LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

MODULAR SYNTAX DEMANDS VERIFICATION

Sylvain Schmitz

Projet LANGAGES

Rapport de recherche
ISRN I3S/RR–2006-32–FR

Octobre 2006

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télecopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ :

Les formalismes grammaticaux modulaires sont un pas essentiel vers de meilleures pratiques en ingénierie des grammaires. Cependant, en nous éloignant des modèles déterministes traditionnels, certaines vérifications statiques intrinsèques sont perdues. L'article montre pourquoi une vérification des grammaires est nécessaire pour une utilisation robuste des grammaires algébriques ou des grammaires d'expressions recognitives comme formalismes syntaxiques modulaires. Des procédures de vérification simples sont présentées pour chacun de ces formalismes.

MOTS CLÉS :

Ingénierie des gammaires, vérification, modules, grammaire algébrique, grammaire d'expressions recognitives, désambiguation

ABSTRACT:

Modular grammatical formalisms provide an essential step towards improved grammar engineering practices. However, as we depart from traditional deterministic models, some intrinsic static checks are lost. The paper shows why grammar verification is necessary for reliable uses of context-free grammars (CFGs) and parsing expression grammars (PEGs) as modular syntax definitions. Simple conservative verification procedures are presented for each formalism.

KEY WORDS :

Grammar engineering, verification, module system, context-free grammar, parsing expression grammar, disambiguation

Modular Syntax Demands Verification

Sylvain Schmitz

Laboratoire I3S, Université de Nice - Sophia Antipolis, France
schmitz@i3s.unice.fr

Abstract

Modular grammatical formalisms provide an essential step towards improved grammar engineering practices. However, as we depart from traditional deterministic models, some intrinsic static checks are lost. The paper shows why grammar verification is necessary for reliable uses of context-free grammars (CFGs) and parsing expression grammars (PEGs) as modular syntax definitions. Simple conservative verification procedures are presented for each formalism.

Key words: Grammar engineering, verification, module system, context-free grammar, parsing expression grammar, disambiguation

ACM categories: D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules ; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems

1 Introduction

New techniques, like Generalized LR [32] and packrat [9] parsing, have opened new opportunities for grammar engineering, notably by accepting *modular* syntactic descriptions. While modularity allows a rational development of grammars and grammar-related software, some reliability issues with the underlying formalisms have been overlooked. The services of modularity are indeed so great that many practitioners prefer to ignore the risk of something going wrong rather than to let go of their improved condition.

In this paper, we argue that we can keep the best of two worlds, by enforcing the use of grammar verification tools. Our position is that the introduction of engineering practice in grammatical development is logically followed by the design of adapted tools, including formal verification ones. We make the need for such tools clear by exhibiting two representative decision issues faced by two modern grammatical formalisms, and we shed light on some possible approaches to the problems by providing some simple resolutions.

In more details, after a few words on the newly-born field of grammar engineering, we motivate the use of modular syntax formalisms by means of a concrete parsing application (Section 2.1). We then implement our example in two modular syntax formalisms, namely context-free grammars with disambiguation filters and parsing expression grammars (Section 2.2), and exhibit two decidability issues that burden them (Section 2.3). The first undecidable problem is well known as the ambiguity problem in context-free grammars [4, 6]. The second problem is a contribution of the article: it is the semi disjointness of two parsing expressions; we show its undecidability in Section 4.1.

Having identified two verification problems, we propose checking procedures based on grammar graph quotienting (Sections 3 and 4). For the sake of simplicity, we describe rather conservative approaches. We hope that they will be taken as incentives for future study over the verification problems that are put forward in this paper.

Omitted implementation and proof details are given in Appendices B and C. Formal definitions and notational conventions for context-free grammars and parsing expression grammars are given in Appendices A.1 and A.2 respectively. The reader unfamiliar with these two syntax formalisms might need to read them in order to follow the arguments of forthcoming Section 2.

2 Modular Grammar Engineering

Grammars hold an uncomfortable position in software engineering. They are at the same time

1. specifications of languages and structures manipulated by the software, and
2. portions of the source code in their own right.

In their first role, they represent some sort of an ideal in engineering practice, where specifications are automatically derived into correct code by tools like YACC [17]. In their second role, they did not gather enough attention when one considers the difficulty of grammar development. A new field of software methodology dedicated to *grammar engineering* is now emerging, supported by the seminal work of Klint *et al.* [20] and a few success stories in its application, notably a COBOL grammar recovery [22] and a C# parser development [23].[1]

One of the challenges for grammar engineering is the *modularity* of grammatical definitions, which would foster both easy prototyping and a better reuse of grammar fragments. The issue can be illustrated in the case of parser generators. Classical parser generators are usually restricted to subclasses (LL(1), LALR(1), up to LR(k)) of the context-free grammars, classes that do not enjoy good closure properties: new *conflicts* can be introduced when modifying the grammar. Several attempts to address this issue are given in the literature, most often by using the whole class of context-free grammars. Employing general [8, 32] or backtracking parsing methods is considered there as a trade of performance for flexibility.

2.1 Modularity in Practice

Let us give a practical account of how modularity can assist in writing a parser for a programming language. We consider a very small and very simple portion

[1]The field of natural language processing has encouraged engineering practice in grammar development for much longer; the sheer complexity of natural languages does not leave much room for hackery.

of the syntax of Standard ML [24, Appendix B], more precisely of the pattern syntax.

The context-free rules we are interested in for our exposition are

$$\begin{array}{rcl} \langle \mathit{pat} \rangle & \rightarrow & \langle \mathit{atpat} \rangle \mid \langle \mathit{pat} \rangle : \langle \mathit{ty} \rangle \\ \langle \mathit{atpat} \rangle & \rightarrow & \mathit{vid} \mid \_ \mid (\langle \mathit{pats} \rangle) \mid (\ ) \\ \langle \mathit{pats} \rangle & \rightarrow & \langle \mathit{pat} \rangle \mid \langle \mathit{pats} \rangle , \langle \mathit{pat} \rangle \end{array} \qquad (\mathcal{G}_1)$$

For instance, using these rules, we can generate the pattern "(x,y,_): int∗int∗int" where "(x,y,_)" is an atomic pattern $\langle \mathit{atpat} \rangle$ further decomposable as a tuple with the value identifiers *vid* "x" and "y" and the wildcard "_" as individual patterns, while "int∗int∗int" is a type $\langle \mathit{ty} \rangle$.

An additional possibility offered by Standard ML is the syntax of layered patterns, following the context-free rules

$$\langle \mathit{pat} \rangle \quad \rightarrow \quad \mathit{vid} : \langle \mathit{ty} \rangle\ \mathit{as}\ \langle \mathit{pat} \rangle \mid \mathit{vid}\ \mathit{as}\ \langle \mathit{pat} \rangle \qquad (\mathcal{G}_2)$$

The syntax of layered patterns is notoriously difficult to parse [18]. Upon reading the partial input

```
val f = fn triple
```

a deterministic parser cannot distinguish whether the *vid* "triple" is an atomic pattern $\langle \mathit{atpat} \rangle$, potentially with an associated type "int∗int∗int", as in

```
val f = fn triple:int*int*int => triple
```

or is the start of a more complex layered pattern as in

```
val f = fn triple:int*int*int as (_, _, z) => z + 1
```

Types $\langle \mathit{ty} \rangle$ in Standard ML can be quite complex expressions, of arbitrary length. The inconclusive lookahead string "int∗int∗int" can be longer than the fixed k of a LR(k) parser. Adding the rules of $\mathcal{G}_2$ to $\mathcal{G}_1$ yields a non-LR(k) grammar: LR(k) grammars are not closed under union. The construction of a Standard ML compiler that used to work flawlessly for the pattern syntax of $\mathcal{G}_1$ would break upon addition of $\mathcal{G}_2$. A modular syntax formalism should allow to generate a parser for the union of $\mathcal{G}_1$ and $\mathcal{G}_2$.

2.2 Implementation

We have implemented $\mathcal{G}_1$ and $\mathcal{G}_1 \cup \mathcal{G}_2$[2] in two different modular formalisms, namely the Modular Syntax Definition Formalism (SDF2) [35] and Parsing Expression Grammars (PEG) [10]. The two formalisms present further advantages for the modular development of grammars: both integrate the lexical aspects of the syntax seamlessly, and use so-called *scannerless* parsing techniques [28]. They are good representatives of the state of the art in grammatical formalisms for computer languages.

It is worth noting that we are merely scratching the possibilities of these formalisms and of their associated tools in our implementations, for we are only interested in the cases that we identified as deserving a verification. The reader is warmly encouraged to consider these tools for any future grammar developments.

[2]Actually, we implemented *augmented* versions of $\mathcal{G}_1$ and $\mathcal{G}_1 \cup \mathcal{G}_2$, with a rule $\langle \mathit{start} \rangle \rightarrow \langle \mathit{pat} \rangle$; from the start symbol $\langle \mathit{start} \rangle$.

2.2.1 SDF2: Modular Syntax Definition Formalism

The Modular Syntax Formalism SDF2 [35] is a concise syntax description language used in the ASF+SDF meta-environment. The syntax descriptions of SDF2 are compiled to generate scannerless Generalized LR parsers [36].

The grammars in SDF2 are CFGs with additional regular constructs as in Extended BNF notation, with modular capabilities, and with *disambiguation filters* [19] as functions from sets of parse trees to smaller sets of parse trees. The set of syntax filters already implemented in SDF2 is described by van den Brand *et al.* [33], and more possibilities are opened by semantic filters [34].

Implementation in SDF2 We have implemented $\mathcal{G}_1$ and $\mathcal{G}_1 \cup \mathcal{G}_2$ using the SDF2 bundle 2.3.3;[3] the definition corresponding to $\mathcal{G}_1$ is given in Appendix B.1. The syntax of Standard ML patterns $\langle pat\rangle$, types $\langle ty\rangle$ and identifiers *vid* is described in the modules `MLPatterns`, `MLTypes` and `MLIdentifiers` respectively. We show here how we added the rules of $\mathcal{G}_2$ in the `Main` module.

```
module Main
imports
  MLPatterns MLTypes MLIdentifiers Layout
exports
  context-free start-symbols START
  sorts START
  context-free syntax
    PAT ";"                      -> START
    VID (":" TY)? "as" PAT       -> PAT
```

This last line is all what we needed in order to add the rules of $\mathcal{G}_2$.

2.2.2 PEG: Parsing Expression Grammar

Parsing expression grammars were "rediscovered" by Ford [10] from an earlier formalism called TS [2]. In contrast with CFGs, PEGs are a recognitive formalism: they act as recognizers for the language they describe, and thus the rules are denoted by $A \leftarrow \alpha$. Another difference with CFGs is that PEGs use a *ordered* choice operator / instead of |. The alternative rules of a nonterminal are tested in order, and the first successful match is employed. While the rules $A{\rightarrow}ab \mid a$ and $A{\rightarrow}a \mid ab$ are equivalent in a CFG and generate the language $\{a, ab\}$, the PEG rules $A \leftarrow ab/a$ and $A \leftarrow a/ab$ are not: they recognize $\{ab, a\}$ and $\{a\}$ respectively.

Parsing expression grammars are thus an extreme variant of the ordered context-free grammars used in first-match backtracking parsing methods. Indeed, the parsers for PEGs, called *packrat* parsers [9], belong to the family of recursive descent parsers, and use memoization techniques to obtain a linear time complexity bound in the length of the input.

The excellent modularity of PEGs stems from the fact that the class of languages defined by PEGs is closed by union, intersection and complement.

[3]The SDF2 homepage can be reached at `http://www.syntax-definition.org/`.

Implementation in *Rats!* We have implemented $\mathcal{G}_1$ and $\mathcal{G}_1 \cup \mathcal{G}_2$ in *Rats!* [12] version 1.9.3,[4] a PEG parser generator for Java that takes advantage of the closure properties of PEGs. As with SDF2, we have used its modular possibilities to separate the lexing and type syntax concerns in two different modules `MLLexing` and `MLTypes`. The code for $\mathcal{G}_1$ is in a module `SimpleMLPatterns` and is given in Appendix B.2. We show here the module `MLPatterns` that adds the rules of $\mathcal{G}_2$ from which we hope to produce a parser for $\mathcal{G}_1 \cup \mathcal{G}_2$:

```
module MLPatterns;
import MLLexing;
modify SimpleMLPatterns;

public generic Start = Pattern void:';' ;

generic Pattern +=
    <Atomic> ...
  / <Layered> ValueID TypeOp void:"as":Keyword Pattern TypeOp
  ;
```

The `+=` operator of *Rats!* allows to add a new alternative to an existing nonterminal, and we use it to add the rules of $\mathcal{G}_2$.

2.3 Tests

Testing is an unavoidable part of any software development. The emerging grammar engineering field has already produced testing tools for grammars and grammar-based software [26, 21, 13]. Our case does not deserve very sophisticated tests, and we are simply going to attempt to parse a correct Standard ML pattern with our implementations. The implementations of $\mathcal{G}_1$ in SDF2 and *Rats!* work flawlessly, and our next step is to test our implementations of $\mathcal{G}_1 \cup \mathcal{G}_2$ on the valid input sentence "t **as** (x,y,_): triple ;". Our implementations and our example are of course carefully chosen in order to unveil issues with the formalisms. Let us see what problems can appear at run-time with our parsers.

2.3.1 Testing with SDF2

```
sglr:error: Ambiguity in input, line 1, col 0:
  PAT  ":"  TY -> PAT;VID  (":" TY )?  "as"  PAT -> PAT
```

This error message issued by the scannerless GLR parser of the SDF2 tools simply states that the sentence "t **as** (x,y,_): triple ;" is ambiguous. This is a classical *dangling* ambiguity, where the type specification can be attached at a high or a low level. Figure 1 shows the two different possibilities for our test sentence. The general rule given in the Standard ML definition is that low attachments should be preferred, and thus that we should obtain the parse tree of Figure 1b. This rule can be enforced by using one of the predefined syntax filters of SDF2:

```
    VID (":" TY)? "as" PAT      -> PAT {prefer}
```

[4]The *Rats!* parser generator can be found at `http://cs.nyu.edu/rgrimm/xtc/rats.html`.

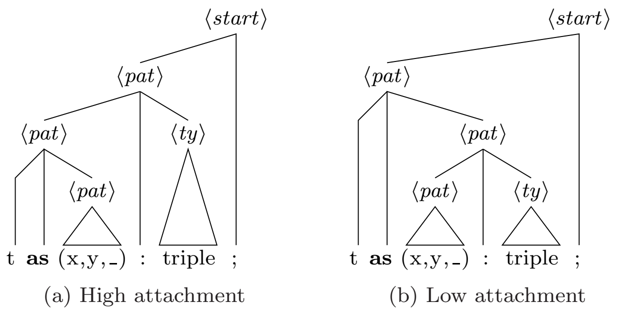

Figure 1: An ambiguity in $\mathcal{G}_1 \cup \mathcal{G}_2$.

2.3.2 Testing with *Rats!*

```
xtc.parser.ParseException:
input:1:2: error: symbol characters expected
t as (x,y,_): triple;
 ^
```

The error message of *Rats!* stems from the way PEGs are recognized: as we want to recognize $\langle start \rangle$, we first need to recognize $\langle pat \rangle$, and the first rule to use in this is $\langle pat \rangle \leftarrow \langle atpat \rangle\ \langle tyop \rangle$. The attempt is fruitful: “t” is an atomic pattern, and we recognize a $\langle tyop \rangle$ as the empty string. Since the first rule of $\langle pat \rangle$ was successfully matched, we now attempt to read the “;” symbol of the $\langle start \rangle$ rule, which fails. In the absence of any other alternative rule for $\langle start \rangle$, the sentence is rejected.

Our implementation of $\mathcal{G}_1 \cup \mathcal{G}_2$ does not recognize the expected language: we did not notice that the language recognized by rule $\langle pat \rangle \leftarrow \langle atpat \rangle\ \langle tyop \rangle$ could be a prefix of the one recognized by $\langle pat \rangle \leftarrow vid\ \langle tyop \rangle\ as\ \langle pat \rangle\ \langle tyop \rangle$. The order then ruled out the possibility for the second choice to be explored. The fix is obvious: we should add the layered pattern rule as a first choice:

```
generic Pattern +=
    <Layered> ValueID TypeOp void:"as":Keyword Pattern TypeOp
  / <Atomic> ...
  ;
```

2.4 Why Verify?

Our tests revealed two issues with our implementations in SDF2 and *Rats!*. The issues are not in the way the tools handled their input grammars, but in the grammatical specifications themselves. Our example is very simple, and the solutions were obvious; with some experience with the formalisms, we could have identified and avoided the problems from the start. Nevertheless, the issues we found in our grammars by testing are quite serious: in general, it is undecidable whether they arise in a given SDF2 or PEG specification.

Classical deterministic parser generators give more guarantees than just performance: they also warrant that the syntax is unambiguous. Ambiguity is undesirable in most computer languages, which are mediums for communicating with quite a finicky and unforgiving entity, the computer. When moving from classical parser generators to the new parsing techniques, this static check is lost.

The two formalisms presented in this paper use very different approaches to ambiguity:

- The disambiguation filters of SDF2 are added to the context-free grammar, without any guarantee that all the ambiguous cases will disappear: ambiguity detection in context-free grammars is a classical undecidable problem in formal language theory [4, 6].
- The ordering is inherent to the PEG specification, and rules out ambiguities. Unfortunately, it might give unexpected results: the disjointness problem becomes undecidable with parsing expression grammars (Section 4.1).

Of course, over-zealous filters could be as bad as ordering, and relaxed ordering schemes might also fall prey to incomplete disambiguation. From a reliability standpoint, both disambiguation strategies have their flaws.

Testing is an important part of the solution, but we believe that the famous quote by Dijkstra [7] applies for grammars as well as it does for programs:

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.

The following Sections 3 and 4 thus present conservative verification algorithms for the two decision problems we just witnessed.

3 Ambiguity in SDF2

3.1 Nondeterministic Automaton

In order to look for ambiguities in our grammar, we need a finite structure abstracting the full graph of all grammar developments. The nondeterministic LR(0) [14] automaton is one such abstraction. We consider a variant constructed as a quotient of the full grammar graph Γ using a local (rule-based) equivalence relation item_0 [29].

Rather than directly building the nondeterministic automaton for the CFG $\mathcal{G}$ at hand, we build it for a bracketed version $\mathcal{G}_b$ of $\mathcal{G}$, where rules $i = A{\rightarrow}\alpha$ of $\mathcal{G}$ are surrounded by d_i and r_i. Formally, our *bracketed grammar* of a context-free grammar $\mathcal{G}$ is the context-free grammar $\mathcal{G}_b = \langle N, T_b, P_b, S\rangle$ where $T_b = T \cup \{d_i \mid i \in P\} \cup \{r_i \mid i \in P\}$ and $P_b = \{i_b = A{\rightarrow}d_i\alpha r_i \mid i = A{\rightarrow}\alpha \in P\}$. A sentence of $\mathcal{G}_b$ represents a single parse tree of $\mathcal{G}$. We define the homomorphism h from V_b^* to V^* by $h(d_i) = h(r_i) = \varepsilon$ for all i in P, and $h(X) = X$ otherwise. We sometimes write δ_b to denote a bracketed string in V_b^* such that $h(\delta_b) = \delta$.

Definition 1 *The* nondeterministic automaton Γ/item_0 *of a context-free grammar $\mathcal{G}$ is a tuple $\langle Q, V_b, R, q_s, Q_f\rangle$ where*

- $Q = \{[\bullet A] \mid A \in N\} \cup \{[A\bullet] \mid A \in N\} \cup \{[A{\rightarrow}\alpha\bullet\alpha'] \mid A{\rightarrow}\alpha\alpha' \in P\}$ *is the state alphabet, where $[A{\rightarrow}\alpha\bullet\alpha']$ is called a LR(0) item,*

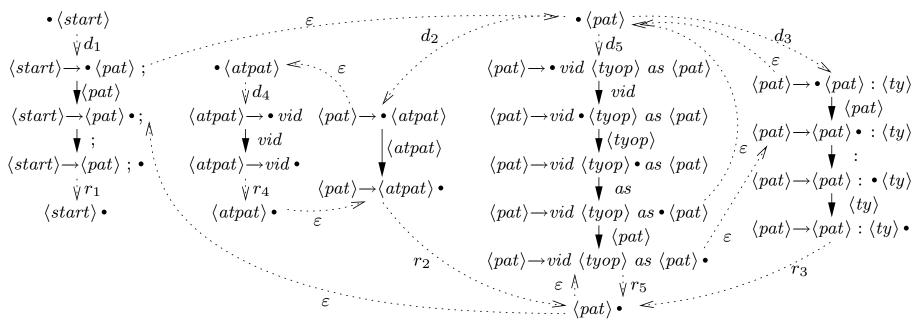

Figure 2: A portion of the nondeterministic automaton for our SDF2 implementation of $\mathcal{G}_1 \cup \mathcal{G}_2$.

- V_b *is the input alphabet,*
- R *is the set of rules*

$$\begin{aligned}
&\{[A{\to}\alpha \bullet X\alpha']X \vdash [A{\to}\alpha X \bullet \alpha']\}\\
\cup&\{[A{\to}\alpha \bullet B\alpha']\varepsilon \vdash [\bullet B]\}\\
\cup&\{[\bullet B]d_i \vdash [B{\to}\bullet \beta] \mid i = B{\to}\beta \in P\}\\
\cup&\{[B{\to}\beta \bullet]r_i \vdash [B\bullet] \mid i = B{\to}\beta \in P\}\\
\cup&\{[B\bullet]\varepsilon \vdash [A{\to}\alpha B \bullet \alpha']\},
\end{aligned}$$

- $q_s = [\bullet S]$ *is the initial state, and*
- $Q_f = \{q_f = [S\bullet]\}$ *is the set of final states.*

Figure 2 presents a portion of the nondeterministic automaton for the augmented union $\mathcal{G}_1 \cup \mathcal{G}_2$.

Theorem 1 *The language generated by* $\mathcal{G}_b$ *is included in the terminal language recognized by* Γ/item_0*:* $\mathcal{L}(\mathcal{G}_b) \subseteq \mathcal{L}(\Gamma/\mathsf{item}_0) \cap T_b^*$.

3.2 An Ambiguity Detection Algorithm

Since a derivation tree for $\mathcal{G}$ is uniquely identified by a sentence of $\mathcal{G}_b$, the existence of an ambiguity in $\mathcal{G}$ implies the existence of two sentences w_b and w'_b in $\mathcal{L}(\mathcal{G}_b)$ such that $w = w'$. By Theorem 1, if such two sentences exist in $\mathcal{L}(\mathcal{G}_b)$, they also do in $\mathcal{L}(\Gamma/\mathsf{item}_0) \cap T_b^*$, and thus we can look for their existence in Γ/item_0. In order to find such sentences in Γ/item_0, we use an accessibility relation between couples of states.

Definition 2 *The* mutual accessibility relation ma *is defined over* Q^2 *as the union* $\mathsf{mal} \cup \mathsf{mar} \cup \mathsf{maa}$*, where*

$$\begin{aligned}
(q_1, q_2)\ \mathsf{mal}\ (q_3, q_2) &\text{ iff } q_1\varepsilon \vdash q_3, \text{ or } \exists i \in P, q_1 d_i \vdash q_3 \text{ or } q_1 r_i \vdash q_3,\\
(q_1, q_2)\ \mathsf{mar}\ (q_1, q_4) &\text{ iff } q_2\varepsilon \vdash q_4, \text{ or } \exists i \in P, q_2 d_i \vdash q_4 \text{ or } q_2 r_i \vdash q_4,\\
(q_1, q_2)\ \mathsf{maa}\ (q_3, q_4) &\text{ iff } \exists a \in T, q_1 a \vdash q_3 \text{ and } q_2 a \vdash q_4.
\end{aligned}$$

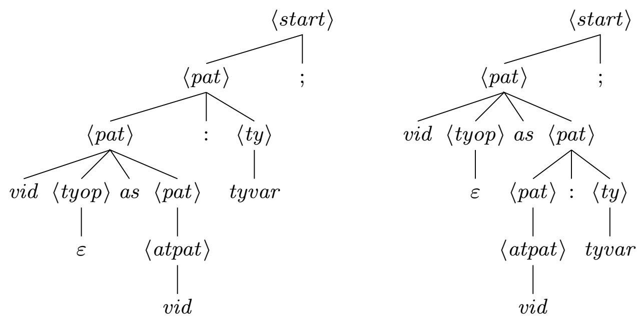

Figure 3: An ambiguity in our SDF2 implementation of $\mathcal{G}_1 \cup \mathcal{G}_2$.

Proposition 1 *Let q_1, q_2, q_3, q_4 be states in Q, and u_b and v_b strings in T_b^* such that $q_1 u_b \models^* q_3$, $q_2 v_b \models^* q_4$. The relation (q_1, q_2) ma^* (q_3, q_4) holds if and only if $u = v$.*

Let us now consider two different parse trees in $\mathcal{G}$ represented by two different strings w_b and w'_b in V_b^* with $w = w'$. The sentences w_b and w'_b share a longest common suffix v_b, such that $w_b = u_b r_i v_b$ and $w'_b = u'_b r_j v_b$ with $i \neq j$. There are thus three states q_i, q_j and q in Q such that $q_s u_b r_i v_b \models^* q_i r_i v_b \models q v_b \models^* q_f$ and $q_s u'_b r_j v_b \models^* q_j r_j v_b \models q v_b \models^* q_f$. By Proposition 1, (q_s, q_s) ma^* (q_i, q_j).

Our algorithm for finding all the ambiguities in a given grammar $\mathcal{G}$ is therefore

1. Construct Γ/item_0: this automaton is of size $|\Gamma/\mathsf{item}_0| = \mathcal{O}(|\mathcal{G}|)$ and can be constructed in time linear with its size.

2. Compute the (symmetric and reflexive) image ma^* $(\{(q_s, q_s)\})$; this computation costs at worst $\mathcal{O}(|\Gamma/\mathsf{item}_0|^2)$ [31].

3. Explore this image and find all couples (q_i, q_j) such that $i \neq j$, $q_i r_i \vdash q$ and $q_j r_j \vdash q$; a state in Q has at most one transition on a r_i symbol, thus this search costs at worst $\mathcal{O}(|\Gamma/\mathsf{item}_0|^2)$.

The overall complexity is thus $\mathcal{O}(|\mathcal{G}|^2)$.

As an illustration, assuming production 6 is $\langle ty \rangle \rightarrow tyvar$ and production 7 $\langle tyop \rangle \rightarrow \varepsilon$, two bracketed sentences representing the trees shown in Figure 3 are

$$d_1 d_3 d_5\ vid\ d_7 r_7\ as\ d_2 d_4\ vid\ r_4 r_2 r_5 : d_6\ tyvar\ r_6 r_3 \ ; \ r_1$$
$$d_1 d_5\ vid\ d_7 r_7\ as\ d_3 d_2 d_4\ vid\ r_4 r_2 : d_6\ tyvar\ r_6 r_3 r_5 \ ; \ r_1$$

We can follow the paths in Figure 2 and see that

$$([\bullet \langle start \rangle], [\bullet \langle start \rangle])\ \mathsf{ma}^*\ ([\langle pat \rangle \rightarrow \langle pat \rangle\ \langle ty \rangle \bullet], [\langle pat \rangle \rightarrow vid\ \langle tyop \rangle\ as\ \langle pat \rangle \bullet]).$$

Since furthermore $[\langle pat \rangle \rightarrow \langle pat \rangle\ \langle ty \rangle \bullet]$ and $[\langle pat \rangle \rightarrow vid\ \langle tyop \rangle\ as\ \langle pat \rangle \bullet]$ have transitions to $[\langle pat \rangle \bullet]$ on r_3 and r_5 respectively, we can conclude that it is possible to have an ambiguity arising from the use of productions 3 and 5. Figure 3 confirms this potential ambiguity to be very real.

3.3 Integrating Filters

Disambiguation filters are often defined by specifying which trees in a set of possible parse trees are "wrong" and removing them from the set of trees returned by the parser [19]. Depending on the amount of context needed by a filter to decide whether a tree is acceptable or not, we can implement the result of the filter *a priori* in our ambiguity detection algorithm.

For instance, the preference attribute `prefer` filter [33] does not depend on much context and can be implemented directly in our scheme. In case of an ambiguity, two (sub)trees yielding the same terminal language share a single root node, but then differ on which rule to follow first. If one of the two rules is marked as preferred, it will be chosen over the other, thus resolving the ambiguity. We can emulate this strategy at the last step of our algorithm and not report ambiguities on a pair (q_i, q_j) if one of i and j is marked as preferred. The dual `avoid` filter can be implemented similarly. This will successfully avoid reporting the ambiguity shown in Figure 3 in presence of a `prefer` directive.

Some other default filters, like follow restrictions or associativity and priority rules, can be implemented provided we use a nondeterministic LR(1) automaton instead of the nondeterministic LR(0) one. In general, the quotienting equivalence relation on the full grammar graph should not allow a "wrong" tree to be equivalent to a "correct" one.

4 Disjointness in PEGs

4.1 The Semi Disjointness Problem

A choice expression e_1/e_2 is commutative if e_1 and e_2 are *disjoint*: their recognized languages $\mathcal{M}(e_1)$ and $\mathcal{M}(e_2)$ are disjoint sets. As seen with our example in Section 2.3, the question we would like to answer to is not whether two parsing expression are disjoint (which is undecidable [10]), but more precisely whether the second choice will be considered during parsing.

For this, we need a stronger notion of the language recognized by a PEG. As recalled in Appendix A.2, if the language $\mathcal{M}(e)$ matched by a parsing expression e contains the string x, then it also contains the string xy for any y in T^*.

Definition 3 *The* minimal matching set *of a parsing expression e is $\mathcal{L}(e) = \{y \in T^* \mid (x, e) \Rightarrow^+ y\}$. Then, $\mathcal{M}(e) = \mathcal{L}(e) \cdot T^*$.*

Let the prefix language of a language L be $\text{Prefix}(L) = \{x \mid xy \in L\}$. Then $\mathcal{M}(e_1) \cap \mathcal{M}(e_2) = \emptyset$ can be rewritten as $(\mathcal{L}(e_1) \cdot T^*) \cap (\mathcal{L}(e_2) \cdot T^*) = \emptyset$, also equivalent to $(\mathcal{L}(e_1) \cap \text{Prefix}(\mathcal{L}(e_2))) \cup (\mathcal{L}(e_2) \cap \text{Prefix}(\mathcal{L}(e_1))) = \emptyset$ by factoring out T^*.

Definition 4 *The parsing expressions e_1 and e_2 are* semi disjoint *if and only if $\mathcal{L}(e_1) \cap \text{Prefix}(\mathcal{L}(e_2)) = \emptyset$.*

The decomposition of the disjoint problem as the union of two instances of the semi disjoint problem shows that any algorithm solving the semi disjoint problem could be used to solve the disjoint problem, hence the following proposition.

Proposition 2 *The semi disjointness of two parsing expressions is undecidable.*

The notions of minimal matching set and of semi disjointness are directly connected to the issue at hand, as shown by the following characterization.

Theorem 2 *Two parsing expressions e_1 and e_2 are semi disjoint if and only if $\mathcal{L}(e_1/e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$ and $\mathcal{L}(e_1) \cap \mathcal{L}(e_2) = \emptyset$.*

4.2 General Semi Disjointness

Semi disjointness does not encompass all the issues created by the ordered choice operator in PEGs. Let us consider for instance the parsing expression $(aa/a)a$. The expressions aa and a are semi disjoint, and the language $\mathcal{L}(aa/a)$ is clearly $\{aa, a\}$. However, the language $\mathcal{L}((aa/a)a)$ is $\{aaa\}$ and not $\{aaa, aa\}$: an input string aa is always matched by the first alternative aa, after which another a is expected but not found, and the input is rejected.

For the general semi disjointness problem, we consider a parsing expression e that has an ordered choice e_1/e_2 as a *prefix* subexpression,[5] denoted by $e[e_1/e_2]$. If we replace this ordered choice by e_2 alone, we obtain an expression $e[e_2]$. The general semi disjointness problem is thus to verify that $\mathcal{L}(e_1) \cap \text{Prefix}(\mathcal{L}(e[e_2])) = \emptyset$ in all the possible $e[e_1/e_2]$ of a PEG. The previous example was an instance of the problem with $e[e_1/e_2] = (aa/a)a$ and $e[e_2] = aa$, and e_1 and e_2 were not generally semi disjoint since $\mathcal{L}(e_1) \cap \text{Prefix}(\mathcal{L}(e[e_2])) = \{aa\}$.

4.3 Checking a PEG for General Semi Disjointness

Repetition-free and predicate-free PEGs are equivalent to CFGs with a total order $<$ on the rules sharing the same left-hand side: a parsing expression $e = \alpha_1/\alpha_2/\ldots/\alpha_n$ can be represented by a set of CFG productions $1 = A \leftarrow \alpha_1$, $2 = A \leftarrow \alpha_2$, …, $n = A \leftarrow \alpha_n$ with the ordering $1 < 2 < \cdots < n$. We call such a CFG the *ordered context-free form* of the PEG.

We can apply the construction of Section 3.1 on this CFG to obtain a nondeterministic automaton, and use the mutual accessibility relation of Definition 2 on couples of states in Q. Figure 4 shows a portion of the nondeterministic automaton in the case of our *Rats!* grammar; we had to convert $\langle tyop \rangle \leftarrow (: \langle ty \rangle)?$ into $\langle tyop \rangle \leftarrow : \langle ty \rangle / \varepsilon$ in order to obtain its ordered context-free form.

Our algorithm for finding all the potential disjointness issues in a given PEG $\mathcal{G}$ in ordered context-free form is therefore

1. Construct Γ/item_0 as in the case of ambiguity checking.
2. For all the couples of productions i and j in P sharing the same left-hand side A with $i < j$, compute the image $\mathsf{ma}^*\ (\{(q_i, q_j)\})$ where $[\bullet A]d_i \vdash q_i$ and $[\bullet A]d_j \vdash q_j$; this computation costs at worst $\mathcal{O}(|\Gamma/\mathsf{item}_0|^2 |P|^2)$.
3. Explore each of these images and find all couples (q'_i, q) such that $q'_i r_i \vdash [A\bullet]$; this search costs at worst $\mathcal{O}(|\Gamma/\mathsf{item}_0|^2 |P|^2)$.

The overall complexity is thus $\mathcal{O}(|\mathcal{G}|^2 |P|^2)$.

We can illustrate this process by examining the couple

$$([\langle pat \rangle \leftarrow \bullet \langle atpat \rangle\ \langle tyop \rangle], [\langle pat \rangle \leftarrow \bullet\, vid\ \langle tyop \rangle\ as\ \langle pat \rangle\ \langle tyop \rangle])$$

[5]The parsing expression e_1 is a prefix subexpression of $e_1 e_2$, and both e_1 and e_2 are prefix subexpressions of e_1/e_2.

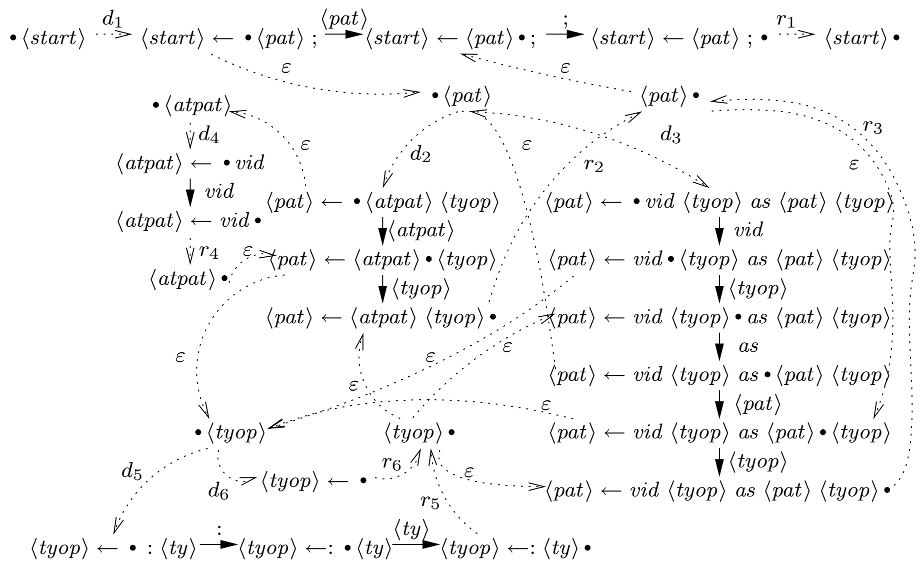

Figure 4: A portion of the nondeterministic automaton for our *Rats!* implementation of $\mathcal{G}_1 \cup \mathcal{G}_2$.

that matches the requirements of step 2 above, and by observing on Figure 4 that it reaches by ma^* the couple

$$([\langle pat\rangle \leftarrow \langle atpat\rangle\ \langle tyop\rangle \bullet], [\langle pat\rangle \leftarrow vid\ \langle tyop\rangle \bullet as\ \langle pat\rangle\ \langle tyop\rangle])$$

that matches the requirements of step 3 above. This indicates that it is possible to see some of the language of production 3, starting in

$$[\langle pat\rangle \leftarrow vid\ \langle tyop\rangle \bullet as\ \langle pat\rangle\ \langle tyop\rangle],$$

unused because of the priority given to production 2. Exchanging the priorities of productions 2 and 3 would make this particular issue disappear.

5 Related Work

The verification of grammars has been approached in the past: common checks look for nonterminating grammar rules (including cycles) or left or hidden left recursions. These verifications are all decidable.

Ambiguity Detection To the best of our knowledge, almost all the algorithms that have been specifically designed so far for ambiguity detection look for ambiguities in all sentences up to some length [11, 5, 30, 16]. As such, they do not qualify as verification tools since ambiguities can appear after this preset length. On the other hand, testing that a deterministic parser could be constructed for a given grammar [15] vouches for its unambiguity, and indicates where probable ambiguities are [27]. Nevertheless, conflicts in a negative LR(k) test do not necessarily stem from real ambiguities, but from the need of a longer, if not unbounded, lookahead in the conflict resolution; such cases are not so uncommon, as seen in Section 2.1.

Two better unambiguity checks have been recently published, by Brabrand *et al.* [3] and by the author [29].

Disjointness Although Ford hinted that verification tools for PEGs could find their uses [10], we are not aware of any such tools.

Others The quotienting approach used in this paper is reminiscent of the approaches used to build regular approximations of context-free languages [25]. The nondeterministic automata are akin to Transition Networks [37] and Recursive State Machines [1].

6 Conclusion

While modular syntax formalisms encourage good engineering practices in grammar and grammar-related software development, their use is not without risk. Indeed, we have been able to exhibit two potential issues when using two modern syntax formalisms: SDF2 definitions and PEGs. Meanwhile, testing grammars and grammar-related software is not entirely satisfactory since it cannot guarantee the absence of an issue. We have thus advocated in this paper the introduction of verification tools in the future Computer-Aided Grammarware Engineering (CAGE) environments foreseen by Klint *et al.* [20].

The SDF2 and PEGs formalisms are representative of two very different approaches to modular syntax. Nevertheless, we are inclined to explain the ambiguity and disjointness problems that hamper their use as two instances of a single *disambiguation* issue that would be faced by all modular formalisms. This vue is supported by the blatant similarities between the solutions we have provided for their verification problems.

We hope that, in the future, more work will be dedicated to the verification of grammatical descriptions, and to more elaborate verification algorithms for the problems exposed here in particular.

Acknowledgements The author is highly grateful to Jacques Farré, Ana Almeida Matos and Jan Cederquist for their help in the preparation of this work.

References

[1] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems*, 27 (4):786–818, 2005. ISSN 0164-0925. doi: 10.1145/1075382.1075387.

[2] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, 1973. ISSN 0019-9958. doi: 10.1016/S0019-9958(73)90851-6.

[3] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. Technical Report RS-06-09, BRICS, University of Aarhus, May 2006. URL `http://www.brics.dk/~brabrand/grambiguity/`.

[4] David G. Cantor. On the ambiguity problem of Backus systems. *Journal of the ACM*, 9(4):477–479, 1962. ISSN 0004-5411. doi: 10.1145/321138.321145.

[5] Bruce S. N. Cheung and Robert C. Uzgalis. Ambiguity in context-free grammars. In *SAC'95*, pages 272–276. ACM Press, 1995. ISBN 0-89791-658-1. doi: 10.1145/315891.315991.

[6] Noam Chomsky and Marcel Paul Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, 1963.

[7] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. ISSN 0001-0782. doi: 10.1145/355604.361591. ACM Turing Award Lecture.

[8] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. ISSN 0001-0782. doi: 10.1145/362007.362035.

[9] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. In *ICFP '02*, pages 36–47. ACM Press, 2002. ISBN 1-58113-487-8. doi: 10.1145/581478.581483.

[10] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04*, pages 111–122. ACM Press, 2004. ISBN 1-58113-729-X. doi: 10.1145/964001.964011.

[11] Saul Gorn. Detection of generative ambiguities in context-free mechanical languages. *Journal of the ACM*, 10(2):196–208, 1963. ISSN 0004-5411. doi: 10.1145/321160.321168.

[12] Robert Grimm. Better extensibility through modular syntax. In *PLDI '06*, pages 38–51. ACM Press, 2006. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133987.

[13] Mark Hennessy and James F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *ASE'05*, pages 104–113. ACM Press, 2005. ISBN 1-59593-993-4. doi: 10.1145/1101908.1101926.

[14] Harry B. Hunt III, Thomas G. Szymanski, and Jeffrey D. Ullman. Operations on sparse relations and efficient algorithms for grammar problems. In *15th Annual Symposium on Switching and Automata Theory*, pages 127–132. IEEE Computer Society, 1974.

[15] Harry B. Hunt III, Thomas G. Szymanski, and Jeffrey D. Ullman. On the complexity of LR(k) testing. *Communications of the ACM*, 18(12):707–716, 1975. ISSN 0001-0782. doi: 10.1145/361227.361232.

[16] Saichaitanya Jampana. Exploring the problem of ambiguity in context-free grammars. Master's thesis, Oklahoma State University, July 2005. URL `http://e-archive.library.okstate.edu/dissertations/AAI1427836/`.

[17] Stephen C. Johnson. YACC — yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, July 1975.

[18] Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, LFCS, 1993. URL `http://www.lfcs.inf.ed.ac.uk/reports/93/ECS-LFCS-93-257/`.

[19] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *ASMICS Workshop on Parsing Theory*, Technical Report 126-1994, pages 89–100. Università di Milano, 1994. URL `http://citeseer.ist.psu.edu/klint94using.html`.

[20] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, 2005. ISSN 1049-331X. doi: 10.1145/1072997.1073000.

[21] Ralf Lämmel. Grammar testing. In Heinrich Hussmann, editor, *FASE'01*, volume 2029 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2001. ISBN 3-540-41863-6. URL `http://www.springerlink.com/content/a799v1rfled2hd2y/`.

[22] Ralf Lämmel and Chris Verhoef. Semi-automatic grammar recovery. *Software: Practice & Experience*, 31:1395–1438, 2001. doi: 10.1002/spe.423.

[23] Brian A. Malloy, James F. Power, and John T. Waldron. Applying software engineering techniques to parser design: the development of a C# parser. In *SAICSIT'02*, pages 75–82. SAICSIT, 2002. ISBN 1-58113-596-3. URL `http://www.cs.nuim.ie/~jpower/Research/Papers/2002/saicsit02.pdf`.

[24] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML.* MIT Press, revised edition, 1997. ISBN 0-262-63181-4.

[25] Mark-Jan Nederhof. Regular approximation of CFLs: a grammatical view. In H. Bunt and A. Nijholt, editors, *Advances in Probabilistic and other Parsing Technologies*, chapter 12, pages 221–241. Kluwer Academic Publishers, 2000. ISBN 0-7923-6616-6. URL `http://odur.let.rug.nl/~markjan/publications/2000d.pdf`.

[26] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972. ISSN 0006-3835. doi: 10.1007/BF01932308.

[27] Janina Reeder, Peter Steffen, and Robert Giegerich. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6:153, 2005. ISSN 1471-2105. doi: 10.1186/1471-2105-6-153.

[28] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *PLDI'89*, pages 170–178. ACM Press, 1989. ISBN 0-89791-306-X. doi: 10.1145/73141.74833.

[29] Sylvain Schmitz. Conservative ambiguity detection in context-free grammars. Technical Report I3S/RR-2006-30-FR, Laboratoire I3S, Université de Nice - Sophia Antipolis, September 2006. URL `http://www.i3s.unice.fr/~mh/RR/2006/RR-06.30-S.SCHMITZ.pdf`.

[30] Friedrich Wilhelm Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, compilertools.net, 2001. URL `http://accent.compilertools.net/Amber.html`.

[31] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. ISSN 0097-5397. doi: 10.1137/0201010.

[32] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986. ISBN 0-89838-202-5.

[33] Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In R. Nigel Horspool, editor, *CC'02*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002. ISBN 3-540-43369-4. URL `http://www.springerlink.com/content/03359k0cerupftfh/`.

[34] Mark van den Brand, Steven Klusener, Leon Moonen, and Jurgen J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. *Electronic Notes in Theoretical Computer Science*, 82(3):575–591, 2003. doi: 10.1016/S1571-0661(05)82629-5.

[35] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, 1997. URL `http://citeseer.ist.psu.edu/visser97syntax.html`.

[36] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam, July 1997. URL `http://citeseer.ist.psu.edu/visser97scannerles.html`.

[37] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, 1970. ISSN 0001-0782. doi: 10.1145/355598.362773.

A Definitions

A.1 Context-Free Grammars

We quickly recall the formal definition of context-free grammars. They are tuples $\mathcal{G} = \langle N, T, P, S\rangle$ where N is the nonterminal alphabet, T the terminal one—$V = N \cup T$ being the vocabulary—, P is a finite set of rules i in $N \times V^*$ denoted by $A \rightarrow \alpha$, and S is the start symbol in N. The derivation relation $\Rightarrow$ in $(V^*)^2$ is defined by $\delta A\sigma \overset{i}{\Rightarrow} \delta\alpha\sigma$ if and only if $i = A \rightarrow \alpha$ is a rule of P.

The language generated by $\mathcal{G}$ is the set of sentences $\mathcal{L}(\mathcal{G}) = \{w \mid S \Rightarrow^* w\}$. A grammar $\mathcal{G}$ is reduced if all the symbols X in V are such that $S \Rightarrow^* \delta X\sigma \Rightarrow^* w$ for some δ, σ in V^* and w in T^*. We always assume our context-free grammars to be reduced. A grammar $\mathcal{G} = \langle N, T, P, S\rangle$ can be augmented to a grammar $\mathcal{G}' = \langle N', T', P', S'\rangle$ where $N' = N \cup \{S'\}$, $T' = T \cup \{\$\}$ and $P' = P \cup \{S' \overset{1}{\rightarrow} \$S\$\}$.

The traditional notation uses the first capital Latin letters A, B, C for nonterminals in N, the first small-case Latin letters a, b, c for terminal symbols in T, the last capital Latin letters X, Y, Z for symbols in V, the last small-case Latin letters $u \ldots z$ for strings in T^*, and Greek letters α, β and so on for strings in V^*. The empty string is denoted by ε. For practical computer languages grammars as given in Section 2, long names are more readable, and thus ⟨*nonterminals*⟩ are given between angle brackets while *terminals* are simply written as such. Context-free rules with the same nonterminal as left part are often associated with a union operator $|$, so that $A \rightarrow \alpha \mid \beta$ stands for the two rules $A \rightarrow \alpha$ and $A \rightarrow \beta$.

A.2 Parsing Expression Grammars

PEGs A parsing expression grammar is a tuple $\mathcal{G} = \langle N, T, R, e_S\rangle$, where N is the nonterminal alphabet, T the terminal alphabet, R a mapping from nonterminal symbols in N to expressions in E and e_S is a distinguished expression. An expression in E is inductively defined as ε the empty string, or a a terminal in T, or A a nonterminal in N, or e_1e_2 a concatenation of two expressions, or e_1/e_2 an ordered choice between two expressions. Expressions using the Kleene star, negations or conjunctions can be rewritten in terms of primitive expressions, provided the grammar does not recognize the empty string—using an augmented grammar, this concern can be avoided. Left recursions can be conservatively avoided as well. We thus consider our PEGs to be well-formed, repetition-free and predicate-free.

Interpretation Following the notations of Ford [10], the interpretation relation $\Rightarrow$ between pairs (e, x) in $E \times T^*$ and pairs (n, o) in $\mathbb{N} \times (T^* \cup \{f\})$ gives the result of a recognition attempt of x by e: n is a step counter and o is the recognized prefix of x, or f if recognition failed. Inductively,

empty $(\varepsilon, x) \Rightarrow (1, \varepsilon)$,

terminal $(a, ax) \Rightarrow (1, a)$, $(a, bx) \Rightarrow (1, f)$, $(a, \varepsilon) \Rightarrow (1, f)$,

nonterminal $(A, x) \Rightarrow (n + 1, o)$ if $A \leftarrow e$ and $(e, x) \Rightarrow (n, o)$,

concatenation $(e_1e_2, x_1x_2y) \Rightarrow (n_1 + n_2 + 1, x_1x_2)$ if $(e_1, x_1x_2y) \Rightarrow (n_1, x_1)$ and $(e2, x_2y) \Rightarrow (n_2, x_2)$, $(e_1e_2, x) \Rightarrow (n_1{+}1, f)$ if $(e_1, x) \Rightarrow (n_1, f)$, $(e_1e_2, x_1y) \Rightarrow (n_1 + n_2 + 1, f)$ if $(e_1, x_1y) \Rightarrow (n_1, x_1)$ and $(e_2, y) \Rightarrow (n_2, f)$,

choice $(e_1/e_2, xy) \Rightarrow (n_1 + 1, x)$ if $(e_1, xy) \Rightarrow (n_1, x)$, $(e_1/e_2, x) \Rightarrow (n_1 + n_2 + 1, o)$ if $(e_1, x) \Rightarrow (n_1, f)$ and $(e_2, x) \Rightarrow (n_2, o)$.

The relation $\Rightarrow^+$ disregards step counts: $(e, x) \Rightarrow^+ o$ if and only if there exists n such that $(e, x) \Rightarrow (n, o)$. The match set of expression e is $\mathcal{M}(e) = \{x \mid (e, x) \Rightarrow^+ y, y \in T^*\}$. Note that if x is in $\mathcal{M}(e)$, then, for any y in T^*, xy is also. The language recognized by $\mathcal{G}$ is $\mathcal{M}(e_S)$.

B Implementations

B.1 Implementation of $\mathcal{G}_1$ in SDF2

We only show here the portions of the SDF file corresponding to the rules of $\mathcal{G}_1$; the modularity of SDF allowed us to separate the other concerns in different modules `MLTypes`, `MLIdentifiers` and `Layout`.

```
definition

module Main
imports
  MLPatterns MLTypes MLIdentifiers Layout
exports
  context-free start-symbols START
  sorts START
  context-free syntax
    PAT ";"                     -> START

module MLPatterns
imports
  MLTypes MLIdentifiers Layout
exports
  sorts PAT ATPAT
  context-free syntax
    ATPAT                       -> PAT
    PAT ":" TY                  -> PAT
    "(" {PAT ","}* ")"          -> ATPAT
    "_"                         -> ATPAT
    VID                         -> ATPAT
```

B.2 Implementation of $\mathcal{G}_1$ in *Rats!*

Packrat parsers being of the recursive descent family, we had to modify $\mathcal{G}_1$ in order to remove the left recursion in $\langle pat \rangle \rightarrow \langle pat \rangle : \langle ty \rangle$. We also took advantage of the extended regular operators `?` and `*` offered by *Rats!*.

```
module SimpleMLPatterns;
import MLLexing;
import MLTypes;

public generic Pattern =
    <Atomic>   AtomicPattern  TypeOp
  ;
generic TypeOp  = (void:":":Symbol Type )? ;

generic AtomicPattern =
    <Tuple>    void:"(":Symbol PatternList? void:")":Symbol
  / <Wildcard> "_":Symbol
  / <Variable> ValueID
  ;
generic PatternList = Pattern ( void:",":Symbol Pattern )* ;
```

C Omitted Proofs

C.1 Ambiguity in SDF2

Lemma 1 *Let q and q' be two states of Γ/item_0 such that $qB \vdash q'$ is a rule in R. If $i = B{\rightarrow}\beta$ is a production in P, then $qd_i\beta r_i \models^* q'$.*

Proof. By Definition 1, such a rule can only occur between $q = [A{\rightarrow}\alpha\,{\bullet}\,B\alpha']$ and $q' = [A{\rightarrow}\alpha B\,{\bullet}\,\alpha']$ for some $A{\rightarrow}\alpha B\alpha'$ in P. Then, there also exist the rules $[A{\rightarrow}\alpha\,{\bullet}\,B\alpha']\varepsilon \vdash [{\bullet}B]$, $[{\bullet}B]d_i \vdash [B{\rightarrow}{\bullet}\beta]$, $[B{\rightarrow}\beta{\bullet}]r_i \vdash [B{\bullet}]$, and $[B{\bullet}]\varepsilon \vdash [A{\rightarrow}\alpha B\,{\bullet}\,\alpha']$ in R, and a trivial induction on the length $|\beta|$ shows that $[B{\rightarrow}{\bullet}\beta]\beta \models^* [B{\rightarrow}\beta{\bullet}]$, proving the lemma. □

Lemma 2 *Let q and q' be two states of Γ/item_0 such that $q\delta_b \models^* q'$. If $\delta_b \Rightarrow^* \gamma_b$ in $\mathcal{G}_b$, then $q\gamma_b \models^* q'$.*

Proof. We proceed by induction on the number n of individual derivations in $\delta_b \Rightarrow^n \gamma_b$. If $n = 0$, then $\delta_b = \gamma_b$ and the lemma trivially holds. We then consider for the induction step $\delta_b \Rightarrow^n \varphi_b A\sigma_b \overset{i}{\Rightarrow} \varphi_b d_i\alpha r_i\sigma_b$; using the induction hypothesis, $q\varphi_b A\sigma_b \models^* q_A A\sigma_b \models q'_A\sigma_b \models^* q'$. By Lemma 1, $q_A d_i\alpha r_i \models^* q'_A$ holds, which concludes the proof. □

Theorem 1 *The language generated by $\mathcal{G}_b$ is included in the terminal language recognized by Γ/item_0: $\mathcal{L}(\mathcal{G}_b) \subseteq \mathcal{L}(\Gamma/\mathsf{item}_0) \cap T_b^*$.*

Proof. Immediate application of Lemma 2 with $q = [{\bullet}S]$, $q' = [S{\bullet}]$, $\delta_b = \delta$ with $S{\rightarrow}\delta$ in P and $\gamma_b = w_b$ any sentence in $\mathcal{L}(\mathcal{G}_b)$ derived from δ. □

C.2 Disjointness in PEGs

Theorem 2 *Two parsing expressions e_1 and e_2 are semi disjoint if and only if $\mathcal{L}(e_1/e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$ and $\mathcal{L}(e_1) \cap \mathcal{L}(e_2) = \emptyset$.*

Proof. Only if part: let us suppose e_1 and e_2 are not semi disjoint. Then, there exist two strings x and y in T^* such that x is in $\mathcal{L}(e_1)$ and xy in $\mathcal{L}(e_2)$. Either $y = \varepsilon$, but then $\mathcal{L}(e_1) \cap \mathcal{L}(e_2) \neq \emptyset$, or $y \neq \varepsilon$, but then $(xy, e_1/e_2) \Rightarrow^+ x$ and thus xy would not belong to $\mathcal{L}(e_1/e_2)$.

If part: let us consider x in $\mathcal{L}(e_2)$; it is not in $\mathcal{L}(e_1)$ since $\mathcal{L}(e_1) \cap \mathcal{L}(e_2) = \emptyset$, nor is any of its prefixes, since otherwise it would not be in $\mathcal{L}(e_1/e_2)$. Thus $\mathcal{L}(e_1) \cap \text{Prefix}(\mathcal{L}(e_2)) = \emptyset$. $\square$