## Recursive descent parsing

First set:

Let there be a production

A →α

then First( α ) is the set of tokens that appear as the first token in the strings generated from α

For example :

First(simple) = {integer, char, num}

First(num dotdot num) = {num}

**Recursive descent parsing is a top down method of syntax analysis in which a set of recursive procedures are executed to process the input. A procedure is associated with each non-terminal of the grammar. Thus, a recursive descent parser is a top-down parser built from a set of mutually-recursive procedures or a non-recursive equivalent where each such procedure usually implements one of the production rules of the grammar.**

**For example, consider the grammar,**

**type →simple | ↑id | array [simple] of type**

**simple →integer | char | num dot dot num**

First ( a ) **is the set of terminals that begin the strings derived from a . If a derives e then e too is in First ( a ). This set is called the first set of the symbol a . Therefore,**

**First (simple) = {integer, char , num}**

**First (num dot dot num) = {num}**

**First (type) = {integer, char, num, ↑, array}**

## Define a procedure for each non terminal

```
procedure type;
  if lookahead in {integer, char, num}
    then simple
    else if lookahead = ↑
           then begin match(↑ );
                      match(id)
                end
           else if lookahead = array
                  then begin match(array);
                             match( [ );
                             simple;
                             match( ] );
                             match(of);
                             type
                       end
                  else error;
```

**Procedure for each of the non-terminal is shown.**

```
procedure simple;
  if lookahead = integer
    then match(integer)
    else if lookahead = char
           then match(char)
           else if lookahead = num
                  then begin match(num);
                             match(dotdot);
                             match(num)
                       end
                  else
                       error;

procedure match(t:token);
    if lookahead = t
        then lookahead = next token
        else error;
```

**Apart from a procedure for each non-terminal we also needed an additional procedure named *match* . *match* advances to the next input token if its argument t matches the lookahead symbol.**

**Left recursion**

. A top down parser with production A →A α may loop forever

. From the grammar A →A α | b left recursion may be eliminated by transforming the grammar to

A →b R

R →α R | ε

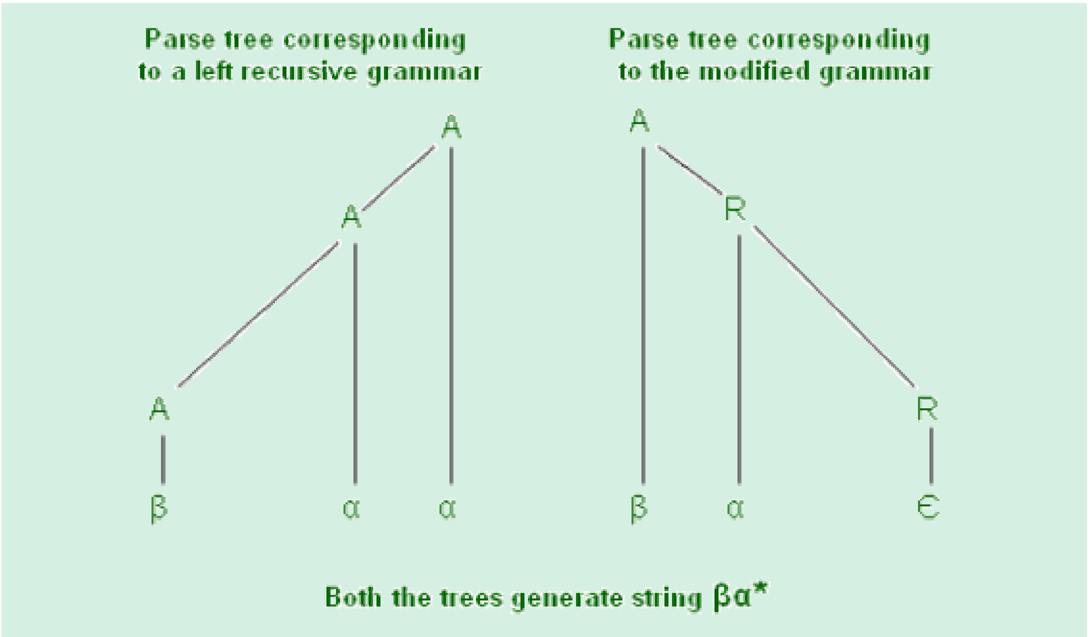
**Left recursion is an issue of concern in top down parsers. A grammar is left-recursive if we can find some non-terminal A which will eventually derive a sentential form with itself as the left-symbol. In other words, a grammar is *left recursive* if it has a non terminal A such that there is a derivation**

**A →+ A a for some string a . These derivations may lead to an infinite loop. Removal of left recursion:**

**For the grammar A →A a | ß , left recursion can be eliminated by transforming the original grammar as:**

**A →ß R**

**R →a R | ε**


Parse tree corresponding to a left recursive grammar
Parse tree corresponding to the modified grammar
A
A
A
β
α
α
A
R
R
β
α
ϵ
Both the trees generate string βα*

This slide shows the parse trees corresponding to the string ßa * using the original grammar (with left factoring) and the modified grammar (without left factoring).

**Example**

. Consider grammar for arithmetic expressions

E →E + T | T

T →T * F | F

F → ( E ) | id

. After removal of left recursion the grammar becomes

E →T E'

E' →+ T E' | ε

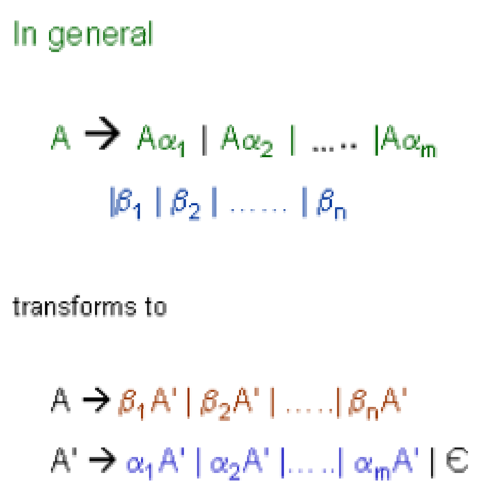
T →F T'

T' →* F T' | ε

F → ( E ) | id

**As another example, a grammar having left recursion and its modified version with left recursion removed has been shown.**

### Removal of left recursion



In general

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \mid A\alpha_m$$
$$\mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$

transforms to

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots\ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots\ldots \mid \alpha_m A' \mid \epsilon$$

The general algorithm to remove the left recursion follows. Several improvements to this method have been made. For each rule of the form

A → A **a**$_1$ | A **a**$_2$ | ... | A **a** $_m$ | β $_1$ |β $_2$ | .. |β $_n$

Where:

. **A** is a left-recursive non-terminal.

. **a** is a sequence of non-terminals and terminals that is not null ( a≠ε ).

. **ß** is a sequence of non-terminals and terminals that does not start with **A** .

Replace the A-production by the production:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$

And create a new non-terminal

$A' \rightarrow \mathbf{a}_1 A' \mid \mathbf{a}_2 A' \mid ... \mid \mathbf{a}_m A' \mid \varepsilon$

This newly created symbol is often called the "tail", or the "rest".

## Left recursion hidden due to many productions

.Left recursion may also be introduced by two or more grammar rules. For example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

there is a left recursion because

$S \rightarrow Aa \rightarrow Sda$

. In such cases, left recursion is removed systematically

- Starting from the first rule and replacing all the occurrences of the first non terminal symbol

- Removing left recursion from the modified grammar

What we saw earlier was an example of immediate left recursion but there may be subtle cases where left recursion occurs involving two or more productions. For example in the grammar

$S \rightarrow A\ a \mid b$

$A \rightarrow \quad A\ c \mid S\ d \mid e$

there is a left recursion because

$S \rightarrow A\ a \rightarrow S\ d\ a$

More generally, for the non-terminals *A* 0, *A* 1,..., *An* , indirect left recursion can be defined as being of the form:

$A_n \rightarrow A_1\ a_1 \mid .$

$A_1 \rightarrow A_2\ a_2 \mid$

. .

$A_n \to A_n a_{(n+1)}$ | .

Where $a_1$, $a_2$ ,..., $a_n$ are sequences of non-terminals and terminals.

Following algorithm may be used for removal of left recursion in general case:

*Input :* Grammar G with no cycles or e -productions.

*Output:* An equivalent grammar with no left recursion .

*Algorithm:*

Arrange the non-terminals in some order $A_1$ , $A_2$ , $A_3$ ..... $A_n$ .

for i := 1 to n do begin

replace each production of the form $A_i \to A_{j\gamma}$

by the productions $A\ i \to d_{1\ \gamma} \mid d_{2\ \gamma} \mid ........... \mid d_{k\gamma}$

where $A_j \to d_1 \mid d_2 \mid ......... \mid d_k$ are all the current Aj -productions;

end

eliminate the immediate left recursion among the Ai-productions.

end.

**Removal of left recursion due to many productions .**

$S \to A\ a \mid b$

$A \to \quad A\ c \mid S\ d \mid e$

. After the first step **(substitute S by its rhs in the rules)** the grammar becomes

$S \to Aa \mid b$

$A \to Ac \mid Aad \mid bd \mid \varepsilon$

. After the second step (removal of left recursion) the grammar becomes

$S \to Aa \mid b$

$A \to bdA' \mid A'$

$A' \to cA' \mid adA' \mid \varepsilon$

**After the first step (substitute S by its R.H.S. in the rules), the grammar becomes**

**$S \rightarrow A\ a \mid b$**

**$A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \varepsilon$**

**After the second step (removal of left recursion from the modified grammar obtained after the first step), the grammar becomes**

**$S \rightarrow A\ a \mid b$**

**$A \rightarrow b\ d\ A' \mid A'$**

**$A' \rightarrow c\ A' \mid a\ d\ A' \mid \varepsilon$**

**Left factoring**

. In top-down parsing when it is not clear which production to choose for expansion of a symbol

defer the decision till we have seen enough input.

In general if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

defer decision by expanding A to a A'

we can then expand A' to $\beta_1$ or $\beta_2$

. Therefore $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

transforms to

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta_1 \mid \beta_2$

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two or more alternative productions to use to expand a non-terminal A, we defer the decision till we have seen enough input to make the right choice.

In general if $A \rightarrow \alpha\ ß_1 \mid \alpha\ ß_2$ , we defer decision by expanding A to a A'.

and then we can expand A ' to $ß_1$ or $ß_1$

Therefore $A \rightarrow \alpha\ ß_1 \mid \alpha\ ß_2$ transforms to

$A \rightarrow \alpha\ A'$

$A' \rightarrow ß_1 \mid ß_2$

**Predictive parsers**

. A non recursive top down parsing method

. **Parser "predicts" which production to use**

. **It removes backtracking by fixing one production for every non-terminal and input token(s)**

. **Predictive parsers accept LL(k) languages**

- First L stands for left to right scan of input

- Second L stands for leftmost derivation

- k stands for number of lookahead token

. **In practice LL(1) is used**

In general, the selection of a production for a non-terminal may involve trial-and-error; that is, we may have to try a production and backtrack to try another production if the first is found to be unsuitable. A production is unsuitable if, after using the production, we cannot complete the tree to match the input string. Predictive parsing is a special form of recursive-descent parsing, in which the current input token unambiguously determines the production to be applied at each step. After eliminating left recursion and left factoring, we can obtain a grammar that can be parsed by a recursive-descent parser that needs no *backtracking* . Basically, it removes the need of *backtracking* by fixing one production for every non-terminal and input tokens. Predictive parsers accept LL(k) languages where:

. First **L** : The input is scanned from left to right.

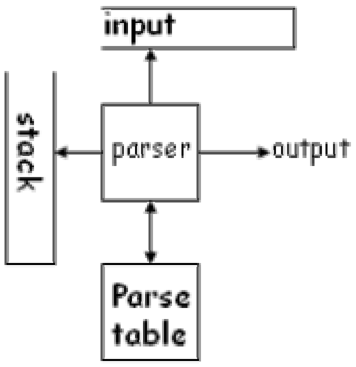
. Second **L** : Leftmost derivations are derived for the strings.

. **k** : The number of lookahead tokens is k.

However, in practice, LL(1) grammars are used i.e., one lookahead token is used.

**Predictive parsing**

. Predictive parser can be implemented by maintaining an external stack



Parse table is a two dimensional array M[X,a] where "X" is a non terminal and "a" is a terminal of the grammar

It is possible to build a non recursive predictive parser maintaining a stack explicitly, rather than implicitly via recursive calls. A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with a $ on the bottom, indicating the bottom of the stack. Initially the stack contains the start symbol of the grammar on top of $. The parsing table is a two-dimensional array M [X,a] , where X is a non-terminal, and a is a terminal or the symbol $ . The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive parser looks up the production to be applied in the parsing table. We shall see how a predictive parser works in the subsequent slides.

**Parsing algorithm**

. The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol

. These two symbols determine the action to be taken by the parser

. Assume that '$' is a special token that is at the bottom of the stack and terminates the input string

```
if X = a = $ then halt

if X = a ≠ $ then pop(x) and ip++

if X is a non terminal

then if M[X,a] = {X à UVW}

then begin pop(X); push(W,V,U)

end

else error
```

The parser is controlled by a program that behaves as follows. The program considers X , the symbol on top of the stack, and a , the current input symbol. These two symbols determine the action of the parser. Let us assume that a special symbol ' $ ' is at the bottom of the stack and terminates the input string. There are three possibilities:

1. If X = a = $, the parser halts and announces successful completion of parsing.

2. If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.

3. If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a] = {X      UVW}, the parser replaces X on top of the stack by UVW (with U on the top). If M[X,a] = error, the parser calls an error recovery routine.

The behavior of the parser can be described in terms of its configurations, which give the stack contents and the remaining input.

## Example

. Consider the grammar

E →T E'

E' →+T E' | ε

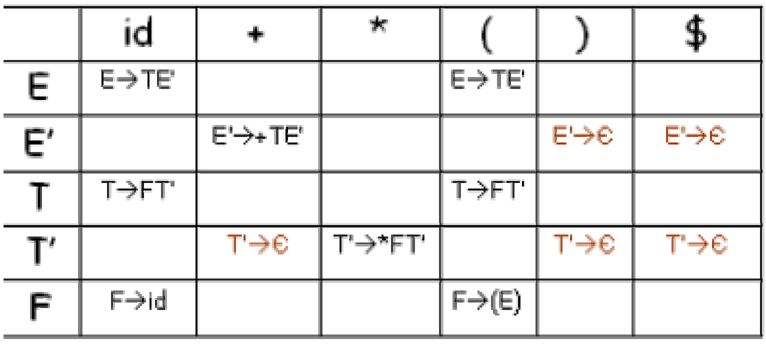
T →F T'

T' →* F T' | ε

F → ( E ) | id

As an example, we shall consider the grammar shown. A predictive parsing table for this grammar is shown in the next slide. We shall see how to construct this table later.

## Parse table for the grammar



|  | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E→TE' |  |  | E→TE' |  |  |
| E' |  | E'→+TE' |  |  | E'→ε | E'→ε |
| T | T→FT' |  |  | T→FT' |  |  |
| T' |  | T'→ε | T'→*FT' |  | T'→ε | T'→ε |
| F | F→id |  |  | F→(E) |  |  |

Blank entries are error states. For example E cannot derive a string starting with '+'

**A predictive parsing table for the grammar in the previous slide is shown. In the table, the blank entries denote the error states; non-blanks indicate a production with which to expand the top nonterminal on the stack.**

**Example**

| **Stack** | **input** | **action** |
|---|---|---|
| $E | id + id * id $ | expand by E TE' |
| $E'T | id + id * id $ | expand by T FT' |
| $E'T'F | id + id * id $ | expand by F id |
| $E'T'id | id + id * id $ | pop id and ip++ |
| $E'T' | + id * id $ | expand by T' ε |
| $E' | + id * id $ | expand by E' +TE' |
| $E'T+ | + id * id $ | pop + and ip++ |
| $E'T | id * id $ | expand by T FT' |

Let us work out an example assuming that we have a parse table. We follow the predictive parsing algorithm which was stated a few slides ago. With input **id id * id** , the predictive parser makes the sequence of moves as shown. The input pointer points to the leftmost symbol of the string in the INPUT column. If we observe the actions of this parser carefully, we see that it is tracing out a leftmost derivation for the input, that is, the productions output are those of a leftmost derivation. The input symbols that have already been scanned, followed by the grammar symbols on the stack (from the top to bottom), make up the left-sentential forms in the derivation.

**Constructing parse table**

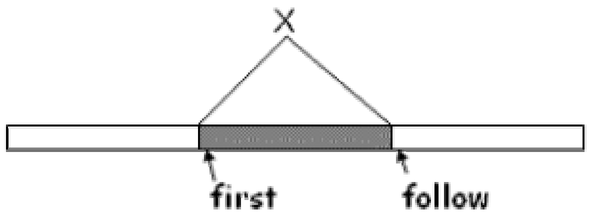
. Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production

. First( a ) for a string of terminals and non terminals a is

- Set of symbols that might begin the fully expanded (made of only tokens) version of a

. Follow(X) for a non terminal X is

- set of symbols that might follow the derivation of X in the input stream



The construction of the parse table is aided by two functions associated with a grammar G. These functions, **FIRST** and **FOLLOW** , allow us to fill in the entries of a predictive parsing table for G, whenever possible. If α is any string of grammar symbols, **FIRST** ( **α** ) is the set of terminals that begin the strings derived from **α** . If α $\Rightarrow^*$ ε , then ε is also in FIRST( α ).

If **X** is a non-terminal, **FOLLOW** ( **X** ) is the set of terminals *a* that can appear immediately to the right of **X** in some sentential form, that is, the set of terminals **a** such that there exists a derivation of the form **S** $\Rightarrow^*$ **α A *a* ß** for some **α** and **ß** . Note that there may, at some time, during the derivation, have been symbols between **A** and **a** , but if so, they derived **ε** and disappeared. Also, if **A** can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

**Compute first sets**

. If X is a terminal symbol then First(X) = {X}

. If X →ε is a production then ε is in First(X)

. If X is a non terminal

and X →Y $_1$ Y $_2$ 2 . Y$_k$ is a production

then

if for some i, a is in First(Y$_i$ )

and ε is in all of First(Y$_j$ ) (such that j<i)

then a is in First(X)

. If ε is in First (Y$_1$ ) . First(Y$_k$ ) then ε is in First(X)

To compute FIRST ( X ) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.

**1. If X is terminal, then First (X) is {X}.**

**2. If X → ε is a production then add e to FIRST(X).**

**3. If X is a non terminal and X →Y $_1$ Y$_k$ .........Y $_k$ is a production, then place *a* in First (X) if for some i, *a* is in FIRST(Y$_i$ ) and e is in all of FIRST(Y $_1$ ), FIRST(Y $_2$ ),.., FIRST(Y$_{i-1}$ );that is, Y$_1$ ..Y $_{i-1}$ $\Rightarrow^*$ ε . If ε is in FIRST(Y$_j$ ) for all i = 1,2,..,k, then add ε to FIRST(X). For example, everything in FIRST(Y$_1$ ) is surely in FIRST(X). If Y $_1$ does not derive ε , then we add nothing more to FIRST(X), but if Y$_1$ $\Rightarrow^*$ ε , then we add FIRST(Y$_2$ ) and so on.**

**Example**

. For the expression grammar

E →T E'

E' →+T E' | ε

T →F T'

T' →* F T' | ε

F → ( E ) | id

First(E) = First(T) = First(F) = { (, id }

First(E') = {+, ε }

First(T') = { *, ε }

Consider the grammar shown above. For example, id and left parenthesis are added to FIRST(F) by rule (3) in the definition of FIRST with i = 1 in each case, since FIRST(id) = {id} and FIRST{'('} = { ( } by rule (1). Then by rule (3) with i = 1, the production T = FT' implies that id and left parenthesis are in FIRST(T) as well. As another example, e is in FIRST(E') by rule (2).

**Compute follow sets**

1. Place $ in follow(S)

2. If there is a production A →a B ß then everything in first( ß ) (except ε ) is in follow(B)

3. If there is a production A →a B then everything in follow(A) is in follow(B)

4. If there is a production A →a B ß and First( ß ) contains e then everything in follow(A) is in follow(B)

Since follow sets are defined in terms of follow sets last two steps have to be repeated until follow sets converge

**To compute FOLLOW ( A ) for all non-terminals A , apply the following rules until nothing can be added to any FOLLOW set:**

**1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker.**

**2. If there is a production A →a Bß, then everything in FIRST(ß) except for e is placed in FOLLOW(B).**

3. If there is a production A →a ß, or a production A →a Bß where FIRST(ß) contains e (i.e., ß⇒ * e ), then everything in FOLLOW(A) is in FOLLOW(B).

**Error handling**

. Stop at the first error and print a message

- Compiler writer friendly

- But not user friendly

. Every reasonable compiler must recover from errors and identify as many errors as possible

. However, multiple error messages due to a single fault must be avoided

. Error recovery methods

- Panic mode

- Phrase level recovery

- Error productions

- Global correction

**Error handling and recovery is also one of the important tasks for a compiler. Errors can occur at any stage during the compilation. There are many ways in which errors can be handled. One way is to stop as soon as an error is detected and print an error message. This scheme is easy for the programmer to program but not user friendly. Specifically, a good parser should, on encountering a parsing error, issue an error message and resume parsing in some way, repairing the error if possible. However, multiple error messages due to a single fault are undesirable and tend to cause confusion if displayed. Error recovery is thus a non trivial task. The following error recovery methods are commonly used:**

**1. Panic Mode**

**2. Phrase level recovery**

**3. Error productions**

**4. Global correction**

**Panic mode**

. Simplest and the most popular method

. Most tools provide for specifying panic mode recovery in the grammar

. When an error is detected

- Discard tokens one at a time until a set of tokens is found whose role is clear

- Skip to the next token that can be placed reliably in the parse tree

**Let us discuss each of these methods one by one. Panic mode error recovery is the simplest and most commonly used method. On discovering the error, the parser discards input symbols one at a time until one of the designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or *end* , whose role in the source program is clear. The compiler designer must select the synchronizing tokens appropriate for the source language, of course.**

**Panic mode .**

. Consider following code

```
begin

a = b + c;

x = p r ;

h = x < 0;

end;
```

. The second expression has syntax error

. Panic mode recovery for begin-end block skip ahead to next ';' and try to parse the next expression

. It discards one expression and tries to continue parsing

. May fail if no further ';' is found

Consider the code shown in the example above. As we can see, the second expression has a syntax error. Now panic mode recovery for begin-end block states that in this situation skip ahead until the next ' ; ' is seen and try to parse the following expressions thereafter i.e., simply skip the whole expression statement if there is an error detected. However, this recovery method might fail if no further ' ; ' is found. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity and, unlike some other methods to be considered later, it is guaranteed not to go into an infinite loop. In situations where multiple errors in the same statement are rare, this method may be quite adequate.