CS406: Compilers

Spring 2022

Week 5: Parsers – Bottom-up Parsing (background concepts), Bottom-up parsing (use of goto and action tables)

Concept: configuration / item

- Configuration or item has a form:

  $A \;\text{->}\; X_1 ... \; X_i \;\bullet\; X_{i+1} \;...\; X_j$

- Dot $\bullet$ can appear anywhere
- Represents a production part of which has been matched (what is to the left of Dot)
- LR parsers keep track of multiple (all) productions that can be potentially matched
  - We need a *configuration set*

Concept: configuration / item

- E.g. configuration set

```
stmt -> ID• := expr
stmt -> ID• : stmt
stmt -> ID•
```

Corresponding to productions:

```
stmt -> ID := expr
stmt -> ID : stmt
stmt -> ID
```

- Dot at the extreme left of RHS of a production denotes that production is predicted
- Dot at the extreme right of RHS of a production denotes that production is recognized
- if Dot precedes a Non-Terminal in a configuration set, more configurations need to be added to the set

Concept: closure

➢ For each configuration in the configuration set,

A -> α•Bγ, where B is a non-terminal,

1 add configurations of the form:

B -> • δ

2 if the addition introduces a configuration with Dot behind a new non-Terminal N, add all configurations having the form N -> • ε

Repeat 2 when another new non-terminal is introduced and so on..

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> •E$}

⇩ ← Non-terminal

S -> •E$

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

⇩ (Non-terminal)

```
S -> •E$
E -> •E+T
```

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

⇩ (Non-terminal)

```
S -> • E$
E -> • E+T
E -> • T
```

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

⇩

```
S -> •E$
E -> •E+T
E -> •T
```

New Non-terminal

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

```
S -> •E$
E -> •E+T
E -> •T
T -> •ID
```

New Non-terminal

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

⇩

```
S -> •E$
E -> •E+T
E -> •T
T -> •ID
T -> •(E)
```

New Non-terminal

Concept: closure

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. closure {S -> • E$}

⇩

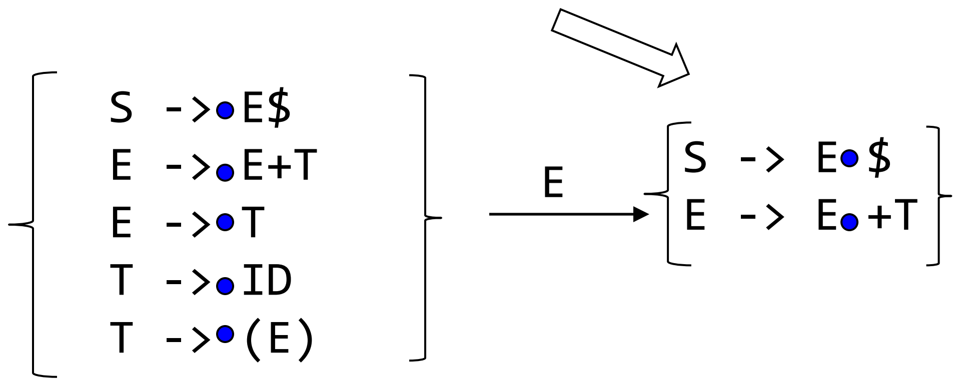
```
{
S -> •E$
E -> •E+T
E -> •T
T -> •ID
T -> •(E)
}
```

Concept: successor

Grammar

```
S -> E$
E -> E+T | T
T -> ID | (E)
```

- E.g. successor ({S ->•E$}, E)



- Consider all symbols that are to the <u>immediate right of Dot</u> and compute respective successors
  - You must compute closure of successor before finalizing items in successor

Concept: CFSM

- Each configuration set becomes a state
- The symbol used as input for computing the successor becomes the transition
- Configuration-set finite state machine (CFSM)
  - The state diagram obtained after computing the chain of all successors (for all symbols) starting from the configuration involving the <u>first production</u>

Example: CFSM

Grammar

P->S

S->x;S

S->e

Start with a configuration for the first production

P->• S

Example: CFSM

Compute closure

P-> • S ← Non-terminal

Grammar

P->S

S->x;S

S->e

Example: CFSM

Add item

P->• S
S->• x;S

Grammar

P->S
S->x;S
S->e

Example: CFSM

Add item

P->•S
S->•x;S
S->•e

Grammar

P->S
S->x;S
S->e

Example: CFSM

Grammar

P->S

S->x;S

S->e

No new non-terminal before Dot. This becomes a state in CFSM

P->•S
S->•x;S
S->•e

state 0

Example: CFSM

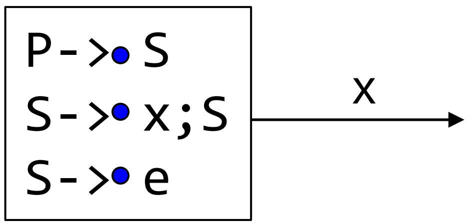
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol x

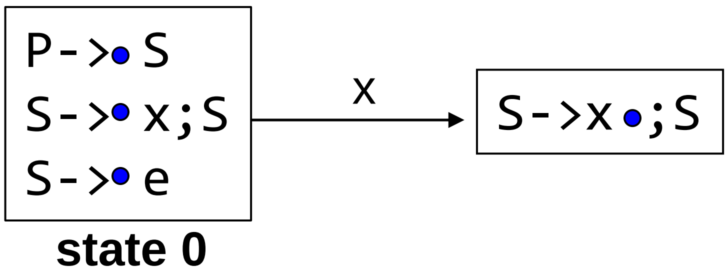


state 0

Consider items (in state 0), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Compute successor (of state 0) under symbol x



Grammar

P->S

S->x;S

S->e

Consider items (in state 0), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

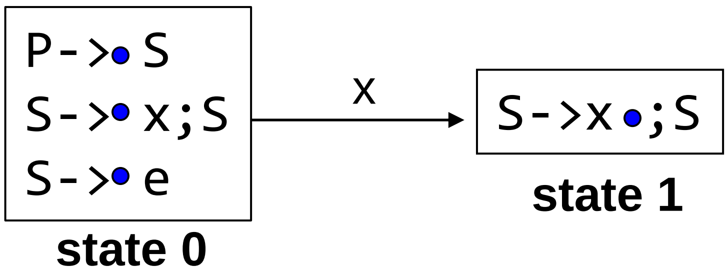
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol x



Consider items (in state 0), where x is to the immediate right of Dot. Advance Dot by one symbol.

No non-terminals immediately after Dot in the successor. So, no configurations get added. Successor becomes another state in CFSM.

Example: CFSM

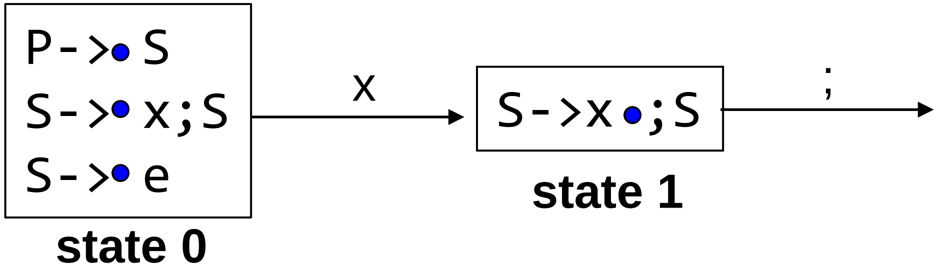
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;



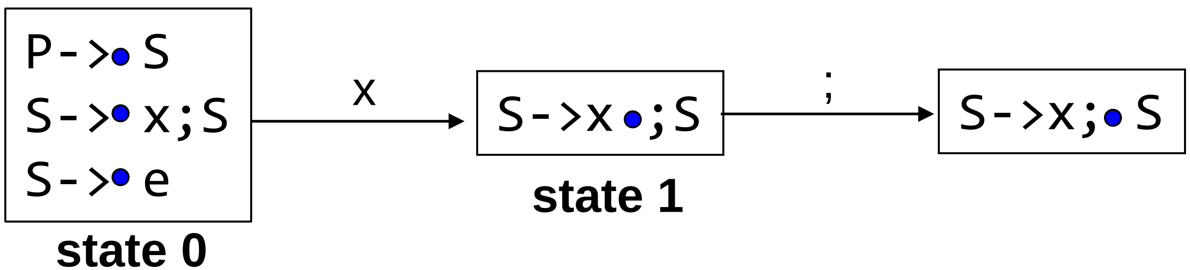
Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

Example: CFSM

Grammar

P->S
S->x;S
S->e

Compute successor (of state 1) under symbol ;



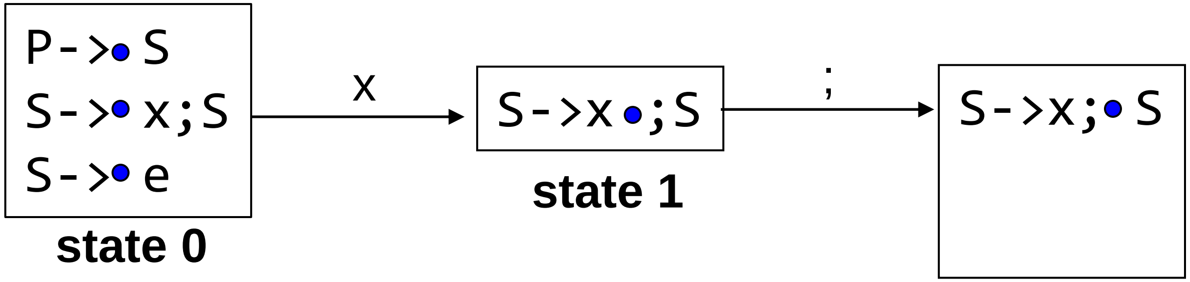
Consider items (in state 1), where ; is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Grammar

P->S
S->x;S
S->e

Compute successor (of state 1) under symbol ;

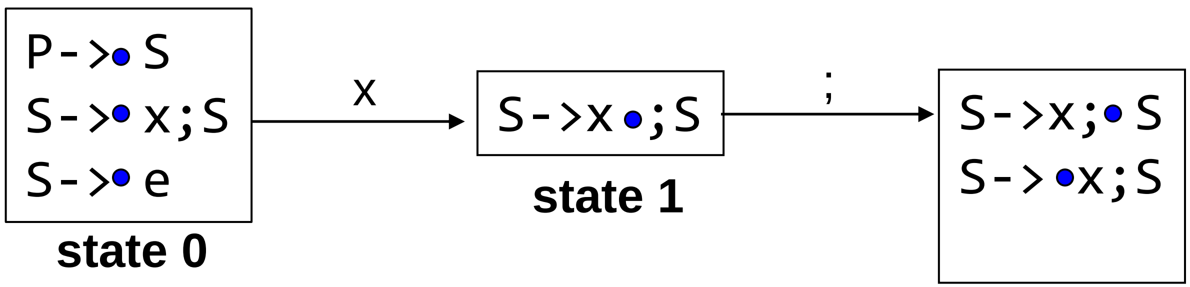


Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

Compute successor (of state 1) under symbol ;



Grammar

P->S

S->x;S

S->e

Consider items (in state 1), where ; is to the immediate right of Dot.
Advance Dot by one symbol.

There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

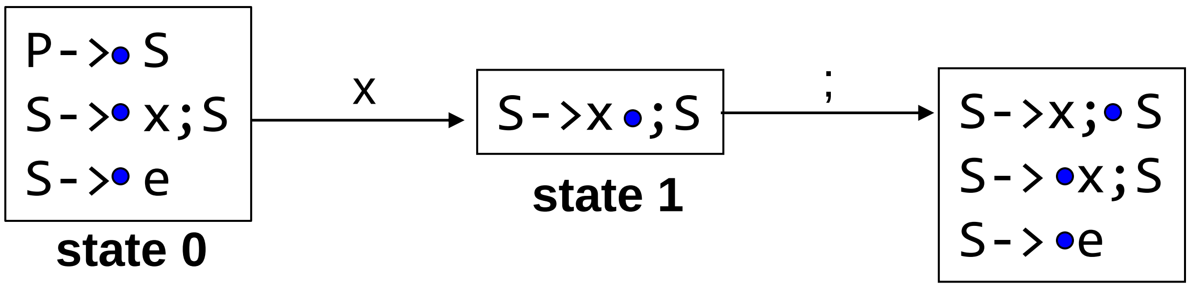
Grammar

P->S

S->x;S

S->e

Compute successor (of state 1) under symbol ;

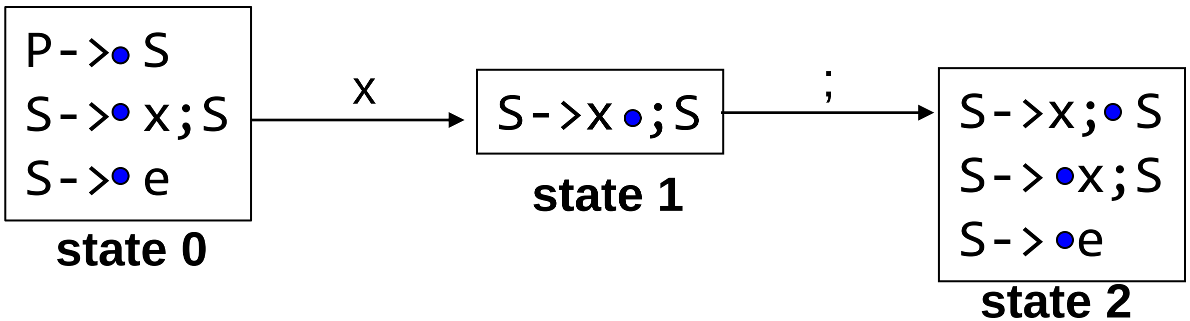


Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations.

Example: CFSM

Compute successor (of state 1) under symbol ;



Grammar

P->S

S->x;S

S->e

Consider items (in state 1), where ; is to the immediate right of Dot. Advance Dot by one symbol.

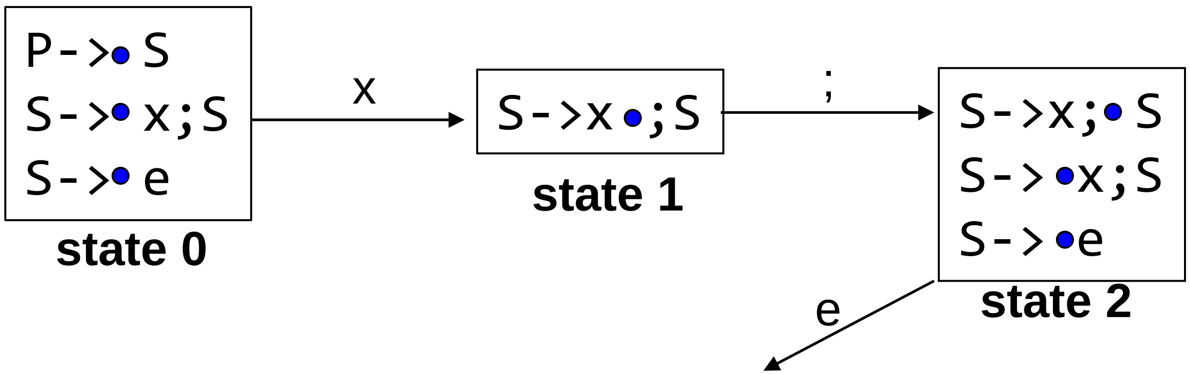
There is a non-terminal immediately after Dot in the successor of state 1. So, add configurations. No more items to be added. Becomes another state in CFSM.

Example: CFSM

Grammar

P->S
S->x;S
S->e

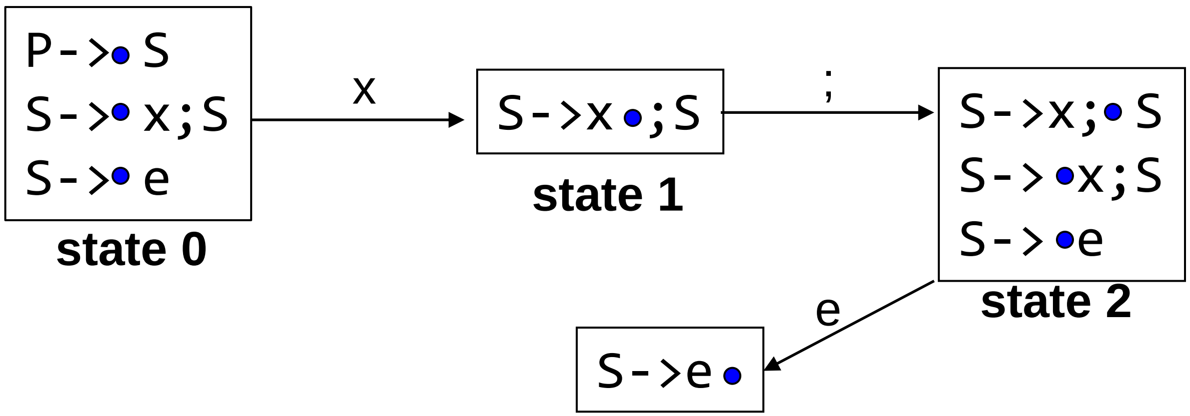
Compute successor (of state 2) under symbol e



Consider items (in state 2), where e is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Compute successor (of state 2) under symbol e



Grammar

P->S

S->x;S

S->e

Consider items (in state 2), where e is to the immediate right of Dot. Advance Dot by one symbol.

Example: CFSM

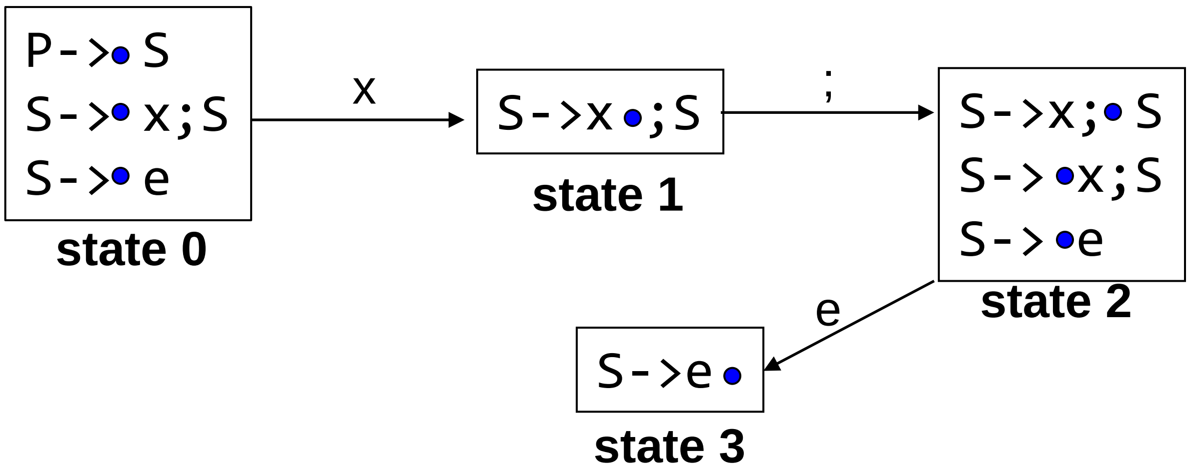
Grammar

P->S

S->x;S

S->e

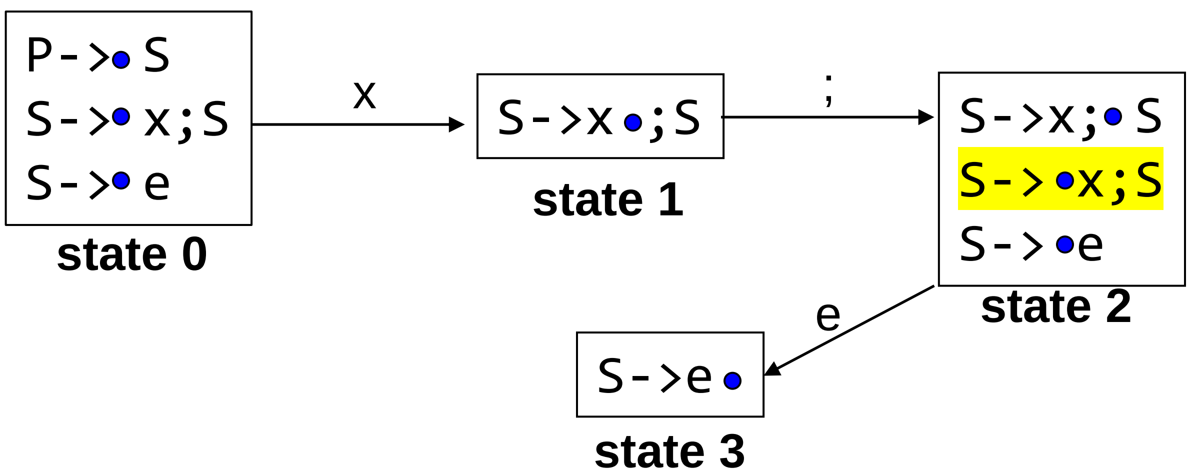
Compute successor (of state 2) under symbol e



Consider items (in state 2), where e is to the immediate right of Dot. Advance Dot by one symbol. No more items to be added. Becomes another state in CFSM.

Example: CFSM

Compute successor (of state 2) under symbol x



Grammar

P->S

S->x;S

S->e

Consider items (in state 2), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

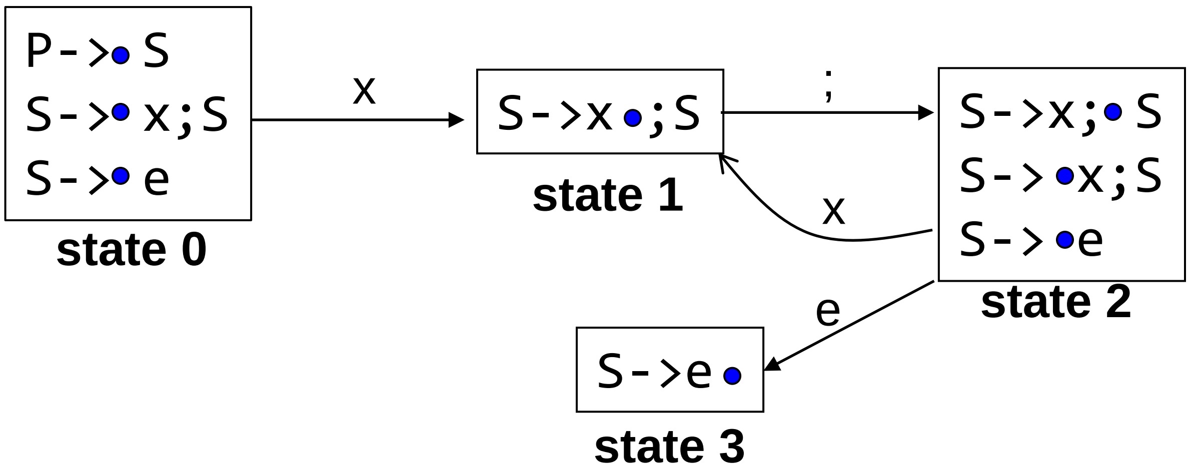
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol x



Consider items (in state 2), where x is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

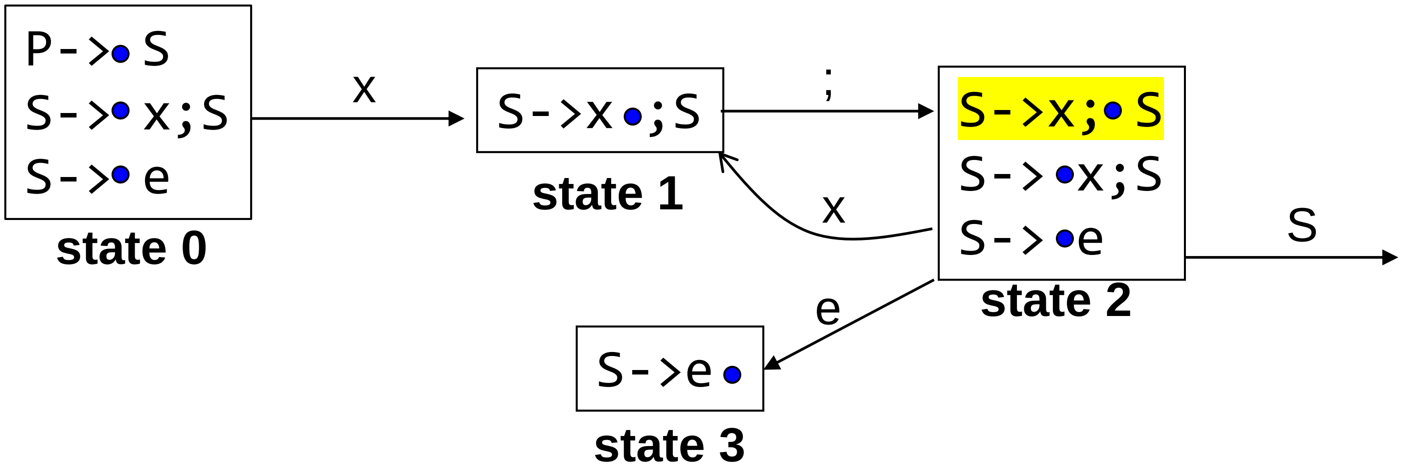
Grammar

P->S

S->x;S

S->e

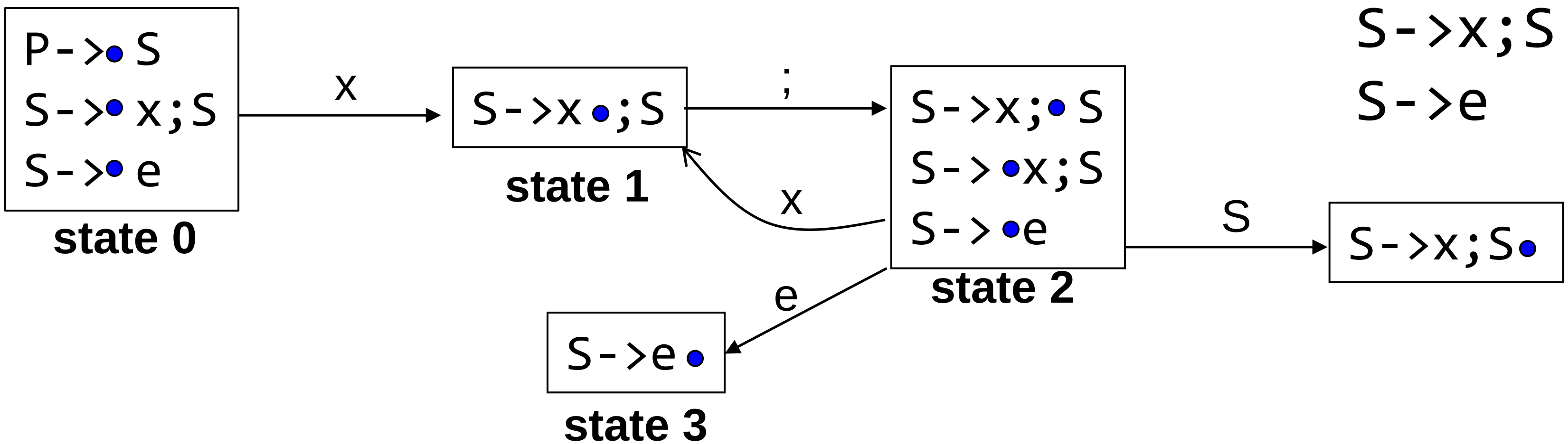
Compute successor (of state 2) under symbol S



Consider items (in state 2), where S is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Compute successor (of state 2) under symbol S



Grammar

P->S

S->x;S

S->e

Consider items (in state 2), where S is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

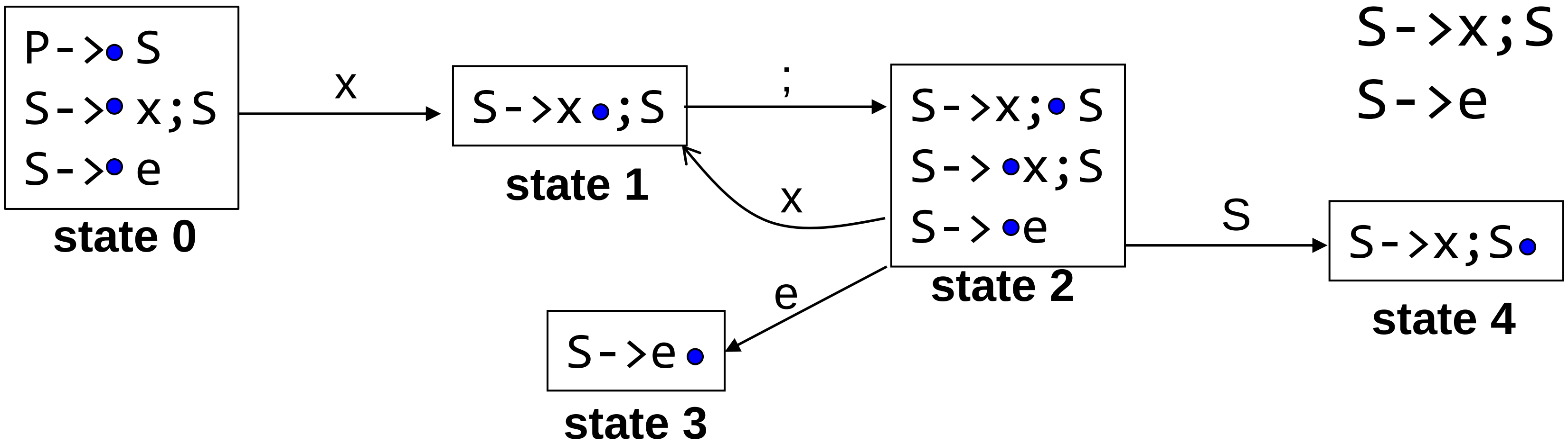
Grammar

P->S

S->x;S

S->e

Compute successor (of state 2) under symbol S



Consider items (in state 2), where S is to the immediate right of Dot. Advance Dot by one symbol. No more items to be added. Becomes another state in CFSM.

Example: CFSM

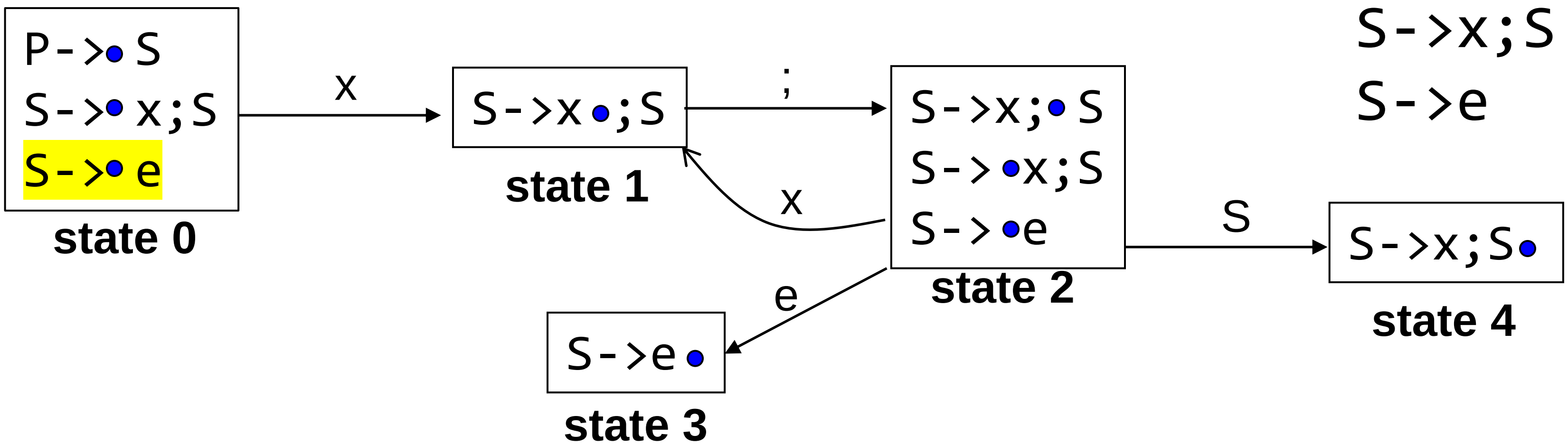
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol e



Consider items (in state 0), where e is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

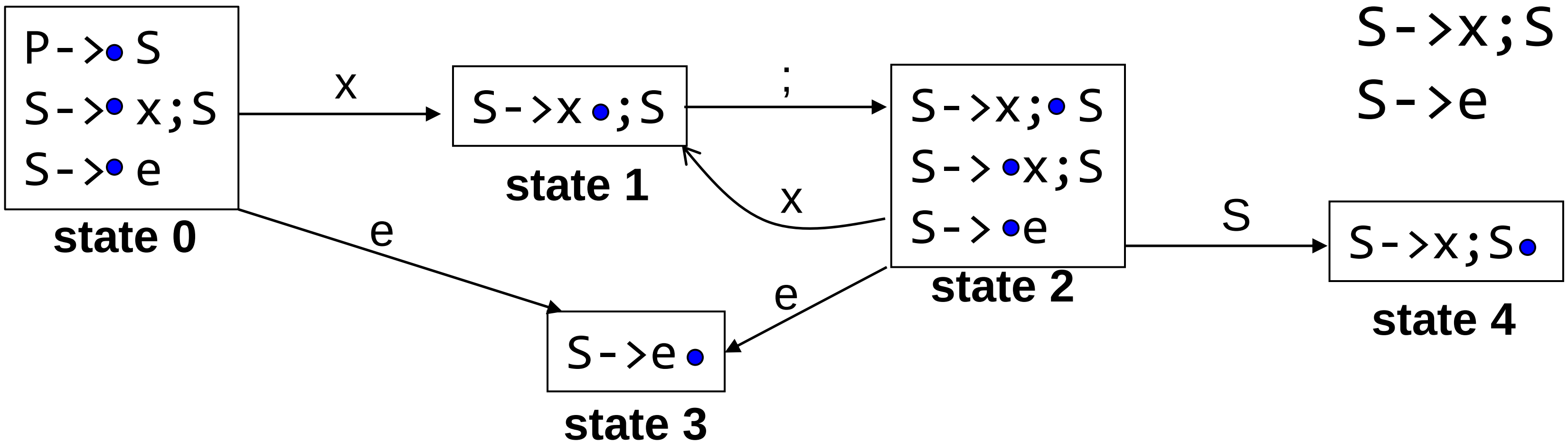
Grammar

P->S

S->x;S

S->e

Compute successor (of state 0) under symbol e



Consider items (in state 0), where e is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

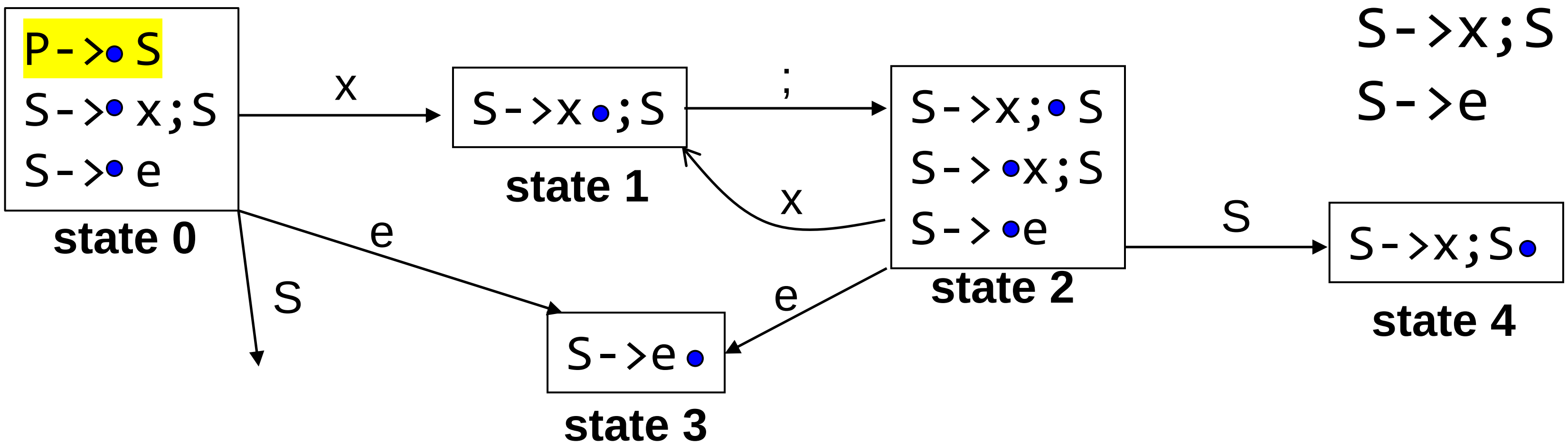
Grammar

P->S

S->x;S

S->e

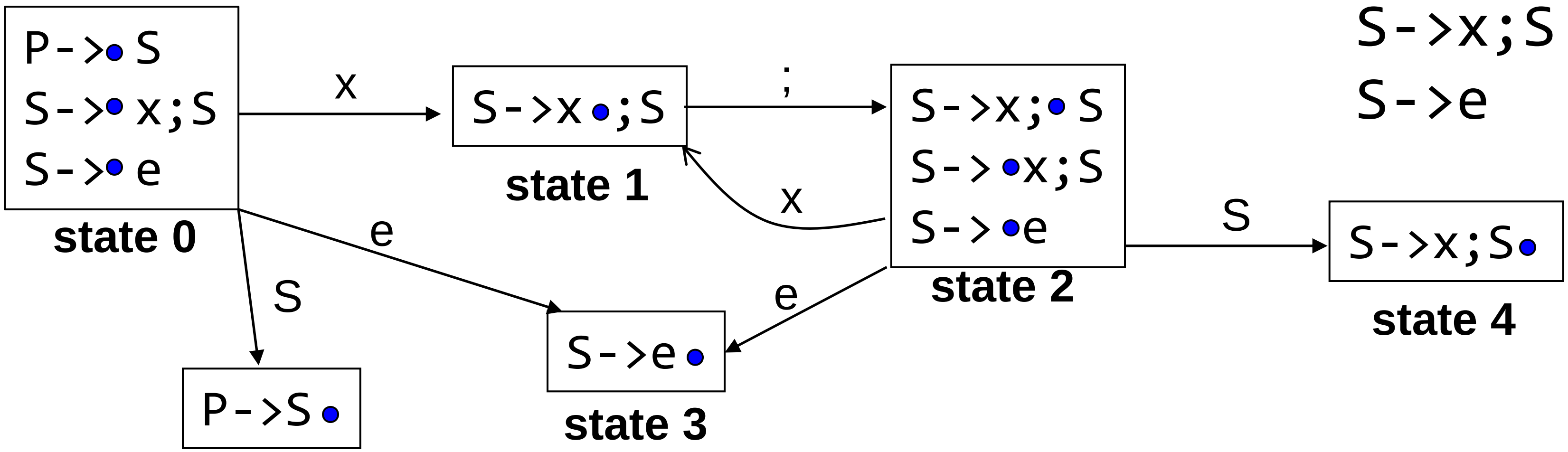
Compute successor (of state 0) under symbol S



Consider items (in state 0), where S is to the immediate right of Dot. Advance Dot by one symbol.

Example: CFSM

Compute successor (of state 0) under symbol S



Grammar

P->S

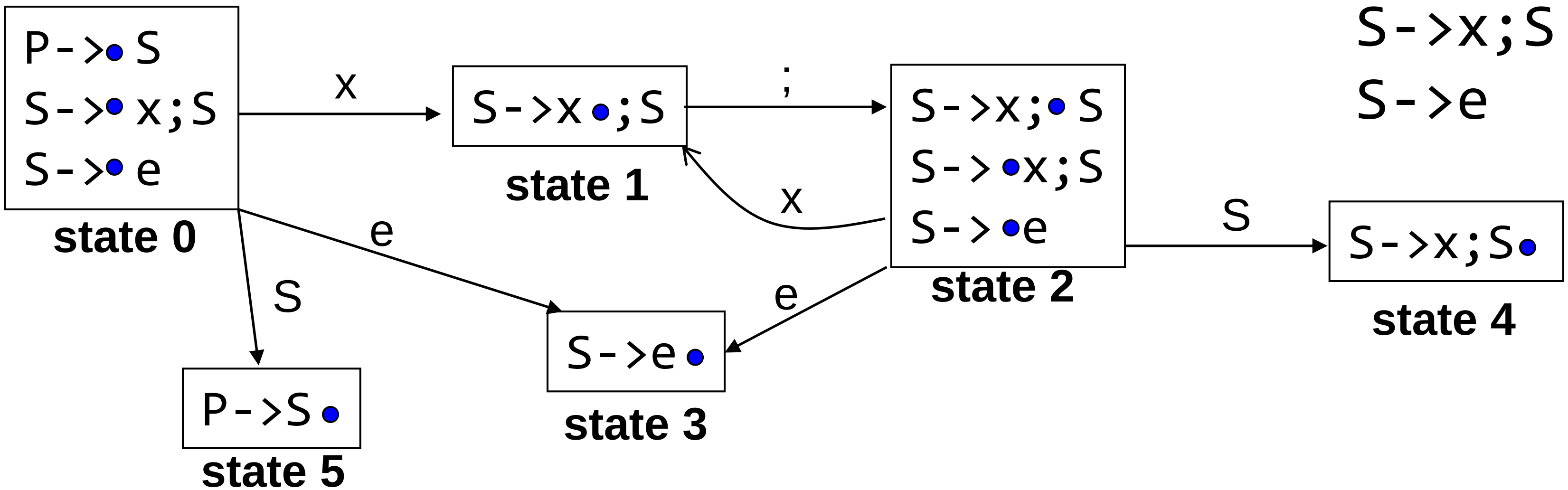
S->x;S

S->e

Consider items (in state 0), where S is to the immediate right of Dot.
Advance Dot by one symbol.

Example: CFSM

Compute successor (of state 0) under symbol S



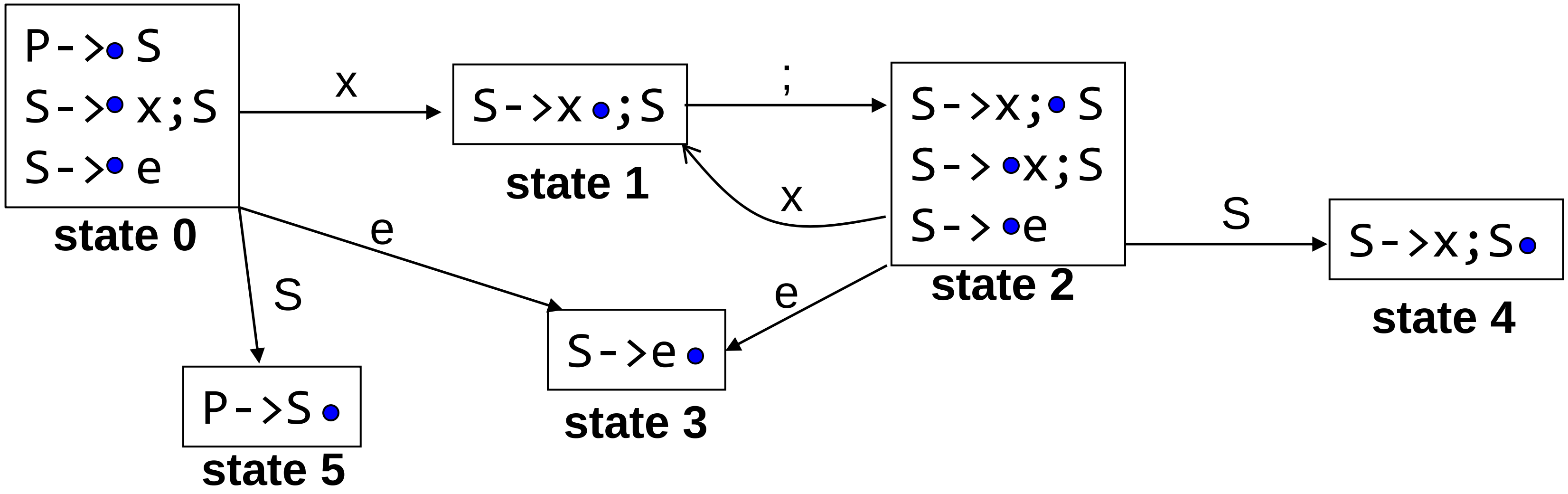
Grammar

P->S
S->x;S
S->e

Consider items (in state 0), where S is to the immediate right of Dot. Advance Dot by one symbol. Cannot expand CFSM anymore.

Example: CFSM

- All states with Dot at extreme right become *reduce* states



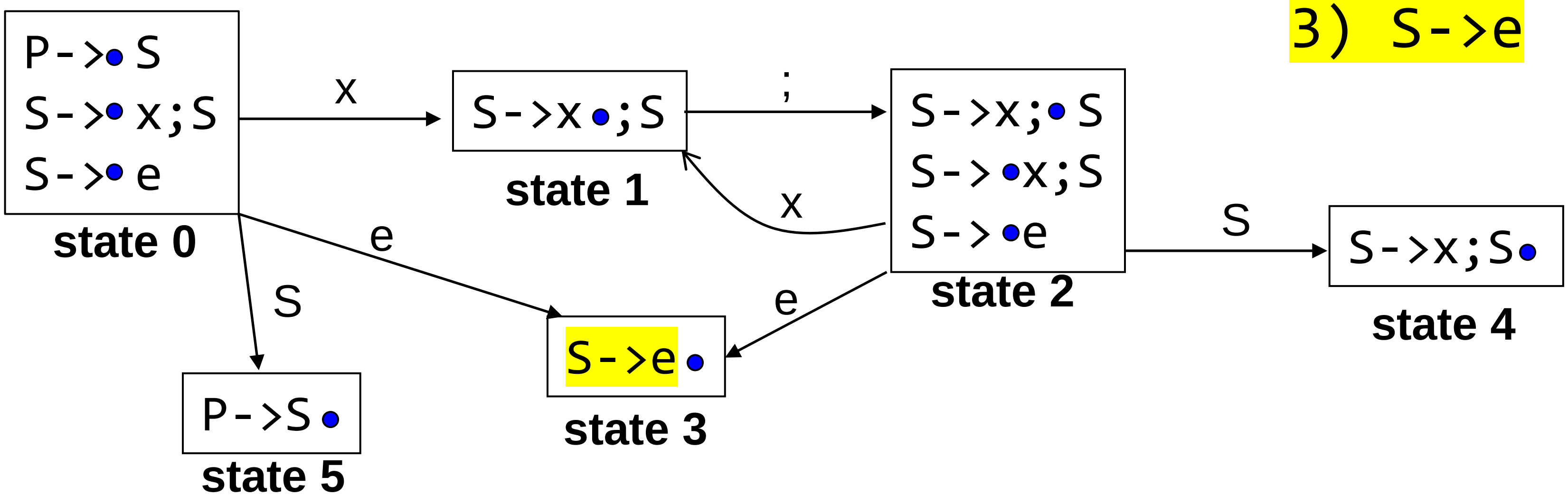
Example: CFSM

- All states with Dot at extreme right become *reduce* states

Grammar

1) P->S
2) S->x;S
3) S->e

Reduce 3



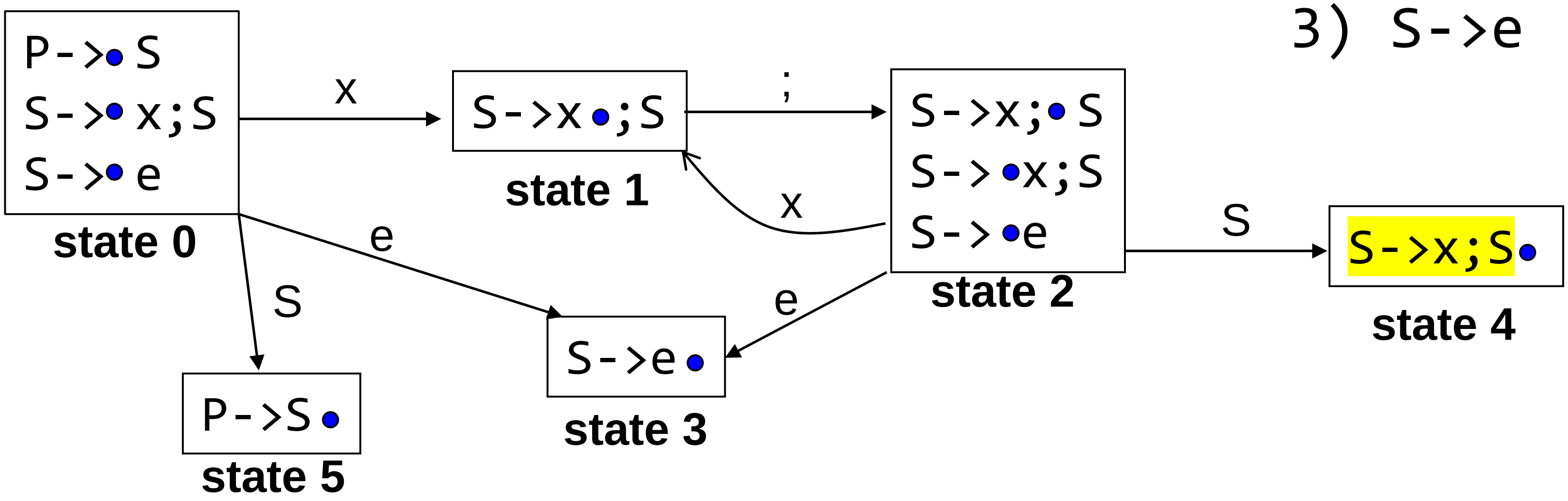
Example: CFSM

- All states with Dot at extreme right become *reduce* states

Reduce 2

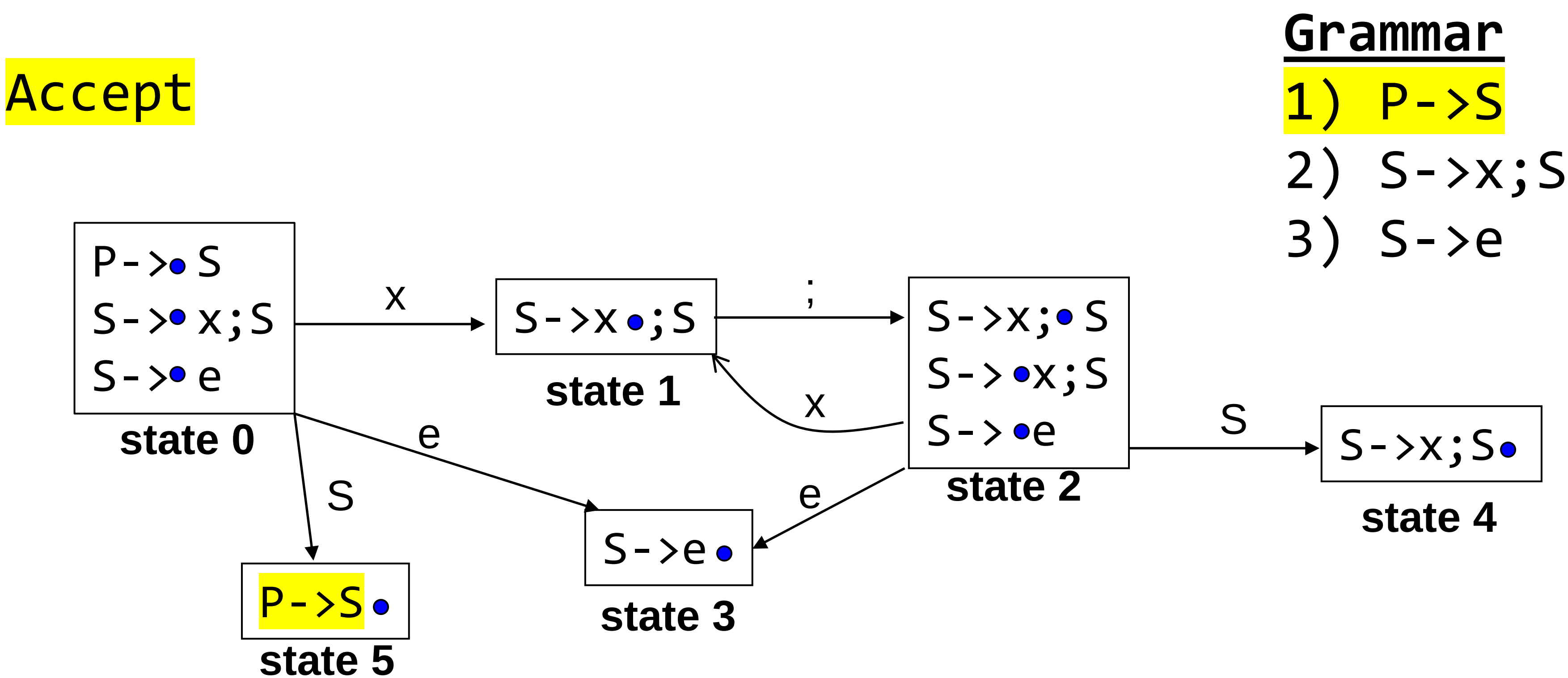
Grammar

1) P->S
2) S->x;S
3) S->e



Example: CFSM

- All states with Dot at extreme right become *reduce* states

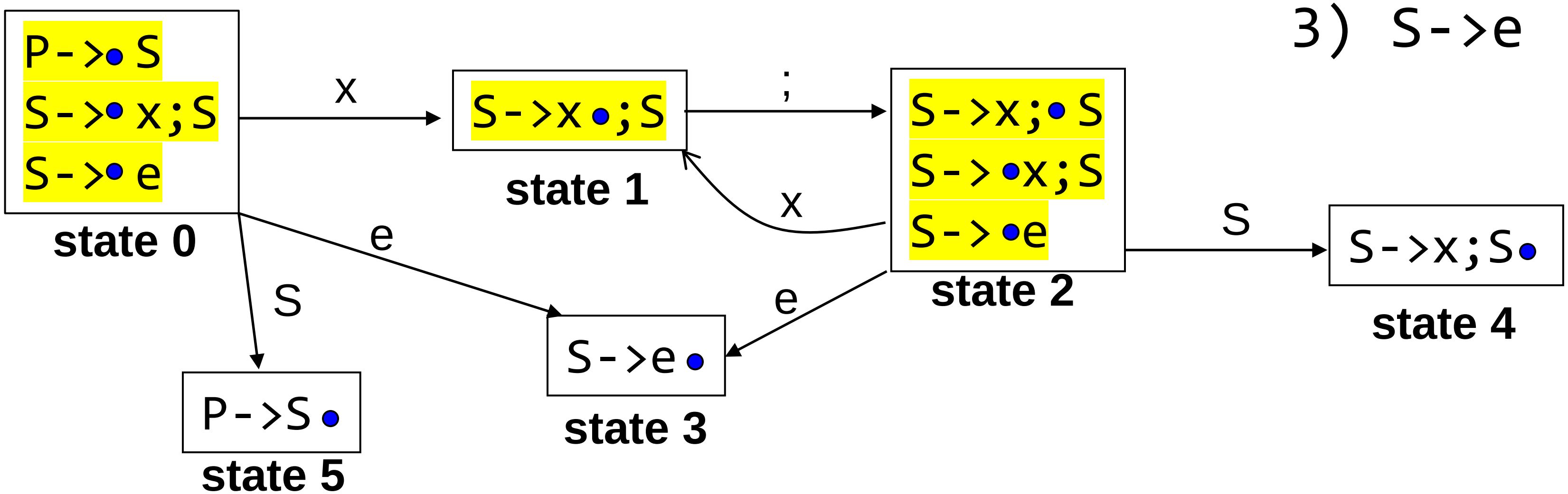


Example: CFSM

- Remaining states become *shift* states

Grammar

1) P->S
2) S->x;S
3) S->e



Conflicts

- What happens when a state has Dot at the extreme right for one item and in the middle for other items?

  Shift-reduce conflict

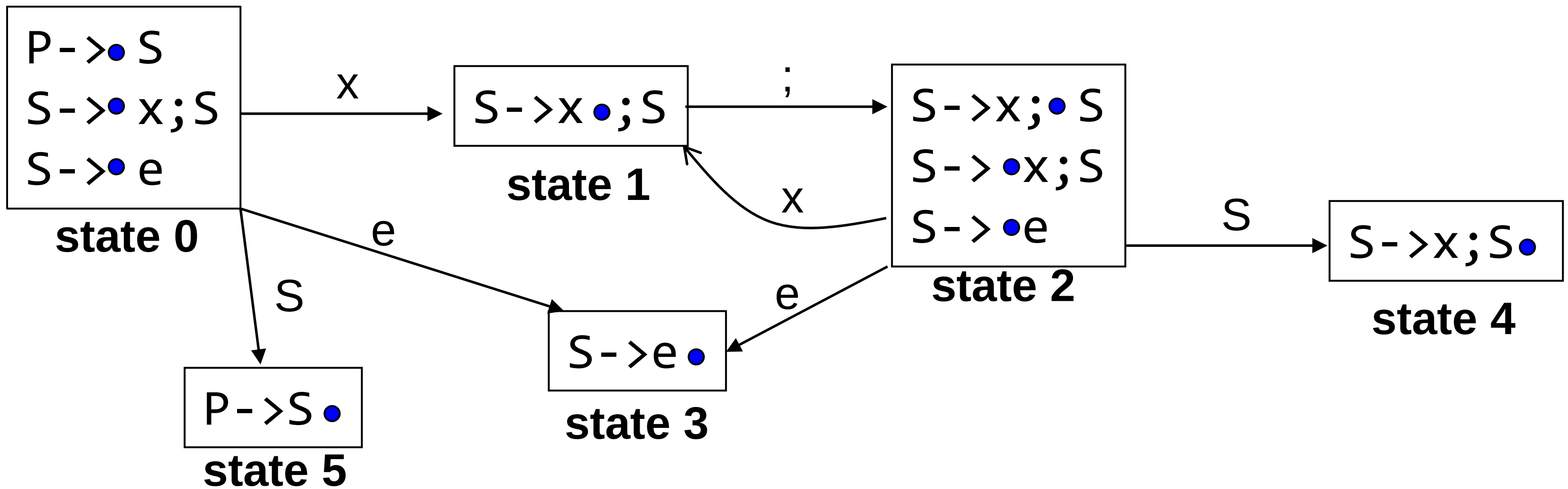
  Parser is unable to decide between shifting and reducing

- When Dot is at the extreme right for more than one items?

  Reduce-Reduce conflict

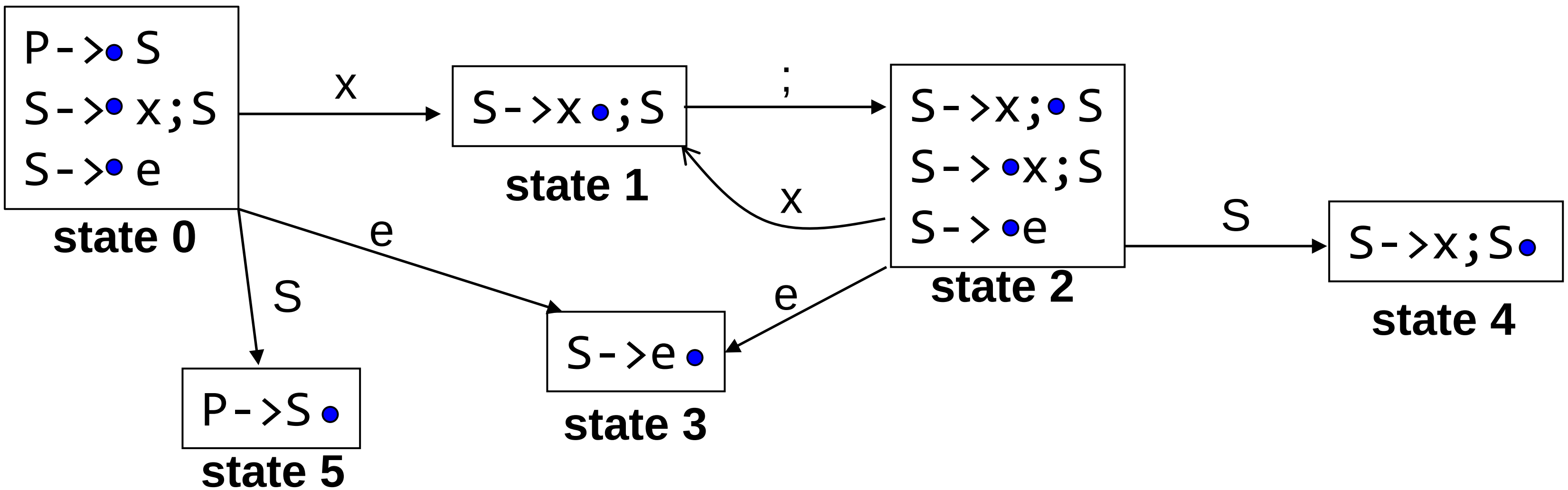
  Parser is unable to decide between which productions to choose for reducing

Example: goto table



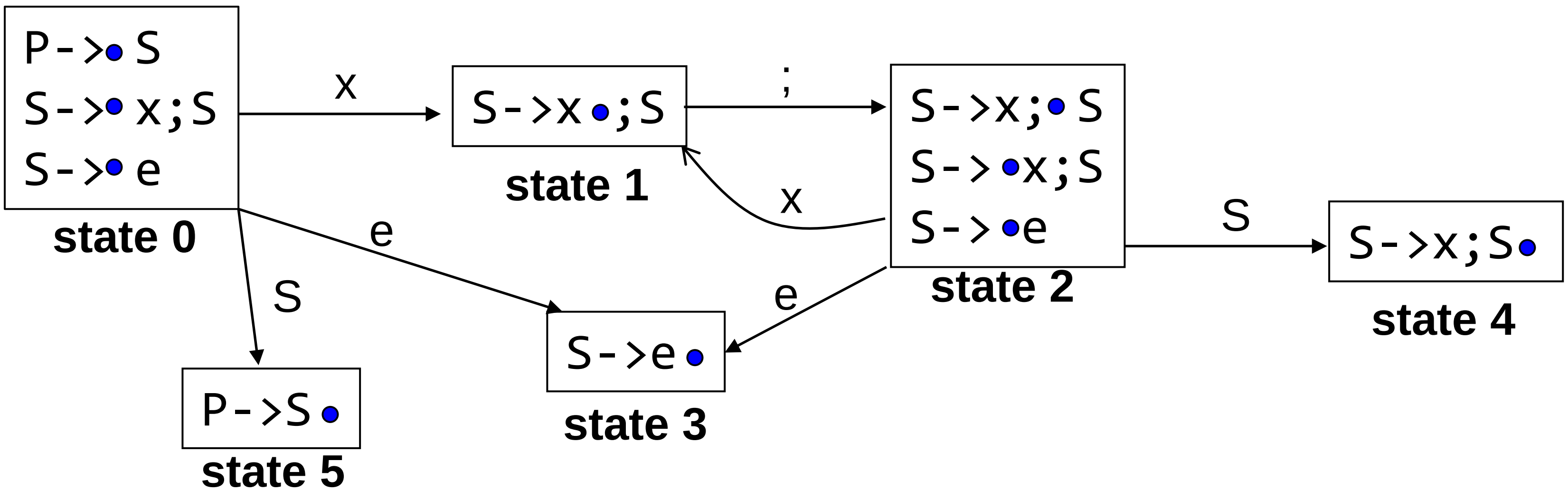
- construct transition table from CFSM.
  - Number of rows = number of states
  - Number of columns = number of symbols

Example: goto table



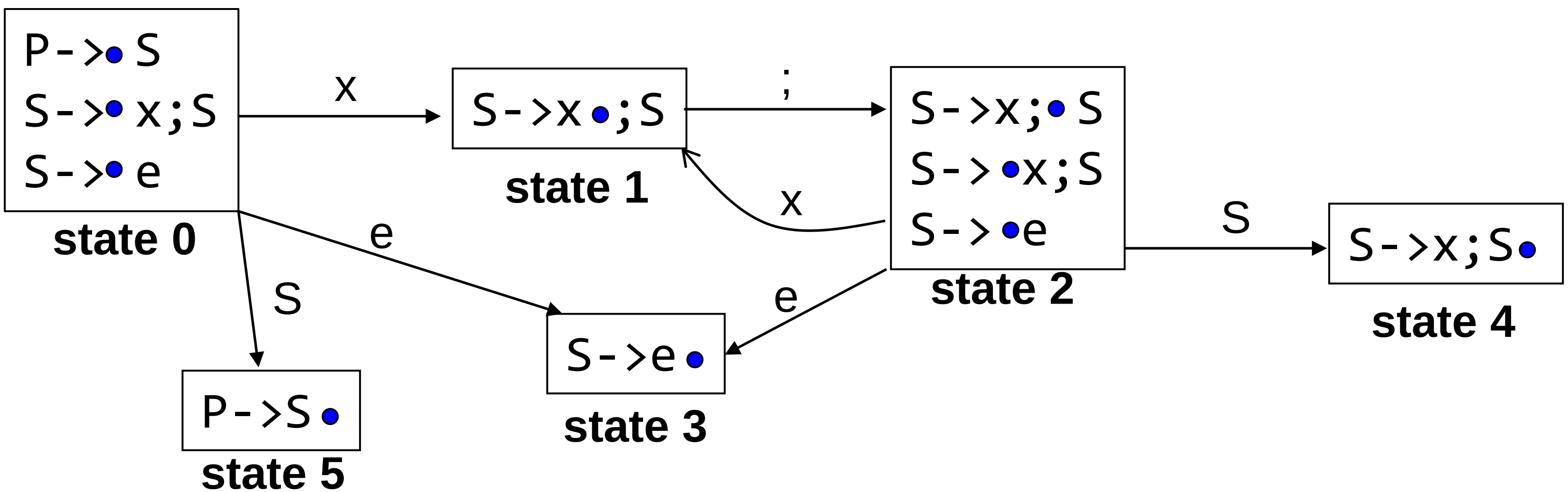
state	x	;	e	P	S
0	1		3		5
1		2			
2	1		3		4
3					
4					
5					

Example: action table



state	x
0	Shift
1	Shift
2	Shift
3	Reduce 3
4	Reduce 2
5	Accept

Example: action table



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

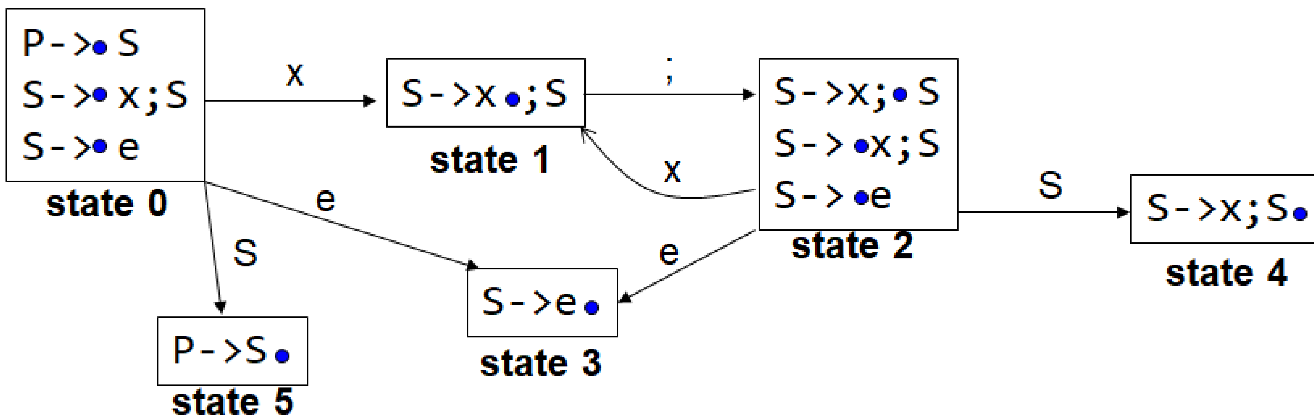
LR(0) Parsing

- Previous Example of LR Parsing was LR(0)
  - No (0) lookahead involved
  - Operate based on the parse stack state and with goto and action tables (How?)

LR(0) Parsing

- Assume: Parse stack contains α == saying that a e.g. prefix of `x;x` is seen in the input string

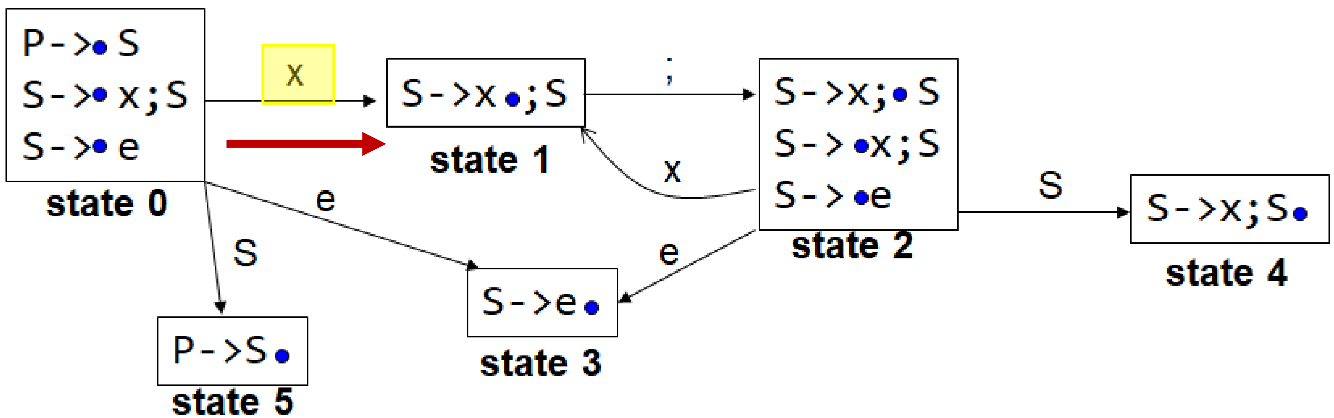
Parse Stack
0
0 1
0 1 2
0 1 2 1



LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1

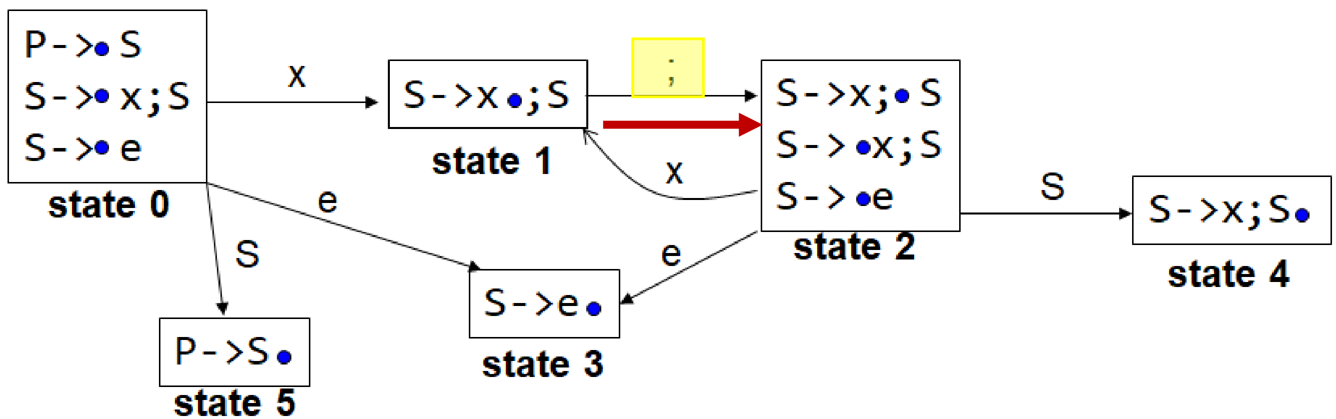


Go from state 0 to state 1 consuming x

LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of `x;x` is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1

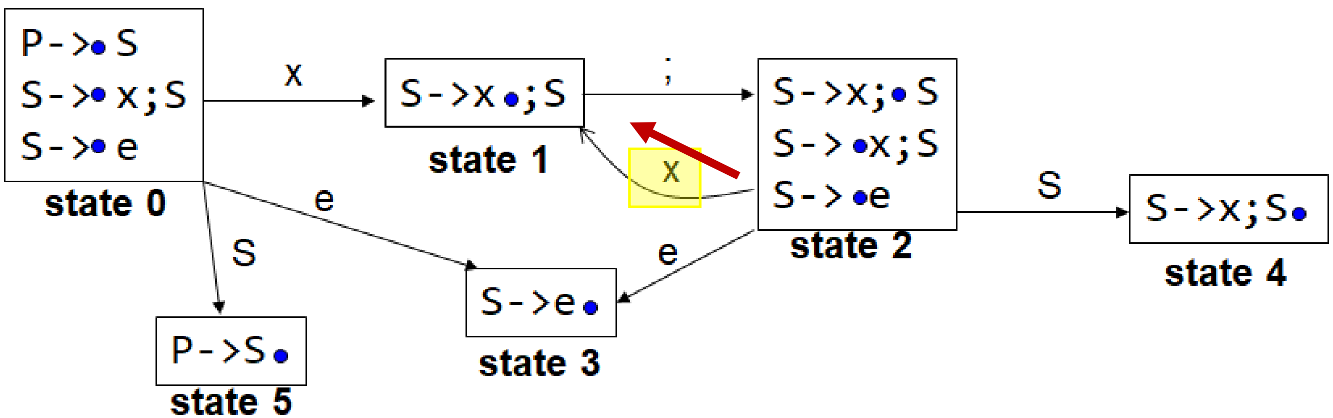


Go from state 1 to state 2 consuming ;

LR(0) Parsing

- Assume: Parse stack contains α == saying that a prefix of x;x is seen in the input string

Parse Stack
0
0 1
0 1 2
0 1 2 1



Go from state 2 to state 1 consuming x

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state s

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state s.

We reduce by X->β if state s contains X->β●

- Note: reduction is done based solely on the current state.

LR(0) Parsing

- Assume: Parse stack contains α.

=> we are in some state `s`.

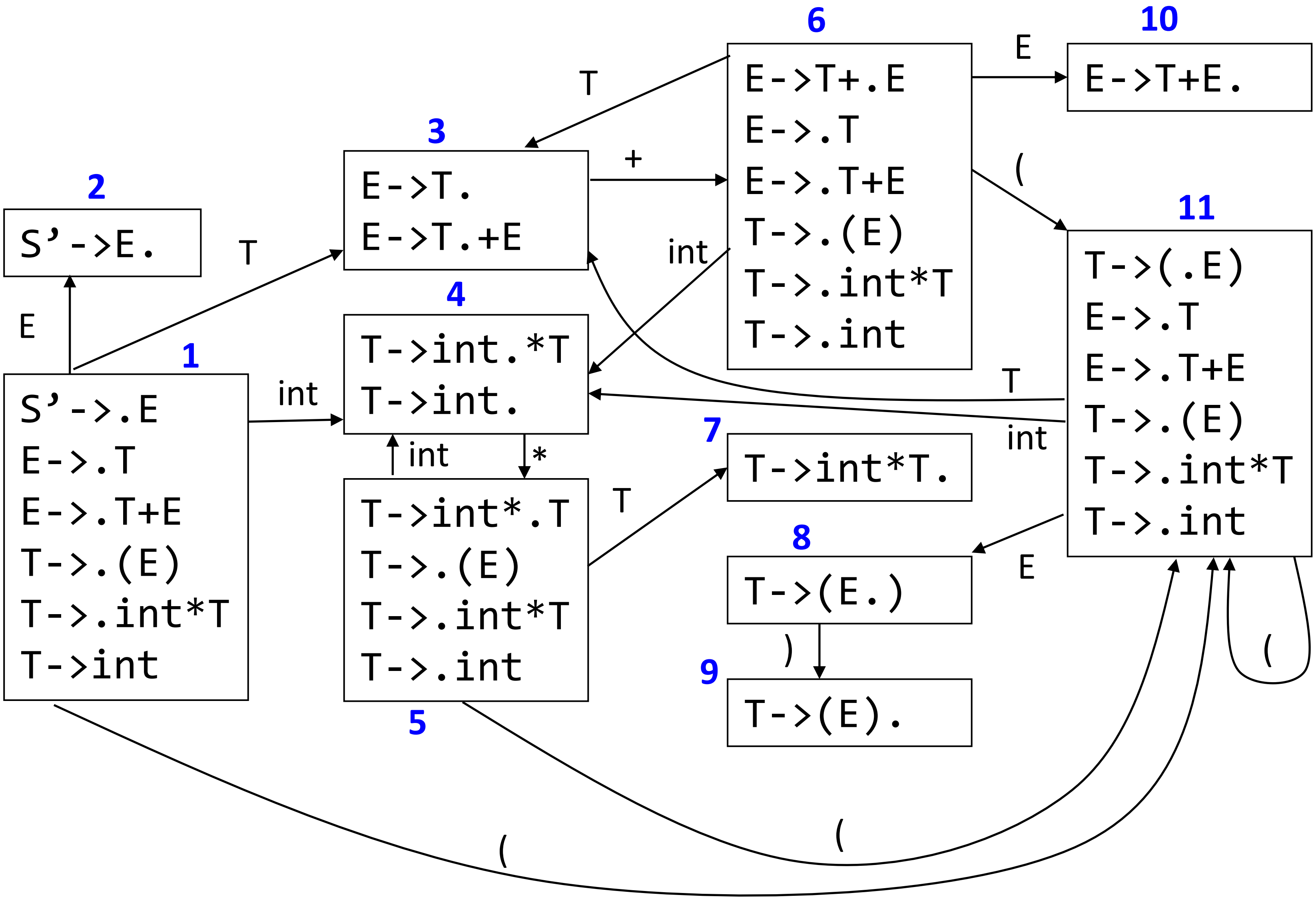
- Assume: Next input is t

We shift if s contains `X->β ● tω`

== `s` has a transition labelled `t`

LR(0) Parsing

- What if s contains $X \text{->} \beta \bullet t\omega$ and $X \text{->} \beta\bullet$?



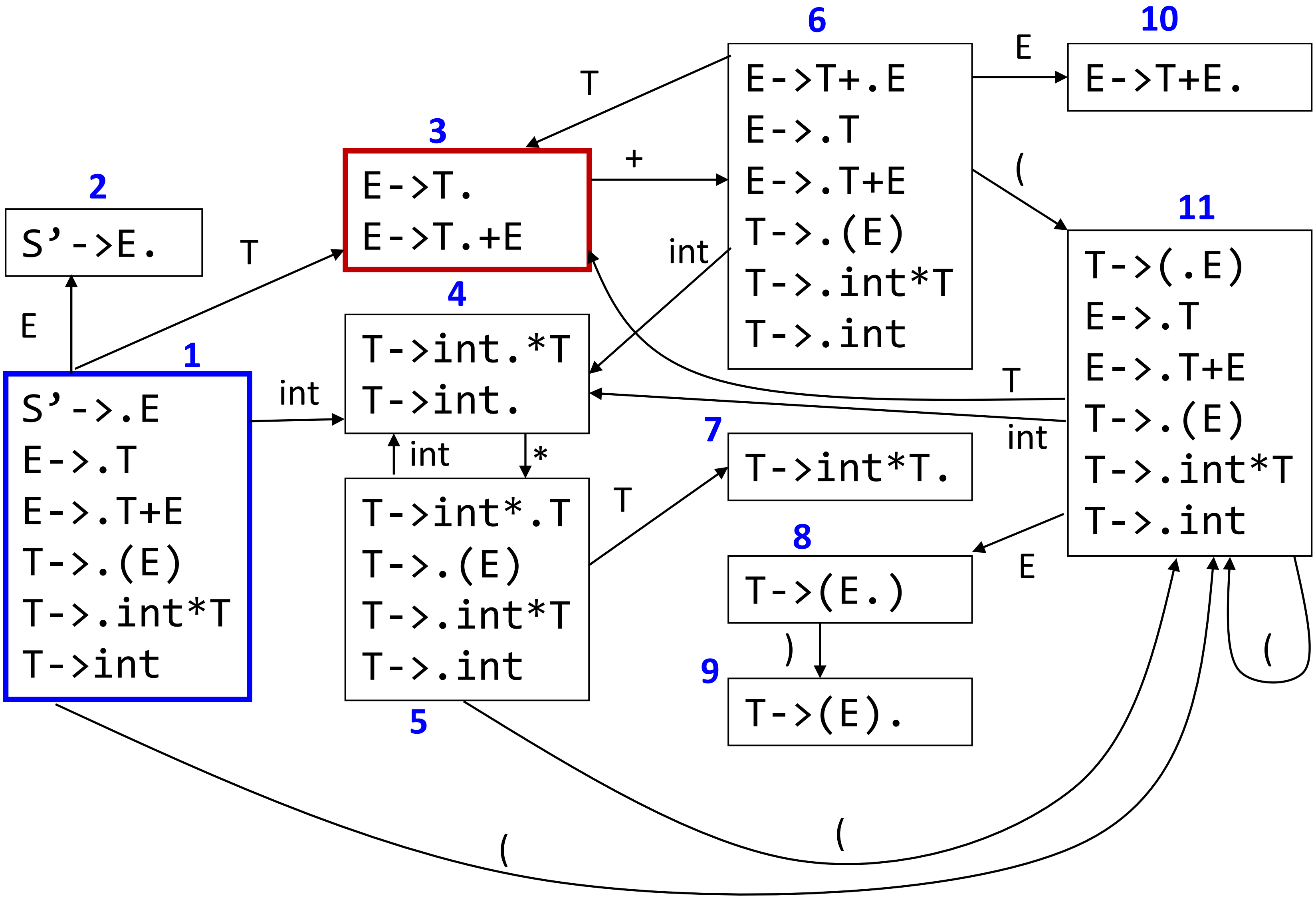
Conflicts or not?

SLR Parsing

- SLR Parsing improves the shift-reduce conflict states of LR(0):

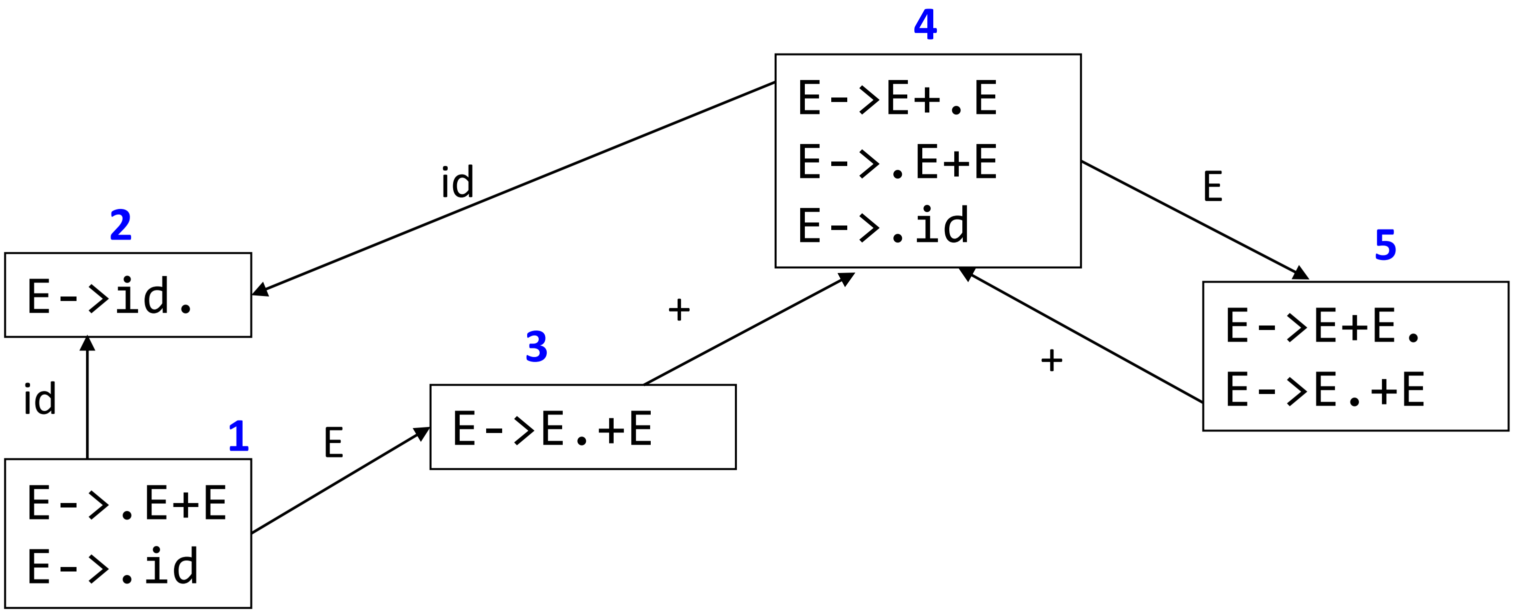
Reduce X->β ● only if

t ϵ Follow(X)



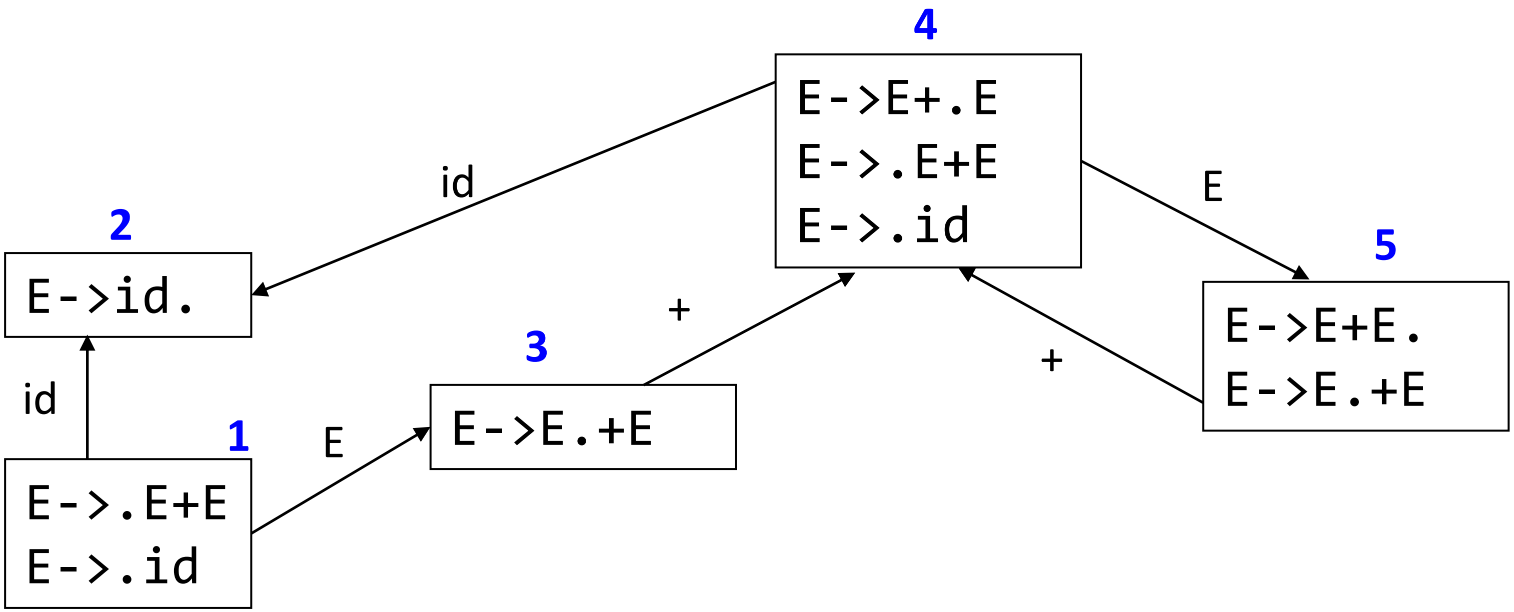
Follow(E) = { $,) } => reduce by E->T. only if <u>next input</u> is $ or)

lookahead 1



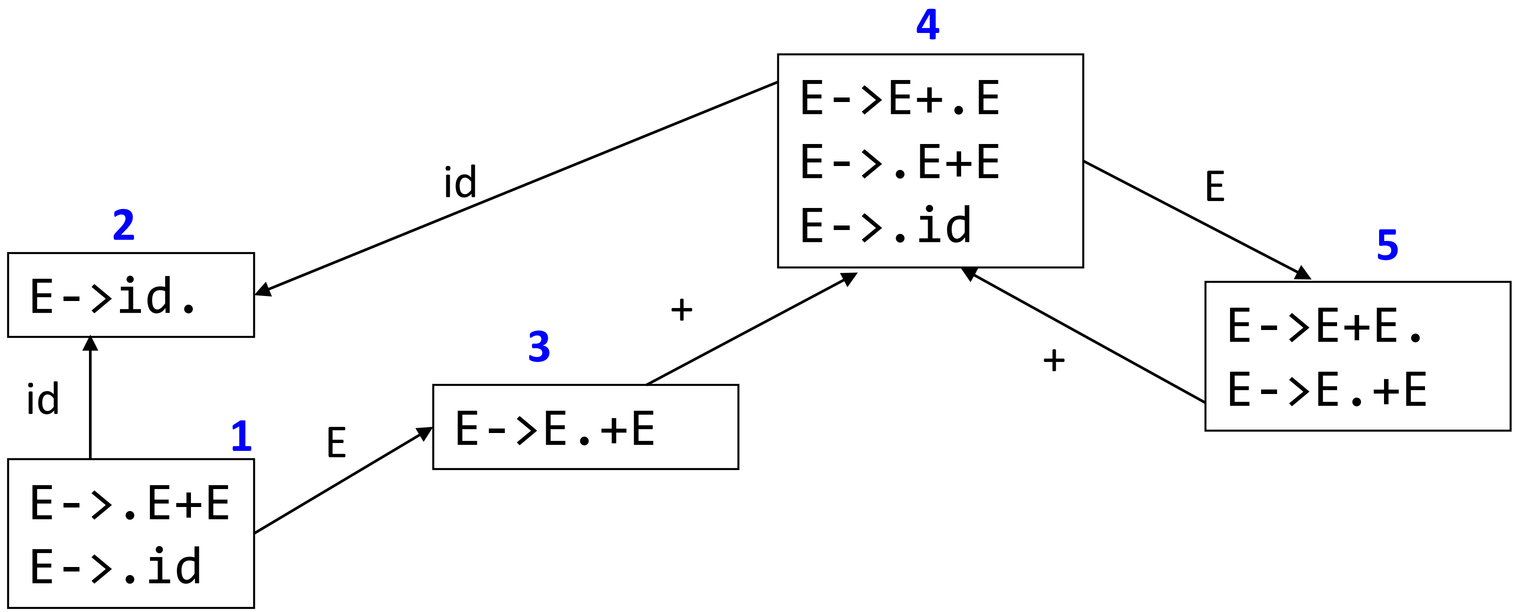
What about the grammar **E-> E + E | id** ?

LR(0)?



What about the grammar **E-> E + E | id** ?

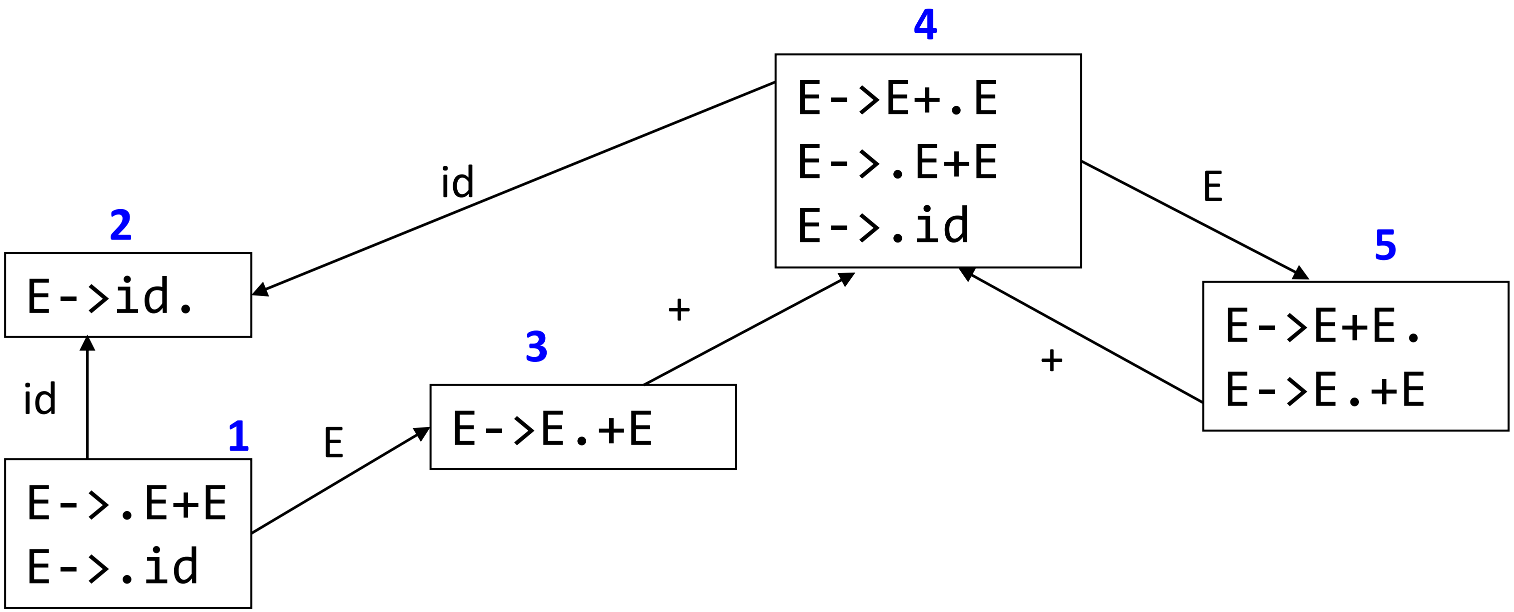
LR(0)? SLR(1)?



What about the grammar **`E-> E + E | id`** ?

`LR(0)? SLR(1)?`

`Follow(E) = {+,$} => in state 5,` reduce by E->E+E. only if <u>next input</u> is $ or +



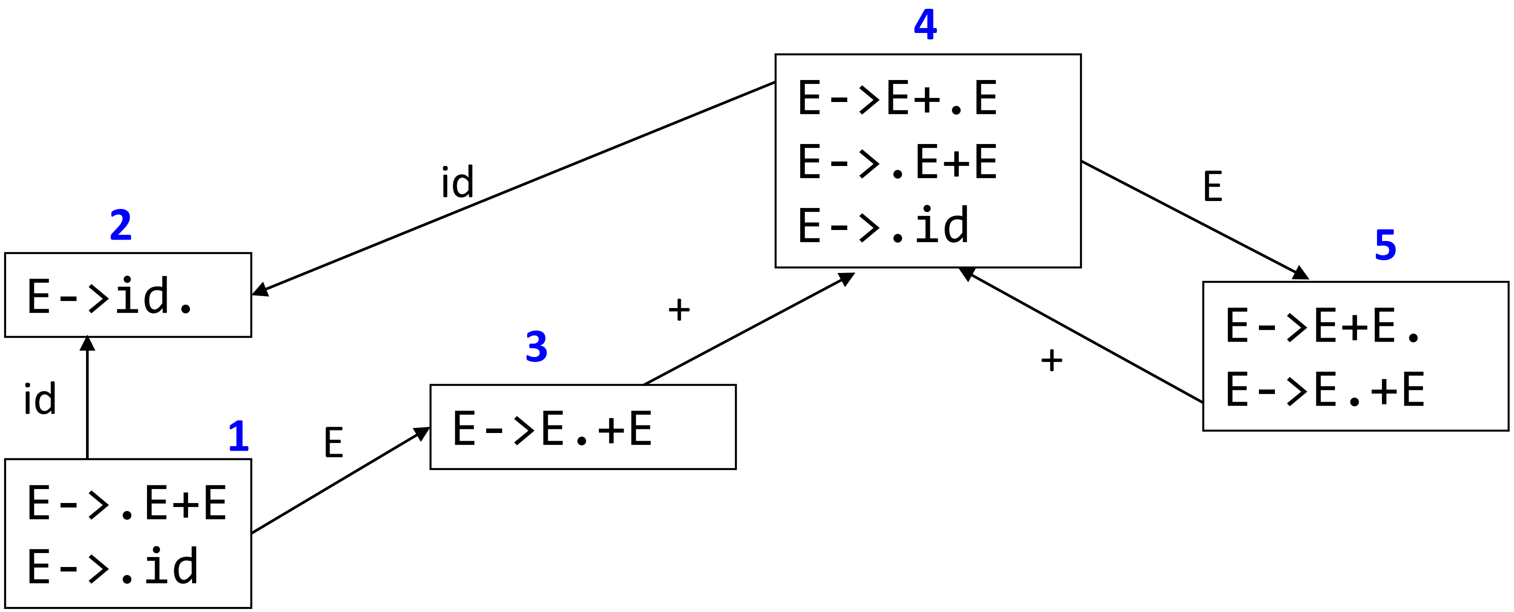
What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, <u>reduce by E->E+E. only if next input is $ or +</u>

But state 5 has E->E.+E (shift if next input is +)

Shift-reduce conflict!



What about the grammar **E-> E + E | id** ?

LR(0)? SLR(1)?

Follow(E) = {+,$} => in state 5, <u>reduce by E->E+E. only if next input is $ or +</u>

But state 5 has E->E.+E (shift if next input is +)

Shift-reduce conflict!

%left +

says reduce if the next input symbol is + i.e. prioritize rule E+E. over E.+E

Discussion: LR and LL Parsers

- LR Parsers:
  - For the next token, `t,` in input sequence, LR parsers try to answer: i) should I put this token on stack? or ii) should I replace a set of tokens that are at the top of a stack?

    In shift states (case i), if there is no transition out of that state for `t`, it is a syntax error.
- LL Parsers:
  - LL parsers ask the question: which rule should I use next based on the next input token `t`?. Only after expanding all non-terminals of the rule considered, they move on to consume the subsequent input tokens

Discussion: LR and LL Parsers

Parse Table (Top-Down)


```
Grammar:
1: S -> F
2: S -> (S + F)
3: F -> a

input:
(a+)

Accepted or Not
accepted?
```

Discussion: LR and LL Parsers

Goto and Action Table?


```
Grammar:
1: S -> F
2: S -> (S + F)
3: F -> a

input:
(a+)

Accepted or Not
accepted?
```

Hand-Written Parser - FPE

- Fully parenthesized expression (FPE)
  - Expressions (algebraic notation) are the normal way we are used to seeing them. E.g. 2 + 3
  - *Fully-parenthesized* expressions are simpler versions: every binary operation is enclosed in parenthesis
    - E.g. 2 + 3 is written as (2+3)
    - E.g. (2 + (3 * 7))
    - We can ignore order-of-operations (PEMDAS rule) in FPEs.

FPE – definition

- Either a:
  1. A number (integer in our example) OR
  2. *Open parenthesis* ‘(‘ followed by

     FPE followed by

     an operator (‘+’, ‘-’, ‘*’, ‘/’) followed by

     FPE followed by

     closed parenthesis ‘)’

FPE – Notation

```
1. E -> INTLITERAL
2. E -> (E op E)
3. op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function checks if the next token returned by the scanner matches the expected token. Returns `true` *if match.* `false` *if no match.*

Assume that a scanner module has been provided. The scanner has one function, `GetNextToken`, that returns the next token in the sequence.

Can be any one of: INTLITERAL, LPAREN, RPAREN, ADD, SUB, MUL, DIV

```
bool IsTerm(Scanner* s, TOKEN tok) {
    return s->GetNextToken() == tok;
}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #1: E->INTLITERAL
Returns `true` *if the next token returned by the scanner is an INTLITERAL.* `false` *otherwise.*

```
bool E1(Scanner* s) {

   return IsTerm(s, INTLITERAL);

}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #2: `E->(E op E)`
Returns `true` *if the Boolean expression on line 2 returns* `true`*.* `false` *otherwise.*

```
1: bool E2(Scanner* s) {

2:   return IsTerm(s, LPAREN) &&
             E(s)              &&
             OP(s)             &&
             E(s)              &&
             IsTerm(s, RPAREN);

3: }
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements production #3: `op->ADD|SUB|MUL|DIV`
Returns `true` *if the next token returned by the scanner is any one from* `ADD, SUB, MUL, DIV`*.* `false` *otherwise.*

```
bool OP(Scanner* s) {

   TOKEN tok = s->GetNextToken();

   if((tok == ADD) || (tok == SUB) || (tok == 
   MUL) || (tok == DIV))
      return true;

   return false;

}
```

Implementing a parser for FPE

1. One function defined for every non-terminal
   - `E, op`
2. One function defined for every production
   - `E1, E2`
3. One function defined for all terminals
   - `IsTerm`

```
1.E -> INTLITERAL
2.E -> (E op E)
3.op -> ADD | SUB | MUL | DIV
```

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

Assume that GetCurTokenSequence returns a reference to the first token in a sequence of tokens maintained by the scanner

```
bool E(Scanner* s) {

    TOKEN* prevToken = s->GetCurTokenSequence();
    if(!E1(s)) {
        s->SetCurTokenSequence(prevToken);
        return E2(s);
    }
    return true;

}
```

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

    TOKEN* prevToken = s->GetCurTokenSequence();
    if(!E1(s)) {
        s->SetCurTokenSequence(prevToken);
        return E2(s);
    }
    return true;

}
```

//This line implements the check to see if the sequence of tokens match production #1: E->INTLITERAL.

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

    TOKEN* prevToken = s->GetCurTokenSequence();
    if(!E1(s)) {
        s->SetCurTokenSequence(prevToken);
        return E2(s);
    }
    return true;
```

//because E1(s) calls s->GetNextToken() internally, the reference to the sequence of tokens would have moved forward. This line restores the reference back to the first node in the sequence so that the scanner provides the correct sequence to the call E2 in next line

Implementing a parser for FPE

This function implements the routine for matching non-terminal E

```
bool E(Scanner* s) {

   TOKEN* prevToken = s->GetCurTokenSequence();
   if(!E1(s)) {
      s->SetCurTokenSequence(prevToken);
      return E2(s);
   }
   return true;
```

//This line implements the check to see if the sequence of tokens match production #2: E->(E op E)

Implementing a parser for FPE

```
IsTerm(Scanner* s, TOKEN tok) { return s->GetNextToken() == tok;}

bool E1(Scanner* s) {
     return IsTerm(s, INTLITERAL);
}

bool E2(Scanner* s) { return IsTerm(s, LPAREN) && E(s) && OP(s) && E(s) && IsTerm(s, RPAREN); }

bool OP(Scanner* s) {
     TOKEN tok = s->GetNextToken();
     if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
          return true;
     return false;
}

bool E(Scanner* s) {
     TOKEN* prevToken = s->GetCurTokenSequence();
     if(!E1(s)) {
          s->SetCurTokenSequence(prevToken);
          return E2(s);
     }
     return true;
}
```

Start the parser by invoking E().
Value returned tells if the expression is FPE or not.

Exercise

- What parsing technique does this parser use?

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead *k* symbols
  - Most powerful class of deterministic bottom-up parsers
  - LR(1) and variants are the most common parsers

slide courtesy: Milind Kulkarni

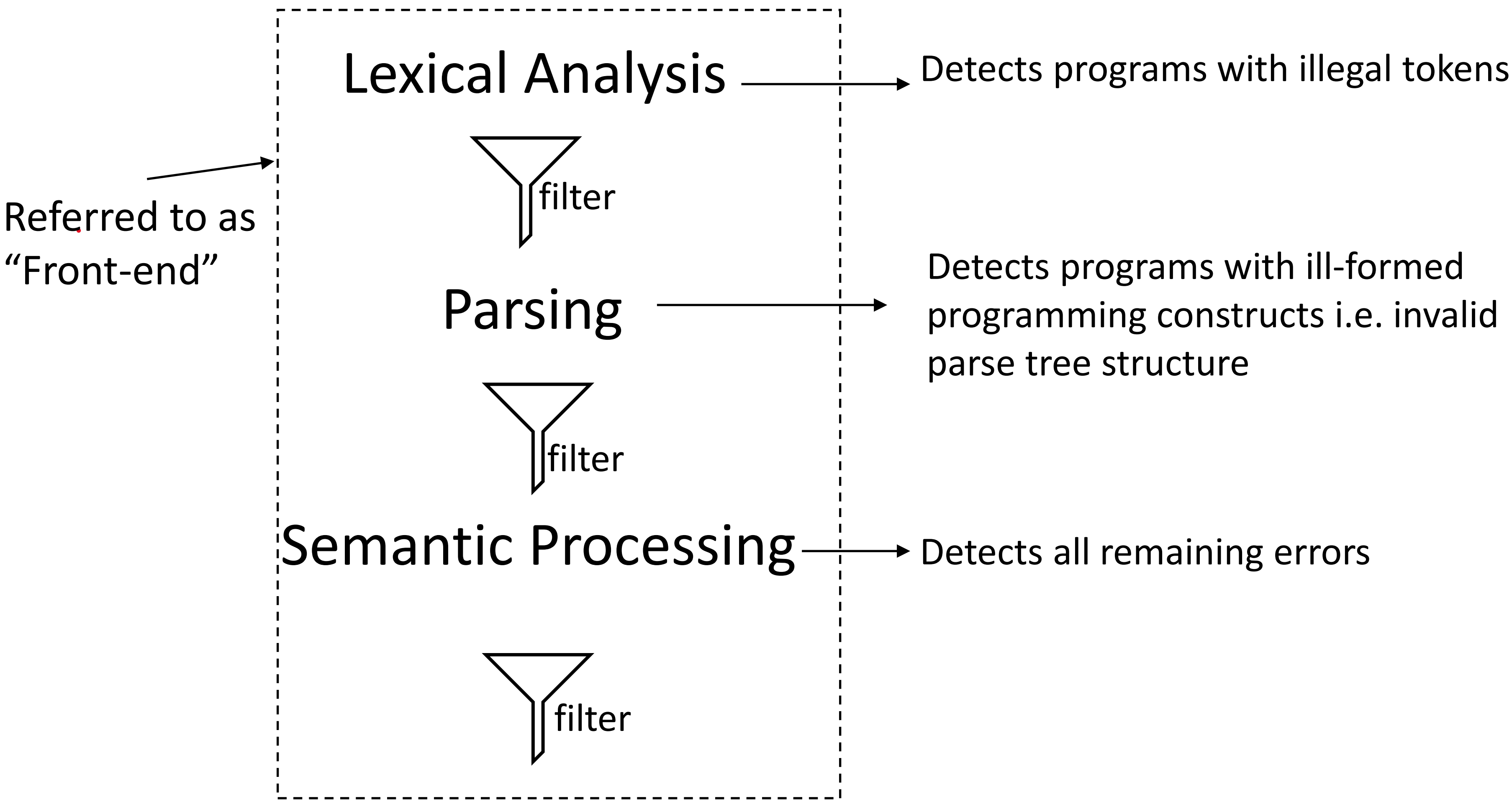
Top-down vs. Bottom-up parsers

- Top-down parsers expand the parse tree in *pre-order*
  - Identify parent nodes before the children
- Bottom-up parsers expand the parse tree in *post-order*
  - Identify children before the parents
- Notation:
  - LL(1): Top-down derivation with 1 symbol lookahead
  - LL(k): Top-down derivation with k symbols lookahead
  - LR(1): Bottom-up derivation with 1 symbol lookahead

Exercise

- https://forms.gle/gd8VwA9UyNaZiWKB8

Semantic Processing



Semantic Processing

- Syntax-directed / syntax-driven
  - Routines (called as semantic routines) interpret the meaning of programming constructs based on the syntactic structure

  - Routines play a dual role
    - Analysis – Semantic analysis
      - undefined vars, undefined types, uninitialized variables, type errors that can be caught at compile time, unreachable code, etc.
    - Synthesis – Generation of intermediate code
      - 3 address code

  - Routines create semantic records to aid the analysis and synthesis

Semantic Processing

- Syntax-directed translation: notation for *attaching* program fragments to grammar productions.
  - Program fragments are executed when productions are matched
  - The combined execution of all program fragments produces the translation of the program

    e.g. `E->E+T   { print('+') }`

`Output:` program fragments may create AST and 3 Address Codes

- Attributes: any 'quality' associated with a terminal and non-terminal e.g. type, number of lines of a code, first line of the code block etc.

Why Semantic Analysis?

- Context-free grammars cannot specify all requirements of a language
  - Identifiers declared before their use (scope)
  - Types in an expression must be consistent

    STRING str:= “Hello”;

    str := str + 2;
  - Number of formal and actual parameters of a function must match
  - Reserved keywords cannot be used as identifiers
  - A Class is declared only once in a OO language, a method of a class can be overridden.
  - …

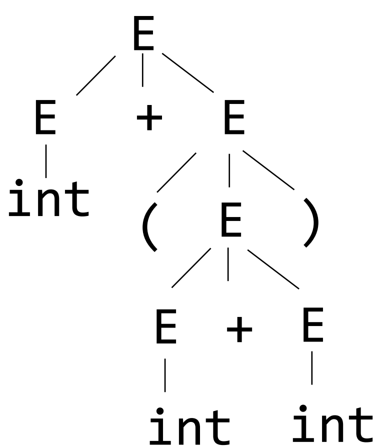
Abstract Syntax Tree

- Abstract Syntax Tree (AST) or Syntax Tree can be the input for semantic analysis.
  - What is Concrete Syntax Tree? – the parse tree
- ASTs are like parse trees but ignore certain details:

E.g. Consider the grammar:

```
E - > E + E
    | ( E )
    | int
```

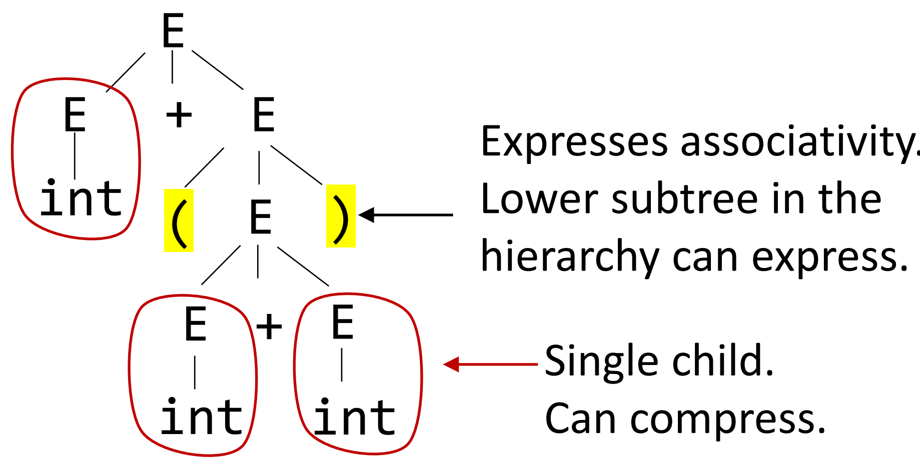
The parse tree for `1+(2+3)`



AST - Example

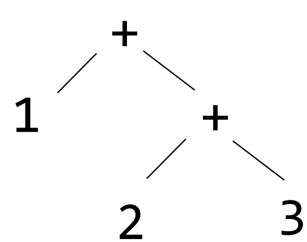
- Not all details (nodes) of the parse tee are helpful for semantic analysis

The *parse tree* for `1+(2+3)` :



We need to compute the result of the expression. So, a simpler structure is sufficient:

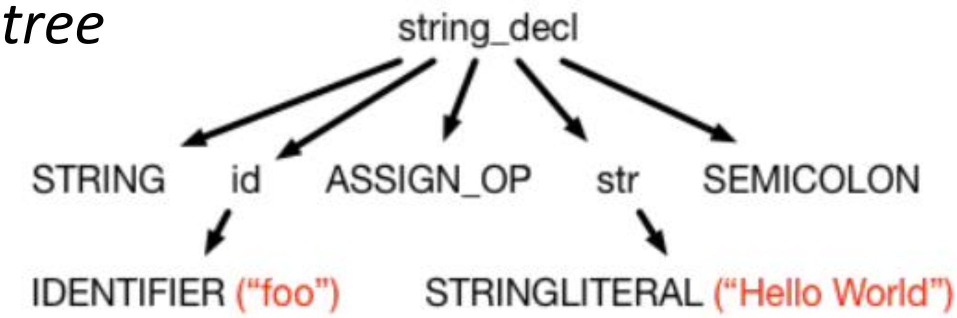
AST for `1+(2+3)`:



AST - Example

parse tree

`STRING foo := “Hello World” ;` ⇨



≡ *AST* ⇩

```
            ASSIGN_OP
           /         \
IDENTIFIER            STRINGLITERAL
(“foo”)               (“Hello World”)
```

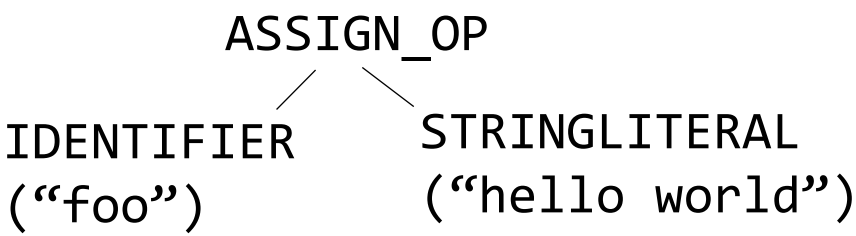
Semantic Analysis – Example

- Context-free grammars cannot specify all requirements of a language
  - Identifiers declared before their use (scope)
  - Types in an expression must be consistent

    Type checks
    STRING str:= “Hello”;
    str := str + 2;
  - Number of formal and actual parameters of a function must match
  - Reserved keywords cannot be used as identifiers
  - A Class is declared only once in a OO language, a method can be overridden.
  - …

Scope

- **Goal:** matching identifier declarations with uses
- Most languages require this!
- Scope confines the activity of an identifier



What if `foo` is declared as a STRING in an enclosing scope but is an INT in the current scope?

in different parts of the program:

- Same identifier may refer to different things
- Same identifier may not be accessible

Static Scope

- Most languages are statically scoped
  - Scope depends on only the program text (not runtime behavior)
  - A variable refers to the closest defined instance

```
INT w, x;
{
    FLOAT x, z;
    f(x, w, z);
}
g(x)
```

x is a FLOAT here

x is an INT here

Dynamic Scope

- In dynamically scoped languages
  - Scope depends on the execution context
  - A variable refers to the <u>closest enclosing binding in the execution</u> of the program

```
f(){
    a=4; g();
}
g() { print(a); }
```

value of a is 4 here

Exercise: Static vs. Dynamic Scope

```
#define a (x+1) //macro definition

int x = 2; //global var definition

//function b definition
void b() {
    int x = 1;
    printf("%d\n",a);
}

//function c definition
void c() {
    printf("%d\n",a);
}

//the main function
int main() { b(); c(); }
```

Is x statically scoped or dynamically scoped?

Symbol Table

- Data structure that tracks the bindings of identifiers. Specifically, returns the current binding.
  - E.g., stores a mapping of names to types
  - Should provide for efficient retrieval and frequent insertion and deletion of names.
  - Should consider scopes

```
{
    int x = 0;
    //accessing y here should be illegal
    {
        int y = 1;
    }
}
```

- Can use stacks, binary trees, hash maps for implementation

Symbol Table and Classes in OO Language

- Class names may be used before their definition
- Can’t use symbol table (to check class definition)
  - Gather all class names first.
  - Check bindings next.

  ⇦ Implies going over the program text multiple times
- Semantic analysis is done in multiple passes
- One of the goals of semantic analysis is to create/update data structures that help the next round of analysis

Semantic Analysis – How?

- Recursive descent of AST
  - Process a node, n
  - Recurse into children of n and process them
  - Finish processing the node, n

$\Rightarrow$Do a postorder processing of the AST

- As you visit a node, you will add information depending upon the analysis performed
  - The information is referred to as <u>attributes</u> of the node

Building AST - Example

- <u>Fully-Parenthesized Expressions (FPE)</u>
  - Can build while parsing via bottom-up building of the tree
  - Create subtrees, make those subtrees left- and right-children of a newly created root.
  - Need to modify the hand-written recursive parser:

    if:

    ➡ token == INTLITERAL, return a reference to newly created node containing a number

    else:

    store references to nodes that are left- and right- expression subtrees
    Create a new node with value = ‘OP’

Building AST - Example

This function creates an AST node and adds information that stores the value of an INTLITERAL in the node. A reference to the AST node is returned.

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
    TreeNode* ret = NULL;
    TOKEN nxtToken = s->GetNextToken();
    if(nxtToken == tok)
        ret = CreateTreeNode(nxtToken.val);
    return ret;
}
```

Building AST

E1 needs to change because IsTerm returns a TreeNode.*
E1 returns a TreeNode now.*

Recall: E1 is the function that gets called when predicting using the production: `E -> INTLITERAL`

```
TreeNode* E1(Scanner* s) {

   return IsTerm(s, INTLITERAL);

}
```

Building AST - Example

- Fully-Parenthesized Expressions (FPE)
  - Can build while parsing via bottom-up building of the tree
  - Create subtrees, make those subtrees left- and right-children of a newly created root.
  - Need to modify the hand-written recursive parser:

    if:

    token == INTLITERAL, return a reference to newly created node containing a number

    else:

    store references to nodes that are left- and right- expression subtrees
    Create a new node with value = ‘OP’

Building AST

This function creates an AST node and adds information that stores the value of an op in the node. A reference to the AST node is returned.

Recall: op is the function that gets called when predicting using the production:
`op -> ADD | SUB | MUL | DIV`

```
TreeNode* OP(Scanner* s, TOKEN tok) {
    TreeNode* ret = NULL;
    TOKEN tok = s->GetNextToken();
    if((tok == ADD) || (tok == SUB) || (tok ==
MUL) || (tok == DIV))
        ret = CreateTreeNode(tok.val);
    return ret;
}
```

Building AST

This function sets the references to left- and right- expression subtrees if those subtrees are valid FPEs. Returns reference to the AST node corresponding to the op value, NULL otherwise.

Recall: E2 is the function that gets called when predicting using the production: `E -> (E op E)`

```
TreeNode* E2(Scanner* s, TOKEN tok) {
   TOKEN nxtTok = s->GetNextToken();
   if(nxtTok == LPAREN) {
      TreeNode* left = E(s); if(!left) return NULL;
      TreeNode* root  = OP(s); if(!root) return NULL;
      TreeNode* right = E(s); if(!right) return NULL;
      nxtTok = s->GetNextToken();
      if(nxtTok != RPAREN); return NULL;
         //set left and right as children of root.
      return root;
}
```

Building AST

E needs to change because E1, E2, and OP return a TreeNode.*
E returns a TreeNode now.*

Recall: E is the higher-level function for a non-terminal that gets called when predicting using either of the productions for `E`:
`E -> (E op E) | INTLITERAL`

```
TreeNode* E(Scanner* s) {

    TOKEN* prevToken = s->GetCurTokenSequence();
    TreeNode* ret = E1(s);
    if(!ret) {
        s->SetCurTokenSequence(prevToken);
        ret = E2(s);
    }
    return ret;
}
```

Building AST

```
TreeNode* IsTerm(Scanner* s, TOKEN tok) {
      TreeNode* ret = NULL;
      TOKEN nxtToken = s->GetNextToken();
      if(nxtToken == tok)
            ret = CreateTreeNode(nxtToken.val);
      return ret;
}

TreeNode* E1(Scanner* s) {
      return IsTerm(s, INTLITERAL);
}

TreeNode* E2(Scanner* s) {
      TOKEN nxtTok = s->GetNextToken();
      if(nxtTok == LPAREN) {
            TreeNode* left = E(s);
            if(!left) return NULL;
            TreeNode* root  = OP(s);
            if(!root) return NULL;
            TreeNode* right = E(s)
            if(!right) return NULL;
            nxtTok = s->GetNextToken();
            if(nxtTok != RPAREN); return NULL;
                  //set left and right as children of root.
            return root;
      }
```

Building AST

```
TreeNode* OP(Scanner* s) {
      TreeNode* ret = NULL;
      TOKEN tok = s->GetNextToken();
      if((tok == ADD) || (tok == SUB) || (tok == MUL) || (tok == DIV))
             ret = CreateTreeNode(tok.val);
      return ret;
}

TreeNode* E(Scanner* s) {
      TOKEN* prevToken = s->GetCurTokenSequence();
      TreeNode* ret = E1(s);
      if(!ret) {
            s->SetCurTokenSequence(prevToken);
            ret = E2(s);
      }
      return ret;
}
```

Start the parser by invoking E().
Value returned is the root of the AST.

Exercise

- Did we build the AST bottom-up or top-down?