View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

Ambiguity Detection: Scaling to Scannerless

H. J. S. Basten and P. Klint and J. J. Vinju

Centrum Wiskunde & Informatica
Science Park 123, 1098 XG Amsterdam, The Netherlands

Abstract. Static ambiguity detection would be an important aspect of language workbenches for textual software languages. However, the challenge is that automatic ambiguity detection in context-free grammars is undecidable in general. Sophisticated approximations and optimizations do exist, but these do not scale to grammars for so-called "scannerless parsers", as of yet. We extend previous work on ambiguity detection for context-free grammars to cover disambiguation techniques that are typical for scannerless parsing, such as longest match and reserved keywords. This paper contributes a new algorithm for ambiguity detection in character-level grammars, a prototype implementation of this algorithm and validation on several real grammars. The total run-time of ambiguity detection for character-level grammars for languages such as C and Java is significantly reduced, without loss of precision. The result is that efficient ambiguity detection in realistic grammars is possible and may therefore become a tool in language workbenches.

1 Introduction

1.1 Background

Scannerless generalized parsers [7], generated from character-level context-free grammars, serve two particular goals in textual language engineering: parsing legacy languages and parsing language embeddings. We want to parse legacy languages when we construct reverse engineering and reengineering tools to help mitigating cost-of-ownership of legacy source code. The syntax of legacy programming languages frequently does not fit the standard scanner-parser dichotomy. This is witnessed by languages that do not reserve keywords from identifiers (PL/I) or do not always apply "longest match" when selecting a token class (Pascal). For such languages we may generate a scannerless generalized parser that will deal with such idiosyncrasies correctly.

Language embeddings need different lexical syntax for different parts of a composed language. Examples are COBOL with embedded SQL, or Aspect/J with embedded Java. The comment conventions may differ, different sets of identifiers may be reserved as keywords and indeed identifiers may be comprised of different sets of characters, depending on whether the current context is the "host language" or the embedded "guest language". Language embeddings are becoming increasingly popular, possibly due to the belief that one should select the right tool for each job. A character-level grammar can be very convenient to implement a parser for such a combined language [8]. The

reason is that the particular nesting of non-terminals between the host language and the guest language defines where the different lexical syntaxes are applicable. The lexical ambiguity introduced by the language embedding is therefore a non-issue for a scannerless parser. There is no need to program state switches in a scanner [15], to use scanner non-determinism [2], or to use any other kind of (ad-hoc) programming solution.

Using a character-level grammar and a generated scannerless parser results in more declarative BNF grammars which may be maintained more easily than partially hand-written parsers [11]. It is, however, undeniable that character-level grammars are more complex than classical grammars since all lexical aspects of a language have to be specified in full detail. The character-level grammar contains more production rules, which may contain errors or introduce ambiguity. In the absence of lexical disambiguation heuristics, such as "prefer keywords" and "longest match", a character-level grammar may contain many ambiguities that need resolving. Ergo, character-level grammars lead to more declarative grammar specifications but increase the risk of ambiguities and makes automated ambiguity detection harder.

1.2 Contributions and Roadmap

We introduce new techniques for scaling ambiguity detection methods to the complexity that is present in character-level grammars for real programming languages. Our point of departure is a fast ambiguity detection framework that combines a grammar approximation stage with a sentence generation stage [5]. The approximation is used to split a grammar into a set of rules that certainly do not contribute to ambiguity and a set that might. The latter is then fed to a sentence generator to obtain a clear and precise ambiguity report. We sketch this global framework (Section 2) and then describe our baseline algorithm (Section 4). The correctness of this framework has been established in [4] and is not further discussed here.

We present several extensions to the baseline algorithm to make it suitable for character-level grammars (Section 5). First, we consider character classes as shiftable symbols, instead of treating every character as a separate token. This is necessary to deal with the increased lexical complexity of character-level grammars. Second, we make use of disambiguation filters [7] to deal with issues such as keyword reservation and longest match. These filters are used for precision improvements at the approximation stage, and also improve the run-time efficiency of the sentence generation stage by preventing spurious explorations of the grammar. Third, we use grammar unfolding as a general optimization technique (Section 6). This is necessary for certain character-level grammars but is also generally applicable. At a certain cost, it allows us to more effectively identify the parts of a grammar that do not contribute to ambiguity.

We have selected a set of real character-level grammars and measure the speed, footprint and accuracy of the various algorithms (Section 7). The result is that the total cost of ambiguity detection is dramatically reduced for these real grammars.

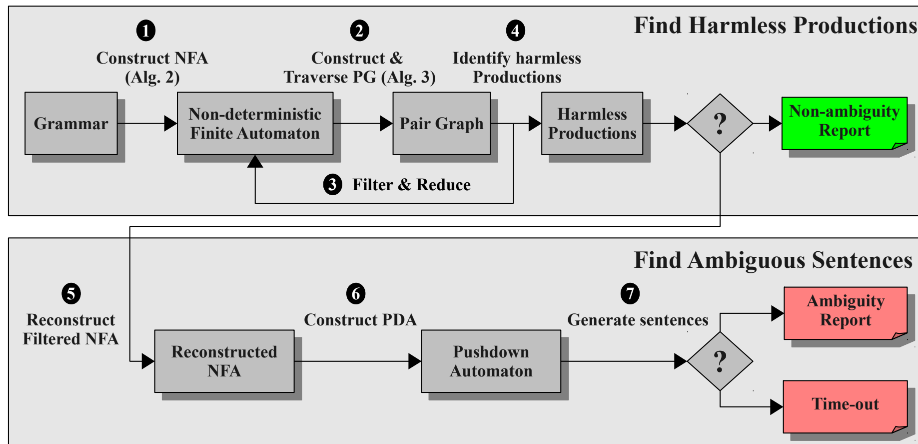

Fig. 1. Baseline architecture for fast ambiguity detection.

2 The Ambiguity Detection Framework

2.1 The Framework

Our starting point is an ambiguity detection framework called AMBIDEXTER [5], which combines an extension of the approximative Noncanonical Unambiguity Test [13] with an exhaustive sentence generator comparable to [14]. The former is used to split a grammar into a set of harmless rules and a set of rules that may contribute to ambiguity. The latter is used to generate derivations based on the potentially ambiguous rules and produce understandable ambiguity reports.

Figure 1 displays the architecture of the baseline algorithm which consists of seven steps, ultimately resulting in a non-ambiguity report, an ambiguity report, or a time-out.

1. In step ❶ the grammar is bracketed, starting and ending each rule with a unique terminal. The language of the bracketed grammar represents all parse trees of the original grammar. In this same step an NFA is constructed that over-approximates the language of the bracketed grammar. This NFA allows us to find strings with multiple parse trees, by approximation, but in finite time.
2. In step ❷ a data-structure called a Pair Graph (PG) is constructed from the NFA. This PG represents all pairs of two different paths through the NFA that produce the same sentence, i.e., potentially ambiguous derivations. During construction, the PG is immediately traversed to identify the part of the NFA that is covered by the potentially ambiguous derivations.
3. In step ❸ we filter the uncovered parts from the NFA and clean up dead ends. This might filter potentially ambiguous derivations from the NFA that are actually false positives, so we reconstruct the PG again to find more uncovered parts. This process is repeated until the NFA cannot be reduced any further.
4. In step ❹ we use the filtered NFA to identify harmless productions. These are the productions that are not used anymore in the NFA. If the NFA is completely filtered then all productions are harmless and the grammar is unambiguous.

5. In step ❺ we prepare the filtered NFA to be used for sentence generation. Due to the removal of states not all paths produce terminal only sentences anymore. We therefore reconstruct the NFA by adding new terminal producing paths.
   In our original approach we generated sentences based on the remaining potentially harmful productions. However, by immediately using the filtered NFA we retain more precision, because the NFA is a more precise description of the potentially ambiguous derivations than a reconstructed grammar.
6. In step ❻ we convert the NFA into a pushdown automaton (PDA) which enables faster sentence generation in the next step.
7. The final step (❼) produces ambiguous strings, including their derivations, to report to the user. This may not terminate, since most context-free grammars generate infinite languages; we need to stop after a certain limited time. All ambiguity that was detected before the time limit is reported to the user.

It was shown in [4] that the calculations employed in this architecture are correct, and in [5] that indeed the efficiency of ambiguity detection can be improved considerably by first filtering harmless productions. However, the baseline algorithm is not suitable for character-level grammars since it is unable to handle their increased complexity and it will still find ambiguities that are already solved. It can even lead to incorrect results because it cannot deal with the non-context-free behaviour of follow restrictions. In this paper we identify several opportunities for optimization and correction:

– We filter nodes and edges in the NFA and PG representations in order to make use of disambiguation information that is found in character-level grammars (Section 5).
– We "unfold" selected parts of a grammar to handle the increased lexical complexity of character-level grammars (Section 6).

For the sake of presentation we have separated the discussion of the baseline algorithm (Section 4), the filtering (Section 5), and the unfolding (Section 6), but it is important to note that these optimizations are not orthogonal.

2.2 Notational Preliminaries

A *context-free grammar* G is a four-tuple (N, T, P, S) where N is the set of non-terminals, T the set of terminals, P the set of productions over $N \times (N \cup T)^*$, and S is the start symbol. V is defined as $N \cup T$. We use $A, B, C, \ldots$ to denote non-terminals, $X, Y, Z, \ldots$ for either terminals or non-terminals, $u, v, w, \ldots$ for sentences: strings of T^*, and $\alpha, \beta, \gamma, \ldots$ for sentential forms: strings over V^*.

A production (A, α) in P is written as $A \to \alpha$. A grammar is augmented by adding an extra non-terminal symbol S', a terminal symbol $\$$ and a production $S' \to S\$$, and making S' the start symbol. We use the function $\mathsf{pid} : P \to \mathbb{N}$ to relate each production to a unique number. An *item* indicates a position in a production rule with a dot, for instance as $S \to A{\bullet}BC$. We use I to denote the set of all items of G.

The relation $\Longrightarrow$ denotes derivation. We say $\alpha B \gamma$ directly derives $\alpha\beta\gamma$, written as $\alpha B \gamma \Longrightarrow \alpha\beta\gamma$ if a production rule $B \to \beta$ exists in P. The symbol $\Longrightarrow^*$ means "derives in zero or more steps". The *language* of G, denoted $\mathcal{L}(G)$, is the set of all

sentences derivable from S. We use $\mathcal{S}(G)$ to denote the *sentential language* of G: the set of all sentential forms derivable from S.

From a grammar G we can create a *bracketed grammar* G_b by surrounding each production rule with unique bracket terminals [9]. The bracketed grammar of G is defined as $G_b = (N, T_b, P_b, S)$ where T_b is the set of terminals and brackets, defined as $T_b = T \cup T_\langle \cup T_\rangle$, $T_\langle = \{\langle_i | i \in \mathbb{N}\}$, $T_\rangle = \{\rangle_i | i \in \mathbb{N}\}$, and $P_b = \{A \to \langle_i \alpha \rangle_i | A \to \alpha \in P,\ i = \mathsf{pid}(A \to \alpha)\}$. V_b is defined as $N \cup T_b$. We use the function bracketP to map a bracket to its corresponding production, and bracketN to map a bracket to its production's left hand side non-terminal. They are defined as $\mathsf{bracketP}(\langle_i) = \mathsf{bracketP}(\rangle_i) = A \to \alpha$ iff $\mathsf{pid}(A \to \alpha) = i$, and $\mathsf{bracketN}(\langle_i) = \mathsf{bracketN}(\rangle_i) = A$ iff $\exists A \to \alpha \in P,\ \mathsf{pid}(A \to \alpha) = i$. A string in the language of G_b describes a parse tree of G. Therefore, if two unique strings exist in $\mathcal{L}(G_b)$ that become identical after removing their brackets, G is ambiguous.

3 Character-Level Grammars

Now we introduce character-level grammars as used for scannerless parsing. Character-level grammars differ from conventional grammars in various ways. They define their syntax all the way down to the character level, without separate token definitions. For convenience, sets of characters are used in the production rules, so-called *character classes*. Regular list constructs can be used to handle repetition, like in EBNF. Also, additional constructs are needed to specify the disambiguation that is normally done by the scanner, so called disambiguation filters [12]. Typical disambiguation filters for character-level grammars are follow restrictions and rejects [7]. Follow restrictions are used to enforce longest match of non-terminals such as identifiers and comments. Rejects are typically used for keyword reservation. Other commonly used disambiguation filters are declarations to specify operator priority and associativity, so these do not have to be encoded manually into the production rules.

3.1 Example

Figure 2 shows an excerpt of a character-level grammar, written in SDF [10,16]. The excerpt describes syntax for C-style variable declarations. A `Declaration` statement consists of a list of `Specifiers` followed by an `Identifier` and a semicolon, separated by whitespace (Rule 1). A `Specifier` is either a predefined type like `int` or `float`, or a user-defined type represented by an `Identifier` (Rule 4). At Rule 5 we see the use of the character class `[a-z]` to specify the syntax of `Identifier`.

The grammar contains both rejects and follow restrictions to disambiguate the lexical syntax. The `{reject}` annotation at Rule 6 declares that reserved keywords of the language cannot be recognized as an `Identifier`. The follow restriction statements at Rules 9–11 declare that any substring that is followed by a character in the range `[a-z]` cannot be recognized as an `Identifier` or keyword. This prevents the situation where a single `Specifier`, for instance an `Identifier` of two or more characters, can also be recognized as a list of multiple shorter `Specifiers`. Basically, the follow restrictions enforce that `Specifiers` should be separated by whitespace.

```
Declaration ::= Specifiers Ws? Identifier Ws? ";"     (1)
Specifiers  ::= Specifiers Ws? Specifier              (2)
Specifiers  ::= Specifier                             (3)
Specifier   ::= Identifier | "int" | "float" | ...    (4)

Identifier  ::= [a-z]+                                (5)
Identifier  ::= Keyword  { reject }                   (6)
Keyword     ::= "int" | "float" | ...                 (7)
Ws          ::= [\ \t\n]+                             (8)

Identifier  -/- [a-z]                                 (9)
"int"       -/- [a-z]                                 (10)
"float"     -/- [a-z]                                 (11)
```

Fig. 2. Example character-level grammar for C-style declarations.

3.2 Definition

We define a *character-level context-free grammar* $G^{\mathcal{C}}$ as the eight-tuple $(N, T, \mathcal{C}, P^{\mathcal{C}}, S, R_D, R_F, R_R)$ where $\mathcal{C} \subset N$ is the set of character classes over $\mathcal{P}(T)$, $P^{\mathcal{C}}$ the set of production rules over $N \times N^*$, R_D the set of derivation restrictions, R_F the set of follow restrictions, R_R the set of rejects.

A *character class* non-terminal is a finite set of terminals in T. For each of its elements it has an implicit production with a single terminal right hand side. We can write $\alpha C \beta \Longrightarrow \alpha c \beta$ iff $C \in \mathcal{C}$ and $c \in C$.

The *derivation restrictions* R_D restrict the application of productions in the context of others. They can be used to express priority and associativity of operators. We define R_D as a relation over $I \times P^{\mathcal{C}}$. Recall that we have defined I as the set of all items of a grammar. An element $(A \to \alpha \bullet B \gamma, B \to \beta)$ in R_D means that we are not allowed to derive a B non-terminal with production $B \to \beta$, if it originated from the B following the dot in the production $A \to \alpha B \gamma$.

The *follow restrictions* R_F restrict the derivation of substrings following a certain non-terminal. We define them as a relation over $N \times T^+$. An element (A, u) in this relation means that during the derivation of a string $\beta A \gamma$, γ can not be derived into a string of form $u\delta$.

The *rejects* R_R restrict the language of a certain non-terminal, by subtracting the language of another non-terminal from it. We define them as a relation over $N \times N$. An element (A, B) means that during the derivation of a string $\alpha A \beta$, A cannot be derived to a string that is also derivable from B.

4 Baseline Algorithm

In this section we explain the baseline algorithm for finding harmless production rules and ambiguous counter-examples. The presentation follows the steps shown in Figure 1. We will mainly focus on the parts that require extensions for character-level grammars, and refer to [4,5] for a complete description of the baseline algorithm. Algorithm 1 gives an overview of the first stage of finding harmless productions. All functions operate on a fixed input grammar $G = (N, T, P, S)$ to which they have global read access.

Algorithm 1 Base algorithm for filtering the NFA and finding harmless productions.

function FIND-HARMLESS-PRODUCTIONS() =
  (Q, R) = BUILD-NFA()
  do
    nfasize = $|Q|$
    Q_a = TRAVERSE-PATH-PAIRS(Q, R) // *returns items used on conflicting path pairs*
    (Q, R) = FILTER-NFA(Q, R, Q_a) // *removes unused items and prunes dead ends*
  while *nfasize* $\neq |Q|$
  return $P \setminus$ USED-PRODUCTIONS(Q)

4.1 Step 1: NFA Construction

The first step of the baseline algorithm is to construct the NFA from the grammar. It is defined by the tuple (Q, R) where Q is the set of states and R is the transition relation over $Q \times V_b \times Q$. Edges in R are denoted by $Q \overset{V_b}{\longmapsto} Q$. The states of the NFA are the items of G. The start state is $S' \to \bullet S\$$ and the end state is $S' \to S\$\bullet$. There are three types of transitions:

- *Shifts* of (non-)terminal symbols to advance to a production's next item,
- *Derives* from items with the dot before a non-terminal to the first item of one of the non-terminal's productions, labeled over $T_{\langle}$,
- *Reduces* from items with the dot at the end, to items with the dot after the non-terminal that is at the first item's production's left hand side, labeled over $T_{\rangle}$.

Algorithm 2 describes the construction of the NFA from G. First, the set of states Q is composed from the items of G. Then the transitions in R are constructed, assuming only items in Q are used. Lines 2–4 respectively build the shift, derive and reduce transitions between the items of G.

Intuitively, the NFA resembles an LR(0) parse automaton before the item closure. The major differences are that also shifts of non-terminals are allowed, and that the NFA has — by definition — no stack. The LR(0) pushdown automaton uses its stack to determine the next reduce action, but in the NFA all possible reductions are allowed. Its language is therefore an overapproximation of the set of parse trees of G. However, the shape of the NFA does allow us to turn it into a pushdown automaton that only generates valid parse trees of G. We will do this later on in the sentence generation stage.

Without a stack the NFA can be searched for ambiguity in finite time. Two paths through it that shift the same sequence of symbols in V, but different bracket symbols in T_b, represent a possible ambiguity. If no conflicting paths can be found then G is

Algorithm 2 Computing the NFA from a grammar.

function BUILD-NFA() =
1 $Q = I$ // *the items of G*
2 $R = \{A \to \alpha \bullet X\beta \overset{X}{\longmapsto} A \to \alpha X \bullet \beta \mid \}$ // *shifts*
3 $\quad \cup \{A \to \alpha \bullet B\gamma \overset{\langle_i}{\longmapsto} B \to \bullet\beta \mid i = \mathsf{pid}(B \to \beta)\}$ // *derives*
4 $\quad \cup \{B \to \beta\bullet \overset{\rangle_i}{\longmapsto} A \to \alpha B \bullet \gamma \mid i = \mathsf{pid}(B \to \beta)\}$ // *reduces*
5 **return** (Q, R)

unambiguous, but otherwise it is uncertain whether or not all conflicting paths represent ambiguous strings in $\mathcal{L}(G)$. However, the conflicting paths can be used to find harmless production rules. These are the rules that are not or incompletely used on these paths. If not all items of a production are used in the overapproximated set of ambiguous parse trees of G, then the production can certainly not be used to create a real ambiguous string in $\mathcal{L}(G)$.

4.2 Step 2: Construct and Traverse Pair Graph

NFA Traversal To collect the items used on all conflicting path pairs we can traverse the NFA with two cursors at the same time. The traversal starts with both cursors at the start item $S' \to \bullet S\$$. From there they can progress through the NFA either independently or synchronized, depending on the type of transition that is being followed. Because we are looking for conflicting paths that represent different parse trees of the same string, the cursors should shift the same symbols in V. To enforce this we only allow synchronized shifts of equal symbols. The derive and reduce transitions are followed asynchronously, because the number of brackets on each path may vary.

During the traversal we wish to avoid the derivation of unambiguous substrings, i.e. an identical sequence of one derive, zero or more shifts, and one reduce on both paths, and prefer non-terminal shifts instead. This enables us to filter more items and edges from the NFA. Identical reduce transitions on both paths are therefore not allowed if no conflicts have occurred yet since their corresponding derives. A path can thus only reduce if the other path can be continued with a different reduce or a shift. This puts the path in conflict with the other. After the paths are conflicting we do allow identical reductions (synchronously), because otherwise it would be impossible to reach the end item $S' \to S\$\bullet$. To register whether a path is in conflict with the other we use boolean flags, one for each path. For a more detailed description of these flags we refer to [13,4].

Algorithm 3 describes the traversal of path pairs through the NFA. It contains gaps ①–④ that we will fill in later on (Algorithm 7), when extending it to handle character-level grammars. To model the state of the cursors during the traversal we use an *item pair* datatype with four fields: two items q_1 and q_2 in Q, and two conflict flags c_1 and c_2 in $\mathbb{B}$. We use Π to denote the set of all possible item pairs.

The function TRAVERSE-EDGES explores all possible continuations from a given item pair. We assume it has access to the global NFA variables Q and R. To traverse each pair graph edge it calls the function TRAVERSE-EDGE — not explained here — which in turn calls TRAVERSE-EDGES again on the next pair. The function SHIFTABLE determines the symbol that can be shifted on both paths. In the baseline setting we can only shift if the next symbols of both paths are identical. Later on we will extend this function to handle character-level grammars. The function CONFLICT determines whether a reduce transition of a certain path leads to a conflict. This is the case if the other path can be continued with a shift or reduce that differs from the first path's reduce.

Pair Graph There can be infinitely many path pairs through the NFA, which can not all be traversed one-by-one. We therefore model all conflicting path pairs with a finite structure, called a *pair graph*, which nodes are item pairs. The function TRAVERSE-EDGES describes the edges of this graph. An infinite amount of path pairs translates to

Algorithm 3 Traversing NFA edge pairs.

function TRAVERSE-EDGES($p \in \Pi$) =
1 **for** each $(p.q_1 \overset{\langle_i}{\longmapsto} q_1') \in R$ **do** // *derive* q_1
2 $\quad p' = p,\ p'.q_1 = q_1',\ p'.c_1 = 0$
3 $\quad$ TRAVERSE-EDGE(p, p')
4 **od**
5 **for** each $(p.q_2 \overset{\langle_i}{\longmapsto} q_2') \in R$ **do** // *derive* q_2
6 $\quad p' = p,\ p'.q_2 = q_2',\ p'.c_2 = 0$
7 $\quad$ TRAVERSE-EDGE(p, p')
8 **od**
9 **for** each $(p.q_1 \overset{X}{\longmapsto} q_1'), (p.q_2 \overset{Y}{\longmapsto} q_2') \in R$
10 **do** // *synchronized shift*
11 $\quad$ **if** SHIFTABLE$(X, Y) \neq \emptyset$ **then**
12 $\qquad p' = p,\ p'.q_1 = q_1',\ p'.q_2 = q_2'$
13 $\qquad$ // . . . ①
14 $\qquad$ TRAVERSE-EDGE(p, p')
15 $\quad$ **fi**
16 **for** each $(p.q_1 \overset{\rangle_i}{\longmapsto} q_1') \in R$ **do**
17 $\quad$ **if** CONFLICT$(p.q_2, \rangle_i)$ **then**
18 $\qquad$ // *conflicting reduction of* q_1
19 $\qquad p' = p,\ p'.q_1 = q_1',\ p'.c_1 = 1$
20 $\qquad$ // . . . ②
21 $\qquad$ TRAVERSE-EDGE(p, p')
22 $\quad$ **fi**
23 **for** each $(p.q_2 \overset{\rangle_i}{\longmapsto} q_2') \in R$ **do**
24 $\quad$ **if** CONFLICT$(p.q_1, \rangle_i)$ **then**
25 $\qquad$ // *conflicting reduction of* q_2
26 $\qquad p' = p,\ p'.q_2 = q_2',\ p'.c_2 = 1$
27 $\qquad$ // . . . ③
28 $\qquad$ TRAVERSE-EDGE(p, p')
29 $\quad$ **fi**
30 **if** $p.c_1 \vee p.c_2$ **then**
31 $\quad$ **for** each $(p.q_1 \overset{\rangle_i}{\longmapsto} q_1'), (p.q_2 \overset{\rangle_i}{\longmapsto} q_2') \in R$
32 $\quad$ **do** // *synchronized reduction*
33 $\qquad p' = p,\ p'.q_1 = q_1',\ p'.q_2 = q_2'$
34 $\qquad p'.c_1 = p'.c_2 = 1$
35 $\qquad$ // . . . ④
36 $\qquad$ TRAVERSE-EDGE(p, p')
37 $\quad$ **od**

function SHIFTABLE($X \in V, Y \in V$) =
1 **if** $X = Y$ **then return** X **else return** $\emptyset$

function CONFLICT($q \in Q,\ \rangle_i \in T_\rangle$) =
1 **return** $\exists q' \in Q, u \in T_\langle^* : (\exists X : q \overset{uX}{\longmapsto}{}^+ q') \vee (\exists \rangle_j \neq \rangle_i : q \overset{u\rangle_j}{\longmapsto}{}^+ q')$

cycles in this finite pair graph. To find the items used on all conflicting paths it suffices to do a depth first traversal of the pair graph that visits each edge pair only once.

4.3 Steps 3–4: NFA Filtering and Harmless Rules Identification

After the items used on conflicting path pairs are collected we can identify harmless production rules from them. As said, these are the productions of which not all items are used. All other productions of G are potentially harmful, because it is uncertain if they can really be used to derive ambiguous strings.

We filter the harmless production rules from the NFA by removing all their items and pruning dead ends. If there are productions of which some but not all items were used, we actually remove a number of conflicting paths that do not represent valid parse trees of G. After filtering there might thus be even more unused items in the NFA. We therefore repeat the traversing and filtering process until no more items can be removed. Then, all productions that are not used in the NFA are harmless. This step concludes the first stage of our framework (*Find Harmless Productions* in Figure 1).

4.4 Steps 5–7: NFA Reconstruction and Sentence Generation

In the second part of our framework we use an inverted SGLR parser [7] as a sentence generator to find real ambiguous sentences in the remainder of the NFA. However, certain states in the NFA might not lead to the generation of terminal-only sentences anymore, due to the removal of terminal shift transitions during filtering. These are the states with outgoing non-terminal shift transitions that have no corresponding derive and reduce transitions anymore. To make such a non-terminal productive again we introduce a new terminal-only production for it that produces a shortest string from its original language. Then we add a new chain of derive, shift, and reduce transitions for this production to the states before and after the unproductive non-terminal shift.

After the NFA is reconstructed we generate an LR(0) pushdown automaton from it to generate sentences with. In contrast to the first stage, we now do need a stack because we only want to generate proper derivations of the grammar. Also, because of the item closure that is applied in LR automata, all derivations are unfolded statically, which saves generation steps at run-time.

The inverted parser generates all sentences of the grammar, together with their parse trees. If it finds a sentence with multiple trees then these are reported to the user. They are the most precise ambiguity reports possible, and are also very descriptive because they show the productions involved [3]. Because the number of derivations of a grammar can be infinite, we continue searching strings of increasing length until a certain time limit is reached. The number of strings to generate can grow exponentially with increasing length, but filtering unambiguous derivations beforehand can also greatly reduce the time needed to reach a certain length as Section 7 will show.

5 Ambiguity Detection for Character-level Grammars

After sketching the baseline algorithm we can extend it to find ambiguities in character-level grammars. We take disambiguation filters into account during ambiguity detection, so we do not report ambiguities that are already solved by the grammar developer. Furthermore, we explain and fix the issue that the baseline harmless rules filtering is unable to properly deal with follow restrictions.

5.1 Application of Baseline Algorithm on Example Grammar

Before explaining our extensions we first show that the baseline algorithm can lead to incorrect results on character-level grammars. If we apply it to the example grammar of Figure 2, the harmless production rule filter will actually remove ambiguities from the grammar. Since the filtering is supposed to be conservative, this behaviour is incorrect.

The baseline algorithm will ignore the reject rule and follow restrictions in the grammar (Rules 6, 7, 9–11), and will therefore find the ambiguities that these filters meant to solve. Ambiguous strings are, among others, "`float f;`" (`float` can be a keyword or identifier) and "`intList l;`" (`intList` can be one or more specifiers). Rules 1–5 will therefore be recognized as potentially harmful. However, in all ambiguous strings, the substrings containing whitespace will always be unambiguous. This is detected by the PG traversal and Rule 8 (`Ws ::= [\ \t\n]+`) will therefore become harmless.

Rule 8 will be filtered from the grammar, and during reconstruction `Ws?` will be terminalized with the shortest string from its language, in this case ε. This effectively removes all whitespace from the language of the grammar. In the baseline setting the grammar would still be ambiguous after this, but in the character-level setting the language of the grammar would now be empty! The follow restriction of line 9 namely dictates that valid `Declaration` strings should contain at least one whitespace character to separate specifiers and identifiers.

This shows that our baseline grammar filtering algorithm is not suitable for character-level grammars as is, because it might remove ambiguous sentences. In addition, it might even introduce ambiguities in certain situations. This can happen when non-terminals are removed that have follow restrictions that prohibit a second derivation of a certain string. In short, follow restrictions have a non-context-free influence on sentence derivation, and the baseline algorithm assumes only context-free derivation steps. In the extensions presented in the next section we repair this flaw and make sure that the resulting algorithm does not introduce or lose ambiguous sentences.

5.2 Changes to the Baseline Algorithm

The differences between character-level grammars and conventional grammars result in several modifications of our baseline algorithm. These modifications deal with the definitions of both the NFA and the pair graph. We reuse the NFA construction of Algorithm 2 because it is compliant with character-level productions, and apply several modifications to the NFA afterwards to make it respect a grammar's derivation restrictions and follow restrictions. An advantage of this is that we do not have to modify the pair graph construction. To keep the test practical and conservative we have to make sure that the NFA remains finite, while its paths describe an overapproximation of $\mathcal{S}(G_b)$.

Character Classes Because of the new shape of the productions, we now shift entire character classes at once, instead of individual terminal symbols. This avoids adding derives, shifts and reduces for the terminals in all character classes, which would bloat the NFA, and thus also the pair graph. In the PG we allow a synchronized shift of two character classes if their intersection is non-empty. To enforce this behaviour we only need to change the SHIFTABLE function as shown in Algorithm 4.

Derivation Restrictions and Follow Restrictions After the initial NFA is constructed we remove derive and reduce edges that are disallowed by the derivation restrictions. This is described in function FILTER-DERIVE-RESTRICTIONS in Algorithm 5. Then we propagate the follow restrictions through the NFA to make it only generate

Algorithm 4 SHIFTABLE function for character-level pair graph.

function SHIFTABLE($X \in N, Y \in N$) =
  // returns the symbol that can be shifted from X and Y
  if $X \in \mathcal{C} \wedge Y \in \mathcal{C}$ **then return** $X \cap Y$ *// X and Y are character classes*
  else if $X = Y$ **then return** X *// X and Y are the same non-terminal*
  else return $\emptyset$ *// no shift possible*

Algorithm 5 Filtering derive restrictions from the NFA.

function FILTER-DERIVE-RESTRICTIONS(R) =

return $R \setminus \{A \to \alpha{\bullet}B\gamma \xmapsto{\langle_i} B \to {\bullet}\beta \mid i = \mathsf{pid}(B \to \beta),\ (A \to \alpha{\bullet}B\gamma, B \to \beta) \in R_D\}$
$\setminus \{B \to \beta{\bullet} \xmapsto{\rangle_i} A \to \alpha B{\bullet}\gamma \mid i = \mathsf{pid}(B \to \beta),\ (A \to \alpha{\bullet}B\gamma, B \to \beta) \in R_D\}$

strings that comply with them. This is described in function PROPAGATE-FOLLOW-RESTRICTIONS in Algorithm 6. The operation will result in a new NFA with states that are tuples containing a state of the original NFA and a set of follow restrictions over $\mathcal{P}(T^+)$. A new state cannot be followed by strings that have a prefix in the state's follow restrictions. To enforce this we constrain character class shift edges according to the follow restrictions of their source states.

The process starts at $(S' \to {\bullet}S\$,\ \emptyset)$ and creates new states while propagating a state's follow restrictions over the edges of its old NFA item. In contrast to the original NFA, which had at most one shift edge per state, states in the new NFA can have multiple. This is because non-terminal or character class edges actually represent the shift of multiple sentences, which can each result in different follow restrictions. Lines 6–9 show the reconstruction of character-class shift edges from a state $(A \to \alpha{\bullet}B\beta, f)$. Shift edges are added for characters in B that are allowed by f. All characters in B that will result in the same new set of follow restrictions are combined into a single shift edge, to not bloat the new NFA unneccesarily. The restrictions after a shift of a are the tails of the strings in f beginning with a, and are calculated by the function NEXT-FOLLOW.

Line 12 describes how a state's restrictions are passed on unchanged over derive edges. Lines 13–20 show how new non-terminal shift edges are added from a state $(A \to \alpha{\bullet}B\beta, f)$ once their corresponding reduce edges are known. This is convenient because we can let the propagation calculate the different follow restrictions that can reach $A \to \alpha B{\bullet}\beta$. Once the restrictions that were passed to the derive have reached a state $B \to \gamma{\bullet}$, we propagate them upwards again over a reduce edge to $A \to \alpha B{\bullet}\beta$. If B has follow restrictions — in R_F — these are added to the new state as well. Note that multiple follow restriction sets might occur at the end of a production, so we might have to reduce a production multiple times. For a given state $B \to {\bullet}\gamma$, the function SHIFT-ENDS returns all states that are at $B \to \gamma{\bullet}$ and that are reachable by shifting.

If the reduced production is of form $B \to \varepsilon$ we create a special non-terminal symbol B^ε and make it the label of the shift edge instead of B. This is a small precision improvement of the PG traversal. It prevents the situation where a specific non-terminal shift that —because of its follow restriction context— only represents the empty string, is traversed together with another instance of the same non-terminal that cannot derive ε.

The propagation ends when no new edges can be added to the new NFA. In theory the new NFA can now be exponentially larger than the original, but since follow restrictions are usually only used sparingly in the definition of lexical syntax this will hardly happen in practice. In Section 7 we will see average increases in NFA size of a factor 2–3.

Rejects Instead of encoding a grammar's rejects in the NFA, we choose to handle them during the PG traversal. Consider an element (A, B) in R_R, which effectively subtracts the language of B from that of A. If the language of B is regular then we could, for instance, subtract it from the NFA part that overapproximates the language of A. This

Algorithm 6 Propagating follow restrictions through the NFA.

function PROPAGATE-FOLLOW-RESTRICTIONS(Q, R) =
// propagate follow restrictions through NFA (Q, R) and return a new NFA (Q', R')

1 $Q' = \{(S' \to \bullet S\$, \emptyset)\},\ R' = \emptyset$
2 **repeat**
3   add all states used in R' to Q'
4   **for** $q_f = (A \to \alpha \bullet B \beta, f) \in Q'$ **do**
5   **if** $B \in \mathcal{C}$ **then** *// B is a character class*
6   **for** $a \in B,\ a \notin f$ **do** *// all shiftable characters in C*
7   **let** $B' = \{b \mid b \in B, b \notin f, \text{NEXT-FOLLOW}(a, f) = \text{NEXT-FOLLOW}(b, f)\}$
8   add $q_f \overset{B'}{\longmapsto} (A \to \alpha B \bullet \beta, \text{NEXT-FOLLOW}(a, f))$ to R'
9   **od**
10   **else** *// B is a normal non-terminal*
11   **for** $A \to \alpha \bullet B \beta \overset{\langle_i}{\longmapsto} q' \in R$ **do**
12   add $q_f \overset{\langle_i}{\longmapsto} (q', f)$ to R' *// propagate f over derivation*
13   **for** $q_f^r = (q^r, f^r) \in \text{SHIFT-ENDS}((q', f))$
14   **let** $q_f^s = (A \to \alpha B \bullet \beta, f^r \cup R_F(B))$ *// shift target*
15   add $q_f^r \overset{\rangle_i}{\longmapsto} q_f^s$ to R' *// reduction to shift target*
16   **if** $\mathsf{bracketP}(\langle_i) = B \to \varepsilon$ **then**
17   add $q_f \overset{B^\varepsilon}{\longmapsto} q_f^s$ to R' *// non-terminal shift representing empty string*
18   **else**
19   add $q_f \overset{B}{\longmapsto} q_f^s$ to R' *// non-terminal shift of non-empty strings*
20   **od**
21   **od**
22 **until** no more edges can be added to R'
23 **return** (Q', R')

function SHIFT-ENDS$((A \to \bullet\alpha, f) \in Q')$ =
1 *// return the states at the end of $A \to \alpha$, reachable from q using only shifts*
2 **let** $\dashrightarrow = \{q \dashrightarrow q' \mid q \overset{B}{\longmapsto} q' \in R'\}$ *// the shift transitions of R'*
3 **return** $\{(A \to \alpha \bullet, f') \mid (A \to \bullet\alpha, f) \dashrightarrow^* (A \to \alpha\bullet, f')\}$

function NEXT-FOLLOW$(a \in T,\ f \in \mathcal{P}(T^+))$
1 **return** $\{\alpha \mid a\alpha \in f,\ \alpha \neq \varepsilon\}$ *// the next follow restrictions of f after a shift of a*

would not violate the finiteness and overapproximation requirements. However, if the language of B is context-free we have to *under*approximate it to finite form first, to keep the NFA an overapproximation and finite. A possible representation for this would be a second NFA, which we could subtract from the first NFA beforehand, or traverse alongside the first NFA in the PG.

Instead, we present a simpler approach that works well for the main use of rejects: keyword reservation. We make use of the fact that keywords are usually specified as a set of non-terminals that represent literal strings — like Rules 6 and 7 in Figure 2. The production rules for `"int"`, `"float"`, etc. are not affected by the approximation, and appear in the NFA in their original form. We can thus recognize that, during the PG traversal, a path has completely shifted a reserved keyword if it reduces `"int"`. After that,

Algorithm 7 Extensions to TRAVERSE-EDGES for avoiding rejected keywords.

// at ① (shift) insert:
  $p'.r_1 = p'.r_2 = \emptyset$ *// clear reduced sets*

// at ② and ④ (conflicting and pairwise reduce) insert:
  if not CHK-REJECT($\rangle_i, p.r_2$) **then continue**
  $p'.r_1$ = NEXT-REJECT($\rangle_i, p.r_1$)

// similarly, insert at ③ and ④:
  if not CHK-REJECT($\rangle_i, p.r_1$) **then continue**
  $p'.r_2$ = NEXT-REJECT($\rangle_i, p.r_2$)

function CHK-REJECT($\rangle_i \in T_\rangle, r \in \mathcal{P}(N)$) =
  // returns whether a reduction with $\rangle_i$ is possible after reductions r on other path
  let $A = \mathsf{bracketN}(\rangle_i)$
  return $\neg\exists\, B \in r : (A, B) \in R_R \vee (B, A) \in R_R$

function NEXT-REJECT($\rangle_i \in T_\rangle, r \in \mathcal{P}(N)$) =
  // adds non-terminal reduced with $\rangle_i$ to r if it is involved in a reject
  let $A = \mathsf{bracketN}(\rangle_i)$
  if $\exists B \in r : (A, B) \in R_R \vee (B, A) \in R_R$ **then**
    return $r \cup \{A\}$
  else return r

we can prevent the other path from reducing `Identifier` before the next shift. This does not restrict the language of `Identifier` in the NFA — it is kept overapproximated —, but it does prevent the ambiguous situation where “`int`” is recognized as an `Identifier` on one path and as an `"int"` on the other path.

Of course, `Identifier` could also be reduced before `"int"`, so we need to register the reductions of both non-terminals. During the PG traversal, we keep track of all reduced non-terminals that appear in R_R, in two sets r_1 and r_2, one for each path. Then, if a path reduces a non-terminal that appears in a pair in R_R, together with a previously reduced non-terminal in the other path's set, we prevent this reduction. The sets are cleared again after each pairwise shift transition. Algorithm 7 shows this PG extension.

5.3 NFA Reconstruction

In Section 5.1 we saw that follow restrictions should be handled with care when filtering and reconstructing a grammar, because of their non-context-free behaviour. By removing productions from a grammar certain follow restrictions can become unavoidable, which removes sentences from the language. On the other hand, by removing follow restrictions new sentences can be introduced that were previously restricted. When reconstructing a character-level grammar we thus need to terminalize filtered productions depending on the possible follow-restrictions they might generate or that might apply to them.

Instead, by directly reusing the filtered NFA for sentence generation, we can avoid this problem. The follow restrictions that are propagated over the states already describe the follow restriction context of each item. For each distinct restriction context of an

item a separate state exists. We can just terminalize each unproductive non-terminal shift edge with an arbitrary string from the language of its previously underlying automaton.

Furthermore, the filtered NFA is a more detailed description of the potentially ambiguous derivations than a filtered grammar, and therefore describes less sentences. For instance, if derive and reduce edges of a production $B \to \beta$ are filtered out at a specific item $A \to \alpha \bullet B\gamma$, but not at other items, we know $B \to \beta$ is harmless in the context of $A \to \alpha \bullet B\gamma$. The propagated follow restrictions also provide contexts in which certain productions can be harmless. We could encode this information in a reconstructed grammar by duplicating non-terminals and productions of course, but this could really bloat the grammar. Instead, we just reuse the baseline NFA reconstruction algorithm.

6 Grammar Unfolding

In Section 7 we will see that the precision of the algorithm described above is not always sufficient for some real life grammars. The reason for this is that the overapproximation in the NFA is too aggressive for character-level grammars. By applying grammar unfoldings we can limit the approximation, which improves the precision of our algorithm.

The problem with the overapproximation is that it becomes too aggressive when certain non-terminals are used very frequently. Remember that due to the absence of a stack, the derive and reduce transitions do not have to be followed in a balanced way. Therefore, after deriving from an item $A \to \alpha \bullet B\beta$ and shifting a string in the language of B, the NFA allows reductions to any item of form $C \to \gamma B \bullet \delta$. This way, a path can jump to another production while being in the middle of a first production. Of course, a little overapproximation is intended, but the precision can be affected seriously if certain non-terminals are used very frequently. Typical non-terminals like that in character-level grammars are those for whitespace and comments, which can appear in between almost all language constructs. Since these non-terminals can usually derive to ε, we can thus jump from almost any item to almost any other item by deriving and reducing them.

To restrict the overapproximation we can unfold the frequently used non-terminals in the grammar, with a technique similar to one used in [6]. A non-terminal is unfolded by creating a unique copy of it for every place that it occurs in the right-hand sides of the production rules. For each of these copies we then also duplicate the entire sub-grammar of the non-terminal. The NFA thus gets a separate isolated sub-automaton for each occurence of an unfolded non-terminal. After the derivation from an item $A \to \alpha \bullet B\beta$ a path can now only reduce back to $A \to \alpha B \bullet \beta$, considering B is unfolded. After unfolding, the NFA contains more states, but has less paths through it because it is more deterministic. In the current implementation we unfold all non-terminals that describe whitespace, comments, or literal strings like keywords, brackets and operators. Later on we will refer to this unfolding extension as CHAR+UNF.

7 Experimental Results

We have evaluated our ambiguity detection algorithm for character-level grammars on the grammar collection shown in Table 1. All grammars are specified in SDF [10,16]. The selection of this set is important for external validity. We have opted for grammars

Table 1. Character-level grammars used for validation.

Name	Prods.	SLOC	Non-terms.	Derive rest.	Follow restr.	Reserved keywords
C[1]	324	415	168	332	10	32
C++[2]	807	4172[a]	430	1	87	74
ECMAScript[3]	403	522	232	1	27	25
Oberon0[4]	189	202	120	132	31	27
SQL-92[5]	419	495	266	23	5	30
Java 1.5[6]	698	1629	387	297	78	56

[1] SDF grammar library, revision 27501, `http://www.meta-environment.org`

[2] TRANSFORMERS 0.4, `http://www.lrde.epita.fr/cgi-bin/twiki/view/Transformers/Transformers`

[3] ECMASCRIPT-FRONT, revision 200, `http://strategoxt.org/Stratego/EcmaScriptFront`

[4] RASCAL Oberon0 project (converted to SDF), rev. 34580, `http://svn.rascal-mpl.org/oberon0/`

[5] SQL-FRONT, revision 20713, `http://strategoxt.org/Stratego/SqlFront`

[6] JAVA-FRONT, revision 17503, `http://strategoxt.org/Stratego/JavaFront`

[a] After removal of additional attribute code

of general purpose programming languages, which makes it easier for others to validate our results. For each grammar we give its name, number of productions, number of source lines (SLOC), number of non-terminals, number of priorities and associativities (derivation restrictions), number of follow restrictions and number of reserved keywords.

7.1 Experiment Setup

We have run both our NFA filtering and sentence generation algorithms on each of these grammars. Most measurements were carried out on an Intel Core2 Quad Q6600 2.40GHz with 8GB DDR2 memory, running Fedora 14. A few memory intensive runs were done on an Amazon computing cloud EC2 High-Memory Extra Large Instance with 17.1GB memory. The algorithms have been implemented in Java and are available for download at `http://homepages.cwi.nl/~basten/ambiguity`. In order to identify the effects of the various extensions, we present our empirical findings for the following combinations:

- **BASE**: the baseline algorithm for token-level grammars as described in Section 4, with the only addition that whole character-classes are shifted instead of individual tokens. Even though this configuration can lead to incorrect results, it is included as a baseline for comparison.
- **CHAR**: the baseline algorithm extended for handling character-level grammars as described in Section 5, including extensions for follow restrictions, derive restrictions and rejects.
- **CHAR+UNF**: the **CHAR** algorithm combined with grammar unfolding (Section 6).

7.2 Results and Analysis

In Table 2 we summarize our measurements of the NFA filtering and harmless production rule detection. For each grammar and extension configuration we give the number of harmless productions found versus total number of productions, number of edges filtered from the NFA, execution time (in seconds) and memory usage (in MB).

Table 2. Timing and precision results of filtering harmless productions.

Grammar	Method	Harmless productions	NFA edges filtered	Time (sec)	Memory (MB)
C	BASE	48 / 324	343 / 14359	64	2128
	CHAR	62 / 324	2283 / 24565	120	3345
	CHAR+UNF	75 / 324	8637 / 30653	97	2616
C++	BASE	0 / 807	0 / 8644	32	1408
	CHAR	0 / 807	0 / 39339	527	7189
	CHAR+UNF[a]	–	–	>9594	>17.3G
ECMAScript	BASE	44 / 403	414 / 4872	12	547
	CHAR	46 / 403	1183 / 10240	46	1388
	CHAR+UNF	88 / 403	9887 / 19890	31	1127
Oberon0	BASE	0 / 189	0 / 3701	4.2	256
	CHAR	70 / 189	925 / 6162	9.0	349
	CHAR+UNF	73 / 189	10837 / 20531	14	631
SQL-92	BASE	13 / 419	98 / 4944	16	709
	CHAR	20 / 419	239 / 9031	83	2093
	CHAR+UNF	65 / 419	7285 / 14862	37	1371
Java 1.5	BASE	0 / 698	0 / 16844	60	2942
	CHAR	0 / 698	0 / 45578	407	7382
	CHAR+UNF[a]	189 / 698	180456 / 262030	1681	15568

[a] Run on Amazon EC2 High-Memory Extra Large Instance

Every configuration was able to filter an increasing number of productions and edges for each of the grammars. For C and ECMAScript **BASE** could already filter a small number rules and edges, although it remains unsure whether these are all harmless because the baseline algorithm cannot handle follow restrictions properly. For C and Oberon0 our character-level extensions of **CHAR** improved substantially upon **BASE**, without the risk of missing ambiguous sentences.

Of all three configurations **CHAR+UNF** was the most precise. For the grammar order of the table, it filtered respectively 23%, 0%, 22%, 39%, 16% and 27% of the production rules, and 28%, 0%, 50%, 53%, 49% and 69% of the NFA edges. Unfolding grammars leads to larger but more deterministic NFAs, which in turn can lead to smaller pair graphs and thus faster traversal times. This was the case for most grammars except the larger ones. ECMAScript, SQL-92 and Oberon0 were checkable in under 1 minute, and C in under 2 minutes, all requiring less than 3GB of memory. Java 1.5 was checkable in just under 16GB in 30 minutes, but for the C++ grammar — which is highly ambiguous — the pair graph became too large. However, the additional cost of unfolding was apparently necessary to deal with the complexity of Java 1.5.

Table 3 allows us to compare the sentence generation times for the unfiltered and filtered NFAs. For each grammar and NFA it shows the number of sentences of a certain length in the language of the NFA, and the times required to search them for ambiguities. The unfiltered sentence generation also takes disambiguation filters into account. C++ is not included because its NFA could not be filtered in the previous experiments.

For all grammars we see that filtering with **CHAR** and **CHAR+UNF** lead to reductions in search space and generation times. To indicate whether the investments in filtering time actually pay off, the last column contains the maximum speedup gained by

Table 3. Timing results of sentence generation. Times are in seconds. For each sentence length, the run-time of the fastest configuration (also taking filtering time into account) is highlighted. Speedup is calculated as $\frac{\text{unfiltered sentence gen. time}}{\text{filtering time} + \text{sentence gen. time}}$.

Grammar	Len	Ambig NTs	Unfiltered Sentences	Unfiltered Time	CHAR Sentences	CHAR Time	CHAR+UNF Sentences	CHAR+UNF Time	Maximum speedup
C	5	6	345K	**7.9**	273K	5.9	267K	5.9	0.08x
	6	8	5.06M	**35**	3.77M	25	3.66M	25	0.29x
	7	8	75.5M	398	53.4M	270	51.6M	**259**	1.1x
	8	9	1.13G	5442	756M	3466	727M	**3362**	1.6x
	9	10	17.0G	78987	10.8G	47833	10.3G	**47018**	1.7x
ECMAScript	3	6	14.2K	**4.5**	11.7K	3.5	9.29K	3.3	0.13x
	4	8	274K	**11**	217K	8.9	159K	6.7	0.29x
	5	10	5.17M	149	3.92M	120	2.64M	**69**	1.5x
	6	11	96.8M	2805	70.5M	2186	43.8M	**1184**	2.3x
	7	12	1.80G	54175	1.26G	41091	719M	**20264**	2.7x
Oberon0	22	0	21.7M	60	320	**1.0**	182	1.0	6.0x
	23	0	62.7M	186	571	**1.0**	248	1.0	19x
	24	0	247M	815	1269	**1.0**	468	1.0	82x
	25	0	1.39G	4951	3173	**1.1**	1343	1.1	490x
	26	0	9.56G	35007	9807	**1.3**	3985	1.3	3399x
	32	0			108M	172	13.8M	**28**	
	33	0			549M	885	55.6M	**101**	
	34	0			2.80G	4524	224M	**393**	
	35	0			14.3G	22530	906M	**1591**	
	36	0					3.66G	**6270**	
SQL-92	11	5	2.65M	**16**	1.54M	9.4	321K	4.2	0.39x
	12	6	15.8M	102	7.36M	47	1.66M	**14**	2.0x
	13	6	139M	1018	51.3M	379	11.5M	**90**	8.0x
	14	6	1.49G	11369	453M	3572	90.8M	**711**	15x
	15	7			4.39G	35024	742M	**5781**	
	16	8					6.13G	**47211**	
Java 1.5	7	0	187K	**33**			39.1K	6.8	0.02x
	8	1	3.15M	**115**			482K	20	0.07x
	9	1	54.7M	**1727**			6.05M	212	0.91x
	10	1	959M	39965			76.2M	**4745**	6.2x

either **CHAR** or **CHAR+UNF**. For sentence lengths that are already relatively cheap to generate, filtering beforehand has no added value. However, the longer the sentences get the greater the pay-off. We witnessed speedup factors ranging from a small 1.1 (C length 7) to a highly significant 3399 (Oberon0 length 26). Filtering Oberon0 with **CHAR+UNF** was so effective that it increased the sentence length checkable in around 15 minutes from 24 to 35.

For most grammars filtering already becomes beneficial after around 15 seconds to 6 minutes. For Java 1.5 this boundary lies around 35 minutes, because of its high filtering time. However, after that we see an immediate speedup of a factor 6.2. In all cases **CHAR+UNF** was superior to **CHAR**, due to its higher precision and lower run-times.

The third column of Table 3 contains the number of ambiguous non-terminals found at each length. Because of filtering, ambiguous non-terminals at larger lengths were

found earlier in multiple grammars. There were 2 ambiguous non-terminals in C that were found faster, and 4 non-terminals in ECMAScript and 3 in SQL-92.

Concluding, we see that our character-level NFA filtering approach was very beneficial on the small to medium grammars. A relatively low investment in filtering time — under 2 minutes — lead to significant speedups in sentence generation. This enabled the earlier detection of ambiguities in these grammars. For the larger Java 1.5 grammar the filtering became beneficial only after 32 minutes, and for the highly ambiguous C++ grammar the filtering had no effect at all. Nevertheless, ambiguity detection for character-level grammars is ready to be used in interactive language workbenches.

7.3 Validation

In [4] we proved the correctness of our baseline algorithm. To further validate our character-level extensions and their implementations we applied them on a series of toy grammars and grammars of real world programming languages. We ran various combinations of our algorithms on the grammars and automatically compared the ambiguous sentences produced, to make sure that only those ambiguities that exist in a grammar were found, so not more and not less. For the version of our implementation that we used for the experiments above, we found no differences in the ambiguous strings generated. The validation was done in the following stages:

- First we built confidence in our baseline sentence generator by comparing it to the external sentence generators AMBER [14] and CFGANALYZER [1]. For this we used a grammar collection also used in [3], which contains 87 small toy grammars and 25 large grammars of real-world programming languages.
- Then we validated the character-level extension of the baseline sentence generator by comparing it to a combination of our baseline sentence generator and the SGLR [7] parser used for SDF. By running the baseline sentence generator on character-level grammars it will report more strings as ambiguous than actually exist in a grammar, because it does not regard disambiguation filters. We therefore filter out the truly ambiguous sentences by using the SGLR parser as an oracle, and test whether our character-level sentence generator finds exactly the same ambiguous sentences. In some situations SGLR will produce non-optimal parse trees, so we had to verify these by hand. In this step and the following we used the SDF grammars in Table 1.
- Third, we validated our NFA filtering algorithms by running the character-level sentence generator on both filtered and unfiltered NFAs. Because a filtered NFA contains only one reconstructed sentence for non-terminals with only harmless productions, it might produce less variations of ambiguous sentences. We therefore reduced all ambiguous sentences to their *core* ambiguous sentential forms [4] before comparison. This is done by removing the unambiguous substrings from an ambiguous sentence, and replacing them with their deriving non-terminal.

8 Conclusion

We have presented new algorithms for ambiguity detection for character-level grammars and by experimental validation we have found an affirmative answer to the question

whether ambiguity detection can be scaled to this kind of grammars. We have achieved significant speedups of up to three orders of magnitude for ambiguity checking of real programming language grammars. Ambiguity detection for character-level grammars is ready to be used in interactive language workbenches, which is good news for the main application areas of these grammars: software renovation, language embedding and domain-specific languages.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Proceedings of the 35th International Colloquium on Automata, Languages, and Programming (ICALP'08). LNCS, vol. 5126 (2008)
2. Aycock, J., Horspool, R.N.: Schrödinger's token. Software: Practice & Experience 31(8), 803–814 (2001)
3. Basten, H.J.S.: The usability of ambiguity detection methods for context-free grammars. In: Johnstone, A., Vinju, J.J. (eds.) Proceedings of the Eigth Workshop on Language Descriptions, Tools and Applications (LDTA 2008). ENTCS, vol. 238 (2009)
4. Basten, H.J.S.: Tracking down the origins of ambiguity in context-free grammars. In: Cavalcanti, A., Deharbe, D., Gaudel, M.C., Woodcock, J. (eds.) Theoretical Aspects of Computing – ICTAC 2010, LNCS, vol. 6255, pp. 76–90. Springer Berlin / Heidelberg (2010)
5. Basten, H.J.S., Vinju, J.J.: Faster ambiguity detection by grammar filtering. In: Brabrand, C., Moreau, P.E. (eds.) Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010). ACM (2010)
6. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Sci. Comput. Program. 75(3), 176–191 (2010)
7. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction. pp. 143–158. Springer-Verlag (2002)
8. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Vlissides, J.M., Schmidt, D.C. (eds.) OOPSLA. pp. 365–383. ACM (2004)
9. Ginsburg, S., Harrison, M.A.: Bracketed context-free languages. Journal of Computer and System Sciences 1(1), 1–23 (1967)
10. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Notices 24(11), 43–75 (1989)
11. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) OOPSLA. pp. 918–932. ACM (2010)
12. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In: Proceedings of the ASMICS Workshop on Parsing Theory. pp. 1–20. Technical Report 126-1994, Università di Milano (1994)
13. Schmitz, S.: Conservative ambiguity detection in context-free grammars. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP'07: 34th International Colloquium on Automata, Languages and Programming. LNCS, vol. 4596 (2007)
14. Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), see `http://accent.compilertools.net/Amber.html`
15. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Consel, C., Lawall, J.L. (eds.) GPCE. pp. 63–72. ACM (2007)
16. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam (September 1997)