SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Lexical State Analyzer for JavaCC grammars

Kartik Gupta[1] and V. Krishna Nandivada[1]

[1] *Dept of CSE, IIT Madras*

SUMMARY

Lexical states in JavaCC provide a powerful mechanism to scan regular expressions in a context sensitive manner. But lexical states also make it hard to reason about the correctness of the grammar. We first categorize the related correctness issues into two classes: errors and warnings. We then extend the traditional context sensitive and a context insensitive analysis to identify errors and warnings in context-free-grammars (CFGs). We have implemented these analyses as a standalone tool (`LSA`), the first of its kind, to identify errors and warnings in JavaCC grammars. The `LSA` tool outputs a graph that depicts the grammar and the error transitions. Importantly, it can also generate counter example strings that can be used to establish the errors. We have used `LSA` to analyze a host of open-source JavaCC grammar files to good effect.

Received . . .

1. INTRODUCTION

JavaCC lexical states provide a convenient mechanism to conditionally activate lexical tokens. For the same input substring, use of lexical states can allow different lexical tokens to be recognized based on prior parsed tokens. For example, when parsing a C program, the parser may put the scanner in a special state (say `COMMENT`) when it encounters "/*"; when the scanner is in this state the input substring "int" is not recognized as a *keyword* token (INT) but is treated as part of the comment string. In other words the token INT is not *active* in the lexical state `COMMENT`. The popularity of lexical states can be seen by the number of open-source grammars, submitted on the JavaCC website [4], that use lexical states. The advantage of lexical states is that they make the specification of the lexical rules simpler. This simplicity comes with its own cost—lexical states make it extremely challenging to manually reason about the correctness of the grammar. We illustrate the same using an example.

Figure 1 shows a snippet of JavaCC grammar to parse a subset of BibTex files. Note that JavaCC expects the rules for lexical analysis (regular expressions) and parsing (context free grammar) to be present in a single file. In JavaCC, the specification $\texttt{<I}_1\texttt{,I}_2\texttt{,...,I}_n\texttt{>TOKEN:<X:RegEx>:O}_s$ indicates that the scanner can return a token `X` when it matches the regular expression `RegEx`, only if the current lexical state is $\texttt{I}_1$, or $\texttt{I}_2$, or .. $\texttt{I}_n$ and after scanning the token the state changes to $\texttt{O}_s$. Specifying the in-state (such as $\texttt{I}_1\texttt{, I}_2$) and out-state (such as $\texttt{O}_s$) are optional; the default in-state is the special state `DEFAULT` and the default out-state is the in-state in which the token is scanned. The regular expression specification declares a set of tokens (e.g., `AT_SYM`, `ANYTHING_BUT_AT`, `ARTICLE`, `INPROC` and so on). If a token is used to define another token (e.g., `OTHERS`), then it has to be declared in a special manner – by prefixing the token with #. Finally, the regex `~[]` is used to match any single character (including a newline character).

```
// regular expression specification
<DEFAULT>TOKEN:{
  <AT_SYM:"@">:ENTRY
 |<ANYTHING_BUT_AT:~["@"]>:DEFAULT}
<ENTRY>TOKEN:{
 <ARTICLE:"article">:FIELDS
 |<INPROC:"inproceedings">:FIELDS}
<FIELDS>TOKEN:{
 <AUTHOR:"author">|<TITLE:"title">}
<FIELDS>TOKEN:{
  <LB:"{"> |<RB:"}">
 |<QT:"\"">:QT_DATA
 |<EQ:"=">|<HASH:"#">|<COMMA:",">
 |<IDENTIFIER:(<OTHERS>)+>
 |<#OTHERS:~["@","","","(",")",
 "\"","=","#",","," ","\t","\n"]>}
<QT_DATA>TOKEN:{
  <QT_IN_QT_DATA:"\"">
 |<ETC_IN_QT_DATA:~[]> }
<BR_DATA>TOKEN:{
  <RB_IN_BR_DATA:"}">
 |<ETC_IN_BR_DATA:~[]> }
```

```
// set of productions
void InputFile():{}{
  (<AT_SYM> Block()
 |<ANYTHING_BUT_AT>)* <EOF> }
void Block():{}{
(<ARTICLE>|<INPROC>)
  <LB>Entry()<RB> }
void Entry():{}{
 Key()(<COMMA>Field())* }
void Key():{}{
 <IDENTIFIER> }
void Field():{}{
 (<AUTHOR>|<TITLE>)<EQ>Data()}
void Data():{}{
 <QT>QtString()|<LB>BrString()}
void QtString():{}{
 (<ETC_IN_QT_DATA>)*<QT_IN_QT_DATA>
}
void BrString():{}{
 (<ETC_IN_BR_DATA>)*<RB_IN_BR_DATA>
}
```

Figure 1. Snippet of JavaCC file for parsing BibTex files.

Each JavaCC production rule looks like a function definition and the body of the function includes production rules. If a non-terminal appears on the RHS of any production, it is written as a function call. For example, the production `Entry` indicates that it starts with a non-terminal `Key` and after that it may contain zero or more occurrences of `COMMA` (a terminal) and `Field` (a non-terminal) pairs.

An input BibTex file (to be parsed by the grammar in Figure 1) is expected to consist of zero or more citation blocks. Suppose we have the following input:

```
@inproceedings{Tarjan71,
  author = "Robert Endre Tarjan",
  title  = "Depth-first search and linear graph algorithms" }
```

A glance at the production rules will let the developer naively believe that the grammar will parse the above input, using the following derivation steps: `InputFile` $\rightarrow$ `AT_SYM Block` $\rightarrow^*$ `AT_SYM IN_PROC LB IDENTIFIER COMMA AUTHOR EQ Data COMMA Field RB` $\rightarrow^*$ `AT_SYM IN_PROC LB IDENTIFIER COMMA AUTHOR EQ QT ETC_IN_QT_DATA QT_IN_QT_DATA COMMA Field RB` and so on. We now see the impact of lexical states.

By default, the scanner starts in the `DEFAULT` state[*]. Upon reading the "`@`" symbol the scanner switches its state to `ENTRY`. In this state, the scanner identifies the `INPROC` token and it switches the state to `FIELDS`. In this state, the scanner identifies a series of tokens such as `LB`, `IDENTIFIER` (to be parsed as `Key`), `COMMA`, `AUTHOR` and `EQ`. The parser now expects to match the production `Data`. The scanner first identifies a quote (`QT`) and switches state to `QT_DATA`. In this state, the scanner matches `ETC_IN_QT_DATA` multiple times and then it identifies `QT_IN_QT_DATA`. At this point, the parser is expecting the token `COMMA` or `RB`, but the

[*]The scanner state can be changed by using the SwitchTo() construct provided by JavaCC, which changes the lexical state of the scanner to the value passed as argument.

DOI: 10.1002/spe

scanner reads these tokens only in the lexical state `FIELDS`, which does not match the current lexical state `QT_DATA`. Thus, the parser will mark the input string as syntactically incorrect.

Thus, contrary to the naive conclusion drawn by the grammar designer, the presence of lexical states has rendered the production rules incorrect. In other words, `Block` has a *dead* production rule that will never be matched: we cannot match `RB` after `Entry` has been matched. Consequently, parts of the grammar rules for `InputFile` (such as, `AT_SYM Block() EOF`) and `Entry` (such as, `Key() COMMA Field() COMMA Field()`) will never be matched. Such errors can be much more complicated in bigger grammars and manual tracking can be hard. Unfortunately, there does not exist any tool that analyzes grammars with respect to lexical states. In this paper, we present a tool to fill this gap. We begin by formulating a classification of bugs in grammars that use lexical states.

Definite errors (abbreviated as *errors*): We call it an error in the grammar to have a (sub) production that will never be matched. For example, the bug discussed in the previous section corresponds to an error. The grammar shown in Figure 1 contains another error that manifests itself when the scanner is in the lexical state `FIELDS` and the parser needs to use the non-terminal `Data` to derive `<LB> BrString()`, to parse something like {`Robert Endre Tarjan`}. Here the parser needs the lexical token `ETC_IN_BR_DATA` (or `RB_IN_BR_DATA`), which can only be identified in the lexical state `BR_DATA`.

We extend the notion of in- and out-states to non-terminals: Given a non-terminal N_1, the in-state of N_1 is the union of all the in-states of the terminals present in the *FIRST set* [6] of N_1. The FIRST set of a non-terminal is the set of all terminals that can occur as the first symbol in some sentential form that can be derived from this non-terminal. Similarly, we can also define the LAST set of a non-terminal N_1: the last terminal contained in any sentence derived from N_1 is a member of the LAST(N_1). The out-state of N_1 is the union of the out-states of the terminals present in LAST(N_1).

Possible errors (abbreviated as *warnings*): Consider a grammar rule $A \to \alpha\beta$, where α and β each represent a sequence of one or more terminal and non-terminal symbols. Say β can be derived from some of the out-states of α, but there exist out-states of α from which β cannot be derived. In such a case, depending on the specific input, after matching α we may reach a state s that is not a valid in-state of β. We term these as *warnings* in the grammar. The grammar snippet shown in Figure 1 has a few warnings as well. For example, we may be able to match `Entry` (as part of `Block`), if the input is something like `@inproceedings`{`Tarjan71`}. But if the input contains some fields that have to be matched to one or more instance of `<COMMA>Field()` in `Entry` then we cannot match it.

It can be easily seen that manually finding errors and warnings is non-trivial and real-world grammars (consisting of numerous terminals and non-terminals) that use lexical states can become a formidable challenge. Similarly, while it is fairly trivial to identify errors in grammars with no lexical states, it may be noted that naive translation of a JavaCC grammar with lexical states to a version that does not use lexical states can lead to an exponential blow up, in terms of the number of non-terminals. This explosion renders the approach impractical (see the discussion in Section 2.7). We present a set of automated techniques to efficiently reason about errors and warnings in context-free-grammars.

Our contributions:

• We formulate the problem of identifying errors and warnings in grammars that use lexical states.

• We extend traditional program analysis techniques to present two analyses to identify errors and warnings. Our first analysis (context insensitive lexical state analysis) computes summary in- and out-states for each non-terminal and it does not take into consideration the position (context) in which the non-terminal appears in any production rule. This summary of in- and out-states is used to conservatively identify the errors and warnings. Our second analysis (context sensitive lexical state analysis) computes the out-states for each non-terminal N_1 specific to the context (position and in-state) in which N_1 may be parsed. Based on the precise out-states we compute all the errors that may occur in a production, for each possible

Softw. Pract. Exper. (0000)
DOI: 10.1002/spe

lexical in-state for that production (Section 2).
• We have implemented these analyses as a standalone tool (`LSA`) that can identify errors and warnings in JavaCC grammars. The `LSA` tool outputs a graph that depicts the grammar and the error transitions. It can generate example strings (counterexamples) that can be used to establish the errors (Section 3). To the best of our knowledge, this is the first tool that finds bugs in grammars that use lexical states.
• We have evaluated our `LSA` tool on a host of open-source JavaCC grammar files to good effect. We find that our techniques help catch errors and warnings that are otherwise not caught by the naive *unreachable* production detection algorithm that marks all the transitively unreachable non-terminals from the start non-terminal without considering the lexical states (Section 4).

2. LEXICAL STATE VERIFIER

In this section, we first discuss the grammar subset over which we illustrate our analysis. Then we present three algorithms to analyze these grammars: the naive *useless* productions removal algorithm (adapted from Hopcroft et al [15]), our context insensitive lexical state analysis, and our context sensitive lexical state analysis. We follow it up with a discussion on the algorithms and our counter example derivation process. Through out this paper, we assume that the input grammar is syntactically valid and is accepted by the current JavaCC tool (that is, has no left recursion, and so on).

2.1. *Grammar subset*

We first discuss a representative scheme for token and grammar specification. We will assume that our input grammar follows this specification. Our specification can be used to generate grammars in JavaCC format trivially. Details of the JavaCC syntax can be found in the manual [2].

A typical definition of lexical tokens is of the form:

```
<I1, I2 ... In> TOKEN: {<Token1:RegEx1> : Os <Token2:RegEx2> }
```

It defines two tokens `Token1` and `Token2` corresponding to two regular expressions `RegEx1` and `RegEx2`. Given a string matching `RegEx1` (or `RegEx2`), the scanner returns the token `Token1` (or `Token2`) if its current state $s \in \{$`I1, I2, ... In`$\}$. If the scanner returns the token `Token2`, the scanner will remain in state s. If the scanner returns the token `Token1`, the scanner will switch to state `Os`. Thus, every lexical token has a non-empty set of in-states and a corresponding set of out-states.

We assume that the input grammar contains rules with only the following forms:

$N_0 \rightarrow N_1 \| N_2$	*// Alternate*	$N_0 \rightarrow N_1 N_2$	*// Sequence*
$N_0 \rightarrow T$	*// Terminal*	$N_e \rightarrow \epsilon$	*// Epsilon*

We use T to denote terminals and N_i to denote non-terminals in the grammar. We expect that N_e is the only non-terminal whose production string is ϵ. We will also assume that every non-terminal must have a unique production associated with it. It should be noted that any LL grammar can be transformed to use only the forms of rules specified above without losing its LL property. All JavaCC grammars have the LL property and are handled by our implementation.

A context-free grammar can be specified using the four tuple (N, T, P, S), where N is a set of non-terminals, T is a set of terminals, P is a set of productions in the above described form and $S \in N$ is the start non-terminal symbol. We use the '.' notation to dereference the elements of the tuple; for example, $G.N$ denotes the set N of grammar G.

2.2. *Useless Production Elimination by detecting the Useful ones*

For the sake of completeness and ease of presentation, we next present a naive procedure (derived from the well understood algorithm of Hopcroft et al [15]) to eliminate *useless*

DOI: 10.1002/spe

```
Find-Useful-Productions(G)
begin
    Visit(G.S);
    Set D = {};
    foreach n ∈ G.N do
        if isVisited[n] == true then
            D.add(n);
    return D;
end
```

```
Visit(N1)
begin
    if the production corresponding to N1 is
    of the form N1 → N2N3 or N1 → N2|N3
    then
        if !isVisited[N2] then
            isVisited[N2] = true; Visit(N2);
        if !isVisited[N3] then
            isVisited[N3] = true; Visit(N3);
end
```

Figure 2. Naive algorithm to find useful productions.

NT: Set of non-terminals	LS: Set of lexical states	TS: Set of terminals $\cup\{\epsilon\}$
$\mathcal{O}$: TS $\rightarrow$ LS | $\mathcal{I}$: TS $\rightarrow$ LS	`inStates`: NT $\rightarrow \mathbb{P}$(LS)	`outStates`: NT $\rightarrow \mathbb{P}$(LS)

Figure 3. Sets and maps used in lexical state analysis

productions (UPs) in the grammar. We call a production as useless, if it cannot be reached from the *start* non-terminal. Figure 2 presents a sketch of the algorithm that works as the basis of this procedure. Starting with the start non-terminal S, we "visit" all the non-terminals and mark the non-terminals used in the corresponding productions. We make a post-pass to collect and return all the marked non-terminals (in variable D). The set of useless productions is given by $N - D$. As it can be seen, this algorithm does not take into consideration the lexical states of the terminals in use. Thus, the effectiveness of this algorithm is limited.

2.3. Context Insensitive Lexical State Analysis

We now present our context insensitive lexical state analysis. The analysis populates two different maps `inStates` and `outStates` (Figure 3) for its internal use. For each non-terminal, the `inStates` and `outStates` maps store the in-states and out-states, respectively. For all the non-terminals, these two maps are initialized to contain empty sets. We use $\mathbb{P}(X)$ to denote the power set of X. We assume that for the set of terminals and ϵ, the out-state map ($\mathcal{O}$) and in-state map ($\mathcal{I}$) are trivially precomputed (code not shown). The identity map represents the out-state and in-state maps for ϵ.

Figure 4 presents a sketch of our context insensitive analysis. The main function `Main-CInsensitive` takes the grammar ($G = (N, T, P, S)$) as input and first calls `Find-Useful-Productions` to identify all the useful productions. It follows a worklist-based approach to compute the out- and in-states for all the non-terminals. We say that a non-terminal N_2 *uses* a non-terminal N_1, if N_1 appears on the right side of the production corresponding to N_2.

`CI-BuildOutStates`: The out-state of a non-terminal depends on the exact production corresponding to the non-terminal. If the production is of the form $N_0 \rightarrow N_1|N_2$, then out-states of N_0 includes the out-states of N_1 and N_2. If the production is of the form $N_0 \rightarrow N_1N_2$, then out-states of N_0 includes the out-states of N_2 and optionally that of N_1, if N_2 derives the empty string ϵ.

`CI-BuildInStates`: Similar to the construction of `outStates`, we update the `inStates` map for each production depending on its form. One main difference between the two is that when the production is of the form $N_0 \rightarrow N_1N_2$: the in-states set for N_0 contains the in-states set for N_1 and optionally that of N_2, if N_1 derives the empty string ϵ.

`CI-Analyze`: After the in-states set and out-states set have been computed for each non-terminal, we check if the start non-terminal ($G.S$) can be parsed in the default lexical state

```
Func Main-CInsensitive(G)
begin
  Worklist wlist = Find-Useful-Productions(G);
  while wlist is not empty do
    N_1 = wlist.removeOne(); CI-BuildOutStates (N_1);
    if outStates[N_1] has changed then
      add to wlist all the non-terminals that use N_1.
  wlist = Find-Useful-Productions(G);
  while wlist is not empty do
    N_1 = wlist.removeOne(); CI-BuildInStates (N_1);
    if inStates[N_1] has changed then
      add to wlist all the non-terminals that use N_1.
  if DEFAULT ∉ inStates(G.S) then // issue an error
  foreach N_i ∈ G.N do  CI-Analyze(N_i);
end

Func CI-BuildOutStates(NonTerminal N_0)
begin
  switch structure of N_0 do
    case N_0 → N_1|N_2:  outStates[N_0] = outStates[N_1] ∪ outStates[N_2];
    case N_0 → N_1N_2:
      outStates[N_0] = outStates[N_2];
      if N_2 →* ε then  outStates[N_0] = outStates[N_0] ∪ outStates[N_1];
    case N_0 → T:  outStates[N_0] = O(T);
    case N_0 → ε:  outStates[N_0] = O(ε);
end
Func CI-BuildInStates(NonTerminal N_0)
begin
  switch structure of N_0 do
    case N_0 → N_1|N_2:  inStates[N_0] = inStates[N_1] ∪ inStates[N_2];
    case N_0 → N_1N_2:
      inStates[N_0] = inStates[N_1];
      if N_1 →* ε then  inStates[N_0] =inStates[N_0] ∪ inStates[N_2];
    case N_0 → T:  inStates[N_0] = I(T);
    case N_0 → ε:  inStates[N_0] = I(ε);
end
Func CI-Analyze(NonTerminal N)
begin
  if production corresponding to N_0 is of the form N_0 → N_1N_2: then
    O_s = outStates[N_1]; I_s = O_s − inStates[N_2];
    if I_s == O_s then  // error -- N_0
    else if I_s ≠ {} then  // warning -- N_0
end
```

Figure 4. Context insensitive lexical state analysis

(`DEFAULT`). We then invoke the `CI-Analyze` method to check if the lexical states (S) in which a non-terminal N_0 can be accessed matches that of its in-states (`inStates`$[N_0]$). If there are no common elements between S and `inStates`$[N_0]$, then it is flagged as an `error`. If S includes lexical states that are not part of `inStates`$[N_0]$, then it is a possible error and hence marked as a `warning`. A context insensitive error/warning consists of just the non-terminal for which the error/warning is identified.

```
<DEFAULT>TOKEN:{ <AT:"a">:DEFAULT }
<LX1>TOKEN:{ <CT:"c">:DEFAULT }
<DEFAULT, LX1>TOKEN:{ <BT:"b"> }
void S():{}{ F()|G() }
void G():{}{ D()E() }
```

```
void A():{}{ <AT> }
void B():{}{ <BT> }
void C():{}{ <CT> }
void D():{}{ E()E() }
void E():{}{ B()C() }
void F():{}{ D()C() }
```

Figure 5. Example grammar with two lexical states

NonTerminal	context insensitive analysis			context sensitive analysis		
	InStates	OutStates	Error/ Warning	OutStates		Error
				DEF	LX1	
S	DEF, LX1	DEF	-	$\mathcal{E}$	$\mathcal{E}$	DEF, LX1
G	DEF, LX1	DEF	-	$\mathcal{E}$	$\mathcal{E}$	DEF, LX1
A	DEF	DEF	-	DEF	$\mathcal{E}$	LX1
B	DEF, LX1	DEF, LX1	-	DEF	LX1	-
C	LX1	DEF	-	$\mathcal{E}$	DEF	DEF
D	DEF, LX1	DEF	-	$\mathcal{E}$	$\mathcal{E}$	DEF, LX1
E	DEF, LX1	DEF	Warning	$\mathcal{E}$	DEF	DEF
F	DEF, LX1	DEF	Error	$\mathcal{E}$	$\mathcal{E}$	DEF, LX1

Figure 6. Effect of applying our context insensitive (CI) and context sensitive (CS) analysis on the example shown in Figure 5. The DEFAULT state is abbreviated to DEF.

Example: Figure 5 shows a sample grammar with two lexical states (`DEFAULT` and `LX1`). The grammar is chosen so as to demonstrate three important features of the tool : i) the errors and warnings issued by our proposed context insensitive analysis, ii) the errors issued by our context sensitive analysis, and iii) an interesting facet of our context sensitive analysis that it may report errors that are not reflected by the context insensitive anlaysis (neither as an error, nor warning). The in-, out-states computed using the context insensitive analysis along with identified errors and warnings are shown in columns 2-4 of Figure 6. For example, it says that non-terminal `F` will always lead to an error state.

Complexity: We will use L to denote the number of lexical states, N to denote the grammar size; in the worst case $L = O(N)$. The complexity of `CI-BuildOutStates` and `CI-BuildInStates` functions is $O(1)$. Each of the while loops in `Main-CInsensitive` is at most invoked $O(N \times L)$ times – in each iteration, size of the `outStates` map of at least one non-terminal increases by one – giving rise to an overall complexity of $O(N \times L)$.

2.4. Context Sensitive Analysis

We now describe our context sensitive analysis. Here the set of lexical states LS, contains an additional error state $\mathcal{E}$. If a terminal or non-terminal cannot be parsed in a specific lexical state (including the error state $\mathcal{E}$), then we consider the resulting lexical state to be $\mathcal{E}$. Compared to the context insensitive analysis, the `outStates` map contains more detailed information. It stores the out-states for each non-terminal for each possible lexical in-state – `outStates`: NT $\times$ LS $\rightarrow P(\text{LS})$. For all the non-terminals, for each lexical token, this map is initialized to contain empty sets. For the `outStates` map, we use a specialized union operator ($\sqcup$) to do an element-wise union of all the elements of the operands.

$$S = \texttt{outStates}[N_1] \sqcup \texttt{outStates}[N_2]$$
$$\equiv$$
$$\forall i \in \text{LT} : S[i] = \texttt{outStates}[N_1][i] \cup \texttt{outStates}[N_2][i]$$

DOI: 10.1002/spe

```
Func Main-CSensitive(G)
begin
    Worklist wlist = Find-Useful-Productions(G);
    while wlist is not empty do
        N1 = wlist.removeOne(); CS-BuildOutStates (N1, φ);
        if outStates[N1] has changed then
            add to wlist all the non-terminals that use N1.
    CS-Analyze(G.S, {DEFAULT})
end

Func CS-BuildOutStates(NonTerminal N0, States S)
begin
    switch structure of N0 do
        case N0 → N1|N2:  outStates[N0] = outStates[N1] ⊔ outStates[N2];
        case N0 → N1N2:
            foreach  l1 ∈ LS do
                foreach  l2 ∈ outStates[N1][l1] do
                    outStates[N0][l1]∪ = outStates[N2][l2];
            if N2 →* ε then  outStates[N0] = outStates[N0] ⊔ outStates[N1];
        case N0 → T:   foreach  l ∈ LS do  outStates[N0][l] = O(T, l);
        case N0 → ε:   foreach  l ∈ LS do  outStates[N0][l] = O(ε, l);
end

Func CS-Analyze(NonTerminal N0, States S)
begin
    sRet = {};
    foreach l ∈ S do
        if isAnalyzed[N0][l] then  S = S − {l};
        else  isAnalyzed[N0][l] = true;
        sRet = sRet ∪ outStates[N0][l];
    if S is empty then // no more analysis to be done, return.
        return  sRet − {E};
    foreach  l ∈ S  do
        if outStates[N0][l] = {E}  then
            // error -- (N0, l)
    switch  structure of N0  do // Now analyze the components of N0
        case  N0 → N1|N2:   CS-Analyze(N1, S); CS-Analyze(N2, S);
        case  N0 → N1N2:   S1 = CS-Analyze(N1, S); CS-Analyze(N2, S1);
    return  sRet − {E};
end
```

Figure 7. Context sensitive lexical state analysis

Figure 7 presents a sketch of our context sensitive analysis. The main function `Main-CSensitive` takes the grammar ($G = (N, T, P, S)$) as input and first calls `Find-Useful-Productions` to identify all the useful productions. It follows a worklist-based approach to compute the out-states for all the non-terminals. The `CS-BuildOutStates` function is similar to that described in the context insensitive analysis (Figure 4). One main variation being the current version maintains separate set of out-states for each lexical state. Once the out-states are computed it calls the `CS-Analyze` to analyze the grammar, starting with the start non-terminal ($G.S$) and default lexical state as the in-states set ({DEFAULT}).

DOI: 10.1002/spe

`CS-Analyze`: We first check if the current non-terminal (N) has already been analyzed for the in-states S. If it has been already analyzed for all the member states in S, then we return the non-error out-states of N over all the in-states. A two dimensional boolean array (isAnalyzed) is used to remember if a production has been analyzed for a particular state; all of its elements are initialized to `false`. For a given lexical state, if the out-states of N contains only the error state $\mathcal{E}$, then it is marked as an error. A context sensitive error consists of the non-terminal and the lexical state in which the error is identified. If N has not been analyzed for a subset of input states we recursively analyze the non-terminals *used* by N. Note that, we avoid issuing warnings for any non-terminal N and lexical state l (when $\mathcal{E} \in \texttt{outStates}[N][l]$), because the source of the warning would anyway be reported as an error; thereby, we avoid too many messages. Importantly, our proposed approach catches and reports the complete set of definite errors present in the grammar.

Example: For the example grammar shown in Figure 5, which is expected to accept the set of strings {`bcbcc`, `bcbcbc`} the out-states of each non-terminal for each lexical state computed using the context sensitive analysis, along with the identified errors (note, the error is specific to a non-terminal and a lexical token) are shown in columns 5-7 of Figure 6. For example, it says that non-terminal `D` leads to an error state when it is matched in lexical state `DEF` or `LX1`. As it can be seen, the context sensitive analysis reports all the errors including those that are otherwise not reported by the context insensitive analysis.

Complexity: The complexity of the $\sqcup$ operator is $O(N)$. The complexity of `CS-BuildOutStates` function is $O(L^2)$. The while loop in `Main-CSensitive` is at most invoked $O(N \times L^2)$ times – in each iteration, the size of the `outStates` map for at least one non-terminal for at least one in-state increases by one. The `CS-Analyze` function can be called at most $O(N \times L)$ times and in each invocation the work done is bound by $O(L)$. This leads to an overall complexity of `Main-CSensitive` as $O(N \times L^4)$. In practice, the size of L is a small number and that makes it almost linear.

2.5. Generating Examples

We now discuss how we can generate counter-example strings that can be used to establish errors in a grammar. We represent the grammar as a graph, and reduce the problem of generating counter-examples, as that of computing an annotated path from the start node to the error node.

Given a context free grammar that uses tokens with lexical states, we represent it as a forest (called lexical-transition-graph), where each connected component corresponds to a different production (labeled by that non-terminal). To avoid the problem of too many edges we keep the forest sparse and omit the edges between the use of a non-terminal and the graph corresponding to its production, in our figures shown in this manuscript; such edges depict parent-child (use of a non-terminal - its corresponding production) relationship. Each connected component can be seen as a graph $G = (N, E)$, where N is the set of nodes consisting of all the non-terminals, terminals and a set of special operators Π present in the production. For the subset of grammar presented in Section 2.1, $\Pi = \{\bullet, |\}$, representing the sequencing and choice operators[†]. Such a graph admits a natural parent-child relationship – each terminal and non-terminal on the right side of a production for a non-terminal is marked as its child. Similarly, each special operator works as a parent for each non-terminal and other special operators contained with in. Each node has an attached set of in-states and out-states. Each member of the set of in-states of an operator node is connected to corresponding member(s) of each set of in- states for an operator nodes children. Similarly the set of out-states of an operator node is connected to the corresponding member(s) of each set of out-states of its children. The members in the set of in- and out-states of a token are connected as per the state transitions defined in the grammar. They represent the lexical state transitions that are taking place in the grammar.

[†]The complete JavaCC grammar syntax allows strings of the form $X*$, $X+$ and $[X]$; thus Π consists of additional operators "$*$", "$+$" and "[]".

DOI: 10.1002/spe

Given a particular context sensitive error (N_1, l), we find a path from N_1 to the *root* (start non-terminal) such that we reach N_1 with l as the in-state; this path in reverse, added with the FIRST token of N_1, can give the counter-example that leads to the error. Figure 8 presents the algorithm. The entry point `Gen-Err-String` is called with the context sensitive error details (N_1, l) as arguments, which in turn calls the `Gen-Err-Path` function to return a queue of strings that correspond to the nodes in the error path. We recursively visit the parents of the current node until we reach the graph for the *start* node (root). During the unwinding of the recursion, we store the strings corresponding to each non-terminal seen in the path (by calling `Gen-Accept-String`); these strings are stored in a Queue (*strQ*). Finally, the queue of strings are output, ending with FIRST(N_1).

Example: For the grammar shown in Figure 5, Figure 9 shows the generated lexical transition graph for two production rules `E` and `F`. The red dotted box shows that there are no "out" edges from `D` thus indicating an error in `F`. And in the graph for node `H`, the edge from the `DEFAULT` state as instate is marked as red because it leads to an error, as `C` is not defined in lexical state `DEFAULT`. Our counter-example generation routine would generate the string `bcbcc` as an example that cannot be parsed. Note that, we have deliberately skipped the box corresponding to the"•" in the graph to avoid clutter of rectangles. Also note that the red box has been dotted here to improve the visibility, the tool actually outputs an undotted red color box.

2.6. Comparing context sensitive and insensitive analysis

We now state the precision of the context sensitive and insensitive analysis.

Theorem 2.1
The context sensitive analysis identifies all the errors identified by the context insensitive analysis and may be more.

We present a sketch of the proof in Appendix A. This theorem ensures that the context sensitive analysis is more precise than the context insensitive analysis.

2.7. Finding errors after eliminating the lexical states

It can be argued that the useless production removal procedure (Figure 2) can be used to identify all the error productions identified by the context sensitive analysis, if the grammar with lexical states can be converted to an equivalent grammar with no lexical states. Such a conversion can be done by duplicating terminals and non-terminals such that each one has a unique in- and out-state; however, such a translation (from grammar with lexical states to one without) can lead to an exponential blow up. One such example is given below:

$S \rightarrow AAA \ldots A$ // *n number of them*
$A \rightarrow A_1 | A_2 | A_3 \ldots | A_n$, $\qquad A_1 \ \rightarrow a_1, A_2 \rightarrow a_2, \cdots, A_n \rightarrow a_n$

Say, we have n number of lexical states $(L_1, L_2, \ldots L_n)$, and each terminal a_i is declared as: $< L_1, L_2, \ldots L_n >$ `TOKEN : <`a_i`:` $Regex_i$`>` : L_i. Thus, each token a_i has n in-states and a unique out-state L_i. A translation as suggested above would lead to $O(n^n)$ productions, rendering the overall analysis impractical.

2.8. Reporting errors based on just lexical state specifications of the tokens

It can argued that one need not look at the grammar production rules and conclude on the erroneous nature of the grammar by only looking at the lexical state specification of the tokens. Such a naive approach may lead to too many false positives. For example, in Figure 5, just looking at the lexical state specifications, one will conclude that the string "`ac`" will lead to an error. But such a string is not even accepted by the grammar. Similarly, in Figure 5, say while keeping the lexical token specification and the specification of the non-terminals `A` and `C` intact, if we replace the rest of the grammar rules with a simple grammar rule `void S():{}{A()|C()}`, then the grammar has no errors. But a naive lexical state based analysis would conclude that

```
Func Gen-Err-String(N_1, l) begin
    Queue strQ = Gen-Err-Path(N_1, l, new Queue());
    while ¬strQ.isEmpty() do output strQ.dequeue();
    output FIRST(N_1);
end
Func Gen-Err-Path(N_1, l, strQ)
begin
    if N_1 = root then return strQ;
    if visited[N_1][l] = true then
        return null; // Do not pursue this path further
    visited[N_1][l] = true;
    foreach parent p of N_1 do
        switch type of p do
            case "•" // can have exactly two children
                if the N_1 is the right child then
                    Say N_0 is the left child;
                    S = set of in-states of N_0 for which l can be be one of the out-states;
                    foreach l_1 ∈ S do
                        Queue nstrQ = Gen-Err-Path (p, l_1, new Queue(strQ));
                        if nstrQ ≠ null then
                            nstrQ.enqueue(Gen-Accept-String (N_0, l_1, l));
                            return nstrQ;
                else // unique parent guaranteed.
                    return Gen-Err-Path(p, l, strQ);
            case "|" // unique parent guaranteed.
                return Gen-Err-Path(p, l, strQ);
            case T // terminal
                return Gen-Err-Path(p, l, strQ);
    return null;
end
Func Gen-Accept-String(N_1, l_1, l_2) begin
    switch type of N_1 do
        case "•"
            Let the production be N_1 → N_2 N_3;
            S = φ;
            foreach l ∈ LS do
                if l_2 ∈ outStates[N_3][l] then S = S ∪ {l};
            Choose an l' such that l' ∈ outStates[N_2][l_1] ∩ S;
            return Gen-Accept-String (N_2, l_1, l').concatenate(Gen-Accept-String (N_3, l', l_2));
        case "|"
            Let the production be N_1 → N_2|N_3;
            if l_2 ∈ outStates[N_2][l_1] then return Gen-Accept-String (N_2, l_1, l_2);
            return Gen-Accept-String (N_3, l_1, l_2);
        case T // terminal
            return T;
end
```

Figure 8. Generate Error String.

the grammar is erroneous. Our proposed analysis ensures that for every error flagged by our context sensitive analysis, we will find a corresponding error string.

3. IMPLEMENTATION

We have implemented our `LSA` tool using JavaCC and Java. `LSA` uses the JavaCC grammar from Sun Microsystems [3]. We extend the code generated by JTB [21] to generate an annotated

DOI: 10.1002/spe

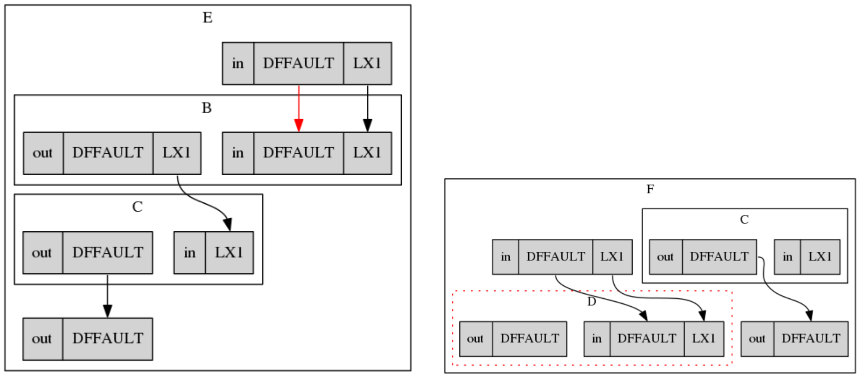

Figure 9. Part of the lexical transition graph for the counter-example shown in Figure 5.

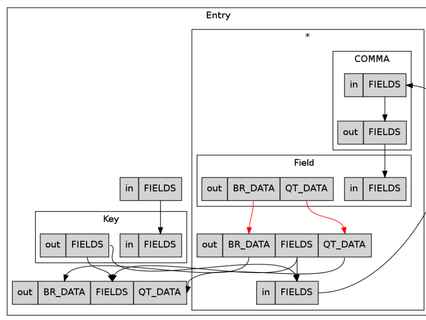

Figure 10. Part of the lexical transition graph for the example shown in Figure 1.

tree, where each node contains information required for the analyses. Further, we recreate the parse tree for efficient traversal; we call this tree the operator tree. The intermediate nodes of this tree are the operators $\rightarrow, \bullet, |, +, *, ?$, and $[]$; the terminals and non-terminals can only appear in the leaf nodes. The $\rightarrow$ node is used to represent grammar productions, and its left child is a non-terminal and right side is a production. The operators along with terminals and non-terminals are used to denote different productions. We later use this tree to generate the graph discussed in Section 2.5, where we drop the $\rightarrow$ operators and use non-terminals as intermediate nodes. Unlike our discussed grammar subset (Section 2.1), all these operators can admit any number of operands. Thus, our implementation is not limited by the grammar restrictions described in this paper. `LSA` can take as input any valid LL(k) grammar in the JavaCC format. We now discuss some implementation details of `LSA`.

3.1. Graph Generation

Given an input grammar, `LSA` performs our analyses to produce warnings and errors. Next, as described in Section 2.5, it creates a lexical transition graph for the input grammar (in DOT [12] format), along with the lexical states. This graph represents the lexical state transitions that are taking place in the grammar. We then highlight the edges (in red) that can lead to *error* states. Figure 10 shows a part of the graph generated for the motivating example shown in Figure 1. It shows that there are no edges from the out-states of `Field` (`BR_DATA` and `QT_DATA`) to the in-states of the "*" sub-production (`FIELDS`). Thus, we cannot use this production to parse more than one `Field`.

Limitations of `LSA`: JavaCC admits inlined Java code as part of the productions that can change the lexical state at runtime, by invoking a special function (called `SwitchTo`), which takes an integer argument (computed from arbitrary Java expressions) representing the target lexical state. Statically identifying the precise target lexical state in such a scenario

DOI: 10.1002/spe

Name	#lines	# lex states	Analysis time (sec)			#UP	#CI		#CS	
			UP	CI	CS		errs	warnings	errs	sources
Ldif	418	7	0.20	0.20	0.23	0	16	32	102	16
HTML	406	8	0.25	0.26	0.27	0	1	5	6	6
RTF	237	3	0.18	0.19	0.19	0	3	0	3	3
PHP	645	8	0.33	0.39	0.47	0	42	222	270	206
FM	3089	7	0.53	0.57	0.63	43	33	6	49	19
Java	1061	1	0.32	0.36	0.36	2	0	0	0	0
DefaultQuery	799	4	0.31	0.31	0.31	0	1	0	1	1
Parser	2616	9	0.45	0.47	0.55	1	6	124	155	84
ICalSyntax	528	7	0.25	0.25	0.27	0	1	16	21	16
XVCalSyntax	319	5	0.19	0.20	0.20	0	0	9	9	9

Figure 11. Evaluation. UP: useless production removal algorithm, CI: Context insensitive lexical state analysis, CS: Context sensitive lexical state analysis, #UP: Number of useless productions detected.

is undecidable in general (the problem reduces to the halting problem). We are working on techniques to model the behavior of the `SwitchTo` function conservatively by the use of standard compiler techniques (such as, global value numbering and conditional constant propagation [18]).

4. EVALUATION

We present the evaluation of our tool on a set of ten open-source JavaCC grammar files downloaded from different websites. These files can be downloaded from our website [1]. Figure 11 presents the summary of our evaluation. The size of these grammar files varied from approximately 200 lines of code to 3000 lines of code. The number of lexical states varied from one to nine. Following the suggestions of the insightful paper of George et al [13], we report the analysis time as an average over 30 runs (on a personal laptop with Intel i3 processor). The reported time includes the time it took to read the grammar files and doing the specific analysis. It can be easily seen from the figure that the running time overhead for our proposed analysis is minimal; all the analyses finish running in less than a second. The context insensitive and sensitive analyses for grammars like PHP, FM and Parser take more time compared to the UP Analysis; this is because of the comparatively increased use of the lexical states in them.

Note that the number of context insensitive errors is less than or equal to the number of context sensitive errors, which agrees with our claim in Section 2.6. To understand the nature of the error better, we also mark the source of each error. For example, say we have two productions of the form $A \to B\ C$ and $B \to D\ E$. If we cannot parse E after parsing of D, then we will report an error in the production for B, and also in the production for A, as that will also be never parsed. The B production here is called the "error source". The last column indicates the number of "error sources" found; each of these errors points to an independent error, which in turn may lead to reporting of one or more errors (in column 10). For the example grammar files, we have also generated the graphs depicting the errors therein; these graphs can be accessed from the above-mentioned website [1].

It is encouraging to see `LSA` find relevant errors in real world grammars. The reason why these grammars may be working in practical situations could be that most of the users take these grammars as a base to start with and hack it according to their needs (similar to what we ourselves did in some other projects). A tool like `LSA` can be really helpful in such a scenario as it will make validation of the correctness of the grammar easier.

Note that, currently there are no other tools that analyze grammars for errors arising due to the use of lexical states and hence we didn't have any other tools/approaches to compare against, except with the naive approach of eliminating unreachable productions. Nevertheless, the utility of `LSA` is evident from the analysis presented above.

DOI: 10.1002/spe

5. RELATED WORK

Researchers have designed grammar analyzers with many different purposes. Identifying ambiguity of context free grammars has received a fair amount of attention [14, 23, 10, 9, 24]. The ANTLR v4 plugin for IntelliJ [22] helps identify syntactic and simple semantic errors in ANTLR grammars. Similarly, there have been prior works on verifying [8] and validating parsers [16]; these focus on ensuring that the semantics of the parser matches that of the grammar. None of these papers deal with lexical states and erroneous situation arising in such a context. We believe that our formulation of the problem of identifying errors and warnings in grammars that use lexical states and our idea of generation of counter examples for the identified errors are novel.

The use of context to improve the precision of program analysis is a well-known technique and is used in many places (points-to analysis [7], escape analysis [11], alias analysis [17], data flow analysis [19], and so on). The trade-offs between context-sensitive (improved precision) and context-insensitive (faster) are well studied [18, 20]. In this paper, we extend the traditional context-sensitive and context-insensitive analysis to present two analyses that help identify errors and warnings in context free grammars (CFGs) that use tokens with lexical states. To the best of our knowledge, such an extension is novel and we believe that it opens up a new opportunity for the use of context-sensitive analysis.

6. CONCLUSION

Lexical states are useful in expressing complex control flow between the lexer and the parser in a convenient way (for example, comments in programs can be easily skipped by using lexical states). Our experience shows that even a few lexical states can make it difficult to reason about the correctness of the grammar. We discuss three techniques to automatically identify errors and warnings in JavaCC grammars that use tokens with lexical states: a naive technique to eliminate useless productions, a novel context insensitive lexical state analysis, and a novel context sensitive lexical state analysis. We have implemented these techniques as a standalone tool (named `LSA`). Besides the specific information about the errors and warning, `LSA` outputs a graph that helps reasons about the errors in a convenient manner. We have used `LSA` to analyze a few open-source JavaCC grammars to good effect. The tool can be downloaded from [5].

As a future work, we plan to extend our results to YACC grammars that use *start* conditions. The main complexity one has to handle in this scenario is that of the `BEGIN` construct, which is similar to the `SwitchTo` construct of JavaCC.

Acknowledgements: We thank Supratik Chakraborty at IIT Bombay for his insightful comments regarding the possible translation of grammar with lexical states to one without and the resulting exponential blow up (Section 2.7). This research is partially supported by the New Faculty Seed Grant, funded by IIT Madras CSE/11-12/567/NFSC/NANV, DAE research grant CSE/13-14/139/BRNS/NANV and DST Fasttrack grant CSE/13-14/140/DSTX/NANV.

REFERENCES

1. Benchmarks used for evaluating `LSA`. http://www.cse.iitm.ac.in/~krishna/lsa/benchmarks/.
2. JavaCC documentation. http://javacc.java.net/doc/docindex.html.
3. JavaCC grammar repository. http://java.net/projects/javacc/downloads/directory/contrib/grammars.
4. JavaCC website. http://javacc.java.net.
5. Link to download `LSA`. http://www.cse.iitm.ac.in/~krishna/lsa.
6. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
7. L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

DOI: 10.1002/spe

set/map	Domain	Definition
L_1		$L \cup \{\mathcal{E}\}$
R	$\subseteq (N \cup T) \times P \to \mathbb{P}(L_1)$	returns the set of lexical states in which a non-terminal N_i or T_i can be reached in a given production
$\mathcal{O}'$	$\subseteq N \cup T \times \mathbb{P}(L) \to \mathbb{P}(L)$	$\forall x \in N \cup T, S \subseteq L_1, \mathcal{O}'(x, S) = \text{igcup}_{l \in S} \mathcal{O}(x, l)$
E_s	$\subseteq P$	$\{p \mid p \in P, p$ is of the form $N_0 \to N_1 N_2$, $\mathcal{O}'(N_1, R(N_1, p)) \cap \mathcal{I}(N_2) = \phi\}$
E_i	$\subseteq P$	$\{p \mid p \in P, p$ is of the form $N_0 \to N_1 N_2$, $\mathcal{O}'(N_1, L) \cap \mathcal{I}(N_2) = \phi\}$

Figure 12. Sets and Maps used in the theorem

8. A. Barthwal and M. Norrish. Verified, executable parsing. In *ESOP*, pages 160–174, 2009.
9. B Basten and T v d Storm. AMBIDEXTER: Practical ambiguity detection. In *SCAM*, pages 101–102, 2010.
10. C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, Mar 2010.
11. Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, 1999.
12. E. R. Gansner, E. Koutsofios, and S. North. *Drawing graphs with dot*, 2010. http://www.graphviz.org/Documentation/dotguide.pdf.
13. A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76, New York, NY, USA, 2007. ACM.
14. Saul Gorn. Detection of generative ambiguities in context-free mechanical languages. *J. ACM*, 10(2):196–208, April 1963.
15. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.
16. Jacques-Henri J., F. Pottier, and X. Leroy. Validating LR(1) parsers. In *ESOP*, pages 397–416, 2012.
17. Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 329–341, New York, NY, USA, 1998. ACM.
18. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
19. V. Krishna Nandivada and Suresh Jagannathan. Dynamic state restoration using versioning exceptions. *Higher Order Symbol. Comput.*, 19(1):101–124, March 2006.
20. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
21. Jens Palsberg. JTB: Java tree builder. http://www.cs.ucla.edu/∼palsberg/jtb/.
22. Terrance Parr. Antlr v4 grammar plugin for intellij, 2014. https://github.com/antlr/intellij-plugin-v4.
23. Friedrich Wilhelm Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, 2001. http://accent.compilertools.net/Amber.html.
24. Naveneetha Vasudevan and Laurence Tratt. Search-based ambiguity detection in context-free grammars. In *ICCSW*, pages 142–148, 2012.

A. COMPARISON OF CONTEXT SENSITIVE AND INSENSITIVE ANALYSIS

Given a grammar (N, T, L, P), we define three sets and two maps in Figure 12. E_i and E_s are the sets of errors identified by context insensitive and context sensitive analysis, respectively. Note that, the map $\mathcal{O}'$ corresponds to the map `outStates` in the algorithm discussed in Figure 7. We will be using these sets and maps, in addition to the ones defined in Figure 3 to state and prove the following theorem:

Theorem A1
The context sensitive analysis identifies all the errors identified by the context insensitive analysis and possibly more. Or in other words, $E_s \supseteq E_i$.

Proof

Notation: Considering the grammar subset described in this paper (Section 2.1), the only production in which a context insensitive error can be encountered is of the form $N_0 \to N_1 N_2$. Say $p = N_0 \to N_1 N_2$ is one such production. We will be using $\mathcal{R}$ as a short form for $R(N_1, p)$.

DOI: 10.1002/spe

We will define the following two sets.

$$\mathcal{S}_1 = \mathcal{O}'(N_1, \mathcal{R}) \cap \mathcal{I}(N_2)$$
$$\mathcal{S}_2 = \mathcal{O}'(N_1, \mathcal{L}) \cap \mathcal{I}(N_2)$$

Sets $\mathcal{S}_1$ and $\mathcal{S}_2$ contain the states in which N_2 can be parsed after N_1, in production p, while doing context sensitive and insensitive analysis, respectively.
We have,

$$\mathcal{S}_1 = \phi \leftrightarrow p \in E_s \tag{1}$$
$$\mathcal{S}_2 = \phi \leftrightarrow p \in E_i \tag{2}$$
$$\mathcal{S}_1 \subseteq \mathcal{S}_2 \tag{3}$$

From (3), we have

$$\begin{aligned} & S_2 = \phi \ \rightarrow S_1 = \phi \\ \rightarrow \quad & p \in E_i \rightarrow p \in E_s \quad \texttt{// From (1), and (2)} \\ \leftrightarrow \quad & E_s \supseteq E_i \quad \Box \end{aligned}$$

$\Box$

DOI: 10.1002/spe