m of levels of operators and for a fixed number $n = 5$ of literals (left), and as a function of n and for a fixed $m = 5$ (right). In the legend, C denotes the process of cleaning useless variables and unit rules.

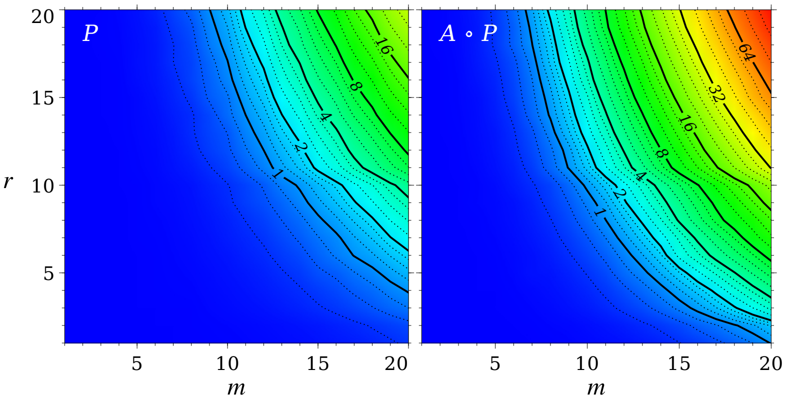

Figure 2: Execution times in seconds of the hashing method when given as input the pair $P(G_{n,m,r})$, $P(G_{n,m,r+1})$ (left) and the pair $(A \circ P)(G_{n,m,r})$, $(A \circ P)(G'_{n,m,r})$ (right), with n fixed to 1.

a new symbol #, and $G'_{n,m,r}$ is obtained analogously except that the last occurrence of # is followed by !. We compare $P(G_{n,m,r})$ with $P(G_{n,m,r+1})$ and $(A \circ P)(G_{n,m,r})$ with $(A \circ P)(G'_{n,m,r})$, in both cases yielding a counterexample of size r. Figure 2 shows the execution times[1] obtained in both tests when varying the parameters m and r, with n fixed to 1. As expected, the execution time depends non-linearly on r. For the second test, the construction takes roughly 1 second with $m = r = 10$, which is acceptable for the judge, and over a minute with $m = 20$ and $r = 15$.

Finally, we adapt a PASCAL parser[2] implemented with ANTLR3 to the judge syntax (see Appendix A) in order to test the performance with a real programming language. The resulting PCFG G_{PAS} has 69 tokens and 72 variables, most of which have non-trivial right-hand sides. Computing the hash up to $L = 20$ takes 0.13 seconds for $P(G_{\text{PAS}})$, and 0.41 seconds for $(A \circ P)(G_{\text{PAS}})$. When comparing G_{PAS} with a variant G'_{PAS} that constructs wrong ASTs for words of size 14, a counterexample is obtained in 23.58 seconds when using the transformation T, whereas with P it only takes 2.48 seconds to find a prefix of size 8.

[1] All measurements have been taken on a 64-bit Intel® Pentium® T4400 at 2.2 GHz with 3GB of RAM.

[2] Available at: http://www.antlr3.org/grammar/list.html

7 Conclusions

We have developed efficient methods to check equivalence of the parsed language as well as the AST construction for grammars interpreted as top-down predictive parsers. The syntax of the grammars and their interpretation as parsers are based on the PCCTS/ANTLR3 tools. These tools are well established in the field, and our methods run fast enough with grammars for some real programming languages. Thus, they can be used in practice to verify whether two syntax definitions, with the corresponding AST construction, are equivalent.

The equivalence problem for parsed languages of grammars interpreted in this way can be reduced to the equivalence of languages recognized by deterministic push-down automata, which is known to be decidable [7, 8]. It would be interesting to study the decidability of equivalence of AST construction as well.

We have implemented the two variants based on the transformations T and P, and use the latter in the judge to evaluate the list of exercises on parsing. These exercises are designed to help students understand the basics of top-down predictive parsing and improve their ability to define adequate grammars for parser generators. Thus far, students have been able to reach an acceptance verdict 1200 times, only giving up on 56 occasions. The mean number of submissions needed to reach an acceptance has been 3.6, whereas it goes up to 7 for those that have given up. Overall, the judge has provided over two thousand counterexamples to submissions that parsed the wrong language and almost a thousand to submissions that constructed the ASTs wrongly. The mean size of these counterexamples has been 2.6 tokens for the former and 2.9 tokens for the latter. These stats are mostly analogous when considering the exercises of the exams performed on the judge, the main differences being that students have given up more often (on 17.96% of the occasions instead of 4.46%) but trying harder (7.8 submissions until giving up instead of 7). Also, the size of the counterexamples for wrong AST construction has increased significantly in exams, up to 4.2 tokens, with the longest one having 14 tokens and obtained in 1.1 seconds. All these differences are a simple consequence of exam exercises asking for more complex languages.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (Second Edition).* Addison-Wesley, 2006.

[2] Carles Creus, Pau Fernández, and Guillem Godoy. Automatic evaluation of reductions between NP-complete problems. In *SAT*, pages 415–421, 2014.

[3] Carles Creus and Guillem Godoy. Automatic evaluation of context-free grammars (system description). In *RTA-TLCA*, pages 139–148, 2014.

[4] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide.* Springer, 2008.

[5] Terence Parr. *Language Translation using PCCTS and C++: A Reference Guide.* Automata Publishing Company, 1993.

[6] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In *PLDI*, pages 425–436, 2011.

[7] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP*, pages 671–681, 1997.

[8] Colin Stirling. Deciding DPDA equivalence is primitive recursive. In *ICALP*, pages 821–832, 2002.

A PCFG for PASCAL

We present the PCFG G_{PAS} for the PASCAL language that is used in the experiments of Section 6. It is described with the syntax expected by the online judge. The only kind of metacharacter that has not been introduced in Section 3 is the postfix operator **?** to denote optional blocks, i.e., an expression such as (*e*)**?** is equivalent to (*e*|). To simplify the presentation, we do not include the lexer rules for the tokens **IDENTIFIER**, **INT_LIT**, **REAL_LIT**, **STRING_LIT**.

```
// Program blocks
program^: heading block '.'! ;
heading: 'PROGRAM'^ id ('('! ids ')'!)? ';'!
        | 'UNIT'^ id ';'! ;
id: IDENTIFIER ;
ids^: id (','! id)* ;
block: (labelDecl | constBlock | typeDef | varBlock | procfunDecl |
    usesUnits)* compoundStmt ;
usesUnits: 'USES'^ ids ';'! ;
labelDecl: 'LABEL'^ label (','! label)* ';'! ;
label: unsignedInteger ;
constBlock: 'CONST'^ constDef ';'! (constDef ';'!)* ;
constDef: id '='^ const ;
const: ('+'^ | '-'^)? (unsignedNumber | id)
      | STRING_LIT
      | constChr ;
constChr: 'CHR'^ '('! unsignedInteger ')'! ;
unsignedNumber: unsignedInteger | REAL_LIT ;
unsignedInteger: INT_LIT ;

// Types
typeDef: 'TYPE'^ typeDefAux ';'! (typeDefAux ';'!)* ;
typeDefAux: id '='^ (type | funcType | procType) ;
funcType: 'FUNCTION'^ formalParams ':'! resType ;
procType: 'PROCEDURE'^ formalParams ;
type: simpleType | structType | pointerType ;
simpleType: scalarType
           | id ('..'^ const)?
           | const '..'^ const
           | 'CHAR'
           | 'BOOLEAN'
           | 'INTEGER'
           | 'REAL'
           | 'STRING'^ ('['! (id|unsignedNumber) ']'!)? ;
scalarType^: '('! ids ')'! ;
typeId: id | 'CHAR' | 'BOOLEAN' | 'INTEGER' | 'REAL' | 'STRING' ;
structType: 'PACKED'^ unpackedStructType
           | unpackedStructType ;
unpackedStructType: arrayType | recordType | setType | fileType ;
arrayType: 'ARRAY'^ '['! indexTypes ']'! 'OF'! componentType ;
indexTypes^: indexType (','! indexType)* ;
indexType: simpleType ;
componentType: type ;
recordType: 'RECORD'^ fields 'END'! ;
fields: recordSection ';'! (recordSection ';'!)* ;
recordSection^: ids ':'! type ;
setType: 'SET'^ 'OF'! baseType ;
baseType: simpleType ;
fileType: 'FILE'^ ('OF'! type)? ;
```

```
 49 pointerType: '^'^ typeId ;
 50 
 51 // Declarations
 52 varBlock: 'VAR'^ varDecl ';'! (varDecl ';'!)* ;
 53 varDecl^: ids ':'! type ;
 54 procfunDecl: (procDecl | funcDecl) ';'! ;
 55 procDecl: 'PROCEDURE'^ id formalParams ';'! block ;
 56 funcDecl: 'FUNCTION'^ id formalParams ':'! resType ';'! block ;
 57 formalParams^: ('('! formalParam (';'! formalParam)* ')'!)? ;
 58 formalParam: ('VAR'^|'FUNCTION'^|'PROCEDURE'^)? ids ':'! typeId ;
 59 resType: typeId ;
 60 
 61 // Expressions
 62 variable: ('@'^ id | id) ('['^ expr (','! expr)* ']'! | '.'^ id | '^'^)*
        ;
 63 expr: add (('='^|'<>'^|'<'^|'<='^|'>='^|'>'^|'IN'^) add)* ;
 64 add: term (('+'^|'-'^|'OR'^) term)* ;
 65 term: signed (('*'^|'/'^|'DIV'^|'MOD'^|'AND'^) signed)* ;
 66 signed: ('+'^|'-'^)? factor ;
 67 factor: id (('['^ expr (','! expr)* ']'! | '.'^ id | '^'^)*
 68                |'('^ expr (','! expr)* ')'!)
 69           | variable // the id expression starting by @
 70           | '('! expr ')'!
 71           | unsignedConst
 72           | set
 73           | 'NOT'^ factor ;
 74 set^: '['! elems ']'! ;
 75 elems: (elem (','! elem)*)? ;
 76 elem: expr ('..'^ expr)? ;
 77 unsignedConst: unsignedNumber | constChr | STRING_LIT | 'NIL' ;
 78 
 79 // Statements
 80 stmt: (label ':'^)? unlabelledStmt ;
 81 unlabelledStmt: simpleStmt | structStmt ;
 82 simpleStmt: assignOrCall | gotoStmt ;
 83 assignOrCall: IDENTIFIER^ ( ( '['^ expr (','! expr)* ']'!
 84                               | '.'^ id
 85                               | '^'^)* ':='^ expr
 86                             | ('('! expr (','! expr)* ')'!)?)
 87                 | variable ':='^ expr ; // assigns starting by @
 88 gotoStmt: 'GOTO'^ label ;
 89 structStmt: compoundStmt | condStmt | repStmt | withStmt ;
 90 compoundStmt^: 'BEGIN'! stmts 'END'! ;
 91 stmts^: stmt (';'! stmt)* ;
 92 condStmt: if | cases ;
 93 if: 'IF'^ expr 'THEN'! stmt ('ELSE'! stmt)? ;
 94 cases: 'CASE'^ expr 'OF'! case ';'! (case ';'!)* ('ELSE'! stmts)? 'END'!
        ;
 95 case: consts ':'^ stmt ;
 96 consts^: const (','! const)* ;
 97 repStmt: while | repeat | for ;
 98 while: 'WHILE'^ expr 'DO'! stmt ;
 99 repeat: 'REPEAT'^ stmts 'UNTIL'! expr ;
100 for: 'FOR'^ id ':='! expr ('TO'^|'DOWNTO'^) expr 'DO'! stmt ;
101 withStmt: 'WITH'^ recordVariables 'DO'! stmt ;
102 recordVariables: variable (','! variable)* ;
```

Note that the previous PCFG has 4 rules with conflicts. First, the second and third alternatives of the variable **simpleType** (lines 27 and 28) have the token **IDENTIFIER** in their

First sets. The same happens for the first and second alternatives of the variable **factor** (lines 67 and 69) and also the first and second alternatives of the variable **assignOrCall** (lines 83 and 87). Finally, the optional block occurring in the rule of the variable **if** (line 93) has a conflict with lookahead **'ELSE'** since this token is in its First and Follow sets.