BOTTOM UP PARSERS

input: sequence of tokens

for compilers

- left-to-right
- no back-up over input
- deterministic – parsing decisions made as input read, not undone (i.e. no backtracking)
- linear time - i.e. parsing decisions made using bounded amount of information
- unambiguous - one parse tree for each sentence in language

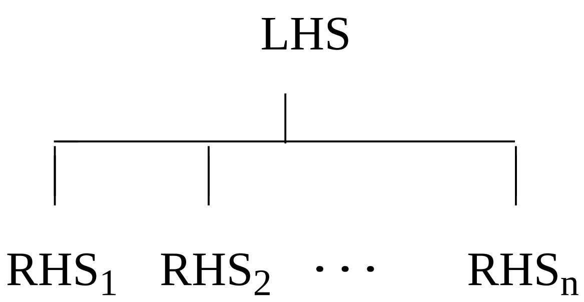
output: parse tree or effect of bottom-up left-to-right tree traversal

Conceptually, parsing algorithm builds tree.

● program

input: x x x ----- x

Each single level subtree has form



corresponding to LHS $\rightarrow$ RHS_1 RHS_2 --- RHS_n

To get parser properties:

Make as many decisions as possible using input already read, then read more.

Possible actions for parser:

ACCEPT: entire parse tree constructed

REDUCE: complete a subtree (attach roots of subtrees forming a RHS to parent which is LHS)

SHIFT: read next input symbol - becomes singleton subtree

ERROR: none of the other actions are valid

each parsing step:

if *done* then ACCEPT

else if *reduce_okay* then REDUCE by RULE I

else if *shift_okay* then SHIFT

else ERROR

Building a parse tree bottom-up left-to-right corresponds to reconstructing the corresponding rightmost derivation ‘‘backwards’’

Since roots of subtrees are connected to their parent as soon as possible, the roots can be kept on a stack

The parser actions then become:

REDUCE: pop RHS from stack;
push LHS (new root) onto stack

SHIFT: push next input symbol (new root) onto stack

Hence these are called *shift-reduce* parsing methods

Notice that the stack symbols (read bottom to top) followed by the remaining input constitute a rightmost sentential form

Example - Expression grammar

nonterminals: E (expression), T(term), F(factor)
token $ means end-of-input

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow id$
$F \rightarrow (E)$

input: id + id * id

stack	input	action
	id+id*id$	shift
id	+id*id$	reduce $F \rightarrow id$
F	+id*id$	reduce $T \rightarrow F$
T	+id*id$	reduce $E \rightarrow T$
E	+id*id$	shift
E +	id*id$	shift
E + id	*id$	reduce $F \rightarrow id$
E + F	*id$	reduce $T \rightarrow F$
E + T	*id$	shift
E + T *	id$	shift
E + T * id	$	reduce $F \rightarrow id$
E + T * F	$	reduce $T \rightarrow T * F$
E + T	$	reduce $E \rightarrow E + T$
E	$	accept

Bottom up parsing methods differ in the information used to decide which action to take

done: no more input and root of connected tree is start symbol (i.e. stack contains only the start symbol)

reduce_okay (by RULE I):
LHS of RULE I and reduction are correct in this context

shift_okay:
there is more input;
[top of stack is prefix of RHS];
next input symbol extends RHS or starts new one;
[prefix of RHS is correct in this context]

one possibility – reduction is correct in this context if *shift_okay* for reduction RHS

LR PARSING

lookahead: symbols in input which are inspected but not ‘‘used up’’.

LR(k) parser – action is determined by stack contents and k lookahead symbols

stack contents are summarized by a **state** whose number is on the top of the stack

i.e., stack contains alternating grammar symbols and state numbers

current state of parser – state whose number is at top of stack

states are also called *item sets*

LR languages and grammars

LR(k) language = language generated by LR(k) grammar

LR(k) grammar – satisfies properties checked by table constructor

Facts

set of LR(k) languages, $k \geq 1$ = set of LR(1) languages

set of LR(k) grammars $\subset$ set of LR(k+1) grammars, $k \geq 0$

LR grammars are largest class of grammars satisfying desired parsing properties

both LR language class and LR grammar classes are larger than corresponding top-down parsable (LL) classes

More about the parsing algorithm:

parsing stack:

$s_0\ X_1\ s_1\ X_2\ s_2\ X_3\ s_3 \cdots X_n\ s_n$

action table: ACTION (s_i , a)

row for each state
column for each terminal symbol

each entry is either

accept
reduce by rule i (abbreviated **r** i)
shift and enter state j (abbreviated **s** j)
error (abbreviated by blank)

goto table: GOTO (s_i , X)

row for each state
column for each nonterminal symbol

each entry is either

j (if j is state to enter after pushing LHS)
blank (if no next state)

each LR parsing step:

assert STATE = stack [stacktop]; -- current state

ACT ← ACTION (STATE, input_symbol);

if ACT = ACCEPT **then** ACCEPT_ACTION

else if ACT = (**r** i) **then**
    pop 2*length(RHS of rule i) symbols from stack;
    STATE ← GOTO (stack [stacktop], LHS of rule i);
    push LHS of rule i onto stack;
    push STATE onto stack

else if ACT = (**s** j) **then**
    push input_symbol onto stack;
    scan;   -- advance input
    STATE ← j;
    push STATE onto stack

else ERROR_ACTION

for LR(k), the second argument of ACTION or GOTO is a lookahead sequence of k input symbols

in practice, k=1

Table Representation

ACTION and GOTO can have different representations

Each can be 2 - dimensional matrix

For each row (i.e. each state), can have list of non–blank entries. Each entry is labelled by its column (i.e. grammar symbol).

If the only action is one reduction, it can be done by default.

For each column, (i.e. each grammar symbol), can have list of non–blank entries. Each entry is labelled by its row (i.e. state).

EXAMPLE

0 $S \rightarrow B$
1 $B \rightarrow$ **begin** SL **end**
2 $SL \rightarrow SL$ **;** STAT
3 $SL \rightarrow STAT$
4 $STAT \rightarrow B$
5 $STAT \rightarrow$ assn

state	ACTION	GOTO
0	**begin**: **s** 2	B: 1
1	$: accept	
2	**begin**: **s** 2 assn: **s** 6	SL: 3 STAT: 4 B: 5
3	**end**: **s** 7 **;**: **s** 8	
4	default: **r** 3	
5	default: **r** 4	
6	default: **r** 5	
7	default: **r** 1	
8	**begin**: **s** 2 assn: **s** 6	B: 5 STAT: 9
9	default: **r** 2	

EXAMPLE

input: **begin** assn **;** assn **end**

stack	input	act.
0	**begin** assn **;** assn **end**$	**s**
0 **begin** 2	assn **;** assn **end**$	**s**
0 **begin** 2 assn 6	**;** assn **end**$	**r** 5
0 **begin** 2 STAT 4	**;** assn **end**$	**r** 3
0 **begin** 2 SL 3	**;** assn **end**$	**s**
0 **begin** 2 SL 3 **;** 8	assn **end**$	**s**
0 **begin** 2 SL 3 **;** 8 assn 6	**end**$	**r** 5
0 **begin** 2 SL 3 **;** 8 STAT 9	**end**$	**r** 2
0 **begin** 2 SL 3	**end**$	**s**
0 **begin** 2 SL 3 **end** 7	$	**r** 1
0 B 1	$	acc.

To Compute Table for LR Parsers

For now, assume that

- k = 0 (no lookahead)
- start symbol of grammar not in any RHS

Define the notion of an **item**:

$X \rightarrow A \cdot B\ C$

The dot shows progress in building a parse tree. Subtree with root A parsed. Subtrees with roots B and C are predicted

$X \rightarrow \cdot A\ B\ C$

None of RHS found yet. Predicted because X was predicted.

$X \rightarrow A\ B\ C \cdot$

All of RHS found (called a *reduce-item*)

$X \rightarrow \cdot$

RHS of length 0 predicted and found

Each state corresponds to a (finite) nonempty set of items.

kernel of state - set of items in state with one or more symbols to left of $\cdot$

start state - ‘‘kernel’’ is $S' \rightarrow \cdot S$

predict(X) = $\{X \rightarrow \cdot\, \alpha \mid X \rightarrow \alpha$ is a production$\}$

To compute each state:

(1) Start with the kernel

(2) Extend kernel of state to its closure as follows:

repeat until no change
  for each item $Y \rightarrow \alpha \cdot X\, \beta$ in state,
    add predict(X) to state

(3) Compute possibly new kernels:

for each symbol X such that $Y \rightarrow \alpha \cdot X\, \beta$ is in state,
  compute kernel of successor state
  by moving $\cdot$ over X in each such item
  to form kernel

EXAMPLE

0

S → · B

B → · **begin** SL **end**

B

1

S → B ·

begin

2

B → **begin** · SL **end**

SL → · SL ; STAT
SL → · STAT
STAT → · B
STAT → · assn
B → · **begin** SL **end**

STAT

4

SL → STAT ·

B

5

STAT → B ·

assn

6

STAT → assn ·

begin   SL

3

B → **begin** SL · **end**
SL → SL · ; STAT

end

7

B → **begin** SL **end** ·

;

8

SL → SL ; · STAT

STAT → · B
STAT → · assn
B → · **begin** SL **end**

STAT

9

SL → SL ; STAT ·

B

$\boxed{5}$

assn

$\boxed{6}$

begin

$\boxed{2}$

Table entries:

if state j constructed by moving from state i across terminal symbol t, then ACTION (i, t) = **s** j

if state j constructed by moving from state i across nonterminal symbol X, then GOTO (i, X) = j

if state i contains reduce-item for rule j

$$X \rightarrow Y_1\, Y_2 \ldots Y_n \cdot$$

then ACTION (i, t) = **r** j for each terminal symbol t†

if state i is successor of start state and contains reduce-item $S' \rightarrow S \cdot$ then ACTION (i, $) = accept

†In general, for each terminal string t that satisfies some condition

Notice that

- state computation yields finite state machine (fsm)

- shifts in parser correspond to moves in fsm

- can compute current state by feeding stack (bottom to top) to fsm

- for any items

$$X \to \alpha_1 \cdot \ \beta_1$$
$$Y \to \alpha_2 \cdot \ \beta_2$$

in a state, α_1 is a suffix of α_2 or vice-versa. When that state is on the parsing stack, α_1 (and also α_2) is the sequence of symbols immediately below on the stack.

- The symbol preceding $\cdot$ in an item in the state is the symbol preceding the state number on the stack. Thus if the states are on the stack, the symbols need not be.

The states can be computed for *any* context free grammar.

The table and the parser have a
reduce/reduce conflict if there are 2 or more reduce actions for the same table entry (i.e. 2 reduce-items in one state)

For example:

$S \rightarrow a\ B | a\ b$
$B \rightarrow b$

stack: a b   input: $

shift/reduce conflict if there are both shift and reduce actions for the same table entry (i.e. a reduce-item and a non reduce-item in the same state)

For example:

$S \rightarrow A\ b | a\ b\ c$
$A \rightarrow a$

stack: a   input: b $

A grammar is LR(0) if no conflicts, i.e. every set of items is *consistent*.

Note that if a grammar is ambiguous, there **must** be a conflict in some state.

If there is a conflict, the grammar may or may not be ambiguous.

Even if the grammar is ambiguous, the language may not be.

Example

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

To get LR(1) or a variant

Use one symbol lookahead to decide to reduce by a rule.

Each reduce-item now has the form

dotted rule, lookahead set.

Example

$X \rightarrow A \cdot B\ C$
$B \rightarrow \cdot\ b\ w\ ,\ t_1 \mid t_2 \mid . . . \mid t_n$

Expected lookahead symbols for B are terminal symbols derived from C.

New definitions of conflicts:

There is a *reduce/reduce* conflict if 2 reduce-items have the same lookahead symbol.

There is a *shift/reduce* conflict if the symbol following the $\cdot$ in one item also appears in the lookahead set of a reduce-item.

SLR(1) versus LALR(1) versus full LR(1)

- same parsing algorithm

- same form of table

- difference in how precisely the lookahead is computed. (The more precise the computation, the smaller the lookahead set for each reduction.)

SLR(1) (Simple (LR(1))

Compute LR(0) fsm

Add lookahead set to each reduce-item. Lookahead set is all those terminal symbols that can follow LHS in a rightmost derivation

FOLLOW(X) = $\{a \mid S \Rightarrow^*_{rm} \ldots X\ a \ldots \}$

FOLLOW is easy to compute. It’s a property of the grammar, not the parser.

Only complication comes from ε rules.

.

... X α Y ...

ε

Z β γ a

ε ε

I. Compute NULL = $\{X \mid X \Rightarrow^* \varepsilon\}$ as follows:

NULL = $\{X \mid X \to \varepsilon\}$

repeat
    for all $Y \to \alpha$
        if $\alpha \in$ NULL* **then** put Y in NULL
until no change;

II. Compute FIRST(X) = $\{Y \mid X \Rightarrow^* Y \beta\}$ as follows:

FIRST(X) = $\{X\} \cup$
    $\{Y \mid$ for some X-rule, $X \to \alpha Y \beta$ and $\alpha \in$ NULL$^*\}$

repeat
    for all $Y \in$ FIRST(X)
        FIRST(X) := FIRST(X) $\cup$ FIRST(Y)
until no change;

FIRST$(X)_T$ = $\{a \mid a$ is a terminal symbol and $a \in$ FIRST(X)$\}$

Notice that if **a** is a terminal, then FIRST(a)=FIRST$_T$(a)={a}

III. Compute $LAST_N(X) =$

$\{Y \mid X \Rightarrow^+ \beta\ Y$ and Y is a nonterminal$\}$

$LAST_N(X) =$

$\{Y \mid$ for some X-rule, $X \rightarrow \beta\ Y\ \alpha$ and $\alpha \in NULL^*\}$

repeat
    for all $Y \in LAST_N(X)$
        $LAST_N(X) := LAST_N(X) \cup LAST_N(Y)$
until no change;

IV. Compute FOLLOW (X) = $\{a \mid S \Rightarrow^* \ldots X\ a \ldots\}$
    where X is a nonterminal symbol and **a** is a terminal symbol

for all X
    FOLLOW(X) := Φ **--** initialization

for all X,Y in any RHS ... $X\alpha Y$ ... where $\alpha \in NULL^*$

    FOLLOW(X) := FOLLOW(X) $\cup$ $FIRST_T(Y)$

    for all $Z \in LAST_N(X)$

        FOLLOW(Z) := FOLLOW(Z) $\cup$ $FIRST_T(Y)$

For LR parsing, FOLLOW may yield too large a lookahead set.

Example

$S \to a\ X\ a \mid b\ X\ b$
$X \to x$

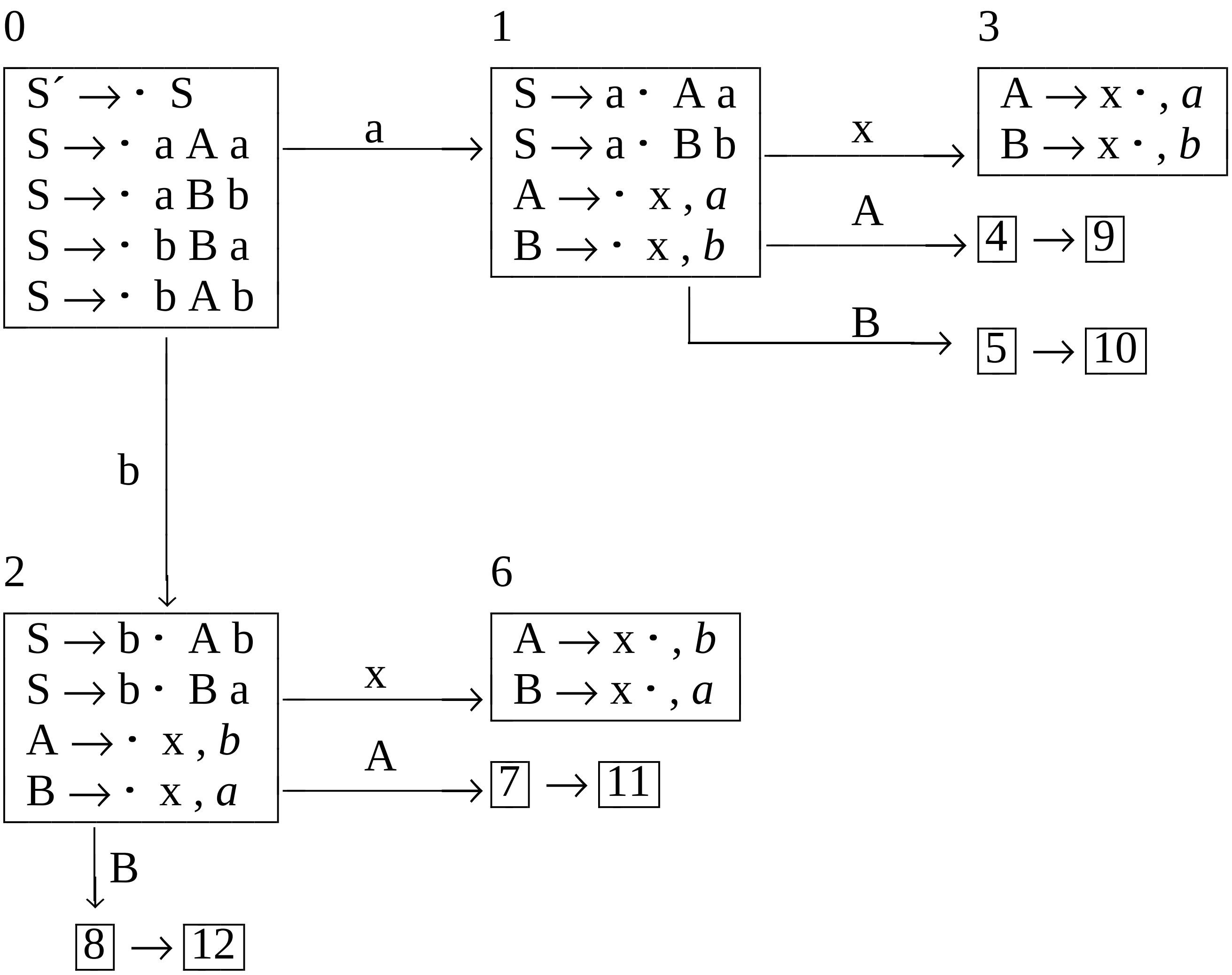
FOLLOW(X) = {a,b}

If the stack is 0 a 1 x
If the input is b $

then the SLR parser will reduce $X \to x$
but will not shift b

EXAMPLE

S → a A a | a B b | b B a | b A b
A → x
B → x



The example is not SLR because FOLLOW(A) = FOLLOW(B)
= {a,b}
Clearly, the paired context determines the reduction.

The idea of canonical (full) LR(k)

Generate the eventual lookahead when the item is predicted.

Every item has a lookahead symbol; i.e. these are *LR(k) items*.

The lookahead is not used by the parser unless is reduce-item.

For LR(0):

predict(X) = $\{X \rightarrow \cdot\, \alpha \mid X \rightarrow \alpha$ is a production$\}$

For LR(k):

predict($[Y \rightarrow \gamma \cdot X\, \beta, t]$) =

$\{[X \rightarrow \cdot\, \alpha, w] \mid X \rightarrow \alpha$ is a production and $\beta t \Rightarrow^{*} w \ldots\}$

One problem is that there are many more items and many more states.

Compromise: **LALR(1) (LookAhead LR(1))**

Lookahead more accurate because generated in context

If two LR(1) item sets have equal LR (0) *cores*, the respective lookaheads are merged and the sets are combined.

Notice that

- the number of states is the same as SLR(1)
- if the LR(1) states have no shift/reduce conflicts, neither do the LALR(1) states
- The previous example is not LALR(1)

An example that is LALR(1) but not SLR(1)

$S \to a\ A\ a \mid b\ A\ b \mid a\ x\ b$
$A \to x$

0		1		3
$S' \to \cdot\ S$ $S \to \cdot\ a\ A\ a$ $S \to \cdot\ b\ A\ b$ $S \to \cdot\ a\ x\ b$	a	$S \to a \cdot\ A\ a$ $S \to a \cdot\ x\ b$ $A \to \cdot\ x\ ,\ a$	x	$S \to a\ x \cdot\ b$ $A \to x \cdot ,\ a$

b

$\boxed{2}$

State 3 has a shift/reduce conflict.

FOLLOW(A) = {a, b}

If context is propagated from state 1, the lookahead set for

$A \to x \cdot$

is {a}

What happens to ε-rules?

$S \rightarrow aXa \mid bXb$
$X \rightarrow y \mid \varepsilon$

1

$S \rightarrow a \cdot X\, a$

$X \rightarrow \cdot\, y$
$X \rightarrow \cdot$

X

2

$S \rightarrow a\, X \cdot a$

grammar is not LR(0)

grammar *is* SLR(1), since FOLLOW(X) = {a,b}

How does the parser work?

If the input was **a a**, then the configuration after shifting the first **a** is

stack: 0 **a** 1 input: **a** $

Next action is to reduce by $X \rightarrow \varepsilon$
 Pop the stack 0 times
 Push X
 Stack GOTO(STATE,X) = GOTO(1,X) = 2

Configuration is
stack: 0 **a** 1 X 2 input: **a** $

Ambiguity

Consider the following example:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

E E E E E E

id + id * id id + id * id

Ambiguous *precedence* of operators

E E E E E E

id + id + id id + id + id

Ambiguous *associativity*

Can be resolved by multi-level grammar we saw before.

The LR(0) states with conflicts are:

7	8
$E \rightarrow E + E \cdot$	$E \rightarrow E * E \cdot$
$E \rightarrow E \cdot + E$	$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$	$E \rightarrow E \cdot * E$

If + is left associative and the next input symbol is +, the action for state 7 should be to reduce.

If * has higher precedence than +, and the next input symbol is *, the action for state 7 should be to shift.

The analysis of state 8 is similar. Notice that these choices are only valid because we know that the language being parsed will still be the same. (I.e., eliminating some trees does not eliminate any valid sentences). That is not true in general.

More about Table Representation

Note that the number of possible states is $2^{\#items}$
It can be exponential in the size of the grammar, even for LR(0). Hence it is important to store the table compactly.

Default reductions

If a state is consistent and has only one possible reduction, that reduction can be done by default (i.e. without checking input)

Shift and reduce actions

If a shift leads to state whose only action is a default reduction, can indicate without separate state

Tagged overlay storage

Compiler handling of syntax errors

- discover the error
- diagnose the error for the user
- restart the parser

error recovery – putting the parser in a state to continue parsing - without cascaded errors

error repair – modifying the input to be syntactically valid

error correction – modifying the input so that the computation is correct

Error Detection

• Syntax errors may be detected many symbols after they occur.

e.g.

```
X : = Y * Z + W ) ;
```

e.g.

A [I] = B **then** read(X);

• Syntax errors may be detected semantically.

e.g.

type complex =
    record re, im: real
    end;
var C: complex;
begin
    ...
    write (C, re)
    ...
end.

Valid prefix property for parsers

A syntax error is detected as soon as the next symbol to be read does not extend the previously read input as the prefix of a sentence in the language.

LR parsers and most top down parsing strategies have the valid prefix property. Some other bottom up strategies (notably precedence methods) do not.

However, in LR parsing, inexact lookahead (SLR (1), default reductions) may cause extra reductions before error detection.

It cannot cause extra shifts.

Context is lost if have default reductions:

for example

A := B C

might be reduced to

assn_st id

Error recovery /repair

Arbitrarily elaborate, depending on performance goals

- speed
- accuracy
- comprehensiveness

Typically, use two - phase strategy

- Local attempt to diagnose and fix error
- Global attempt to clean up and restart parser

Some compilers use only the second phase

Diagnosis usually describes the successful recovery or repair

Local Error Recovery

Try to

- Insert symbol before remaining input
- delete next input symbol
- replace next input symbol

What to choose?

- LR table and LL table give candidates

- choices not equally likely

- use likelihood weights

Must have some way to prune the set of possibilities!

- stop as soon as have ‘‘good enough’’ candidate

- put possibilities in equivalence classes

- have fast feasibility test

Lexical considerations

- spelling repair

- token merging

end if

end loop

:   =

How to evaluate choices?

Continue parsing and see whether get another error nearby.

Compute cost of each change and select cheap one

Check semantics of change

Difficulties

- must have explicit provision to prevent looping

- missing **if** - example of error occurring earlier than detected.

    To solve, recovery can sometimes change parsing stack if semantic phase separate or if semantic actions can be undone.

    simple case - ‘unshift’ terminal symbols on top of LR stack

    alternative - make semantic processing lag behind

- Easier to make local recovery if have some context.

To provide context for diagnosis and recovery

Berkeley Pascal: inhibit some default reductions by expanding lookahead in ‘‘important’’ states.

Burke and Fisher: defer all reductions until next **k** shifts - just save rule numbers. (Deferral is easier than undoing the parse.)

Corbett: combine semantic checking with idea of deferral. Make semantic checks on recovery candidates.

Problems for error recovery

if Q then A := X ; else B :=X

The **if** statement will be reduced before the **else** is seen.

Solve by ‘error production’ (which extends the syntax!)

var X,Y: integer;
const max=100;

Relax syntactic ordering constraint and check semantically.

Second - level strategy

Simple strategy - panic mode

Flush part of stack and input until parser is synchronized at a ‘major’ construct such as statement.

Can be used both top down and bottom up.

Advantages: fast, guaranteed to terminate

Disadvantages: not very good information

- Don’t know what’s wrong
- Parser will skip over errors
- Semantic problems - e.g. scope

Refinements

Build in knowledge of scopes and paired bracketing symbols

YACC:

Put error token in stack. Add rules with that token to indicate synchronization points

Example

$$\text{STAT} \rightarrow \text{error} \; ;$$

Back up stack to get Shift on STAT.
Advance input to ;