CS406: Compilers

Spring 2021

Week 6: Parsers (LR(k)) and Semantic Processing

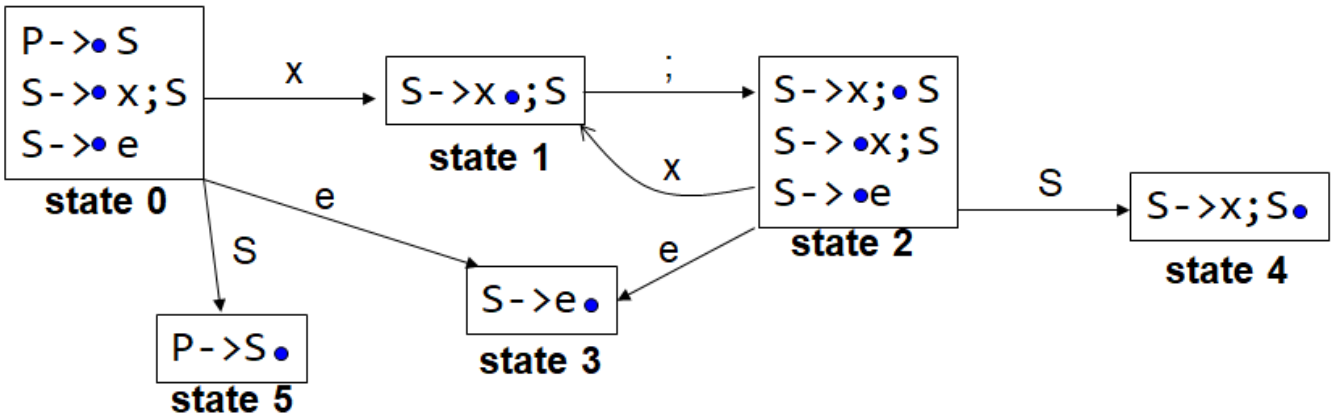
LR(0) Parsing

- 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

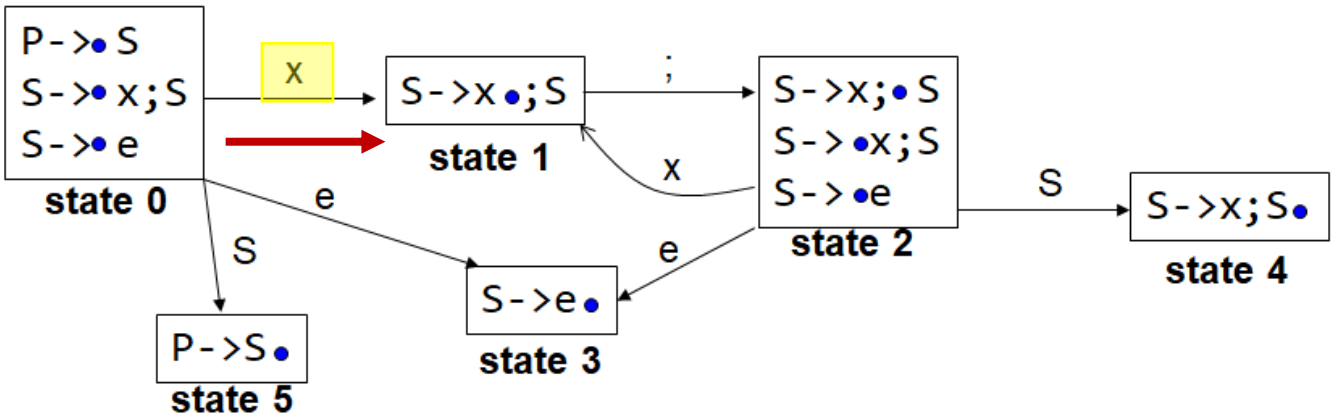
Parse Stack
0
0 1
0 1 2
0 1 2 1



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

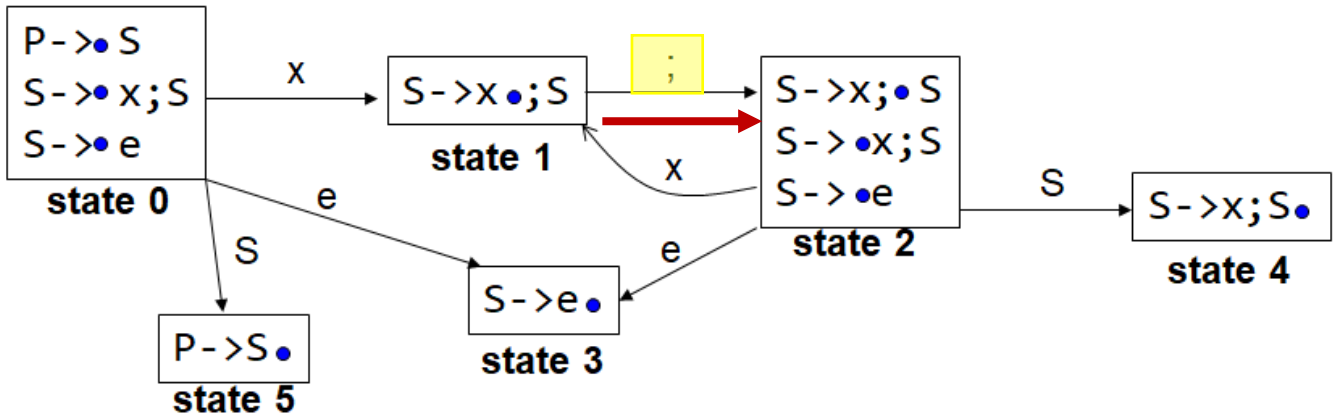


Go from state 0 to state 1 consuming x

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

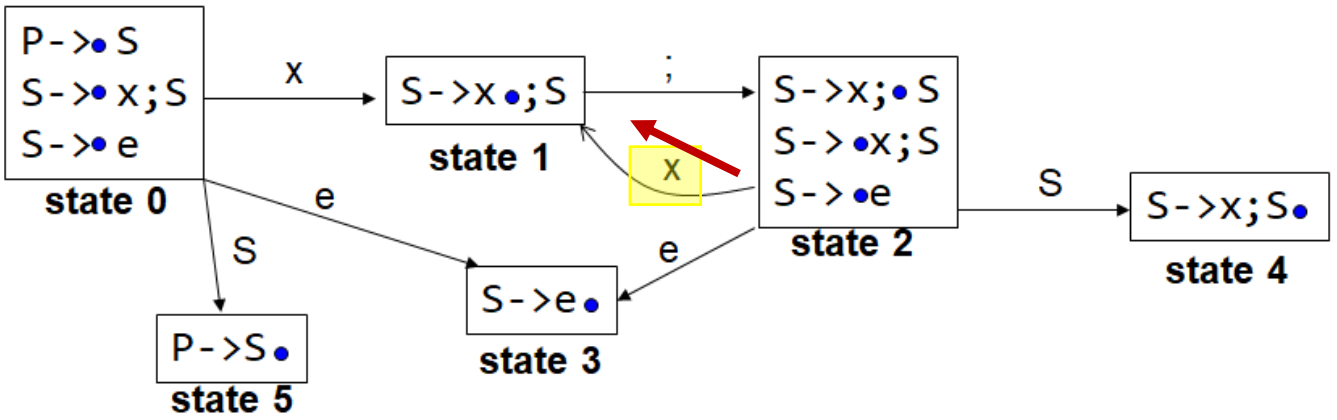


Go from state 1 to state 2 consuming ;

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 .

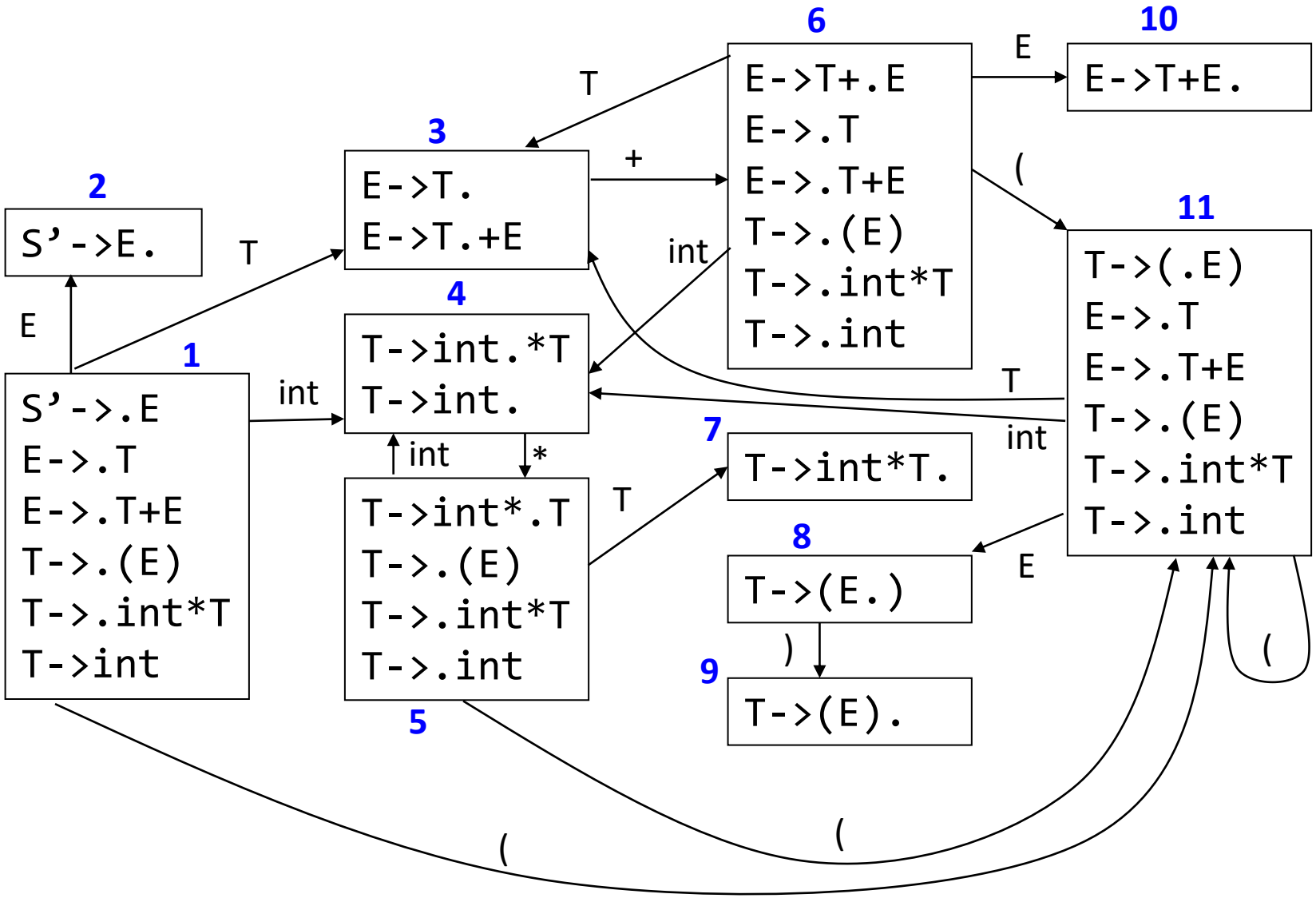
- Assume: Next input is t

We shift if s contains X->β ● tω

== s has a transition labelled t

LR(0) Parsing

- What if s contains X->β • tω and X->β• ?



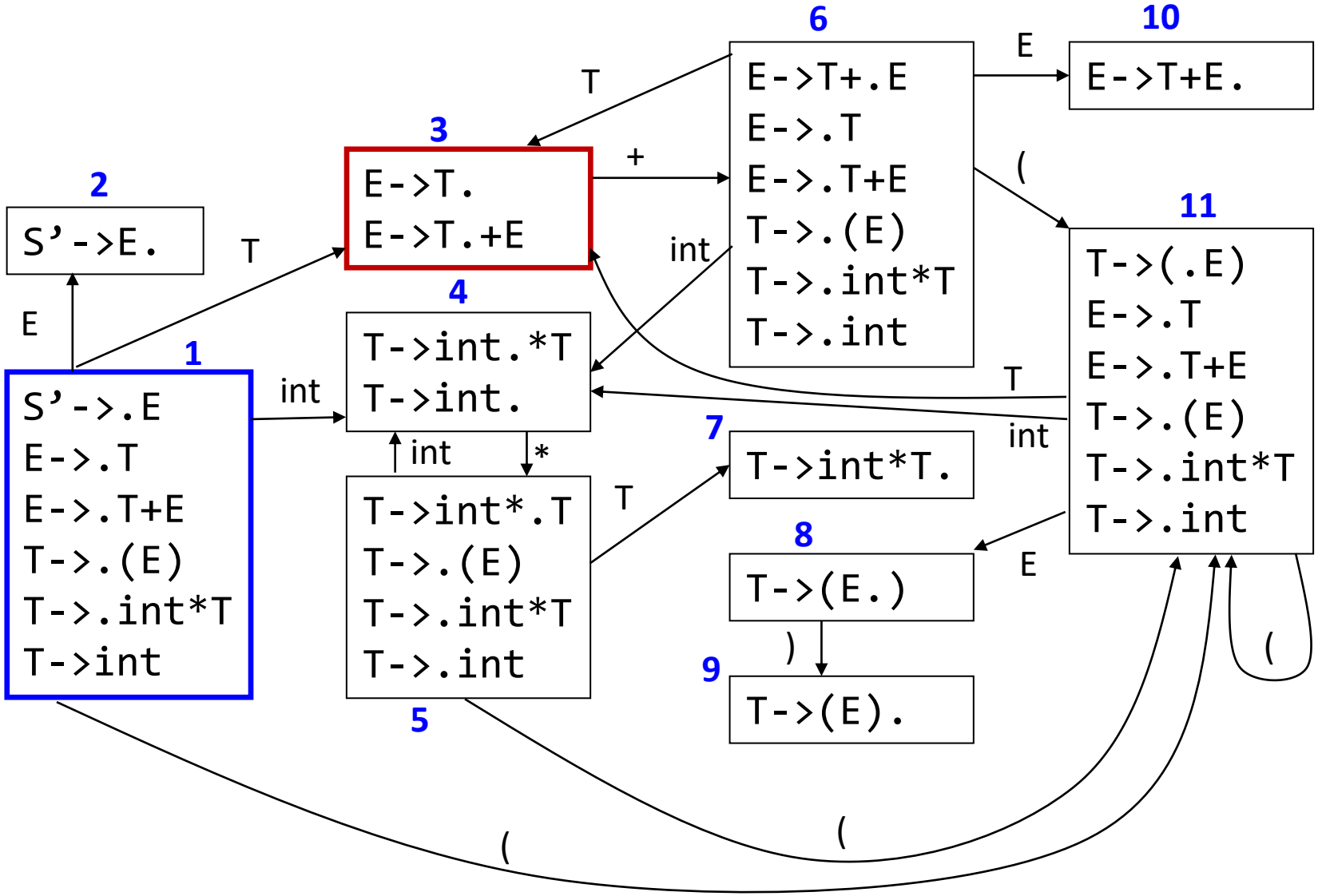
Conflicts or not?

SLR Parsing

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

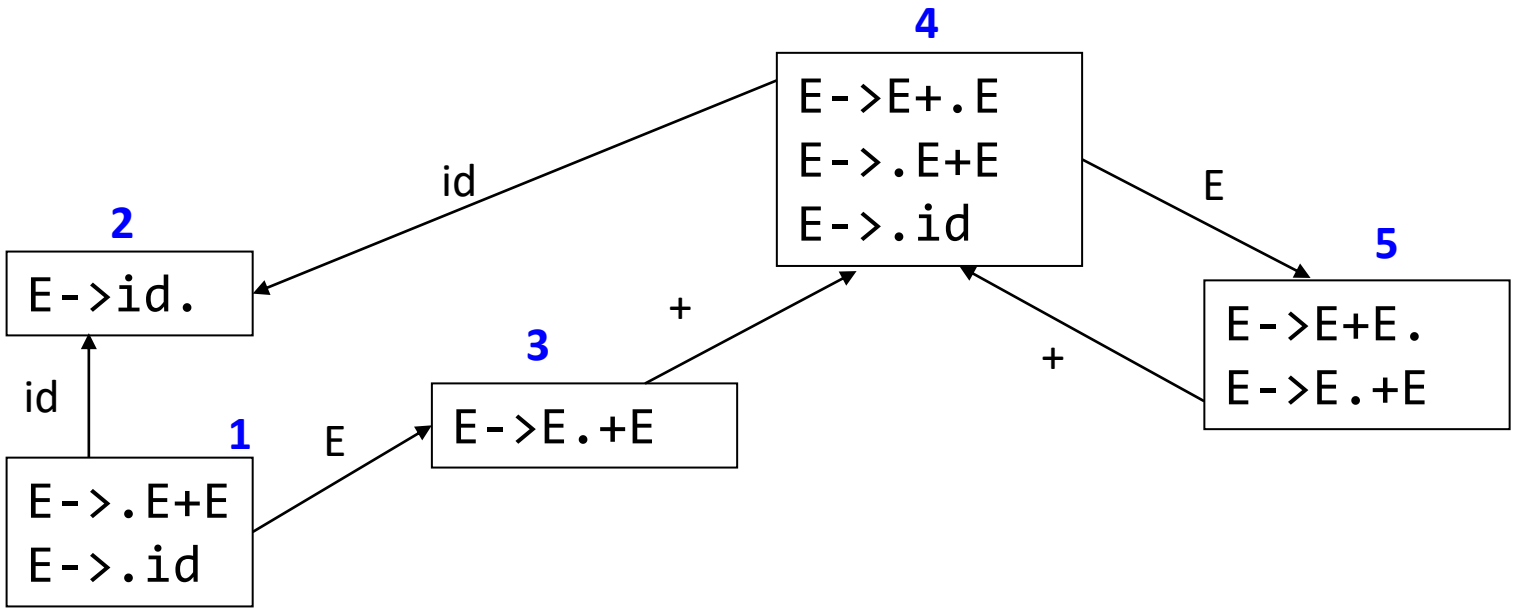
Reduce X->β● only if

t ∈ Follow(X)



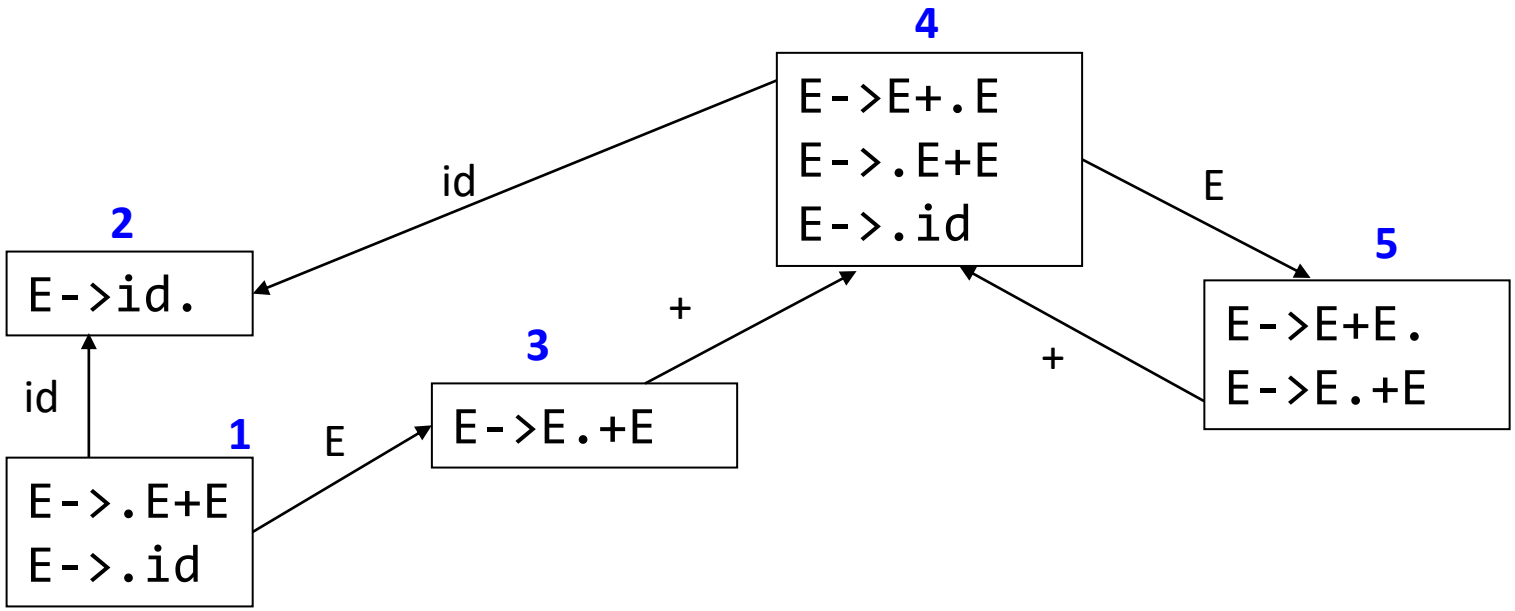
Follow(E) = { $,) } => reduce by E->T. only if next input is $ or)

lookahead 1



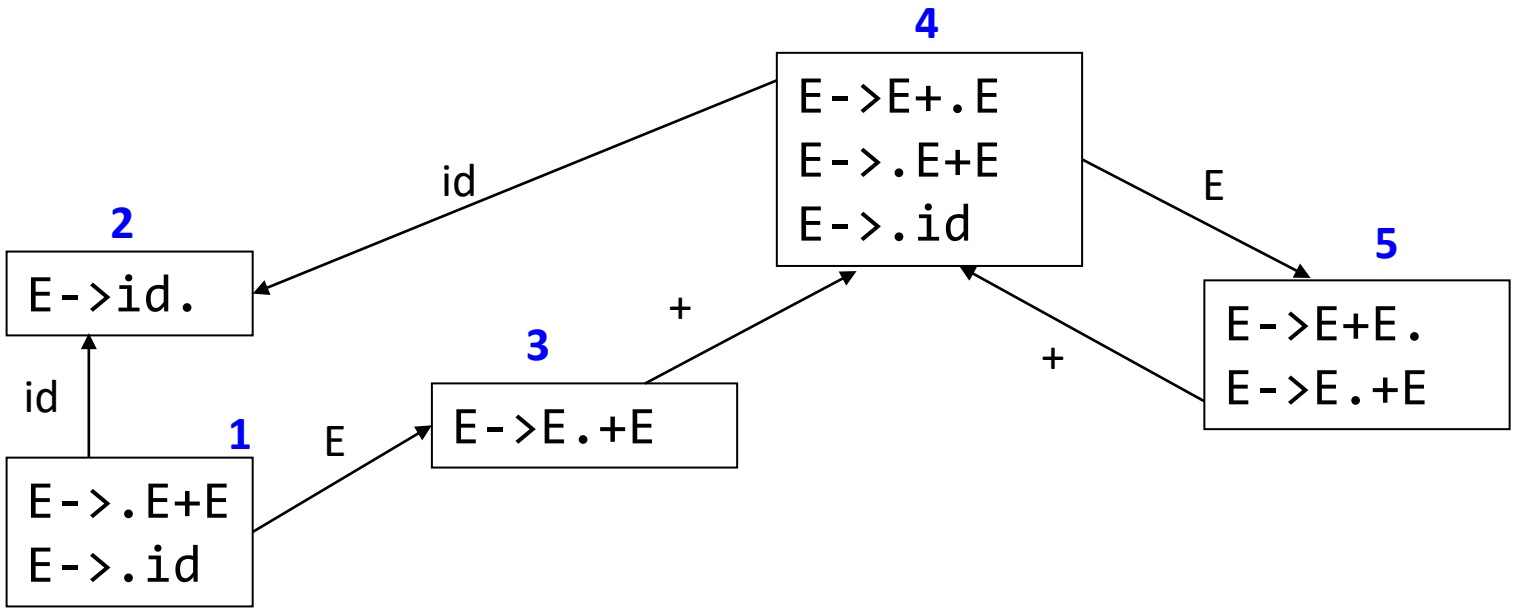
What about the grammar **E-> E + E | id** ?

LR(0)?



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

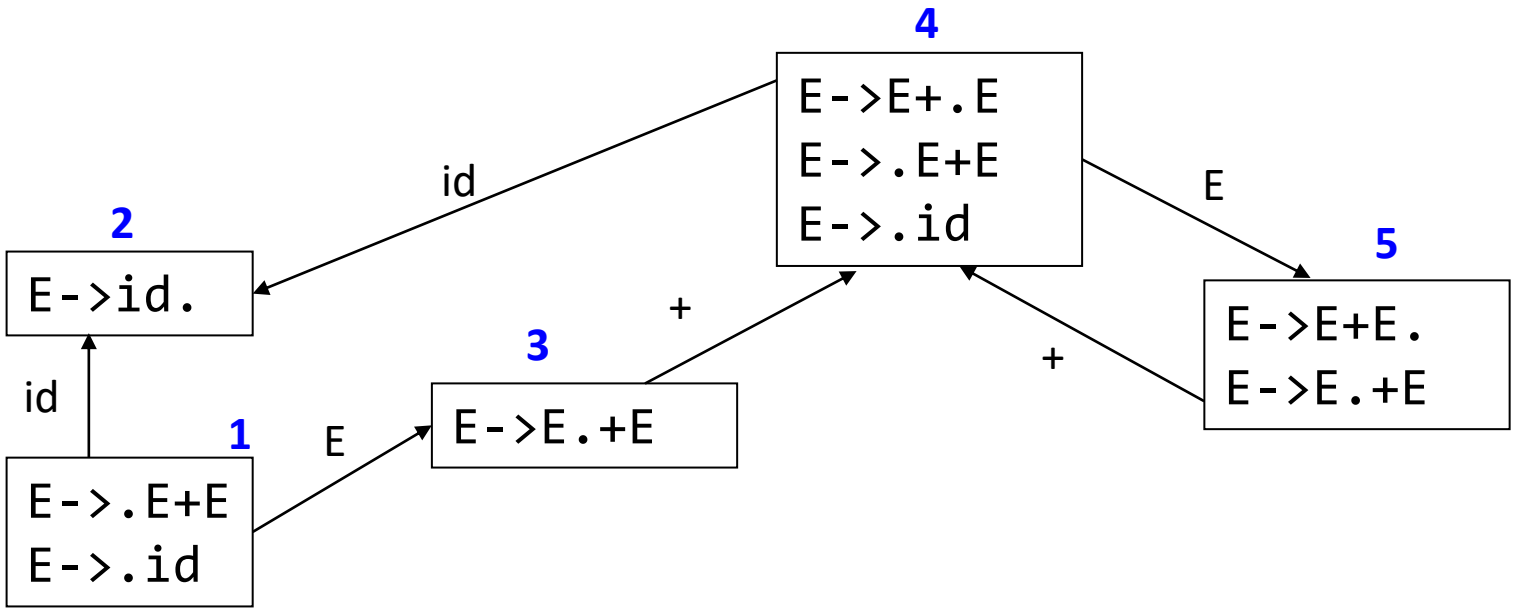
LR(0)? SLR(1)?



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

LR(0)? SLR(1)?

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

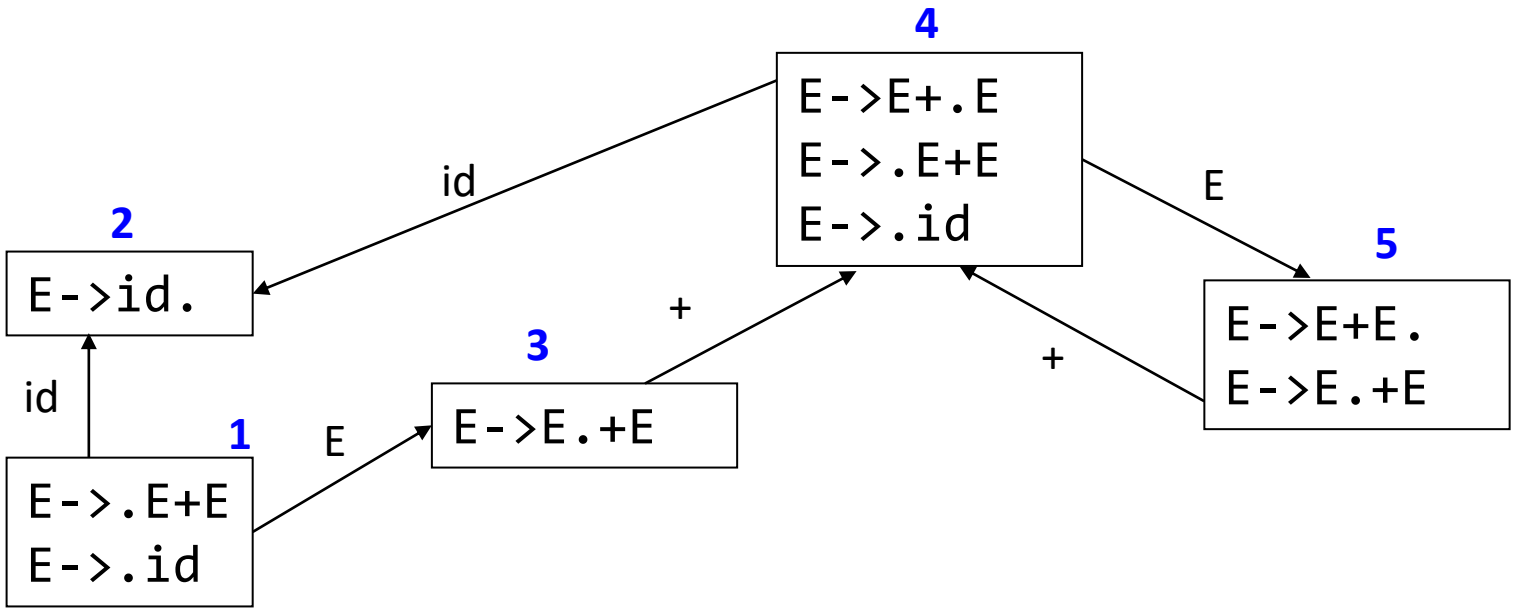
But state 5 has E->E.+E (shift if next input is +)

Shift-reduce conflict!

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



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

LR(0)? SLR(1)?

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

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

%left +

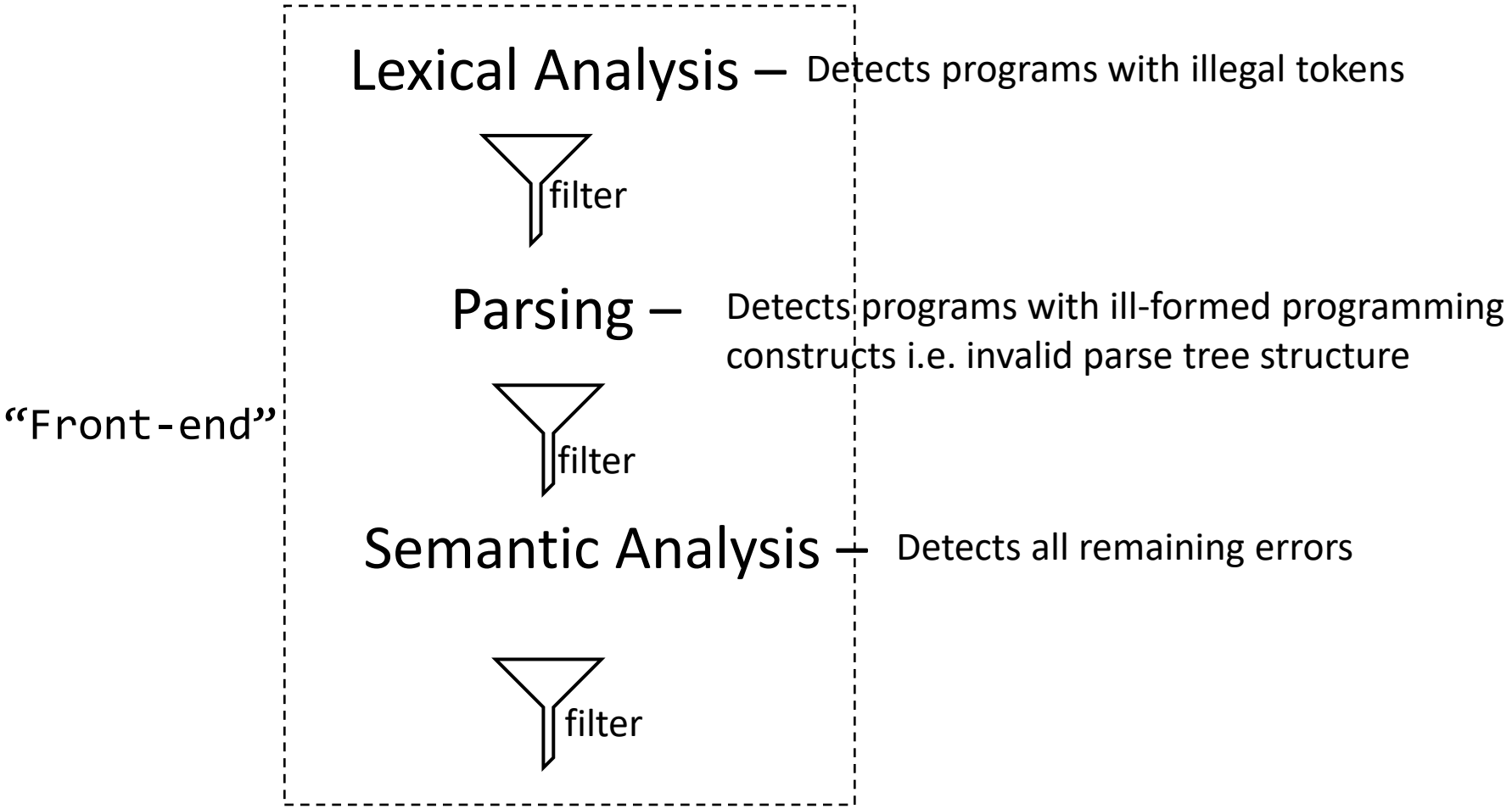
says reduce if the next input symbol is + i.e. prioritize rule E+E. over E.+E

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

slide courtesy: Milind Kulkarni

Semantic Analysis



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

    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.
  - …

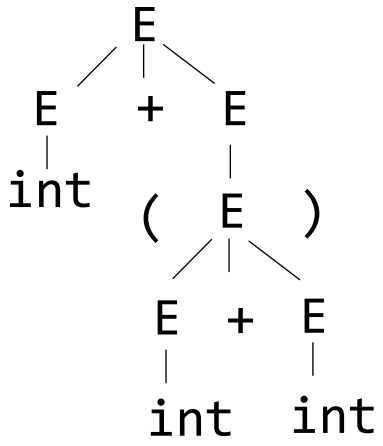
Abstract Syntax Tree

- Abstract Syntax Tree (AST) or Syntax Tree is 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
```

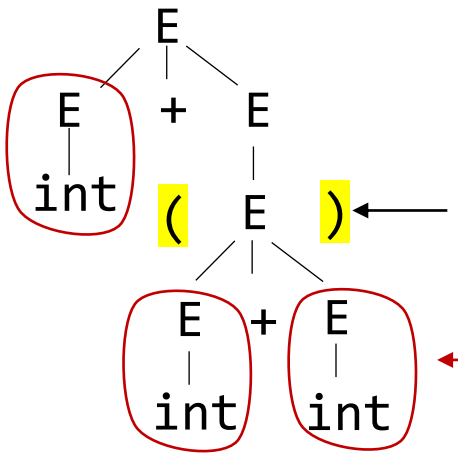
The parse tree for 1+(2+3)



Abstract Syntax Tree - Example

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

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

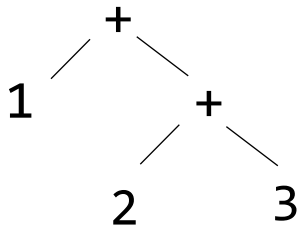


Expresses associativity.
Lower subtree in the hierarchy can express.

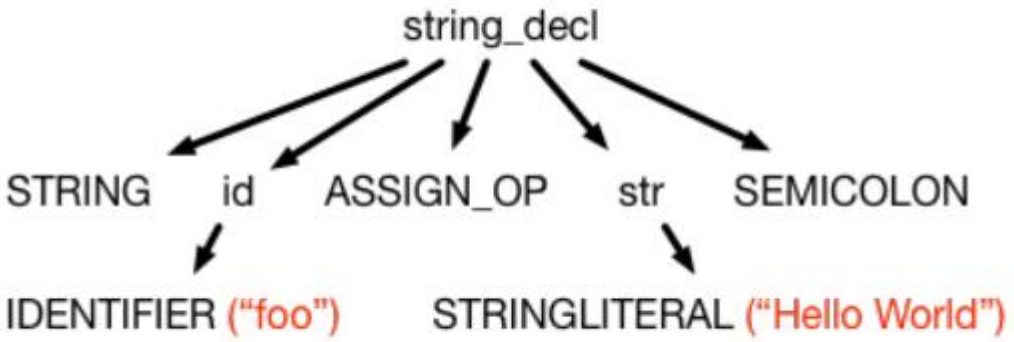
Single child.
Can compress.

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

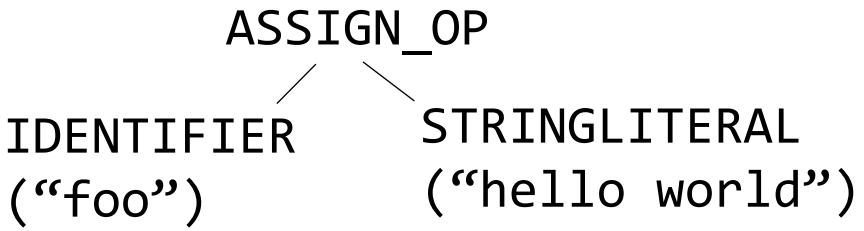
AST for `1+(2+3)` :



AST - Example


string_decl
STRING
id
ASSIGN_OP
str
SEMICOLON
IDENTIFIER ("foo")
STRINGLITERAL ("Hello World")

≡


ASSIGN_OP
IDENTIFIER
("foo")
STRINGLITERAL
("hello world")

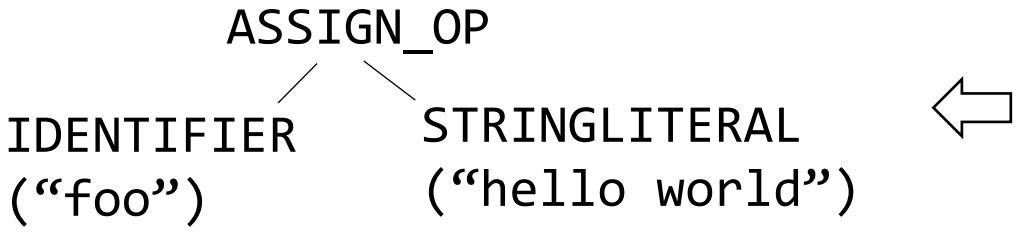
Semantic Analysis – How?

- 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 vs. Dynamic Scope

- Most languages are statically scoped
  - Scope depends 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

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

value of a is 4 here

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