# Topic 3: Parsing and Yaccing

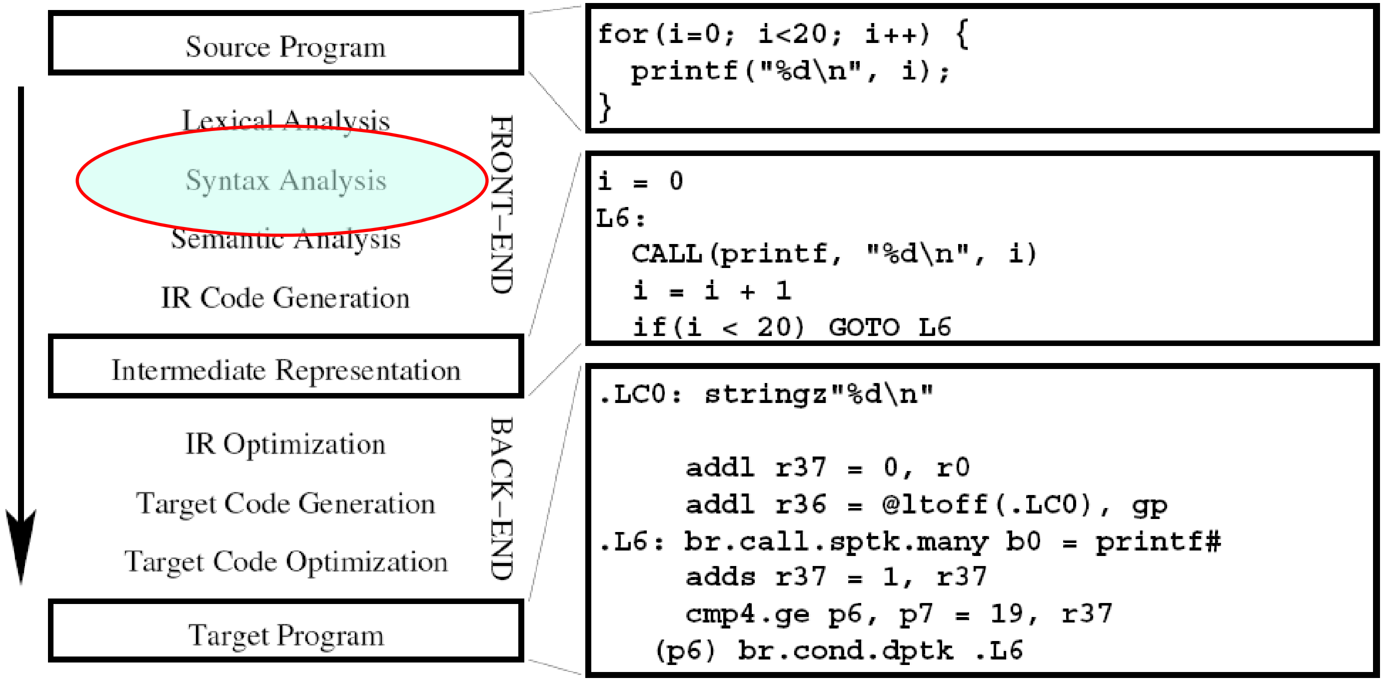
COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

# The Compiler




- Lexical Analysis: Break into tokens (think words, punctuation)
- Syntax Analysis: Parse phrase structure (think document, paragraphs, sentences)
- Semantic Analysis: Calculate meaning

# Role of parser

Lexer partitioned document into stream of tokens.

But not all token lists represent programs.

| >= | THEN | ID(foo) | IF | NUM(42) | • • • |
|---|---|---|---|---|---|
| IF | ID(foo) | >= | NUM(42) | THEN | • • • |

- verify that stream of tokens is valid according to the language definition
- report violations as (informative) syntax error and recover
- build abstract syntax tree (AST) for use in next compiler phase

# Syntactical Analysis

- each language definition has rules that describe the syntax of well-formed programs.
- format of the rules: **context-free grammars**
- why not regular expressions/NFA's/DFA's?

# Syntactical Analysis

- each language definition has rules that describe the syntax of well-formed programs.
- format of the rules: **context-free grammars**
- why not regular expressions/NFA's/DFA's?
- source program constructs have recursive structure:

digits = [0-9]+;

expr = digits | "(" expr "+" expr ")"

# Syntactical Analysis

- each language definition has rules that describe the syntax of well-formed programs.
- format of the rules: **context-free grammars**
- why not regular expressions/NFA's/DFA's?
- source program constructs have recursive structure:

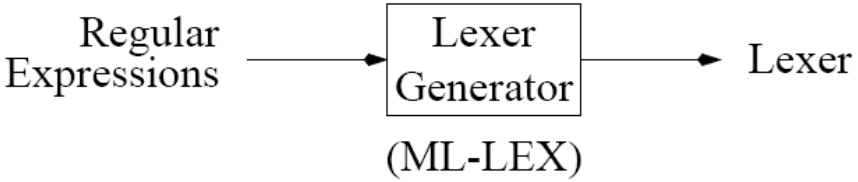
digits = [0-9]+;

expr = digits | "(" expr "+" expr ")"

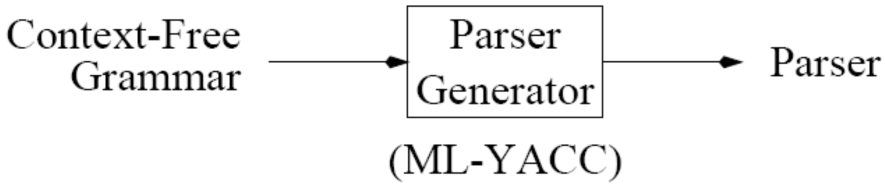
- finite automata can't recognize recursive constructs, so cannot ensure expressions are well-bracketed: a machine with N states cannot remember parenthesis—nesting depth greater than N
- CFG's are more powerful, but also more costly to implement

# Context-Free Grammar

Regular Expressions - describe lexical structure of tokens.




Context-Free Grammars - describe syntactic nature of programs.



# Context-Free Grammars: definitions

- **language**: set of **strings**
- **string:** finite sequence of **symbols** taken from finite **alphabet**

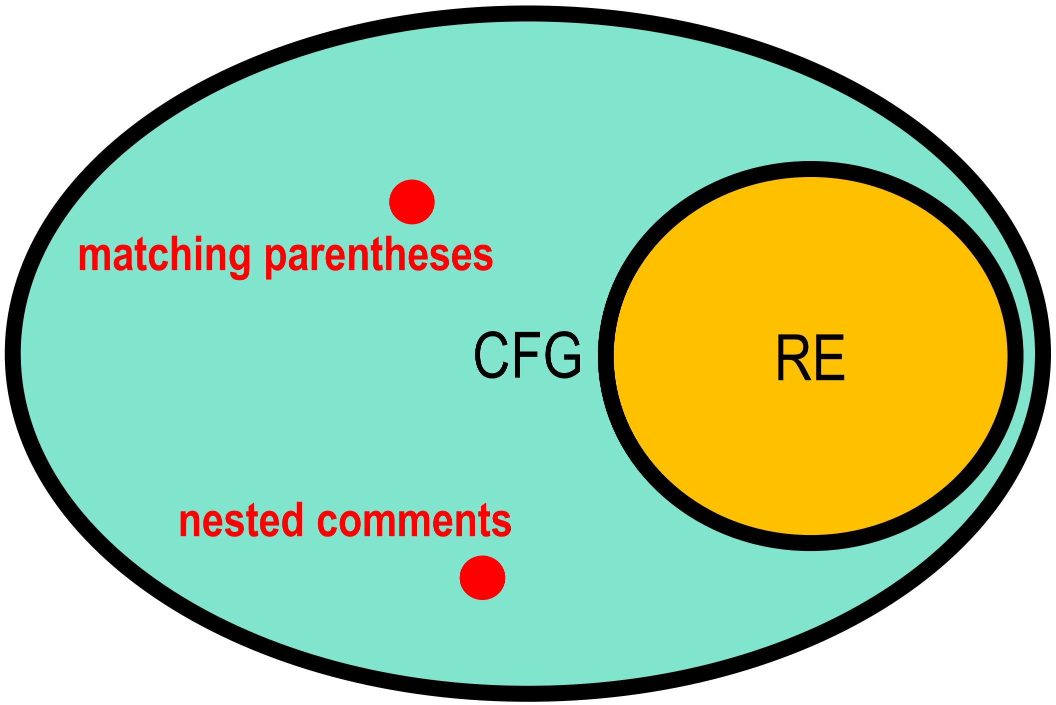
Regular expressions and CFG’s both describe languages, but over different alphabets

| | Lexical analysis | Syntax analysis |
|---|---|---|
| symbols/alphabet | ASCII | token |
| strings | lists of tokens | lists of phrases |
| language | set of (legal)<br>token sequences | set of (legal)<br>phrase sequences<br>(“programs”) |

# Context-free grammars versus regular expressions

CFG's strictly more expressive than RE's:

Any language recognizable/generated by a RE can also be recognized/generated by a CFG, **but not vice versa**.




Also known as Backus-Naur Form (BNF, Algol 60)

# Context-Free Grammars: definition

A CFG consists of

- a finite set $\boldsymbol{N} = N_1$ , .., $N_k$ of **non-terminal** symbols, one of which is singled out as the **start symbol**
- a finite set $\boldsymbol{T} = T_1$, …, $T_m$ of **terminal** symbols (representing token types)
- a finite set of **productions** **LHS** $\longrightarrow$ **RHS** where **LHS** is a **single** non-terminal, and **RHS** is a **list** of terminals and non-terminals

Each production specifies one way in which terminals and non-terminals may be combined to form a legal (sub)string.

Recursion (e.g. well-bracketing) can be modeled by referring to the LHS inside the RHS: **stmt** $\longrightarrow$ **IF exp THEN stmt ELSE stmt**

The language **recognized** by the CFG is the set of **terminal**-only strings derivable from the **start symbol** .

# CFG's: derivations

1. Start with start symbol
2. While the current string contains a non-terminal **N**, replace it by the RHS of one of the rules for **N**.

**Non-determinism**

- Choice of non-terminal **N** to be replaced in each step
- Choice of production/**RHS** once **N** has been chosen.

Acceptance of a string is independent of these choices (cf. NFA), so a string may have multiple derivations.

**Notable derivation strategies**

- left-most derivation: in each step, replace the left-most non-terminal
- right-most derivation: …

(In both cases, use the chosen symbol's first rule)

# Example (cf HW1)

**Nonterminals**

stmt /*statements/*
expr /*expressions*/
expr_list /*expression lists*/

**Terminals : tokens**

SEMI
ID
ASSIGN
LPAREN
RPAREN
NUM
PLUS
PRINT
COMMA

**Productions**

i.e. stmt SEMI stmt

stmt ➡ stmt; stmt
stmt ➡ ID := expr
stmt ➡ print ( expr_list )

i.e. PRINT LPAREN expr_list RPAREN

expr ➡ ID
expr ➡ NUM
expr ➡ expr + expr
expr ➡ ( stmt, expr )

expr_list ➡ expr
expr_list ➡ expr_list, expr

# Example: leftmost Derivation for a:=12; print(23)

Show that the following token sequence
ID := NUM; PRINT(NUM) is a legal phrase

ID ASSIGN NUM SEMI PRINT LPAREN NUM RPAREN

**Productions**

stmt → stmt; stmt
stmt → ID := expr
stmt → print ( expr_list )

expr → ID
expr → NUM
expr → expr + expr
expr → ( stmt, expr )

expr_list → expr
expr_list → expr_list, expr

# Example: leftmost Derivation for a:=12; print(23)

**Productions**

| | | |
|---|---|---|
| stmt | → | stmt; stmt |
| stmt | → | ID := expr |
| stmt | → | print ( expr_list ) |
| expr | → | ID |
| expr | → | NUM |
| expr | → | expr + expr |
| expr | → | ( stmt, expr ) |
| expr_list | → | expr |
| expr_list | → | expr_list, expr |

Show that the following token sequence
ID := NUM; PRINT(NUM) is a legal phrase

ID ASSIGN NUM SEMI PRINT LPAREN NUM RPAREN

**stmt**

**stmt** SEMI stmt

ID ASSIGN **expr** SEMI stmt

ID ASSIGN NUM SEMI **stmt**

ID ASSIGN NUM SEMI PRINT LPAREN **expr_list** RPAREN

ID ASSIGN NUM SEMI PRINT LPAREN **expr** RPAREN

ID ASSIGN NUM SEMI PRINT LPAREN NUM RPAREN

# Example: rightmost Derivation for a:=12; print(23)

**Productions**

stmt → stmt; stmt
stmt → ID := expr
stmt → print ( expr_list )

expr → ID
expr → NUM
expr → expr + expr
expr → ( stmt, expr )

expr_list → expr
expr_list → expr_list, expr

Show that the following token sequence
ID := NUM; PRINT(NUM) is a legal phrase
ID ASSIGN NUM SEMI PRINT LPAREN NUM RPAREN

**stmt**

stmt SEMI **stmt**

stmt SEMI PRINT LPAREN **expr_list** RPAREN

stmt SEMI PRINT LPAREN **expr** RPAREN

**stmt** SEMI PRINT LPAREN NUM RPAREN

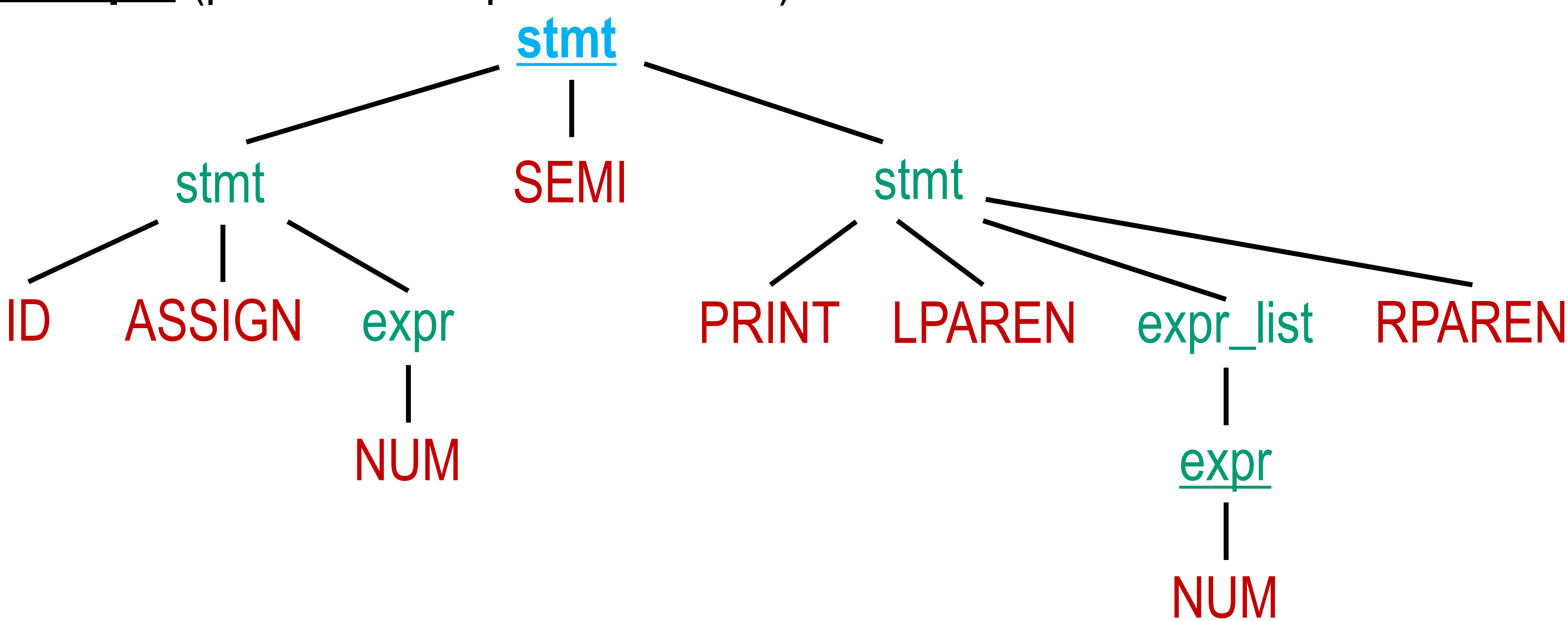
ID ASSIGN **expr** SEMI PRINT LPAREN NUM RPAREN

ID ASSIGN NUM SEMI PRINT LPAREN NUM RPAREN

# Parse tree: graphical representation of derivation results

- **Root**: **start symbol**
- **Inner nodes** labeled with **non-terminals**; #branches according to chosen production
- **Leaf nodes** labeled with **terminals**

**Example** (parse tree of previous CFG):




Different derivations may or may not yield different parse trees.

# Ambiguous Grammars

A grammar is **ambiguous** is it can derive a string of tokens with two or more different parse trees

**Example?**

# Ambiguous Grammars

A grammar is **ambiguous** is it can derive a string of tokens with two or more different parse trees

## Example?

Non-Terminals:

```
expr : Expression
```

Terminals (tokens):

```
ID
NUM
PLUS   "+"
MULT   "*"
```

$$
\begin{aligned}
expr &\rightarrow ID \\
expr &\rightarrow NUM \\
expr &\rightarrow expr + expr \\
expr &\rightarrow expr * expr
\end{aligned}
$$

# Ambiguous Grammars

A grammar is **ambiguous** is it can derive a string of tokens with two or more different parse trees

## Example?

Non-Terminals:

```
expr : Expression
```

Terminals (tokens):

```
ID
NUM
PLUS   "+"
MULT   "*"
```

$$
\begin{aligned}
expr &\rightarrow ID \\
expr &\rightarrow NUM \\
expr &\rightarrow expr + expr \\
expr &\rightarrow expr * expr
\end{aligned}
$$

Consider: $4 + 5 * 6$

# Ambiguous Grammars

A grammar is **ambiguous** is it can derive a string of tokens with two or more different parse trees

## Example?

Non-Terminals:

```
expr : Expression
```

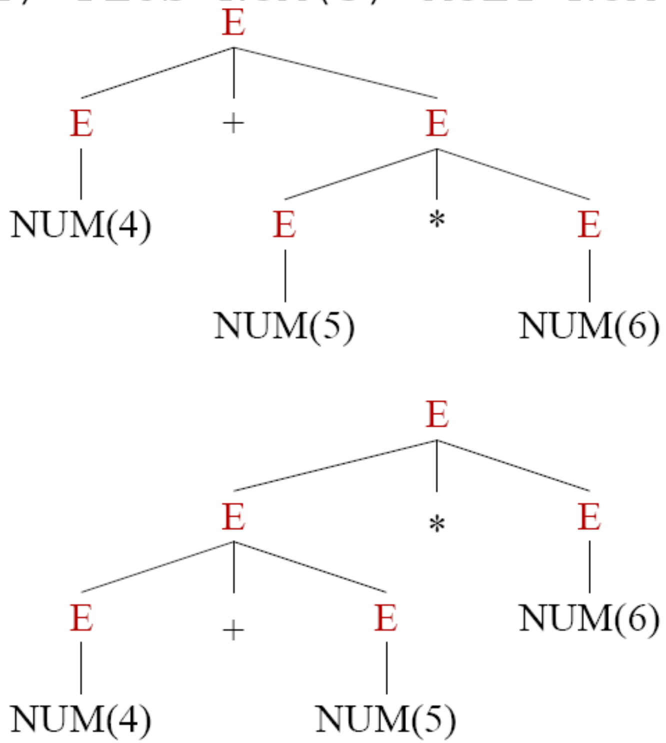
Terminals (tokens):

```
ID
NUM
PLUS   "+"
MULT   "*"
```

$$expr \rightarrow ID$$
$$expr \rightarrow NUM$$
$$expr \rightarrow expr + expr$$
$$expr \rightarrow expr * expr$$

Consider: $4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`



# Ambiguous Grammars

- *Problem*: compilers use parse trees to interpret meaning of parsed expressions.
  - Different parse trees may have different meanings, resulting in different interpreted results.
  - For example, does $4 + 5 * 6$ equal 34 or 54?

# Ambiguous Grammars

- *Problem*: compilers use parse trees to interpret meaning of parsed expressions.
  - Different parse trees may have different meanings, resulting in different interpreted results.
  - For example, does $4 + 5 * 6$ equal 34 or 54?
- *Solution*: rewrite grammar to eliminate ambiguity.
  - If language doesn't have unambiguous grammar, then you have a bad programming language.
  - Operators have a relative *precedence*. We say some operands *bind tighter* than others. ("*" binds tighter than "+")
  - Operators with the same precedence must be resolved by *associativity*. Some operators have *left associativity*, others have *right associativity*.

plus, minus, times,…

exponentiation,…

# Ambiguous Grammars

Operators with same precedence should have same associativity Why?

Example: suppose plus right-assoc, minus left-assoc, equal precedence

a – b – c + d + e

(a – b) – c + (d + e)

Conflict: ((a – b) – c) + (d + e) vs. (a – b) – (c + (d + e))

((10 – 4) – 3) + (20 + 15) =
(6 – 3) + 35 = 3 + 35 = 38

(10 - 4) – (3 + (20 + 15)) =
6 – (3 + 35) = 6 – 38 = -32

Next: how to rewrite an ambiguous grammar

# Ambiguous Grammars

Non-Terminals:

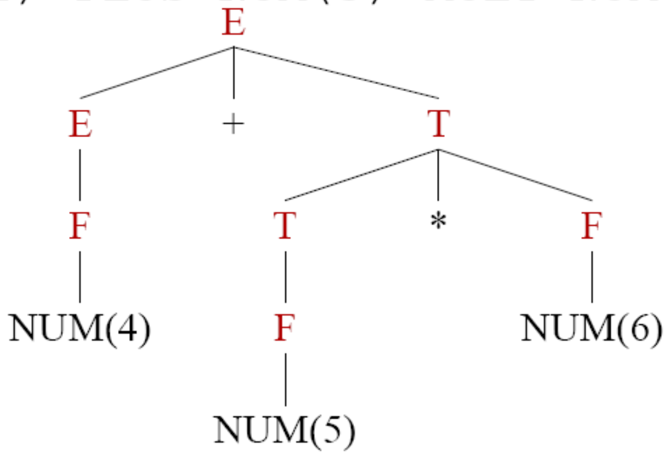
```
expr : Expression
term : Term (add)
fact : Factor (mult)
```

Terminals (tokens):

$$expr \rightarrow expr + term$$
$$expr \rightarrow term$$
$$term \rightarrow term * fact$$
$$term \rightarrow fact$$
$$fact \rightarrow ID$$
$$fact \rightarrow NUM$$

$4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`



# End-Of-File Marker

- Parse must also recognize the End-of-File (EOF).
- EOF marker in the grammar is "$"
- Introduce new start symbol and the production $E' \rightarrow \; E\$$

# Grammars and Lexical Analysis

CFG's are sufficiently powerful to describe regular expressions.

Example: language (a | b)* abb is described by the following CFG

$W \rightarrow aW \qquad X \rightarrow bY$

$W \rightarrow bW \qquad Y \rightarrow b$

$W \rightarrow aX \qquad$ (W start symbol)

So can combine lexical and syntactic analysis (parsing) into one module

+ ok for small languages, with simple/few RE's and syntactic grammar

- regular expression specification arguably more concise

- separating phases increases compiler modularity

## Context-Free Grammars versus REs (I)

**Claim:** context-free grammars are strictly more powerful that RE's (and hence NFA's and DFA's).

**Part 1**: any language that can be generated using RE's can be generated by a CFG.

**Proof**: give a translation that transforms an RE **R** into a CFG **G**(**R**) such that L( **R** ) = L ( **G**(**R**) ).

**Part2**: define a grammar **G** such that there is no finite automaton **F** with L(**G**)=L( **F** ).

# Context Free Grammars and REs (II)

Part 1: any language that can be generated using RE's can be generated by a CFG.
Proof: give a translation that transforms an RE **R** into a CFG **G**(**R**) such that L( **R** ) = L ( **G**(**R**) ).

Construction of the translation by **induction** over structure of RE's:

1. Introduce one non-terminal (**R**) for each subterm **R** of the RE

# Context Free Grammars and REs (II)

Part 1: any language that can be generated using RE's can be generated by a CFG.
Proof: give a translation that transforms an RE **R** into a CFG **G**(**R**) such that L( **R** ) = L ( **G**(**R**) ).

Construction of the translation by **induction** over structure of RE's:

1. Introduce one non-terminal (**R**) for each subterm **R** of the RE
2. Rules (base cases):
   i. Symbol (**a**): **R** $\rightarrow$ **a**
   ii. Epsilon(**ϵ**): **R** $\rightarrow$ **ϵ**

# Context Free Grammars and REs (II)

Part 1: any language that can be generated using RE's can be generated by a CFG.
Proof: give a translation that transforms an RE **R** into a CFG **G**(**R**) such that L( **R** ) = L ( **G**(**R**) ).

Construction of the translation by **induction** over structure of RE's:

1. Introduce one non-terminal (**R**) for each subterm **R** of the RE
2. Rules (base cases):
   i. Symbol (**a**): **R** → **a**
   ii. Epsilon(**ϵ**): **R** → **ϵ**
3. Rules (inductive cases):
   i. Alternation (**M | N** ): **R** → **M** **R** → **N**
   ii. Concatenation (**M N**): **R** → **M N**
   iii. Kleene closure (**M ***): **R** → **M R** **R** → **ε**

# Context-Free Grammar with no RE/FA

Suggestions?

NFA? DFA?

# Context-Free Grammar with no RE/FA

NFA? DFA?

**Well-bracketing!**

$$S \rightarrow ( S )$$
$$S \rightarrow \varepsilon$$

**Either one is fine!**

**Proof by contradiction**:
assume there's a NFA/DFA accepting the language

- FA must have **finite** number of states, say **N**
- FA must "remember" number of "(" to generate equally many ")"
- At or before seeing the **N**+1st "(", FA must revisit a state already traversed, say s
- s represents two different counts of ")", both of which must be accepted
- One count will be invalid, ie not match up the number of "("'s seen.

# Grammars versus automata models (for the curious)

Q: is there a class of grammars capturing exactly the power of RE's?
A: yes, the "regular, right-linear, finite state grammars" for example

non-terminals occur only in right-most position of RHS

at most one non-terminal in each RHS

Q: is there an automaton model that precisely captures context-free grammars?
A: yes, "push-down automata" (ie automata with a stack)

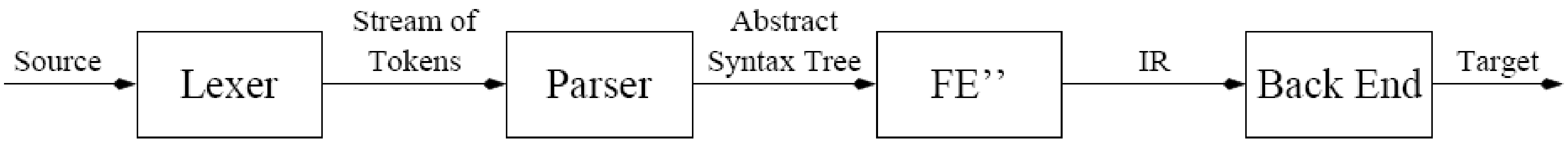
Q: we use context-**free** grammars to specify parsing. Are there grammars that are not context-free?
A: yes, context-**sensitive** grammars, for example (LHS of productions are strings with > 0 nonterminals and >=0 terminals)

# Parsing

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

Our Compiler:




Parser Functions:

- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

# Outline

- Recursive Descent Parsing
- Shift-Reduce Parsing
- ML-Yacc
- Recursive Descent Parser Generation

# Recursive Descent Parsing

Reminder: context-free grammars: symbols (terminals, non-terminals), productions (rules), derivations, …

Many CFG's can be parsed with an algorithm called **recursive descent:**

- one function for each non-terminal
- each production rule yields one clause in the function for the LHS symbol
- functions (possibly mutually) recursive

Other names for recursive descent:

- **predictive** parsing: rule selection is based on next symbol(s) seen
- top-down parsing (algorithm starts with initial symbol)
- LL(1) parsing (**L**eft-to-right parsing, **L**eftmost derivation, **1** symbol look-ahead)

# Recursive descent: example

**Grammar:**

non-terminals: `S, L, E`

terminals: `IF`(*if*), `THEN`(*then*), `ELSE`(*else*), `BEGIN`(*begin*), `PRINT`(*print*), `END`(*end*), `SEMI`(*;*), `NUM`, `EQ`(=)

$S \rightarrow \mathit{if}\ E\ \mathit{then}\ S\ \mathit{else}\ S$

$S \rightarrow \mathit{begin}\ S\ L$

$S \rightarrow \mathit{print}\ E$

$L \rightarrow \mathit{end}$

$L \rightarrow ;\ S\ L$

$E \rightarrow \mathit{num} = \mathit{num}$

```
datatype token = EOF | IF | THEN | ELSE | BEGIN |
                 PRINT | END | SEMI | NUM | EQ

val tok = ref (getToken())
fun advance() = tok := getToken()
fun eat(t) = if (!tok = t) then advance() else error()

fun S() = case !tok of
            IF    => (eat(IF); E(); eat(THEN); S();
                      eat(ELSE); S())
            BEGIN => (eat(BEGIN); S(); L())
            PRINT => (eat(PRINT); E())
and L() = case !tok of
            END   => (eat(END))
            SEMI  => (eat(SEMI); S(); L())
and E() =            (eat(NUM); eat(EQ); eat(NUM))
```

# Recursive descent: another example

**Grammar:**

$A \rightarrow S\ EOF$
$S \rightarrow id := E$
$S \rightarrow print\ (\ L\ )$

$E \rightarrow id$
$E \rightarrow num$
$L \rightarrow E$
$L \rightarrow L, E$

```
fun A() =                  (S(); eat(EOF))
and S() = case !tok of
            ID       => (eat(ID); eat(ASSIGN); E())
            PRINT    => (eat(PRINT); eat(LPAREN);
                          L(); eat(RPAREN))
and E() = case !tok of
            ID       => (eat(ID))
            NUM      => (eat(NUM))
and L() = case !tok of
            ID       => (?????)
            NUM      => (?????)
```

# The Problem

- If !tok = ID, parser cannot determine which production to use:

  $L \rightarrow E$ (E could be ID)

  $L \rightarrow L, E$ (L could be ID)

- Predictive parsing only works for grammars where first terminal symbol of each subexpression provides enough information to choose which production to use.
- Can write predictive parser by eliminating *left recursion*.

$$
\begin{array}{l} L \rightarrow E \\ L \rightarrow L, E \end{array} \quad \Longrightarrow \quad \begin{array}{l} L \rightarrow E\,M \\ M \rightarrow , E\,M \\ M \rightarrow \epsilon \end{array}
$$

A → S EOF
S → id := E
S → print ( L )
E → id
E → num

```
and L() = case !tok of
            ID     => (E(); M())
            NUM    => (E(); M())
and M() = case !tok of
            COMMA    => (eat(COMMA); E(); M())
            RPAREN   => ()
```

# The Problem

- If !tok = ID, parser cannot determine which production to use:

  $L \rightarrow E$ (E could be ID)

  $L \rightarrow L, E$ (L could be ID)

- Predictive parsing only works for grammars where first terminal symbol of each subexpression provides enough information to choose which production to use.
- Can write predictive parser by eliminating *left recursion*.

$$\begin{array}{l} L \rightarrow E \\ L \rightarrow L, E \end{array} \quad \Longrightarrow \quad \begin{array}{l} L \rightarrow E\ M \\ M \rightarrow\ \boxed{,}\ E\ M \\ M \rightarrow \epsilon \end{array}$$

A ⟶ S EOF
S ⟶ id := E
S ⟶ print ( L )
E ⟶ id
E ⟶ num

```
and L() = case !tok of
            ID      => (E(); M())
            NUM     => (E(); M())
and M() = case !tok of
            COMMA     => (eat(COMMA); E(); M())
            RPAREN    => ()
```

RPAREN is the (only) nonterminal that may **follow** a string derivable from M (why?...). Seeing RPAREN thus indicates we should use rule M ⟶ ε.

# Limitation of Recursive Descent Parsing

- Based on current function and next token-type in input stream, parser must predict which production to use.
- If !tok = ID, parser cannot determine which production to use:

  $L \rightarrow E$ (E could be ID)

  $L \rightarrow L, E$ (L could be ID)

- Predictive parsing only works for grammars where first terminal symbol of each subexpression provides enough information to choose which production to use.

Sometimes, we can modify a grammar so that is amenable to predictive parsing without changing the language recognized, for example by replacing left recursion by right recursion.

Need **algorithm** that

- **decides** if a grammar is suitable for recursive descent parsing, and
- if so, calculates the **parsing table**, ie a table indicating which production to apply in each situation, given the next terminal in the token stream

Key ingredient: analysis of **first** and **follow**.

# Formal Techniques for predictive parsing

Let γ range over strings of terminals and nonterminals from our grammar.

(i.e. RHS of grammar rules)

## FIRST(γ)

For each γ that is a RHS of one of the rules, must determine the set of all terminals that can begin a string derivable from γ: **First(γ)**

**Examples**: First (id := E) = { id } for grammar on previous slide.

# Formal Techniques for predictive parsing

Let γ range over strings of terminals and nonterminals from our grammar.

(i.e. RHS of grammar rules)

## FIRST(γ)

For each γ that is a RHS of one of the rules, must determine the set of all terminals that can begin a string derivable from γ: **First(γ)**

**Examples**: First (id := E) = { id } for grammar on previous slide.

First ( T * F ) = { id, num, LPAREN } for

$T \rightarrow id$
$T \rightarrow num$
$T \rightarrow ( E )$
$F \rightarrow . . .$

# Formal Techniques for predictive parsing

Let γ range over strings of terminals and nonterminals from our grammar.

(i.e. RHS of grammar rules)

## FIRST(γ)

For each γ that is a RHS of one of the rules, must determine the set of all terminals that can begin a string derivable from γ: **First(γ)**

**Examples**: First (id := E) = { id } for grammar on previous slide.

First ( T * F ) = { id, num, LPAREN } for

$T \rightarrow id$
$T \rightarrow num$
$T \rightarrow ( E )$
$F \rightarrow . . .$

Predictive parsing impossible whenever there are productions $X \rightarrow γ1$ and $X \rightarrow γ2$ such that First (γ1) and First (γ2) overlap: can't decide which rule to use!

# Formal Techniques for predictive parsing

Consider $\gamma$ = X Y Z where

$S \rightarrow X\ Y\ Z$

$Y \rightarrow \ldots$
$Y \rightarrow \varepsilon$

$X \rightarrow Y$
$Z \rightarrow \ldots$

Then, First (S) should contain First (Z), because Y and X can derive $\varepsilon$.

# NULLABLE(N)

$S \rightarrow X\ Y\ Z$

$Y \rightarrow \ldots$
$Y \rightarrow \varepsilon$

$X \rightarrow Y$
$Z \rightarrow \ldots$

Consider $\gamma = X\ Y\ Z$ where

Then, First (S) should contain First (Z), because Y and X can derive ε.

Hence, for calculating First sets we need to determine which nonterminals N can derive the empty string ε: **Nullable(N)**.

Here, Y and (hence) X are nullable.

Extension to strings: γ nullable if all symbols in γ are nullable.

# Computation of First

- If $T$ is a terminal symbol, then $\text{first}(T) = \{T\}$.

# Computation of First

- If $T$ is a terminal symbol, then first$(T) = \{T\}$.
- If $X$ is a non-terminal and $X \rightarrow Y_1Y_2Y_3...Y_n$, then

# Computation of First

- If $T$ is a terminal symbol, then first($T$) = $\{T\}$.
- If $X$ is a non-terminal and $X \rightarrow Y_1Y_2Y_3...Y_n$, then

first($Y_1$) $\in$ first($X$)
first($Y_2$) $\in$ first($X$), if $Y_1$ is nullable
first($Y_3$) $\in$ first($X$), if $Y_1$, $Y_2$ is nullable
$\vdots$
first($Y_n$) $\in$ first($X$), if $Y_1$, $Y_2$, ... $Y_{n-1}$ is nullable

# Computation of First

- If $T$ is a terminal symbol, then $\text{first}(T) = \{T\}$.
- If $X$ is a non-terminal and $X \rightarrow Y_1Y_2Y_3...Y_n$, then

$$
\begin{aligned}
&\text{first}(Y_1) \in \text{first}(X)\\
&\text{first}(Y_2) \in \text{first}(X), \text{ if } Y_1 \text{ is nullable}\\
&\text{first}(Y_3) \in \text{first}(X), \text{ if } Y_1, Y_2 \text{ is nullable}\\
&\vdots\\
&\text{first}(Y_n) \in \text{first}(X), \text{ if } Y_1, Y_2, \ldots\ Y_{n-1} \text{ is nullable}
\end{aligned}
$$

Similarly: first(γ) where γ is a string of terminals and nonterminals.

# FOLLOW(N)

For each nonterminal **N**, we also must determine the set of all terminal symbols t that can immediately follow **N** in a derivation: **Follow(N).**
In particular, must know which terminals may follow a nullable nonterminal.

# FOLLOW(N)

For each nonterminal N, we also must determine the set of all terminal symbols t that can immediately follow N in a derivation: **Follow(N).**
In particular, must know which terminals may follow a nullable nonterminal.

Example (S start symbol):

$X \rightarrow Y$
$S \rightarrow P\ X\ t$
$S \rightarrow u$

$Y \rightarrow \ldots$
$Y \rightarrow \varepsilon$
$P \rightarrow \ldots$

- Follow(X) contains t, due to $S \rightarrow P\ X\ t$

# FOLLOW(N)

For each nonterminal N, we also must determine the set of all terminal symbols t that can immediately follow N in a derivation: **Follow(N).**
In particular, must know which terminals may follow a nullable nonterminal.

Example (S start symbol):

| | | |
|---|---|---|
| X | $\rightarrow$ | Y |
| S | $\rightarrow$ | P X t |
| S | $\rightarrow$ | u |

| | | |
|---|---|---|
| Y | $\rightarrow$ | ... |
| Y | $\rightarrow$ | ε |
| P | $\rightarrow$ | ... |

- Follow(X) contains t, due to S $\rightarrow$ P X t
  - hence Follow(P) contains t, since X is nullable
    - in case P is also nullable, First (PXt) and hence First(S) contain t

# FOLLOW(N)

For each nonterminal N, we also must determine the set of all terminal symbols t that can immediately follow N in a derivation: **Follow(N).**
In particular, must know which terminals may follow a nullable nonterminal.

Example (S start symbol):

$X \rightarrow Y$
$S \rightarrow P\ X\ t$
$S \rightarrow u$

$Y \rightarrow \ldots$
$Y \rightarrow \varepsilon$
$P \rightarrow \ldots$

- Follow(X) contains t, due to $S \rightarrow P\ X\ t$
    - hence Follow(P) contains t, since X is nullable
        - in case P is also nullable, First (PXt) and hence First(S) contain t
    - hence Follow(Y) contains t, due to $X \rightarrow Y$

# Computation of Follow

Let X and Y be nonterminals, and γ, γ1 strings of terminals and nonterminals.

- whenever the grammar contains a production $X \rightarrow \gamma Y$:

$$\boxed{\text{follow}(X) \subseteq \text{follow}(Y)}$$

# Computation of Follow

Let X and Y be nonterminals, and γ, γ1 strings of terminals and nonterminals.

- whenever the grammar contains a production $X \rightarrow \gamma Y$:

$$\boxed{\text{follow}(X) \subseteq \text{follow}(Y)}$$

- whenever the grammar contains a production $X \rightarrow \gamma\ Y\ \delta$:

$$\text{first}(\delta) \subseteq \text{follow}(Y)$$

$$\text{follow}(X) \subseteq \text{follow}(Y) \text{ whenever } \delta \text{ is nullable}$$

# Computation of Follow

Let X and Y be nonterminals, and γ, γ1 strings of terminals and nonterminals.

- whenever the grammar contains a production $X \longrightarrow \gamma Y$:

$$\boxed{\text{follow}(X) \subseteq \text{follow}(Y)}$$

- whenever the grammar contains a production $X \longrightarrow \gamma\ Y\ \delta$:

$$\boxed{\begin{array}{c} \text{first}(\delta) \subseteq \text{follow}(Y) \\ \text{follow}(X) \subseteq \text{follow}(Y) \text{ whenever } \delta \text{ is nullable} \end{array}}$$

Iteratively visit grammar productions to compute nullable, first, follow for each nonterminal in grammar.

# Computation of Follow

Let X and Y be nonterminals, and γ, γ1 strings of terminals and nonterminals.

- whenever the grammar contains a production $X \longrightarrow \gamma Y$:

$$\boxed{\text{follow}(X) \subseteq \text{follow}(Y)}$$

- whenever the grammar contains a production $X \longrightarrow \gamma\, Y\, \delta$:

$$\text{first}(\delta) \subseteq \text{follow}(Y)$$

$$\text{follow}(X) \subseteq \text{follow}(Y) \text{ whenever } \delta \text{ is nullable}$$

Iteratively visit grammar productions to compute nullable, first, follow for each nonterminal in grammar.

want smallest sets!

- **start** the iteration with **empty first** and **follow sets** and **nullable=No** for all nonterminals and only add elements if forced to do so.

# Computation of Follow

Let X and Y be nonterminals, and γ, γ1 strings of terminals and nonterminals.

- whenever the grammar contains a production $X \longrightarrow \gamma Y$:

$$\boxed{\text{follow}(X) \subseteq \text{follow}(Y)}$$

- whenever the grammar contains a production $X \longrightarrow \gamma\, Y\, \delta$:

$$\text{first}(\delta) \subseteq \text{follow}(Y)$$

$$\text{follow}(X) \subseteq \text{follow}(Y) \text{ whenever } \delta \text{ is nullable}$$

Iteratively visit grammar productions to compute nullable, first, follow for each nonterminal in grammar.

want smallest sets!

- **start** the iteration with **empty first** and **follow sets** and **nullable=No** for all nonterminals and only add elements if forced to do so.
- may need to visit some rules repeatedly. Order in which rules are visited affects the number of iterations needed, but not the final result.

# Nullable, First, Follow: example

$$Z \xrightarrow{1} d \qquad Y \xrightarrow{3} \varepsilon \qquad X \xrightarrow{5} Y$$
$$Z \xrightarrow{2} X\,Y\,Z \qquad Y \xrightarrow{4} c \qquad X \xrightarrow{6} a$$

| | nullable | first | follow |
|---|---|---|---|
| X | No | | |
| Y | No | | |
| Z | No | | |

Visit rules
1, 3, 4, 6 $\longrightarrow$

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\quad$ $Y \xrightarrow{3} \varepsilon$ $\quad$ $X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z$ $\quad$ $Y \xrightarrow{4} c$ $\quad$ $X \xrightarrow{6} a$

| | nullable | first | follow |
|---|---|---|---|
| X | No | | |
| Y | No | | |
| Z | No | | |

Visit rules 1, 3, 4, 6

| | nullable | first | follow |
|---|---|---|---|
| X | No | a(6) | |
| Y | Yes (3) | c(4) | |
| Z | No | d(1) | |

Visit rule 5

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\quad Y \xrightarrow{3} \varepsilon$ $\quad X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z$ $\quad Y \xrightarrow{4} c$ $\quad X \xrightarrow{6} a$

| | nullable | first | follow |
|---|---|---|---|
| X | No | | |
| Y | No | | |
| Z | No | | |

Visit rules 1, 3, 4, 6

| | nullable | first | follow |
|---|---|---|---|
| X | No | a(6) | |
| Y | Yes (3) | c(4) | |
| Z | No | d(1) | |

Visit rule 5

| | nullable | first | follow |
|---|---|---|---|
| X | Yes(5) | a, c(5) | |
| Y | Yes | c | |
| Z | No | d | |

# Nullable, First, Follow: example

$Z \xrightarrow{1} d \qquad Y \xrightarrow{3} \varepsilon \qquad X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z \qquad Y \xrightarrow{4} c \qquad X \xrightarrow{6} a$

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | |
| Y | Yes | c | |
| Z | No | d | |

.....

Visit rule 2

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\quad Y \xrightarrow{3} \varepsilon$ $\quad X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z$ $\quad Y \xrightarrow{4} c$ $\quad X \xrightarrow{6} a$

.....

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | |
| Y | Yes | c | |
| Z | No | d | |

Visit rule 2

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | ? |
| Y | Yes | c | ? |
| Z | No | d, a(2), c(2) | ? |

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\quad$ $Y \xrightarrow{3} \varepsilon$ $\quad$ $X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\ Y\ Z$ $\quad$ $Y \xrightarrow{4} c$ $\quad$ $X \xrightarrow{6} a$

.....

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | |
| Y | Yes | c | |
| Z | No | d | |

Visit rule 2

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | c (2) |
| Y | Yes | c | ? |
| Z | No | d, a, c | ? |

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\qquad$ $Y \xrightarrow{3} \varepsilon$ $\qquad$ $X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\ Y\ Z$ $\qquad$ $Y \xrightarrow{4} c$ $\qquad$ $X \xrightarrow{6} a$

.....

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | |
| Y | Yes | c | |
| Z | No | d | |

Visit rule 2

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | c |
| Y | Yes | c | a, c, d (2) |
| Z | No | d, a, c | ? |

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\quad Y \xrightarrow{3} \varepsilon$ $\quad X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\ Y\ Z$ $\quad Y \xrightarrow{4} c$ $\quad X \xrightarrow{6} a$

…..

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | |
| Y | Yes | c | |
| Z | No | d | |

Visit rule 2

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | c |
| Y | Yes | c | a, c, d |
| Z | No | d, a, c | ? |

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | c, a(2), d(2) |
| Y | Yes | c | a, c, d |
| Z | No | d, a, c | ? |

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\quad Y \xrightarrow{3} \varepsilon$ $\quad X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z$ $\quad Y \xrightarrow{4} c$ $\quad X \xrightarrow{6} a$

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | c, a, d |
| Y | Yes | c | a, c, d |
| Z | No | d, a, c | |

…..

next rule?

# Nullable, First, Follow: example

$Z \xrightarrow{1} d$ $\quad Y \xrightarrow{3} \varepsilon$ $\quad X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z$ $\quad Y \xrightarrow{4} c$ $\quad X \xrightarrow{6} a$

.....

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | c, a, d |
| Y | Yes | c | a, c, d |
| Z | No | d, a, c | |

Visit rules 1-6

| | nullable | first | follow |
|---|---|---|---|
| X | Yes | a, c | c, a, d |
| Y | Yes | c | a, c, d |
| Z | No | d, a, c | |

(no change)

# Parse table extraction

- contains “action” for each nonterminal * terminal pair (N,t)
- predictive parsing: “action” is “rule to be applied” when seeing token type t during execution of function for nonterminal N
- lookup attempt in blank cells indicates syntax error (phrase rejected)

| | a | c | d |
|---|---|---|---|
| X | | | |
| Y | | | |
| Z | | | |

# Parse table extraction

**Filling the table:**

1. Add rule N $\longrightarrow$ γ in row N, column **t** whenever **t** occurs in First(γ).

Motivation: seeing **t** when expecting N selects rule N $\longrightarrow$ γ.

| | a | c | d |
|---|---|---|---|
| X | | | |
| Y | | | |
| Z | | | |

# Parse table extraction

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Z | $\xrightarrow{1}$ | d | Y | $\xrightarrow{3}$ | ε | X | $\xrightarrow{5}$ Y |
| Z | $\xrightarrow{2}$ | X Y Z | Y | $\xrightarrow{4}$ | c | X | $\xrightarrow{6}$ a |

| | null. | first | follow |
|---|---|---|---|
| X | Yes | ac | acd |
| Y | Yes | c | acd |
| Z | No | acd | |

Add rule N $\longrightarrow$ γ in row N, column **t** whenever **t** occurs in First(γ).

| | a | c | d |
|---|---|---|---|
| X | | | |
| Y | | | |
| Z | | | |

# Parse table extraction

$Z \xrightarrow{1} d$ $\quad$ $Y \xrightarrow{3} \varepsilon$ $\quad$ $X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z$ $\quad$ $Y \xrightarrow{4} c$ $\quad$ $X \xrightarrow{6} a$

| | null. | first | follow |
|---|---|---|---|
| X | Yes | ac | acd |
| Y | Yes | c | acd |
| Z | No | acd | |

Add rule $N \rightarrow \gamma$ in row N, column t whenever t occurs in First(γ).

| | a | c | d |
|---|---|---|---|
| X | $X \rightarrow a$ | | |
| Y | | $Y \rightarrow c$ | |
| Z | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow d$<br>$Z \rightarrow X\,Y\,Z$ |

first (XYZ) = {a,c,d}

# Parse table extraction

$Z \xrightarrow{1} d$ $\quad Y \xrightarrow{3} \varepsilon$ $\quad X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\ Y\ Z$ $\quad Y \xrightarrow{4} c$ $\quad X \xrightarrow{6} a$

| | null. | first | follow |
|---|---|---|---|
| X | Yes | ac | acd |
| Y | Yes | c | acd |
| Z | No | acd | |

Add rule $N \rightarrow \gamma$ in row N, column t whenever t occurs in First(γ).

| | a | c | d |
|---|---|---|---|
| X | $X \rightarrow a$ | $X \rightarrow Y$ | |
| Y | | $Y \rightarrow c$ | |
| Z | $Z \rightarrow X\ Y\ Z$ | $Z \rightarrow X\ Y\ Z$ | $Z \rightarrow d$<br>$Z \rightarrow X\ Y\ Z$ |

2. If γ is nullable, add rule $N \rightarrow \gamma$ in row N, column t whenever t in Follow(N).

Motivation: seeing t when expecting N selects rule $N \rightarrow \gamma$.

# Parse table extraction

| | | |
|---|---|---|
| $Z \xrightarrow{1} d$ | $Y \xrightarrow{3} \varepsilon$ | $X \xrightarrow{5} Y$ |
| $Z \xrightarrow{2} X\,Y\,Z$ | $Y \xrightarrow{4} c$ | $X \xrightarrow{6} a$ |

| | null. | first | follow |
|---|---|---|---|
| X | Yes | ac | acd |
| Y | Yes | c | acd |
| Z | No | acd | |

Add rule $N \rightarrow \gamma$ in row N, column t whenever t occurs in First(γ).

If γ is nullable, add rule $N \rightarrow \gamma$ in row N, column t whenever t in Follow(N).

| | a | c | d |
|---|---|---|---|
| X | $X \rightarrow a$<br>$X \rightarrow Y$ | $X \rightarrow Y$ | $X \rightarrow Y$ |
| Y | $Y \rightarrow \varepsilon$ | $Y \rightarrow c$<br>$Y \rightarrow \varepsilon$ | $Y \rightarrow \varepsilon$ |
| Z | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow d$<br>$Z \rightarrow X\,Y\,Z$ |

# Parse table extraction

$Z \xrightarrow{1} d$ $\quad$ $Y \xrightarrow{3} \varepsilon$ $\quad$ $X \xrightarrow{5} Y$

$Z \xrightarrow{2} X\,Y\,Z$ $\quad$ $Y \xrightarrow{4} c$ $\quad$ $X \xrightarrow{6} a$

| | null. | first | follow |
|---|---|---|---|
| X | Yes | ac | acd |
| Y | Yes | c | acd |
| Z | No | acd | |

Add rule $N \rightarrow \gamma$ in row N, column t whenever t occurs in First(γ).

If γ is nullable, add rule $N \rightarrow \gamma$ in row N, column t whenever t in Follow(N).

| | **a** | **c** | **d** |
|---|---|---|---|
| X | $X \rightarrow a$<br>$X \rightarrow Y$ | $X \rightarrow Y$ | $X \rightarrow Y$ |
| Y | $Y \rightarrow \varepsilon$ | $Y \rightarrow c$<br>$Y \rightarrow \varepsilon$ | $Y \rightarrow \varepsilon$ |
| Z | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow d$<br>$Z \rightarrow X\,Y\,Z$ |

**Observation: some cells contain more than one rule – which one should be used???**

# Predictive Parsing Table

If the predictive parsing table contains *no* duplicate entries, can build predictive parser for grammar.

- Grammar is LL(1) (left-to-right parse, left-most derivation, 1 symbol lookahead).
- Grammar is LL(k) if its LL(k) predictive parsing table has no duplicate entries.
  - Rows correspond to non-terminals, columns correspond to every possible sequence of k terminals.
  - The first($\gamma$) = set of all k-length terminal sequences that can begin any string derived from $\gamma$.
  - LL(k) paring tables can be too large.
  - Ambiguous grammars are not LL(k), $\forall$ k.

| | aa | .. | dd |
|---|---|---|---|
| X | | | |
| Y | | | |
| Z | | | |

(k=2)

# Another Example

| | | |
|---|---|---|
| $S' \rightarrow S\$$ | $S \rightarrow$ IF $E$ THEN $A$ ELSE $A$ | $T \rightarrow$ NUM |
| $S \rightarrow E$ | $E \rightarrow E + T$ | $A \rightarrow$ ID = NUM |
| $S \rightarrow$ IF $E$ THEN $A$ | $E \rightarrow T$ | |

**Iteration 1:**

| | nullable | first | follow |
|---|---|---|---|
| $S'$ | no | | |
| $S$ | no | IF | \$ |
| $E$ | no | | \$, THEN, + |
| $T$ | no | NUM | \$, THEN, + |
| $A$ | no | ID | \$, ELSE |

**Iteration 2:**

| | nullable | first | follow |
|---|---|---|---|
| $S'$ | no | IF | |
| $S$ | no | IF | \$ |
| $E$ | no | NUM | \$, THEN, + |
| $T$ | no | NUM | \$, THEN, + |
| $A$ | no | ID | \$, ELSE |

# Another Example

$S' \rightarrow S\$$

$S \rightarrow E$

$S \rightarrow$ IF $E$ THEN $A$

$S \rightarrow$ IF $E$ THEN $A$ ELSE $A$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow$ NUM

$A \rightarrow$ ID = NUM

**Iteration 3:**

| | nullable | first | follow |
|---|---|---|---|
| $S'$ | no | IF | |
| $S$ | no | IF, NUM | \$ |
| $E$ | no | NUM | \$, THEN, + |
| $T$ | no | NUM | \$, THEN, + |
| $A$ | no | ID | \$, ELSE |

**Iteration 4:**

| | nullable | first | follow |
|---|---|---|---|
| $S'$ | no | IF, NUM | |
| $S$ | no | IF, NUM | \$ |
| $E$ | no | NUM | \$, THEN, + |
| $T$ | no | NUM | \$, THEN, + |
| $A$ | no | ID | \$, ELSE |

**No futher changes**

# Predictive Parsing Table

| | nullable | first | follow |
|---|---|---|---|
| $S'$ | no | IF, NUM | |
| $S$ | no | IF, NUM | \$ |
| $E$ | no | NUM | \$, THEN, + |
| $T$ | no | NUM | \$, THEN, + |
| $A$ | no | ID | \$, ELSE |

Build *predictive parsing table* from nullable, first, and follow sets.

| | IF | THEN | ELSE | + | NUM | ID | = | \$ |
|---|---|---|---|---|---|---|---|---|
| $S'$ | $S' \to S$ | | | | $S' \to S$ | | | |
| $S$ | $S \to$ IF $E$ THEN $A$<br>$S \to$ IF $E$ THEN $A$ ELSE $A$ | | | | $S \to E$ | | | |
| $E$ | | | | | $E \to E + T$<br>$E \to T$ | | | |
| $T$ | | | | | $T \to$ NUM | | | |
| $A$ | | | | | | $A \to$ ID = NUM | | |

Table has duplicate entries $\Rightarrow$ grammar is not LL(1)!

# Problem 1

1. $E \rightarrow E + T$
   $E \rightarrow T$

- first(E+T) = first(T)
- When in function `E()`, if next token is NUM, parser will get stuck.
- Grammar is *left-recursive* - left-recursive grammars cannot be LL(1).
- Solution: rewrite grammar so that it is *right-recursive*.

  $E \rightarrow TE'$
  $E' \rightarrow \epsilon$
  $E' \rightarrow +TE'$

- In general, $\begin{array}{l} X \rightarrow X\gamma \\ X \rightarrow \alpha \end{array}$ derives strings of form $\alpha\gamma^*$ ( $\alpha$ doesn't start with $X$).

  These two productions can be rewritten as follows:

  $X \rightarrow \alpha X'$
  $X' \rightarrow \epsilon$
  $X' \rightarrow \gamma X'$

# Problem 2

2. $S \rightarrow$ IF $E$ THEN $A$
   $S \rightarrow$ IF $E$ THEN $A$ ELSE $A$

- Two productions begin with same symbol.
- first(IF $E$ THEN $A$) = first(IF $E$ THEN $A$ ELSE $A$)
- Solution: use *left-factoring*

  $S \rightarrow$ IF $E$ THEN $A$ $V$

  $V \rightarrow \epsilon$

  $V \rightarrow$ ELSE $A$

# Example (try at home)

Show that modified grammar is LL(1).

$$
\begin{array}{lll}
S' \rightarrow S\$ & V \rightarrow \text{ ELSE } A & T \rightarrow \text{ NUM} \\
S \rightarrow E & E \rightarrow TE' & A \rightarrow \text{ ID} = \text{NUM} \\
S \rightarrow \text{ IF } E \text{ THEN } A\ V & E' \rightarrow \epsilon & \\
V \rightarrow \epsilon & E' \rightarrow +TE & 
\end{array}
$$

# Example

Show that modified grammar is LL(1). Build predictive parsing table.

| | nullable | first | follow |
|---|---|---|---|
| $S'$ | no | IF,NUM | |
| $S$ | no | IF,NUM | \$ |
| $V$ | yes | ELSE | \$ |
| $E$ | no | NUM | \$, THEN |
| $E'$ | yes | + | \$, THEN |
| $T$ | no | NUM | \$, THEN, + |
| $A$ | no | ID | \$, ELSE |

| | IF | THEN | ELSE | + | NUM | ID | = | \$ |
|---|---|---|---|---|---|---|---|---|
| $S'$ | $S' \to S$ | | | | $S' \to S$ | | | |
| $S$ | $S \to$ IF $E$ THEN $A$ $V$ | | | | $S \to E$ | | | |
| $V$ | | | $V \to$ ELSE $A$ | | | | | $V \to \epsilon$ |
| $E$ | | | | | $E \to TE'$ | | | |
| $E'$ | | $E' \to \epsilon$ | | $E' \to +TE'$ | | | | $E' \to \epsilon$ |
| $T$ | | | | | $T \to$ NUM | | | |
| $A$ | | | | | | $A \to$ ID = NUM | | |

Table does not have duplicate entries $\Rightarrow$ modified grammar is LL(1)!

# Limitation of Recursive Descent Parsing

Reminder: predictive parsing selects rule based on the next input token(s)

- LL(1): single next token
- LL(k): next k tokens

Sometimes, there's no k that does the job.

# Shift-Reduce Parsing

**Idea**: delay decision until **all** tokens of an RHS have been seen

# Shift-Reduce Parsing

**Idea**: delay decision until **all** tokens of an RHS have been seen

- use a **stack** to remember ("shift") symbols
- based on stack content and next symbol, select one of two actions:

# Shift-Reduce Parsing

**Idea**: delay decision until **all** tokens of an RHS have been seen

- use a **stack** to remember ("shift") symbols
- based on stack content and next symbol, select one of two actions:
  - **shift**: push input token onto stack

# Shift-Reduce Parsing

**Idea**: delay decision until **all** tokens of an RHS have been seen

- use a **stack** to remember ("shift") symbols
- based on stack content and next symbol, select one of two actions:
  - **shift**: push input token onto stack
  - **reduce**: chose a production (X $\longrightarrow$ A B C); pop its RHS (C B A); push its LHS (X). So can only reduce if we see a full RHS on top of stack.

# Shift-Reduce Parsing

**Idea**: delay decision until **all** tokens of an RHS have been seen

- use a **stack** to remember ("shift") symbols
- based on stack content and next symbol, select one of two actions:
  - **shift**: push input token onto stack
  - **reduce**: chose a production (X $\rightarrow$ A B C); pop its RHS (C B A); push its LHS (X). So can only reduce if we see a full RHS on top of stack.
- **initial state**: empty stack, parsing pointer at beginning of token stream

# Shift-Reduce Parsing

**Idea**: delay decision until **all** tokens of an RHS have been seen

- use a **stack** to remember ("shift") symbols
- based on stack content and next symbol, select one of two actions:
  - **shift**: push input token onto stack
  - **reduce**: chose a production (X $\longrightarrow$ A B C); pop its RHS (C B A); push its LHS (X). So can only reduce if we see a full RHS on top of stack.
- **initial state**: empty stack, parsing pointer at beginning of token stream
- shifting of end marker $: input stream has been parsed successfully.
- exhaustion of input stream with nonempty stack: parse fails

# Shift-Reduce Parsing

**Idea**: delay decision until **all** tokens of an RHS have been seen

- use a **stack** to remember ("shift") symbols
- based on stack content and next symbol, select one of two actions:
  - **shift**: push input token onto stack
  - **reduce**: chose a production (X $\rightarrow$ A B C); pop its RHS (C B A); push its LHS (X). So can only reduce if we see a full RHS on top of stack.
- **initial state**: empty stack, parsing pointer at beginning of token stream
- shifting of end marker $: input stream has been parsed successfully.
- exhaustion of input stream with nonempty stack: parse fails

- a.k.a. bottom-up parsing
- a.k.a LR(k): left-to-right parse, rightmost derivation, k-token lookahead
- shift-reduce parsing can parse more grammars than predictive parsing

# Shift-Reduce Parsing

**How does parser know when to shift or reduce?**

- DFA: applied to stack contents, not input stream
- Each state corresponds to contents of stack at some point in time.
- Edges labelled with terms/non-terms that can appear on stack.

# Example

**Grammar:**

**1** A $\rightarrow$ S EOF

**2** S $\rightarrow$ *(* L *)*

**3** S $\rightarrow$ *id* = *num*

**4** L $\rightarrow$ L*;* S

**5** L $\rightarrow$ S

**Input:**

$$(a = 4; b = 5) \rightarrow (ID_a = NUM_4; ID_b = NUM_5)$$

```
 input:  ( ID = NUM ; ID = NUM )
         |
 stack:
action:  shift
```

**0**

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
    input:  ( ID = NUM ; ID = NUM )
               |
1   stack:  (
```

# Example

1. A $\rightarrow$ S EOF
2. S $\rightarrow$ ( L )
3. S $\rightarrow$ id = num
4. L $\rightarrow$ L; S
5. L $\rightarrow$ S

**1**

```
 input:  ( ID = NUM ; ID = NUM )
             |
 stack:  (
action:  shift
```

**2**

```
 input:  ( ID = NUM ; ID = NUM )
                  |
 stack:  ( ID
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
1     input:  ( ID = NUM ; ID = NUM )
                |
      stack:  (
     action:  shift
```

```
2     input:  ( ID = NUM ; ID = NUM )
                   |
      stack:  ( ID
     action:  shift
```

```
3     input:  ( ID = NUM ; ID = NUM )
                     |
      stack:  ( ID =
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
      input:  ( ID = NUM ; ID = NUM )
                 |
1     stack:  (
     action:  shift
```

```
      input:  ( ID = NUM ; ID = NUM )
                   |
2     stack:  ( ID
     action:  shift
```

```
      input:  ( ID = NUM ; ID = NUM )
                     |
3     stack:  ( ID =
     action:  shift
```

```
      input:  ( ID = NUM ; ID = NUM )
                         |
4     stack:  ( ID = NUM
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
1      input:  ( ID = NUM ; ID = NUM )
                 |
       stack:  (
      action:  shift
```

```
2      input:  ( ID = NUM ; ID = NUM )
                    |
       stack:  ( ID
      action:  shift
```

```
3      input:  ( ID = NUM ; ID = NUM )
                      |
       stack:  ( ID =
      action:  shift
```

```
4      input:  ( ID = NUM ; ID = NUM )
                          |
       stack:  ( ID = NUM
      action:  reduce 3
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
input:  ( ID = NUM ; ID = NUM )
                   |
5   stack:  ( S
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
         input:  ( ID = NUM ; ID = NUM )
                              |
5        stack:  ( S
        action:  reduce 5
-----------------------------------------
         input:  ( ID = NUM ; ID = NUM )
                              |
6        stack:  ( L
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
     input:  ( ID = NUM ; ID = NUM )
                           |
5
     stack:  ( S
    action:  reduce 5
```

```
     input:  ( ID = NUM ; ID = NUM )
                           |
6
     stack:  ( L
    action:  shift
```

```
     input:  ( ID = NUM ; ID = NUM )
                             |
7
     stack:  ( L ;
```

# Example

1. A $\rightarrow$ S EOF
2. S $\rightarrow$ ( L )
3. S $\rightarrow$ id = num
4. L $\rightarrow$ L; S
5. L $\rightarrow$ S

**5**

```
 input:  ( ID = NUM ; ID = NUM )
                     |
 stack:  ( S
action:  reduce 5
```

**6**

```
 input:  ( ID = NUM ; ID = NUM )
                     |
 stack:  ( L
action:  shift
```

**7**

```
 input:  ( ID = NUM ; ID = NUM )
                       |
 stack:  ( L ;
action:  shift
```

**8**

```
 input:  ( ID = NUM ; ID = NUM )
                          |
 stack:  ( L; ID
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

**5**
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( S
action: reduce 5
```

**6**
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( L
action: shift
```

**7**
```
 input: ( ID = NUM ; ID = NUM )
                     |
 stack: ( L ;
action: shift
```

**8**
```
 input: ( ID = NUM ; ID = NUM )
                          |
 stack: ( L; ID
action: shift
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
   input:  ( ID = NUM ; ID = NUM )
                            |
9  stack:  ( L ; ID =
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

9

```
  input:  ( ID = NUM ; ID = NUM )
                            |
  stack:  ( L ; ID =
 action:  shift
```

---

10

```
  input:  ( ID = NUM ; ID = NUM )
                                |
  stack:  ( L ; ID = NUM
```

# Example

1. A → S EOF
2. S → ( L )
3. $\boxed{\text{S → id = num}}$
4. L → L; S
5. L → S

```
9    input:  ( ID = NUM ; ID = NUM )
                          |
     stack:  ( L ; ID =
    action:  shift
```

```
10   input:  ( ID = NUM ; ID = NUM )
                                   |
     stack:  ( L ; ID = NUM
    action:  reduce 3
```

```
11   input:  ( ID = NUM ; ID = NUM )
                                   |
     stack:  ( L ; S
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

| | |
|---|---|
| 9 | input: ( ID = NUM ; ID = NUM ) <br> stack: ( L ; ID = <br> action: shift |
| 10 | input: ( ID = NUM ; ID = NUM ) <br> stack: ( L ; ID = NUM <br> action: reduce 3 |
| 11 | input: ( ID = NUM ; ID = NUM ) <br> stack: ( L ; S <br> action: reduce 4 |
| 12 | input: ( ID = NUM ; ID = NUM ) <br> stack: ( L |

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
9        input:  ( ID = NUM ; ID = NUM )
                                   |
         stack:  ( L ; ID =
        action:  shift
```

```
10       input:  ( ID = NUM ; ID = NUM )
                                         |
         stack:  ( L ; ID = NUM
        action:  reduce 3
```

```
11       input:  ( ID = NUM ; ID = NUM )
                                         |
         stack:  ( L ; S
        action:  reduce 4
```

```
12       input:  ( ID = NUM ; ID = NUM )
                                         |
         stack:  ( L
        action:  shift
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
input: ( ID = NUM ; ID = NUM )

13
stack: ( L )
```

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
      input:  ( ID = NUM ; ID = NUM )
                                     |
13
      stack:  ( L )
     action:  reduce 2
```

---

```
      input:  ( ID = NUM ; ID = NUM )
                                     |
14
      stack:  S
```

# Example

1. A $\rightarrow$ S EOF
2. S $\rightarrow$ ( L )
3. S $\rightarrow$ id = num
4. L $\rightarrow$ L; S
5. L $\rightarrow$ S

13

```
 input: ( ID = NUM ; ID = NUM )
                                 |
 stack: ( L )
action: reduce 2
```

14

```
 input: ( ID = NUM ; ID = NUM )
                                 |
 stack: S
action: ACCEPT
```

i.e. shift (implicit) EOF

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
      input:  ( ID = NUM ; ID = NUM )
                                      |
13
      stack:  ( L )
     action:  reduce 2
```

```
      input:  ( ID = NUM ; ID = NUM )
                                      |
14
      stack:  S
     action:  ACCEPT
```

i.e. shift (implicit) EOF

Next lecture: how to build the shift-reduce DFA, ie parser table

# Example

1. A → S EOF
2. S → ( L )
3. S → id = num
4. L → L; S
5. L → S

```
      input:  ( ID = NUM ; ID = NUM )
                                      |
13
      stack:  ( L )
     action:  reduce 2
```

---

```
      input:  ( ID = NUM ; ID = NUM )
                                      |
14
      stack:  S
     action:  ACCEPT
```

i.e. shift (implicit) EOF

Next lecture: how to build the shift-reduce DFA, ie parser table

Again, duplicate entries in cells denote parse conflicts:
shift-reduce, reduce-reduce

# The Dangling Else Problem

- Valid Program: if a then if b then S1 else S2

  **1** S $\rightarrow$ *if* E *then* S *else* S
  **2** S $\rightarrow$ *if* E *then* S
  **3** S $\rightarrow$ OTHER

# The Dangling Else Problem

- Valid Program: if a then if b then S1 else S2

  **1** S $\rightarrow$ *if* E *then* S *else* S
  **2** S $\rightarrow$ *if* E *then* S
  **3** S $\rightarrow$ OTHER

Potential problem?

# The Dangling Else Problem

- Valid Program: if a then if b then S1 else S2

  **1** S → *if* E *then* S *else* S
  **2** S → *if* E *then* S
  **3** S → OTHER

- 2 interpretations: if a then [if b then S1 else S2]
  if a then [if b then S1] else S2

# The Dangling Else Problem

- Valid Program: if a then if b then S1 else S2

  **1** S → *if* E *then* S *else* S
  **2** S → *if* E *then* S
  **3** S → OTHER

- 2 interpretations: if a then [if b then S1 else S2]
  if a then [if b then S1] else S2

- Want first behavoir, but parse will report *shift-reduce conflict* when S1 is on top stack.

# The Dangling Else Problem

- Valid Program: if a then if b then S1 else S2

  **1** S $\rightarrow$ *if* E *then* S *else* S
  **2** S $\rightarrow$ *if* E *then* S
  **3** S $\rightarrow$ OTHER

- 2 interpretations: if a then [if b then S1 else S2]
  if a then [if b then S1] else S2

- Want first behavoir, but parse will report *shift-reduce conflict* when S1 is on top stack.

- Eliminate Ambiguity by modifying grammar (matched/unmatched):

  **1** S $\rightarrow$ M
  **2** S $\rightarrow$ U
  **3** M $\rightarrow$ *if* E *then* M *else* M
  **4** M $\rightarrow$ OTHER
  **5** U $\rightarrow$ *if* E *then* S
  **6** U $\rightarrow$ *if* E *then* M *else* U

# Summary

- Construction of recursive-descent parse tables

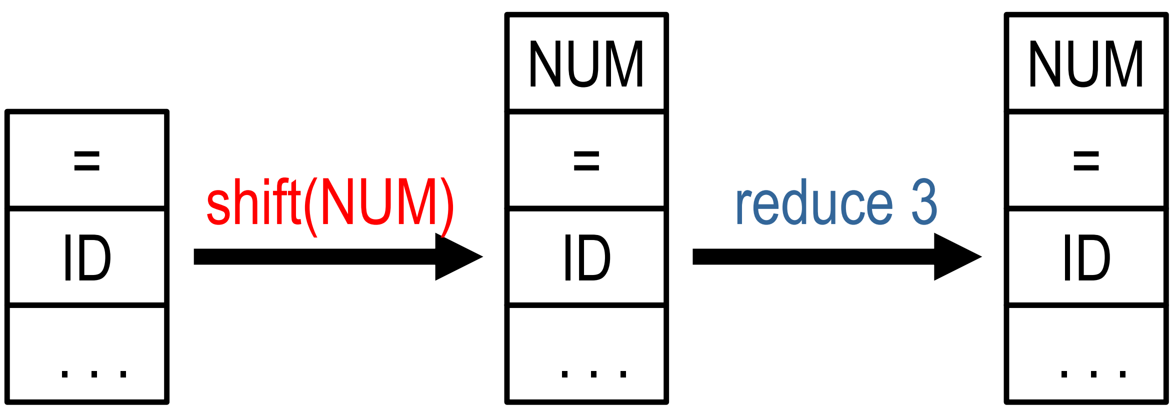
First

Nullable

Follow

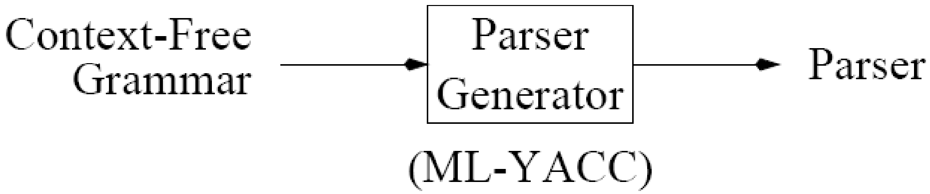
| | a | c | d |
|---|---|---|---|
| X | $X \rightarrow a$<br>$X \rightarrow Y$ | $X \rightarrow Y$ | $X \rightarrow Y$ |
| Y | $Y \rightarrow \varepsilon$ | $Y \rightarrow c$<br>$Y \rightarrow \varepsilon$ | $Y \rightarrow \varepsilon$ |
| Z | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow X\,Y\,Z$ | $Z \rightarrow d$<br>$Z \rightarrow X\,Y\,Z$ |

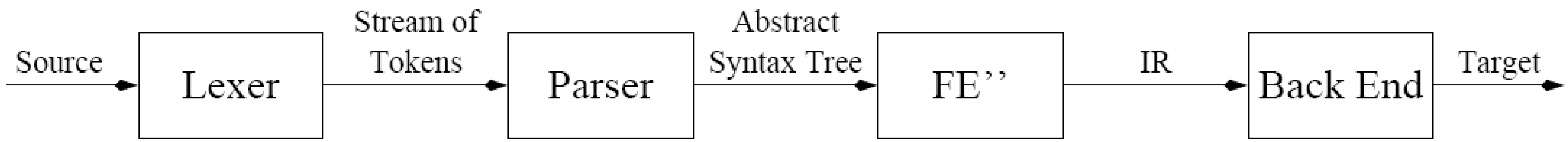
- Shift-reduce parsing overcomes requirement to make decision which rule to apply before entire RHS is seen




- "dangling-else" as typical shift-reduce conflict

# ML-YACC (Yet Another Compiler-Compiler)




- Input to **ml-yacc** is a context-free grammar specification.
- Output from **ml-yacc** is a shift-reduce parser in ML.

# CFG Specification

Specification of a parser has three parts:

**User Declarations**
%%
**ML-YACC-Definitions**
%%
**Rules**

**User declarations**: definitions of values to be used in rules

**ML-YACC Definitions**:

- definitions of terminals and non-terminals
- precedence rules that help resolve shift-reduce conflicts

**Rules**:

- production rules of grammar
- **semantic actions** associated with reductions

# ML-YACC Declarations

- Need to specify type associated with positions of tokens in input file

```
%pos int
```

- Need to specify terminal and non-terminal symbols (no symbols can be in both lists)

```
%term IF | THEN | ELSE |...
%nonterm prog | stmt | expr |...
```

- Optionally specify end-of-parse symbol - terminals which may follow start symbol

```
%eop EOF
```

- Optionally specify start symbol - otherwise, LHS non-terminal of first rule is taken as start symbol

```
%start prog
```

# Attribute Grammar

- nonterminal and terminal symbols are associated with “attribute values”
- for rule A ⟶ γ, synthesize attribute for A from attribute for the symbols in γ
  - can be used to do simple calculation inside parser:

terminal symbol of grammar

ML’s infix operator int -> int -> int

exp ⟶ exp PLUS exp (exp1 + exp2)

Semantic action:
synthesize attribute value

  - requires type-correctness of synthesized attributes w.r.t. ML operators like +, based on association of types to symbols
- ML-YACC’s result of successful parse: attribute synthesized for start symbol (unit value if no attribute given)
- other use: abstract syntax tree (topic of future lecture)

# Rules

$$symbol_0 \ : \ symbol_1 \ symbol_2 \ \ldots \ symbol_n \ (\textit{semantic\_action})$$

- Semantic action typically builds piece of AST corresponding to derived string
- Can access attribute/value of RHS symbol X using X<n>, where n specifies a particular occurrence of X on RHS.

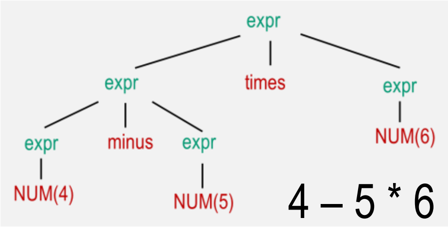
```
%term     PLUS | MINUS | NUM of int | ...
%nonterm exp of int | ...

exp: exp PLUS exp       (exp1 + exp2)
|    exp MINUS exp      (exp1 - exp2)
|    NUM                (NUM)
```

- Type of value computed by semantic action must match type of value associated with LHS non-terminal.

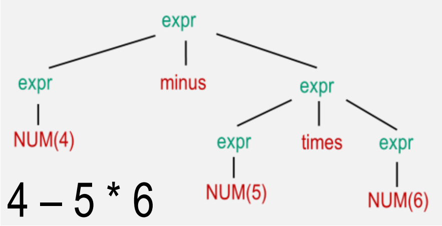
# ML-YACC and Ambiguous Grammars

Remember: grammar is ambiguous if there are strings with > 1 parse trees

**Example**: expr ⟶ expr – expr
expr ⟶ expr * expr …




vs.




Shift-reduce conflicts:

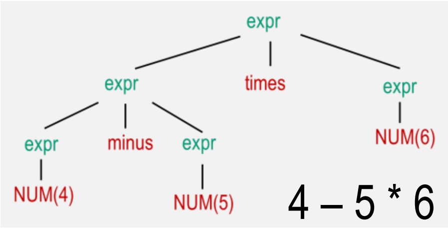
when parsing 4 – 5 * 6: eventually,

- top element of stack is E – E
- TIMES is current symbol
- can reduce using rule E → E – E, or can shift TIMES.
- shift preferred!

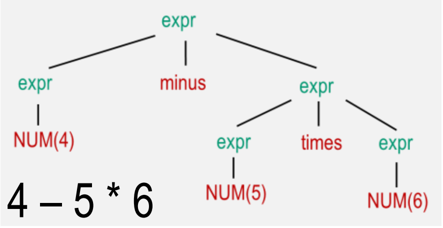
# ML-YACC and Ambiguous Grammars

Remember: grammar is ambiguous if there are strings with > 1 parse trees

**Example**: expr ➡ expr – expr
expr ➡ expr * expr …




vs.




Shift-reduce conflicts:

when parsing 4 – 5 * 6: eventually,
- top element of stack is E – E
- TIMES is current symbol
- can reduce using rule E → E – E, or can shift TIMES.
- shift preferred!

when parsing 4 – 5 – 6: eventually,
- top element of stack is E – E
- MINUS is current symbol
- can reduce using rule E → E – E, or can shift MINUS.
- reduce preferred!

Also: reduce-reduce conflicts if multiple rules can be reduced

ML-YACC warns about conflicts, and applies default choice (see below)

# Directives

**Three Solutions:**

1. Let YACC complain, but demonstrate that its choice (to shift) was correct.
2. Rewrite grammar to eliminate ambiguity.
3. Keep grammar, but add *precedence directives* which enable conflicts to be resolved. Use `%left`, `%right`, `%nonassoc`

   - For this grammar:
     ```
     %left  PLUS  MINUS
     %left  MULT  DIV
     ```
   - PLUS, MINUS are left associative, bind equally tightly
   - MULT, DIV are left associative, bind equally tightly
   - MULT, DIV bind tighter than PLUS, MINUS

# Directives

- Given directives, ML-YACC assigns precedence to each *terminal* and *rule*
  - Precedence of terminal based on order in which associativity specified
  - Precedence of rule is the precedence of right-most terminal. For example, precedence($E \rightarrow E + E$) = precedence(PLUS).
- Given shift-reduce conflict, ML-YACC performs the following:
  1. Find precedence of rule to be reduced, terminal to be shifted.
  2. prec(terminal) $>$ prec(rule) $\Rightarrow$ shift.
  3. prec(rule) $>$ prec(terminal) $\Rightarrow$ reduce.
  4. prec(terminal) $=$ prec(rule), then:
     - assoc(terminal) $=$ left $\Rightarrow$ reduce.
     - assoc(terminal) $=$ right $\Rightarrow$ shift.
     - assoc(terminal) $=$ nonassoc $\Rightarrow$ report as error.

# Precedence Examples

**1**

```
 input: 4 + 5 * 6
              |
 stack: 4 + 5
action: prec(*) > prec(+) -> shift
```

---

**2**

```
 input: 4 * 5 + 6
              |
 stack: 4 * 5
action: prec(*) > prec(+) -> reduce
```

---

**3**

```
 input: 4 + 5 + 6
              |
 stack: 4 + 5
action: assoc(+) = left -> reduce
```

# Default Behavior

**What if directives not specified?**

- shift-reduce: report error, *shift* by default.
- reduce-reduce: report error, reduce by rule that occurs first.

**What to do:**

- shift-reduce: acceptable in well defined cases (dangling else).
- reduce-reduce: unacceptable. Rewrite grammar.

# Direct Rule Precedence Specification

**Can assign *specific* precedence to rule, rather than precedence of last terminal.**

- Use the %prec directive.
- Commonly used for the *unary minus* problem.

```
%left PLUS MINUS
%left MULT DIV
```

- Consider $-4 * 6$, `MINUS NUM(4) MULT NUM(6)`
- We perfer to bind left unary minus ("-") tighter. Here, precedence of MINUS is lower than MULT, so we get $-(4 * 6)$, not $(-4) * 6$.
- Solution:

```
%left PLUS MINUS
%left MULT DIV
%left UMINUS

exp : MINUS expr %prec UMINUS ()
    | expr PLUS expr ()...
```

# Syntax vs. Semantics

**Consider language with two classes of expressions**

- *Arithmetic* expressions (ae)

```
ae  :  ae PLUS ae  ()
    |  ID          ()
```

- *Boolean* expressions (be)

```
be  :  be AND be  ()
    |  be OR be   ()
    |  be EQ be   ()
    |  ID         ()
```

- Consider: `a := b`, ID(a) ASSIGN ID(b):
  - Reduce-reduce conflict - parser can't choose between be $\rightarrow$ ID or ae $\rightarrow$ ID.
  - For now ae and be should be aliased - let semantic analysis (next phase) determine that `a & b + c` is a type error.
  - Type checking cannot be done easily in context free grammars.

# Constructing LR parsing tables

- LR(0)
- SLR
- LR(1)
- LALR(1)

# Shift-Reduce, Bottom Up, LR(1) Parsing

- Shift-reduce parsing can parse more grammars than predictive parsing.
- *Shift-reduce parsing* has stack and input.
- Based on stack contents and next input token, one of two action performed:
  1. *Shift* - push next input token onto top of stack.
  2. *Reduce* - choose production ($X \rightarrow ABC$); pop off RHS (C, B, A); push LHS (X).
- If $ is shifted, then input stream has been parsed successfully.

# LR(k)

Can generalize to case where parser makes decision based on stack contents and next $k$ tokens. LR(k):

- Left-to-right parse
- right-most derivation
- $k$-symbol lookahead

LR($k$) parsing, $k > 1$, rarely used in compilation:

- DFA too large: need transition for every sequence of $k$ terminals.
- Most programming languages can be described by LR(1) grammars.

# Shift Reduce Parsing DFA

Parser uses DFA to make shift/reduce decisions:

- Each state corresponds to contents of stack at some point in time.
- Edges labeled with terminals/non-terminals.

Rather than scanning entire stack to determine current DFA state, parser can remember state reached for each stack element.

- Transition table for LR(1) or LR(0) DFA:

| | Terminals $(T_1, T_2, \ldots, T_n)$ | Non-Terminals $(N_1, N_2, \ldots, N_n)$ |
|---|---|---|
| 1 | *actions* | *actions* |
| 2 | sn $\longrightarrow$ shift n | gz $\longrightarrow$ goto z |
| 3 | rk $\longrightarrow$ reduce k | |
| $\vdots$ | a $\longrightarrow$ accept | |
| n | $\longrightarrow$ error | |

# Parsing Algorithm

Look up DFA state on top of stack, next terminal in input:

- shift($n$):
  - Advance input by one.
  - Push input token on stack with $n$ (the new state).
- reduce($k$):
  - Pop stack as many times as number of symbols on RHS of rule $k$.
  - Let $X$ be LHS of rule $k$
  - In state now on top of stack, look up X to get goto($z$)
  - Push $X$ on stack with $z$ (the new state).
- accept $\longrightarrow$ stop, report success.
- error $\longrightarrow$ stop, report syntax error.

To understand LR($k$) parsing, first focus on LR(0) parser construction using an example.

# LR(0) Parsing

**Grammar 3.20 (book):**

| | | |
|---|---|---|
| 1 $S' \rightarrow S\ \$$ | 3 $S \rightarrow \mathrm{x}$ | 5 $L \rightarrow L, S$ |
| 2 $S \rightarrow (\ L\ )$ | 4 $L \rightarrow S$ | |

Initially, stack empty, input contains '$S$' string followed by a '\$':

1
$$\boxed{\begin{array}{l} S' \rightarrow .S\$ \\ S \rightarrow .(\ L\ ) \\ S \rightarrow .\mathrm{x} \end{array}}$$

- Combination of production and '.' called LR(0) *item*.
- '.' specifies parser position.
- Three items represent *closure* of: $S' \rightarrow .S\ \$$
- Closure adds more items to a set when dot exists to left of a non-terminal.

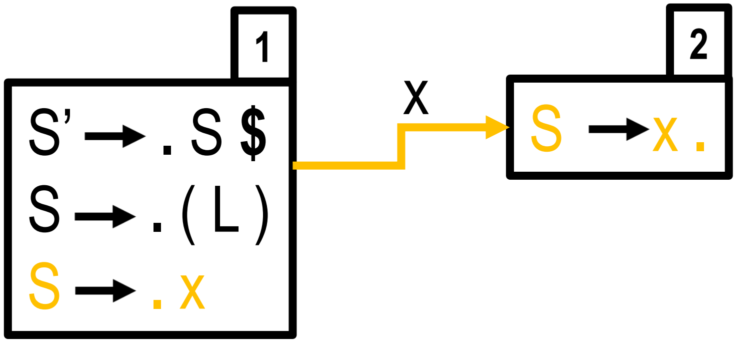
1 $\boxed{\ldots}$ represents the initial parser state. Develop other states step by step, by considering all possible transitions:

1. move cursor past x
2. move cursor past (
3. move cursor past S

# LR(0) States

S’ → S $    S → x    L → L , S

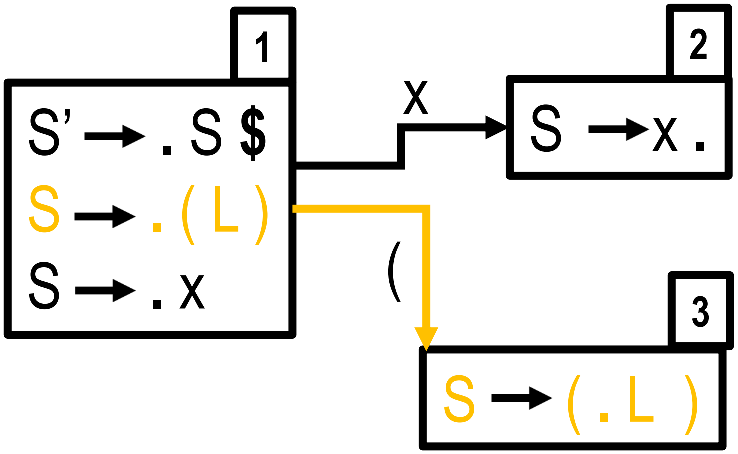
S → ( L )    L → S




Moving the dot past a terminal **t** represents “shift **t**”.

# LR(0) States

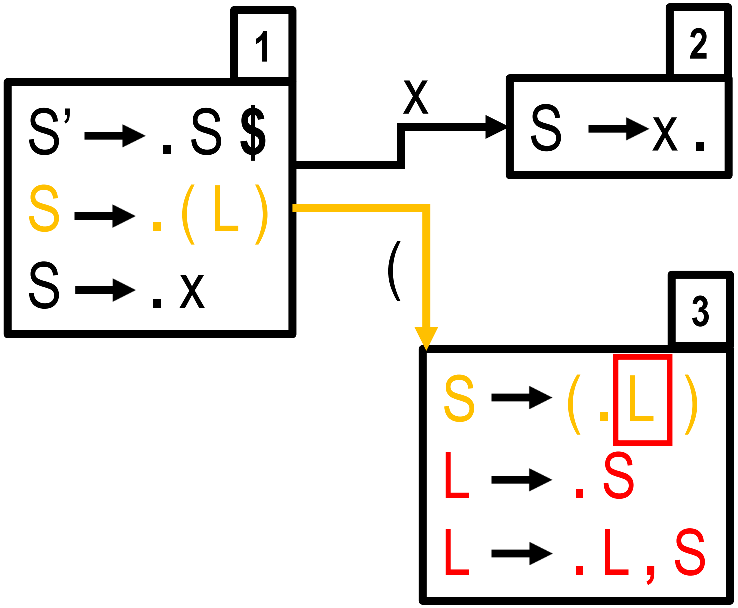
S' → S $    S → x    L → L , S
S → ( L )    L → S




Moving the dot past a terminal **t** represents “shift **t**”.

# LR(0) States

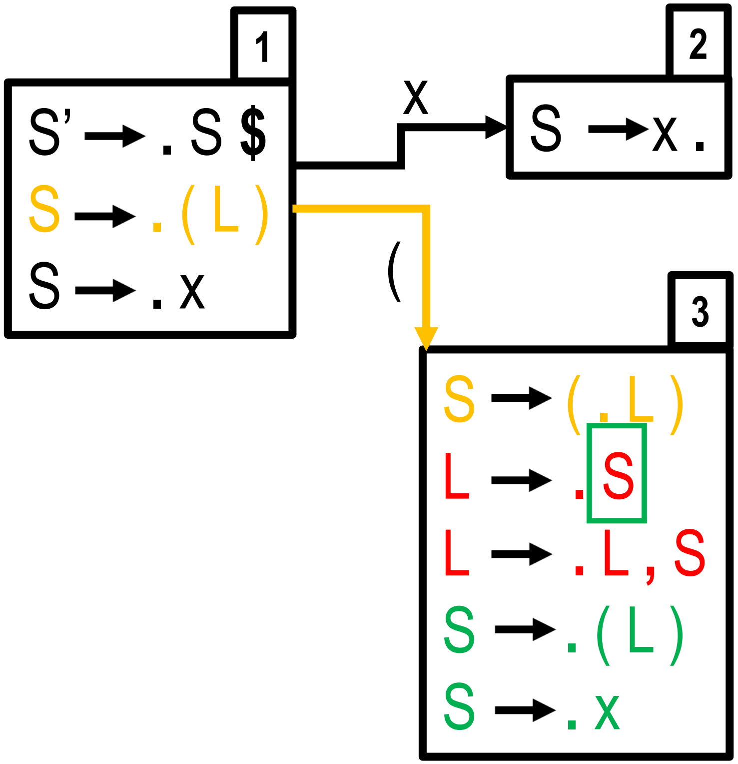
S' → S $ S → x L → L , S
S → ( L ) L → S




"shift (". Don't forget the closure!

# LR(0) States

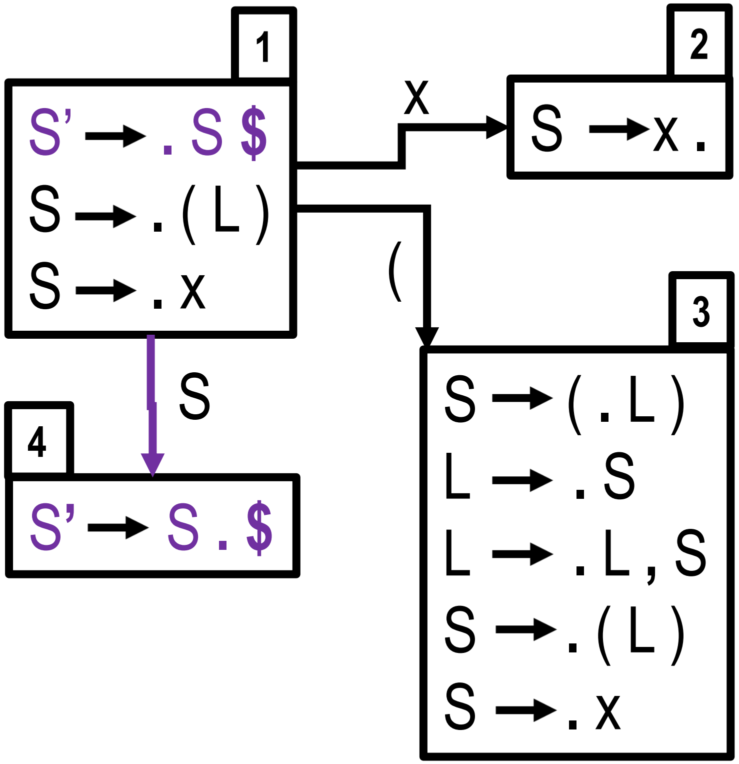
S' → S $     S → x     L → L , S
S → ( L )     L → S




"shift (". Don't forget the closure! Yes, CLOSURE!

# LR(0) States

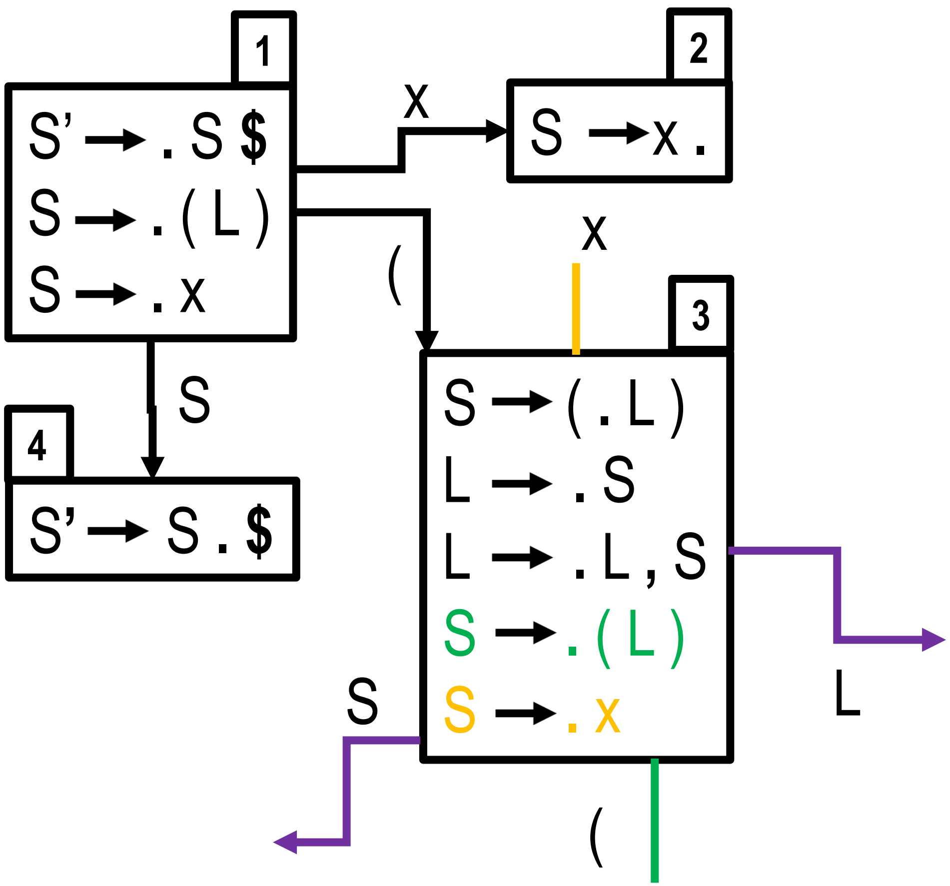
S' → S $ S → x L → L , S
S → ( L ) L → S




Moving the dot past a nonterminal **N** represents "goto" after a reduction for a rule with LHS **N** has been performed.

# LR(0) States

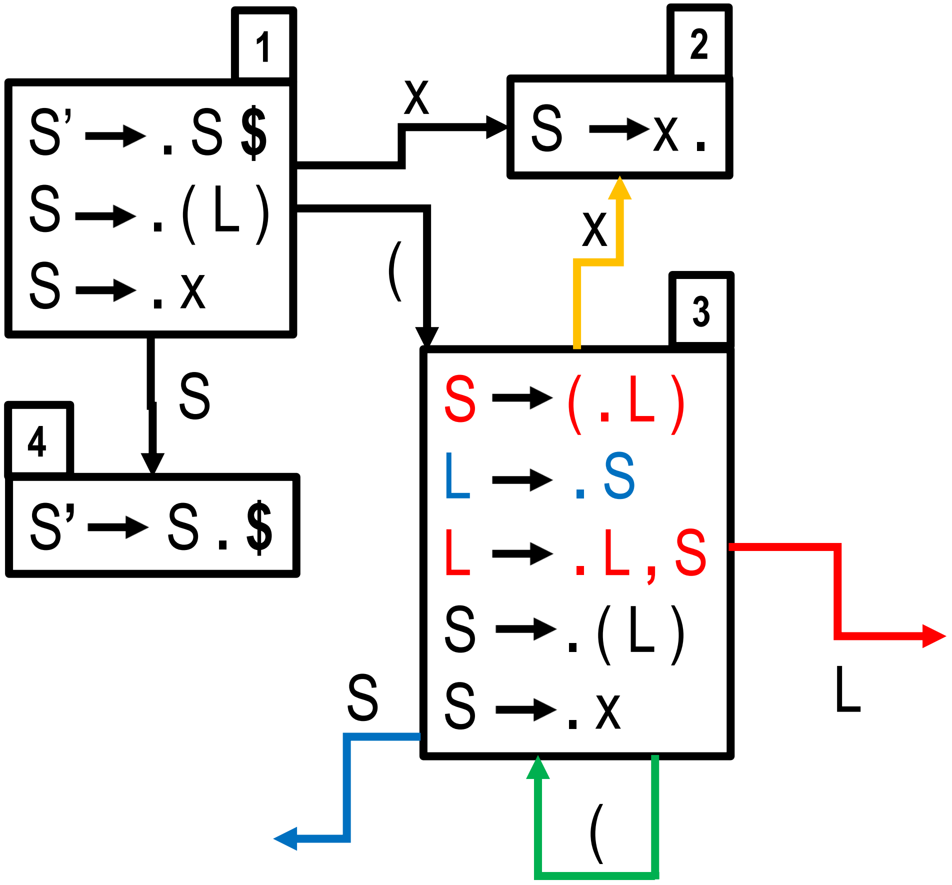
| | | |
|---|---|---|
| S’ → S $ | S → x | L → L , S |
| S → ( L ) | L → S | |




In $\boxed{3}$ , we have four transitions to consider. Let’s start with the terminals…

# LR(0) States

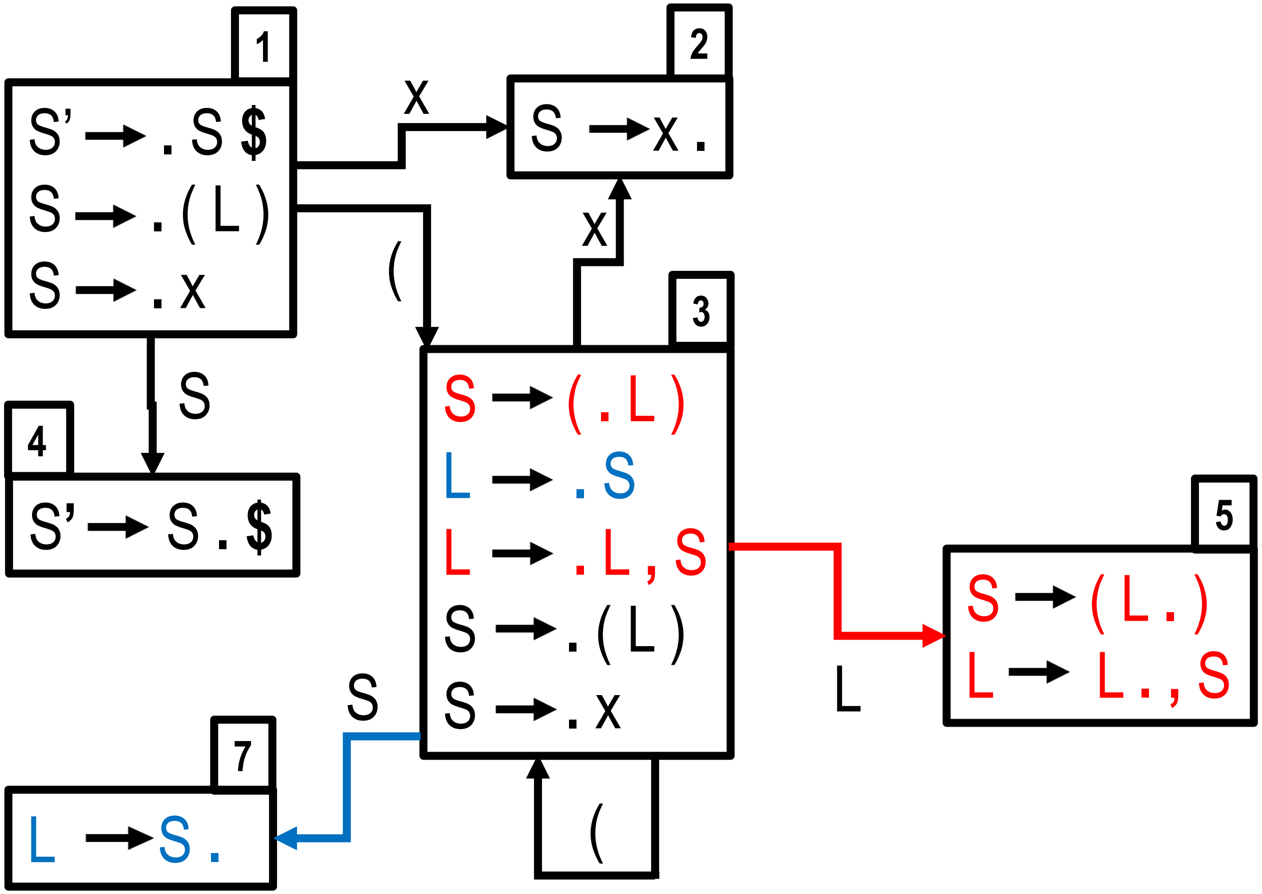
S' → S $  S → x  L → L , S
S → ( L )  L → S




Next, nonterminals S and L.

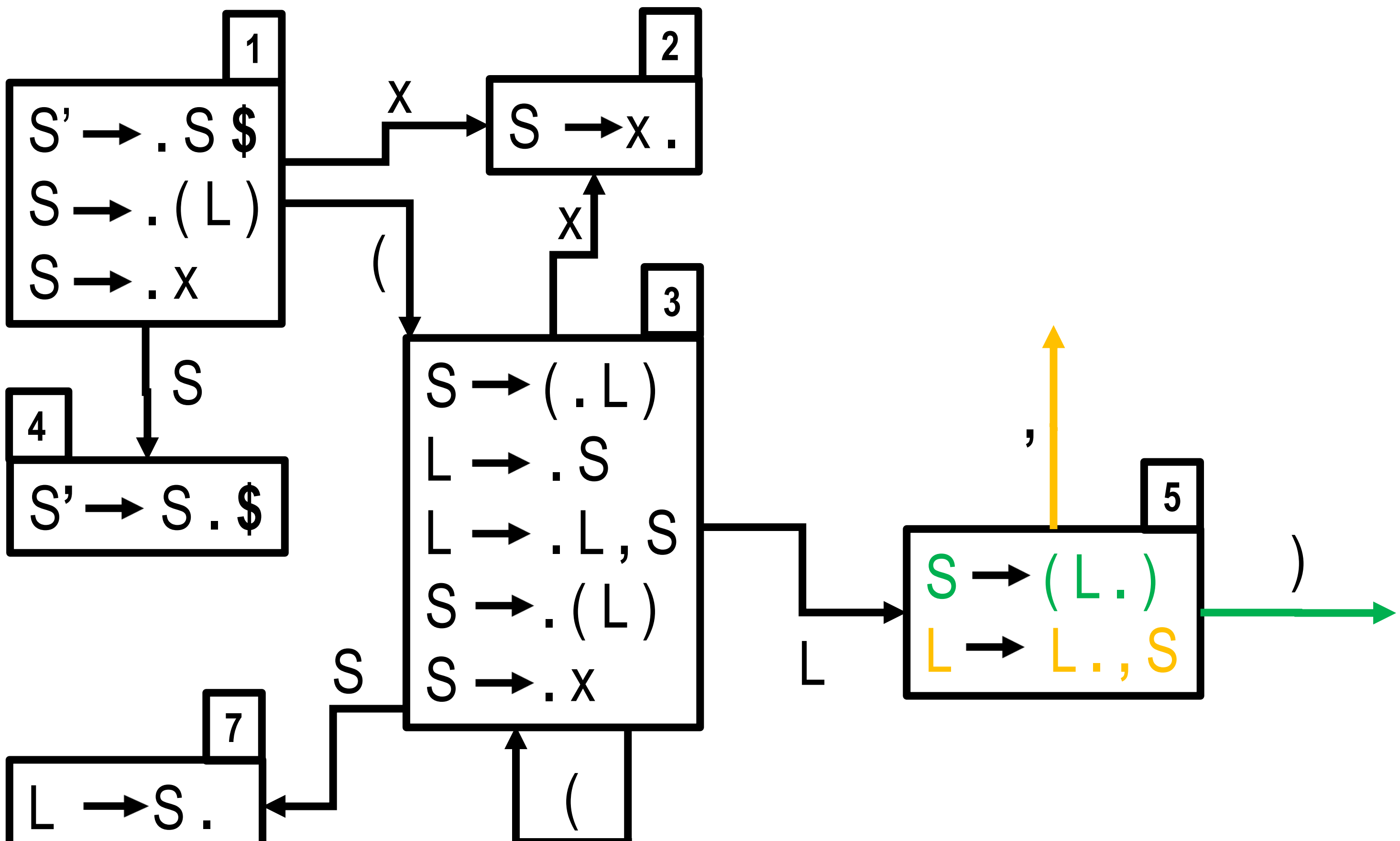
# LR(0) States

| | | |
|---|---|---|
| S' → S $ | S → x | L → L , S |
| S → ( L ) | L → S | |



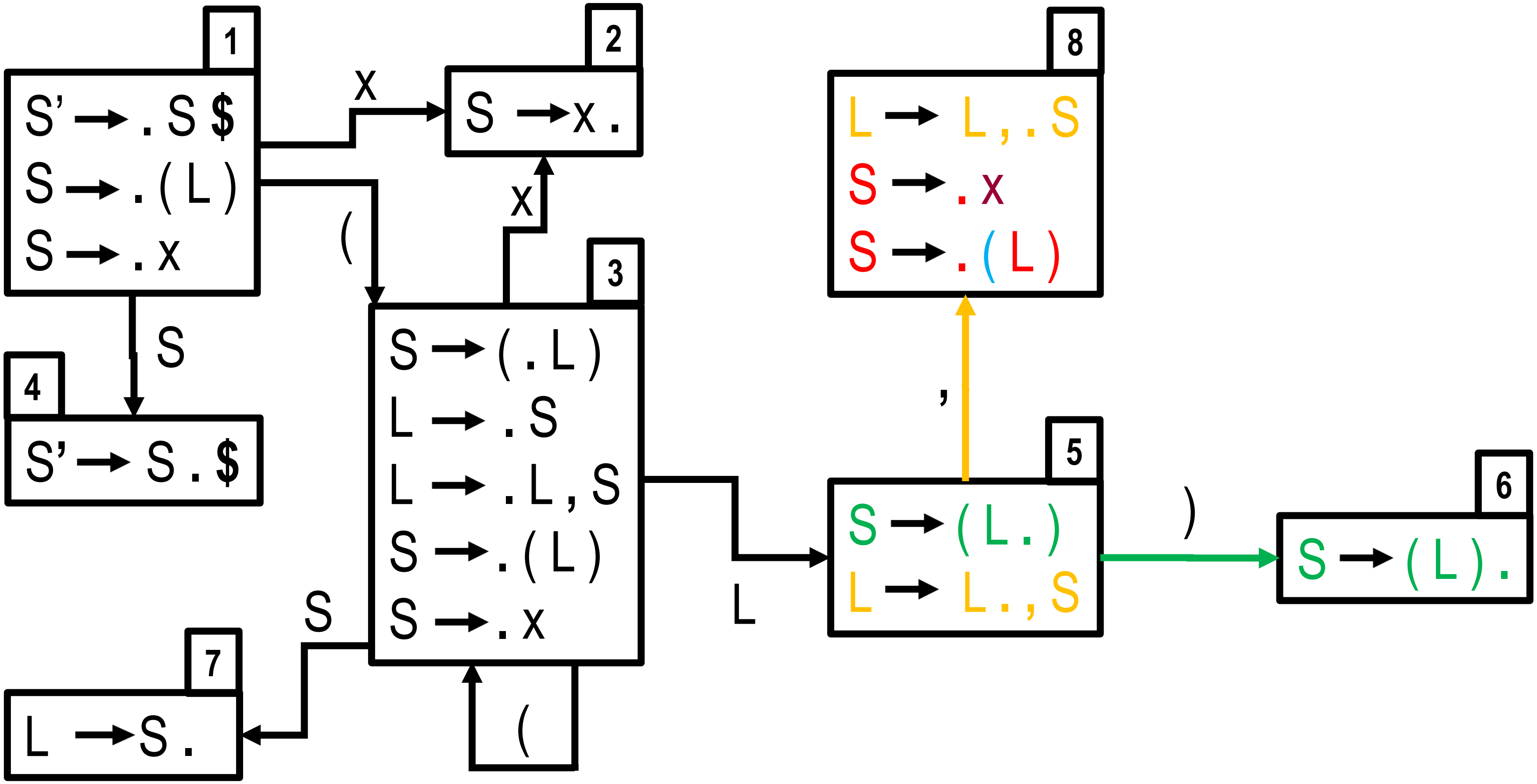
# LR(0) States

| | | |
|---|---|---|
| S' → S $ | S → x | L → L , S |
| S → ( L ) | L → S | |



# LR(0) States

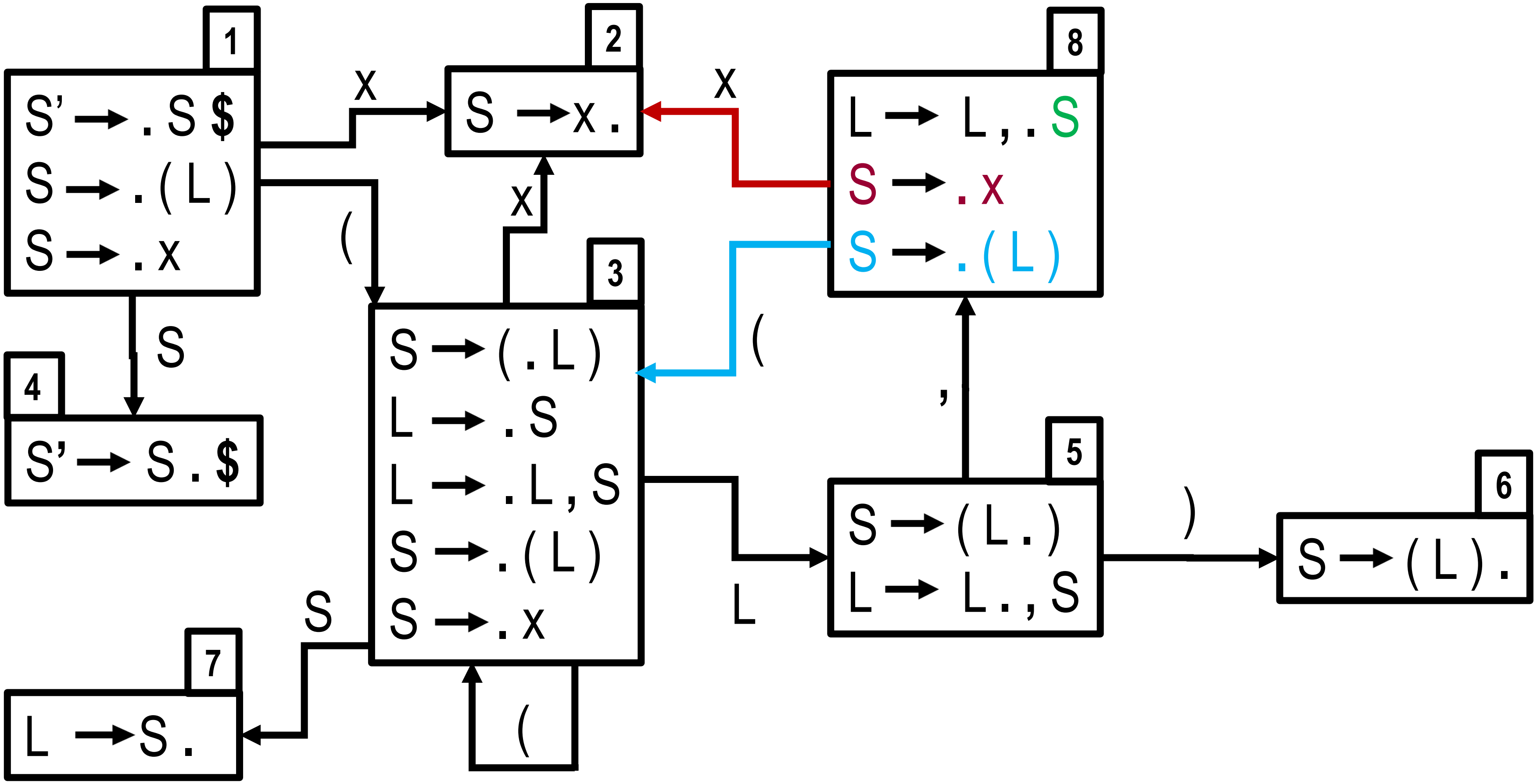
S' → S $  S → x  L → L , S
S → ( L )  L → S




Closure! In $\boxed{8}$, we have three transitions to consider. Terminals **x** and (…

# LR(0) States

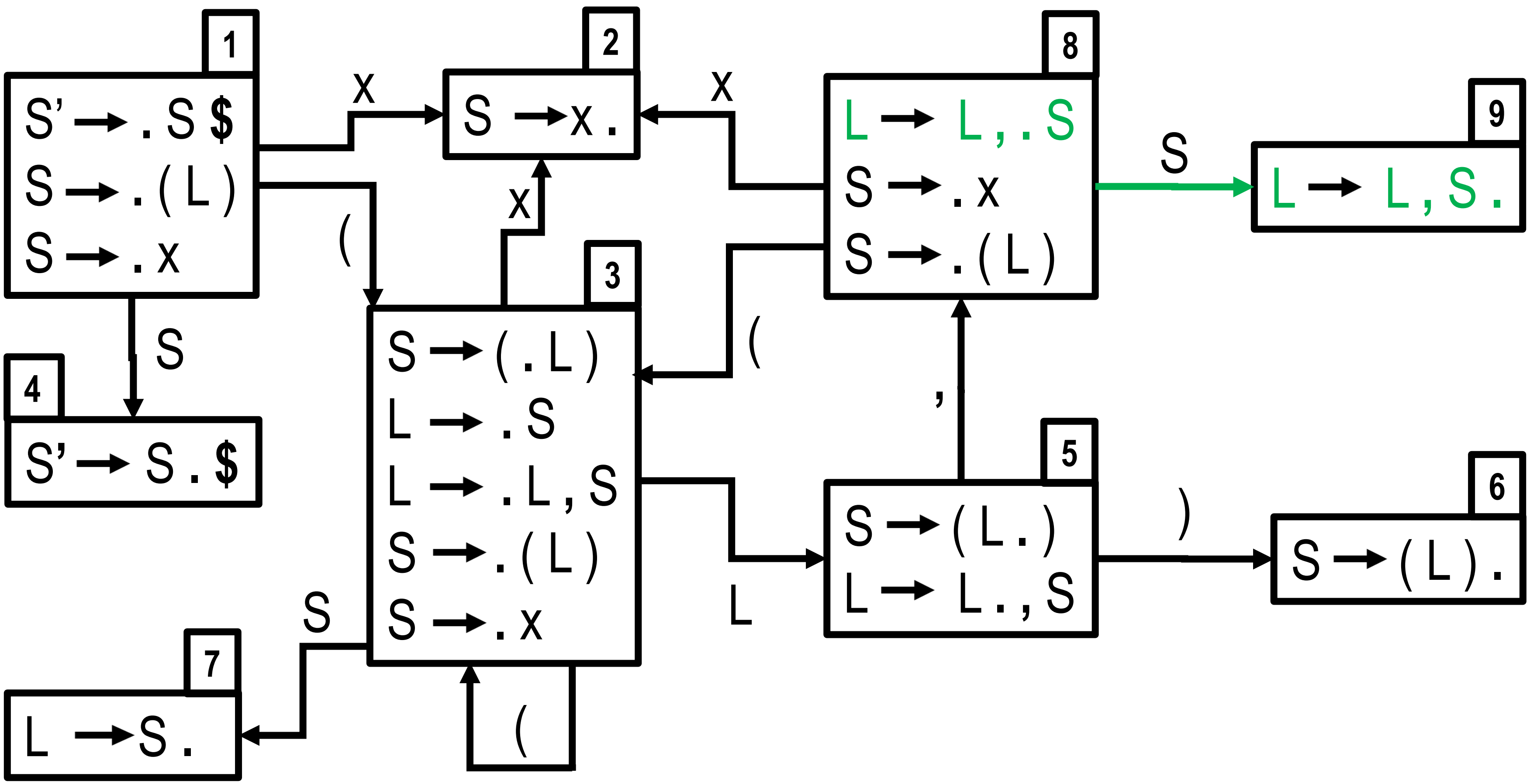
S' → S $    S → x    L → L , S
S → ( L )    L → S




and nonterminal S...

# LR(0) States

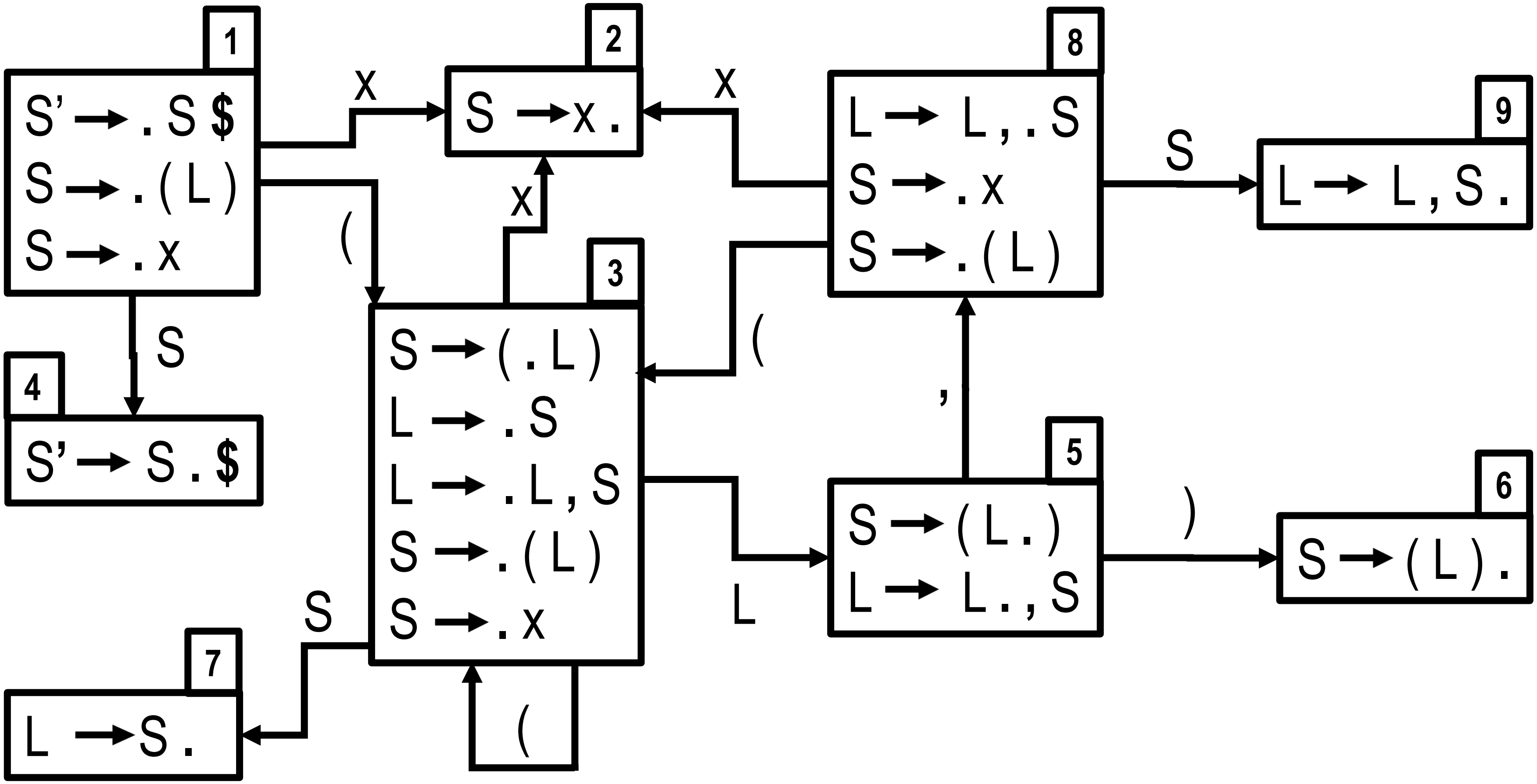
S' → S $    S → x    L → L , S
S → ( L )    L → S



# LR(0) States

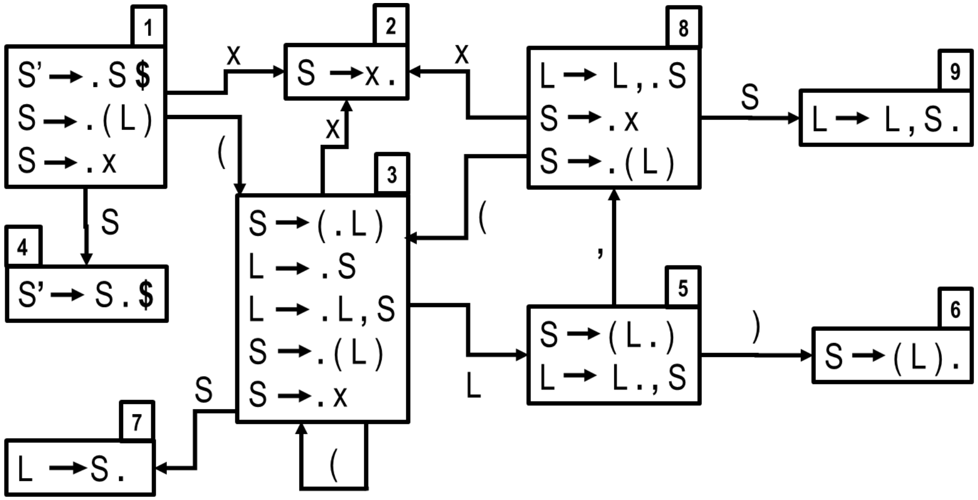
$S' \rightarrow S\ \$$ $\quad S \rightarrow x \quad L \rightarrow L , S$

$S \rightarrow ( L ) \quad L \rightarrow S$




Done. How can we extract the parsing table?

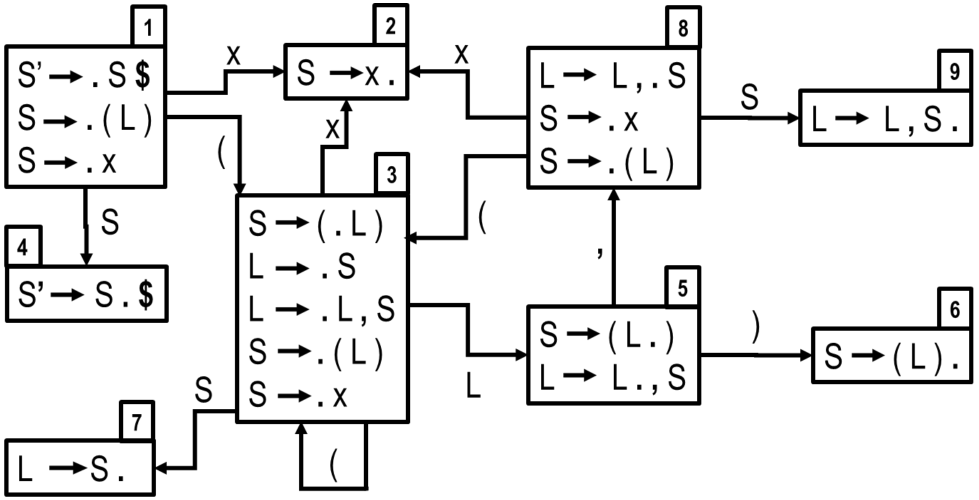
# LR(0) parsing table




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |

# LR(0) parsing table

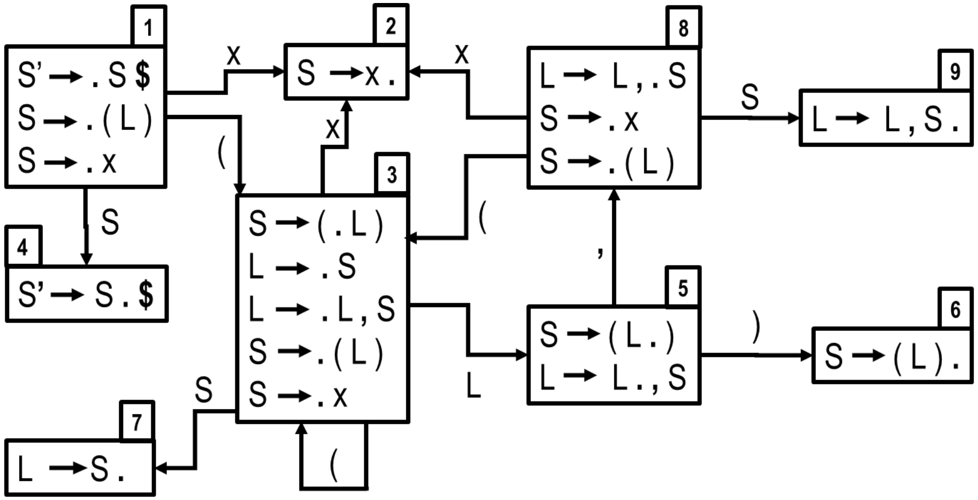
For each transition $X \xrightarrow{t} Y$ where **t** is a terminal, add "shift-to Y" in cell (X, t).




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |

# LR(0) parsing table

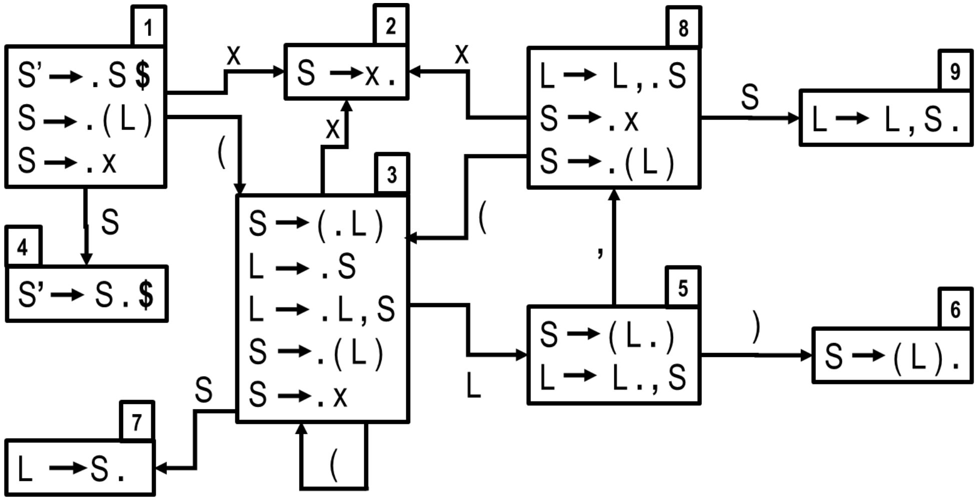
For each transition X $\xrightarrow{t}$ Y where t is a terminal, add "shift-to Y" in cell (X, t).




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | s3 | | s2 | | | | |
| 2 | | | | | | | |
| 3 | s3 | | s2 | | | | |
| 4 | | | | | | | |
| 5 | | s6 | | s8 | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | s3 | s2 | | | | |
| 9 | | | | | | | |

# LR(0) parsing table

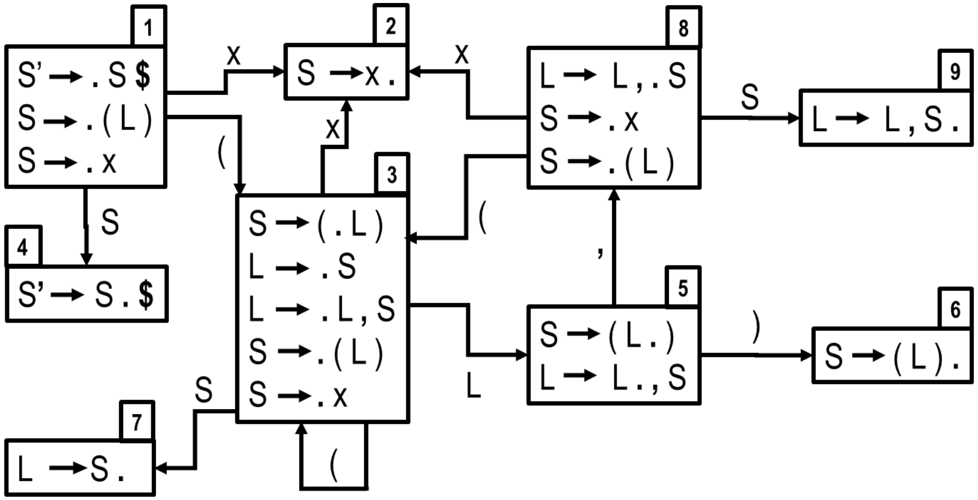
For each transition X $\xrightarrow{N}$ Y where N is a nonterminal, add "goto Y" in cell (X, N).




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | | |
| 2 | | | | | | | |
| 3 | **s3** | | **s2** | | | | |
| 4 | | | | | | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | **s3** | **s2** | | | | |
| 9 | | | | | | | |

# LR(0) parsing table

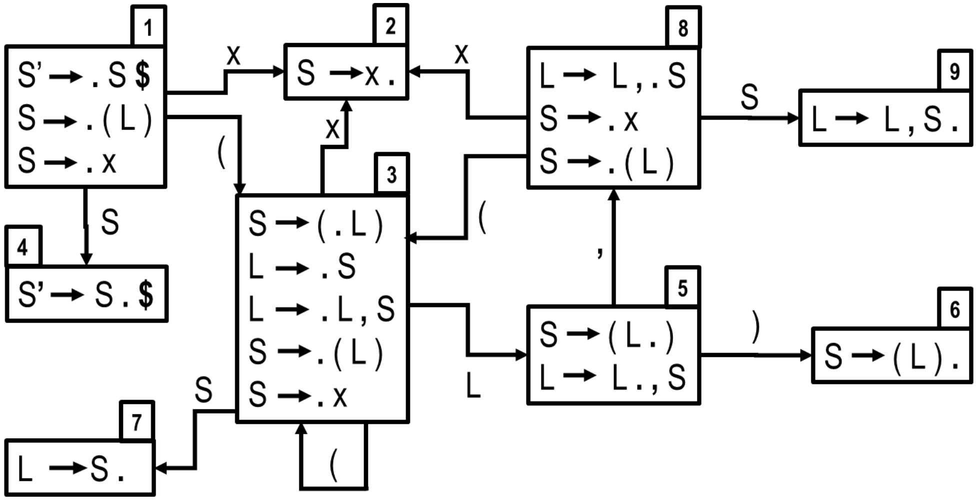
For each transition X $\xrightarrow{N}$ Y where N is a nonterminal, add "goto Y" in cell (X, N).




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | s3 | | s2 | | | g4 | |
| 2 | | | | | | | |
| 3 | s3 | | s2 | | | g7 | g5 |
| 4 | | | | | | | |
| 5 | | s6 | | s8 | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | s3 | s2 | | | g9 | |
| 9 | | | | | | | |

# LR(0) parsing table

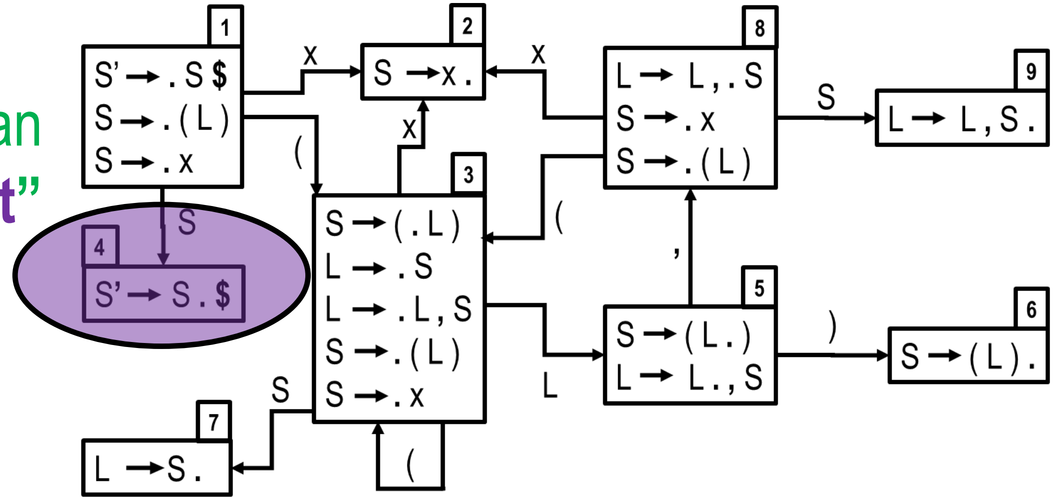
For each state **X** containing an item **S' ⟶ S . $** put "**accept**" in cell (**X**, **$**).




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | | | | | | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | | | | | | | |

# LR(0) parsing table

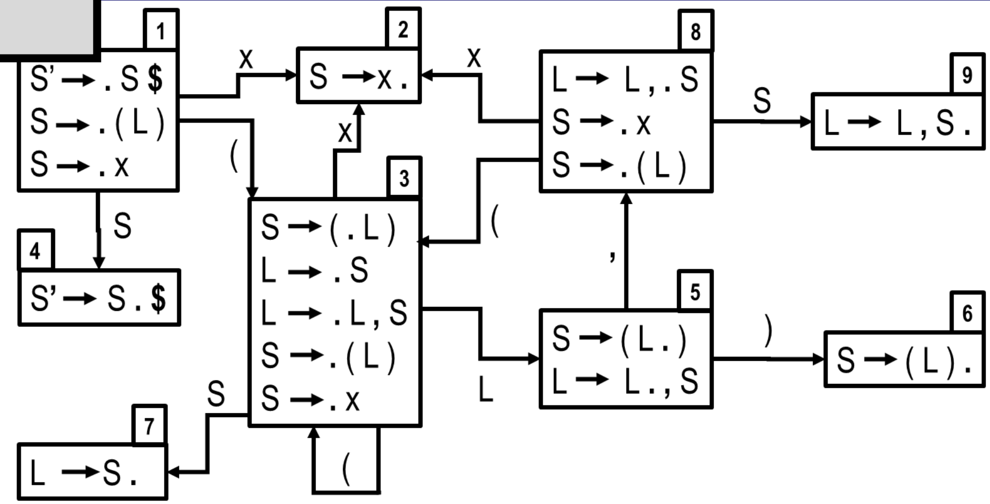
For each state **X** containing an item **S' → S . $** put "**accept**" in cell (**X**, **$**).




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | | | | | | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| **4** | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | | | | | | | |

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x$ $\quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L )$ $\quad L \xrightarrow{3} S$



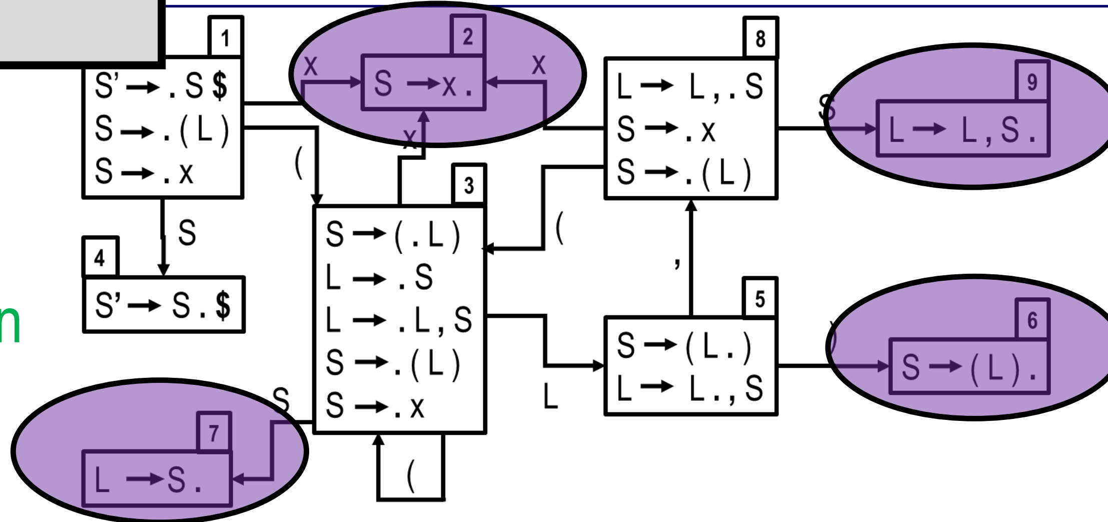

For each state **X** containing an item $N \xrightarrow{n} \gamma$ **.** (dot at end of item for rule n) put "**reduce n**" in all cells (**X**, **t**).

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | | | | | | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | | | | | | | |

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L ) \quad L \xrightarrow{3} S$

For each state **X** containing an item $N \xrightarrow{n} \gamma$ . (dot at end of item for rule n) put "**reduce n**" in all cells (**X**, **t**).




| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | s3 | | s2 | | | g4 | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s3 | | s2 | | | g7 | g5 |
| 4 | | | | | accept | | |
| 5 | | s6 | | s8 | | | |
| 6 | r1 | r1 | r1 | r1 | r1 | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | | s3 | s2 | | | g9 | |
| 9 | r4 | r4 | r4 | r4 | r4 | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x$ $\quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L )$ $\quad L \xrightarrow{3} S$

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x$ $\quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L )$ $\quad L \xrightarrow{3} S$

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

# LR(0) parsing table

$$S' \xrightarrow{0} S\ \$ \qquad S \xrightarrow{2} x \qquad L \xrightarrow{4} L\ ,\ S$$
$$S \xrightarrow{1} (\ L\ ) \qquad L \xrightarrow{3} S$$

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | s3 | | s2 | | | g4 | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s3 | | s2 | | | g7 | g5 |
| 4 | | | | | accept | | |
| 5 | | s6 | | s8 | | | |
| 6 | r1 | r1 | r1 | r1 | r1 | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | | s3 | s2 | | | g9 | |
| 9 | r4 | r4 | r4 | r4 | r4 | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad$ $S \xrightarrow{2} x$ $\quad$ $L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L )$ $\quad$ $L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S? | , x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# LR(0) parsing table

$$S' \xrightarrow{0} S\ \$ \qquad S \xrightarrow{2} x \qquad L \xrightarrow{4} L\ ,\ S$$
$$S \xrightarrow{1} (\ L\ ) \qquad L \xrightarrow{3} S$$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L\ ,\ S$

$S \xrightarrow{1} (\ L\ ) \quad L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L? | , x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L ) \quad L \xrightarrow{3} S$

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | s3 | | s2 | | | g4 | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s3 | | s2 | | | g7 | g5 |
| 4 | | | | | accept | | |
| 5 | | s6 | | s8 | | | |
| 6 | r1 | r1 | r1 | r1 | r1 | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | | s3 | s2 | | | g9 | |
| 9 | r4 | r4 | r4 | r4 | r4 | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L ) \quad L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | |
| | | |
| | | |
| | | |
| | | |
| | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x$ $\quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L )$ $\quad L \xrightarrow{3} S$

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | |
| | | |
| | | |
| | | |
| | | |

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L\ ,\ S$

$S \xrightarrow{1} (\ L\ ) \quad L \xrightarrow{3} S$

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S? | ) $ | |
| | | |
| | | |
| | | |

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $S \xrightarrow{2} x$ $L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L )$ $L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S9 | ) $ | |
| | | |
| | | |
| | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x$ $\quad L \xrightarrow{4} L\ ,\ S$

$S \xrightarrow{1} (\ L\ )$ $\quad L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S9 | ) $ | Reduce 4 |
| 1 (3 L? | ) $ | |
| | | |
| | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L\ ,\ S$

$S \xrightarrow{1} (\ L\ )$ $\quad L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S9 | ) $ | Reduce 4 |
| 1 (3 L5 | ) $ | |
| | | |
| | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L\ ,\ S$

$S \xrightarrow{1} (\ L\ ) \quad L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S9 | ) $ | Reduce 4 |
| 1 (3 L5 | ) $ | Shift 6 |
| 1 (3 L5 )6 | $ | |
| | | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L\ ,\ S$

$S \xrightarrow{1} (\ L\ ) \quad L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S9 | ) $ | Reduce 4 |
| 1 (3 L5 | ) $ | Shift 6 |
| 1 (3 L5 )6 | $ | Reduce 1 |
| 1 S? | $ | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x \quad L \xrightarrow{4} L , S$

$S \xrightarrow{1} ( L ) \quad L \xrightarrow{3} S$

How to use the table: parse (x, x)

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S9 | ) $ | Reduce 4 |
| 1 (3 L5 | ) $ | Shift 6 |
| 1 (3 L5 )6 | $ | Reduce 1 |
| 1 S4 | $ | |

# LR(0) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} x$ $\quad L \xrightarrow{4} L\ ,\ S$

$S \xrightarrow{1} (\ L\ )$ $\quad L \xrightarrow{3} S$

No cell with >1 action => LR(0)

| | ( | ) | x | , | $ | S | L |
|---|---|---|---|---|---|---|---|
| 1 | **s3** | | **s2** | | | **g4** | |
| 2 | **r2** | **r2** | **r2** | **r2** | **r2** | | |
| 3 | **s3** | | **s2** | | | **g7** | **g5** |
| 4 | | | | | **accept** | | |
| 5 | | **s6** | | **s8** | | | |
| 6 | **r1** | **r1** | **r1** | **r1** | **r1** | | |
| 7 | **r3** | **r3** | **r3** | **r3** | **r3** | | |
| 8 | | **s3** | **s2** | | | **g9** | |
| 9 | **r4** | **r4** | **r4** | **r4** | **r4** | | |

How to use the table: parse (x, x)

| Stack | Input | Action |
|---|---|---|
| 1 | (x, x) $ | Shift 3 |
| 1 (3 | x, x) $ | Shift 2 |
| 1 (3 x2 | , x) $ | Reduce 2 |
| 1 (3 S7 | , x) $ | Reduce 3 |
| 1 (3 L5 | , x) $ | Shift 8 |
| 1 (3 L5 ,8 | x) $ | Shift 2 |
| 1 (3 L5 ,8 x2 | ) $ | Reduce 2 |
| 1 (3 L5 ,8 S9 | ) $ | Reduce 4 |
| 1 (3 L5 | ) $ | Shift 6 |
| 1 (3 L5 )6 | $ | Reduce 1 |
| 1 S4 | $ | **Accept** |

# Another Example (3.23 in book)

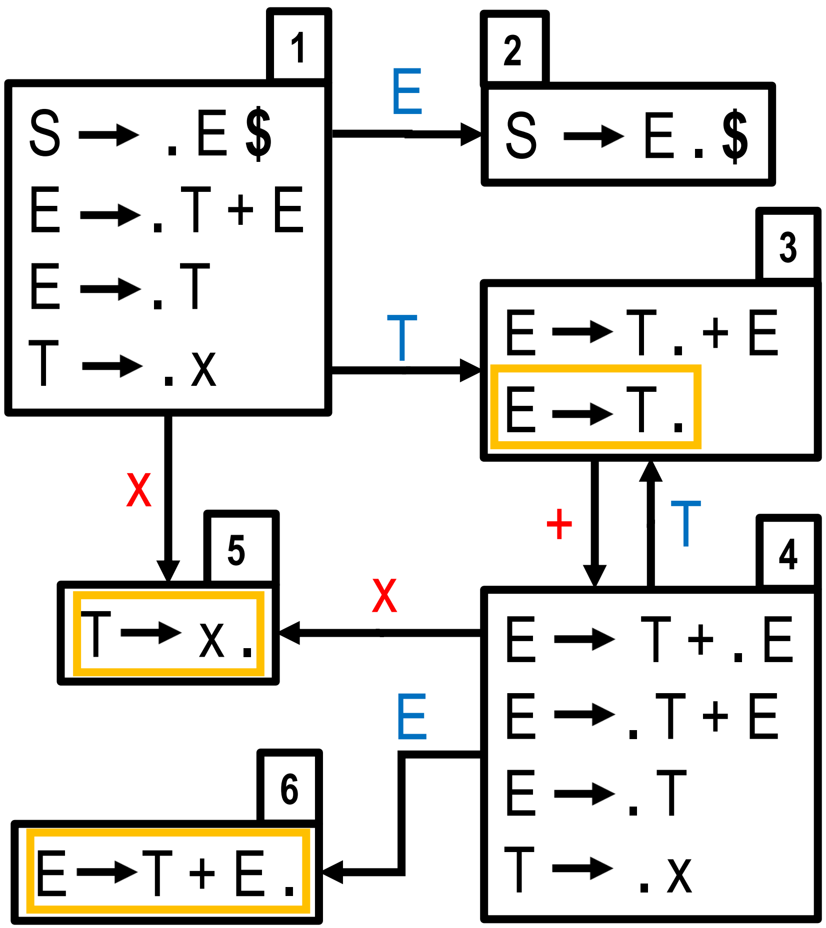
Do on blackboard

| | |
|---|---|
| $S \xrightarrow{0} E\ \$$ | $E \xrightarrow{2} T$ |
| $E \xrightarrow{1} T + E$ | $T \xrightarrow{3} x$ |

# Another Example (3.23 in book)

$S \xrightarrow{0} E\ \$$ $\quad E \xrightarrow{2} T$

$E \xrightarrow{1} T + E$ $\quad T \xrightarrow{3} x$




| | x | + | $ | E | T |
|---|---|---|---|---|---|
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | r2 | s4, r2 | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | r3 | r3 | r3 | | |
| 6 | r1 | r1 | r1 | | |

**Duplicate entry – grammar is not LR(0)!**

# Another Example - SLR

| | x | + | $ | E | T |
|---|---|---|---|---|---|
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | r2 | s4, r2 | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | r3 | r3 | r3 | | |
| 6 | r1 | r1 | r1 | | |

Can make grammar bottom-up parsable using more powerful parsing techniques: **SLR** (Simple LR)

- Use same LR(0) states.
- $^{i}\boxed{A \to \gamma.} \Rightarrow$ table$[i, T]$ = reduce($k$), for all terminals $T \in$ follow($A$)

Compare to LR(0) rule for filling parse table:

- $^{i}\boxed{A \to \gamma.} \Rightarrow$ table$[i, T] = \mathrm{r}k$, for all terminals $T$.

# Another Example – SLR

| | x | + | $ | E | T |
|---|---|---|---|---|---|
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | r2 | s4, r2 | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | r3 | r3 | r3 | | |
| 6 | r1 | r1 | r1 | | |

$S \xrightarrow{0} E\ \$$ $E \xrightarrow{2} T$
$E \xrightarrow{1} T + E$ $T \xrightarrow{3} x$

| | nullable | first | follow |
|---|---|---|---|
| S | ? | ? | ? |
| E | ? | ? | ? |
| T | ? | ? | ? |

# Another Example – SLR

| | x | + | $ | E | T |
|---|---|---|---|---|---|
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | r2 | s4, r2 | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | r3 | r3 | r3 | | |
| 6 | r1 | r1 | r1 | | |

$S \xrightarrow{0} E\ \$$ $\quad E \xrightarrow{2} T$

$E \xrightarrow{1} T + E$ $\quad T \xrightarrow{3} x$

| | nullable | first | follow |
|---|---|---|---|
| S | No | x | |
| E | No | x | $ |
| T | No | x | + $ |

| | x | + | $ | E | T |
|---|---|---|---|---|---|
| 1 | s5 | | | g2 | g3 |
| 2 | | | accept | | |
| 3 | | s4 | r2 | | |
| 4 | s5 | | | g6 | g3 |
| 5 | | r3 | r3 | | |
| 6 | | | r1 | | |

No conflicts – grammar is SLR!

# Yet Another Example

Sometimes grammar can't be parsed using SLR techniques.

| | | |
|---|---|---|
| $1\ S' \rightarrow S\ \$$ | $3\ S \rightarrow E$ | $5\ V \rightarrow \mathrm{x}$ |
| $2\ S \rightarrow V = E$ | $4\ E \rightarrow V$ | $6\ V \rightarrow *\ E$ |

This grammar is not SLR. Need more powerful parsing algorithm $\Rightarrow$ LR(1)

**Convince yourself at home!**

# LR(1) Parsing

- LR(1) item consists of two components: ($A \rightarrow \alpha.\beta$, x)
  1. Production
  2. Lookahead symbol (x)
- $\alpha$ is on top of stack, head of input is string derivable from $\beta$ x.

**LR(0) closure computation**

- Initial: $A \rightarrow \alpha.X$
- Add all items $X \rightarrow .\gamma$
- Repeat closure computation

**LR(1) closure computation**

- Initial: $A \rightarrow \alpha.X\beta,$ z
- Add all items ($X \rightarrow .\gamma, \omega$) for each $\omega \in$ first($\beta$z)
- Repeat closure computation

- shift, goto, accept table entries computed same way as LR(0)/SLR.
- reduce entries computed differently:

$$^{i}\boxed{A \rightarrow \gamma.\ ,\ \text{z}} \Rightarrow \text{table}[i,\text{z}] = \text{reduce}(k)$$

# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E$ $\quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E$ $\quad E \xrightarrow{3} V$ $\quad V \xrightarrow{5} *\ E$

1

$S' \rightarrow .\ S\ \$ \quad ?$

Initial item as in LR(0); lookahead ? arbitrary since first($z) = { $ } for all z.

# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E$ $\quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E$ $\quad E \xrightarrow{3} V$ $\quad V \xrightarrow{5} * E$

**1**

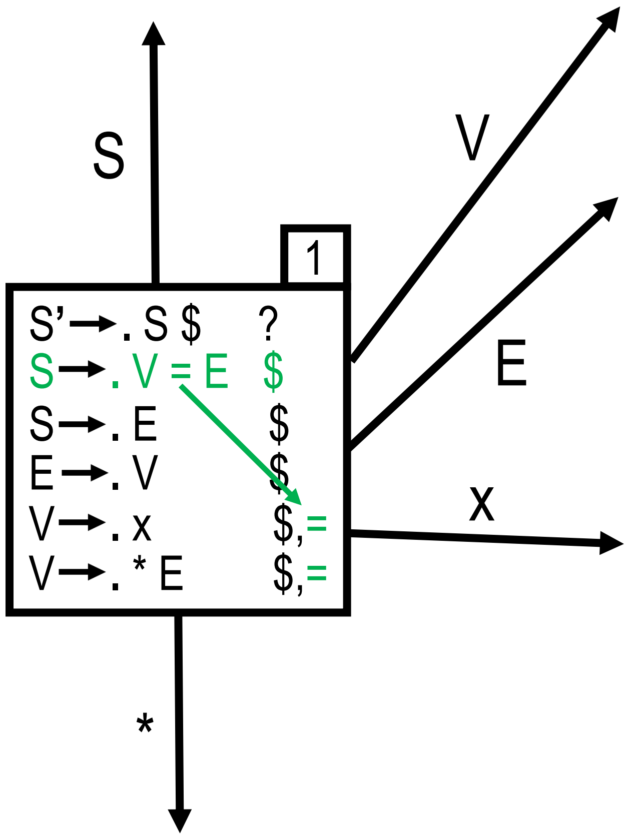
| | |
|---|---|
| $S' \rightarrow .\ S\ \$$ | ? |
| $S \rightarrow .\ V = E$ | \$ |
| $S \rightarrow .\ E$ | \$ |
| $E \rightarrow .\ V$ | \$ |
| $V \rightarrow .\ x$ | \$ |
| $V \rightarrow .\ * E$ | \$ |

Closure similar to LR(0), but with lookahead $.

# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E \quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E \quad E \xrightarrow{3} V \quad V \xrightarrow{5} {*}\ E$

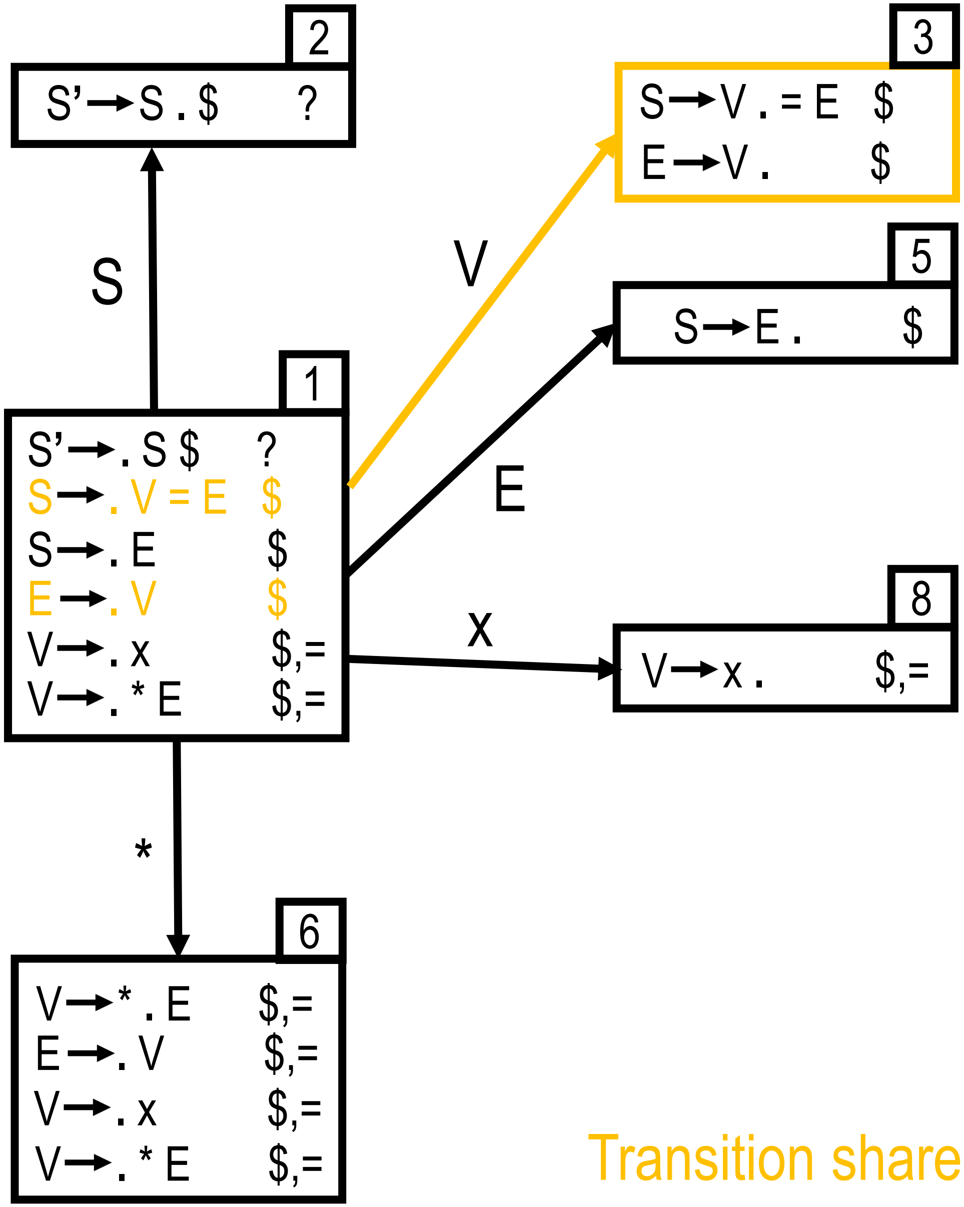



Repeated closure yields two new items, with new lookaheads.

# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E$ $\quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E$ $\quad E \xrightarrow{3} V$ $\quad V \xrightarrow{5} *\ E$




Transition shared by items from different rules.

# LR(1) example

$S' \xrightarrow{0} S\ \$$

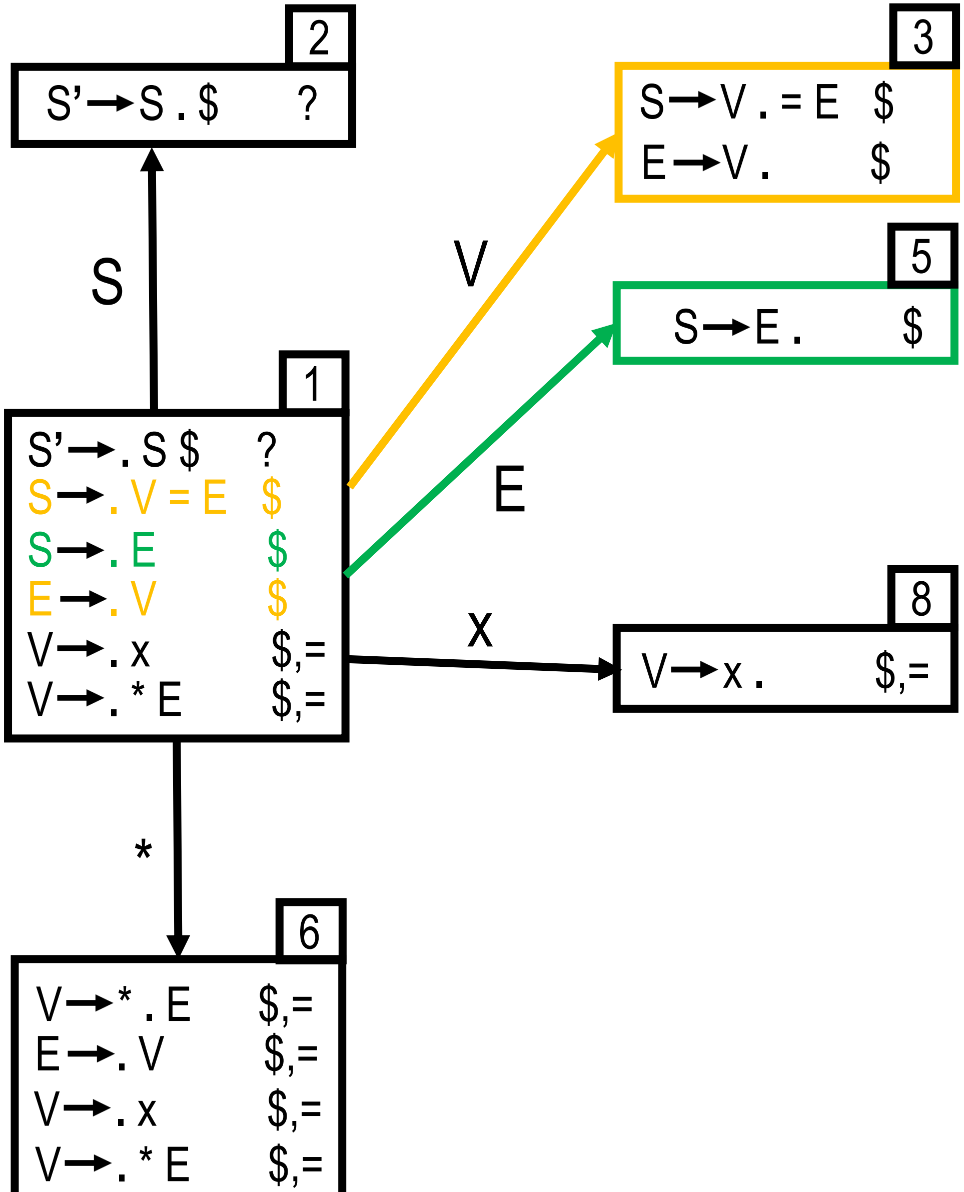
$S \xrightarrow{1} V = E$

$S \xrightarrow{2} E$

$E \xrightarrow{3} V$

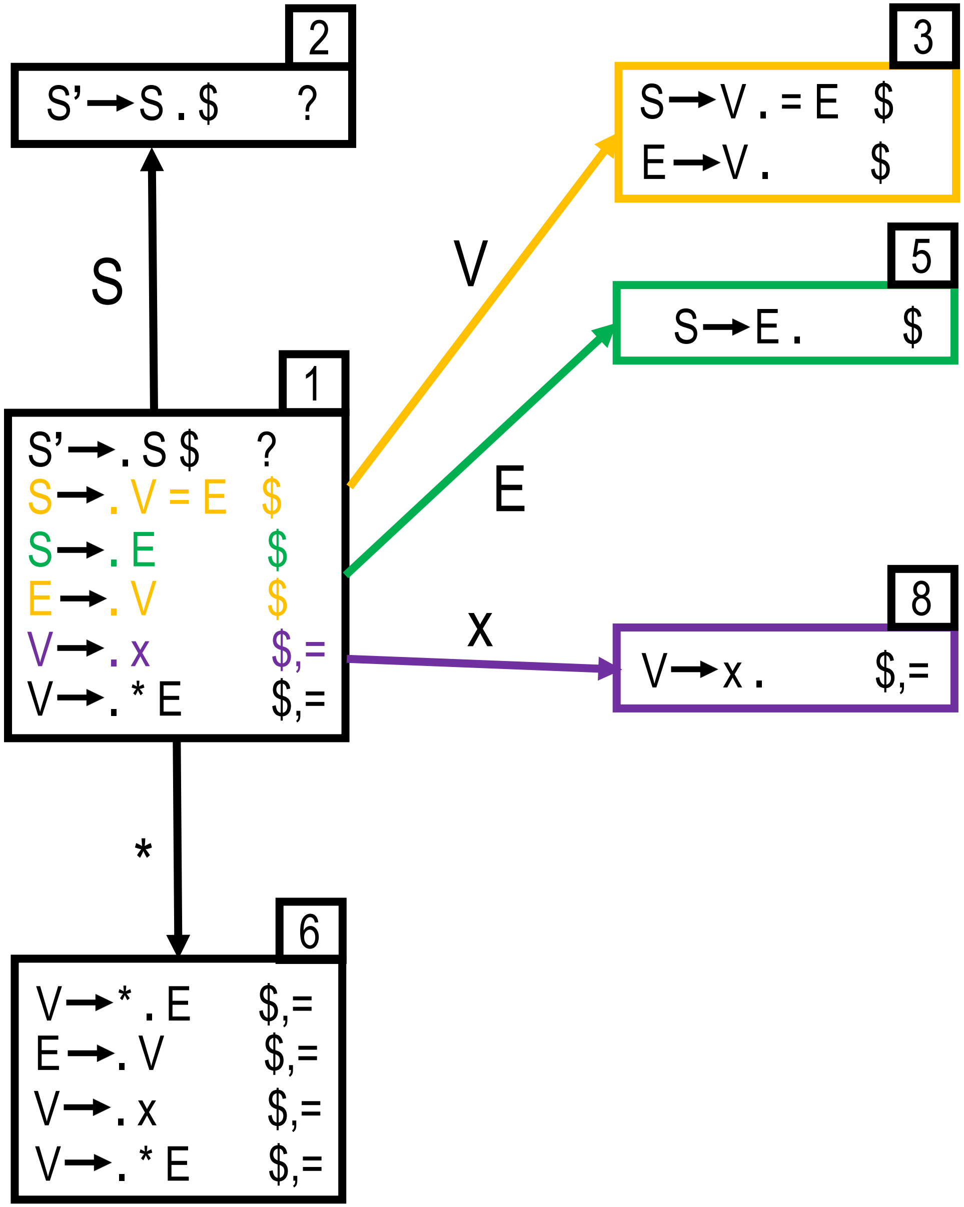
$V \xrightarrow{4} x$

$V \xrightarrow{5} {*}\ E$



# LR(1) example

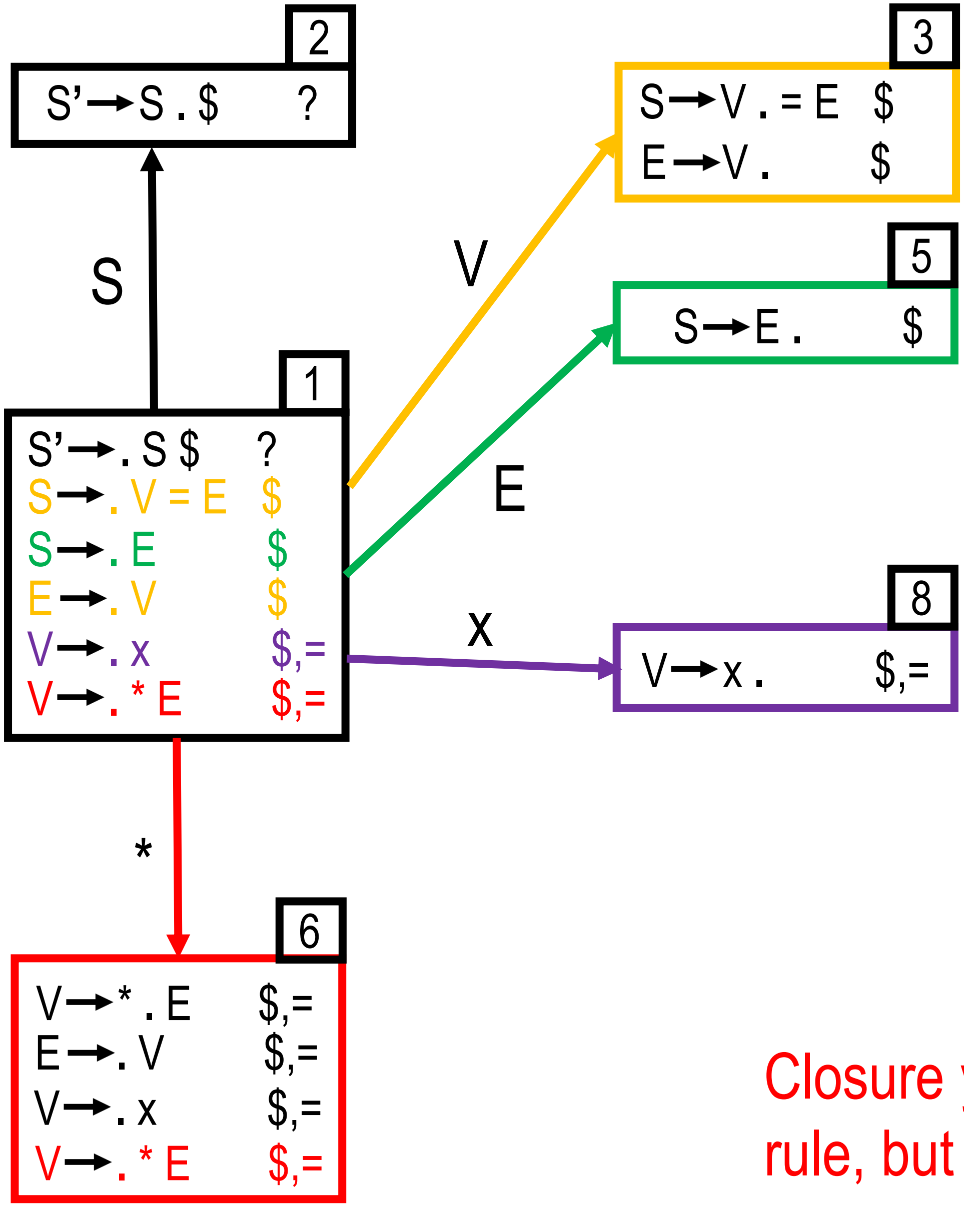
| | | |
|---|---|---|
| $S' \xrightarrow{0} S\ \$$ | $S \xrightarrow{2} E$ | $V \xrightarrow{4} x$ |
| $S \xrightarrow{1} V = E$ | $E \xrightarrow{3} V$ | $V \xrightarrow{5} {*}\ E$ |



# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E \quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E \quad E \xrightarrow{3} V \quad V \xrightarrow{5} *\ E$

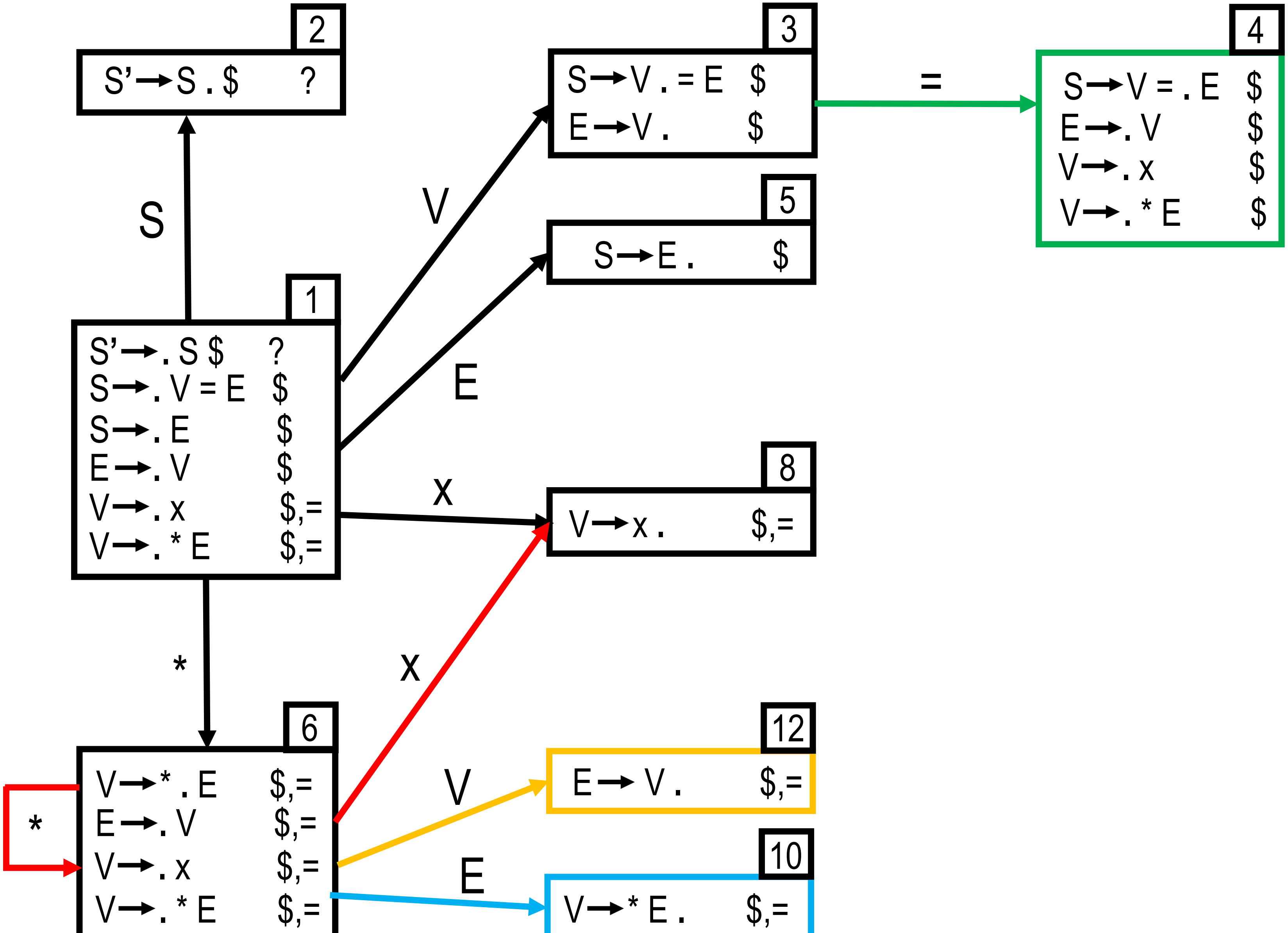



Closure yields second item for same rule, but different cursor position.

# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E \quad V \xrightarrow{4} x$

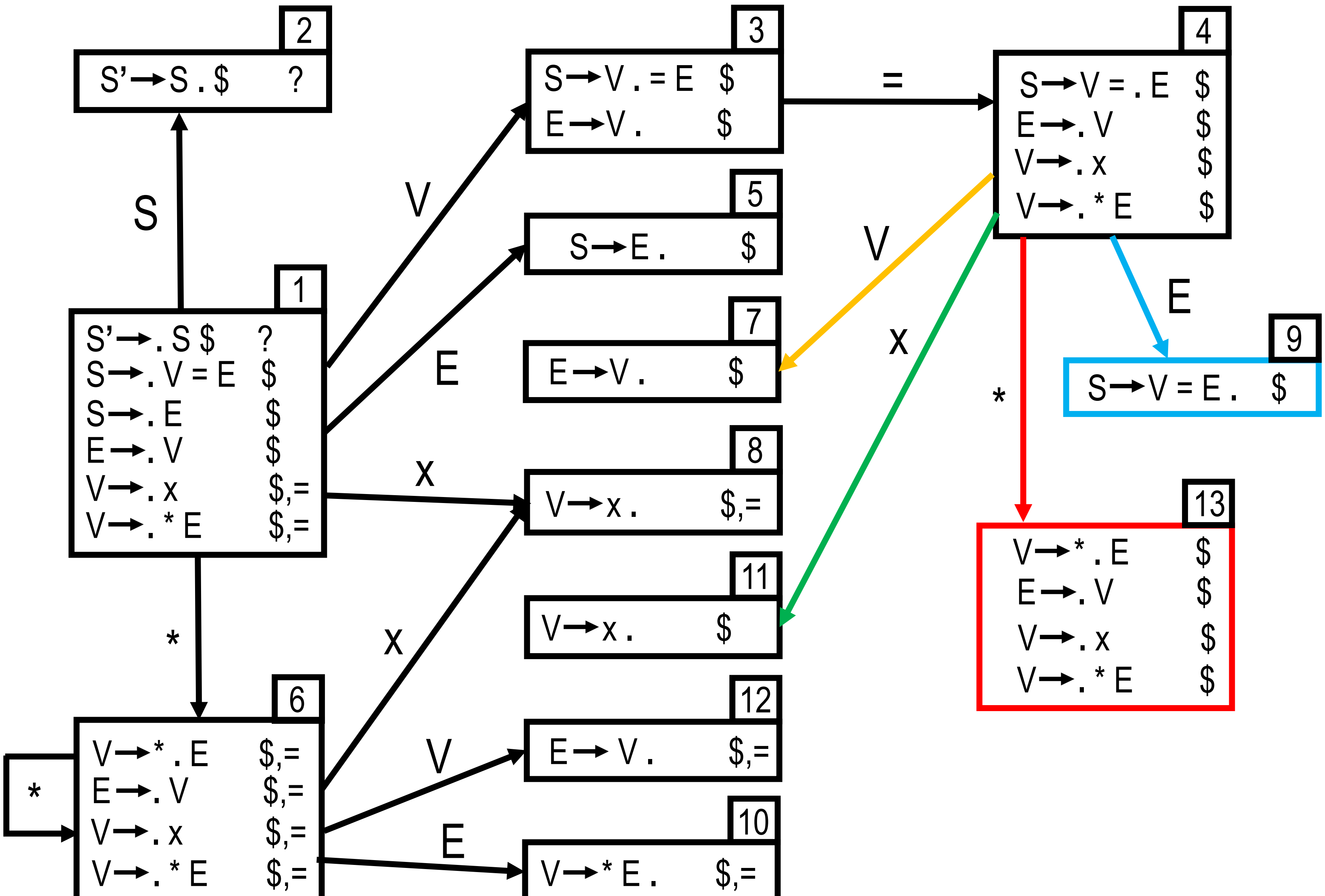
$S \xrightarrow{1} V = E \quad E \xrightarrow{3} V \quad V \xrightarrow{5} * E$



# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\qquad S \xrightarrow{2} E$ $\qquad V \xrightarrow{4} x$

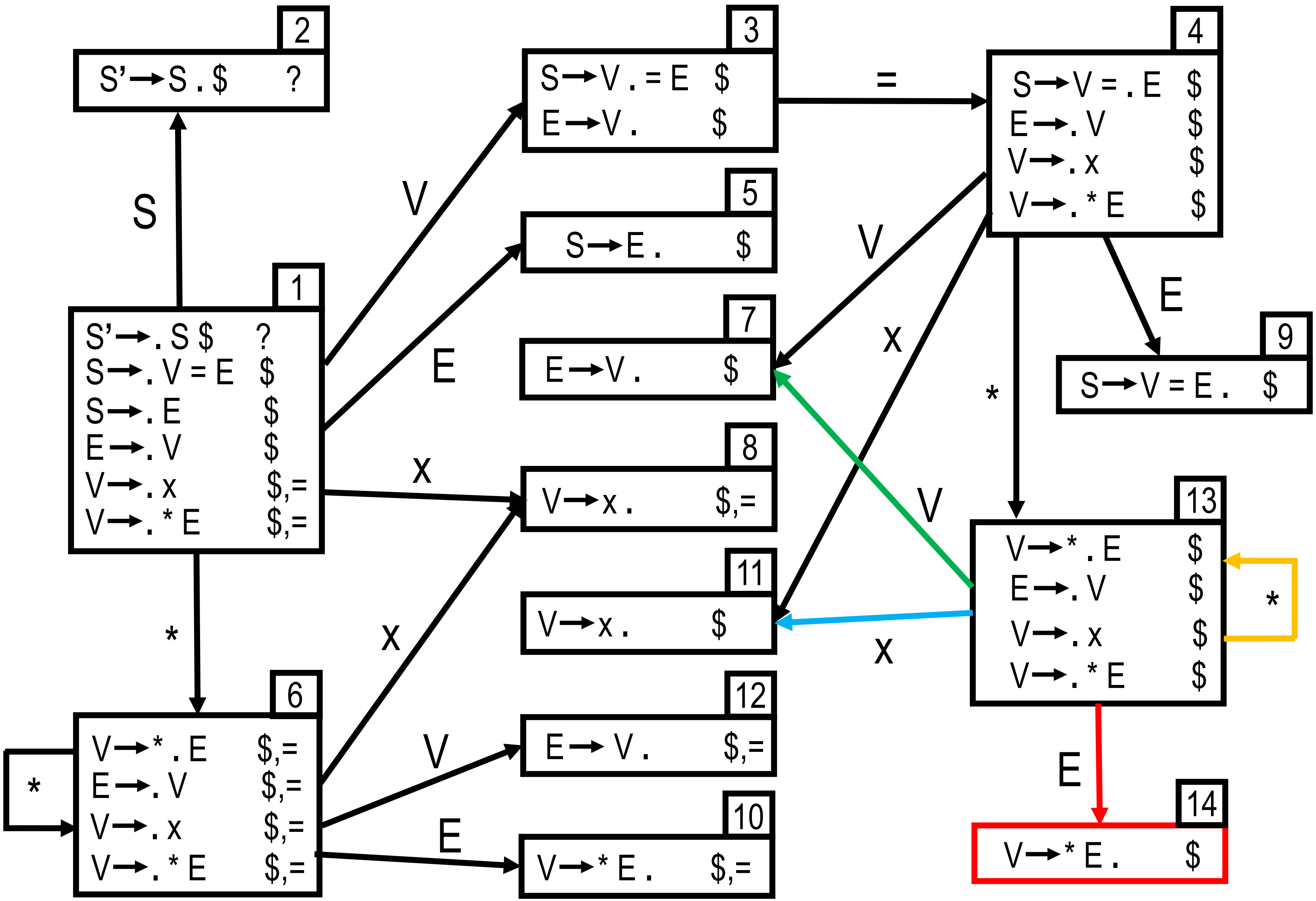
$S \xrightarrow{1} V = E$ $\qquad E \xrightarrow{3} V$ $\qquad V \xrightarrow{5} * E$



# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E \quad V \xrightarrow{4} x$

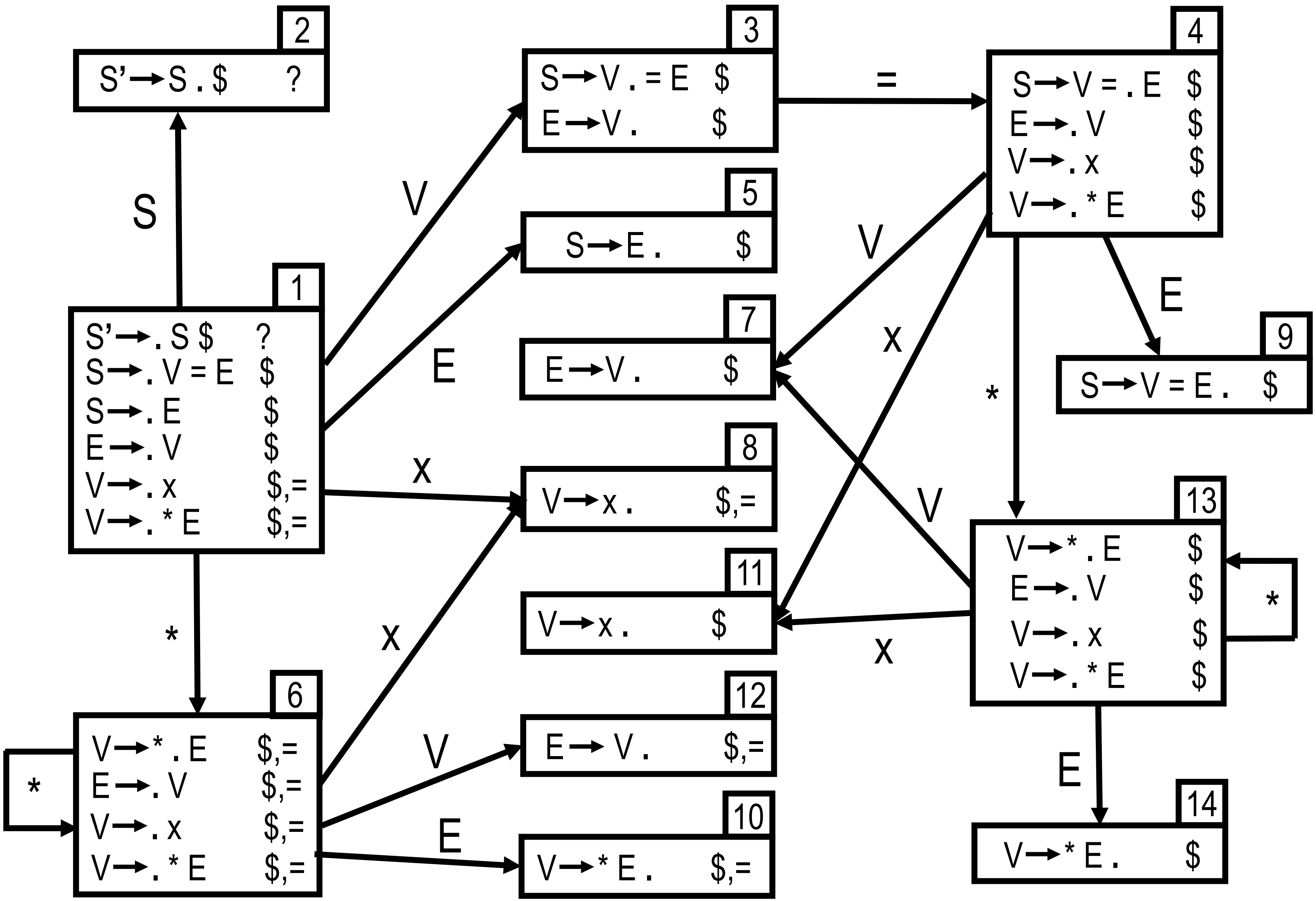
$S \xrightarrow{1} V = E \quad E \xrightarrow{3} V \quad V \xrightarrow{5} * E$



# LR(1) example

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E \quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E \quad E \xrightarrow{3} V \quad V \xrightarrow{5} * E$

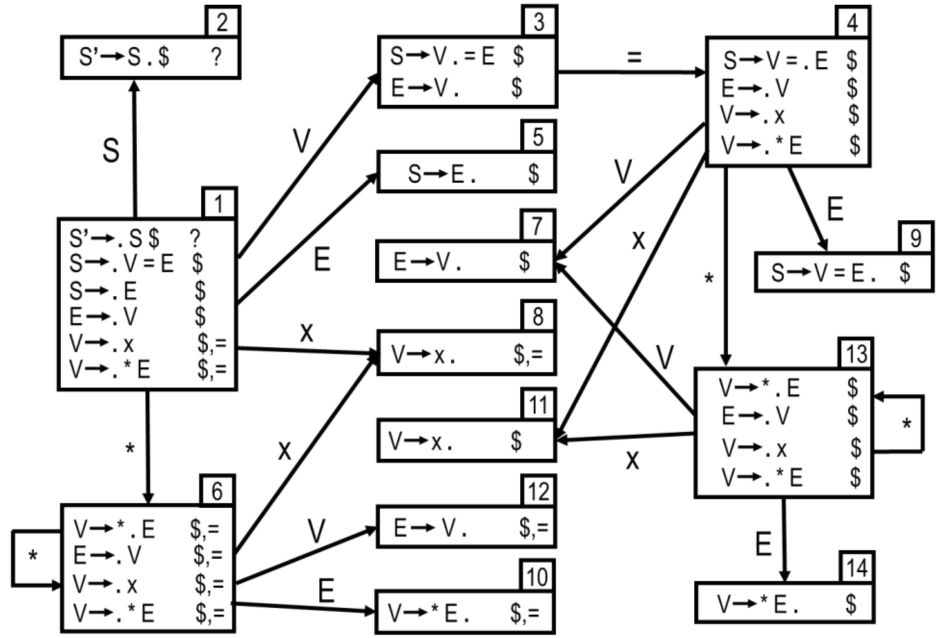


# LR(1) parsing table

$S' \xrightarrow{0} S\ \$$ $S \xrightarrow{2} E$ $V \xrightarrow{4} x$

$S \xrightarrow{1} V = E$ $E \xrightarrow{3} V$ $V \xrightarrow{5} {*}\ E$

| | x | * | = | $ | S | E | V |
|---|---|---|---|---|---|---|---|
| 1 | s8 | s6 | | | | | |
| 2 | | | | | | | |
| 3 | | | s4 | | | | |
| 4 | s11 | s13 | | | | | |
| 5 | | | | | | | |
| 6 | s8 | s6 | | | | | |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | s11 | s13 | | | | | |
| 14 | | | | | | | |



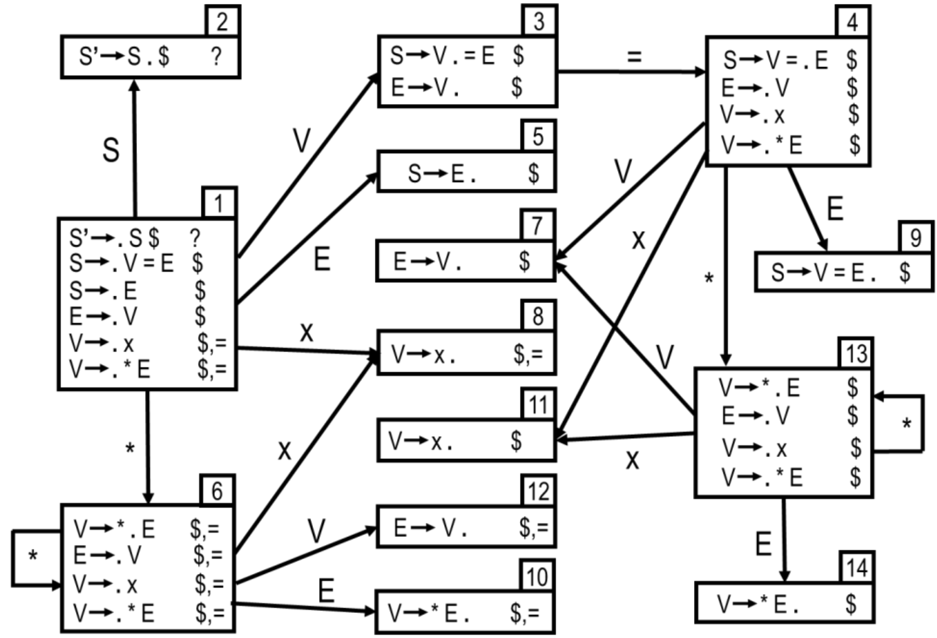

shift as before: when dot is before terminal, ie transition marked by terminal.

# LR(1) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E \quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E \quad E \xrightarrow{3} V \quad V \xrightarrow{5} *\ E$

| | x | * | = | $ | S | E | V |
|---|---|---|---|---|---|---|---|
| 1 | s8 | s6 | | | g2 | g5 | g3 |
| 2 | | | | | | | |
| 3 | | | s4 | | | | |
| 4 | s11 | s13 | | | | g9 | g7 |
| 5 | | | | | | | |
| 6 | s8 | s6 | | | | g10 | g12 |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | s11 | s13 | | | | g14 | g7 |
| 14 | | | | | | | |



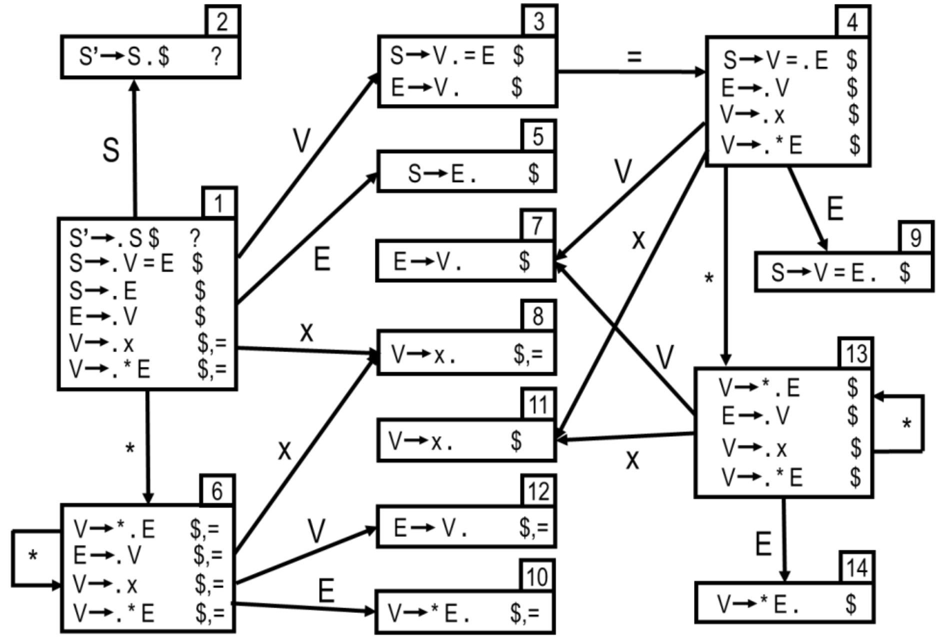

goto as before: when dot is before nonterminal, ie transition marked by nonterminal.

# LR(1) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E$ $\quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E$ $\quad E \xrightarrow{3} V$ $\quad V \xrightarrow{5} * E$

| | x | * | = | $ | S | E | V |
|---|---|---|---|---|---|---|---|
| 1 | s8 | s6 | | | g2 | g5 | g3 |
| 2 | | | | acc | | | |
| 3 | | | s4 | | | | |
| 4 | s11 | s13 | | | | g9 | g7 |
| 5 | | | | | | | |
| 6 | s8 | s6 | | | | g10 | g12 |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | s11 | s13 | | | | g14 | g7 |
| 14 | | | | | | | |



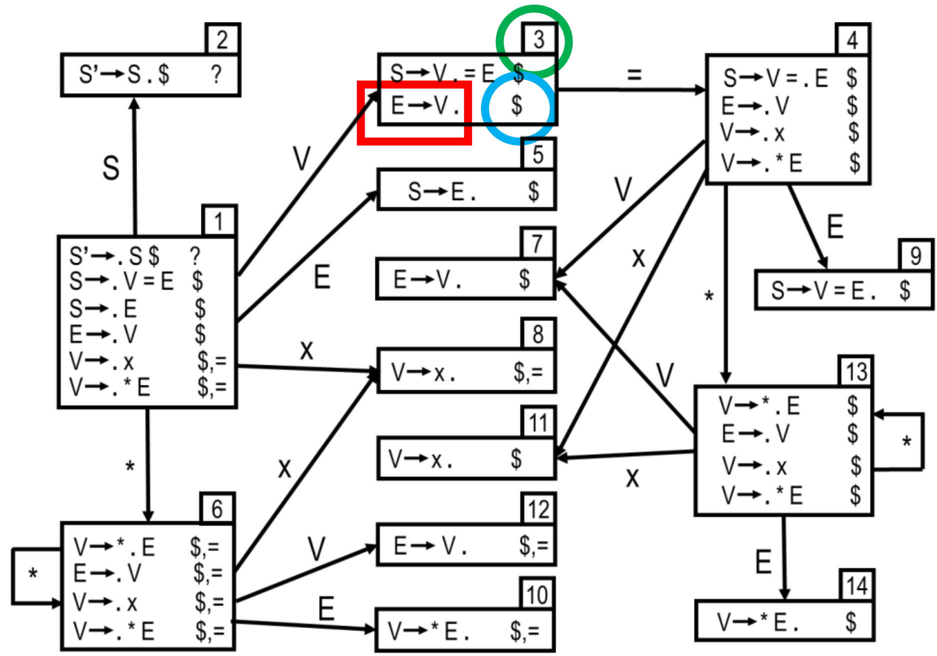

accept as before: when dot is before $.

# LR(1) parsing table

$S' \xrightarrow{0} S\ \$$ $S \xrightarrow{2} E$ $V \xrightarrow{4} x$

$S \xrightarrow{1} V = E$ $\boxed{E \xrightarrow{3} V}$ $V \xrightarrow{5} *\ E$

| | x | * | = | $ | S | E | V |
|---|---|---|---|---|---|---|---|
| 1 | s8 | s6 | | | g2 | g5 | g3 |
| 2 | | | | acc | | | |
| 3 | | | s4 | **r3** | | | |
| 4 | s11 | s13 | | | | g9 | g7 |
| 5 | | | | | | | |
| 6 | s8 | s6 | | | | g10 | g12 |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | s11 | s13 | | | | g14 | g7 |
| 14 | | | | | | | |



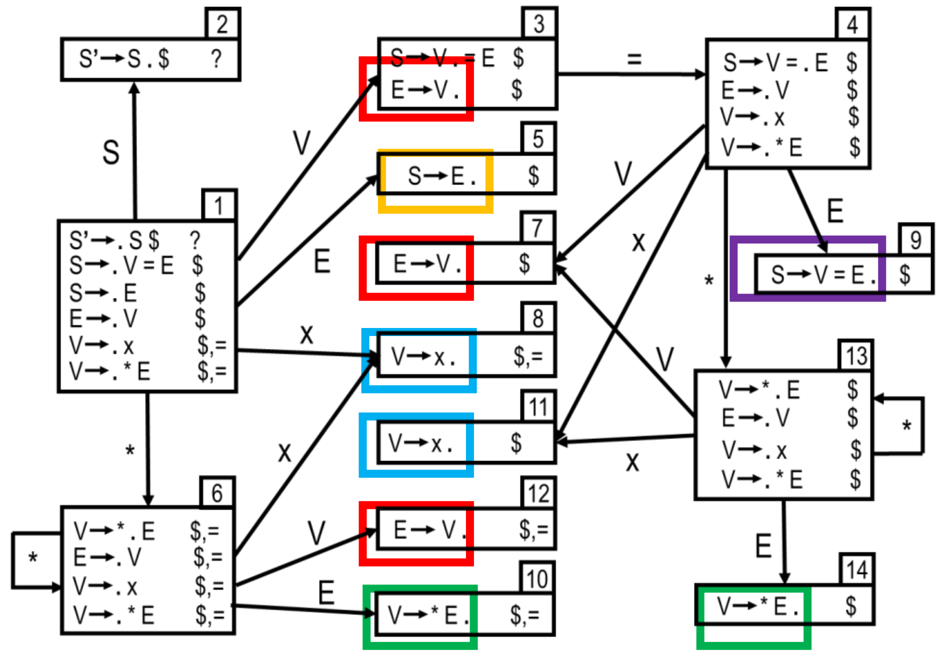

reduce n: when dot is at end of item for rule n with lookahead t in state X, add reduce n in cell (X, t).

# LR(1) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E$ $\quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E$ $\quad E \xrightarrow{3} V$ $\quad V \xrightarrow{5} *\ E$

| | x | * | = | $ | S | E | V |
|---|---|---|---|---|---|---|---|
| 1 | s8 | s6 | | | g2 | g5 | g3 |
| 2 | | | | acc | | | |
| 3 | | | s4 | r3 | | | |
| 4 | s11 | s13 | | | | g9 | g7 |
| 5 | | | | r2 | | | |
| 6 | s8 | s6 | | | | g10 | g12 |
| 7 | | | | r3 | | | |
| 8 | | | r4 | r4 | | | |
| 9 | | | | r1 | | | |
| 10 | | | r5 | r5 | | | |
| 11 | | | | r4 | | | |
| 12 | | | r3 | r3 | | | |
| 13 | s11 | s13 | | | | g14 | g7 |
| 14 | | | | r5 | | | |



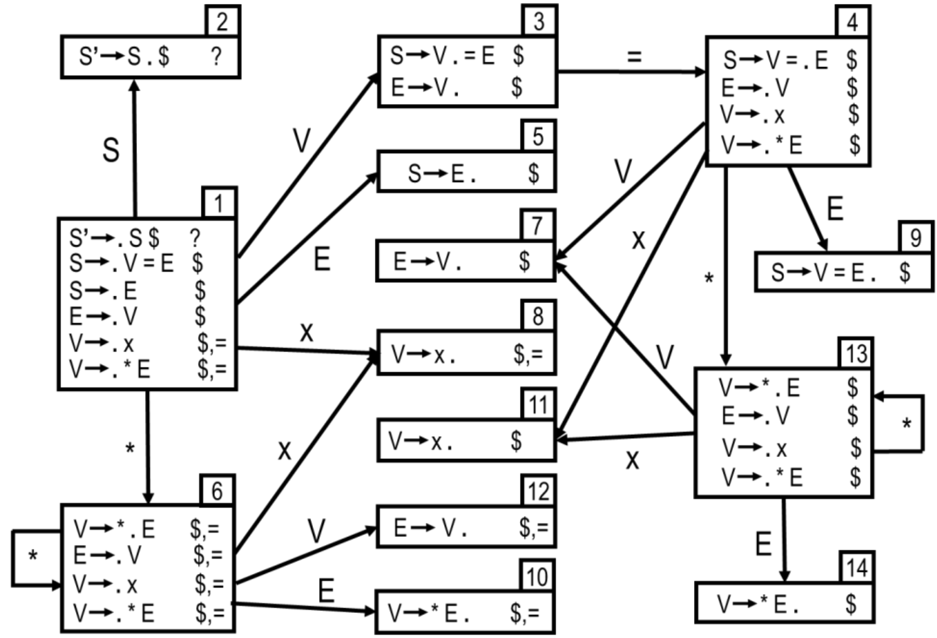

reduce n: when dot is at end of item for rule n with lookahead t in state X, and add reduce n in cell (X, t).

# LR(1) parsing table

$S' \xrightarrow{0} S\ \$$ $\quad S \xrightarrow{2} E \quad V \xrightarrow{4} x$

$S \xrightarrow{1} V = E \quad E \xrightarrow{3} V \quad V \xrightarrow{5} {*}\ E$

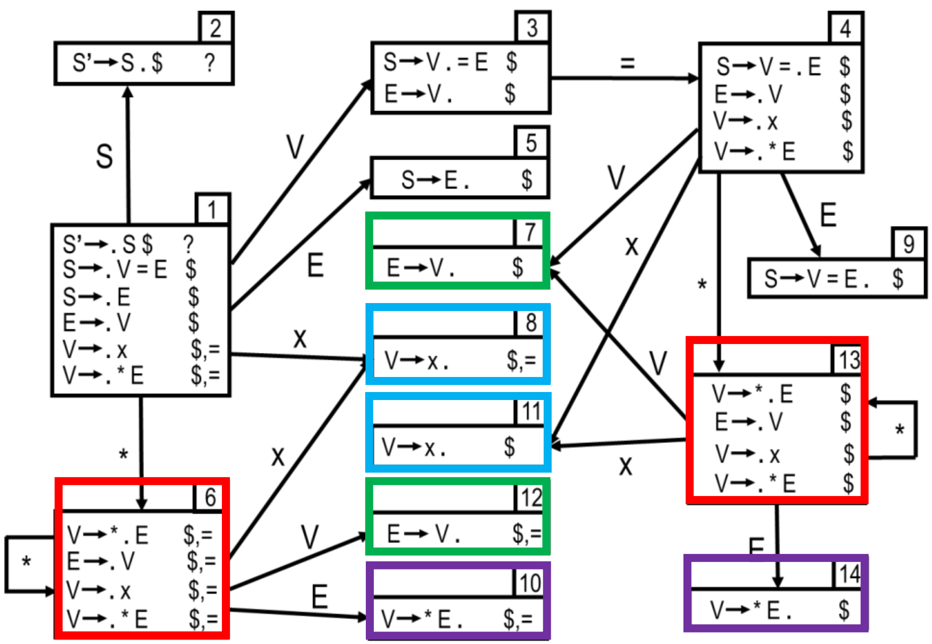
| | x | * | = | $ | S | E | V |
|---|---|---|---|---|---|---|---|
| 1 | s8 | s6 | | | g2 | g5 | g3 |
| 2 | | | | acc | | | |
| 3 | | | s4 | r3 | | | |
| 4 | s11 | s13 | | | | g9 | g7 |
| 5 | | | | r2 | | | |
| 6 | s8 | s6 | | | | g10 | g12 |
| 7 | | | | r3 | | | |
| 8 | | | r4 | r4 | | | |
| 9 | | | | r1 | | | |
| 10 | | | r5 | r5 | | | |
| 11 | | | | r4 | | | |
| 12 | | | r3 | r3 | | | |
| 13 | s11 | s13 | | | | g14 | g7 |
| 14 | | | | r5 | | | |




No duplicate entries ➔ Grammar is LR(1)!

# LALR(1)

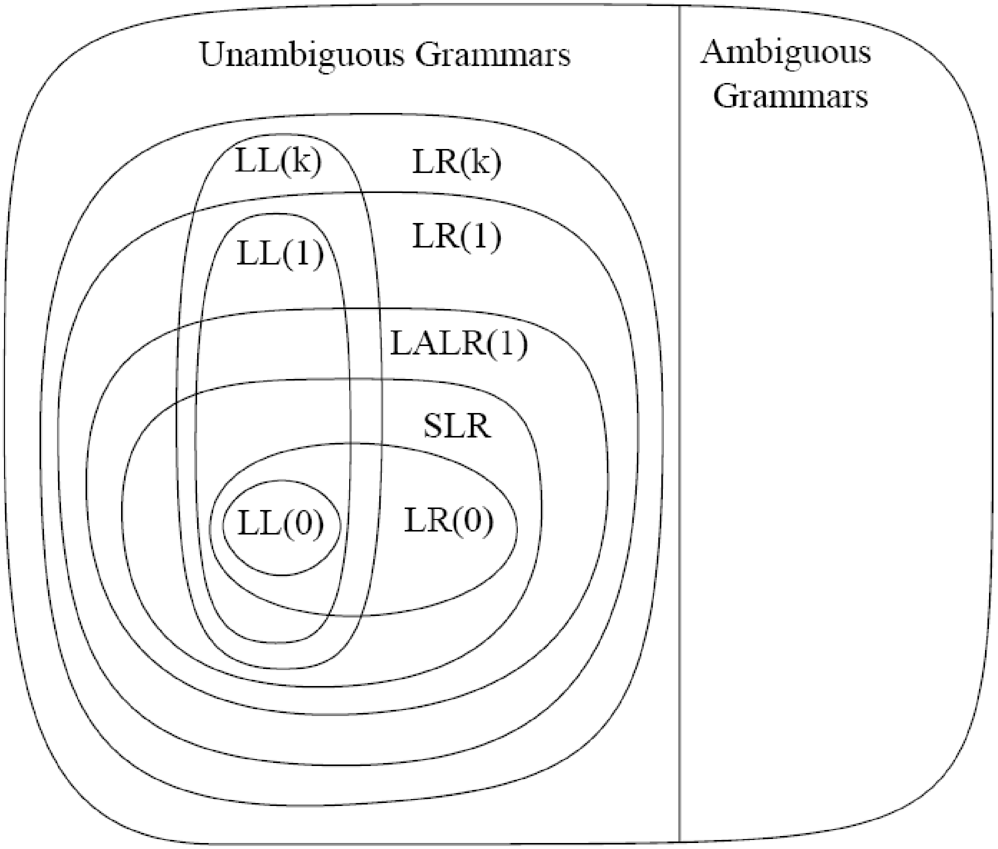
- Problem with LR(1) parsers: tables too large!
  - Can make smaller table by merging states whose items are identical except for look-ahead sets ⇒ LALR(1) (Look-Ahead LR(1)).
  - LALR(1) transition table may contain shift-reduce/reduce-reduce conflicts where LR(1) table has none.




s11=s8 s12=s7 s13=s6 s14=s10

| | x | * | = | $ | S | E | V |
|---|---|---|---|---|---|---|---|
| 1 | s8 | s6 | | | g2 | g5 | g3 |
| 2 | | | | acc | | | |
| 3 | | | s4 | r3 | | | |
| 4 | s8 | s6 | | | | g9 | g7 |
| 5 | | | | r2 | | | |
| 6 | s8 | s6 | | | | g10 | g7 |
| 7 | | | r3 | r3 | | | |
| 8 | | | r4 | r4 | | | |
| 9 | | | | r1 | | | |
| 10 | | | r5 | r5 | | | |

# Parsing Power




ML-YACC uses LALR(1) parsing because reasonable programming languages can be specified by an LALR(1) grammar. (Figure from MCI in ML.)

# Parsing 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 errors.

**How should syntax errors be handled?**

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

# Error Recovery

***Error Recovery*: process of adjusting input stream so that parsing may resume after 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 detected.

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

# LL Local Error Recovery

Consider LL(1) parsing context:

$$Z \rightarrow XYZ \qquad Y \rightarrow \text{c} \qquad X \rightarrow \text{a}$$
$$Z \rightarrow \text{d} \qquad Y \rightarrow \epsilon \qquad X \rightarrow \text{b}\, Y\, \text{e}$$

| | 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 |
|---|---|---|---|---|---|
| $Z$ | $Z \rightarrow XYZ$ | $Z \rightarrow XYZ$ | | $Z \rightarrow d$ | |
| $Y$ | $Y \rightarrow \epsilon$ | $Y \rightarrow \epsilon$ | $Y \rightarrow c$ | $Y \rightarrow \epsilon$ | $Y \rightarrow \epsilon$ |
| $X$ | $X \rightarrow a$ | $X \rightarrow bYe$ | | | |

# LL Local Error Recovery

Local Recovery Technique: in function A(), delete token types from input stream until 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 error();
...
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

Example:

| | | |
|---|---|---|
| E → ID | E → ( ES ) | ES → ES ; E |
| E → E + E | ES → E | |

- match a sequence of erroneous input tokens using the **error** token – a fresh terminal
- preferable: have **error** followed with synchronizing lookahead token, like RPAREN and SEMI here, by adding rules like this:

E → (error)        ES → error ; E

(alternative: add E → error but that does not allow us to skip ahead to RPAREN, SEMI)

# LR Local Error Recovery

Example:

| | | |
|---|---|---|
| E → ID | E → ( ES ) | ES → ES ; E |
| E → E + E | ES → E | |

- match a sequence of erroneous input tokens using the **error** token – a fresh terminal
- preferable: have **error** followed with synchronizing lookahead token, like RPAREN and SEMI here, by adding rules like this:

E → (error)        ES → error ; E

(alternative: add E → error but that does not allow us to skip ahead to RPAREN, SEMI)

- build parse table for the extended grammar

# LR Local Error Recovery

Example:

| | | |
|---|---|---|
| E → ID | E → ( ES ) | ES → ES ; E |
| E → E + E | ES → E | |

- match a sequence of erroneous input tokens using the **error** token – a fresh terminal
- preferable: have **error** followed with synchronizing lookahead token, like RPAREN and SEMI here, by adding rules like this:

E → (error)    ES → error ; E

(alternative: add E → error but that does not allow us to skip ahead to RPAREN, SEMI)

- build parse table for the extended grammar
- LR parse engine: when trying to read from empty cell,
  1. pop the stack until a state is reached in which the action for **error** is **shift**
  2. do the **shift** action

# LR Local Error Recovery

Example:

| | | |
|---|---|---|
| E → ID | E → ( ES ) | ES → ES ; E |
| E → E + E | ES → E | |

- match a sequence of erroneous input tokens using the **error** token – a fresh terminal
- preferable: have **error** followed with synchronizing lookahead token, like RPAREN and SEMI here, by adding rules like this:

| | |
|---|---|
| E → (error) | ES → error ; E |

(alternative: add $\boxed{\text{E} \rightarrow \text{error}}$ but that does not allow us to skip ahead to RPAREN, SEMI)

- build parse table for the extended grammar
- LR parse engine: when trying to read from empty cell,
  1. pop the stack until a state is reached in which the action for **error** is **shift**
  2. do the **shift** action
  3. discard the input symbols (if necessary) until a state is reached that has a proper shift/goto/reduce/accept action in the current state (in case we have indeed synchronizing lookahead, this will be a shift action for one of the lookaheads)
  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 ⇒ *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, deletion, and substitution at all points between $(n-k)^{th}$ and $n^{th}$ token.

| n-k | n-k+1 | - - - - - | n-1 | n |
|---|---|---|---|---|

- Best repair: one that allows parser to parse furthest past $n^{th}$ token.
- If languages has $N$ token types, then:

$$\text{total \# of repairs} = \text{deletions} + \text{insertions} + \text{substitutions}$$
$$\text{total \# of repairs} = (k) + (k+1)\,N + (k)\,(N-1)$$

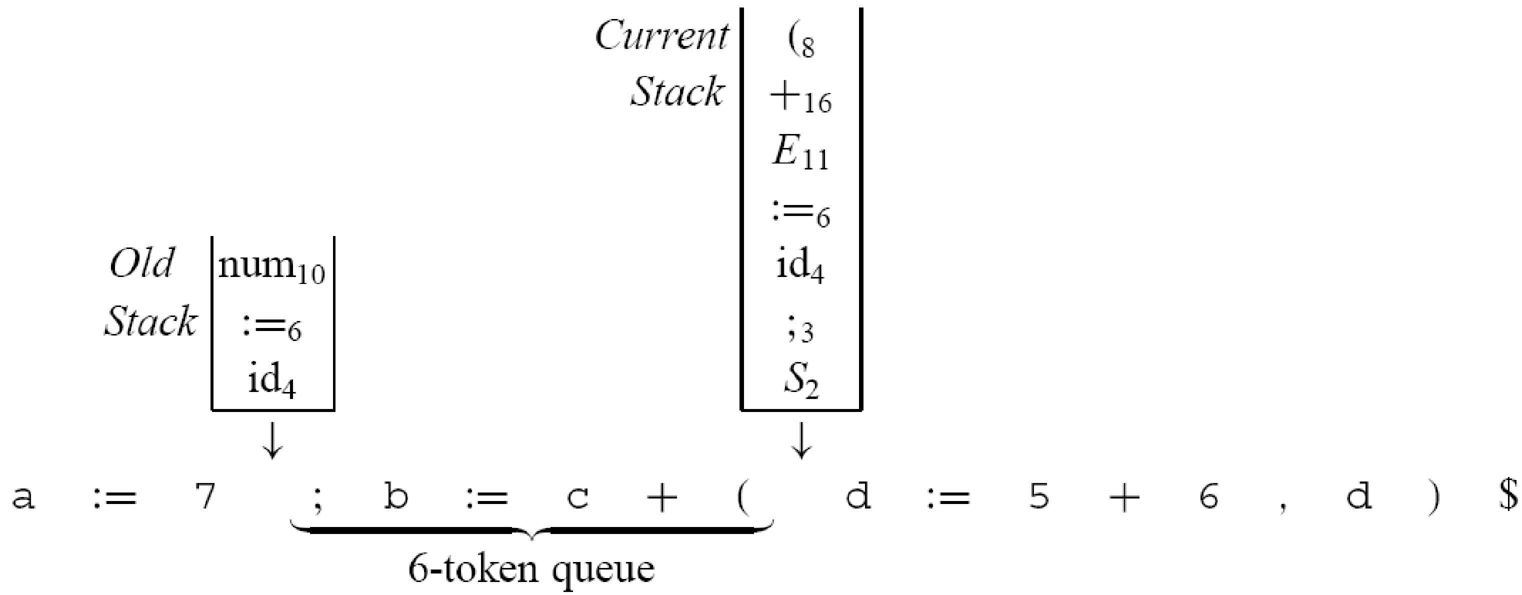
# Burke-Fisher

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

1. *$k$-length buffer/queue* - if parser currently processing $n^{th}$ token, queue contains tokens $(n-k) \rightarrow (n-1)$. (ML-Yacc $k = 15$)
2. *old parse stack* - if parser currently processing $n^{th}$ token, old stack represents stack state when parser was processing $(n-k)^{th}$ token.

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

# Burke-Fisher Example




- 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 -> ELSE
           |    -> IN INT END
```