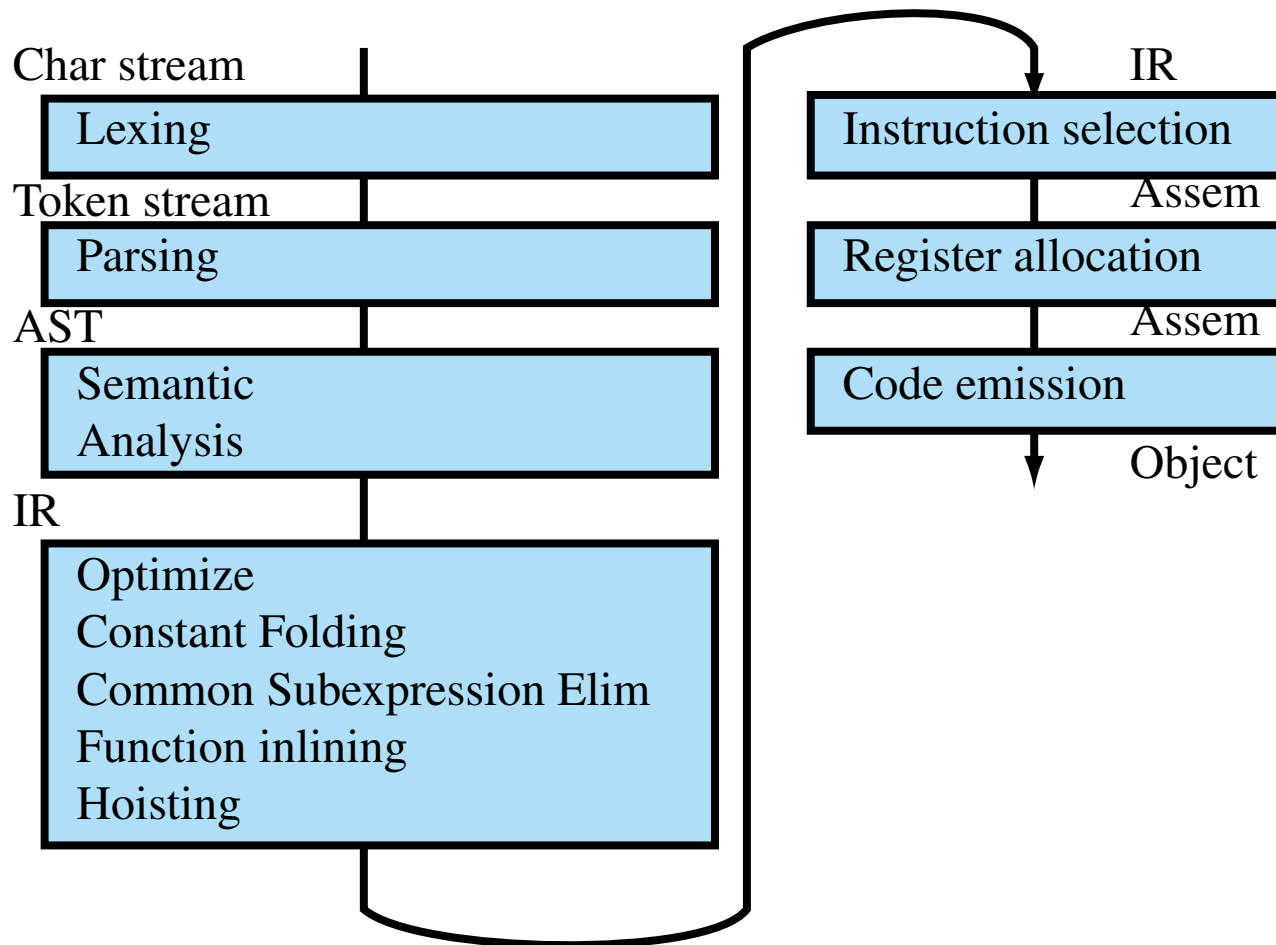


CS134b: Lexing and Parsing

- Compiler stages
- Lexing
- Parsing
- Intermediate Representation



Typical stages in a compiler



Lexing

- The process of breaking the input into a sequence of *tokens*
- Tokens will be called *terminals* in the parser
- Tokens are usually specified with *regular expressions*



Example tokens

- TokId of string: *v*, *main*, *_f*, *_175*
- TokInt of int: 1, 7, 57491, 0xab73 (usually minus signs are not included)
- TokFloat of float: 1.2, 7.67e-23, 3.1415926
- Keywords:
 - *TokLeftParen*: (
 - *TokRightCurly*: }
 - *TokIf*: *if*
 - *TokEq*: =
 - *TokEqEq*: ==



Regular expressions

- RE are defined over an *alphabet* of atomic symbols (we'll use ASCII)
- RE are defined by induction:
 - *epsilon is a RE (it stands for the empty string)*
 - *if 'c' is a letter in the alphabet, it is a RE (it stands for c)*
 - *if e is a RE, so is*
 - *(e) : (grouping)*
 - *e* : (Kleene closure: 0 or more occurrence of e)*
 - *if e1 and e2 are RE, so are:*
 - *e1 | e2 : (alternation: either e1 or e2)*
 - *e1 e2 : (concatentation: e1 followed by e2)*



Examples of regular expressions

- $e? == (e | e)$: 0 or 1 occurrence of e
- $e+ == e e^*$: 1 or more occurrence of e
- $[c1 - c2]$: any character c in the range $c1$ to $c2$
- $[^c]$: any character except c
- `"s"` : matches string s literally
- integer: $['0' - '9']^+$
- string: ``" ' [^ " '] * " '`
- identifier: $['a' - 'z' 'A' - 'Z' '_'] ['a' - 'z' 'A' - 'Z' '0' - '9' '_']^*$
- comment: `"//" [^ '\n'] * '\n'`



Languages that are not regular

- string of balanced parenthesis
 - $s = e \mid '(s)'$ $\mid ss$
 - *balanced parenthesis with finite nesting are regular*
- Arithmetic expressions
 - $s = [0-9]^+ \mid s ['+' '-' '*' '/'] s \mid '(s)'$



Options for building lexical analyzers

- Use *ocamllex*

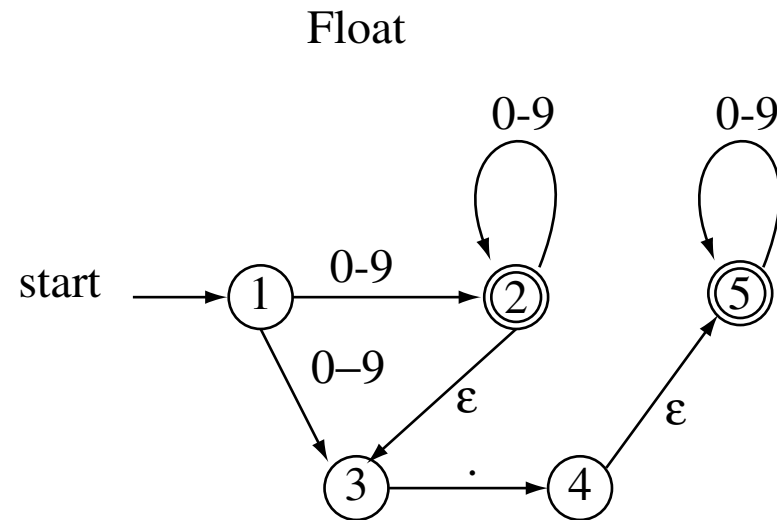
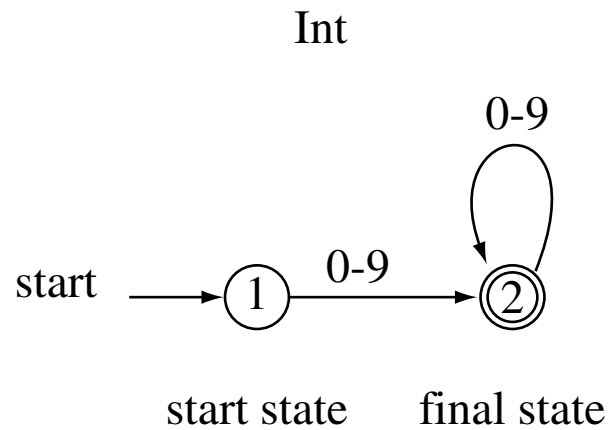


Another option

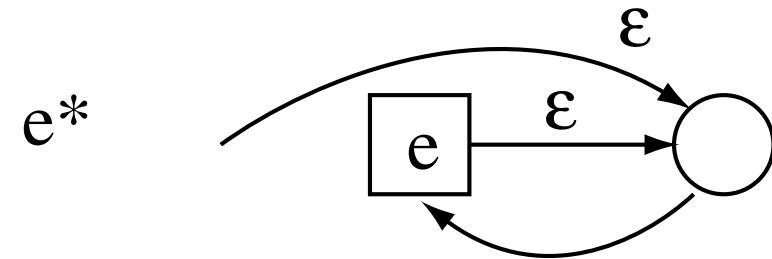
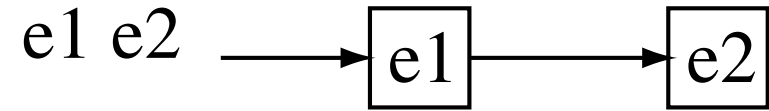
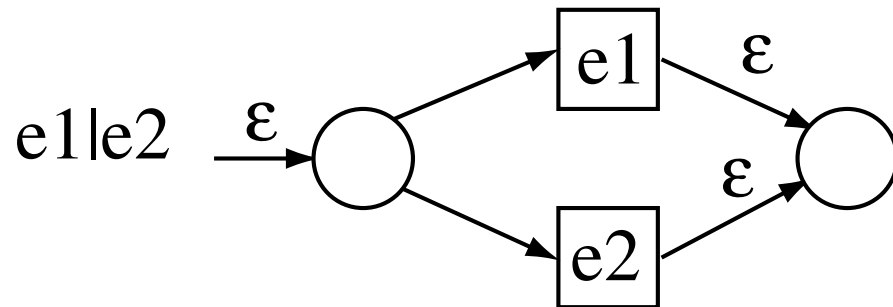
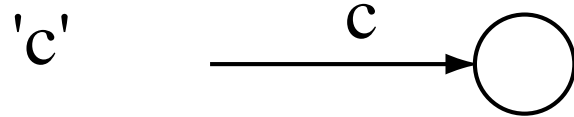
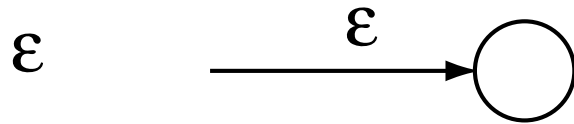
- Compile the RE to *finite automata*
- A *nondeterministic finite automaton* is a 5-tuple:
 - type Σ : *the input alphabet*
 - type S : *a set of states*
 - $\delta : S \rightarrow \Sigma \rightarrow 2^S$ *a transition function*
 - Start : S *an initial or start state*
 - Final : 2^S *a set of final states*



Example automata



Converting RE to NFA



Converting NFA to DFA

```
open Set_sig
```

```
module type DFASig =
```

```
sig
```

```
  type sigma (* alphabet *)
```

```
  type state (* states of the machine *)
```

```
  (* start state *)
```

```
  val start : state
```

```
  (* transition function *)
```

```
  val delta : state -> sigma -> state
```

```
  (* final states *)
```

```
  val final : state -> bool
```

```
end
```



NFA Signature

```
module type NFASig =
```

```
sig
```

```
  type sigma
```

```
  type state
```

```
  module Set : SetSig with type elt = state
```

```
  val start : state
```

```
  val delta : state -> sigma -> Set.t
```

```
  val epsilon : state -> Set.t
```

```
  val final : state -> bool
```

```
end
```



Conversion mechanics

- We build a DFA with states that are *sets* of states of the NFA
- The initial state is the ε -closure of the NFA initial state
- The final states are those that contain a final state of the NFA
- The transition function is the *union* of the NFA transition function, followed by the ε -closure



DFAofNFA functor

```
module DFAOfNFA (NFA : NFASig) =  
struct  
  type sigma = NFA.sigma  
  
  (* States are sets of NFA states *)  
  type state = NFA.Set.t  
  
  (* Start state is closure of initial state *)  
  let start = NFA.epsilon NFA.start  
  
  (* Final state *)  
  let final s =  
    NFA.Set.fold (fun flag s' -> flag || NFA.final s') false s
```



Transition function

(Transition function *)*

let delta s c =

(Get a list of states defined by the transition function *)*

let states = NFA.Set.to_list s in

let states = List.map (**fun** s -> NFA.delta s c) states in

(Gather the states in a single set *)*

let states = List.fold_left (**fun** states s -> NFA.Set.union states s) NFA.Set.empty states in

(Add epsilon transitions *)*

let states = NFA.Set.to_list states in

let states = List.map NFA.epsilon states in

List.fold_left (**fun** states s -> NFA.Set.union states s) NFA.Set.empty states



Parsing

- Regular expressions are not expressive enough to interpret a programming language
 - *For example, languages allowing arbitrary nesting of balanced parens are not regular*
- However, most programming languages are *context-free*
- A *grammar* is a set of *productions*
- A *production* has the form:
 - *non-terminal -> symbols*
- A *symbol* is either a *terminal* (a word in the alphabet), or a *non-terminal* (defined by a production)



Context-free grammars

- The *start-symbol* defines a language
- A *derivation* is the systematic replacement of non-terminals by their definitions
 - *The result contains no non-terminals*
- Example (e is a non-terminal symbol; number, +, -, *, / are terminal symbols)
 - $e \rightarrow \text{number}$
 - $e \rightarrow e + e$
 - $e \rightarrow e - e$
 - $e \rightarrow e * e$
 - $e \rightarrow e / e$

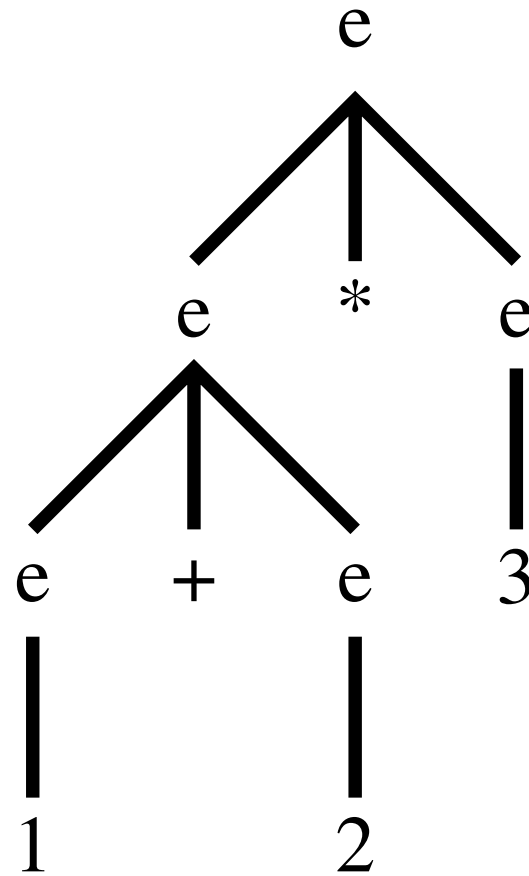
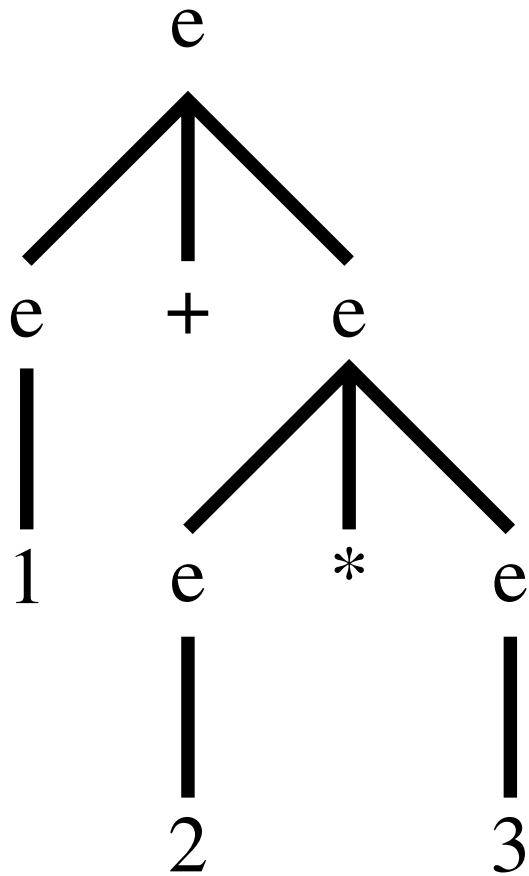


Context-free grammars

- A grammar is *ambiguous* if there are multiple derivations of the same sentence
- A *derivation tree* is a tree where each node is labeled by a symbol, and the children of a non-terminal node form a valid production.
- The AST is often a derivation tree--ambiguity is bad



Ambiguity



Parsing

- Context-free languages are recognized by *pushdown automata* (a finite automaton with a stack)
- How do we construct it?
- A simple non-ambiguous language
- Recursive-descent parsing

$$\begin{aligned} e ::= & \mathbf{i} \\ & | \mathbf{v} \\ & | (e) \\ & | \mathbf{binop} \ e \ e \\ & | \mathbf{let} \ v = e \ \mathbf{in} \ e \\ & | \mathbf{fun} \ v \ \mathbf{->} \ e \\ & | \mathbf{apply} \ e \ e \end{aligned}$$


LL(1) Parsing

- Left-to-right parsing, leftmost derivation, 1 token lookahead
- Not very expressive, what about this grammar:
 - $e ::= \textit{number}$
 $| (e)$
 $| e + e$
 $| e * e$
- Ambiguous
- Worse, the productions can't be identified by their first token



LR(k) Parsing

- Left-to-right parsing, rightmost derivation, k tokens lookahead
- Stack machine
- Example:
 - $S \rightarrow E \$$
 - $E \rightarrow number$
 - $E \rightarrow E + E$
 - $E \rightarrow E * E$



Parsing $1 + 2 * 3$

stack	lookahead	action
	1	shift
1	+	reduce $E \rightarrow \text{number}$
E(1)	+	shift
E(1) +	2	shift
E(1) + 2	*	reduce $E \rightarrow \text{number}$
E(1) + E(2)	*	shift/reduce $E \rightarrow E + E$
E(1) + E(2) *	3	shift
E(1) + E(2) * 3	\$	reduce $E \rightarrow \text{number}$
E(1) + E(2) * E(3)	\$	reduce $E \rightarrow E * E$
E(1) + E(2 * 3)	\$	reduce $E \rightarrow E + E$
E(1 + (2 * 3))	\$	accept



LR(0)

- How do we build the parser table?
- New grammar:
 - $S \rightarrow E \$$
 - $E \rightarrow \textit{number}$
 - $E \rightarrow (L)$
 - $L \rightarrow E$
 - $L \rightarrow L + E$

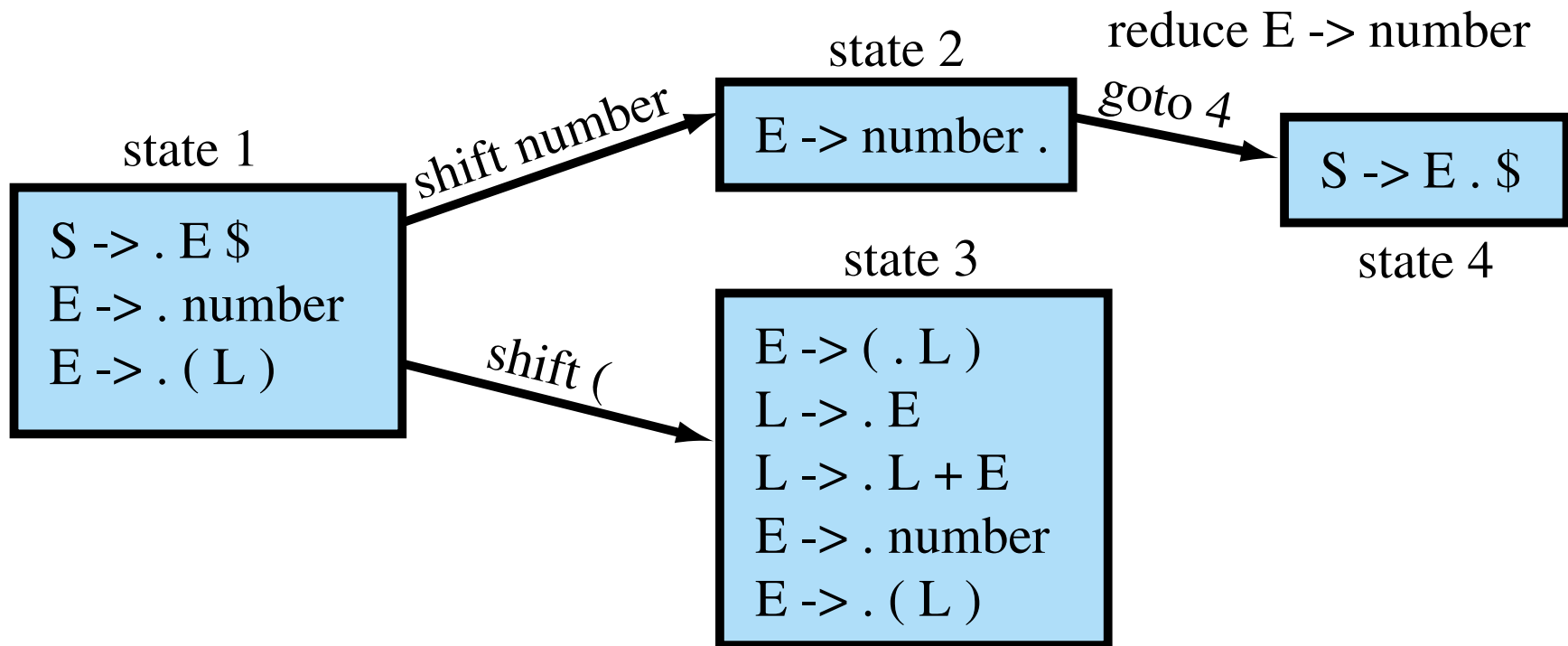


LR(0) parsing

- Build a DFA
 - States are sets of productions + position info
 - state 1:
 - $S \rightarrow \cdot S \$$
 - $E \rightarrow \cdot \text{number}$
 - $E \rightarrow \cdot (L)$
- Actions:
 - shift t : add token t to the stack
 - reduce i : apply the i 'th production to the stack
 - goto i : goto state i



LR(0) DFA



Operations

- closure : state \rightarrow state
- goto : state \rightarrow symbol \rightarrow state

let closure $I =$

repeat until I does not change:

for each $A \rightarrow \alpha.X\beta \in I$

for each production $X \rightarrow \gamma$

$I \leftarrow I \cup \{X \rightarrow \cdot\gamma\}$



Goto

- goto $I X$ moves the $.$ past the X symbol

let goto $I X =$

let $I' = \{\}$ **in**

for each $A \rightarrow \alpha.X\beta \in I$ **do**

$I' \leftarrow I' \cup \{A \rightarrow \alpha X.\beta\};$

return (closure I')



Building the table

let $T = \text{closure}(\{S' \rightarrow .S\})$ **in**

let $E = \{\}$ **in**

repeat until T and E do not change

for each $I \in T$

for each $A \rightarrow \alpha.X\beta \in I$

let $J = \text{goto}(I, X)$ **in**

$T \leftarrow T \cup \{J\}$

$E \leftarrow E \cup \{I \rightarrow^X J\}$



Building the table

- If E contains $I \rightarrow X J$
 - if X is a terminal, add “shift X ” to state I
 - if X is a non-terminal, add “goto J ” to state I
- If I contains “ $A \rightarrow \alpha \cdot$ ” add “reduce $A \rightarrow \alpha$ ” to state I



LR(1) parsing

- An LR(1) item contains
 - *a production*
 - *the position (represented by a dot)*
 - *a lookahead token*

$$A \rightarrow \alpha \cdot X \beta, z$$



FIRST function

- Returns the set of terminals that may begin a production

$FIRST(X) = \{X\}$ if X is a terminal

$FIRST(Y_1 Y_2 \dots Y_n) = \begin{cases} FIRST(Y_1) \cup FIRST(Y_2 \dots Y_n) & \text{if } Y_1 \text{ is nullable} \\ FIRST(Y_1) & \text{otherwise} \end{cases}$



Closure function

- Modified to get the next lookahead symbol

let closure $I =$

repeat until I does not change:

for each $(A \rightarrow \alpha.X\beta, z) \in I$ **do**

for each production $X \rightarrow \gamma$

for each $w \in FIRST(\beta z)$

$I \leftarrow I \cup \{(X \rightarrow \cdot\gamma, w)\}$

return I



Goto function

- Modified for LR(1) items

```
let Goto  $I X =$   
  let  $J = \{\}$   
  for each  $(A \rightarrow \alpha.X\beta, z) \in I$  do  
     $J \leftarrow J \cup \{(A \rightarrow \alpha X.\beta, z)\}$   
  return (closure  $J$ )
```

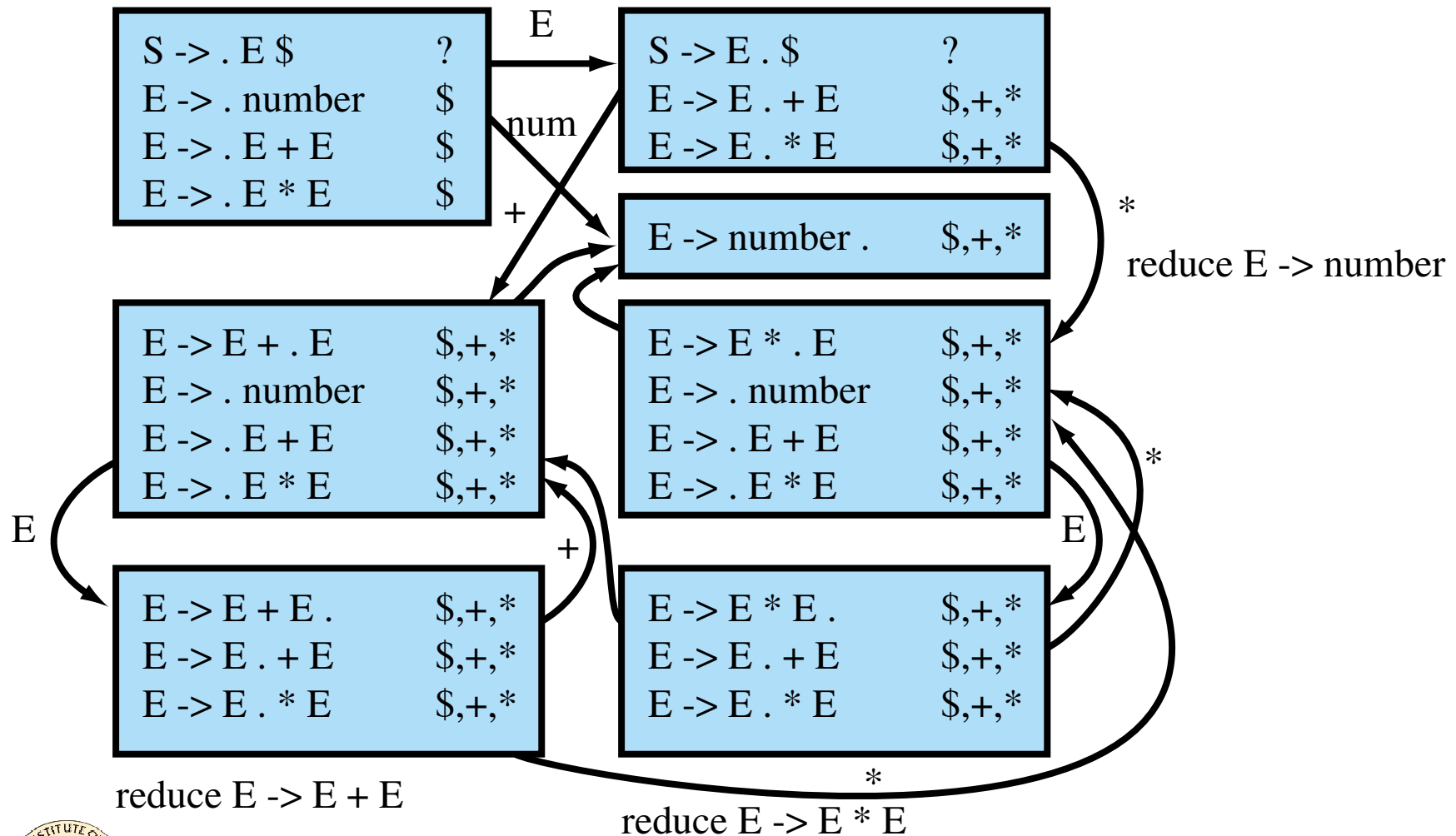


An example

- $E \rightarrow \text{number}$
- $E \rightarrow E + E$
- $E \rightarrow E * E$



LR(1) state diagram



Disambiguating expressions

- $E \rightarrow \text{number} \mid E + E \mid E * E \mid (E)$
- Inherently ambiguous
- Rewrite into a two-level grammar:
 - *E: general expression*
 - $E \rightarrow T \mid E + T$
 - *T: pure multiplication*
 - $T \rightarrow S \mid T * S$
 - *S: “simple” expression*
 - $S \rightarrow \text{number} \mid (E)$



How can we disambiguate ifthenelse?

- $E \rightarrow \text{number}$
- $E \rightarrow E + E \mid E * E \mid (E)$
- $E \rightarrow \text{if } E \text{ then } E$
- $E \rightarrow \text{if } E \text{ then } E \text{ else } E$



Matched expression

- $E \rightarrow M \mid U$
- $M \rightarrow \text{number} \mid M + M \mid M * M \mid (E)$
- $M \rightarrow \text{if } E \text{ then } M \text{ else } M$
- $U \rightarrow \text{if } E \text{ then } U$
- $U \rightarrow \text{if } E \text{ then } M \text{ else } U$

