Ambiguity detection for programming language grammars
Basten, H.J.S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
In this chapter we present a set of extensions to our grammar filtering technique to make it suitable for character-level grammars. Character-level grammars are used for generating scannerless parsers, and are typically more complex than token-based grammars. We present several character-level specific extensions that take disambiguation filters into account, but also a general precision improving extension called grammar unfolding. We test an implementation of our approach on a series of real world programming languages and measure the improvements in sentence generation times. Although the gains are not as large as in Chapter 3, our technique proves very useful on most grammars.

5.1 Introduction

5.1.1 Background

Scannerless generalized parsers [vdBSVV02], 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 [BV04]. 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 [VS07], to use scanner non-determinism [AH01], 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 [KVW10]. 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.

5.1.2 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 — see Chapter 3. 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 5.2) and then describe our baseline algorithm (Section 5.4). The correctness of this framework has been established in Chapter 4 and is not further discussed here.

We present several extensions to the baseline algorithm to make it suitable for character-level grammars (Section 5.5). 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 [vdBSVV02] 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 5.6). 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 5.7). The result is that the total cost of ambiguity detection is dramatically reduced for these real grammars.

### 5.2 The Ambiguity Detection Framework

#### 5.2.1 The Framework

Our starting point is an ambiguity detection framework called AMBIDEXTER, which combines an extension of the approximative Noncanonical Unambiguity Test [Sch07b] with an exhaustive sentence generator comparable to [Sch01]. 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 5.1 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 1 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 2 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 3 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 4 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 5 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 6 we convert the NFA into a pushdown automaton (PDA) which enables faster sentence generation in the next step.

7. The final step 7 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 Chapter 4 that the calculations employed in this architecture are correct, and in Chapter 3 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 chapter 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 5.5).
   - We “unfold” selected parts of a grammar to handle the increased lexical complexity of character-level grammars (Section 5.6).

   For the sake of presentation we have separated the discussion of the baseline algorithm (Section 5.4), the filtering (Section 5.5), and the unfolding (Section 5.6), but it is important to note that these optimizations are not orthogonal.
5.3. Character-Level Grammars

5.2.2 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 $(A, \alpha)$ 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 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 $S \to A \cdot BC$. We use $I$ to denote the set of all items of $G$.

The relation $\Rightarrow$ denotes derivation. We say $\alpha B \gamma$ directly derives $\alpha \beta \gamma$, written as $\alpha B \gamma \Rightarrow \alpha \beta \gamma$ if a production rule $B \to \beta$ exists in $P$. The symbol $\Rightarrow^*$ means “derives in zero or more steps”. The language of $G$, denoted $L(G)$, is the set of all sentences derivable from $S$. We use $S(G)$ 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 [GH67]. 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 $T_b = T \cup T_i \cup T_x$, $T_i = \{_i | i \in \mathbb{N}\}$, $T_x = \{i | i \in \mathbb{N}\}$, and $P_b = \{A \to \langle \langle \alpha \rangle \rangle | A \to \alpha \in P, i = \text{pid}(A \to \alpha)\}$. $V_b$ 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 bracketP($_i$) = bracketP($i$) = $A \to \alpha$ iff pid($A \to \alpha$) = $i$, and bracketN($\langle \alpha \rangle$) = bracketN($\langle \alpha \rangle$) = $A \to \alpha$ iff $\exists \alpha \in P, \text{pid}(A \to \alpha) = i$. A string in the language of $G_b$ describes a parse tree of $G$. Therefore, if two unique strings exist in $L(G_b)$ that become identical after removing their brackets, $G$ is ambiguous.

5.3 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 [KV94]. Typical disambiguation filters for character-level grammars are follow restrictions and rejects [vdBSVV02]. 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.
Chapter 5. Scaling to Scannerless

5.3.1 Example

Figure 5.2 shows an excerpt of a character-level grammar, written in SDF2 [HHKR89, Vis97]. 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.

5.3.2 Definition

We define a character-level context-free grammar \(G_C\) as the eight-tuple \((N, T, C, P_C, S, R_D, R_F, R_R)\) where \(C \subseteq N\) is the set of character classes over \(\mathcal{P}(T)\), \(P_C\) 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 \(\alpha C \beta \Rightarrow \alpha c \beta\) iff \(C \in C\) 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 \(I \times P_C\). Recall that we have defined \(I\) as the set of all items of a grammar.

An element \((A \rightarrow \alpha \cdot B \gamma, B \rightarrow \beta)\) in \(R_D\) means that we are not allowed to derive a \(B\)
non-terminal with production $B \rightarrow \beta$, if it originated from the $B$ following the dot in the production $A \rightarrow \alpha B \gamma$.

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 $\beta A \gamma$, $\gamma$ cannot 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 $B$.

5.4 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 5.1. 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.

5.4.1 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 $Q \xrightarrow{V_b} Q$. The states of the NFA are the items of $G$. The start state is $S' \rightarrow \cdot S\$ and the end state is $S' \rightarrow S\$. 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 $T_1$,
- **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 $T_1$.

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
Chapter 5. Scaling to Scannerless

Algorithm 2 Computing the NFA from a grammar.

function BUILD-NFA() =

1. \( Q = \emptyset \) \# the items of \( G \)
2. \( R = \{ A \rightarrow \alpha \cdot X \cdot \beta \mapsto A \rightarrow \alpha X \cdot \beta \mid \} \) \# shifts
3. \( \cup \{ A \rightarrow \alpha \cdot B \cdot \gamma \mapsto B \rightarrow \cdot \beta \mid i = \text{pid}(B \rightarrow \beta) \} \) \# derives
4. \( \cup \{ B \rightarrow \cdot \beta \mapsto A \rightarrow \alpha \cdot B \cdot \gamma \mid i = \text{pid}(B \rightarrow \beta) \} \) \# reduces
5. return \((Q, R)\)

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 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 \( \mathcal{L}(G) \).

5.4.2 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 \( S' \rightarrow \cdot S \). From there they can progress through the NFA either independently or synchronized, depending on the type of the followed transition. Because we are looking for conflicting paths that represent different parse trees of the same string, the cursors should shift the same symbols. 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 $S' \rightarrow SS^*$. 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 [Sch07a] and Chapter 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 9), 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 $B$. We use $II$ 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 — see Algorithm 4 — 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 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. Since the pair graph might contain dead ends, the traversal also needs to keep track of which item pairs are ‘alive’. This is the case if they can reach an end pair: a pair with $S' \rightarrow SS^*$ for both its items.

Algorithm 4 describes how to find all items used in alive item pairs in the pair graph. It basically is the depth first traversal algorithm of Tarjan to find the strongly connected components (SCCs) of a graph [Tar72], extended to mark alive item pairs. We will not go into the details of the traversal algorithm, but will only comment on the marking of the alive pairs.

The start function for the traversal is TRAVERSE-PATH-PAIRS. It initializes alive, the global set of alive pairs, with the possible end pairs of the pair graph (line 3), and begins the traversal by calling TRAVERSE-PAIR with the start pair (line 4). This function traverses all edge pairs originating at a given item pair using the earlier described function TRAVERSE-EDGES — see Algorithm 3 —, which in turn calls the function TRAVERSE-EDGE for each edge pair. Line 6 of this function adds the source pair of an edge to alive if its target pair is alive. Lines 5–8 of TRAVERSE-PAIR add all elements of a strongly connected component to alive if the root of the component is alive. Thus, if a traversal of a path reaches an end pair or an already visited alive pair, all pairs on the path get marked alive as the path is unwound.
Algorithm 3 Traversing NFA edge pairs.

function TRAVERSE-EDGES(p ∈ Π) =
1  for each (p.q₁ → q’₁) ∈ R do // derive of q₁
2    p’ = p, p’.q₁ = q’₁, p’.c₁ = 0
3    TRAVERSE-EDGE(p, p’)
4  od
5  for each (p.q₂ → q’₂) ∈ R do // derive of q₂
6    p’ = p, p’.q₂ = q’₂, p’.c₂ = 0
7    TRAVERSE-EDGE(p, p’)
8  od
9  for each (p.q₁ X → q’₁), (p.q₂ Y → q’₂) ∈ R do // synchronized shift
10    if SHIFTABLE(X, Y) ≠ ∅ then
11       p’ = p, p’.q₁ = q’₁, p’.q₂ = q’₂
12       // ... ⊗
13       TRAVERSE-EDGE(p, p’)
14    fi
15  for each (p.q₁ λ → q’₁) ∈ R do // conflicting reduction of q₁
16    if CONFLICT(p.q₂), λ then // conflicting reduction of q₂
17       p’ = p, p’.q₂ = q’₂, p’.c₂ = 1
18       // ... ⊗
19       TRAVERSE-EDGE(p, p’)
20    fi
21  for each (p.q₂ λ → q’₂) ∈ R do // conflicting reduction of q₂
22    if CONFLICT(p.q₁), λ then // conflicting reduction of q₁
23       p’ = p, p’.q₁ = q’₁, p’.q₂ = q’₂, p’.c₁ = p’.c₂ = 1
24       // ... ⊗
25       TRAVERSE-EDGE(p, p’)
26    fi
27  if p.c₁ ∨ p.c₂ then
28    for each (p.q₁ λ → q’₁), (p.q₂ λ → q’₂) ∈ R do // synchronized reduction
29       p’ = p, p’.q₁ = q’₁, p’.q₂ = q’₂, p’.c₁ = p’.c₂ = 1
30       // ... ⊗
31       TRAVERSE-EDGE(p, p’)
32  od

function SHIFTABLE(X ∈ V, Y ∈ V) =
1  // returns whether X and Y can be shifted together
2  if X = Y then return X else return ⊖

function CONFLICT(q ∈ Q, λ ∈ T₁) =
1  // returns whether a shift or reduce transition that conflicts with λ can be taken from q
2  return ∃q’ ∈ Q, u ∈ T₁ : (∃X : q →⁺ uXq’) ∨ (∃)q ∈ Q, u ∈ T₁ : q →⁺ uq’
Algorithm 4 Traversing the pair graph.

function TRAVERSE-PATH-PAIRS(Q, R) =
1  unnumber all pairs in \( \Pi \), \( i = 0 \)
2  initialize stack to empty
3  alive = \{ (S' \rightarrow S^\#, S' \rightarrow S^\#, 0, 1), (S' \rightarrow S^\#, S' \rightarrow S^\#, 1, 1) \}
4  TRAVERSE-PAIR((S' \rightarrow S^\#, S' \rightarrow S^\#, 0, 0))
5  return \{ \( q_1, q_2 \mid \exists (q_1, q_2, c_1, c_2) \in \text{alive} \} \}

function TRAVERSE-PAIR(p ∈ \( \Pi \)) =
1  \( i = i + 1 \), lowlink(p) = number(p) = \( i \)
2  push p on stack
3  TRAVERSE-EDGES(p)
4  if lowlink(p) = number(p) then // p is the root of a SCC
5     do // add all pairs in SCC to alive if p is alive
6        pop p' from stack
7        if p ∈ alive then add p' to alive
8     while p' ≠ p

function TRAVERSE-EDGE(p_1 ∈ \( \Pi \), p_2 ∈ \( \Pi \)) =
1  if p_2 is unnumbered then
2     TRAVERSE-PAIR(p_2)
3     lowlink(p_1) = min(lowlink(p_1), lowlink(p_2))
4  else if p_2 is on stack then
5     lowlink(p_1) = min(lowlink(p_1), number(p_2))
6  if p_2 ∈ alive then add p_1 to alive
Algorithm 5 Filtering unused states and transitions from an NFA.

function FILTER-NFA($Q, R, Q_a \subseteq Q$) =
1. // gather subNFA of completely used productions
2. $Q_u = \{ A \rightarrow \alpha \cdot \beta \mid A \rightarrow \alpha \beta \in \text{USED-PRODUCTIONS}(Q_a)\}$
3. $R_u = \{ q_0 \xrightarrow{X_b} q_1 \in R \mid q_0, q_1 \in Q_u \} \quad // R restricted to $Q_u$
4. // prune dead ends
5. $Q' = \{ q \in Q_u \mid S' \rightarrow \cdot S$ $R_u^* q, q \ R_u^* S' \rightarrow S$ $\} \quad // items alive in $R_u$
6. $R' = \{ q_0 \xrightarrow{X_b} q_1 \in R_u \mid q_0, q_1 \in Q' \} \quad // R_u restricted to $Q'$
7. return $(Q', R')$

function USED-PRODUCTIONS($Q' \subseteq Q$) =
1. return $\{ A \rightarrow \alpha \mid \text{proditems}(A \rightarrow \alpha) \subseteq Q' \}$

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

After the items used on conflicting path pairs are collected in $Q_a$, 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. They can be calculated using the function USED-PRODUCTIONS shown in Algorithm 5.

We filter the harmless production rules from the NFA by removing all their items and pruning dead ends. This is described by the function FILTER-NFA in Algorithm 5. 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 — see again Algorithm 1. 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 5.1).

5.4.4 Steps 5–7: NFA Reconstruction and Sentence Generation

In the second part of our framework we use an inverted SGLR parser [vdBSVV02] 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
5.5. Ambiguity Detection for Character-level Grammars

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. 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 5.7 will show.

5.5 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.

5.5.1 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 5.2, 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 ::= \[ \ \t
\]+$) 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 $\varepsilon$. 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
Algorithm 6 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 \(X \in \mathcal{C} \land Y \in \mathcal{C}\) then \# \(X\) and \(Y\) are character classes
    return \(X \cap Y\)
  else if \(X = Y\) then \# \(X\) and \(Y\) are the same non-terminal
    return \(X\)
  else \# no shift possible
    return \(\emptyset\)

Algorithm 7 Filtering derive restrictions from the NFA.

function FILTER-DERIVE-RESTRICTIONS\((R)\) = 
  return \[\{A \to \alpha \cdot B \gamma \vdash \beta \mid i = \text{pid}(B \to \beta), (A \to \alpha \cdot B \gamma, B \to \beta) \in RD\}\]
  \[\{B \to \beta \cdot \vdash A \to \alpha B \cdot \gamma \mid i = \text{pid}(B \to \beta), (A \to \alpha \cdot B \gamma, B \to \beta) \in RD\}\]

section we repair this flaw and make sure that the resulting algorithm does not introduce or
lose ambiguous sentences.

5.5.2 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 \(S(G_6)\).

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 6.

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 7. Then we propagate the follow restrictions through the NFA to make it only generate strings that comply with them. This is described in function PROPAGATE-FOLLOW-RESTRICTIONS in Algorithm 8. 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 $(S' \rightarrow \varepsilon, S$, $\emptyset)$ 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 $(A \rightarrow \alpha B \beta, f)$. 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 unnecessarily. The restrictions after a shift of $\alpha$ are the tails of the strings in $f$ beginning with $\alpha$, 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 $(A \rightarrow \alpha B \beta, f)$ once their corresponding reduce edges are known. This is convenient because we can let the propagation calculate the different follow restrictions that can reach $A \rightarrow \alpha B \beta$. Once the restrictions that were passed to the derive have reached a state $B \rightarrow \gamma \varepsilon$, we propagate them upwards again over a reduce edge to $A \rightarrow \alpha B \beta$. 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 $B \rightarrow \gamma \varepsilon$, the function SHIFT-ENDS returns all states that are at $B \rightarrow \gamma \varepsilon$ and that are reachable by shifting.

If the reduced production is of form $B \rightarrow \varepsilon$ we create a special non-terminal symbol $B^\varepsilon$ 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 $\varepsilon$.

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 5.7 we will see average increases in NFA size of a factor 2–3.

Rejets

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 would not violate the finiteness and overapproximation requirements. However, if the language of $B$ is context-free we have to
Algorithm 8 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')
  Q' = {(S' → •S$, $)}, R' = ∅
  repeat
    add all states used in R' to Q'
    for q_f = (A → α•Bβ, f) ∈ Q' do
      if B ∈ C then # B is a character class
        for a ∈ B, a /∈ f do # all shiftable characters in B
          let B' = {b | b ∈ B, b /∈ f, NEXT-FOLLOW(a, f) = NEXT-FOLLOW(b, f)}
          add q_f mapper (A → αB•β, NEXT-FOLLOW(a, f)) to R'
        od
      else # B is a normal non-terminal
        for A → α•Bβ do # propagate f over derivation
          add q_f mapper (q', f) to R' # propagate f over derivation
          let q_f' = (A → αB•β, f' ∪ R_F(B)) # shift target
          add q_f' mapper (q', f') to R' # reduction to shift target
          if bracketP((q)) = B → ε then # non-terminal shift representing empty string
            add q_f mapper (q') to R' # non-terminal shift of non-empty strings
          else
            add q_f mapper (q') to R' # non-terminal shift of non-empty strings
          od
        od
      od
  until no more edges can be added to R'
  return (Q', R')

function SHIFT-ENDS((A → •α, f) ∈ Q') =
  # return the states at the end of A → α, reachable from q using only shifts
  let → = {q → q' | q mapper B' → q' ∈ R'} # the shift transitions of R'
  return {(A → •α, f') | (A → •α, f) →* (A → α•, f')}
5.5. Ambiguity Detection for Character-level Grammars

Algorithm 9 Extensions to TRAVERSE-EDGES for avoiding rejected keywords.

```plaintext
// at ⊗ (shift) insert:
\[ p'.r_1 = p'.r_2 = \emptyset \] // clear reduced sets

// at ⊘ and ⊙ (conflicting and pairwise reduce) insert:
if not CHK-REJECT(⟩_i, p.r_2) then continue
\[ p'.r_1 = NEXT-REJECT(⟩_i, p.r_1) \]

// similarly, insert at ⊘ and ⊙:
if not CHK-REJECT(⟩_i, p.r_1) then continue
\[ p'.r_2 = NEXT-REJECT(⟩_i, p.r_2) \]
```

function CHK-REJECT(⟩_i \in T, r \in P(N)) =
// returns whether a reduction with ⟩_i is possible after reductions r on other path
let \[ A = \text{bracket}(⟩_i) \]
return \[ \neg \exists B \in r : (A, B) \in R_R \lor (B, A) \in R_R \]

function NEXT-REJECT(⟩_i \in T, r \in P(N)) =
// adds non-terminal reduced with ⟩_i to r if it is involved in a reject
let \[ A = \text{bracket}(⟩_i) \]
if \[ \exists B \in r : (A, B) \in R_R \lor (B, A) \in R_R \]
then return \[ r \cup \{A\} \]
else return \[ r \]

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, 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 9 shows this PG extension.

5.5.3 NFA Reconstruction

In Section 5.5.1 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 \( B \rightarrow \beta \) are filtered out at a specific item \( A \rightarrow \alpha \cdot B \gamma \),
but not at other items, we know \( B \rightarrow \beta \) is harmless in the context of \( A \rightarrow \alpha \cdot B \gamma \). 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.

5.6 Grammar Unfolding

In Section 5.7 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 \( A \rightarrow \alpha \cdot B \beta \) and shifting a string in the language of \( B \), the NFA allows reductions
to any item of form \( C \rightarrow \gamma B \delta \). 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 \( \varepsilon \), 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 [BGM10]. 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 occurrence
of an unfolded non-terminal. After the derivation from an item \( A \rightarrow \alpha \cdot B \beta \) a path can now
only reduce back to \( A \rightarrow \alpha B \beta \), 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
5.7. Experimental Results

We have evaluated our ambiguity detection algorithm for character-level grammars on the grammar collection shown in Table 5.1. All grammars are specified in SDF2 [HHKR89, Vis97]. The selection of this set is important for external validity. We have opted for grammars 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.

### Table 5.1: Character-level grammars used for validation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Productions</th>
<th>SLOC</th>
<th>Non-terminals</th>
<th>Derive restrictions</th>
<th>Follow restrictions</th>
<th>Reserved keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>324</td>
<td>415</td>
<td>168</td>
<td>332</td>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>C++</td>
<td>807</td>
<td>4172</td>
<td>430</td>
<td>1</td>
<td>87</td>
<td>74</td>
</tr>
<tr>
<td>ECMAScript</td>
<td>403</td>
<td>522</td>
<td>232</td>
<td>1</td>
<td>27</td>
<td>25</td>
</tr>
<tr>
<td>Oberon0</td>
<td>189</td>
<td>202</td>
<td>120</td>
<td>132</td>
<td>31</td>
<td>27</td>
</tr>
<tr>
<td>SQL-92</td>
<td>419</td>
<td>495</td>
<td>266</td>
<td>23</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>Java 1.5</td>
<td>698</td>
<td>1629</td>
<td>387</td>
<td>297</td>
<td>78</td>
<td>56</td>
</tr>
</tbody>
</table>

1. SDF2 grammar library, revision 27501, http://www.meta-environment.org
2. TRANSFORMERS 0.4, http://www.lrde.epita.fr/cgi-bin/twiki/view/Transformers/Transformers
4. RASCAL Oberon0 project (converted to SDF2), rev. 34580, http://svn.rascal-mpi.org/oberon0/
5. SQL-FRONT, revision 20713, http://strategoxt.org/Stratego/SqlFront

* After removal of additional attribute code

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 5.4, with the only addition that whole character-classes are shifted instead of individual literal strings like keywords, brackets and operators. Later on we will refer to this unfolding extension as CHAR+UNF.

5.7 Experimental Results

We have evaluated our ambiguity detection algorithm for character-level grammars on the grammar collection shown in Table 5.1. All grammars are specified in SDF2 [HHKR89, Vis97]. The selection of this set is important for external validity. We have opted for grammars 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.

5.7.1 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 5.4, with the only addition that whole character-classes are shifted instead of individual literal strings like keywords, brackets and operators. Later on we will refer to this unfolding extension as CHAR+UNF.
Chapter 5. Scaling to Scannerless

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Method</th>
<th>Harmless productions</th>
<th>NFA edges filtered</th>
<th>Time (sec)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BASE</td>
<td>48 / 324</td>
<td>343 / 14359</td>
<td>64</td>
<td>2128</td>
</tr>
<tr>
<td>C</td>
<td>CHAR</td>
<td>62 / 324</td>
<td>2283 / 24565</td>
<td>120</td>
<td>3345</td>
</tr>
<tr>
<td></td>
<td>CHAR+UNF</td>
<td>75 / 324</td>
<td>8637 / 30653</td>
<td>97</td>
<td>2616</td>
</tr>
<tr>
<td></td>
<td>BASE</td>
<td>0 / 807</td>
<td>0 / 8644</td>
<td>32</td>
<td>1408</td>
</tr>
<tr>
<td>C++</td>
<td>CHAR</td>
<td>0 / 807</td>
<td>0 / 39339</td>
<td>527</td>
<td>7189</td>
</tr>
<tr>
<td></td>
<td>CHAR+UNF</td>
<td>–</td>
<td>–</td>
<td>&gt;9594</td>
<td>&gt;17.3G</td>
</tr>
<tr>
<td></td>
<td>BASE</td>
<td>44 / 403</td>
<td>414 / 4872</td>
<td>12</td>
<td>547</td>
</tr>
<tr>
<td>ECMAScript</td>
<td>CHAR</td>
<td>46 / 403</td>
<td>1183 / 10240</td>
<td>46</td>
<td>1388</td>
</tr>
<tr>
<td></td>
<td>CHAR+UNF</td>
<td>88 / 403</td>
<td>9887 / 19890</td>
<td>31</td>
<td>1127</td>
</tr>
<tr>
<td></td>
<td>BASE</td>
<td>0 / 189</td>
<td>0 / 3701</td>
<td>4.2</td>
<td>256</td>
</tr>
<tr>
<td>Oberon0</td>
<td>CHAR</td>
<td>70 / 189</td>
<td>925 / 6162</td>
<td>90</td>
<td>349</td>
</tr>
<tr>
<td></td>
<td>CHAR+UNF</td>
<td>73 / 189</td>
<td>10837 / 20531</td>
<td>14</td>
<td>631</td>
</tr>
<tr>
<td></td>
<td>BASE</td>
<td>13 / 419</td>
<td>98 / 4944</td>
<td>16</td>
<td>709</td>
</tr>
<tr>
<td>SQL-92</td>
<td>CHAR</td>
<td>20 / 419</td>
<td>239 / 9031</td>
<td>83</td>
<td>2093</td>
</tr>
<tr>
<td></td>
<td>CHAR+UNF</td>
<td>65 / 419</td>
<td>7285 / 14862</td>
<td>37</td>
<td>1371</td>
</tr>
<tr>
<td></td>
<td>BASE</td>
<td>0 / 698</td>
<td>0 / 16844</td>
<td>60</td>
<td>2942</td>
</tr>
<tr>
<td>Java 1.5</td>
<td>CHAR</td>
<td>0 / 698</td>
<td>0 / 45578</td>
<td>407</td>
<td>7382</td>
</tr>
<tr>
<td></td>
<td>CHAR+UNF</td>
<td>189 / 698</td>
<td>180456 / 262030</td>
<td>1681</td>
<td>15568</td>
</tr>
</tbody>
</table>

*Run on Amazon EC2 High-Memory Extra Large Instance

| Table 5.2: Timing and precision results of filtering harmless productions.

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 5.5, including extensions for follow restrictions, derive restrictions and rejects.

- **CHAR+UNF**: the CHAR algorithm combined with grammar unfolding (Section 5.6).

5.7.2 Results and Analysis

In Table 5.2 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).

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 \texttt{CHAR} improved substantially upon \texttt{BASE}, without the risk of missing ambiguous sentences.

Of all three configurations \texttt{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 5.3 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 \texttt{CHAR} and \texttt{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 either \texttt{CHAR} or \texttt{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 \texttt{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 \texttt{CHAR+UNF} was superior to \texttt{CHAR}, due to its higher precision and lower run-times.

The third column of Table 5.3 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.
### Table 5.3: Timing results of sentence generation. Times are in seconds. For each sentence length, the run-time of the fastest configuration (after taking filtering time into account) is highlighted. Speedup is calculated as \( \frac{\text{unfiltered sentence gen. time}}{\text{filtering time + sentence gen. time}} \).

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Len</th>
<th>Ambig NTs</th>
<th>Unfiltered</th>
<th>CHAR</th>
<th>CHAR+UNF</th>
<th>Maximum speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>#Sent.</td>
<td>Time</td>
<td>#Sent.</td>
<td>Time</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td></td>
<td>3.45K</td>
<td>7.9</td>
<td>273K</td>
<td>5.9</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>5.06M</td>
<td>35</td>
<td>3.77M</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td>7.55M</td>
<td>398</td>
<td>53.4M</td>
<td>270</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>1.13G</td>
<td>5442</td>
<td>756M</td>
<td>3466</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td>17.0G</td>
<td>78987</td>
<td>10.8G</td>
<td>47833</td>
</tr>
<tr>
<td>ECMAScript</td>
<td>3</td>
<td></td>
<td>14.2K</td>
<td>4.5</td>
<td>11.7K</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>274K</td>
<td>11</td>
<td>217K</td>
<td>8.9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>5.17M</td>
<td>149</td>
<td>3.92M</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td>96.8M</td>
<td>2805</td>
<td>7.05M</td>
<td>2186</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td>1.80G</td>
<td>54715</td>
<td>1.26G</td>
<td>41091</td>
</tr>
<tr>
<td>Oberon0</td>
<td>22</td>
<td></td>
<td>21.7M</td>
<td>60</td>
<td>320</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td></td>
<td>62.7M</td>
<td>186</td>
<td>571</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td></td>
<td>247M</td>
<td>815</td>
<td>1269</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td></td>
<td>1.39G</td>
<td>4951</td>
<td>3173</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td></td>
<td>9.56G</td>
<td>35007</td>
<td>9807</td>
<td>1.3</td>
</tr>
<tr>
<td>SQL-92</td>
<td>11</td>
<td></td>
<td>2.65M</td>
<td>16</td>
<td>1.54M</td>
<td>9.4</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td>15.8M</td>
<td>102</td>
<td>7.36M</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td>139M</td>
<td>1018</td>
<td>51.3M</td>
<td>379</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
<td>1.49G</td>
<td>11369</td>
<td>453M</td>
<td>3572</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
<td>14.3G</td>
<td>22530</td>
<td>4.39G</td>
<td>35024</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Java 1.5</td>
<td>7</td>
<td></td>
<td>187K</td>
<td>33</td>
<td>39.1K</td>
<td>6.8</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>3.15M</td>
<td>115</td>
<td>482K</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td>54.7M</td>
<td>1727</td>
<td>6.05M</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>959M</td>
<td>3965</td>
<td>76.2M</td>
<td>4745</td>
</tr>
</tbody>
</table>

#### 5.7.3 Validation

In Chapter 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 [Sch01] and CFG ANALYZER [AHL08]. For this we used a grammar collection also used in Chapter 2, 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 [vdBSVV02] parser used for SDF2. 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 SDF2 grammars in Table 5.1.

- 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 — see Chapter 4 before comparison. This is done by removing the unambiguous substrings from an ambiguous sentence, and replacing them with their deriving non-terminal.

## 5.8 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.