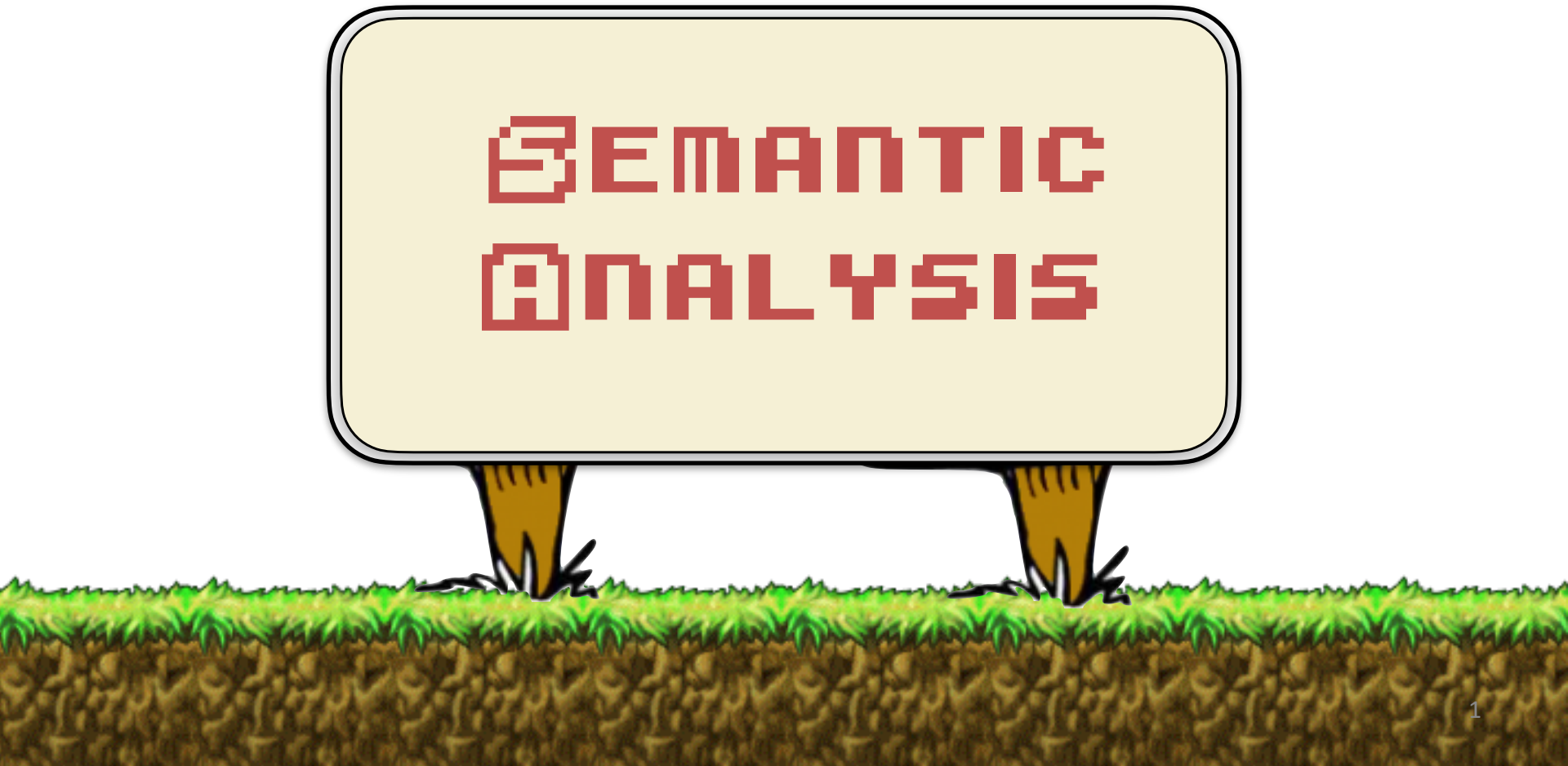
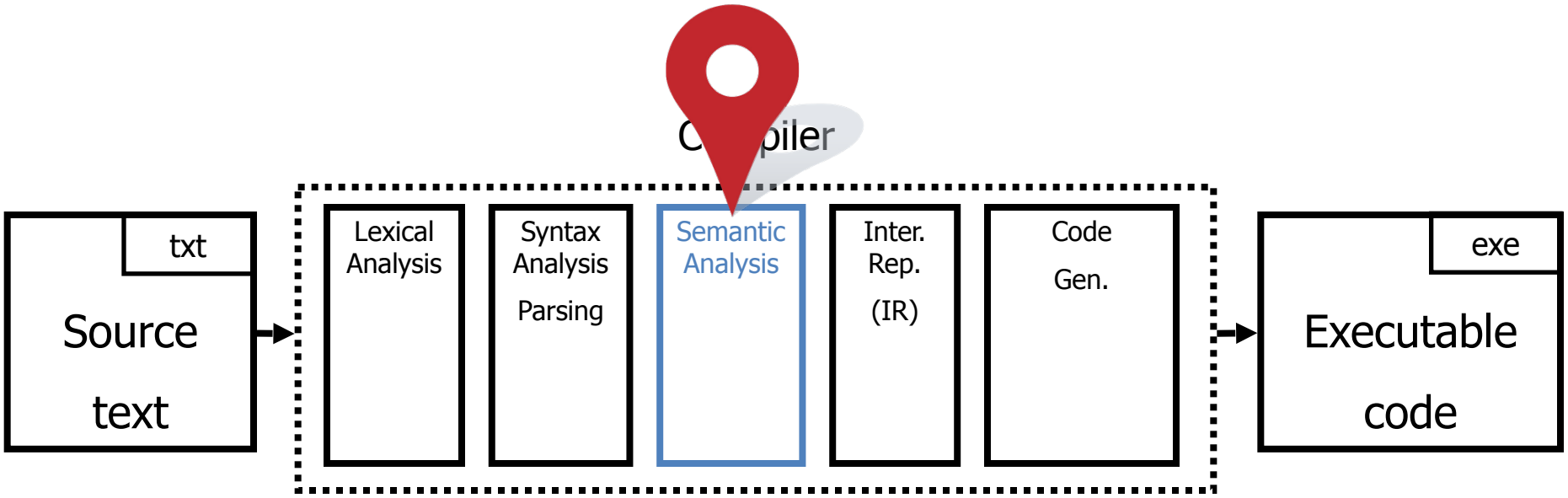
Theory of Compilation

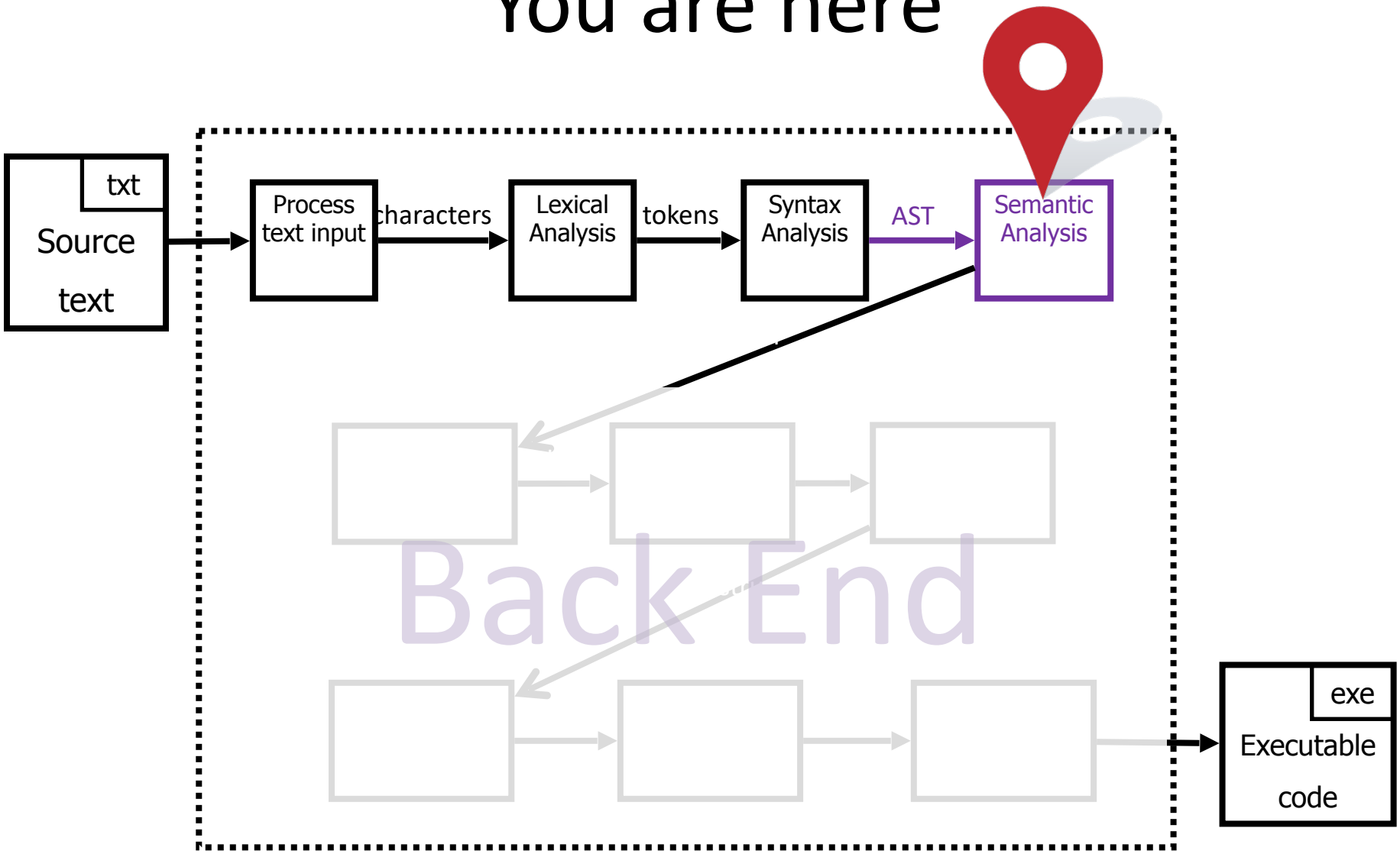
Lecture 04



You are here

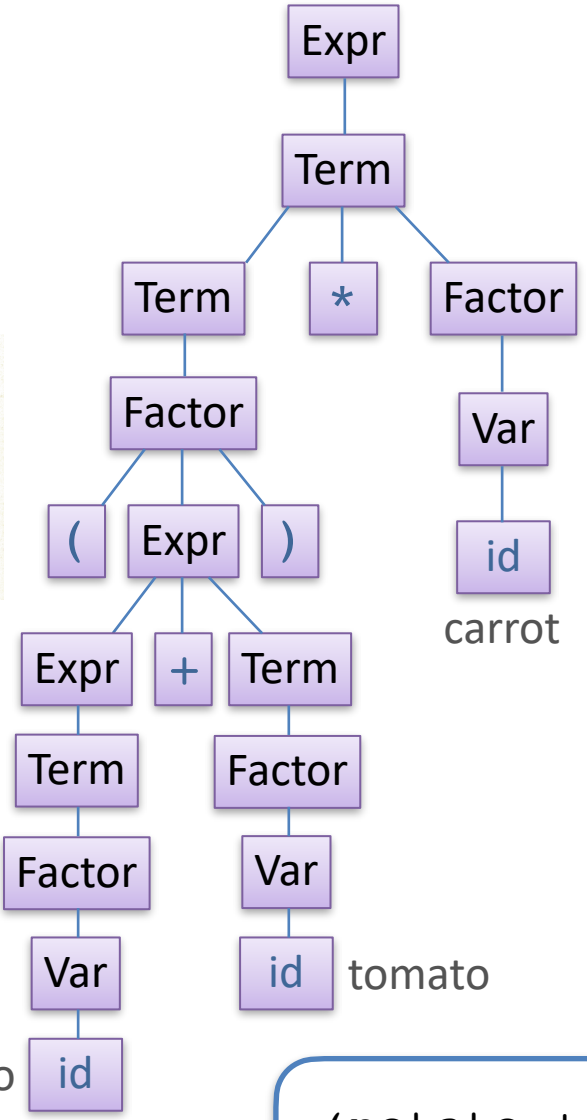

Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Inter. Rep.
(IR)
Code Gen.
exe
Executable code

You are here


txt
Source text
Process text input
characters
Lexical Analysis
tokens
Syntax Analysis
AST
Semantic Analysis
Back End
exe
Executable code

Building the Abstract Syntax Tree

Syntax Tree



Abstract Syntax Tree


```
(potato + tomato) * carrot
```

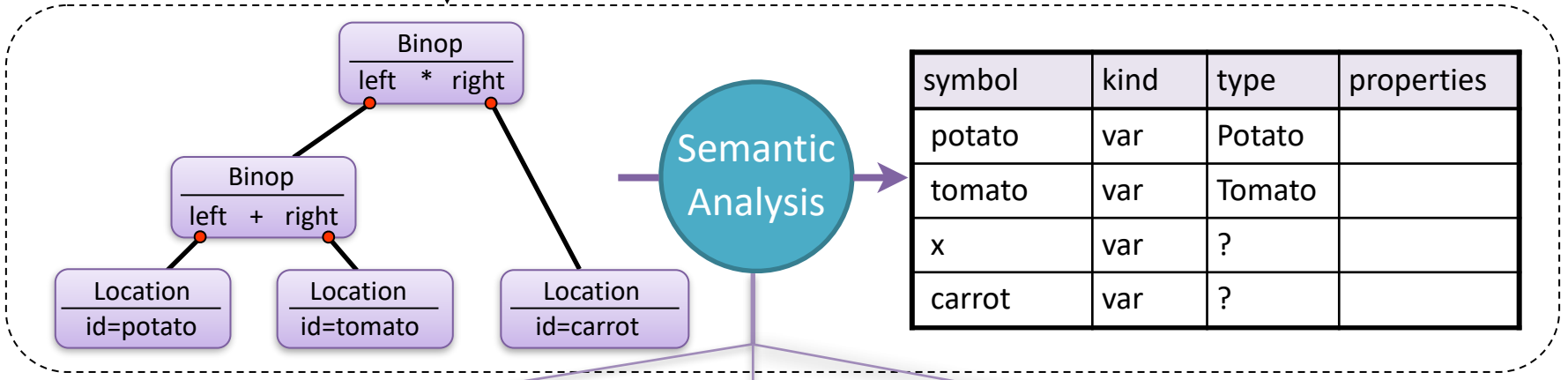
What We Want

```
Potato potato;
Tomato tomato;
x = (potato + tomato) * carrot
```

Lexical analyzer

```
… <ID,potato> <+> <ID,tomato> <)> <*> <ID,carrot> EOF
```

Parser



symbol	kind	type	properties
potato	var	Potato	
tomato	var	Tomato	
x	var	?	
carrot	var	?	

‘carrot’ is undefined

‘potato’ used before initialized

Cannot add ‘Potato’ and ‘Tomato’

Semantic Analysis

- Often called “Contextual analysis”
  - As opposed to our syntax analysis — which was “context free”
- Properties that cannot be formu
  - Declare before use
  - Type checking
  - Initialization
  - …
- Properties that are clumsy to fo
  - “break” only appears inside a loo
  - …

stmts → ε | stmt ; stmts
stmt → if expr { stmts } s_else
| while expr { stmts$^{\ell}$ }
| let id : type = expr
| id = expr
s_else → ε | else { stmts }

stmts$^{\ell}$ → ε | stmt$^{\ell}$; stmts$^{\ell}$
stmt$^{\ell}$ → if expr { stmts$^{\ell}$ } s_else$^{\ell}$
| while expr { stmts$^{\ell}$ }
| let id : type = expr
| id = expr | break
s_else$^{\ell}$ → ε | else { stmts$^{\ell}$ }

he
pter 11

Semantic Analysis

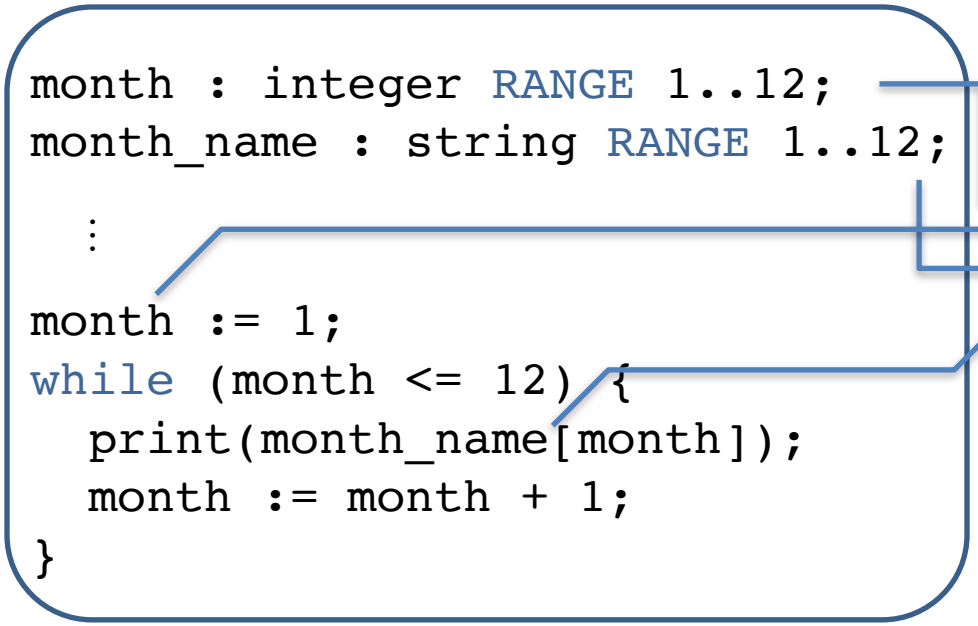
- Identification
  - Gather information about each named item in the program
  - *e.g.*, what is the declaration for each usage
- Context checking
  - Type checking
  - *e.g.*, the condition in an if-statement is a Boolean

Identification

```
month : integer RANGE 1..12;
month := 1;

while (month <= 12) {
   print(month_name[month]);
   month := month + 1;
}
```

Symbol table



name	pos	type	…
month	1	int 1..12	
month_name	…	string[1..12]	
…			

- A table containing information about identifiers in the program
- Single entry for each named item

Not so fast…

```
struct one_int {
   int i;
} i;

main() {
 i.i = 42;
 int t = i.i;
 printf("%d",t);
}
```

A struct field named "i"

A struct variable named "i"

Assignment to the "i" field of struct "i"

Reading the "i" field of struct "i"

Not so fast…

```
struct one_int {
   int i;
} i;

main() {
 i.i = 42;
 int t = i.i;
 printf("%d",t);
 {
  int i = 73;
  printf("%d",i);
 }
}
```

A struct field named "i"

A struct variable named "i"

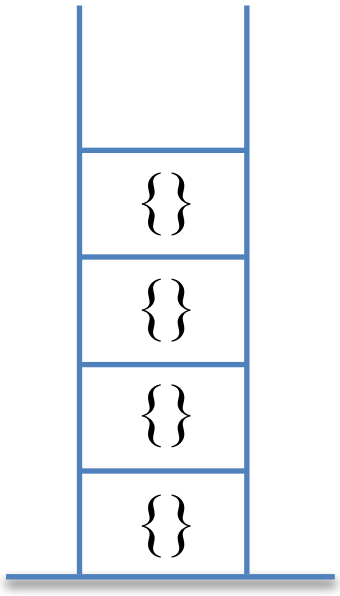
Assignment to the "i" field of struct "i"

Reading the "i" field of struct "i"

another variable named "i"?!

Now what?!

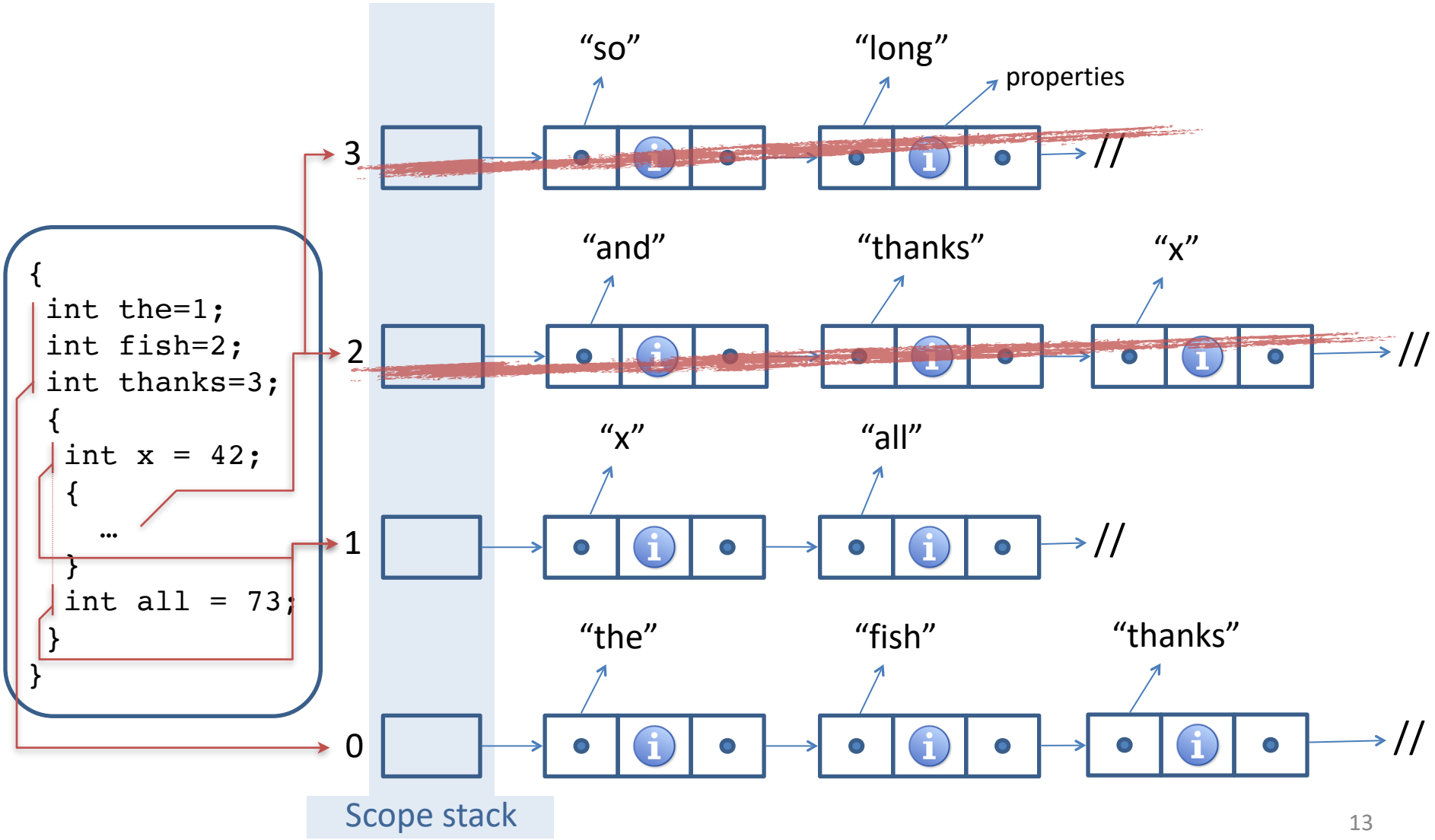
Scopes



- Typically: stack structured scopes

- Scope entry
  - push new empty scope element
- Scope exit
  - pop scope element and discard its content
- Identifier declaration
  - identifier created inside (current) top scope
- Identifier Lookup
  - Search for identifier top-down in scope stack

Scope-structured Symbol Table

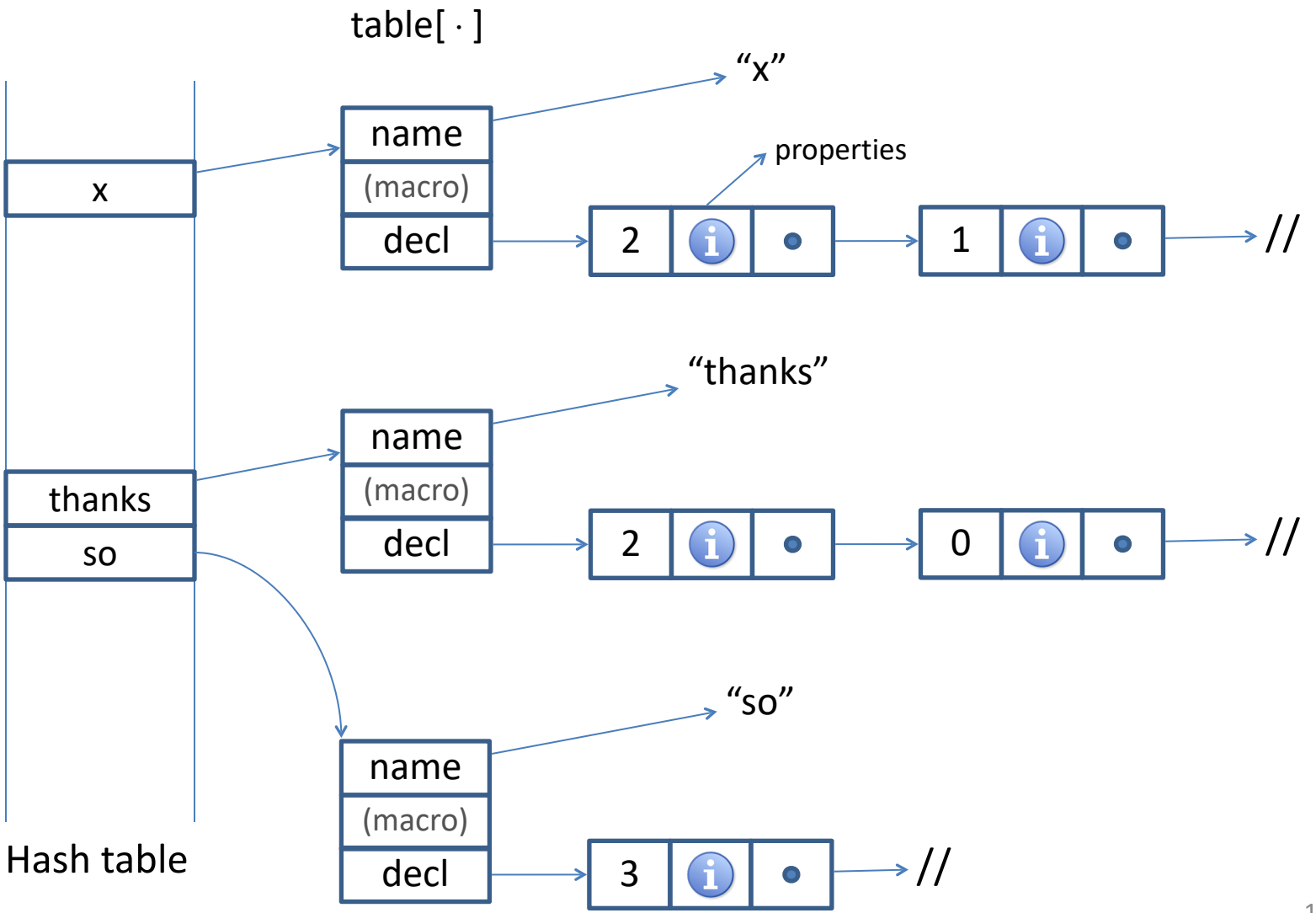


Scope and Symbol Table

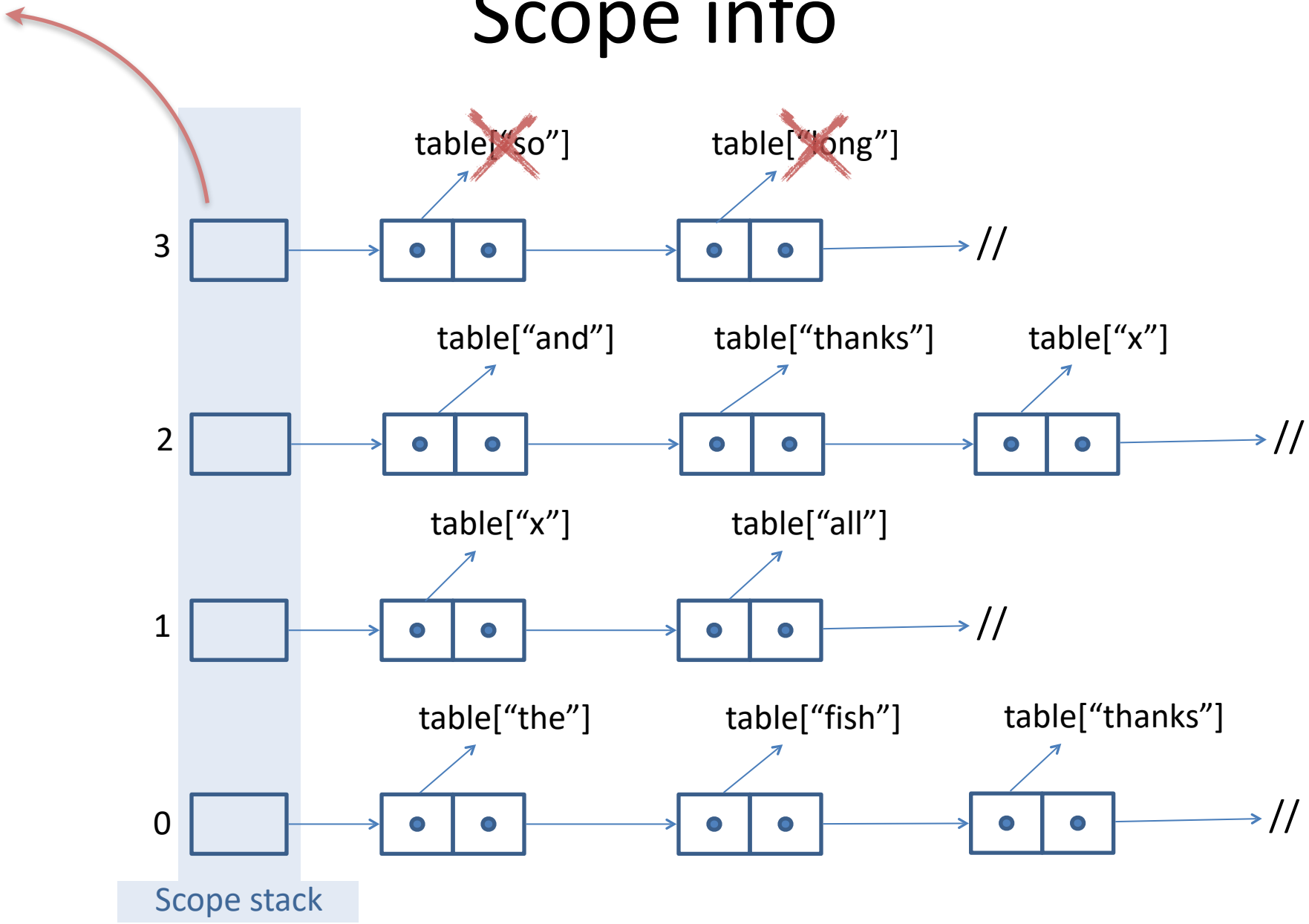
- Scope × Identifier ⟶ properties
  - Expensive lookup: when a reference to x is encountered, the entire table needs to be searched

- A better solution
  - Hash table over identifiers
  - For each key: list of scopes where it has been defined (innermost first)

Hash Table-based Symbol Table



Scope info



(now just pointers to the corresponding record in the symbol table)

Remember Lexing+Parsing?

- How did we know to always map an identifier to the same token?
  - We didn't! Each occurrence is a new token (but with the same lexical attribute, "name").
  - Now is the time to match them up.

How does this magic happen?

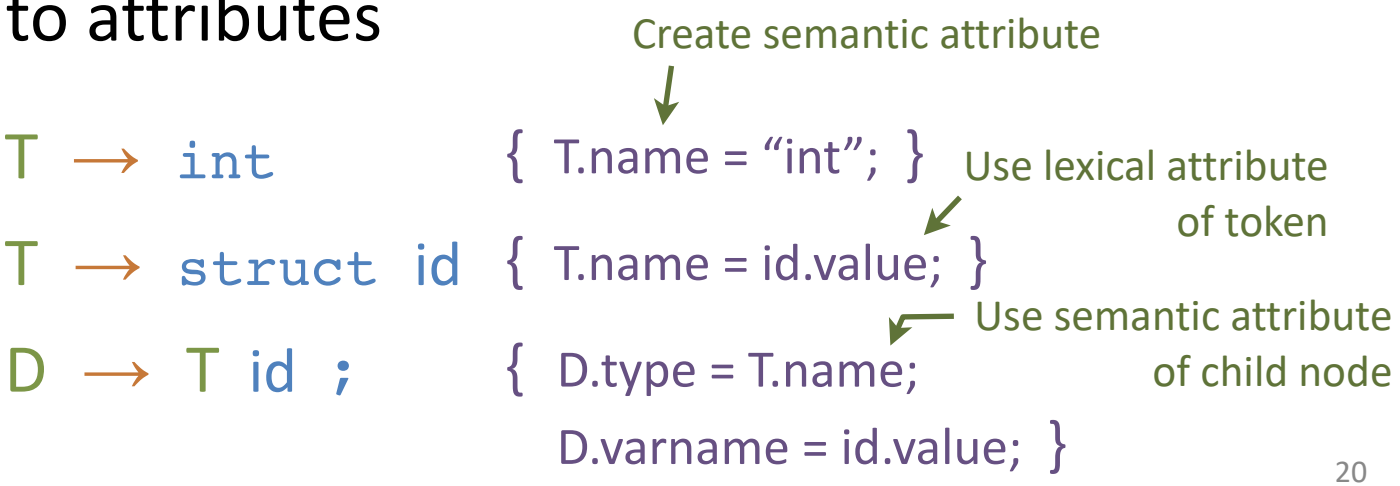
- We probably need to go over the AST?
- How does this relate to the clean formalism of the parser?

Syntax Directed Translation

- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - Define how to update the attributes
- Attribute grammars

Attribute Grammars

- Semantic attributes
  - Every grammar symbol has attached attributes
    - Example: N.name
- Semantic actions
  - Every production rule can define how to assign values to attributes



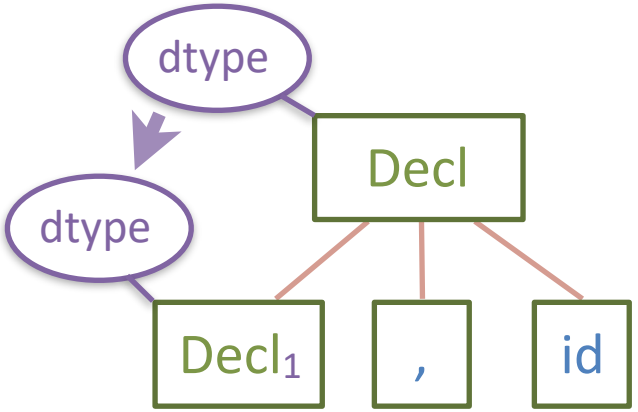
Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions

Decl → Decl , id

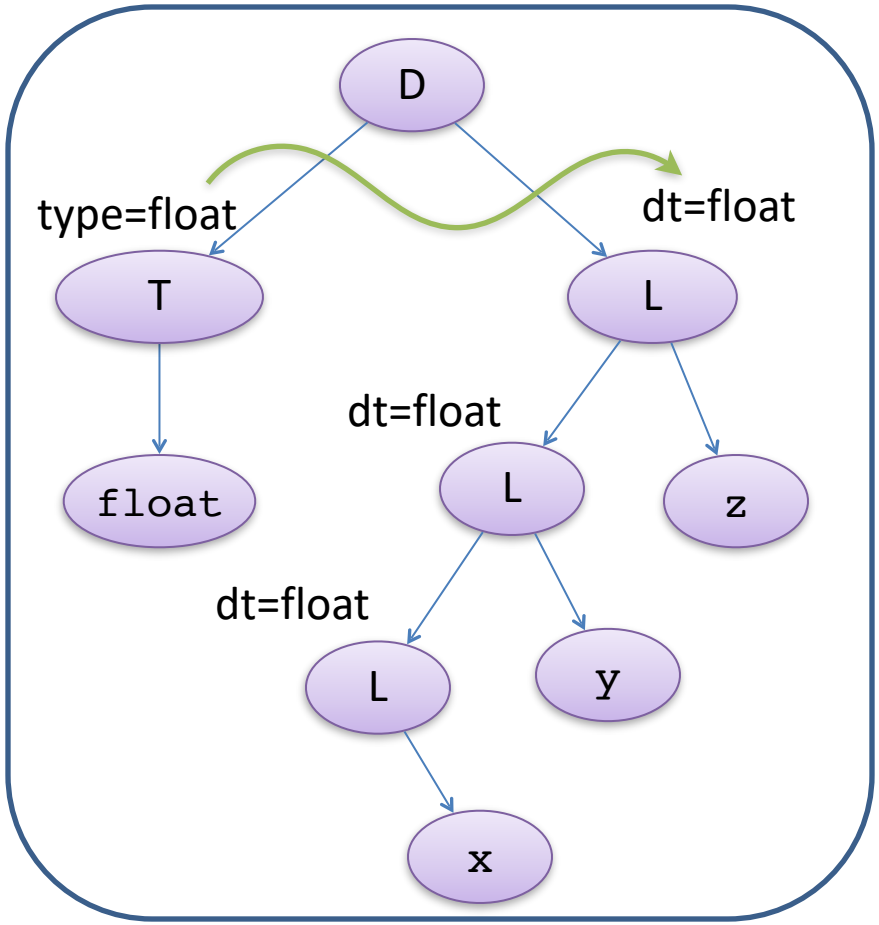
Decl → $Decl_1$, id

{ $Decl_1$.dtype = Decl.dtype; }



Example — Analyzing Declarations

```
float x, y, z
```



Production	Semantic Rule
D → T L	L.dt = T.type
T → `int`	T.type = int
T → `float`	T.type = float
L → L_1 , id	L_1.dt = L.dt addType(id.entry, L.dt)
L → id	addType(id.entry, L.dt)

Attribute Evaluation

- Build the AST
- Fill attributes of terminals with their lexical values
- Execute semantic actions of the nodes to assign values, until no new values can be assigned, *in the right order* such that
  - No attribute value is used before it's available
  - Each attribute will get a value only once

Attribute Dependencies

- All semantic actions take the form

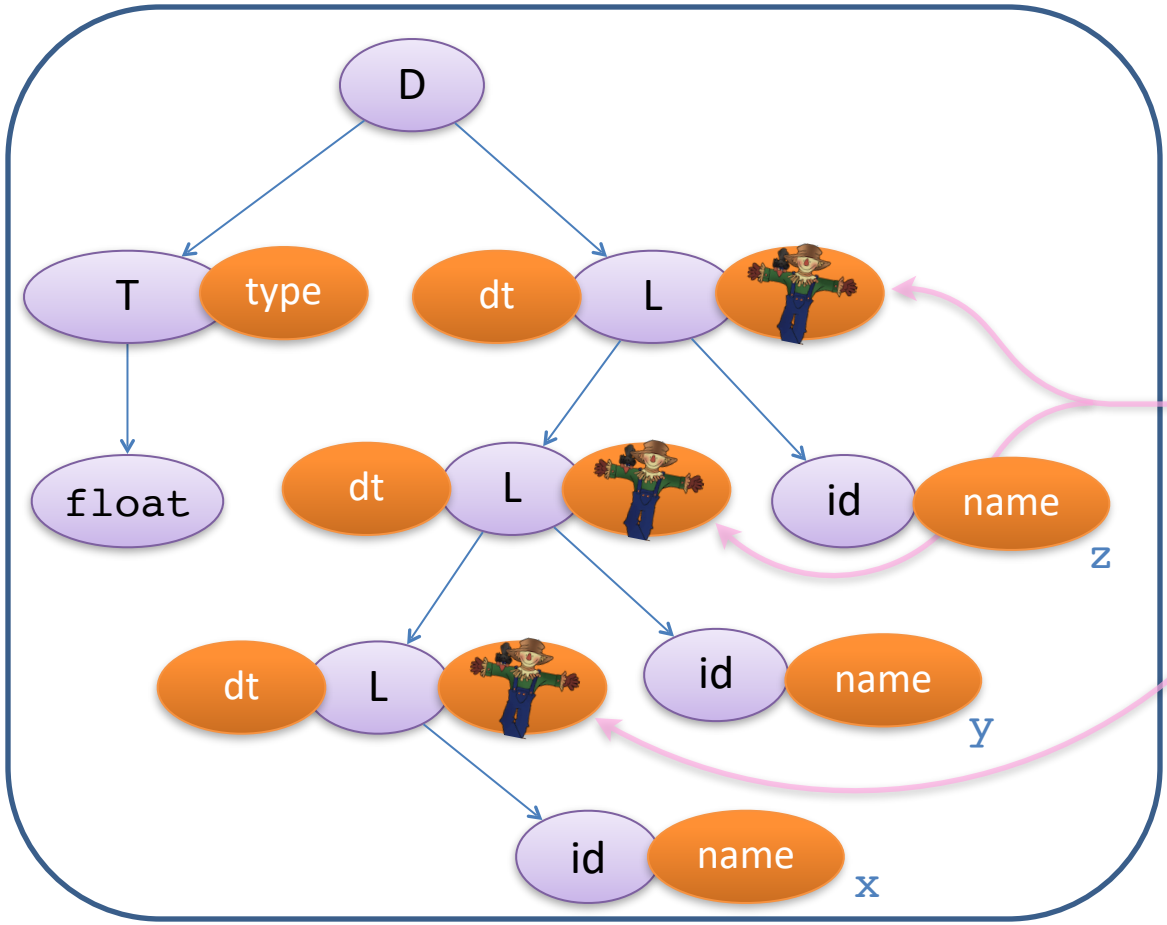
$$a_1 = f_1(b_{1\text{-}1}, b_{1\text{-}2}, \ldots)$$
$$a_2 = f_2(b_{2\text{-}1}, b_{2\text{-}2}, \ldots)$$

  - For actions with side effects, *e.g.* print(b.name): we introduce a dummy attribute as their result.

- Build a directed dependency graph G
  - For every attribute a of a node u in the AST, create a node u.a
  - For every dependency of the form $u.a_i = \cdots v.b_j \cdots$ create an edge $v.b_j \longrightarrow u.a_i$

Example

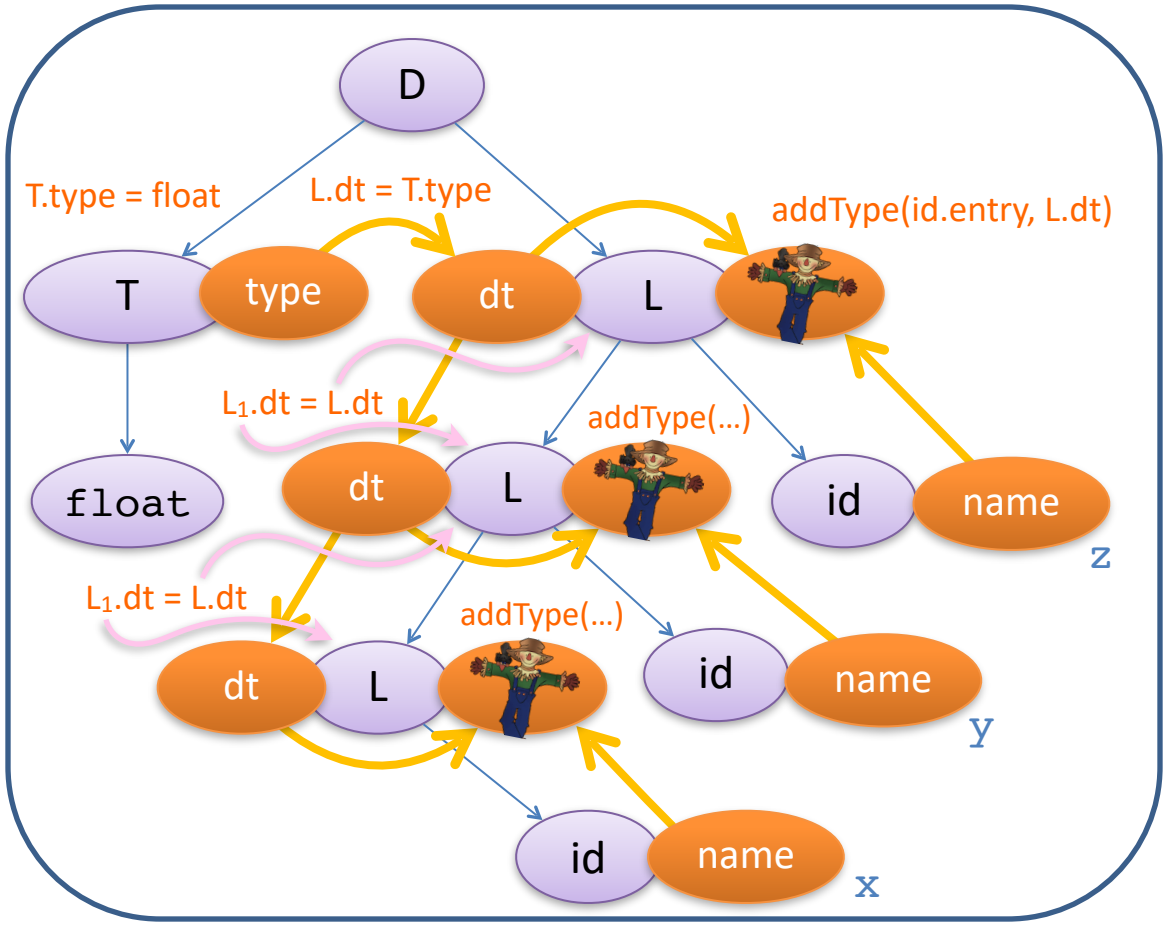
```
float x, y, z
```



Prod.	Semantic Rule
D → T L	L.dt = T.type
T → `int`	T.type = int
T → `float`	T.type = float
L → L_1, id	L_1.dt = L.dt addType(id.name, L.dt)
L → id	addType(id.name, L.dt)

Example

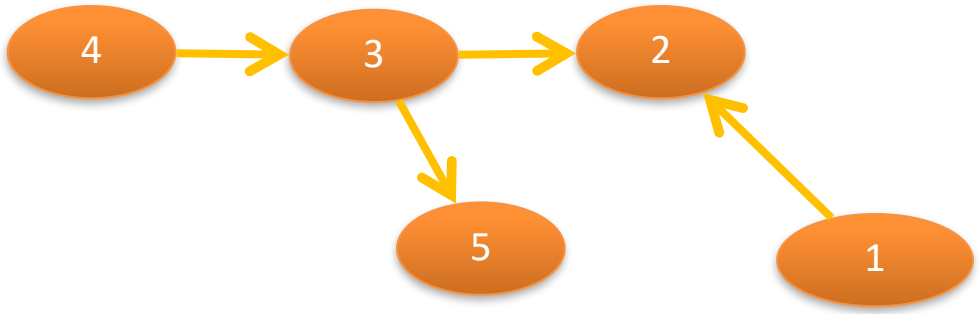
```
float x, y, z
```



Prod.	Semantic Rule
D → T L	L.dt = T.type
T → `int`	T.type = int
T → `float`	T.type = float
L → L_1, id	L_1.dt = L.dt addType(id.name, L.dt)
L → id	addType(id.name, L.dt)

Topological Order

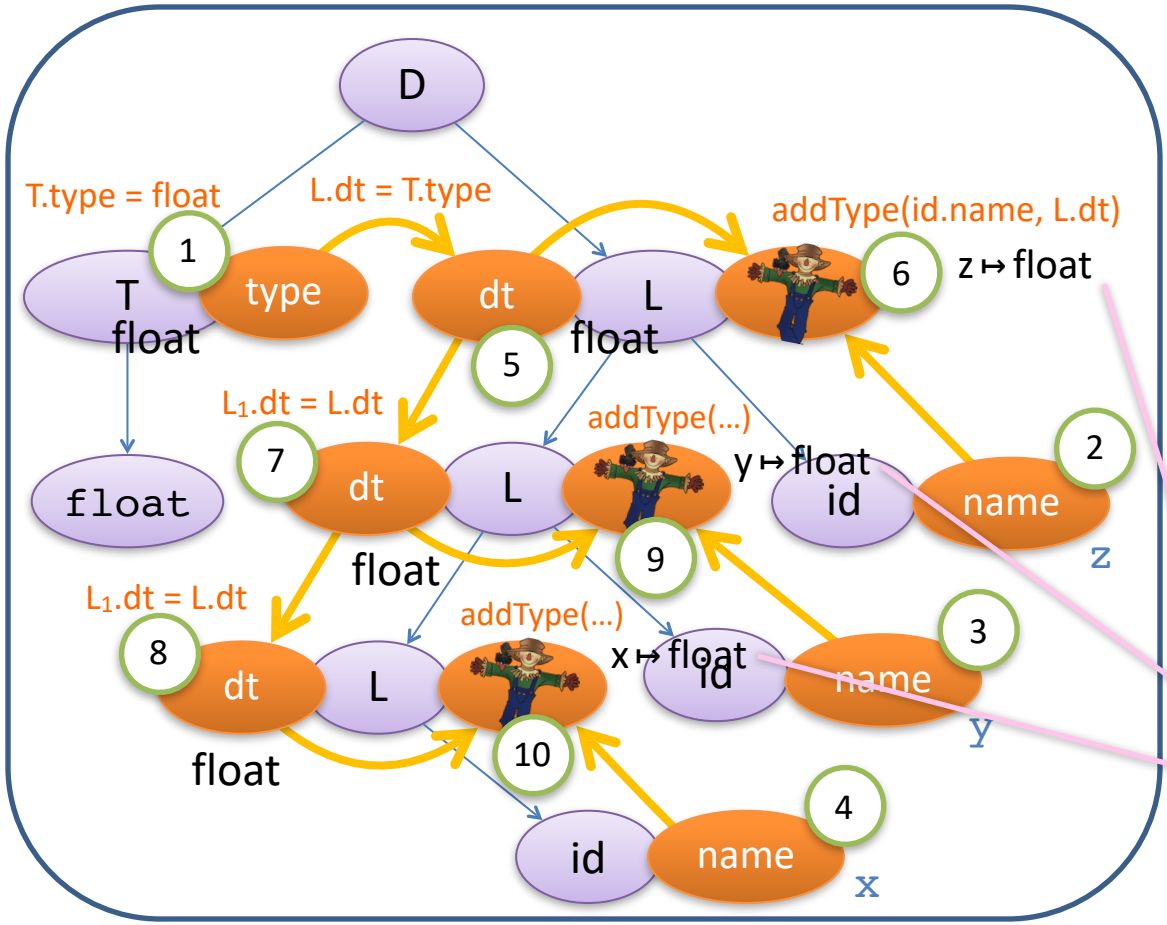
- For a graph G=(V, E), $|V|=k$
- Ordering of the nodes as $\langle v_1, v_2, \ldots, v_k \rangle$ such that for every edge $(v_i, v_j) \in E$, $i < j$



Example topological orderings: ⟨1 4 3 2 5⟩, ⟨4 3 5 1 2⟩

Example

```
float x, y, z
```



Prod.	Semantic Rule
D → T L	L.dt = T.type
T → `int`	T.type = int
T → `float`	T.type = float
L → L_1 , id	L_1.dt = L.dt addType(id.name, L.dt)
L → id	addType(id.name, L.dt)

symbol	kind	type	properties
z	var	float	
y	var	float	
x	var	float	

Symbol Table

But what about cycles?

- For a given attribute grammar — hard to detect if it has cyclic dependencies
  - Exponential cost

- Special classes of attribute grammars
  - Our “usual trick”:

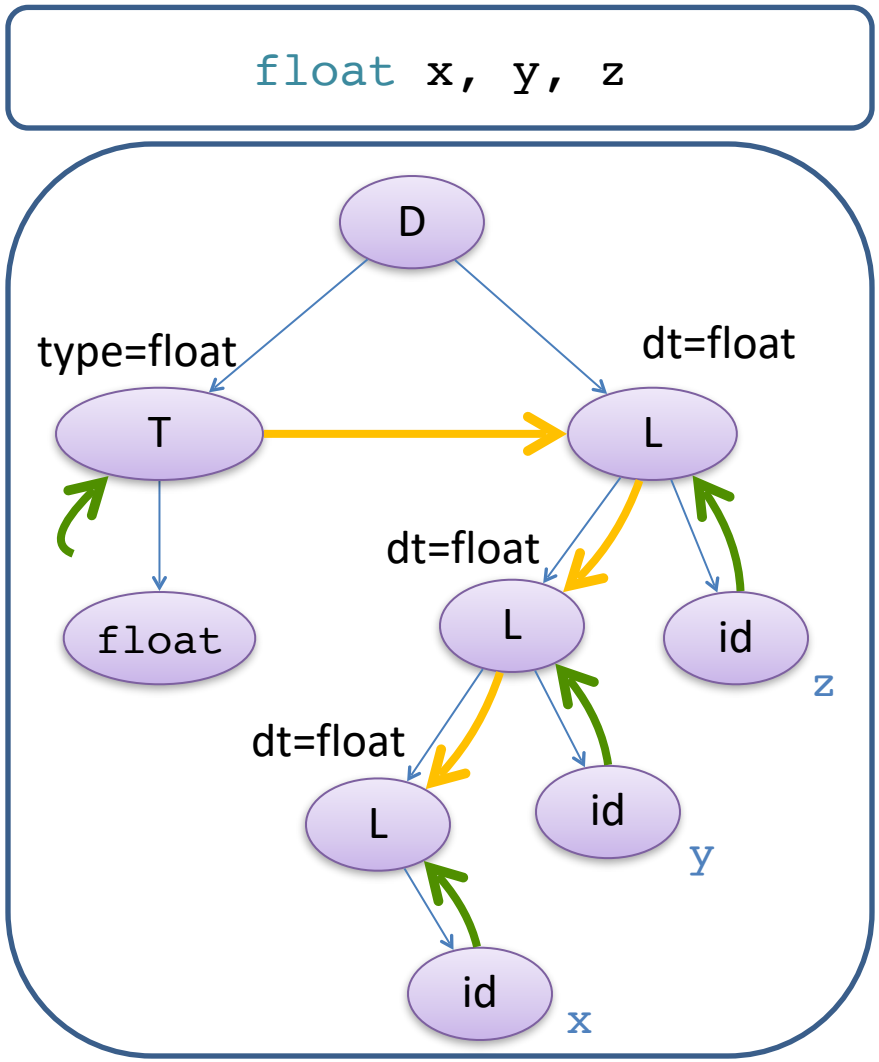
    sacrifice generality for predictable performance

Synthesized vs. Inherited Attributes

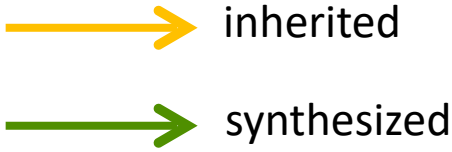
- *Synthesized* attributes
  - Attributes whose values at a given node depend ***only*** on the attributes of its children (and itself)
- *Inherited* attributes
  - Attributes whose values at a given node depend ***only*** on the attributes of its parent and siblings

Attributes that don't depend on anything (*e.g.* those of a token) — by convention, are classified as *synthesized* attributes.

Synthesized vs. Inherited Attributes



Prod.	Semantic Rule
D → T L	L.dt = T.type
T → int	T.type = integer
T → float	T.type = float
L → L_1 , id	L_1.dt = L.dt addType(id.name, L.dt)
L → id	addType(id.name, L.dt)



S-attributed Grammars

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
  - *No inherited attributes*

- Can be computed by any bottom-up parser during parsing — no need to construct dependency graph
  - Attributes can be stored on the parsing stack
  - Reduce operation computes the (synthesized) attribute from attributes of children

S-attributed Grammars

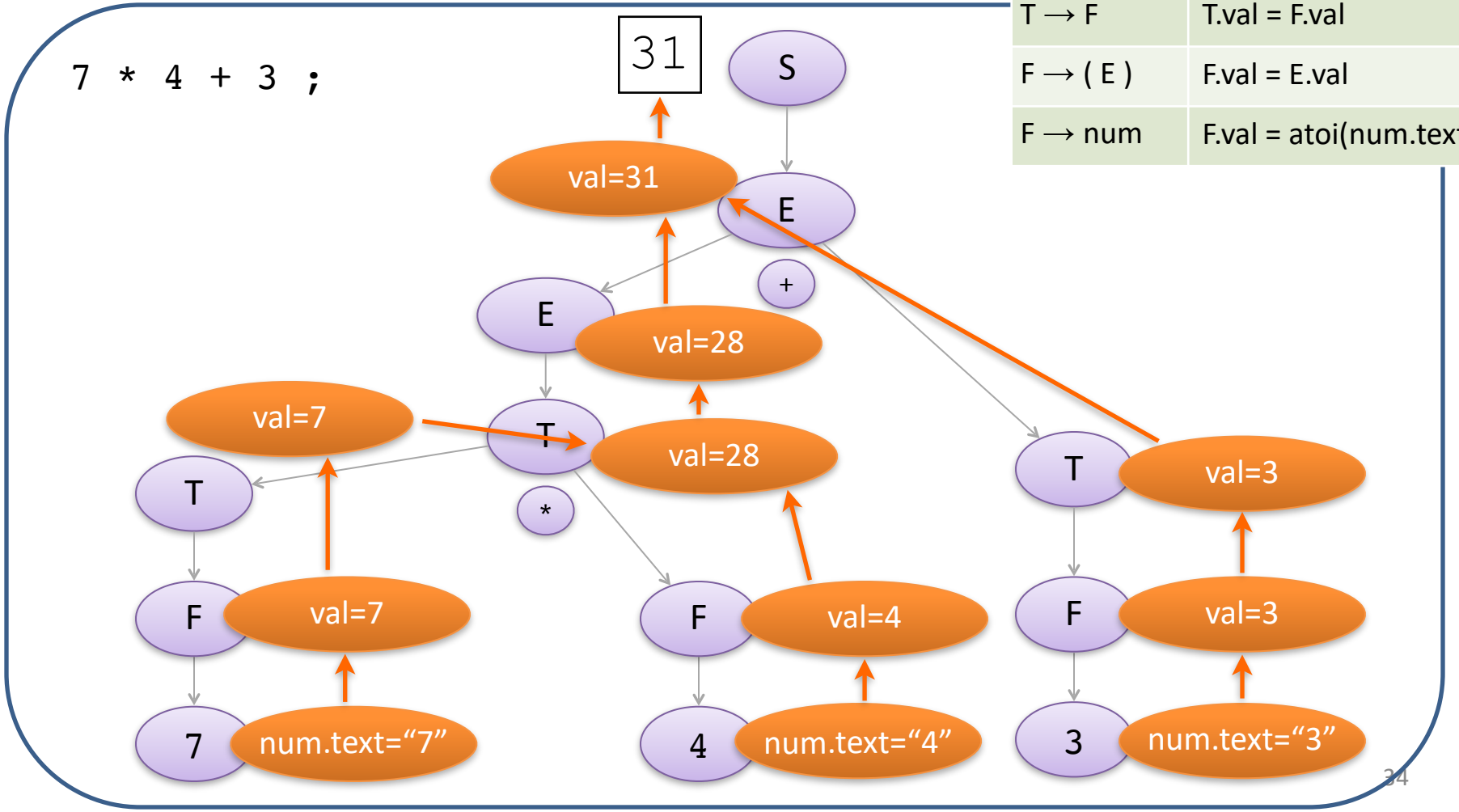
Arithmetic Calculator

Production	Semantic Rule
$S \rightarrow E$;	print(E.val)
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val = T_1.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ num	F.val = atoi(num.text)

S-attributed Grammars

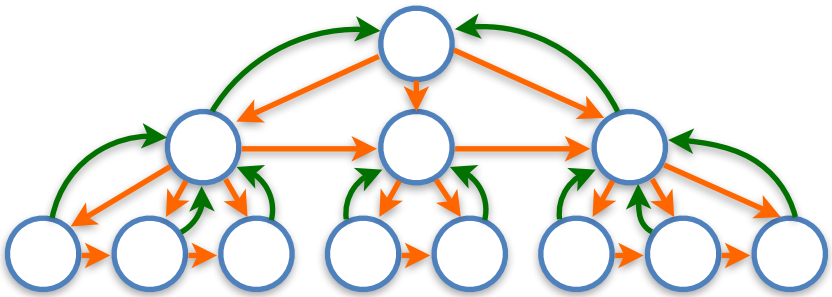
Production	Semantic Rule
S → E ;	print(E.val)
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → num	F.val = atoi(num.text)

Arithmetic Calculator



L-attributed Grammars

- L-attributed attribute grammar: when every attribute in a production $A \rightarrow X_1 \ldots X_n$ is either
  - A synthesized attribute, *or*
  - An inherited attribute of X_j , $1 \leq j \leq n$ that only depends on
    - Attributes (synthesized or inherited) of $X_1 \ldots X_{j-1}$ to the **left** of X_j
    - Inherited attributes of A



L-attributed Grammars

Arithmetic Calculator with Variables

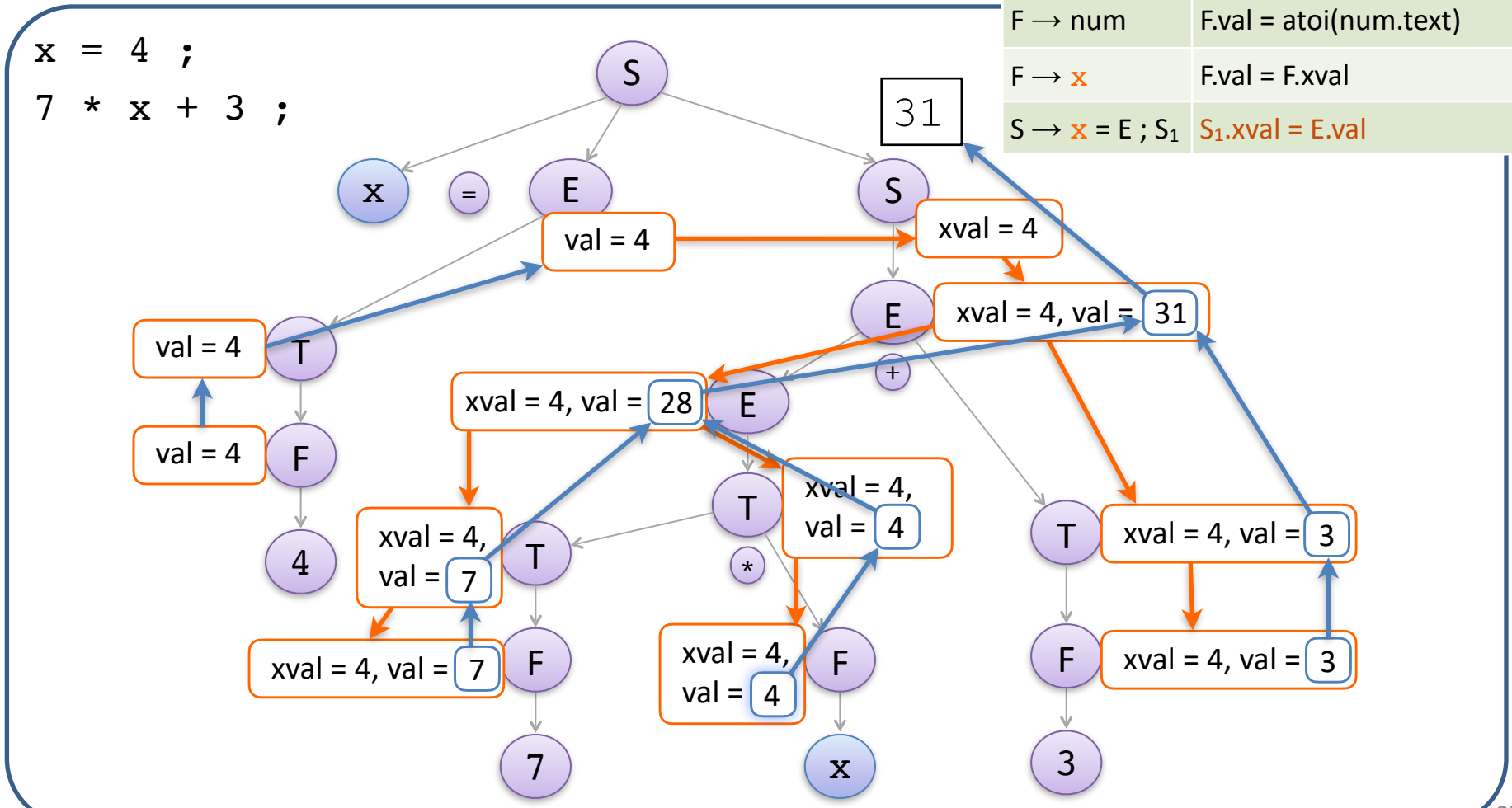
e.g., `x = 4 ; 7 * x + 3 ;`

Production	Semantic Rule	
S → E ;	print(E.val)	E.xval = S.xval
E → E_1 + T	E.val = E_1.val + T.val	E_1.xval = T.xval = E.xval
E → T	E.val = T.val	E.xval = T.xval
T → T_1 * F	T.val = T_1.val * F.val	T_1.xval = F.xval = T.xval
T → F	T.val = F.val	F.xval = T.xval
F → (E)	F.val = E.val	E.xval = F.xval
F → num	F.val = atoi(num.text)	
F → x	F.val = F.xval	
S → x = E ; S_1	S_1.xval = E.val	

?

L-attributed Grammars

S → E ;	print(E.val)	E.xval = S.val
E → E_1 + T	E.val = E_1.val + T.val	E_1.xval = T.xval = E.xval
E → T	E.val = T.val	T.xval = E.val
T → T_1 * F	T.val = T_1.val * F.val	T_1.xval = F.xval = T.xval
T → F	T.val = F.val	F.xval = T.val
F → (E)	F.val = E.val	F.xval = T.val
F → num	F.val = atoi(num.text)	
F → x	F.val = F.xval	
S → x = E ; S_1	S_1.xval = E.val	



L-attributed Grammars

- In recursive-descent parsers:
  - Pass inherited attributes down as arguments

Prod.	Semantic Rule
D → T L	L.dt = T.type
T → int	T.type = integer
T → float	T.type = float
L → id A	A.dt = L.dt addType(id.entry, L.dt)
A → , L	L.dt = A.dt
A → ε	

```
D() {
   T();
   L();
}
L() {
   match(ID);
   A();
}
A() {
   if (current == COMMA) {
      match(COMMA);
      L();
   } else if (current == EOF) {
   } else error();
}
```

```
T() {
   if (current == INT) {
      match(INT);
   } else if (current == FLOAT) {
      match(float);
   } else error();
}
```

L-attributed Grammars

- In recursive-descent parsers:
  - Pass inherited attributes down as arguments
  - Pass synthesized attributes up as return values

Prod.	Semantic Rule
D → T L	L.dt = T.type
T → int	T.type = integer
T → float	T.type = float
L → id A	A.dt = L.dt addType(id.entry, L.dt)
A → , L	L.dt = A.dt
A → ε	

```
D() {
    Attrs t = T();
    L({dt ↦ t.type});
}

L(Attrs ih) {
    Token id = match(ID);
    addType(id.entry, ih.dt);
    A({dt ↦ ih.dt});
}

A(Attrs ih) {
    if (current == COMMA) {
        match(COMMA);
        L({dt ↦ ih.dt});
    } else if (current == EOF) {
    } else error();
}
```

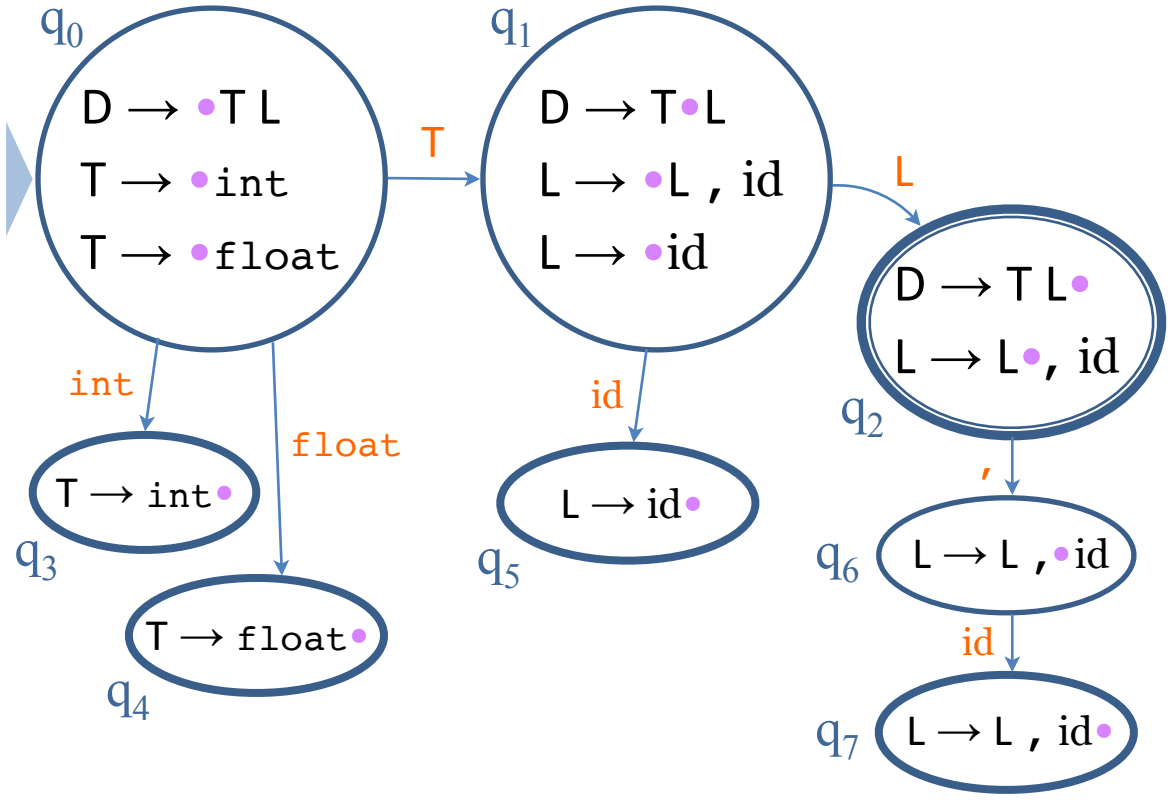
```
T() {
    if (current == INT) {
        match(INT);
        return {type ↦ int};
    } else if (current == FLOAT) {
        match(float);
        return {type ↦ float};
    } else error();
}
```

L-attributed Grammars

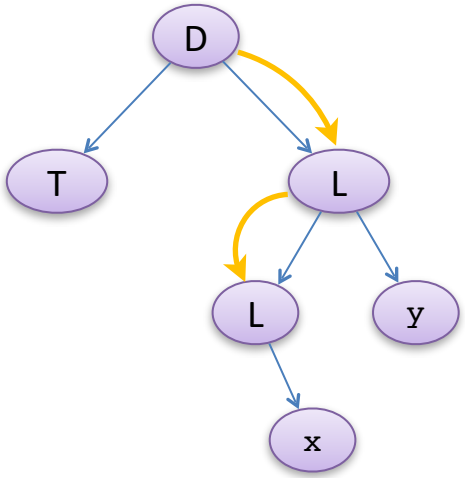
- In shift-reduce parsers:

  Semantic actions are performed during reduce.

  We have a problem — tree is built bottom-up

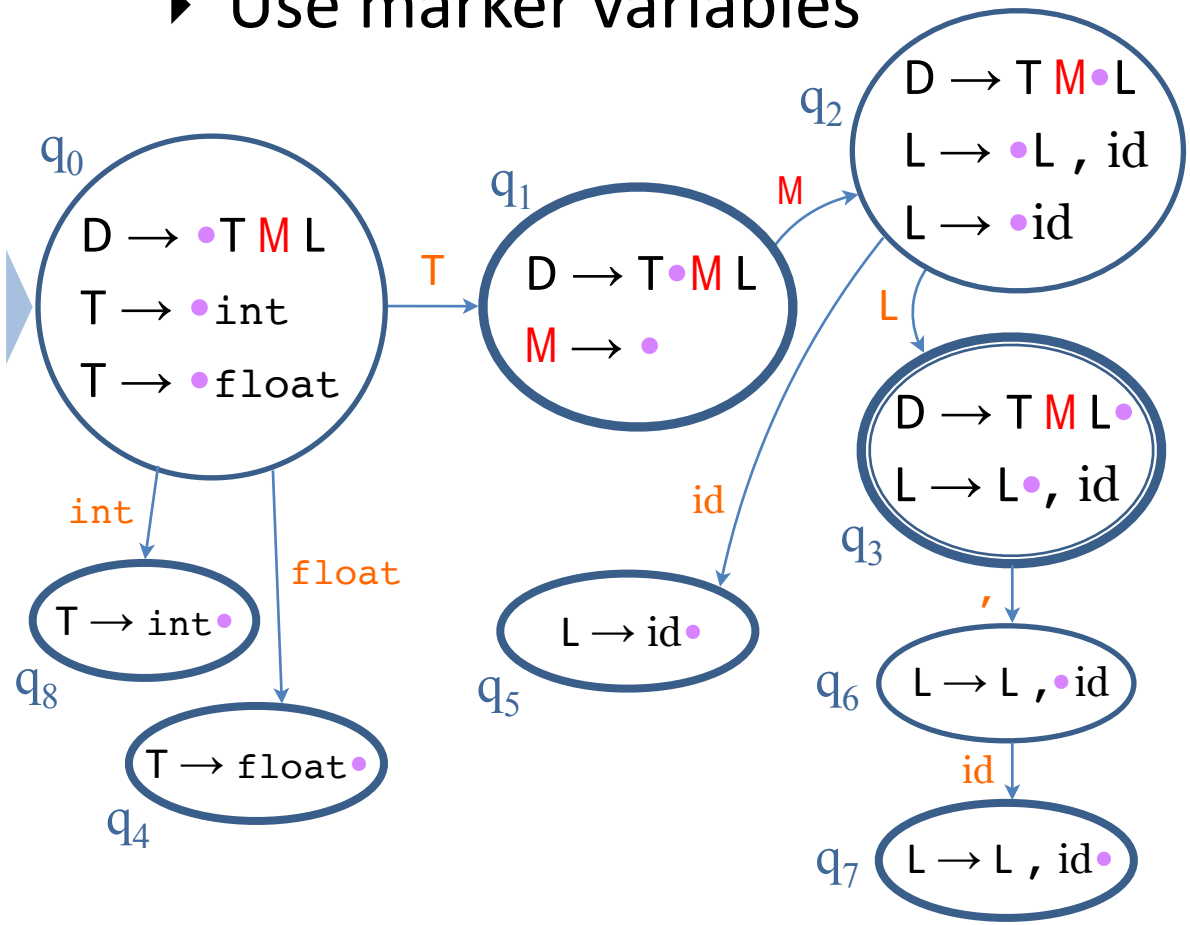


Prod.	Semantic Rule
D → T L	L.dt = T.type
T → int	T.type = integer
T → float	T.type = float
L → L_1 , id	L_1.dt = L.dt addType(id.entry, L.dt)
L → id	addType(id.entry, L.dt)

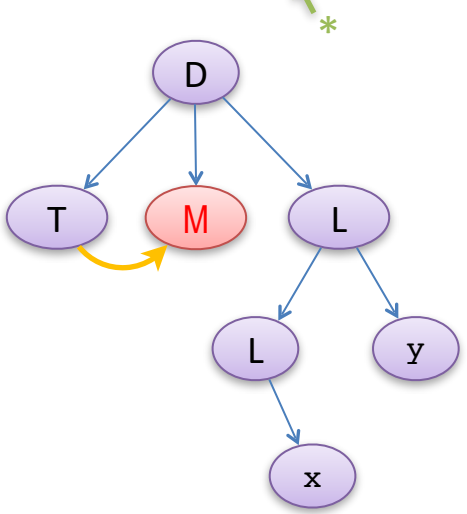


L-attributed Grammars

- In shift-reduce parsers:
  - Use marker variables



Prod.	Semantic Rule
D → T M L	dtype = null
T → int	T.type = integer
T → float	T.type = float
L → L_1 , id	addType(id.entry, dtype)
L → id	addType(id.entry, dtype)
M → ε	dtype = T.type



Marker Variables

- Since a marker only appears in one production rule, it is commonly abbreviated:

D ⟶ T M L	$action_1$
M ⟶ ε	$action_2$

⟹

D ⟶ T {$action_2$} L	$action_1$

- $action_2$ is called a mid-rule action.

It is important to remember that adding mid-rule actions **inherently changes the grammar**. The marker variable is there even if it is not explicitly visible.

Marker Variables

- In particular, marker variables and the associated ε-productions can violate your grammar's LR(0)/SLR/LALR/LR(1)-ness

Prod.	Semantic Rule
S → D	
D → T L	
D → T L [ ]	
T → int	T.type = integer
T → float	T.type = float
L → id	addType(id.entry, dtype)

This grammar is LR(0)

Prod.	Semantic Rule
S → D	
D → T M L	
D → T N L [ ]	
T → int	T.type = integer
T → float	T.type = float
L → id	addType(id.entry, dtype)
M → ε	dtype = T.type
N → ε	dtype = array(T.type)

This grammar is not even LR(1)

Semantic Checks

- Scope rules
  - Use symbol table to check:
    - No multiple definition of same identifier
    - Identifiers defined before used
- Type checking
  - Check that types in the program are consistent
    - How?

Types

- What is a type?
  - Simplest answer: a set of values
  - Integers, real numbers, booleans, …
- Why do we care?
  - Safety
    - Guarantee that certain errors cannot occur at runtime
  - Abstraction
    - Hide implementation details
  - Documentation
  - Optimization

Type Checking

- Typing rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples

string string
```
"drive" + "drink"
```
string

int string
```
42 + "the answer"
```
ERROR

Type System

- A type system of a programming language is a way to define how "good" programs behave
  - Good programs = well-typed programs
  - Bad programs = programs containing type errors

Static Typing	Dynamic Typing
most checking done at compile time	most checking done at runtime

Static Typing *vs.* Dynamic Typing

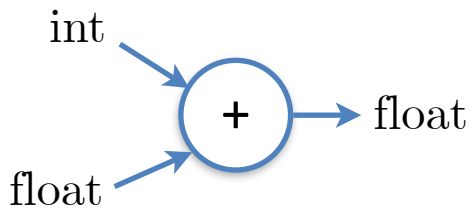
- Static type checking is conservative
  - Any program that is determined to be well-typed is free from certain kinds of errors
  - May reject programs that cannot be *statically* determined to be safe
    - Why?

```
if (some-function(x))
    y = 42;
else
    y = "the answer"
print(y * y);
```

- Dynamic type checking
  - May accept more programs as valid (runtime info)
  - Errors not caught at compile time
  - Runtime overhead

Type Checking Rules

- Specify for each operator
  - Types of operands
  - Type of result



- Basic Types
  - Building blocks for the type system (type rules)
  - *e.g.*, int, boolean, (sometimes) string

- Type Expressions
  - Array types
  - Function types
  - Record types / Classes

Typing Rules

If E_1 has type int and E_2 has type int,

then $E_1 + E_2$ has type int

$$\frac{E_1 : \text{int} \qquad E_2 : \text{int}}{E_1 + E_2 : \text{int}}$$

If E_1 has type int and E_2 has type float,

then $E_1 + E_2$ has type float

$$\frac{E_1 : \text{int} \qquad E_2 : \text{float}}{E_1 + E_2 : \text{float}}$$

Similarly,

$$\frac{E_1 : \text{int} \qquad E_2 : \text{int}}{E_1 * E_2 : \text{long}}$$

$$\frac{E_1 : \text{float} \qquad E_2 : \text{int}}{E_1 + E_2 : \text{float}}$$

…and the other way around

Basic Type System

$$\frac{}{\text{true} : \text{boolean}} \qquad \frac{}{\text{false} : \text{boolean}}$$

Constants

$$\frac{}{\text{int-literal} : \text{int}} \qquad \frac{}{\text{string-literal} : \text{string}}$$

$$\frac{E_1 : \text{int} \qquad E_2 : \text{int}}{E_1 \diamond E_2 : \text{int}} \quad \diamond \in \{ +, -, /, *, \% \}$$

Binary Arithmetic Operators

$$\frac{E_1 : \text{int} \qquad E_2 : \text{int}}{E_1 \ominus E_2 : \text{boolean}} \quad \ominus \in \{ <=, <, >, >= \}$$

Comparison Operators

$$\frac{E_1 : T \qquad E_2 : T}{E_1 \ominus E_2 : \text{boolean}} \quad \ominus \in \{ ==, != \}$$

Basic Type System

$$\frac{E_1 : \text{boolean} \qquad E_2 : \text{boolean}}{E_1 \circledast E_2 : \text{boolean}} \qquad \circledast \in \{ \&\&, \,||\, \}$$

Binary Logical Operators

$$\frac{E_1 : \text{int}}{-\, E_1 : \text{int}} \qquad \frac{E_1 : \text{boolean}}{!\, E_1 : \text{boolean}}$$

Unary Operators

$$\frac{E_1 : T[]}{E_1.\text{length} : \text{int}}$$

Array Operations

$$\frac{E_1 : T[] \qquad E_2 : \text{int}}{E_1[E_2] : T} \qquad \frac{E_1 : \text{int}}{\text{new } T[E_1] : T[]}$$

Type Checking

- Traverse AST and assign types for AST nodes
  - Use typing rules to compute node types

- Alternative: type-check during parsing
  - (Slightly) more complicated
  - But naturally also more efficient

Type Checking

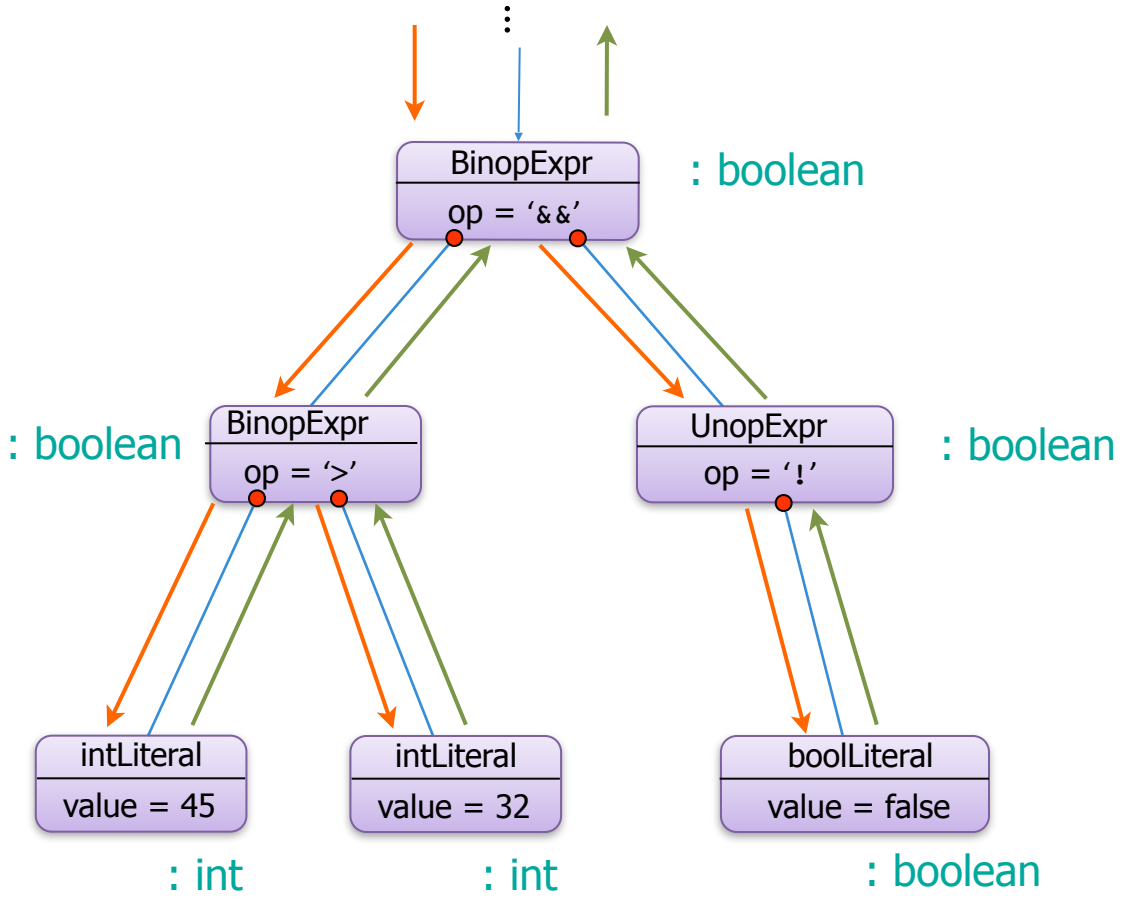
- Use semantic actions
  - In derivations corresponding to expressions, check that typing rules are satisfied

$$\text{Expr} \rightarrow \text{Expr}_1 == \text{Expr}_2$$

$$\frac{E_1 : T \qquad E_2 : T}{E_1 == E_2 : \text{boolean}}$$

```
{
    if (Expr1.type == Expr2.type)
        Expr.type = boolean;
    else error;
}
```

Type Checking



$$\frac{E_1 : \text{boolean} \qquad E_2 : \text{boolean}}{E_1 \circledast E_2 : \text{boolean}} \quad \text{for } \circledast \in \{ \&\&, \mid\mid \}$$

$$\frac{E_1 : \text{boolean}}{! E_1 : \text{boolean}}$$

$$\frac{E_1 : \text{int} \qquad E_2 : \text{int}}{E_1 \ominus E_2 : \text{boolean}} \quad \text{for } \ominus \in \{ <=, <, >, >= \}$$

$$\frac{}{\texttt{false} : \text{boolean}}$$

$$\frac{}{\text{int-literal} : \text{int}}$$

```
45 > 32 && !false
```

Strongly Typed *vs.* Weakly Typed

- Coercion
- Strongly typed
  - C, C++, Java
- Weakly typed
  - JavaScript, Perl, PHP

(Not everybody agrees on this classification)

Output: 73

Perl

```
$a = 31;
$b = "42x";
$c = $a + $b;
print $c;
```

warning: initialization makes integer from pointer without a cast

C

```
main() {
 int a=31;
 char b[3]="42x";
 int c = a + b;
}
```

error: Incompatible type for declaration. Can't convert java.lang.String to int

Java

```
public class… {
 public static void main() {
  int a = 31;
  String b ="42x";
  int c = a + b;
 }
}
```

Type Declarations

- So far, we had a fixed set of types
- But types can also be user-defined

```
type Temperature is range –20..50                    (explicitly)
type Some_Numbers is array (1..42) of Integer;

enum day_t { YESTERDAY, TODAY, TOMORROW };

series : array (1..42) of Real;                      (anonymously)

struct { float x; float y; int color; } my_point;
```

Type Declarations

```
series : array (1..42) of Real;
```

```
type type01@line_73 is array (1..42) of Real;
series : type01@line_73;
```

Type Table

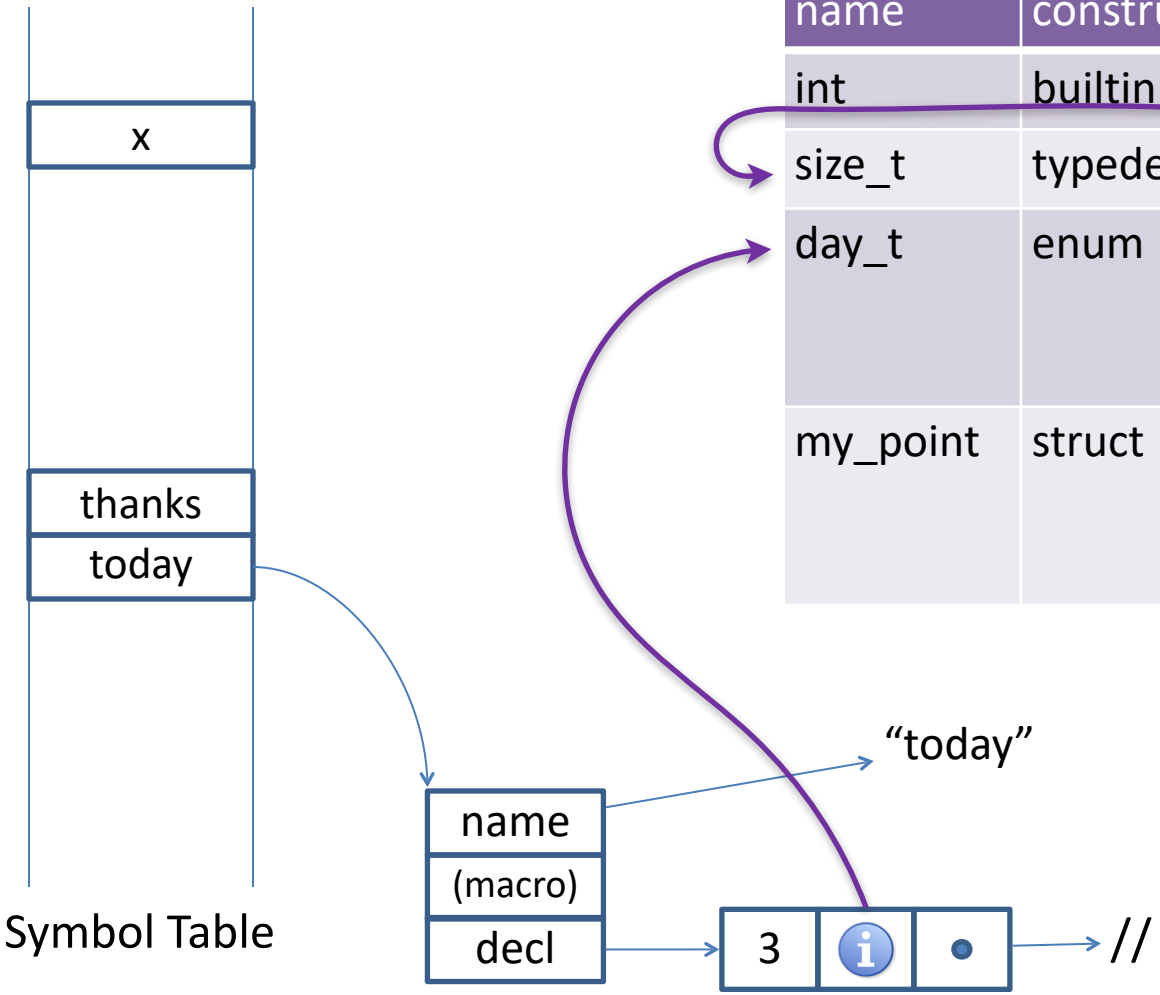
- All types in a compilation unit are collected in a type table

- For each type, its table entry contains
  - Type constructor: basic, record, array, pointer,…
  - Size and alignment requirements
    - to be used later in code generation
  - Types of components (if applicable)
    - *e.g.*, type of array element, types of record fields

Type Table + Symbol Table

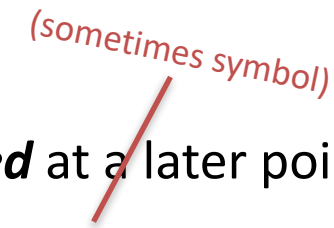
Type Table

name	constructor	structure
int	builtin	–
size_t	typedef	unsigned int
day_t	enum	0: YESTERDAY 1: TODAY 2: TOMORROW
my_point	struct	x: float y: float sz: size_t



Forward References

```
typedef struct List_Entry *Ptr_List_Entry;
⋮
struct List_Entry {
        int element;
        Ptr_List_Entry next;
};
```



- Forward references must be ***resolved*** at a later point
- A forward reference is added to the type table (as “unresolved”), and later updated when the type declaration is met
- At the end of scope, check that all forward refs have been resolved
- Must add check for circular dependencies between types

Nominal Type System

Type Equivalence = Name Equality

```
Type t1 = ARRAY[Integer] OF Integer;
Type t2 = ARRAY[Integer] OF Integer;
```

t1 not (nominally) equivalent to t2

```
Type t3 = ARRAY[Integer] OF Integer;
Type t4 = t3
```

t3 equivalent to t4

Structural Type System

Type Equivalence = Structure Isomorphism

```
Type t5 = RECORD c: Integer; p: POINTER TO t5; END RECORD;
Type t6 = RECORD c: Integer; p: POINTER TO t6; END RECORD;
Type t7 =
  RECORD
    c: Integer;
    p: POINTER TO
      RECORD
        c: Integer;
        p: POINTER TO t5;
      END RECORD;
END RECORD;
```

t5, t6, t7 are all (structurally) equivalent

In Practice

- Almost all modern languages use a nominal type system
  - Why?

- One exception to this is type aliases (C's `typedef`)

  typedef struct Node *listptr; ⟵ struct Node * ≡ listptr

- Some languages (*e.g.* Ada) have two kinds for this purpose:

  type listptr is pointer to Node ⟵ pointer to Node ≡ listptr

  type listptr is **new** pointer to Node ⟵ pointer to Node ≠ listptr

Coercions

- If we expect a value of type T_1 at some point in the program, and find a value of type T_2, is that acceptable?

```
float x = 3.141;
int y = x;
```

✗ Java
C#
Scala
Rust
Ada

✓ C
C++

Coercions

- If we expect a value of type T_1 at some point in the program, and find a value of type T_2, is that acceptable?

```
int x = 22;
float y = x;
```

✓

```
int x = 220904525;
float y = x;
// y == 220904528.0
// meh.
```

```
int x = 22;
float y = int_to_float(x);
```

l-values and r-values

```
dst := src
```

- What is `dst`? What is `src`?

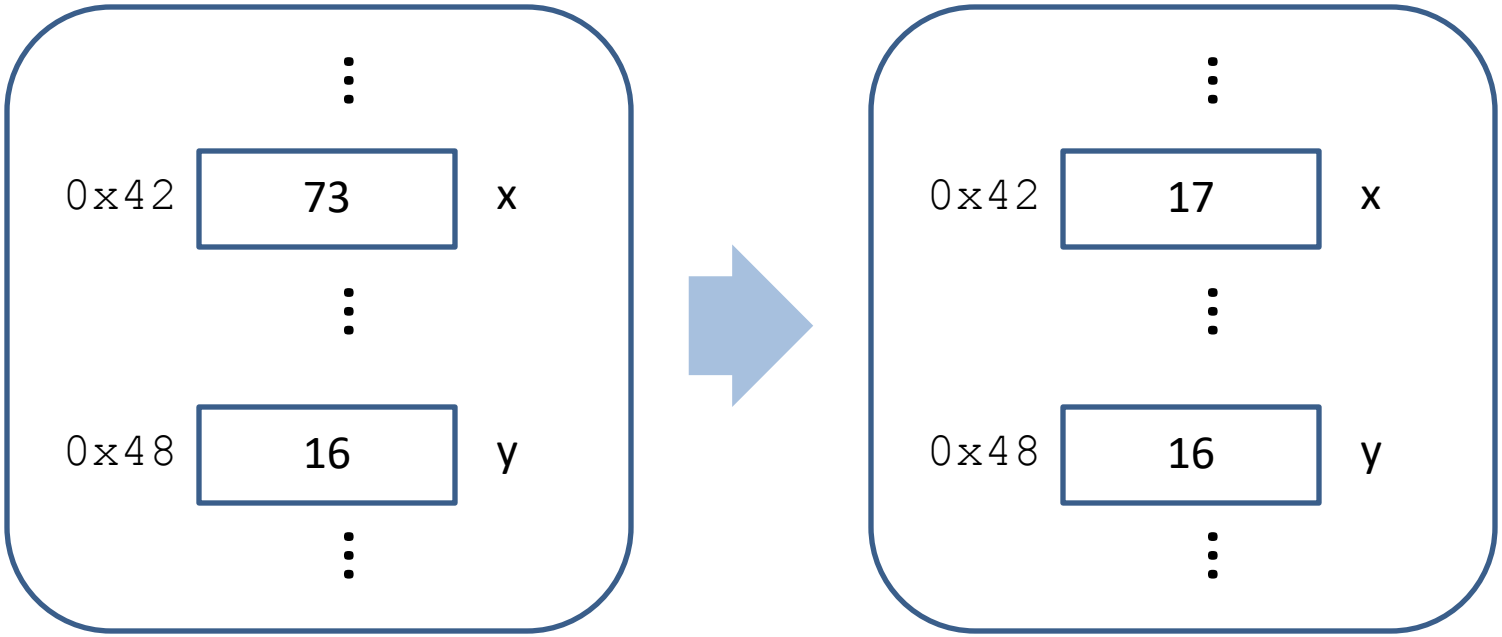
  `dst` is a memory location where data should be stored

  `src` is a plain value (a datum)

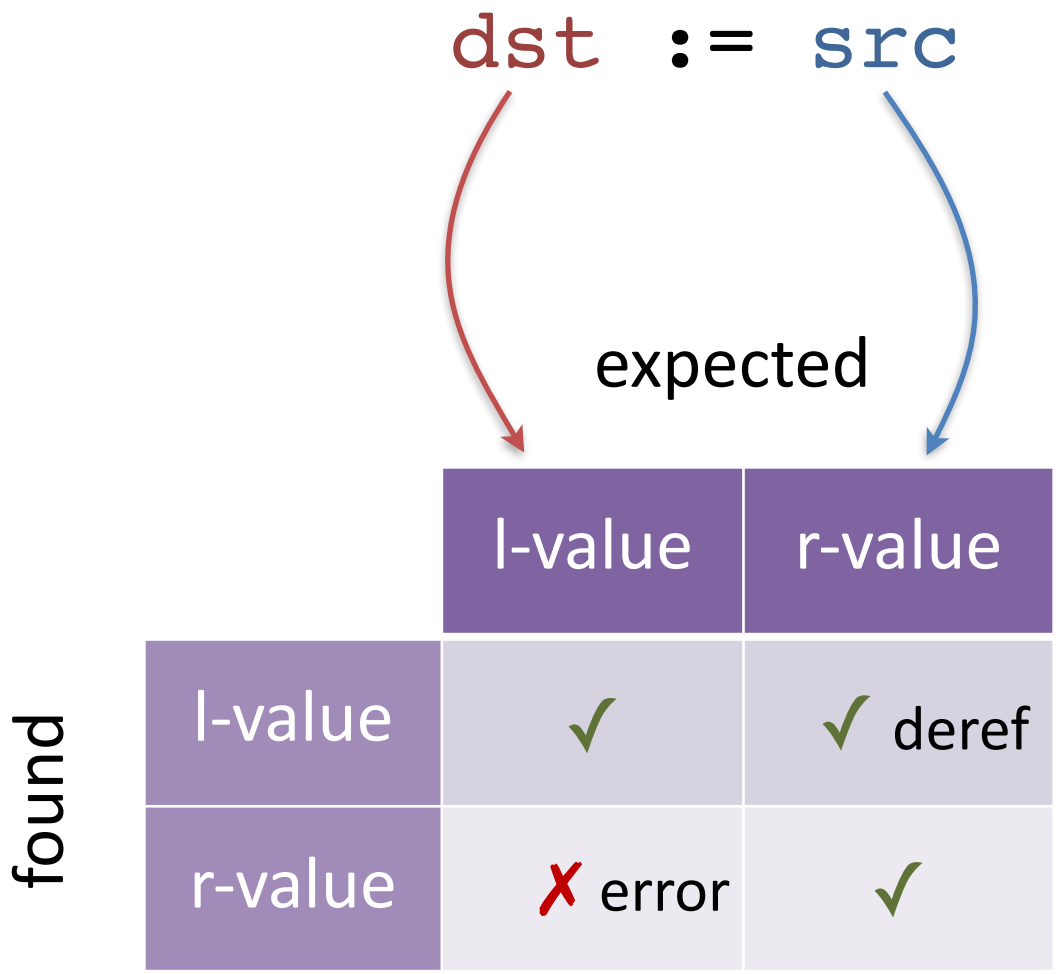
- A “location” that can be assigned to is called an l-value — it evaluates to an *address*
- A “datum” that can be read and copied is called an r-value — it evaluates to the *actual value*

l-values and r-values (example)

```
          16
x := y + 1
0x42    17
```



l-values and r-values



l-values and r-values (example)

```
x := A[1]
```
✓

```
x := A[A[1]]
```
✓ *ok*

```
A[A[1]] := x + 1
```
✓

```
x + 1 := A[1]
```
✗ *not ok*

Reference and `const` modifiers

- These fall neatly into the l-value/r-value categories:

`const int x` — `x` *is now an r-value*

`int& f()` — `f()` *is now an l-value*

`void f(int& a)` — `f`*'s argument* `a` ***expects*** *an l-value*

✳ What about `const int& y` ?

Subtyping

- A basic concept in Object-oriented Programming
  - Every class has (can have) a superclass

C++

```
class GeniusMouse : public Mouse
{
public:
  void initiatePairing();
  rk_t bond(SerialPort& port);
protected:
  …
};
```

Java

```
class GeniusMouse extends Mouse
{
  public void initiatePairing();
  public Rk bond(SerialPort socket);
  protected …
}
```

Subtyping

- Subtyping relation:
  - "T is a sub-type of S"

T <: S

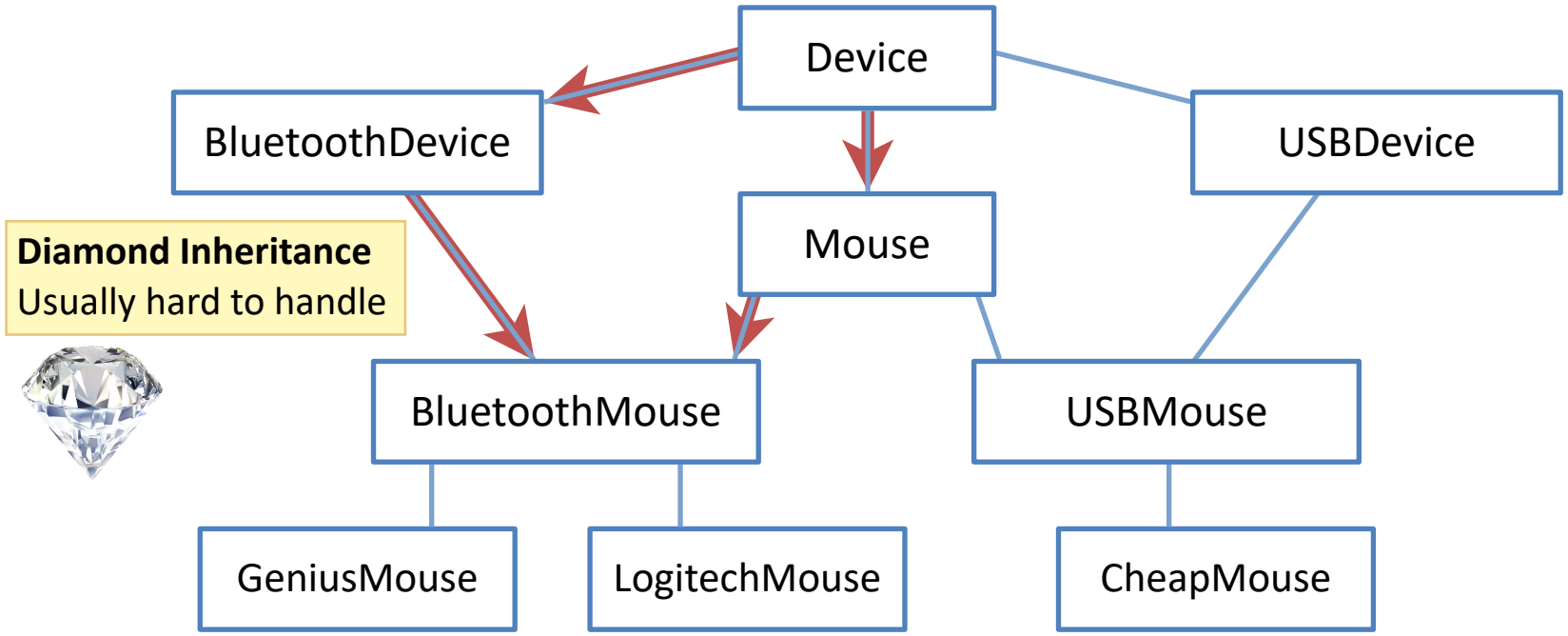
- Handled with corresponding typing rule

$$\frac{E_1 : T \qquad T <: S}{E_1 : S}$$

- As a consequence:
  - Each term has more than one type
  - Need to find the appropriate type for each context

Subtyping

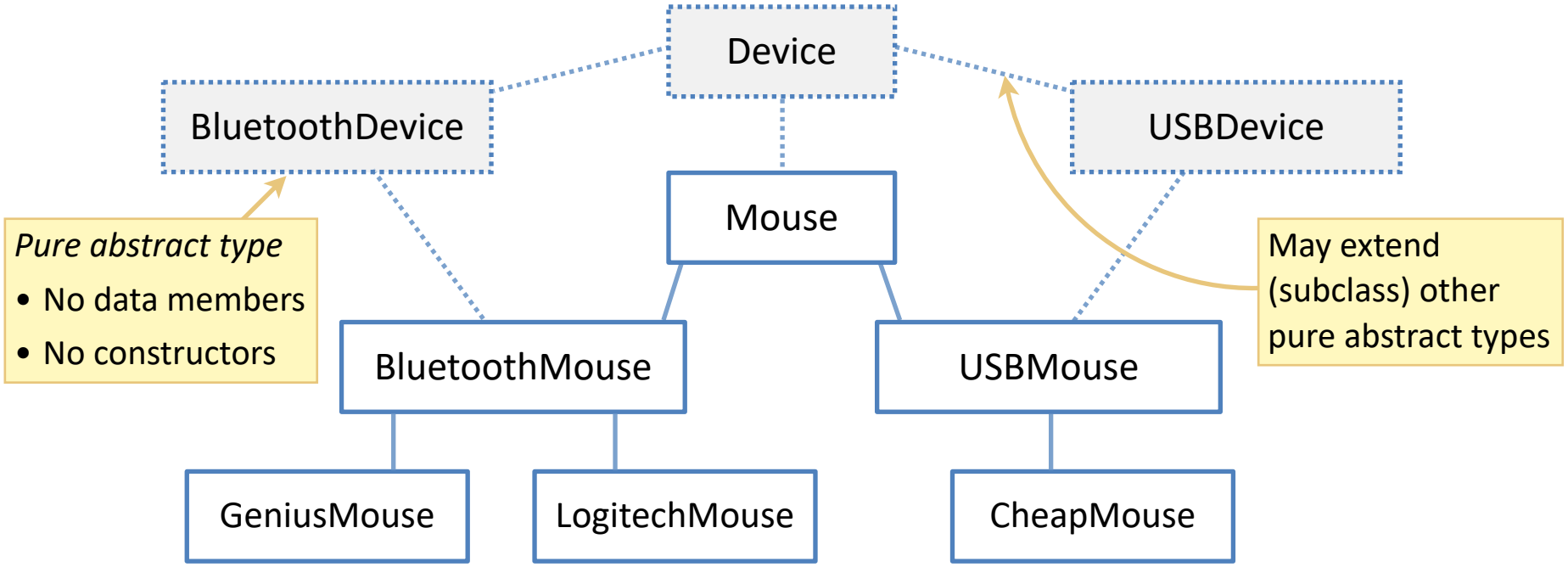
- A basic concept in Object-oriented Programming
  - Every class has (can have) a superclass



Inheritance ~~Tree~~
DAG

Subtyping

- Marking some of the types as pure abstract
  helps alleviate some of the difficulty in multiple inheritance



Inheritance ~~Tree~~ DAG

Subtyping

- Upcast
  - The compiler will insert an implicit conversion from subtype to supertype (similar to coercions)

```
Mouse m = new GeniusMouse();
registerDevice(m);
```

```
Mouse m = GeniusMouse_to_Mouse(
      new GeniusMouse());
registerDevice(Mouse_to_Device(m));
```

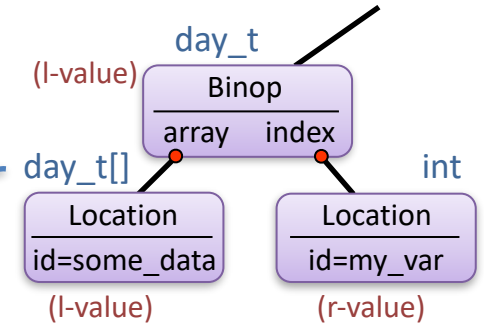
Summary

Static semantic checks

- Identification matches applied occurrences of identifier to its defining occurrence
  - The symbol table maintains this information
- Type checking checks which type combinations are legal
  - Each node in the AST of an expression represents either an l-value (location) or an r-value (value) of some type

name	pos	type
month	1	int 1..12
month_name	2	string[1..12]
day	19	day_t

name	constructor	structure
int	builtin	–
size_t	typedef	unsigned int
day_t	enum	0: YESTERDAY 1: TODAY 2: TOMORROW



Summary

- ✓ Contextual analysis can move information between nodes in the AST
  - Even when they are not “local”
- ✓ Attribute grammars
  - Attach attributes and semantic actions to grammar
- ✓ Attribute evaluation
  - Build dependency graph, topological sort, evaluate
- ✓ Special classes with pre-determined evaluation order: S-attributed, L-attributed
  - with a few tricks, attributes can be computed *while* building the AST

Coming Up