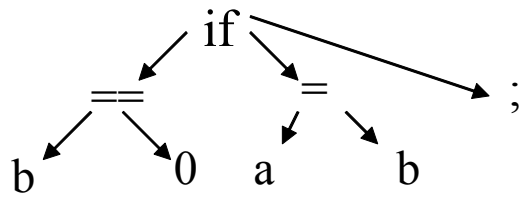
Acknowledgements

The slides for this lecture are a modified versions of the offering by **Prof. Sanjeev K Aggarwal**

Syntax Analysis

- Check syntax and construct abstract syntax tree

if	(	b	==	0	)	a	=	b	;



- Error reporting and recovery
- Model using context free grammars
- Recognize using Push down automata/Table Driven Parsers

Limitations of regular languages

- How to describe language syntax precisely and conveniently. Can regular expressions be used?

- Many languages are not regular, for example, string of balanced parentheses
  - ((((…))))
  - $\{ (^i)^i \mid i \geq 0 \}$
  - There is no regular expression for this language

- A finite automata may repeat states, however, it cannot remember the number of times it has been to a particular state

- ***A more powerful language is needed to describe a valid string of tokens***
  - ***Context-free languages suffice!***

Syntax definition

- Context free grammars
  - a set of tokens (terminal symbols)
  - a set of non terminal symbols
  - a set of productions of the form
    nonterminal →String of terminals & non terminals
  - a start symbol

  <T, N, P, S>

- A grammar derives strings by beginning with a start symbol and repeatedly replacing a non terminal by the right hand side of a production for that non terminal.

- The strings that can be derived from the start symbol of a grammar G form the language L(G) defined by the grammar.

Examples

- String of balanced parentheses
  S → (S) S | ϵ

- Grammar
  list → list + digit
  | list – digit
  | digit
  digit → 0 | 1 | … | 9

  Consists of the language which is a list of digit separated by + or -.

Derivation

list ➔ list + digit
➔ list – digit + digit
➔ digit – digit + digit
➔ 9 – digit + digit
➔ 9 – 5 + digit
➔ 9 – 5 + 2

Therefore, the string 9-5+2 belongs to the language specified by the grammar

The name context free comes from the fact that use of a production X → … does not depend on the context of X

Examples …

- Grammar for Pascal block

  block → begin statements end
  statements → stmt-list | ϵ
  stmt-list → stmt-list ; stmt
  | stmt

Syntax analyzers

- Testing for membership whether w belongs to L(G) is just a "yes" or "no" answer

- However the syntax analyzer
  - Must generate the parse tree
  - Handle errors gracefully if string is not in the language

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar

Derivation

- If there is a production $A \rightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ then $\alpha_1 \overset{+}{\Rightarrow} \alpha_n$
- Given a grammar G and a string w of terminals in $L(G)$ we can write $S \overset{+}{\Rightarrow} w$
- If $S \overset{*}{\Rightarrow} \alpha$ where α is a string of terminals and non terminals of G then we say that α is a sentential form of G

Derivation …

- If in a sentential form only the leftmost non terminal is replaced then it becomes leftmost derivation

- Every leftmost step can be written as
  $wA\gamma \underset{lm}{\Rightarrow}{}^{*} w\delta\gamma$
  where w is a string of terminals and $A \rightarrow \delta$ is a production

- Similarly, right most derivation can be defined

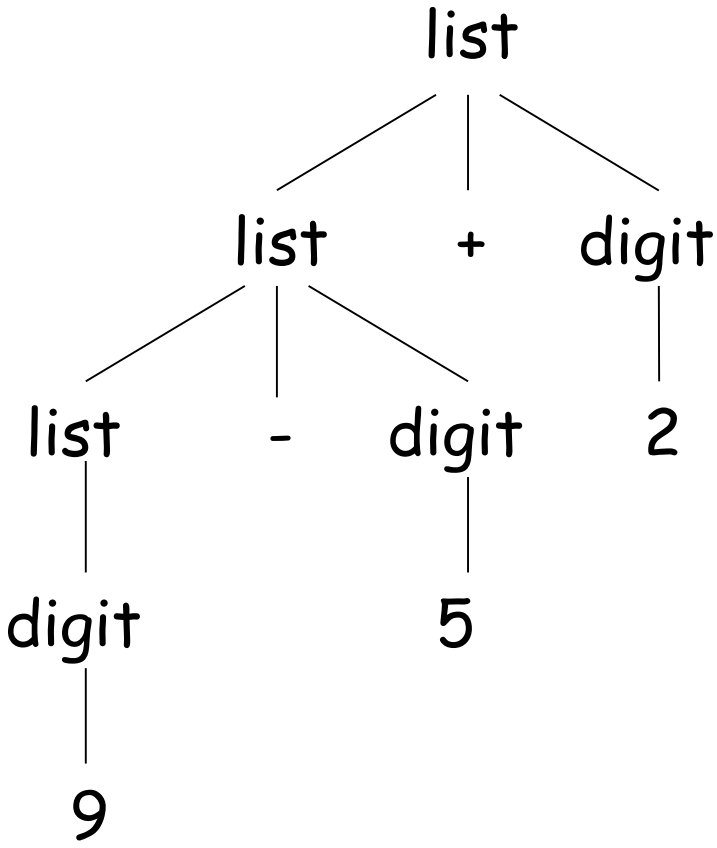
- An ambiguous grammar is one that produces more than one leftmost/rightmost derivation of a sentence

Parse tree

- It shows how the start symbol of a grammar derives a string in the language
- root is labeled by the start symbol
- leaf nodes are labeled by tokens
- Each internal node is labeled by a non terminal
- if A is a non-terminal labeling an internal node and $x_1, x_2, \ldots x_n$ are labels of the children of that node then $A \rightarrow x_1\ x_2\ \ldots\ x_n$ is a production

Example

Parse tree for 9-5+2



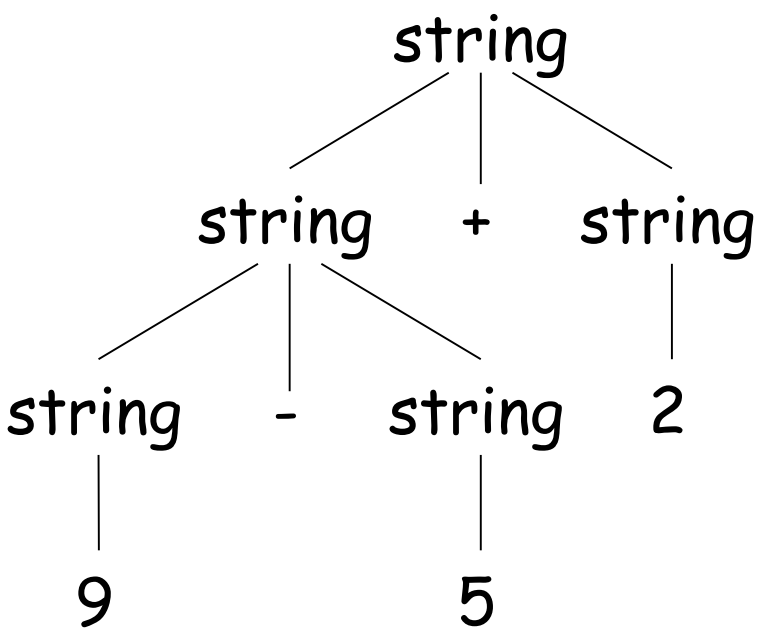
Ambiguity

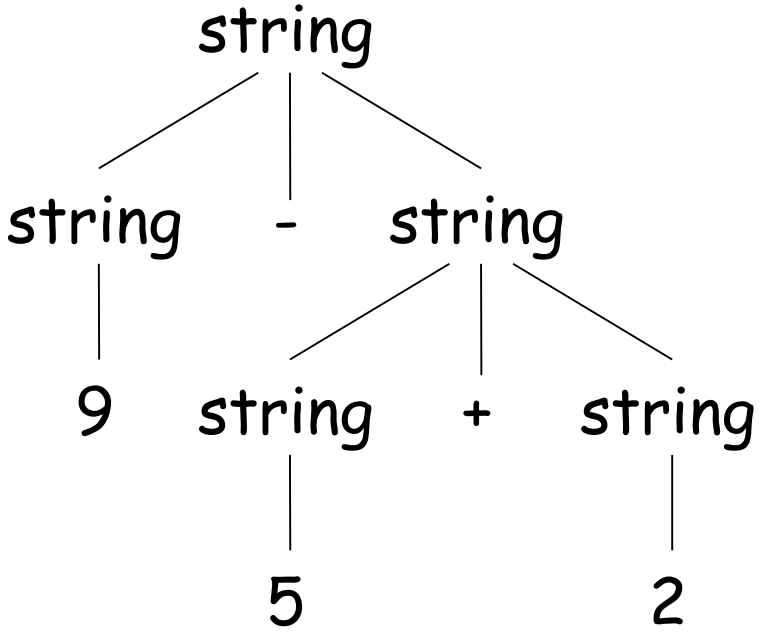
- A Grammar can have more than one parse tree for a string

- Consider grammar

  string → string + string
  | string – string
  | 0 | 1 | … | 9

- String 9-5+2 has two parse trees


string
string
+
string
string
-
string
2
9
5


string
string
-
string
9
string
+
string
5
2

Ambiguity …

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  - Enforce associativity and precedence
  - Rewrite the grammar (cleanest way)
- There are no general techniques for handling ambiguity
- It is impossible to convert automatically an ambiguous grammar to an unambiguous one; **Why?**

Associativity

- If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator
- In a+b+c b is taken by left +
- +, -, *, / are left associative
- ^, = are right associative
- Grammar to generate strings with right associative operators
  right → letter = right | letter
  letter → a| b |…| z

Precedence

- String a+5*2 has two possible interpretations because of two different parse trees corresponding to

  (a+5)*2 and a+(5*2)

- Precedence determines the correct interpretation.

Associativity verus Precedence

- What is the difference?

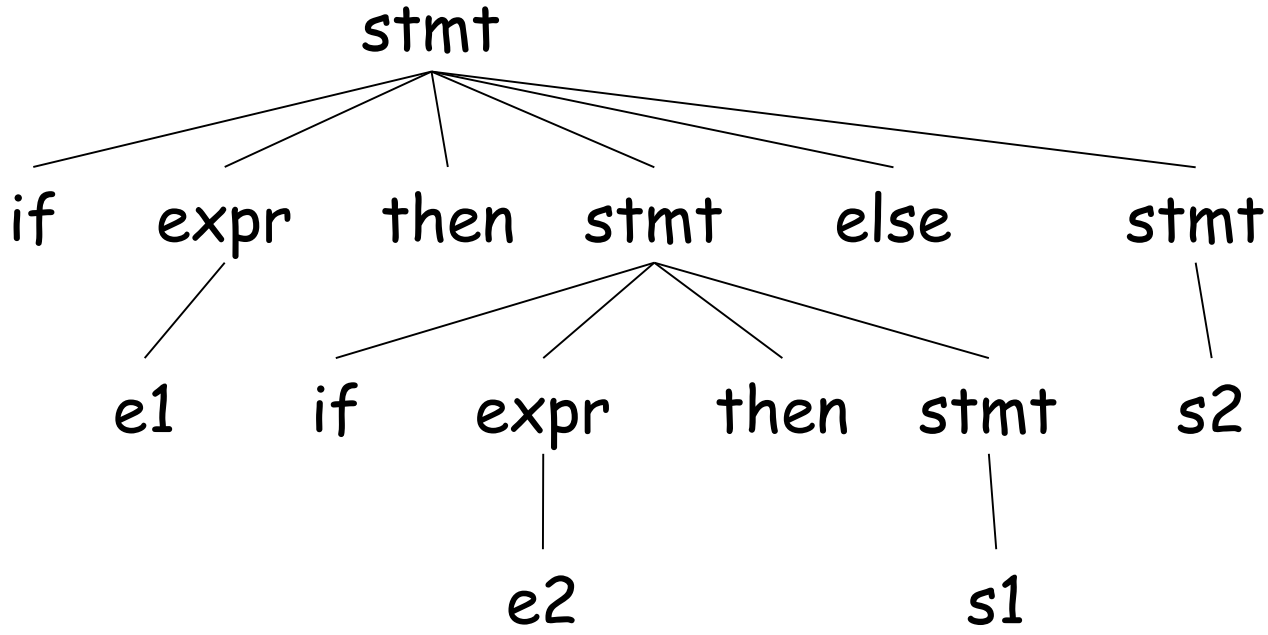
Ambiguity

- Dangling else problem

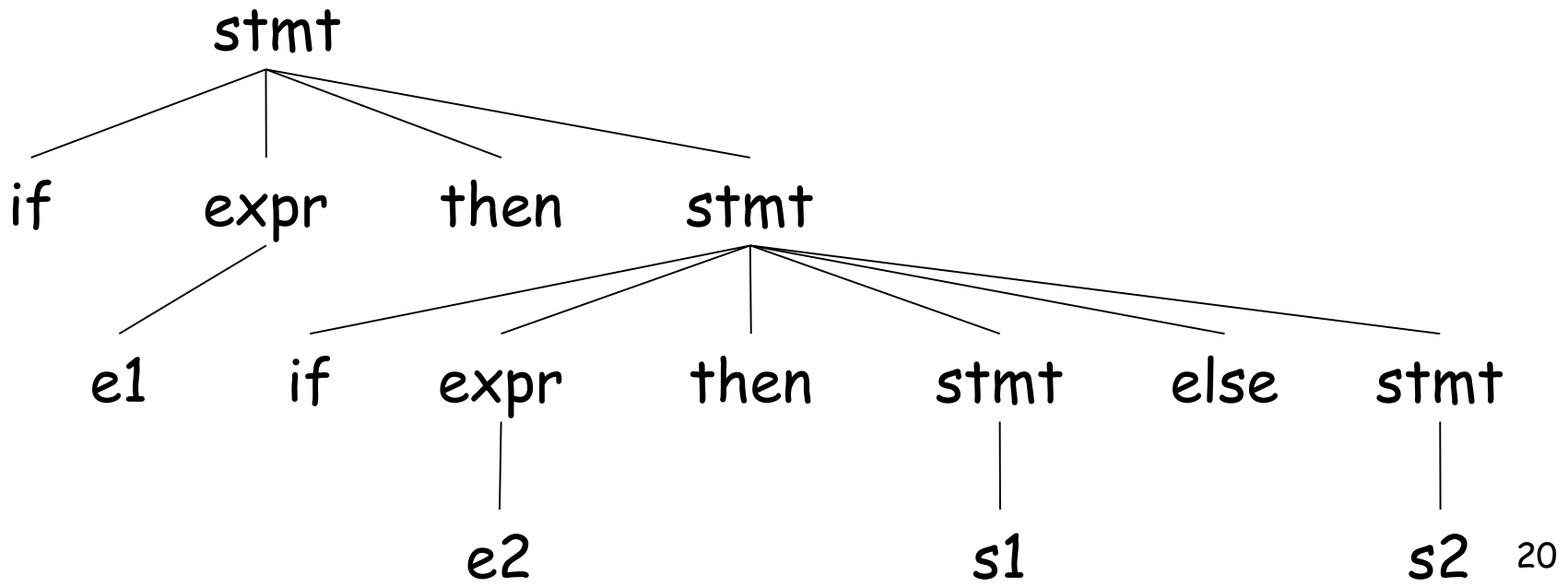
  Stmt $\rightarrow$ if expr then stmt
  | if expr then stmt else stmt

- according to this grammar, string
  if e1 then if e2 then S1 else S2
  has two parse trees

```
if e1
  then if e2
           then s1
  else s2
```




```
if e1
  then if e2
           then s1
           else s2
```



Resolving dangling else problem

- General rule: match each **else** with the closest previous **then.** The grammar can be rewritten as

stmt → matched-stmt
| unmatched-stmt

matched-stmt → if expr then matched-stmt
else matched-stmt
| others

unmatched-stmt → if expr then stmt
| if expr then matched-stmt
else unmatched-stmt

Parsing

- Process of determination whether a string can be generated by a grammar

- Parsing falls in two categories:

  – Top-down parsing:
    Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals)

  – Bottom-up parsing:
    Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol)

Top-down parsing

- Recursive Descent Parser

Recursive Descent Parsing (First Cut)

- Construction of a parse tree is done by starting the root labeled by a start symbol
- repeat following steps
  - at a node labeled with non terminal A select one of the productions of A and construct children nodes (Which production?)
  - find the next node at which subtree is Constructed (Which node?)
  - Backtrack on failure

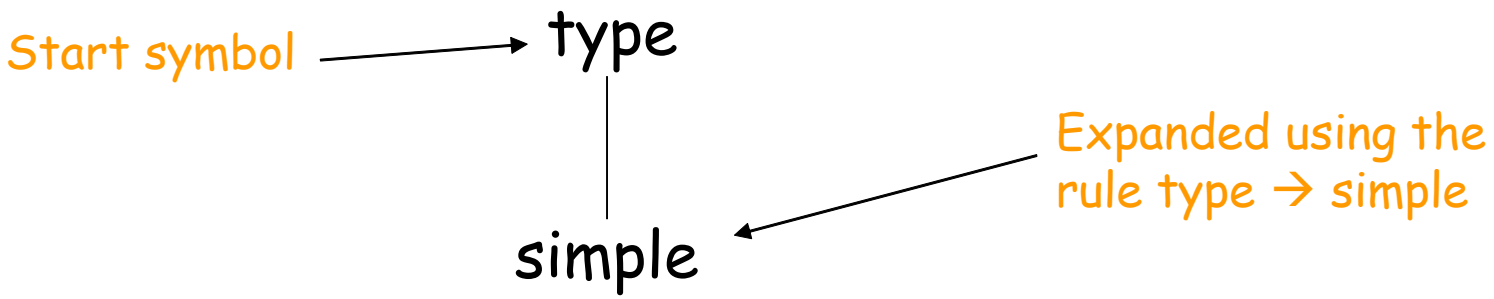
Example: Top down Parsing

- Following grammar generates types of Pascal

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

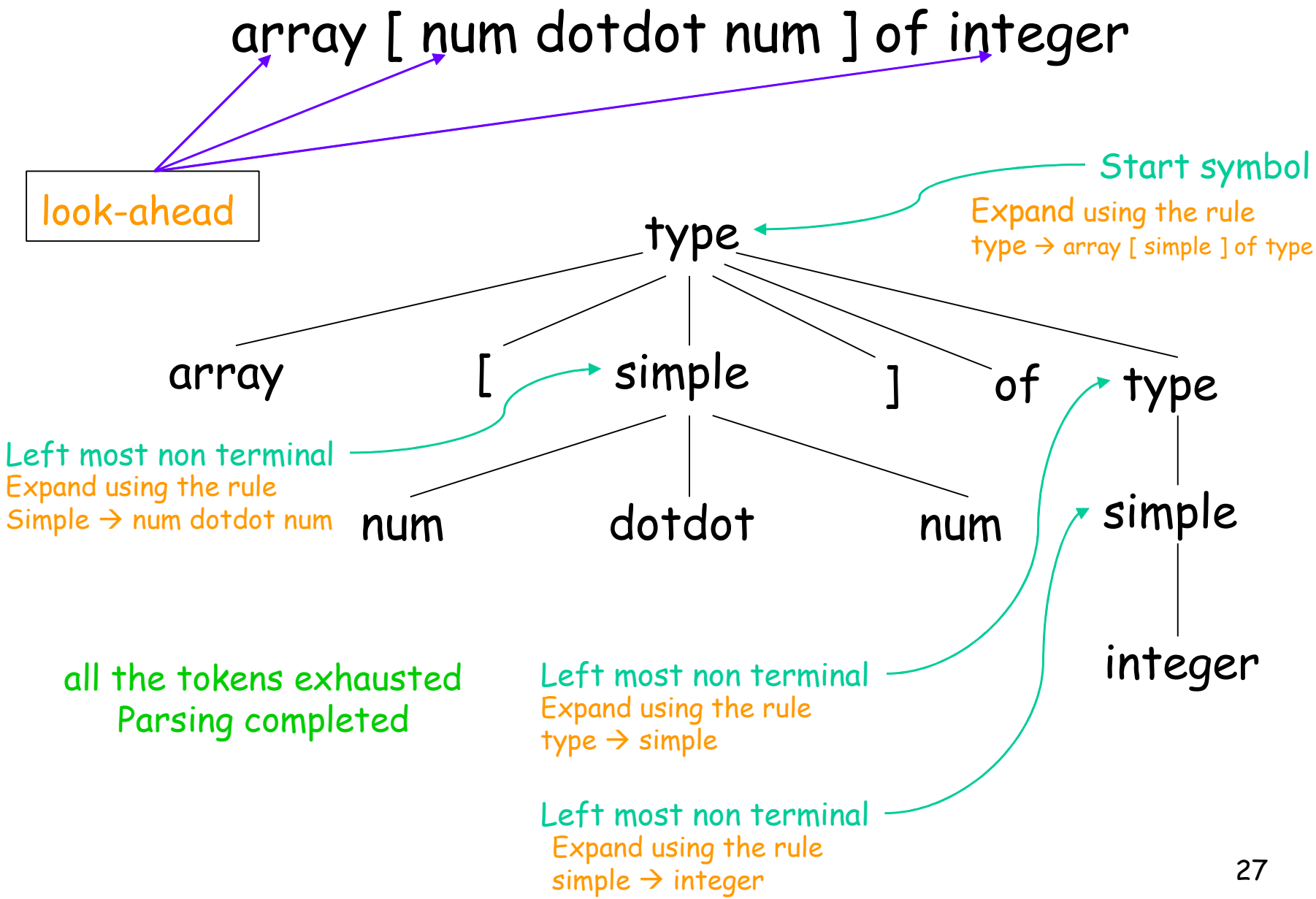
simple → integer
| char
| num dotdot num

- Parse
  array [num dotdot num] of integer



- Cannot proceed as non terminal "simple" never generates a string beginning with token "array". Therefore, requires back-tracking.

- Back-tracking is not desirable!
  - Soln: "look-ahead" token (restricts the class of grammars)


array [num dotdot num] of integer
look-ahead
Start symbol
Expand using the rule
type → array [simple] of type
type
array
[
simple
]
of
type
Left most non terminal
Expand using the rule
Simple → num dotdot num
num
dotdot
num
simple
integer
all the tokens exhausted
Parsing completed
Left most non terminal
Expand using the rule
type → simple
Left most non terminal
Expand using the rule
simple → integer

Recursive descent parsing

First set:

Let there be a production

$$A \rightarrow \alpha$$

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}

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 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;
```

Problems

- When can our alorithm fail?
  - What about left-recursive grammars?
  - When FirstSet cannot completely disambiguate which rule has to be applied?

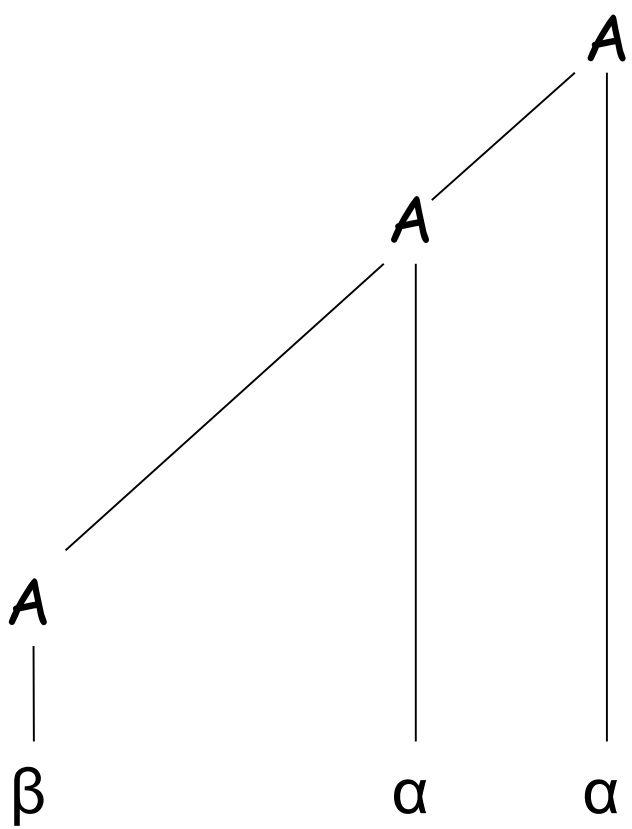
Left recursion

- A top down parser with production
  $A \rightarrow A\ \alpha$ may loop forever

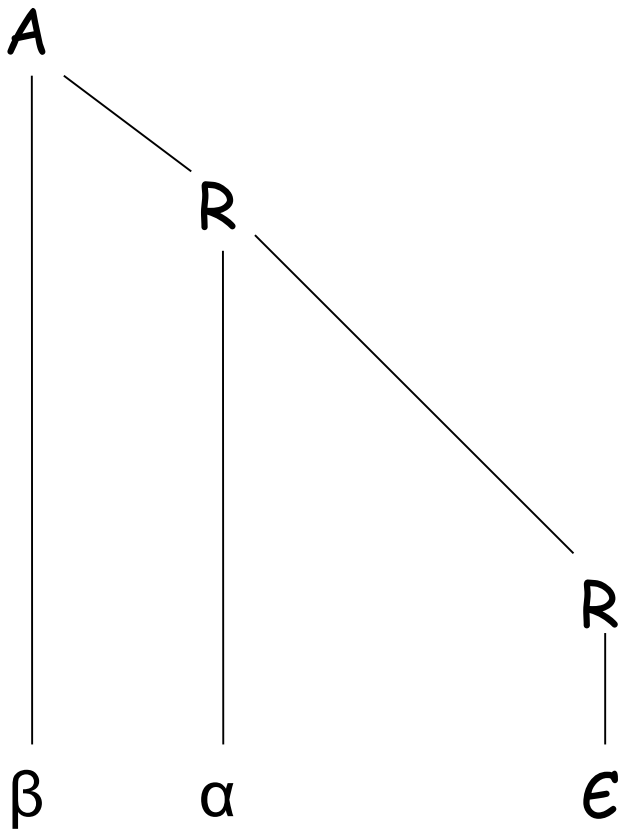
- From the grammar $A \rightarrow A\ \alpha \mid \beta$
  left recursion may be eliminated by transforming the grammar to

  $A \rightarrow \beta\ R$
  $R \rightarrow \alpha\ R \mid \varepsilon$

Parse tree corresponding to a left recursive grammar



Parse tree corresponding to the modified grammar



Both the trees generate string βα*

Example

- Consider grammar for arithmetic expressions

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

- After removal of left recursion the grammar becomes

  $E \rightarrow T\ E'$
  $E' \rightarrow + T\ E' \mid \epsilon$
  $T \rightarrow F\ T'$
  $T' \rightarrow * F\ T' \mid \epsilon$
  $F \rightarrow (E) \mid id$

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

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

Left recursion hidden due to many productions

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

  $S \rightarrow Aa \mid b$
  $A \rightarrow Ac \mid Sd \mid \epsilon$

  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

Removal of left recursion due to many productions …

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

  S → Aa | b
  A → Ac | Aad | bd | ϵ

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

  S → Aa | b
  A → bdA' | A'
  A' → cA' | adA' | ϵ

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 $\alpha A'$

  we can then expand A' to β_1 or β_2

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

  transforms to

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

Dangling else problem again

Dangling else problem can be handled by left factoring

stmt → if expr then stmt else stmt
| if expr then stmt

can be transformed to

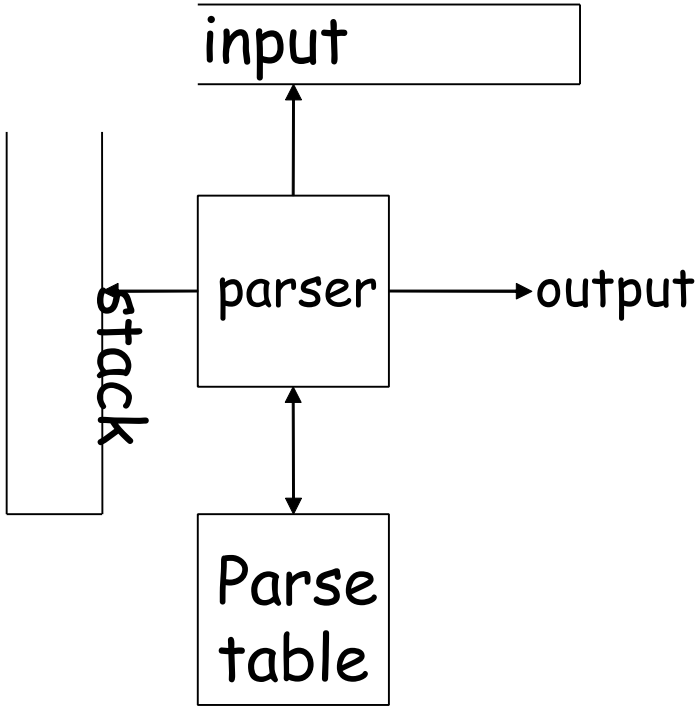
stmt → if expr then stmt S'
S' → else stmt | ϵ

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

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

Example

- Consider the grammar

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \epsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \epsilon$

$F \rightarrow (\ E\ ) \mid id$

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 '+'

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 → UVW}
        then begin pop(X); push(W,V,U)
            end
        else error

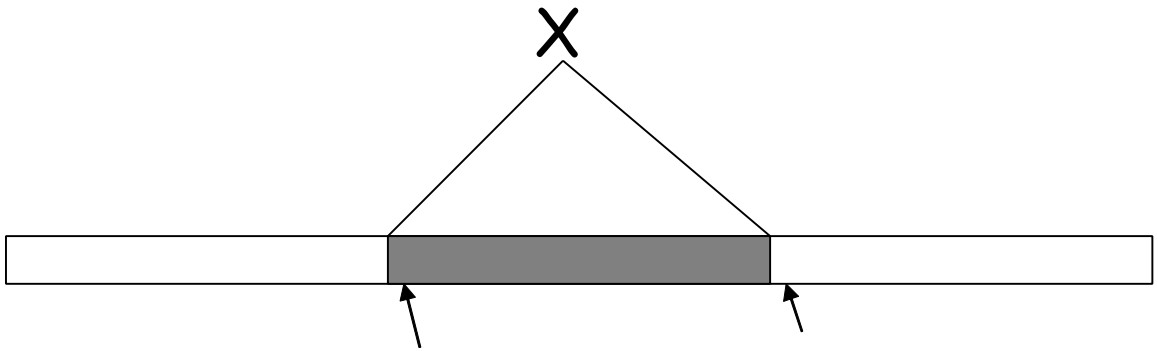
Example

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'

Example …

Stack	input	action
$E'T'F	id * id $	expand by F→id
$E'T'id	id * id $	pop id and ip++
$E'T'	* id $	expand by T'→*FT'
$E'T'F*	* id $	pop * and ip++
$E'T'F	id $	expand by F→id
$E'T'id	id $	pop id and ip++
$E'T'	$	expand by T'→ϵ
$E'	$	expand by E'→ϵ
$	$	halt

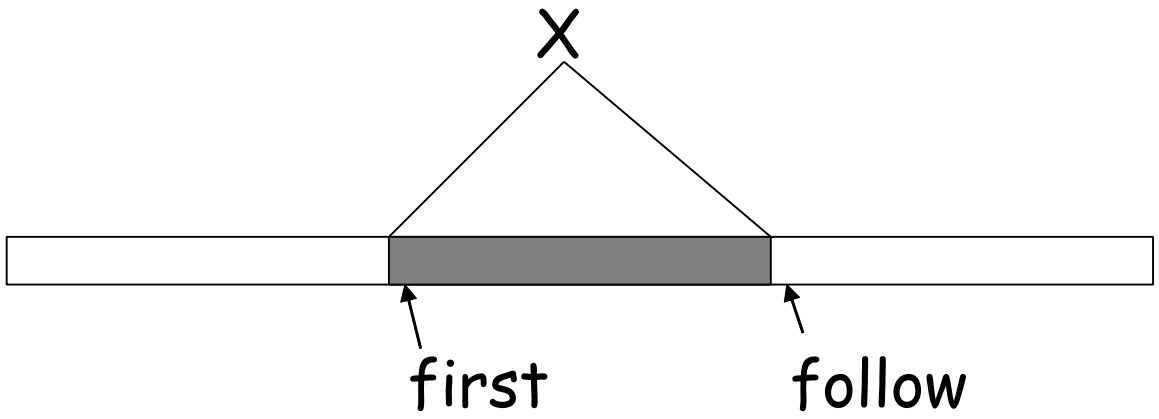
Constructing parse table



- How can we select a rule to expand X, say out of X->AB, X->CD?
- Select according to the lookahead being in the *first* characters in A and C...
- What if you also have a rule X->ε ?
- Select X->ε only if X can be followed by the lookahead in some rule...

Constructing parse table

- Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production
- First(α) for a string of terminals and non terminals α is
  - Set of symbols that might begin the fully expanded (made of only tokens) version of α
- Follow(X) for a non terminal X is
  - set of symbols that might follow the derivation of X in the input stream



Compute first sets

- If X is a terminal symbol then First(X) = {X}
- If $X \rightarrow \epsilon$ is a production then ϵ is in First(X)
- If X is a non terminal
  and $X \rightarrow Y_1 Y_2 \ldots Y_k$ 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)

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}

Compute follow sets

1. Place $ in follow(S)

2. If there is a production $A \rightarrow \alpha B \beta$
   then everything in first(β) (except ε) is in follow(B)

3. If there is a production $A \rightarrow \alpha B$
   then everything in follow(A) is in follow(B)

4. If there is a production $A \rightarrow \alpha B \beta$
   and First(β) contains ε
   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

Example

- For the expression grammar

  E → T E'

  E' → + T E' | ϵ

  T → F T'

  T' → * F T' | ϵ

  F → (E) | id

  follow(E) = follow(E') = { $,) }
  follow(T) = follow(T') = { $,), + }
  follow(F) = { $,), +, *}

Construction of parse table

- for each production $A \rightarrow \alpha$ do
  - for each terminal 'a' in first(α)

    $M[A,a] = A \rightarrow \alpha$

  - If ϵ is in First(α)

    $M[A,b] = A \rightarrow \alpha$

    for each terminal b in follow(A)

  - If ε is in First(α) and \$ is in follow(A)

    $M[A,\$] = A \rightarrow \alpha$

- A grammar whose parse table has no multiple entries is called LL(1)

LL Parser Generators

- ANTLR
- LLGen
- LLnextGen
- Many more like Tiny Parser Generator, Wei parser generator, SLK parser generator, Yapps ….

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

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

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

Phrase level recovery

- Make local correction to the input
- Works only in limited situations
  - A common programming error which is easily detected
  - For example insert a ";" after closing "}" of a class definition
- Does not work very well!

Error productions

- Add erroneous constructs as productions in the grammar
- Works only for most common mistakes which can be easily identified
- Essentially makes common errors as part of the grammar
- Complicates the grammar and does not work very well

Global corrections

- Considering the program as a whole find a correct "nearby" program
- Nearness may be measured using certain metric
- PL/C compiler implemented this scheme: anything could be compiled!
- It is complicated and not a very good idea!

Error Recovery in LL(1) parser

- Error occurs when a parse table entry M[A,a] is empty
- Skip symbols in the input until a token in a selected set (synch) appears
- Place symbols in follow(A) in synch set. Skip tokens until an element in follow(A) is seen.
  Pop(A) and continue parsing
- Add symbol in first(A) in synch set. Then it may be possible to resume parsing according to A if a symbol in first(A) appears in input.

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root

OR

- Reduce a string w of input to start symbol of grammar

Consider a grammar

S → aABe
A → Abc | b
B → d

And reduction of a string

a b b c d e
a A b c d e
a A d e
a A B e
S

Right most derivation

S ➔ a A B e
➔ a A d e
➔ a A b c d e
➔ a b b c d e

Bottom up parsing …

- A more powerful parsing technique
- LR grammars – more expressive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible
- Allows better error recovery

Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character "."
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is .w

Shift reduce parsing …

- Bottom up parsing has two actions

- Shift: move terminal symbol from right string to left string

  if string before shift is α.pqr

  then string after shift is αp.qr

- Reduce: immediately on the left of "." identify a string same as RHS of a production and replace it by LHS

  if string before reduce action is αβ.pqr

  and A→β is a production

  then string after reduction is αA.pqr

Example

Assume grammar is E → E+E | E*E | id
Parse id*id+id

String	**action**
.id*id+id	shift
id.*id+id	reduce E→id
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce E→id
E*E.+id	reduce E→E*E
E.+id	shift
E+.id	shift
E+id.	Reduce E→id
E+E.	Reduce E→E+E
E.	ACCEPT

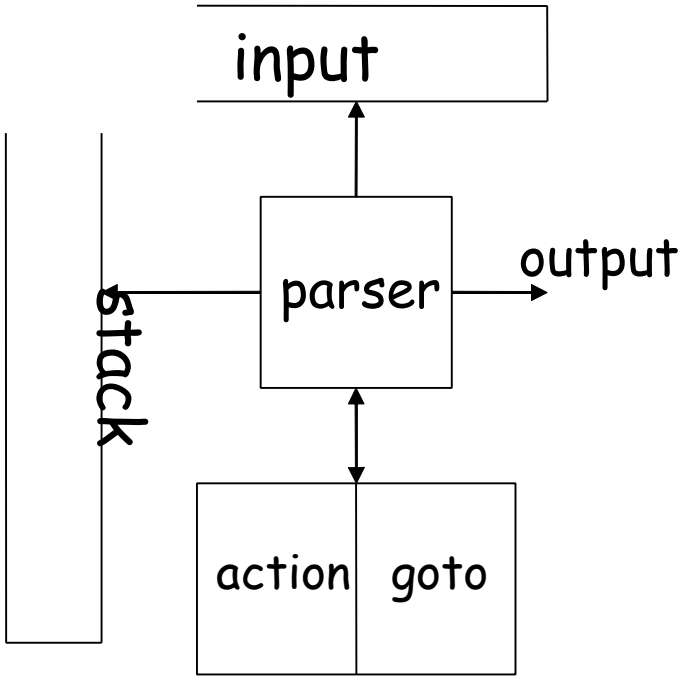
Shift reduce parsing …

- Symbols on the left of "." are kept on a stack
  - Top of the stack is at "."
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack
- The most important issues:
  - when to shift and when to reduce
  - what rule to use for reduce
- Reduce action should be taken only if the result can be reduced to the start symbol

Parser states

- Goal is to know the valid reductions at any given point
- Summarize all possible stack prefixes α as a parser state
- Parser state is defined by a DFA state that reads in the stack α
- Accept states of DFA are unique reductions

LR parsing



Parse table

- Input contains the input string.
- Stack contains a string of the form $S_0X_1S_1X_2\ldots\ldots X_nS_n$ where each X_i is a grammar symbol and each S_i is a state.
- Tables contain action and goto parts.
- action table is indexed by state and terminal symbols.
- goto table is indexed by state and non terminal symbols.

Example

Consider the grammar
And its parse table

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

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_i is current input symbol

- Action $[S_i,a_i]$ can have four values

1. shift a_i to the stack and goto state S_j
2. reduce by a rule
3. Accept
4. error

Configurations in LR parser

Stack: $S_0X_1S_1X_2\ldots X_mS_m$ Input: $a_ia_{i+1}\ldots a_n\$$

- If action[S_m,a_i] = shift S
  Then the configuration becomes
  Stack: $S_0X_1S_1\ldots\ldots X_mS_ma_iS$ Input: $a_{i+1}\ldots a_n\$$

- If action[S_m,a_i] = reduce $A\rightarrow\beta$
  Then the configuration becomes
  Stack: $S_0X_1S_1\ldots X_{m-r}S_{m-r}AS$ Input: $a_ia_{i+1}\ldots a_n\$$
  Where r = $|\beta|$ and S = goto[S_{m-r},A]

- If action[S_m,a_i] = accept
  Then parsing is completed. HALT

- If action[S_m,a_i] = error
  Then invoke error recovery routine.

Example

Consider the grammar
And its parse table

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

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parse id + id * id

Stack	Input	Action
0	id+id*id$	shift 5
0 id 5	+id*id$	reduce by F→id
0 F 3	+id*id$	reduce by T→F
0 T 2	+id*id$	reduce by E→T
0 E 1	+id*id$	shift 6
0 E 1 + 6	id*id$	shift 5
0 E 1 + 6 id 5	*id$	reduce by F→id
0 E 1 + 6 F 3	*id$	reduce by T→F
0 E 1 + 6 T 9	*id$	shift 7
0 E 1 + 6 T 9 * 7	id$	shift 5
0 E 1 + 6 T 9 * 7 id 5	$	reduce by F→id
0 E 1 + 6 T 9 * 7 F 10	$	reduce by T→T*F
0 E 1 + 6 T 9	$	reduce by E→E+T
0 E 1	$	ACCEPT

LR parsing Algorithm

Initial state: Stack: S_0 Input: w$

```
Loop{
   if action[S,a] = shift S'
      then push(a); push(S'); ip++
      else if action[S,a] = reduce A→β
               then pop (2*|β|)  symbols;
                    push(A); push (goto[S'',A])
                     (S'' is the state after popping symbols)
               else if action[S,a] = accept
                      then exit
                      else error
      }
```

Example

Consider the grammar
And its parse table

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

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Constructing parse table

Augment the grammar

- *G* is a grammar with start symbol S
- The augmented grammar *G'* for *G* has a new start symbol S' and an additional production S' → S
- When the parser reduces by this rule it will stop with accept

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - Which production to use for reduction?

- Sometimes parser can reduce but it should not:
  $X \rightarrow \epsilon$ can always be reduced!

- Sometimes parser can reduce in different ways!

- Given stack δ and input symbol a, should the parser
  - Shift a onto stack (making it δa)
  - Reduce by some production $A \rightarrow \beta$ assuming that stack has form $\alpha\beta$ (making it αA)
  - Stack can have many combinations of $\alpha\beta$
  - How to keep track of length of β?

The Shift-Reduce Dilemma

- [Strategy] Reduce whenever you can
  - Grammar: E -> id+E | id
  - String: id + id + id
  - Does not work!
- [Effective Strategy] Reduce only if the resultant string can be reduced to the start symbol

The Shift-Reduce Dilemma

- Reduce $\alpha Aw \Rightarrow_{rm} \alpha\beta w$
  only if $S \Rightarrow_{rm}^{*} \alpha Aw$

Reduce if and only if the reduction keeps us on the correct path - that of the rightmost derivation

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If $S \rightarrow^{rm*} \alpha Aw \rightarrow^{rm} \alpha\beta w$ then β (corresponding to production $A \rightarrow \beta$) in the position following α is a handle of $\alpha\beta w$. The string w consists of only terminal symbols
- We only want to reduce handle and not any rhs
- Handle pruning: If β is a handle and $A \rightarrow \beta$ is a production then replace β by A
- A right most derivation in reverse can be obtained by handle pruning.

Handles …

- Handles always appear at the top of the stack and never inside it

  [Prove it - proof by induction]

- This makes stack a suitable data structure

- Bottom up parsing is based on recognizing handles

Handle always appears on the top

- Consider two cases of right most derivation to verify the fact that handle appears on the top of the stack
  - S → αAz → αβByz → αβγyz
  - S → αBxAz → αBxyz → αγxyz

Handle always appears on the top

Case I: S ➔ αAz ➔ αβByz ➔ αβγyz

stack	**input**	**action**
αβγ	yz	reduce by B→γ
αβB	yz	shift y
αβBy	z	reduce by A→ By
αA	z	

Case II: S ➔ αBxAz ➔ αBxyz ➔ αγxyz

stack	**input**	**action**
αγ	xyz	reduce by B→γ
αB	xyz	shift x
αBx	yz	shift y
αBxy	z	reduce A→y
αBxA	z	

Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce

- However, what happens when there is a choice
  - What action to take in case both shift and reduce are valid?

    shift-reduce conflict
  - Which rule to use for reduction if reduction is possible by more than one rule?

    reduce-reduce conflict

- Conflicts come either because of ambiguous grammars or parsing method is not powerful enough

Shift reduce conflict

Consider the grammar E → E+E | E*E | id

and input id+id*id

stack	input	action
E+E	*id	reduce by E→E+E
E	*id	shift
E*	id	shift
E*id		reduce by E→id
E*E		reduce byE→E*E
E		

stack	input	action
E+E	*id	shift
E+E*	id	shift
E+E*id		reduce by E→id
E+E*E		reduce by E→E*E
E+E		reduce by E→E+E
E		

Reduce reduce conflict

Consider the grammar M → R+R | R+c | R

R → c

and input c+c

Stack	input	action
	c+c	shift
c	+c	reduce by R→c
R	+c	shift
R+	c	shift
R+c		reduce by R→c
R+R		reduce by M→R+R
M		

Stack	input	action
	c+c	shift
c	+c	reduce by R→c
R	+c	shift
R+	c	shift
R+c		reduce by M→R+c
M		

How to recognize handles?

- Let us understand the structure of the stack:
- $[a_1 a_2 a_3 \ldots a_n]$
- a_i is a **prefix of a handle**
- a_n transforms to a handle (via shifts) and then is reduced to non-terminal: $[a_1 a_2 a_3 \ldots a_{n-1} X]$
- $a_{n-1} X$ transforms to a handle (via shifts) and then is reduced to a non-terminal: $[a_1 a_2 a_3 \ldots a_{n-2} Y]$
- so on...

How to recognize handles?

- Can we identify when the stack has a "correct" structure (sequence of prefixes of handles)?
- Consider the language over these "viable" stack configurations (read as strings over terminals and non-terminals in the grammar)
- A "viable" stack configuration α is such that there is a possible (yet to be read) string w such that αw is accepted by the grammara and has a "correct" structure
- We will refer to such viable stack configurations as a "viable prefix"

Viable prefixes

- α is a viable prefix of the grammar if
  - There is a w such that αw is a right sentential form
  - α.w is a configuration of the shift reduce parser

- Not all prefixes of right sentential forms are prefixes; a viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.

- As long as the parser has viable prefixes on the stack no parser error has been seen

- How to recognize these viable prefixes?

Viable prefixes

- Theorem: The set of viable prefixes is a regular language
  (not obvious)

- How to recognize? Construct an automaton that accepts viable prefixes

- For grammar with production A → XYZ, what are the possible viable prefixes? {"", "X", "XY", "XYZ"}

LR(0) items

- An LR(0) item of a grammar G is a production of G with a special symbol "." at some position of the right side

- Thus production A→XYZ gives four LR(0) items
  A → .XYZ
  A → X.YZ
  A → XY.Z
  A → XYZ.

- An item indicates how much of a production has been seen at a point in the process of parsing

  – Symbols on the left of "." are already on the stacks

  – Symbols on the right of "." are ***expected*** in the input

Expected strings...

- For the item A → α.Bβ, what we are expecting to see "Bβ"
- What other strings can we expect? If B->γ is a production, we can expect to see "γ" (followed by other things)

Closure

- **Closure** of a state adds items for all productions whose LHS occurs in an item in the state, just after "."
  - Set of possible productions to be reduced next
  - Added items have "." located at the beginning
  - No symbol of these items is on the stack as yet

Closure operation

- If I is a set of items for a grammar G then closure(I) is a set constructed as follows:
  - Every item in I is in closure (I)
  - If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production then $B \rightarrow .\gamma$ is in closure(I)
- Intuitively $A \rightarrow \alpha.B\beta$ indicates that we might see a string derivable from $B\beta$ as input
- If input $B \rightarrow \gamma$ is a production then we might see a string derivable from γ at this point

Example

Consider the grammar

E′ → E
E → E + T | T
T → T * F | F
F → (E) | id

If I is { E′ → .E } then closure(I) is

E′ → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .id
F → .(E)

Start State

- Start state of DFA is an empty stack corresponding to S'→.S item
  - This means no input has been seen
  - The parser expects to see a string derived from S

Applying symbols in a state

- In the new state include all the items that have appropriate input symbol just after the "."

- Advance "." in those items and take closure

State Transitions: Goto operation

- Goto(I,X) , where I is a set of items and X is a grammar symbol,
  - is closure of set of item $A \rightarrow \alpha X.\beta$
  - such that $A \rightarrow \alpha.X\beta$ is in I

- Intuitively if I is a set of items for some valid prefix α then goto(I,X) is set of valid items for prefix αX

- If I is { E'→E. , E→E. + T } then goto(I,+) is

  E → E + .T
  T → .T * F
  T → .F
  F → .(E)
  F → .id

Sets of items

C : Collection of sets of LR(0) items for grammar *G'*

C = { closure ({ S' → .S }) }

repeat

for each set of items I in *C*

and each grammar symbol X

such that goto (I,X) is not empty and not in *C*

ADD goto(I,X) to *C*

until no more additions

Example

Grammar:

E' → E
E → E+T | T
T → T*F | F
F → (E) | id

I_0: closure(E'→.E)

E' → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I_1: goto(I_0,E)

E' → E.
E → E. + T

I_2: goto(I_0,T)

E → T.
T → T. *F

I_3: goto(I_0,F)

T → F.

I_4: goto(I_0,( )

F → (.E)
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I_5: goto(I_0,id)

F → id.

I_6: goto(I_1,+)
E → E + .T
T → .T * F
T → .F
F → .(E)
F → .id

I_7: goto(I_2,*)
T → T * .F
F →.(E)
F → .id

I_8: goto(I_4,E)
F → (E.)
E → E. + T

goto(I_4,T) is I_2
goto(I_4,F) is I_3
goto(I_4,( ) is I_4
goto(I_4,id) is I_5

I_9: goto(I_6,T)
E → E + T.
T → T. * F

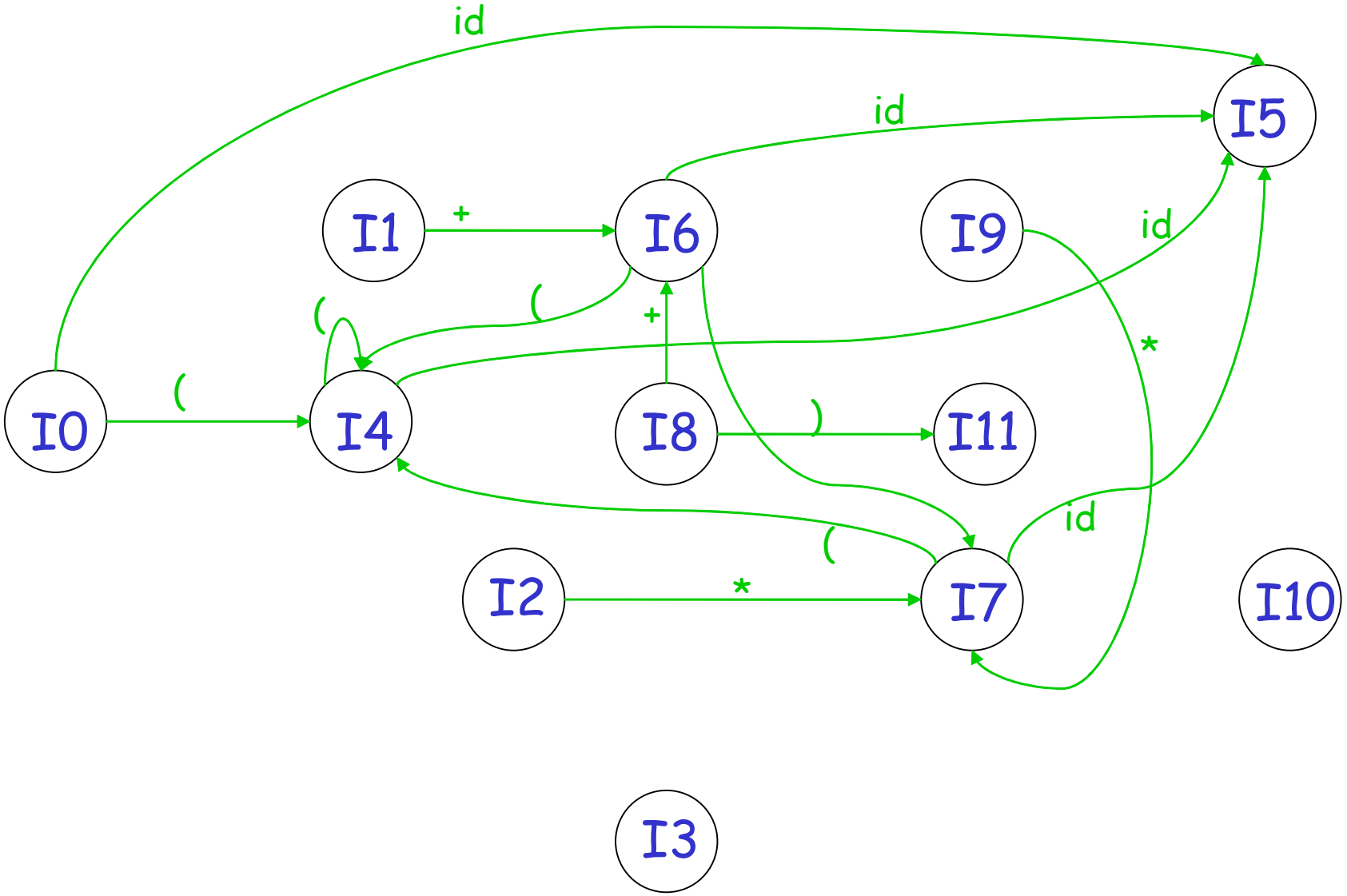
goto(I_6,F) is I_3
goto(I_6,( ) is I_4
goto(I_6,id) is I_5

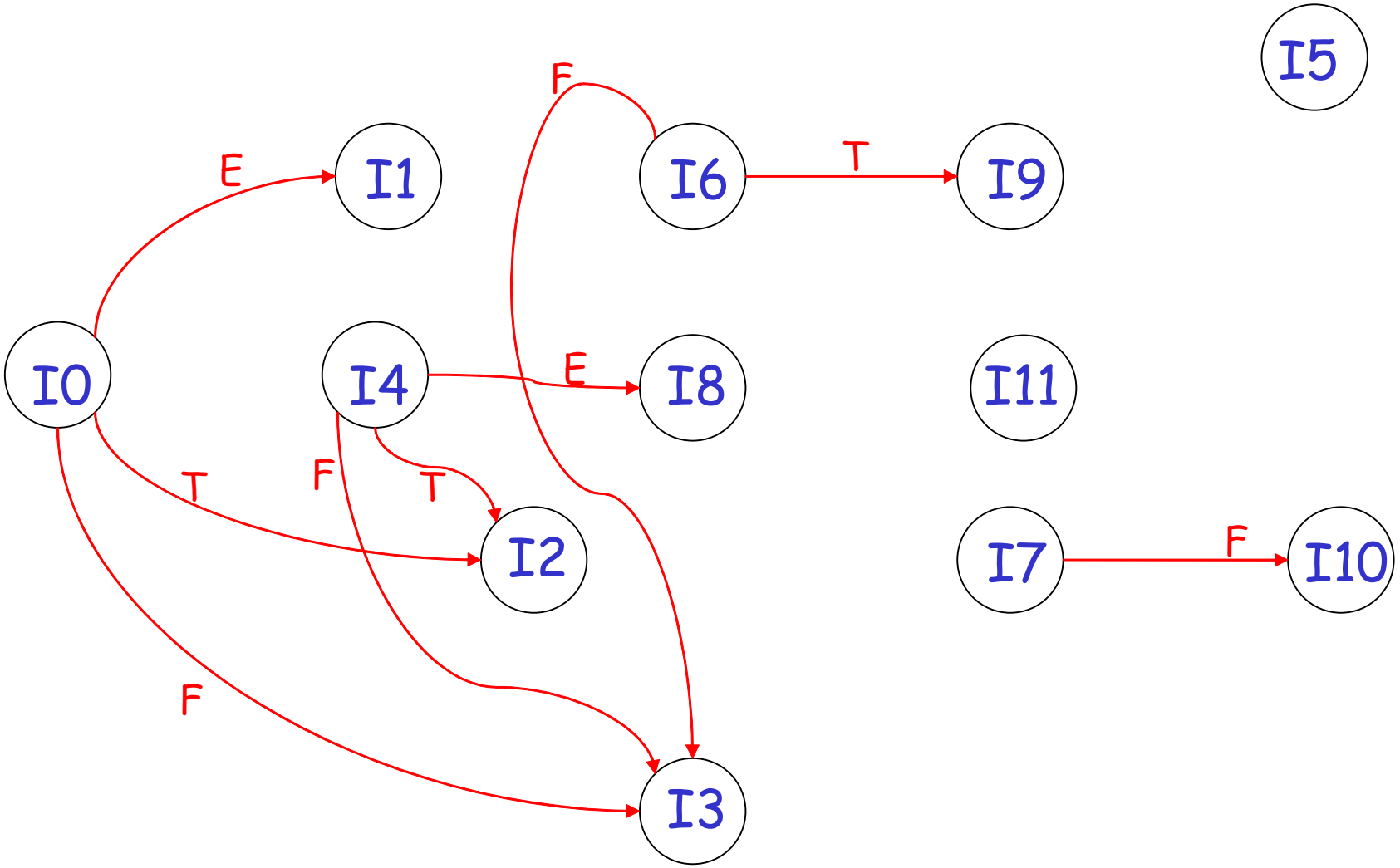
I_{10}: goto(I_7,F)
T → T * F.

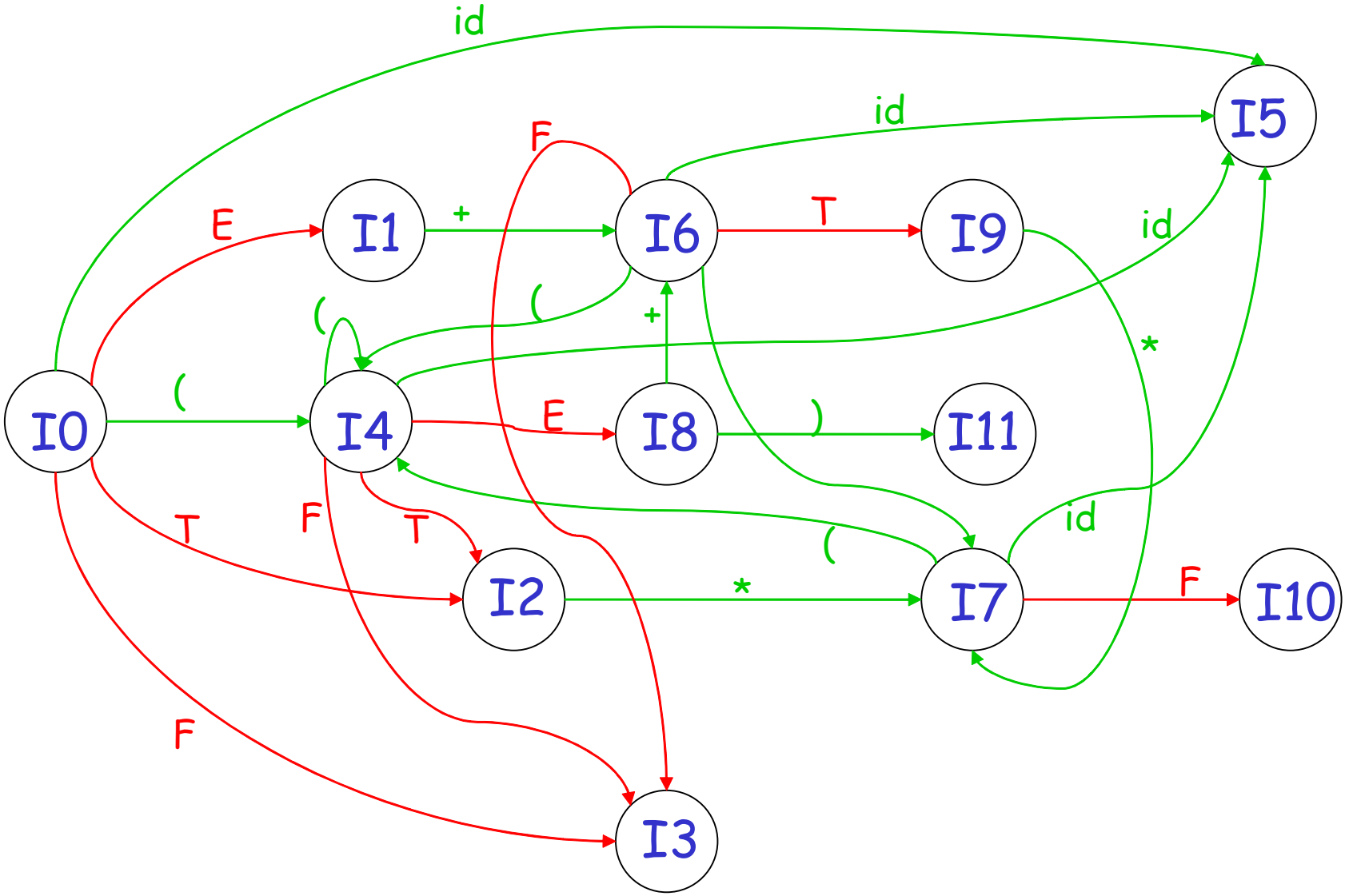
goto(I_7,( ) is I_4
goto(I_7,id) is I_5

I_{11}: goto(I_8,))
F → (E).

goto(I_8,+) is I_6
goto(I_9,*) is I_7


id
id
I5
I1
+
I6
I9
id
(
(
+
*
I0
(
I4
I8
)
I11
id
(
I2
*
I7
I10
I3


I5
F
E
I1
I6
T
I9
I0
I4
E
I8
I11
T
F
T
I2
I7
F
I10
F
I3


id
I5
id
F
E
I1
+
I6
T
I9
id
(
(
+
*
I0
(
I4
E
I8
)
I11
id
T
F
T
(
I2
*
I7
F
I10
F
I3

Lookaheads

- How many tokens of lookahead are we using?

Algorithm

- Given a configuration of the stack, read the string α on the stack (from bottom to top)
- Run the string through the DFA to check if α is a viable prefix; signal error if it is not
- Consult the final state of the DFA to decide on the parser's step

Two Questions

- Why does the DFA accept viable prefixes?
  - what about "E+T*"
  - stack config [a1 | a2 | a3] = [ϵ | E+ | T*]
- How can we construct the parse table from the DFA?

Algorithm

- Given a configuration of the stack, read the string α on the stack (from bottom to top)
- Run the string through the DFA
  to check if α is a viable prefix;
  signal error if it is not
- *Consult the final state of the DFA to decide on the parser's step* - **Lookup on the Parse table**

Can we make it more efficient?

Making it efficient

- The string on the stack can change only at the end (i.e. at the top of the stack) due to a shift/reduce rule
- Remember the state in the DFA after each transition through the string
- Where to remember it - on the stack itself!

Algorithm

- *Given a configuration of the stack, read the string* α *on the stack (from bottom to top)* - **Not needed**
- *Run the string through the DFA to check if* α *is a viable prefix; signal error if it is not* - **Lookup the state on the stack**
- *Consult the final state of the DFA to decide on the parser's step* - **Lookup on the Parse table and record the new state on the stack**

Can we make it more efficient?

When does it not work?

- E -> id + E | id
- Consider the state:
  - E->id.+E; E->id.
- LR (0) parsing has a [Shift-Reduce Conflict]

Construct SLR parse table

- Construct $C=\{I_0, \ldots, I_n\}$ the collection of sets of LR(0) items
- If $A \rightarrow \alpha.a\beta$ is in I_i and goto(I_i,a) = I_j
  then action[i,a] = shift j
- If $A \rightarrow \alpha.$ is in I_i
  then action[i,a] = reduce $A \rightarrow \alpha$ **for all a in follow(A)**
- If $S' \rightarrow S.$ is in I_i then action[i,$] = accept
- If goto(I_i,A) = I_j
  then goto[i,A]=j for all non terminals A
- All entries not defined are errors

Notes

- This method of parsing is called SLR (Simple LR)
- LR parsers accept LR(k) languages
  - L stands for left to right scan of input
  - R stands for rightmost derivation
  - k stands for number of lookahead token
- SLR is the simplest of the LR parsing methods. SLR is too weak to handle most languages!
- If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous
- All SLR grammars are unambiguous
- Are all unambiguous grammars in SLR?

Example

- Consider following grammar and its SLR parse table:

S′ → S
S → L = R
S → R
L → *R
L → id
R → L

I_0: S′ → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

I_1: goto(I_0, S)
S′ → S.

I_2: goto(I_0, L)
S → L.=R
R → L.

Assignment (not to be submitted): Construct rest of the items and the parse table.

SLR parse table for the grammar

	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6,r6			r6			
3				r3			
4		s4	s5			8	7
5	r5			r5			
6		s4	s5			8	9
7	r4			r4			
8	r6			r6			
9				r2			

The table has multiple entries in action[2,=]

- There is both a shift and a reduce entry in action[2,=]. Therefore state 2 has a shift-reduce conflict on symbol "=", However, the grammar is not ambiguous.

- Parse id=id assuming reduce action is taken in [2,=]

Stack	**input**	**action**
0	id=id	shift 5
0 id 5	=id	reduce by L→id
0 L 2	=id	reduce by R→L
0 R 3	=id	error

- if shift action is taken in [2,=]

Stack	**input**	**action**
0	id=id$	shift 5
0 id 5	=id$	reduce by L→id
0 L 2	=id$	shift 6
0 L 2 = 6	id$	shift 5
0 L 2 = 6 id 5	$	reduce by L→id
0 L 2 = 6 L 8	$	reduce by R→L
0 L 2 = 6 R 9	$	reduce by S→L=R
0 S 1	$	ACCEPT

Problems in SLR parsing

- No sentential form of this grammar can start with R=…
- However, the reduce action in action[2,=] generates a sentential form starting with R=
- Therefore, the reduce action is incorrect
- In SLR parsing method state i calls for reduction on symbol "a", by rule $A \rightarrow \alpha$ if I_i contains $[A \rightarrow \alpha.]$ and "a" is in follow(A)
- However, when state I appears on the top of the stack, the viable prefix $\beta\alpha$ on the stack may be such that βA can not be followed by symbol "a" in any right sentential form
- Thus, the reduction by the rule $A \rightarrow \alpha$ on symbol "a" is invalid
- SLR parsers cannot remember the left context

What is the problem?

S′ → S
S → L = R
S → R
L → *R
L → id
R → L

I_0: S′ → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

I_1: goto(I_0, S)
S′ → S.

I_2: goto(I_0, L)
S → L.=R
R → L.

Solution ?

Canonical LR Parsing

- Carry extra information in the state so that wrong reductions by $A \rightarrow \alpha$ will be ruled out
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol)
- The general form of the item becomes $[A \rightarrow \alpha.\beta, a]$ which is called LR(1) item.
- Item $[A \rightarrow \alpha., a]$ calls for reduction only if next input is a. The set of symbols "a"s will be a subset of Follow(A).

Closure(I)

repeat
  for each item [A → α.Bβ, a] in I
    for each production B → γ in G'
    and for each terminal b in First(βa)
      add item [B → .γ, b] to I
until no more additions to I

Example

Consider the following grammar

S'→ S
S → CC
C → cC | d

Compute closure(I) where I={[S' → .S, $]}

S'→ .S,	$
S → .CC,	$
C → .cC,	c
C → .cC,	d
C → .d,	c
C → .d,	d

Example

Construct sets of LR(1) items for the grammar on previous slide

I_0: $S' \rightarrow .S$, $
$S \rightarrow .CC$, $
$C \rightarrow .cC$, c/d
$C \rightarrow .d$, c/d

I_1: goto(I_0,S)
$S' \rightarrow S.$, $

I_2: goto(I_0,C)
$S \rightarrow C.C$, $
$C \rightarrow .cC$, $
$C \rightarrow .d$, $

I_3: goto(I_0,c)
$C \rightarrow c.C$, c/d
$C \rightarrow .cC$, c/d
$C \rightarrow .d$, c/d

I_4: goto(I_0,d)
$C \rightarrow d.$, c/d

I_5: goto(I_2,C)
$S \rightarrow CC.$, $

I_6: goto(I_2,c)
$C \rightarrow c.C$, $
$C \rightarrow .cC$, $
$C \rightarrow .d$, $

I_7: goto(I_2,d)
$C \rightarrow d.$, $

I_8: goto(I_3,C)
$C \rightarrow cC.$, c/d

I_9: goto(I_6,C)
$C \rightarrow cC.$, $

Construction of Canonical LR parse table

- Construct $C=\{I_0, \ldots, I_n\}$ the sets of LR(1) items.
- If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and $goto(I_i, a)=I_j$ then action[i,a]=shift j
- If $[A \rightarrow \alpha., a]$ is in I_i then action[i,a] reduce $A \rightarrow \alpha$
- If $[S' \rightarrow S., \$]$ is in I_i then action[i,$] = accept
- If $goto(I_i, A) = I_j$ then goto[i,A] = j for all non terminals A

Parse table

State	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Notes on Canonical LR Parser

- Consider the grammar discussed in the previous two slides. The language specified by the grammar is c*dc*d.
- When reading input cc…dcc…d the parser shifts cs into stack and then goes into state 4 after reading d. It then calls for reduction by C→d if following symbol is c or d.
- IF $ follows the first d then input string is c*d which is not in the language; parser declares an error
- On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error
- Problem: Canonical LR parse table has a large number of states

LALR Parse table

- Look Ahead LR parsers
- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items

  I_4: $C \rightarrow d.$, c/d I_7: $C \rightarrow d.$, \$
- Replace I_4 and I_7 by a new state I_{47} consisting of ($C \rightarrow d.$, c/d/\$)
- Similarly I_3 & I_6 and I_8 & I_9 form pairs
- Merge LR(1) items having the same core

Construct LALR parse table

- Construct $C=\{I_0,\ldots\ldots,I_n\}$ set of LR(1) items
- For each core present in LR(1) items find all sets having the same core and replace these sets by their union
- Let $C' = \{J_0,\ldots\ldots,J_m\}$ be the resulting set of items
- Construct action table as was done earlier
- Let $J = I_1 \cup I_2\ldots\ldots \cup I_k$

  since I_1 , $I_2\ldots\ldots$, I_k have same core, goto(J,X) will have he same core

  Let K=goto(I_1,X) U goto(I_2,X)……goto(I_k,X) the goto(J,X)=K

Example

Construct sets of LR(1) items for the grammar on previous slide

I_0: $S' \rightarrow .S$, $
$S \rightarrow .CC$, $
$C \rightarrow .cC$, c/d
$C \rightarrow .d$, c/d

I_1: goto(I_0,S)
$S' \rightarrow S.$, $

I_2: goto(I_0,C)
$S \rightarrow C.C$, $
$C \rightarrow .cC$, $
$C \rightarrow .d$, $

I_3: goto(I_0,c)
$C \rightarrow c.C$, c/d
$C \rightarrow .cC$, c/d
$C \rightarrow .d$, c/d

I_4: goto(I_0,d)
$C \rightarrow d.$, c/d

I_5: goto(I_2,C)
$S \rightarrow CC.$, $

I_6: goto(I_2,c)
$C \rightarrow c.C$, $
$C \rightarrow .cC$, $
$C \rightarrow .d$, $

I_7: goto(I_2,d)
$C \rightarrow d.$, $

I_8: goto(I_3,C)
$C \rightarrow cC.$, c/d

I_9: goto(I_6,C)
$C \rightarrow cC.$, $

LALR parse table …

State	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Notes on LALR parse table

- Modified parser behaves as original except that it will reduce C→d on inputs like ccd. The error will eventually be caught before any more symbols are shifted.
- In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.
- Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.
- SLR and LALR parse tables have same number of states.

Notes on LALR parse table…

- Merging items may result into conflicts in LALR parsers which did not exist in LR parsers

- New conflicts can not be of shift reduce kind:
  - Assume there is a shift reduce conflict in some state of LALR parser with items
    {[X→α.,a],[Y→γ.aβ,b]}
  - Then there must have been a state in the LR parser with the same core
  - Contradiction; because LR parser did not have conflicts

- LALR parser can have new reduce-reduce conflicts
  - Assume states
    {[X→α., a], [Y→α., b]} and {[X→α., b], [Y→α., a]}
  - Merging the two states produces
    {[X→α., a/b], [Y→α., a/b]}

Notes on LALR parse table…

- LALR parsers are not built by first making canonical LR parse tables
- There are direct, complicated but efficient algorithms to develop LALR parsers
- Relative power of various classes
  - SLR(1) ≤ LALR(1) ≤ LR(1)
  - SLR(k) ≤ LALR(k) ≤ LR(k)
  - LL(k) ≤ LR(k)

Error Recovery

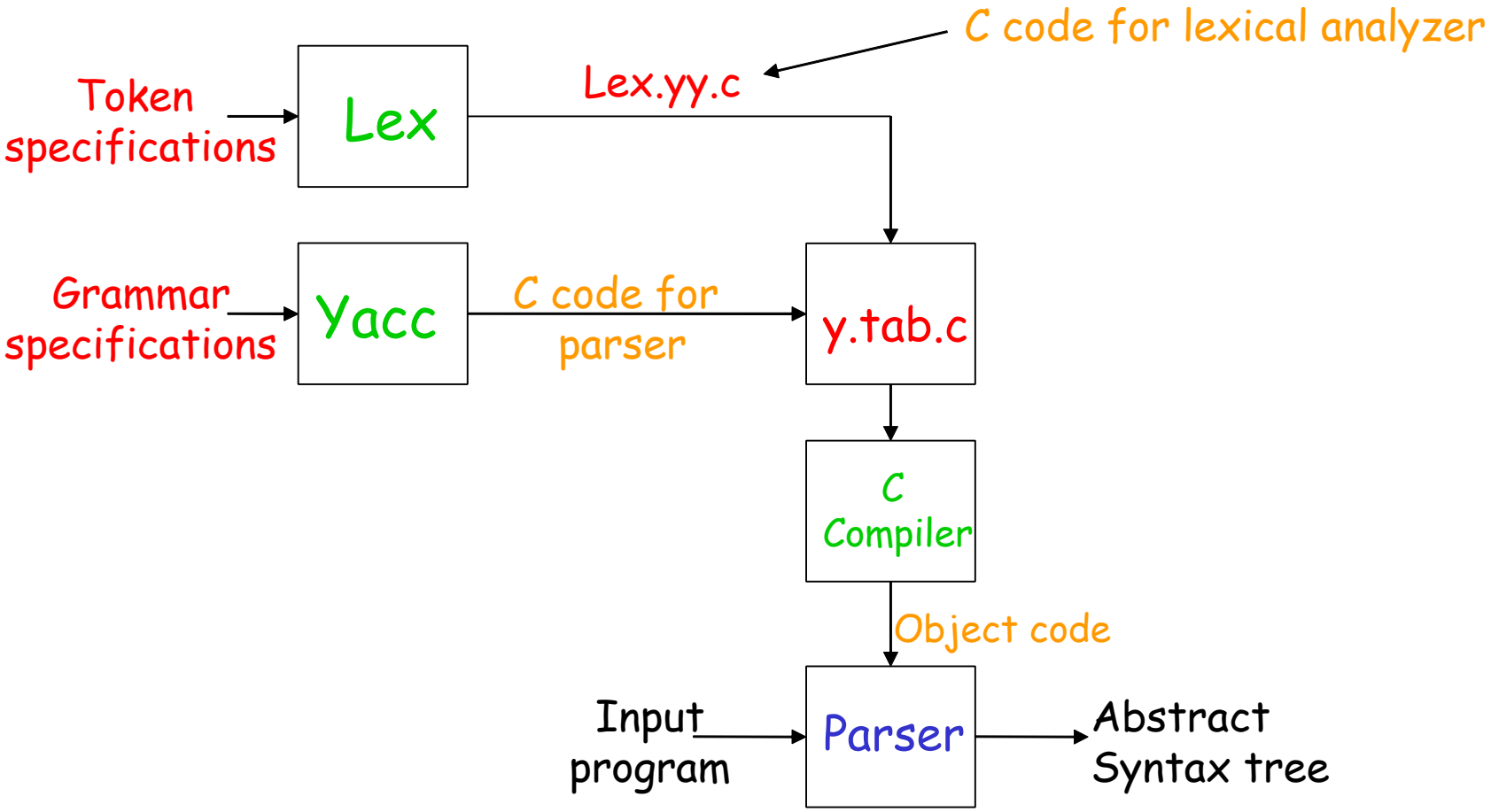
- An error is detected when an entry in the action table is found to be empty.
- Panic mode error recovery can be implemented as follows:
  - scan down the stack until a state S with a goto on a particular nonterminal *A* is found.
  - discard zero or more input symbols until a symbol a is found that can legitimately follow *A*.
  - stack the state goto[S,*A*] and resume parsing.
- **Choice of *A*:** Normally these are non terminals representing major program pieces such as an expression, statement or a block. For example if *A* is the nonterminal stmt, a might be semicolon or end.

Parser Generator

- Some common LR parser generators
  - YACC: **Y**et **A**nother **C**ompiler **C**ompiler
  - Bison: GNU Software
- Yacc/Bison source program specification (accept LALR grammars)

  declaration
  %%
  translation rules
  %%
  supporting C routines

Yacc and Lex schema



Refer to YACC Manual

Assignment

- Reading assignment: Read about error recovery in LL parsers

- Construct LL(1) parse table for the expression grammar

  bexpr → bexpr or bterm | bterm
  bterm → bterm and bfactor | bfactor
  bfactor → not bfactor | (bexpr) | true | false

- Steps to be followed
  - Remove left recursion
  - Compute first sets
  - Compute follow sets
  - Construct the parse table

- Introduce synch symbols (using both follow and first sets) in the parse table created for the boolean expression grammar in the previous assignment
- Parse "not (true and or false)" and show how error recovery works

Assignment

Construct SLR parse table for following grammar

E → E + E | E - E | E * E | E / E | (E) | digit

Show steps in parsing of string
9*5+(2+3*7)

- Steps to be followed
  - Augment the grammar
  - Construct set of LR(0) items
  - Construct the parse table
  - Show states of parser as the given string is parsed

- Due on February 13, 2012

Reading Assignment Up to Jan 31, 2012

- Chapter 1 and 2 of ALSU Book
- Chapter 3 (may skip 3.6-3.9) of ALSU Book
- Chapter 4 of ALSU Book

What syntax analysis cannot do!

- To check whether variables are of types on which operations are allowed
- To check whether a variable has been declared before use
- To check whether a variable has been initialized
- These issues will be handled in semantic analysis