Error Recovery

Syntax Errors:

- A *Syntax Error* occurs when stream of tokens is an invalid string.
- In LL(k) or LR(k) parsing tables, blank entries refer to syntax erro

How should syntax errors be handled?

1. Report error, terminate compilation ⇒ not user friendly
2. Report error, *recover* from error, search for more errors ⇒ better

Error Recovery

***Error Recovery*: process of adjusting input stream so that parsing n**
syntax error reported.

- Deletion of token types from input stream
- Insertion of token types
- Substitution of token types

Two classes of recovery:

1. *Local Recovery*: adjust input at point where error was detected.
2. *Global Recovery*: adjust input *before* point where error was detect

These may be applied to both LL and LR parsing techniques.

LL Local Error Recovery

Local Recovery Technique: in function A(), delete token types from i
token type in follow(A) found ⇒ *synchronizing* token types.

$Z \rightarrow XYZ$ $\qquad Y \rightarrow$ c $\qquad X \rightarrow$

$Z \rightarrow$ d $\qquad Y \rightarrow \epsilon$ $\qquad X \rightarrow$

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

	a	b	c	d	e
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$		$Z \rightarrow d$	
Y	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$	$Y \rightarrow c$	$Y \rightarrow \epsilon$	$Y \rightarrow$
X	$X \rightarrow a$	$X \rightarrow bYe$			

LL Local Error Recovery

Local Recovery Technique: in function A(), delete token types from i
token type in follow(A) found ⇒ *synchronizing* token types.

```
datatype token = a | b | c | d | e;
val tok = ref(getToken());
fun advance() = tok := getToken();
fun eat(t) = if(!tok = t) then advance() else e
...
and X() = case !tok of
      a => (eat(a))
    | b => (eat(b); Y(); eat(e))
    | c => (print "error!"; skipTo[a,b,c,d])
    | d => (print "error!"; skipTo[a,b,c,d])
    | e => (print "error!"; skipTo[a,b,c,d])

and skipTo(synchTokens) =
    if member(!tok, synchTokens) then ()
    else (eat(!tok); skipTo(synchTokens))
```

LR Local Error Recovery

Consider:

$1\ E \rightarrow$ ID $\quad 3\ E \rightarrow$ (E) $\quad 5\ ES \rightarrow$

$2\ E \rightarrow E$ + $E \quad 4\ ES \rightarrow E$

- Match a sequence of erroneous input tokens using the *error* token

$6\ E \rightarrow$ (error) $\qquad 7\ ES \rightarrow$ error ; E

- In general, follow *error* with synchronizing lookahead token.

1. Pop stack (if necessary) until a state is reached in which the ac
   token is *shift*.
2. Shift the *error* token.
3. Discard input symbols (if necessary) until a state is reached tha
   action in the current state.
4. Resume normal parsing.

Global Error Recovery

Consider LR(1) parsing:

```
let type a := intArray[10] of 0 in ... end
```

Local Recovery Techniques would:

1. report syntax error at ':='
2. substitute '=' for ':='
3. report syntax error at '['
4. delete token types from input stream, synchronizing on 'in'

Global Recovery Techniques would substitute 'var' for 'type':

- Actual syntax error occurs *before* point where error was detected.
- ML-Yacc uses global error recovery technique $\Rightarrow$ *Burke-Fisher*
- Other Yacc versions employ local recovery techniques.

Burke-Fisher

Suppose parser gets stuck at n^{th} token in input stream.

- Burke-Fisher repairer tries every *single-token-type* insertion, delet
tion at all points between $(n - k)^{th}$ and n^{th} token.

- Best repair: one that allows parser to parse furthest past n^{th} token.

- If languages has N token types, then:

total # of repairs = deletions + insertions + substitu

total # of repairs = $(k) + (k + 1) N + (k) (N -$

Burke-Fisher

In order to backup K tokens and reparse repaired input, 2 structures ne

1. *k-length buffer/queue* - if parser currently processing n^{th} token, q kens $(n-k) \rightarrow (n-1)$. (ML-Yacc $k = 15$)

2. *old parse stack* - if parser currently processing n^{th} token, old stacl state when parser was processing $(n-k)^{th}$ token.

- Whenever token shifted onto current stack, also put onto queue tai
- Simultaneously, queue head removed, shifted onto old stack.
- Whenever token shifted onto either stack, appropriate reductions p

Burke-Fisher

- Semantic actions are only applied to old stack.
  - Not desirable if semantic actions affect lexical analysis.
  - Example: `typedef` in C.

(Figure from MCI/ML.)

Burke-Fisher

For each repair R that can be applied to token $(n-k) \rightarrow n$:

1. copy queue, copy n^{th} token
2. copy old parse stack
3. apply R to copy of queue or copy of n^{th} token
4. reparse queue copy (and copy of n^{th} token) from old stack copy
5. evaluate R

Choose best repair R, and apply.

Burke-Fisher in ML-YACC

Semantic Values

- Insertions need semantic values

```
%value ID {"bogus"}
%value INT {1}
%value STRING {"STRING")
```

Programmer-Specified Substitutions

- Some single token insertions and deletions are common.
- Some multiple token insertions and deletions are common.

```
%change EQ -> ASSIGN | SEMICOLON ELSE -> ELS
        |   -> IN INT END
```