Error management

Errors can occur at each phase of compilation.

Lexical analysis

- Characters outside the alphabet appear, e.g. "`$`", "`%`"
- Character sequences which do not result in a token, e.g. "`55ES`".

Syntactic analysis

- "`;`" missing.
- Badly spelled reserved words, e.g. "`BEGNI`".

Semantic analysis

- Type conflicts of operands.
- Non-declared variables.
- Incorrect procedure calls (e.g. wrong number of parameters).

Code optimization

- Uninitiated variables.
- Dead code, e.g. procedures which are never called.

Code generation

- Too large constants.
- Run out of memory.

Table management

- Overflow in the table.

And all run-time errors which can occur during execution:

- "`Array index out of bounds`".
- Write in or read from unopened files.
- "`Illegal reference at 470105`".

The task of the compiler

- Discover errors.
- Report errors.
- Restart after errors, recovery.
- Correct errors, repair.

Requirements on the error manager

- Find the error when it occurs.
- Provide correct and exact error messages which are not redundant.
- Find all errors.
- Not to introduce any new errors.
- Effective, particularly in time-sharing systems.

Errors

1. Lexical errors
2. Syntactic errors
3. Semantic errors

Lexical and semantic errors are local, i.e. you do not go backwards and forwards in the parse stack or in the token sequence to fix the error. The error is fixed where it occurs, locally.

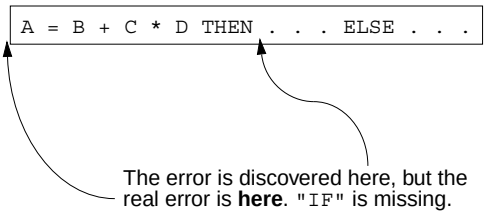
Syntax errors

Syntax errors are discovered when we can not go from one configuration to another as decided by the stack contents and input plus parse tables (applies to bottom-up).

LL- and LR-parsers have a *valid prefix property* i.e. discover the error when the substring being analysed together with the next symbol do not form a prefix of the language.

LL- and LR-parsers discover errors as early as a *left-to-right* parser can.

Example. From PL/1 (where "=" is also used for assigment).



Two methods:

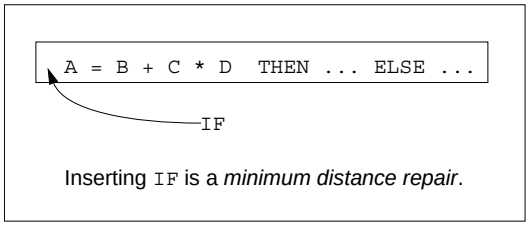
1. Methods that assume a *valid prefix* (called *phrase level* in ASU).
2. Methods based on a *valid prefix* (called *global correction* in ASU).

Minimum distance error correction

Definition:

> The least number of operations (such as removal, inserting or replacing) which are needed to transform a string with syntax errors to a string without errors, is called the *minimum distance (Hamming distance)* between the strings.

Example. Correct the string below using this principle.

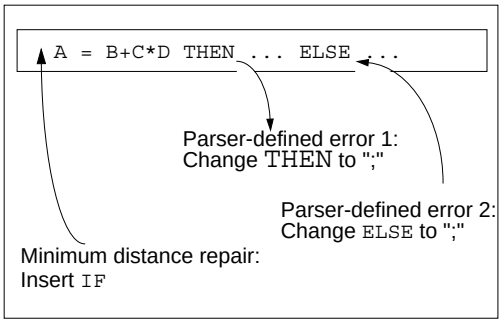


The principle leads to a high level of inefficiency as you have to try all possibilities and choose the one with the least distance!

Parser-defined errors

Let G be a CFG and w = xty an incorrect string, i.e. w ∉ L(G).

If x is a valid prefix while xt is not a valid prefix, t is called a parser defined error.



Methods for syntax error management

1. Panic mode
2. Coding error entries in the `ACTION`-table
3. Error productions
4. Language-independent methods (not included in this course)

   4a) Continuation method, Röchrich (1980)

   4b) Automatic error recovery, Burke & Fisher (1982)

1. Panic mode

   a) Skip input until either

      i) Parsing can continue, or

      ii) An important symbol has been found (e.g. `PROCEDURE,BEGIN, WHILE,...`)

   b) If the parsing can not continue:

      Pop the stack until the important symbol is accepted.

      If you reach the stack bottom:

      ```
      "Quit --Unrecoverable error."
      ```

- Much input can be removed.
- Semantic info on the stack disappears.
+ Systematic, easy to implement.
+ Efficient, very fast and does not require extra memory.

2. Code error entries in the `ACTION`-table

   - In the `ACTION`-table there are many entries corresponding to `ERROR`.
   - Study first what types of error occur most and go into the table and instead of `ERROR` insert a pointer to an error management routine which is to be activated when this particular error state arises.

- Difficult to foresee all possible cases.
- Much coding.
- Modifying the grammar means recoding the error entries.
+ Can provide very good error messages.

3. Error productions

   Extend the grammar with extra productions that allow certain errors.

   Example. From Pascal:

```
IF P THEN A := X ; ELSE B := X ;
```

   A kinder grammar which allows "`;`" here but provides an error message.

Error productions in Yacc (*Controlled panic mode*)

Extend the grammar with error productions of the form

```
A ::= error α
```

which correspond to the most common errors.

`A`: is a nonterminal in the grammar

`error`: fictitious token, reserved word in Yacc

α: is a string of vocabulary symbols or the empty string.

When an error occurs:

1. Pop the stack elements until some state at the top of the stack has an item of the following form in its item-set:
   `A ::= • error α`
2. Shift `error` in as a token.

3. If α is the empty string, reduce using this rule
   `A ::= error {semantic action}`

   and perform the rule's semantic action which in this case is a user-defined syntax error management routine.

   If α is not the empty string, Yacc jumps over all symbols until it finds a string derivable from α, and reduces it using this rule:

   `A ::= error α`

Example. Yacc jumps over all input symbols until the next symbol is a semicolon (inclusive) if the error prediction is:

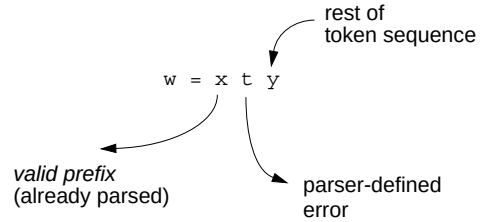
```
A ::= error ;
```


4. Language-independent error management methods

- All information about a language is in the parse tables.
- By looking in the tables you know what is allowed in a configuration.

4a) "Röhrich Continuation Method"

Input: `w`



The algorithm

1. Construct a continuation `u, u ∈` Σ^*, and `w' = xu ∈` L(G).

   Example:

```
program foo;
begin
    while a > b then begin
                    Parser-defined error
    end
end;
```

   `x = program foo; begin while a > b`

   `u = do ε end . _!_`

2. Remove input symbols until an *important* symbol is found (*anchor, beacon*) e.g. `while, if, repeat, begin` etc.

   In this case: `then` is removed as `begin` is the anchor symbol.

3. Insert parts of `u` after `x`, and provide an error message.

   `"DO"` *expected instead of* `"THEN"`.

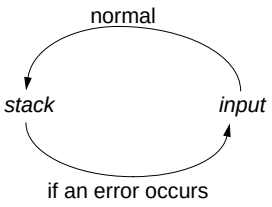
"Röhrich Continuation Method"

+ Language-independent
+ Efficient
– A *valid prefix* can not cause an error.
– Much input can be thrown away.

4b) "Automatic error recovery", Burke & Fisher

Takes into consideration that a *valid prefix* can be error-prone.

Problem: you have to ''back up'' the stack:



This works if information is still in the stack but this is not always the case!

```
if a > b then
c := d;  ⇒ is reduced to <statement>
else
e := 1;
```

Solution: delay a predetermined number of reductions in a buffer.

The algorithm has three phases:

1. *Simple error recovery*
2. *Scope recovery*
3. *Secondary recovery*

Phase 1: *Simple Error Recovery* (a so-called token error)

- Removal of a token
- Insertion of a token
- Replace a token with something else
- *Merging*: Concatenate two adjacent tokens.
- Error spelling (`BEGNI` → `BEGIN`)

Phase 2: *Scope Recovery*

Insertion of several tokens to switch off open *scope*.

Opener	***Closer***
`PROGRAM`	`BEGIN END.`
	`.`
`PROCEDURE`	`BEGIN END;`
	`;`
`BEGIN`	`END`
`(`	`)`
`[`	`]`
`REPEAT`	`UNTIL` *identifier* `;`
	`UNTIL` *identifier*
`ARRAY`	`OF` *identifier* `;`
	`OF` *identifier*

Phase 3: *Secondary recovery*

Similar to *panic mode*.

Phase 3 is called if phase 1 and 2 did not succeed in putting the parser back on track.

"Automatic error recovery", Burke & Fisher

+ Language-independent
+ Provides very good error messages
+ Able to make modifications to the parse stack (by ''backing up'' the stack)
– Consumes some time and memory.

Test program for error recovery

```
 1    PROGRRAM scoptest(input,output);
 2
 3    CONST mxi dlen = 10
 4
 5    VAR a,b,c;d :INTEGER;
 6
 7        arr10 : ARRAY [1..mxidlen]  ;
 8
 9
10       PROCEDURE foo(VAR k:INTEGER) : BOOLEAN;
11
12       VAR i, : INTEGER;
13
14       BEGIN )* foo *)
15
16          REPEAT
17
18             a:= (a + c);
19
20             IF (a > b) THEN a:= b ; ELSE b:=a;
21
22      PROCEDURE fie(VAR i,j:INTEGER);
23
24      BEGIN (* fie *)
25
26         a = a + 1;
27
28      END (* fie *);
29
30
31
32      A := B + C;
33
34    END.
```


Error messages from Hedrick Pascal

```
    1   PROGRRAM scoptest(input,output);
P*  1**         ^          **************^
1.^:  "BEGIN" expected
2.^:  ":=" expected

    3   CONST mxi dlen = 10
P*  1**     ^        ^  **
1.^:  "END" expected
2.^:  "=" expected
2.^:  Identifier not declared

    5   VAR a,b,c;d :INTEGER;
P*  1**   ^     ^
1.^:  ";" expected
2.^:  Can't have that here (or something extra or missing
before)
2.^:  ":" expected

    7       arr10 : ARRAY [1..mxidlen]  ;
P*  1**                                ^^  ^
1.^:  Identifier not declared
2.^:  Incompatible subrange types
3.^:  "OF" expected

   10      PROCEDURE foo(VAR k:INTEGER) : BOOLEAN;
P*  1**                                 ^********
1.^:  Can't have that here (or something extra or missing
before)

   12      VAR i, : INTEGER;
P*  1**           ^
1.^:  Identifier expected

   14      BEGIN )* foo *)
P*  1**          ^*******
1.^:  Can't have that here (or something extra or missing
before)

   20            IF (a > b) THEN a:= b ; ELSE b:=a;
P*  1**                                   ^*****
1.^:  ELSE not within an IF-THEN (extra ";","END",etc. before
```



```
it?)

   22     PROCEDURE fie(VAR i,j:INTEGER);
P*  1**              ^
1.^:  "UNTIL" expected
1.^:  "END" expected
1.^:  ";" expected

   26        a = a + 1;
P*  1**       ^******^
1.^:  Can't have that here (or something extra or missing
before)
2.^:  ":=" expected

   32     A := B + C;
P*  1**   ^**********
1.^:  Can't have that here (or something extra or missing
before)

   34   END.
P*  1** ***
?  Unexpected end of file
```

Error messages from Sun Pascal

```
     1  PROGRAM scoptest(input,output);
e ------^--- Inserted '['
E -------------------------------------^--- Expected ']'

     3  CONST mxi dlen = 10
e ----------------^--- Deleted identifier

     5  VAR a,b,c;d :INTEGER;
e ------^--- Inserted ';'
e --------------^--- Replaced ';' with a ','

     7      arr10 : ARRAY [1..mxidlen] ;
E -------------------------------------^- Expected keyword of
E -------------------------------------^- Inserted identifier

   PROCEDURE foo(VAR k:INTEGER) : BOOLEAN;
E--------- Procedures cannot have types

    12     VAR i, : INTEGER;
E --------------^--- Deleted ','

    14     BEGIN )* foo *)
E ---------------^--- Malformed statement

    20           IF (a > b) THEN a:= b ; ELSE b:=a;
e -----------------------------------^--- Deleted ';' before
keyword else

    22    PROCEDURE fie(VAR i,j:INTEGER);
E --------^--- Expected keyword until
E --------^--- Expected keyword end
E --------^--- Inserted keyword end matching begin on line 14
e --------^--- Inserted ';'

    26       a = a + 1;
e -------------^--- Replaced '=' with a keyword (null)


    32    A := B + C;
e --------^--- Inserted keyword (null)

    34  END.
E ------^--- Malformed declaration
E ------^--- Unrecoverable syntax error - QUIT
```


Error messages from Burke & Fisher's "Automatic Error Recovery"

```
   1  PROGRRAM scoptest(input,output);
      ^^^^^^^^
*** Lexical Error: Reserved word "PROGRAM" misspelled

   3  CONST mxi dlen = 10
             ^^^ ^^^
*** Lexical Error: "MXIDLEN" expected instead of "MXI"  "DLEN"

   3  CONST mxi dlen = 10
                       ^^
*** Syntax Error: ";" expected after this token

   5  VAR a,b,c;d :INTEGER;
               ^
*** Syntax Error: "," expected instead of ";"

   7      arr10 : ARRAY [1..mxidlen]  ;
                                  ^
*** Syntax Error: "OF IDENTIFIER" inserted to match "ARRAY"

  10     PROCEDURE foo(VAR k:INTEGER) : BOOLEAN;
         ^^^^^^^^^
*** Syntax Error: "FUNCTION" expected instead of "PROCEDURE"

  12     VAR i, : INTEGER;
                ^
*** Syntax Error: "IDENTIFIER" expected before this token

  14     BEGIN )* foo *)
                <------->
*** Syntax Error: Unexpected input

  20           IF (a > b) THEN a:= b ; ELSE b:=a;
                                       ^
*** Syntax Error: Unexpected ";" , ignored

  20           IF (a > b) THEN a:= b ; ELSE b:=a;
                                                ^
*** Syntax Error: "UNTIL IDENTIFIER" inserted to match "REPEAT"
*** Syntax Error: "END" inserted to match "BEGIN"


  26       a = a + 1;
             ^
```



```
*** Syntax Error: ":=" expected instead of "="

  32    A := B + C;
        ^
*** Syntax Error: "BEGIN" expected before this token

12 errors detected
```