On Testing for Insert-Correctability In Context-Free Grammars

by

Charles N. Fischer
Bernard A. Dion

Computer Sciences Technical Report #355

June 1979

1. Introduction

Various context-free error recovery and error correction algorithms employ a wide variety of repair operations. Such operations as insertion of a symbol, deletion of a symbol, replacement of one symbol with another, transposition of two adjacent symbols and condensation of a sequence of symbols to a new symbol are commonly employed ([1],[5],[7],[8],[10],[11],[12]). Of these operations, insertion and deletion are the most fundamental; that is, all correction and recovery algorithms employ them.

However, recent research ([6],[7]) has shown that for a class of context-free languages termed insert-correctable, only insertion operations are needed to correct any input -- deletions need never be employed. Since an error correction or recovery algorithm can often be simplified by restricting it to only insertion operations, a means of deciding whether a context-free grammar generates an insert-correctable language is of interest. In [7] a method of testing whether an LL(1) grammar generated an insert-correctable language was presented. Here we present a more general algorithm which can test whether an LR(1) grammar generates an insert-correctable language. Since LR(1) grammars subsume virtually all context-free grammars used in practice (SLR(1), LL(1), LALR(1), Simple Precedence, etc.) [3], the method presented is very broadly applicable. Indeed, since every deterministic context-free language can be generated by an LR(1) grammar (in fact an LR(0) grammar if endmarkers are provided) ([3], Theorem 8.16), this method is applicable to any deterministic parsing technique independent of the amount of lookahead used.

We then consider the general case of testing whether an arbitrary context-free grammar generates an insert-correctable language. This problem is shown to be recursively undecidable. In what follows, the reader is assumed to be familiar with the basic notions of context-free grammars and parsing. An excellent introduction to these ideas may be found in [2].

2. Testing LR(1) Grammars for Insert-Correctability

In this section, we will present an algorithm which tests if an LR(1) grammar, G, is insert-correctable. Without loss of generality, we will limit our attention to augmented CFG's, in which all terminal strings are terminated by an endmarker, \$. That is, all terminal strings may be written as $x\$$, where $x \in V_T^*$ and $\$ \notin V_T$. Let $\hat{V}_T = V_T \cup \{\$\}$ be the augmented terminal vocabulary. Similarly, $V = V_N \cup V_T$ and $\hat{V} = V_N \cup \hat{V}_T$. Further, let ϵ denote the null or empty string.

A context-free language, L, is insert-correctable iff $\forall x \in V_T^*$ and $\forall a \in \hat{V}_T$ such that $x\ldots \in L$ and $xa\ldots \notin L$, $\exists\, y \in V_T^+$ such that $xya\ldots \in L$. Clearly, in an insert-correctable language any syntax error $(xa\ldots)$ can be corrected into a prefix of a valid terminal string $(xya\ldots)$ by the insertion of a suitable terminal string. Multiple errors can be corrected by doing an insertion each time a syntax error is discovered. A context-free grammar is insert-correctable iff its language is insert-correctable.

We shall test LR(1) grammars for insert-correctability by extending the LR(1) test presented in ([2] sec. 5.2, [4] Ch. 6). A review of terminology is therefore in order. An LR(1) item is a pair $[A \rightarrow \alpha . \beta, u]$ where $A \rightarrow \alpha\beta$ is a production and u is in $\hat{V}_T$.

For $\alpha \in \hat{V}^*$ First(α) = $\{a \in \hat{V}_T \mid \alpha \overset{*}{==>} a \ldots\}$. A string γ is a *viable prefix* iff there exists a derivation sequence $S \underset{r}{\overset{*}{==>}} \alpha A w \underset{r}{==>} \alpha\beta w$ and γ is a prefix of $\alpha\beta$. Let $\alpha\beta_1 \cdot \beta_2 w$ denote a right sentential form for which $\alpha\beta_1$ is selected as a viable prefix. $[A \rightarrow \alpha.\beta,u]$ is *valid* for the right sentential form $\delta\alpha \cdot \beta w$ if there is a derivation $S \underset{r}{\overset{*}{==>}} \delta A w \underset{r}{==>} \delta\alpha\beta w$ and either $u \in$ First(w) or ($w=\epsilon$ and $u=\$$). $[A \rightarrow \alpha.\beta,u]$ is *valid* for a viable prefix $\rho\alpha$ iff it is valid for some right sentential form $\rho\alpha \cdot \beta v$. LR(1) operates by creating a collection of distinct *item sets*, each of which contains a number of LR(1) items. Each item set I can be partitioned into two disjoint subsets: Basis(I), which contains the *basis items* of I, and Cl(I), which contains the *closure items* of I. I(ϵ) is the *initial* item set and contains a single basis item $[S' \rightarrow .\alpha\$,\$]$ where $S' \rightarrow \alpha\$$ is the augmenting production which generates the endmarker. For any $\alpha \in \hat{V}^*$, $X \in \hat{V}$, I(αX) is computed from I(α). Basis(I(αX)) = $\{[A \rightarrow \alpha X.\beta,u] \mid [A \rightarrow \alpha.X\beta,u] \in I(\alpha)\}$. For any item set I, Cl(I) = $\{[C \rightarrow .\gamma,v] \mid C \rightarrow \gamma$ is a production, $[A \rightarrow \alpha.C\beta,u] \in I$ and $v \in \text{First}(\beta u)\}$. By construction, I($\alpha$) contains exactly those items valid for α.

To construct our extended LR(1) item sets, we will add a *third component* to each item. This component, a function $t:\hat{V}_T \rightarrow \{\text{True},\text{False}\}$, will be, in effect, an extended lookahead function in which t(b) = True iff b can appear *somewhere* in the extended lookahead of the item in question. More precisely, given $i = [A \rightarrow \beta_1 \cdot \beta_2,u,t]$ in item set I , t will be defined so as to satisfy the following condition (call it condition (*)):

For any $a \in \hat{V}_T$, any viable prefix $\alpha\beta_1$ and any item $i = [A \to \beta_1 \cdot \beta_2, u, t] \in I(\alpha\beta_1)$, $t(a)$ = true iff i is valid for some right sentential form $\alpha\beta_1 \cdot \beta_2 w$ where $w = \ldots a \ldots$.

We now present an algorithm to compute extended LR(1) item sets, followed by a lemma that proves that the t-functions which are computed satisfy condition (*).

<u>Algorithm</u> <u>2.1</u>. Extended LR(1) item set computation.

[1] $\mathrm{Basis}(I(\epsilon)) = \{[S \to .\alpha\$, \$, t]\}$ where

$\forall a \in \hat{V}_T$ $t(a)$ = False

[2] For $\alpha \in V^*, X \in \hat{V}$

$\mathrm{Basis}(I(\alpha X)) =$

$\{[A \to \alpha X.\beta, u, t] \mid [A \to \alpha.X\beta, u, t] \in I(\alpha)\}$

[3] For $\alpha \in \hat{V}^*$, $I(\alpha) = \mathrm{Basis}(I(\alpha)) \cup \mathrm{Cl}(I(\alpha))$

where $\mathrm{Cl}(I(\alpha))$ is the smallest set such that

If $[A \to \gamma.C\beta, u, t] \in I(\alpha)$, $C \to \delta$ is a production and $v \in \mathrm{First}(\beta u)$

Then $[C \to .\delta, v, \hat{t}] \in \mathrm{Cl}(I(\alpha))$

where $\forall a \in \hat{V}_T$ $\hat{t}(a)$ = true if $t(a)$ = true or if $\beta \overset{*}{\Longrightarrow} \ldots a \ldots$

|X|

As an example, consider G_1 which generates the skeletal block structure of Algol 60.

G_1: PROG→BLOCK \$

BLOCK→BEGIN STMTLIST END

STMTLIST→STMTLIST ; STMT

STMTLIST→STMT

STMT→BLOCK

STMT→S

Consider first I(ϵ). We represent the t-table by a sequence of t's and f's , representing, in order, t(BEGIN), t(END),t(;),t(S),t($). Basis(I(ϵ)) = {[PROG→.BLOCK $,$,(fffff)]}. From step [3] we then obtain Cl(I(ϵ)) = {[BLOCK→.BEGIN STMTLIST END,$,(fffft)]}

Continuing, Basis(I(BEGIN)) =

{[BLOCK→BEGIN.STMTLIST END,$,(fffft)]}

Further, Cl(I(BEGIN)) =

{[STMTLIST→.STMTLIST;STMT,END,(ftfft)],

[STMTLIST→.STMT,END,(ftfft)],

[STMTLIST→.STMTLIST;STMT,;,(ttttt)],

[STMTLIST→.STMT,;,(ttttt)],

[STMT→.BLOCK,END,(ftfft)],

[STMT→.S,END,(ftfft)],

[STMT→.BLOCK,;,(ttttt)],

[STMT→.S,;,(ttttt)],

[BLOCK→.BEGIN STMTLIST END,END,(ftfft)],

[BLOCK→.BEGIN STMTLIST END,;,(ttttt)]}

The reader is invited to verify that the t-table does, in fact, represent an extended lookahead, telling whether a given terminal can ever appear in the remaining input of an item. We now establish that condition (*) does correctly characterize the items created by Algorithm 2.1.

<u>Lemma 2.2</u>

Condition (*) holds for all extended LR(1) items created by Algorithm 2.1.

<u>Proof</u>

An induction on the order in which items are created. (*) trivially holds for the sole basis item of $I(\epsilon)$. If $I(\alpha X)$ is created from $I(\alpha)$ then (*) holds for each item $[A \to \gamma X.\delta, u, t] \in Basis(I(\alpha X))$ because it holds (by induction) for $[A \to \gamma.X\delta, u, t] \in I(\alpha)$. Now consider closure items.

Assume $j = [C \to .\delta, v, \hat{t}]$ is created from $i = [A \to \gamma.C\beta, u, t] \in I(\alpha)$.

(Only if part): $\hat{t}(a)$ = true ==> $t(a)$ = true or $\beta \overset{*}{\Longrightarrow} \ldots a \ldots$. If $t(a)$ = true then by (*) and induction hypothesis, i is valid for $\rho\gamma.C\beta w$ where $\rho\gamma = \alpha$ and $w = \ldots a \ldots$. But then j is valid for some $\rho\gamma.\delta xw$ where $\beta \overset{*}{\Longrightarrow} x, w = \ldots a \ldots$ and $v \in First(xw)$. Similarly, if $\beta \overset{*}{\Longrightarrow} \ldots a \ldots$ then again j is valid for some $\rho\gamma.\delta x'w'$ where $x' = \ldots a \ldots$.

(If part) Assume j is valid for $\sigma.\delta w$ where $w = \ldots a \ldots$. Since j was created from i, it must be that i is valid for some $\rho\gamma.C\beta z$ where $\sigma = \rho\gamma$, $w = xz$, $\beta \overset{*}{\Longrightarrow} x$ and $v \in First(xz)$. Either $x = \ldots a \ldots$ or $z = \ldots a \ldots$. In the former case, $\beta \overset{*}{\Longrightarrow} \ldots a \ldots$ and $\hat{t}(a)$ = true. In the latter case, by (*) and the induction hypothesis, $t(a)$ = true ==> $\hat{t}(a)$ = true.

|X|

Let us call an item set I a <u>shift set</u> iff $I = I(\epsilon)$ or $I = I(\alpha a)$ for $\alpha \in V^*, a \in V_T$. Because an LR(1) parser never makes a move when an invalid symbol is the lookahead, all syntax errors are detected when a shift set is valid. That is, syntax errors are always detected immediately after the last valid input (if any) has been shifted. Call an item set <u>safe</u> iff $\forall a \in \hat{V}_T$, $\exists$ $i = [A \to \alpha.\beta,u,t] \in \mathrm{Basis}(I)$ such that $\beta \overset{*}{\Longrightarrow} ...a...$ or $t(a)$ = true. If I is safe then a syntax error detected when I is valid can always be corrected by a suitable insertion. This observation can be formalized in the following theorem which characterizes insert-correctable LR(1) grammars.

<u>Theorem 2.3</u>

An augmented LR(1) grammar G is insert-correctable iff all distinct shift sets created by Algorithm 2.1 are safe.

<u>Proof</u>

(If part): Assume x has been read and reduced to γ when a syntax error involving a as the lookahead is detected. As noted above, all syntax errors are detected when shift sets are valid; that is, just after the last valid input symbol (if any) has been shifted. Thus $I(\gamma)$, the currently valid item set, must be a shift set. Since $I(\gamma)$ is safe there must exist $i = [A \to \alpha.\beta,u,t] \in \mathrm{Basis}(I(\gamma))$ for which either $\beta \overset{*}{\Longrightarrow} ya...$ $(y \in V_T^{\,*})$ or $t(a)$ = true. In the former case $xya... \in L(G)$. In the latter case, by (*), $S \underset{r}{\overset{*}{\Longrightarrow}} \delta\alpha\cdot\beta w$ where $\delta\alpha = \gamma$ and $w = ...a...$. Thus $S \overset{*}{\Longrightarrow} \gamma\beta w \overset{*}{\Longrightarrow} x\beta w = x...a...$.

(Only if part): Assume a shift set $I(\beta)$ is not safe because of $b \in \hat{V}_T$ and that $\beta \overset{*}{\Longrightarrow} x$. For each $i = [A \to \gamma.\delta,u,t] \in \mathrm{Basis}(I(\beta))$, $\delta \overset{*}{\not\Longrightarrow} ...b...$ and $t(b)$ = false. Thus while attempting to parse xb... a syntax error must be detected in $I(\beta)$ after reducing x to β. Since $t(b)$ = false, by (*) no right sentential form $\alpha\gamma\cdot\delta w$ for which w = ...b... can exist. Also $\delta =/=>^*$...b... . Therefore i can never participate in any parsing move sequence which will allow b to be accepted. But neither can any other item in Basis($I(\beta)$). Thus $S =/=>^*$x...b...

|X|

Theorem 2.3 gives us an effective method of testing whether an LR(1) grammar is insert-correctable. We compute the finite set of distinct item sets via Algorithm 2.1. We then test whether each shift set is safe. However this approach is not especially attractive as a large number of item sets may need to be created and tested. Indeed, LR(1) parsers are known to create thousands of distinct item sets for grammars used to define programming languages such as Algol 60. Clearly a means of limiting the number of item sets which need to be created and tested is needed. A way of doing this follows from the fact that LR(1) parsers have the <u>valid prefix property</u> ([4] p.391). That is, if $x \in V_T^*$ is accepted by an LR(1) parser then there exists $y \in V_T^*$ such that $xy\$ \in L(G)$. We can use this property as follows. If G is insert-correctable then for any error situation ($x... \in L(G), xa... \notin L(G)$) we can do a correction in two steps. First insert $y \in V_T^*$ such that $xy\$ \in L(G)$, then insert $z \in V_T^*$ such that $xyza... \in L(G)$. Similarly if G is not insert-correctable, then for some error situation ($x... \in L(G), \forall v \in V_T^*$ $xva... \notin L(G)$) we can again insert $y \in V_T^*$ such that $xy\$ \in L(G)$. But

this time $\nexists z \in V_T^*$ such that $xyza\ldots \in L(G)$ (otherwise G would be insert-correctable). Thus we can restrict our attention to parsing situations (and item sets) for which \$ might be the next input symbol. Call an item set I <u>\$-compatible</u> iff $\exists i = [A \rightarrow \alpha . \beta, u, t] \in I$ such that First(β) = \$ or ($\beta = \epsilon$ and u=\$). I is \$-compatible iff \$ can be shifted in I or \$ is a valid lookahead for some configuration $A \rightarrow \alpha .$ in I . We can now state and prove the following:

<u>Theorem 2.4</u>

An augmented LR(1) grammar G is insert-correctable iff all distinct \$-compatible shift sets created by Algorithm 2.1 are safe.

<u>Proof</u>

(Only if part): Follows immediately from Theorem 2.3.

(If part): As in the proof of Theorem 2.3 assume x has been read and reduced to γ when a syntax error involving a as a lookahead is discovered. I(γ) must be a shift set. If I(γ) is not \$-compatible then (by the valid prefix property) there exists a $y \in V_T^+$ such that xy can be reduced to $\hat{\gamma}$ and I($\hat{\gamma}$) is a \$-compatible shift set. It may be that a is a valid lookahead in I($\hat{\gamma}$). Otherwise by the same arguments used in Theorem 2.3, since I($\hat{\gamma}$) is safe $\exists z \in V_T^+$ such that $xyza\ldots \in L(G)$.

|X|

We now need a means of generating all reachable \$-compatible shift sets without the overhead of generating a large number of extraneous item sets. This can be done by observing that a

\$-compatible item set I must have an item $i = [A \rightarrow \alpha.\beta,\$,t] \in Basis(I)$. Further an item set $I(\alpha X)$ can have $[B \rightarrow \beta.\gamma,\$,t] \in Basis(I(\alpha X))$ only if there exists an item $i = [C \rightarrow \delta.\rho,\$,\hat{t}] \in Basis(I(\alpha))$. That is, items with a lookahead of \$ are always created from other items with \$-lookaheads and ultimately all are propagated from $Basis(I(\epsilon))$. Thus starting with $I(\epsilon)$, we need only create, and test, those item sets I which have an item $[A \rightarrow \alpha.\beta,\$,t] \in Basis(I)$. This leads to the following algorithm.

Algorithm 2.5 Test if an LR(1) grammar is insert-correctable

[1] Create $I(\epsilon)$ via Algorithm 2.1.

~~If $I(\epsilon)$ is \$-compatible and not safe~~

Then Return ('Not Insert Correctable')

Else Insert $I(\epsilon)$ as unmarked into an initially empty item set collection Z.

[2] While Z contains unmarked item sets Do

[A] Select and mark an unmarked item set $I(\alpha)$ from Z

[B] For each $X \in \hat{V}$ Do

[i] Compute $Basis(I(\alpha X))$ via Algorithm 2.1.

[ii] If an item $[A \rightarrow \alpha.\beta,\$,t] \in Basis(I(\alpha X))$

Then

(a) Compute $I(\alpha X)$ via Algorithm 2.1.

(b) If $I(\alpha X) \notin Z$

Then If $I(\alpha X)$ is a \$-compatible shift set and not safe

Then Return ('Not Insert Correctable')

Else Insert $I(\alpha X)$ an unmarked into Z

```
            END{For}
        END{While}
    [3] Return ('Insert Correctable')
```

|X|

<u>Theorem 2.6</u>

Algorithm 2.5 correctly tests LR(1) grammars for insert-correctability.

<u>Proof</u>

Algorithm 2.5 considers, in turn, all reachable item sets which have a \$-lookahead in a basis item. As noted above this guarantees that all \$-compatible item sets are considered and by Theorem 2.4 testing only these item sets is sufficient to determine insert-correctability.

|X|

Algorithm 2.5 is attractive in that it generates and tests only a small fraction of all the item sets which Algorithm 2.1 can create. As an example, reconsider G_1. I(ϵ) is first computed (see above) but is not \$-compatible. From I(ϵ), I(BLOCK) and I(BEGIN) need to be considered. I(BLOCK) = {[PROG→BLOCK.\$,\$,(fffff)]} is \$-compatible but is not a shift set. Its sole successor, I(BLOCK \$) = {[PROG→BLOCK \$.,\$,(fffff)]} is also not a shift set (since $\$ \notin V_T$). We then consider I(BEGIN) which was computed earlier. I(BEGIN) is not \$-compatible. Only one successor, I(BEGIN STMTLIST) =

{[BLOCK→BEGIN STMTLIST.END,\$,(ffft)],

[STMTLIST→STMTLIST.;STMT,END,(ftfft)],

[STMTLIST→STMTLIST.;STMT,;,(ttttt)]}

has a \$-lookahead in a basis item. This set is not a shift set. Again, only one successor, I(BEGIN STMTLIST END) = {[BLOCK→BEGIN STMTLIST END.,\$,(fffft)]}, has a \$-lookahead in a basis item. This set is a \$-compatible shift set. Further it is not safe since, e.g., $\$ \overset{*}{=/=>}$ BEGIN and t(BEGIN) = False. Thus $L(G_1)$ is not insert-correctable. The reason $L(G_1)$ must be rejected is obvious from I(BEGIN STMTLIST END) -- once the outermost BEGIN-END pair is matched, no more symbols in V_T can be read. Indeed if a full grammar for Algol 60 is tested, Algorithm 2.5 will test essentialy the same item sets as it does for G_1. This is because the additional structure is enclosed within the BEGIN-END delimiter and thus is shielded for the \$-lookahead Algorithm 2.5 concentrates on.

The interested reader is invited to verify that the following modification of G_1 is in fact insert-correctable.

G_1': PROG→BLOCKLIST \$

BLOCKLIST→BLOCKLIST;BLOCK

BLOCKLIST→BLOCK

BLOCK→BEGIN STMTLIST END

STMTLIST→STMTLIST;STMT

STMTLIST→STMT

STMT→BLOCK

STMT→S

If we extend this structural modification to Algol 60, so that programs are composed of a sequence of blocks rather than a single block, then this slightly extended Algol is also insert-correctable. This suggests strongly that insert-correctable context-free languages are of practical interest and could actually be used to simplify error correction or recovery algorithms.

Because we never actually use the item sets produced by Algorithms 2.1 and 2.5 to do parsing, there is a temptation to try to use Algorithm 2.5 to test non-LR(1) grammars. This fails because Theorem 2.3 depends crucially on the fact that an item set contains <u>all</u> the items which are valid at a given point in a parse. Item sets created for non-LR(1) grammars don't always contain all valid items. Consider G_2 which generates $\{a\}^*\$$, an obviously insert-correctable language.

$$
\begin{aligned}
G_2\colon\ & S \rightarrow S1\ \$ \\
& S1 \rightarrow a \mid S2 \\
& S2 \rightarrow S2\ a \mid \epsilon
\end{aligned}
$$

Now $I(a) = \{[S1 \rightarrow a., \$, (ft)]\}$. This is a \$-compatible shift set which is unsafe since $t(a) = \text{false}$. Thus Algorithm 2.5 would incorrectly label $L(G_2)$ as not insert-correctable. The problem of course is that since G_2 is ambiguous, an a could also be generated from S2 a and I(S2 a) is safe.

We might try to extend Algorithm 2.5 somehow so that all context-free languages can be handled. As we shall show in the next section no such extension is possible -- the problem of testing whether an arbitrary context-free language is insert-correctable is <u>undecidable</u>.

3. Testing Context-Free Languages for Insert-Correctability

In the following lemma, we show that if we could test an arbitrary context-free language (CFL) for insert-correctability, then we could test for arbitrary CFL's, L_1 and L_2 whether $L_1 \subseteq L_2$. Without loss of generality, we assume L_1 and L_2 are over the same vocabulary, V_T.

Lemma 3.1

Let L_1 and L_2 be arbitrary CFL's and assume $\#, \$ \notin V_T$. Let $L_A = L_1\{\#\$\}$, $L_B = L_2\{\#\}(V_T \cup \{\#\})^*\{\$\}$ and $L_3 = L_A \cup L_B$.

Then $L_1 \subseteq L_2$ iff L_3 is insert-correctable.

Proof

$L_1 \subseteq L_2$ iff L_3 is IC (insert-correctable) is equivalent to NOT$(L_1 \subseteq L_2)$ iff L_3 is not IC which is equivalent to $L_1 - L_2 \neq \emptyset$ iff L_3 is not IC.

1. $(L_1 - L_2 \neq \emptyset \Longrightarrow L_3$ is not IC$)$: Let $\tilde{V}_T = V_T \cup \{\#\}$ and let $x \in (L_1 - L_2)$.

   Now $x\#\$ \in L_A$ since $x \in L_1$.

   But $x\#\ldots \notin L_B$ since $x \notin L_2$. Thus

   $x\#\ldots \in L_3$ but $x\#\# \notin L_3$. Further

   $\nexists y \in \tilde{V}_T^{+}$ such that $x\#y\#\ldots \in L_3 \Longrightarrow$

   L_3 is not IC.

2. $(L_3$ not IC$\Longrightarrow L_1 - L_2 \neq \emptyset)$:

   Note that $L_1 - L_2 \neq \emptyset$ iff $L_1\{\#\} - L_2\{\#\} \neq \emptyset$.

   Since L_3 is not IC, assume $x\ldots \in L_3$,

   $xa\ldots \notin L_3$ and $\nexists y$ such that $xya\ldots \in L_3$

   $(x, y \in \tilde{V}_T^{*}, a \in \tilde{V}_T \cup \{\$\})$. Now $x\ldots \in L_3$

$\Longrightarrow x\ldots \in L_A$ or $x\ldots \in L_B$.

If $x\ldots \in L_B$ then $x\ldots \in L_2\{\#\}$ (Otherwise

$x = x_1x_2$ where $x_1 \in L_2\{\#\}$. But

$x_2a\ldots \in (V_T \mathbf{U}\{\#\})^*\$$ for any a and

thus $x_1x_2a\ldots \in L_B$, a contradiction).

Now $x\ldots \in L_2\{\#\} \Longrightarrow \exists y$ such that $xy \in L_2\{\#\}$

$\Longrightarrow xya\ldots \in L_B$, a contradiction.

It must then be that $x\ldots \in L_A \Longrightarrow \exists y$ such that

$xy \in L_A = L_1\{\#\}$. But $xy \notin L_2\{\#\}$ since

(from above) $x\ldots \in L_2\{\#\}$ leads to a contradiction.

Thus $L_1\{\#\} - L_2\{\#\} \neq \emptyset$.

|X|

We can now establish our desired result.

Theorem 3.2

If is undecidable if an arbitrary CFL, L is insert-correctable.

Proof

If this were decidable then, by Lemma 3.1, we could decide for arbitrary CFL's, L_1 and L_2 whether $L_1 \subseteq L_2$. But this problem is known to be undecidable ([4] p. 230).

|X|

It is interesting to note that insert-correctability testing has decidability results analogous to that of a similar problem - testing if a CFL generates V_T*. In both cases, the problem is solvable for deterministic CFL's but undecidable for arbitrary

CFL's.

4. Conclusion

A simple and efficient means of testing insert-correctability for LR(1) grammars has been presented. Since LR(1) grammars subsume virtually all common grammar classes (SLR(1), LALR(1), Simple Precedence, etc.), the technique can be readily used to test those CFG's used in practice. Further, since all deterministic CFL's have an LR(1) grammar, all such languages can be tested for insert-correctability. The general problem of testing an arbitrary CFL for insert-correctability has been shown to be undecidable.

Because insert-correctable languages allow for very simple "insertion-only" error correction and recovery algorithms, the technique presented is of practical interest. This is especially true in the case of error-recovery techniques, where simple and efficient methods which allow a parser to be restarted after any syntax error are required. Certainly the ability to decide which error repair operations are necessary and which are optional is fundamental when dealing with syntax errors. The insert-correctability test presented above can therefore be viewed as a basic (and most useful) tool in designing syntactic error-handling routines.

References

1. Aho, A.V., Peterson, T.G.: A minimum distance error-correcting parser for context-free languages. SIAM Journal of Computing 1,4, 305-312 (1972).

2. Aho, A.V., Ullman, J.D.: The theory of parsing, translation and compiling, Vol. 1. Englewood Cliffs, N.J.: Prentice-Hall 1972.

3. Aho, A.V., Ullman, J.D.: The theory of parsing, translation and compiling, Vol. 2. Englewood Cliffs, N.J.: Prentice-Hall 1973.

4. Aho, A.V., Ullman, J.D.: Principles of compiler design. Reading, Mass.: Addison-Wesley 1977.

5. Dion, B.A.: Locally least-cost error correctors for context-free and context-sensitive parsers. University of Wisconsin, Ph.D. Thesis (December 1978).

6. Dion, B.A., Fischer, C.N.: An insertion-only error corrector for LR(1), LALR(1), SLR(1) parsers. Computer Sciences Department, University of Wisconsin, Report No. 315 (February 1978).

7. Fischer, C.N., Milton, D.R., Quiring, S.B.: Efficient LL(1) error correction and recovery using only insertions. In: Proc. 4th ACM Symposium of Principles of Programming Languages 1977. To appear in Acta Informatica.

8. Graham, S.L., Rhodes, S.P.: Practical syntax error recovery. Comm. ACM 18, 639-650 (1975).

9. Hopcroft, J.E., Ullman, J.D.: Formal languages and their relation to automata. Reading, Mass.: Addison-Wesley 1969.

10. Penello, T.J., DeRemer, F.L.: A forward move algorithm for LR error recovery. In: Proc. 5th ACM Sym. on Principles of Programming Languages 1978.

11. Poplawski, D.A.: Error recovery for extended LL-Regular parsers. Purdue University, Ph.D. Thesis (August 1978).

12. Tai, K.C.: Syntactic error correction in programming languages. IEEE Trans. on Software Eng. SE-4,5, 414-425 (1978).