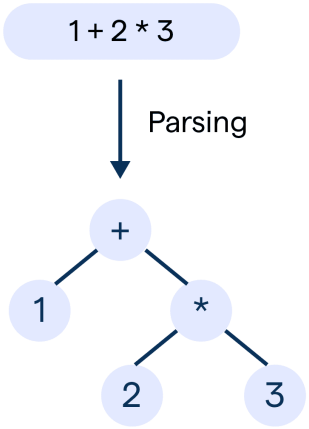
Compilation

0368-3133

Lecture 5a:

Syntax Analysis (The End)



Slides' credit: Eran Yahav, Shachar Itzhaky, Mooly Sagiv

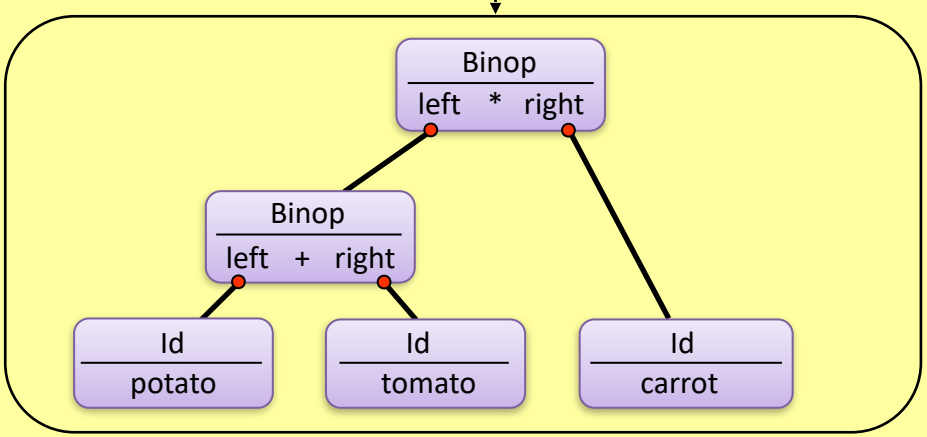
Syntax Analysis: Tokens ➡ AST

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

Lexical analyzer

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

Parser



Broad kinds of parsers

- Top-Down parsers (*LL(k) grammars*)
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-Up parsers (*LR(k) grammars*)
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation (in a reverse order)

Efficient ($O(n=|input|)$)

- Parsers for arbitrary grammars (incl. for inherently ambiguous langs.)
  - GLR (Generalized LR): "Parallel" LR parsing handling conflicts by forking
  - CYK method ($O(n^3)$)
  - Earley's method ($O(n^3)$ for ambiguoug grammars; $O(n^2)$ for unambiguoug grammars; $O(n)$ for DPDA-recognaizable grammars)
  - ▶ May generate multiple parse trees
  - ▶ Usually, not used in practice

Cocke-Younger-Kasami (CYK) [1961] Parsing Algorithm

- *CYK* (and *Earley*) use dynamic programming
- A worst case time complexity of $\mathrm{O}(n^3 \cdot |\mathrm{G}|)$

CYK by Example

G = ⟨V,T,P,S⟩ in CNF:

- $A \rightarrow BC$, where $B, C \in V$
- $A \rightarrow t$, where $t \in T$

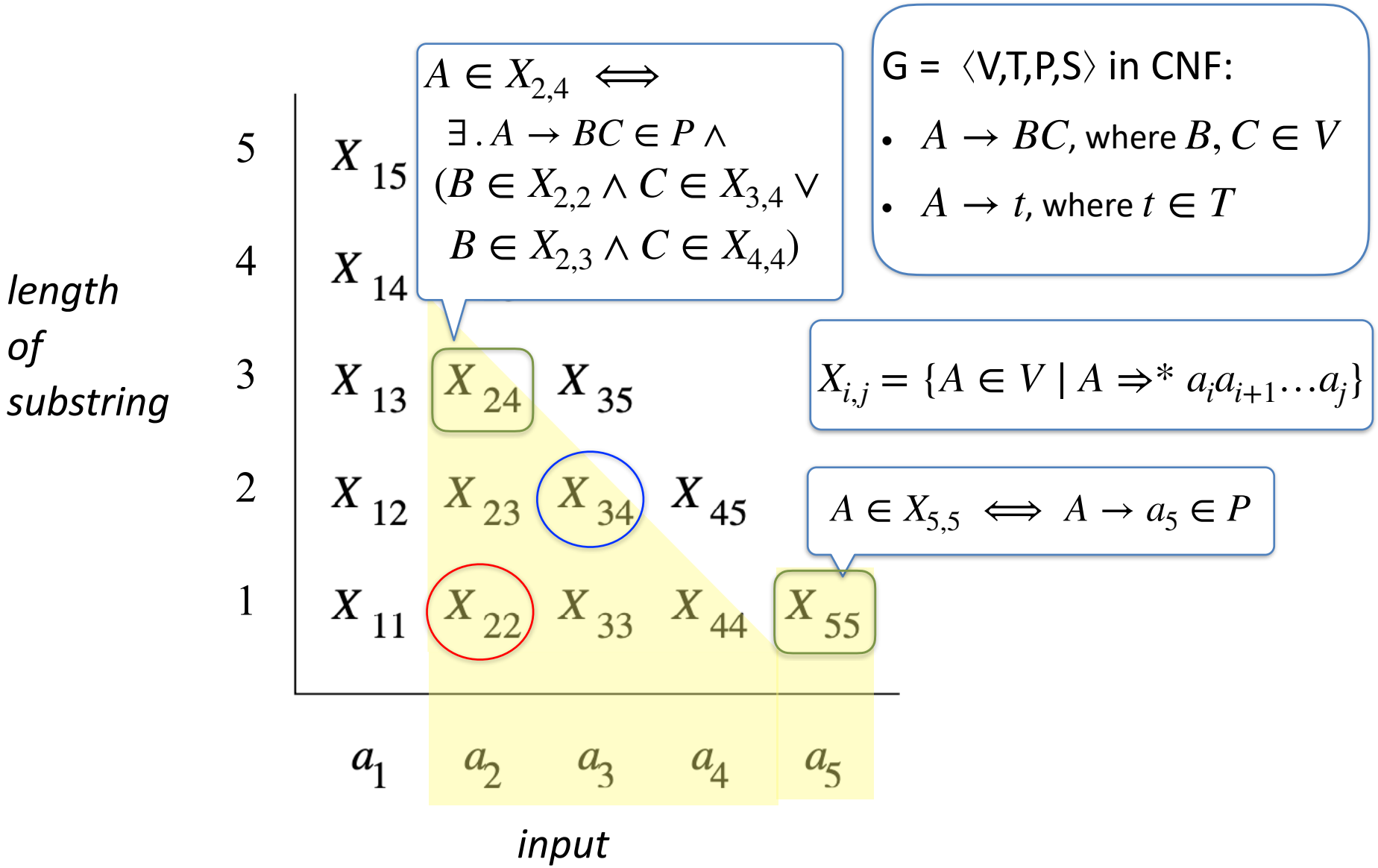
$$X_{i,j} = \{A \in V \mid A \Rightarrow^* a_i a_{i+1} \ldots a_j\}$$

length of substring

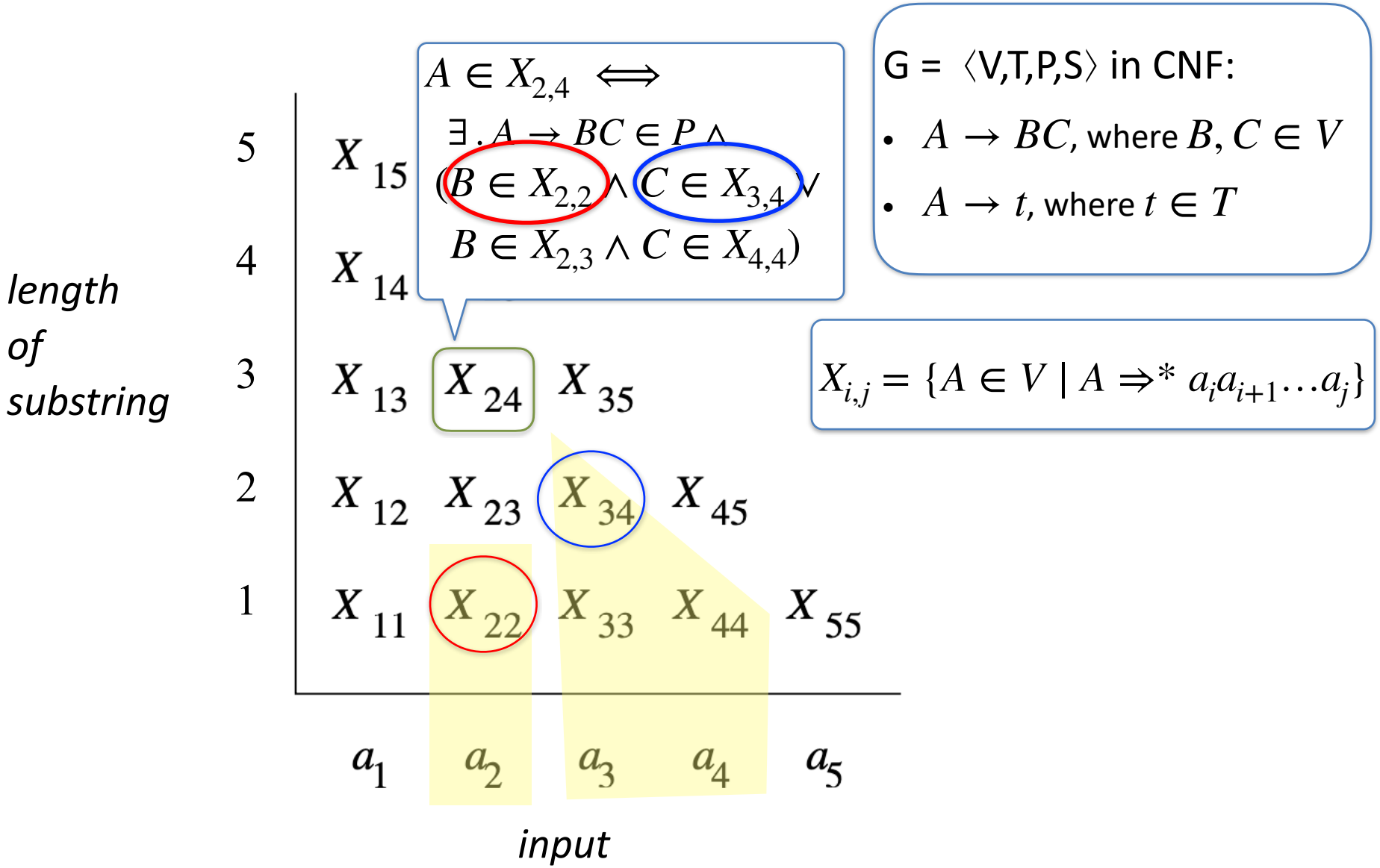
5	X_{15}				
4	X_{14}	X_{25}			
3	X_{13}	X_{24}	X_{35}		
2	X_{12}	X_{23}	X_{34}	X_{45}	
1	X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
	a_1	a_2	a_3	a_4	a_5

input

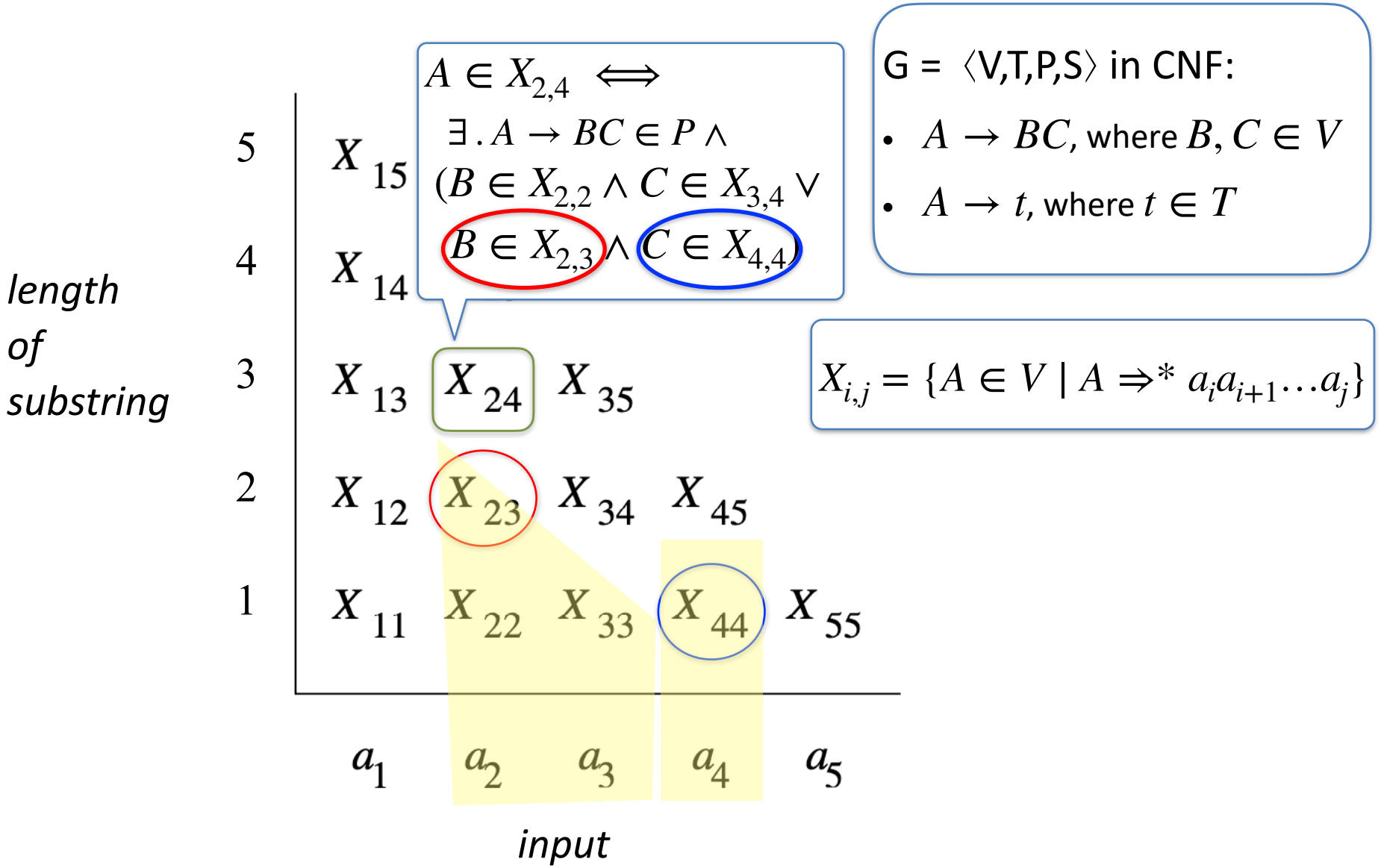
CYK by Example



CYK by Example



CYK by Example



CYK Algorithm

$G = \langle V,T,P,S \rangle$

input = array[1..n]

// assume $X_{j,i} = \{\}$ for $1 \le j \le i \le n$

for m = 1 … n:

$X_{m,m} = \{ A \in V \mid A \longrightarrow input[m] \in P \}$

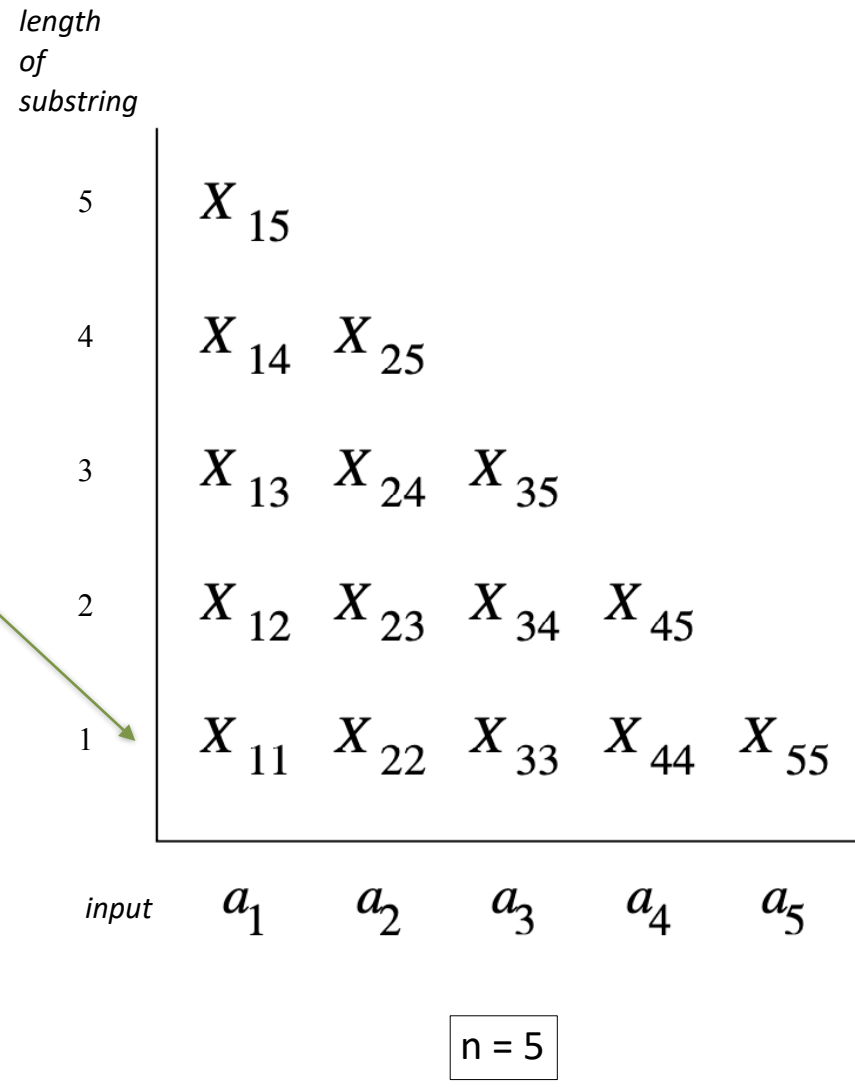
for i = 2 … n: // length of span

for j = 1 … n-(i-1): // start position of span

for k = j … j+(i-2): // partition of span

$X_{j,j+(i-1)} = X_{j,j+(i-1)} \cup \{A \in V \mid A \rightarrow BC \in P \wedge B \in X_{j,k} \wedge C \in X_{k+1,j+(i-1)}\}$

if $S \in X_{1,n}$ return ACCECT else return REJECT



n = 5

CYK Algorithm

$G = \langle V,T,P,S \rangle$

input = array[1..n]

// assume $X_{j,i} = \{\}$ for $1 \le j \le i \le n$

for m = 1 … n:

$X_{m,m} = \{ A \in V \mid A \longrightarrow input[m] \in P \}$

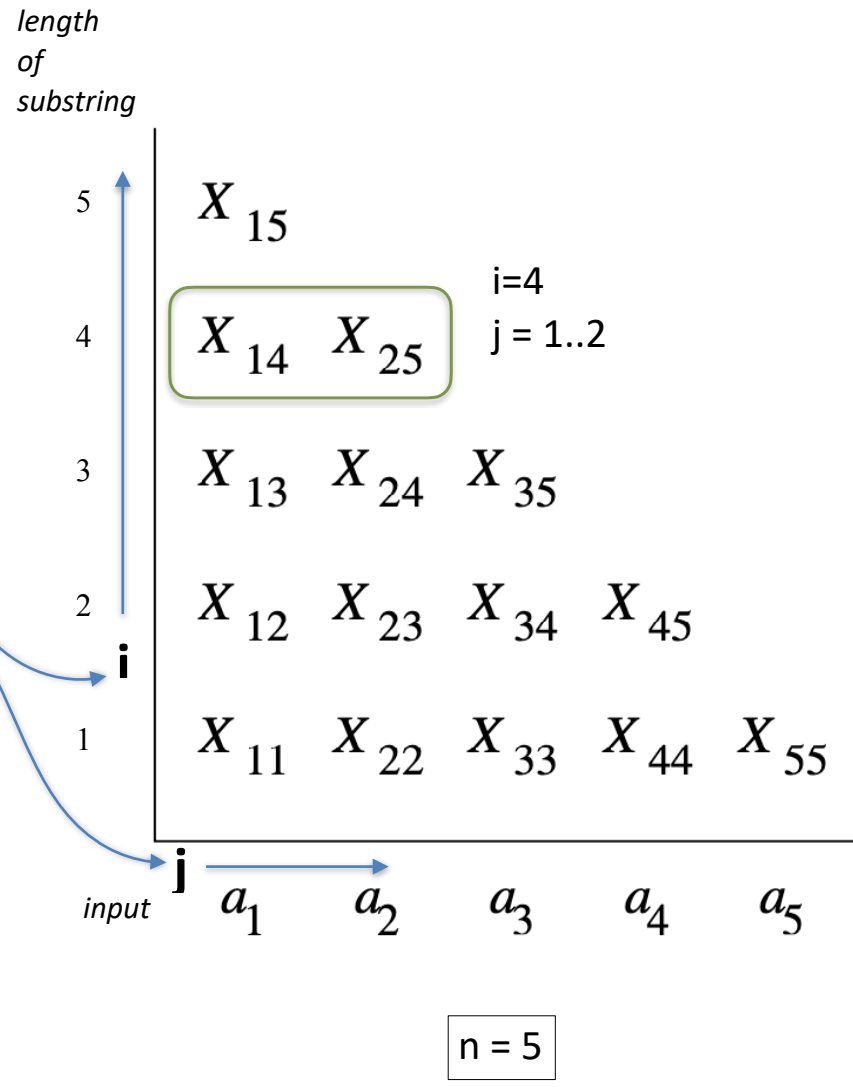
for i = 2 … n: // length of span

for j = 1 … n-(i-1): // start position of span

for k = j … j+(i-2): // partition of span

$X_{j,j+(i-1)} = X_{j,j+(i-1)} \cup \{A \in V \mid A \rightarrow BC \in P \wedge B \in X_{j,k} \wedge C \in X_{k+1,j+(i-1)}\}$

if $S \in X_{1,n}$ return ACCECT else return REJECT



CYK Algorithm

$G = \langle V,T,P,S \rangle$

input = array[1..n]

// assume $X_{j,i} = \{\}$ for $1 \le j \le i \le n$

for m = 1 ... n:

$X_{m,m} = \{ A \in V \mid A \longrightarrow input[m] \in P \}$

for i = 2 ... n: // length of span

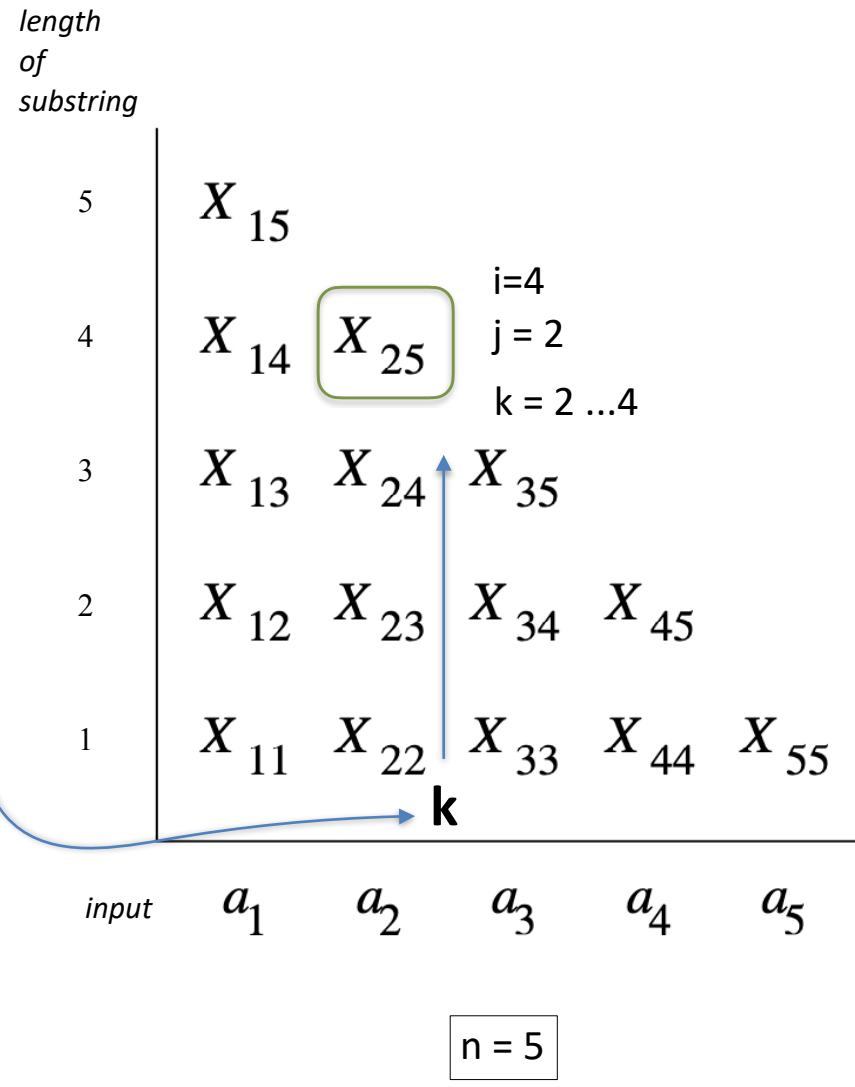
for j = 1 ... n-(i-1): // start position of span

for k = j ... j+(i-2): // partition of span

$X_{j,j+(i-1)} = X_{j,j+(i-1)} \cup \{A \in V \mid A \rightarrow BC \in P \wedge B \in X_{j,k} \wedge C \in X_{k+1,j+(i-1)}\}$

$X_{2,5}$

if $S \in X_{1,n}$ return ACCECT else return REJECT



n = 5

CYK Algorithm

$G = \langle V,T,P,S \rangle$

input = array[1..n]

// assume $X_{j,i} = \{\}$ for $1 \le j \le i \le n$

for m = 1 ... n:

$X_{m,m} = \{ A \in V \mid A \rightarrow input[m] \in P \}$

for i = 2 ... n: // length of span

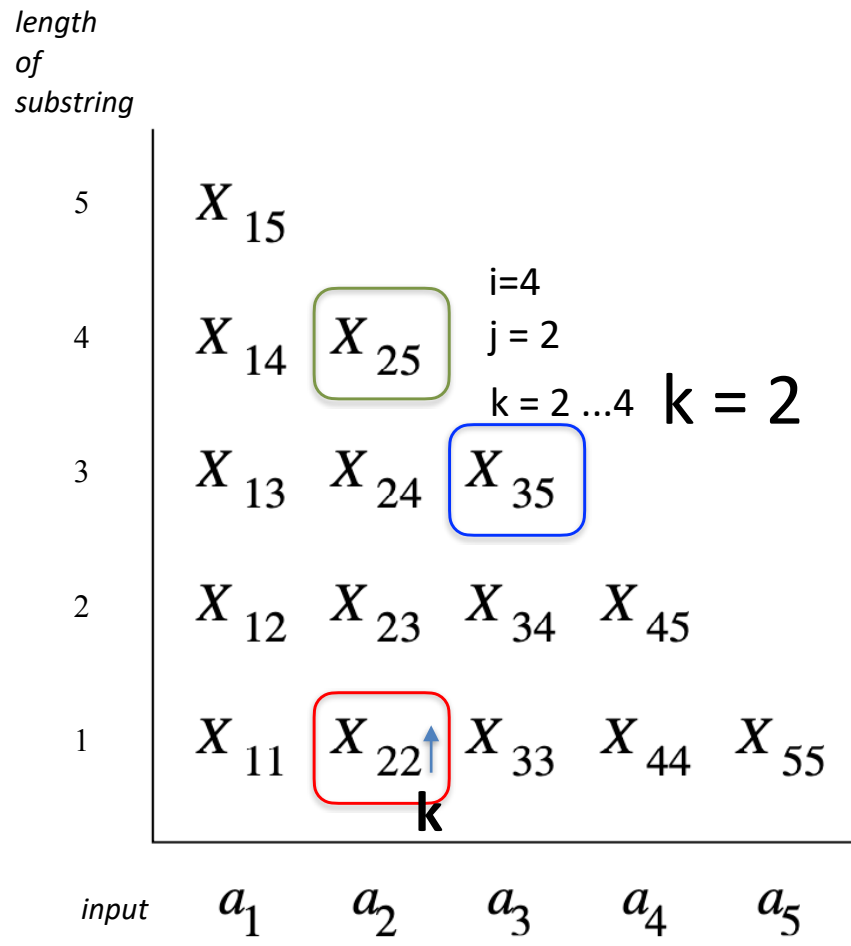
for j = 1 ... n-(i-1): // start position of span

for k = j ... j+(i-2): // partition of span

$X_{j,j+(i-1)} = X_{j,j+(i-1)} \cup \{A \in V \mid A \rightarrow BC \in P \wedge B \in X_{j,k} \wedge C \in X_{k+1,j+(i-1)} \}$

$X_{2,5}$

if $S \in X_{1,n}$ return ACCECT else return REJECT



$X_{2,5} = \{ A \in V \mid A \rightarrow BC \in P \wedge B \in X_{2,2} \wedge C \in X_{3,5} \}$

CYK Algorithm

$G = \langle V,T,P,S \rangle$

input = array[1..n]

// assume $X_{j,i} = \{\}$ for $1 \le j \le i \le n$

for m = 1 ... n:

$X_{m,m} = \{ A \in V \mid A \rightarrow input[m] \in P \}$

for i = 2 ... n: // length of span

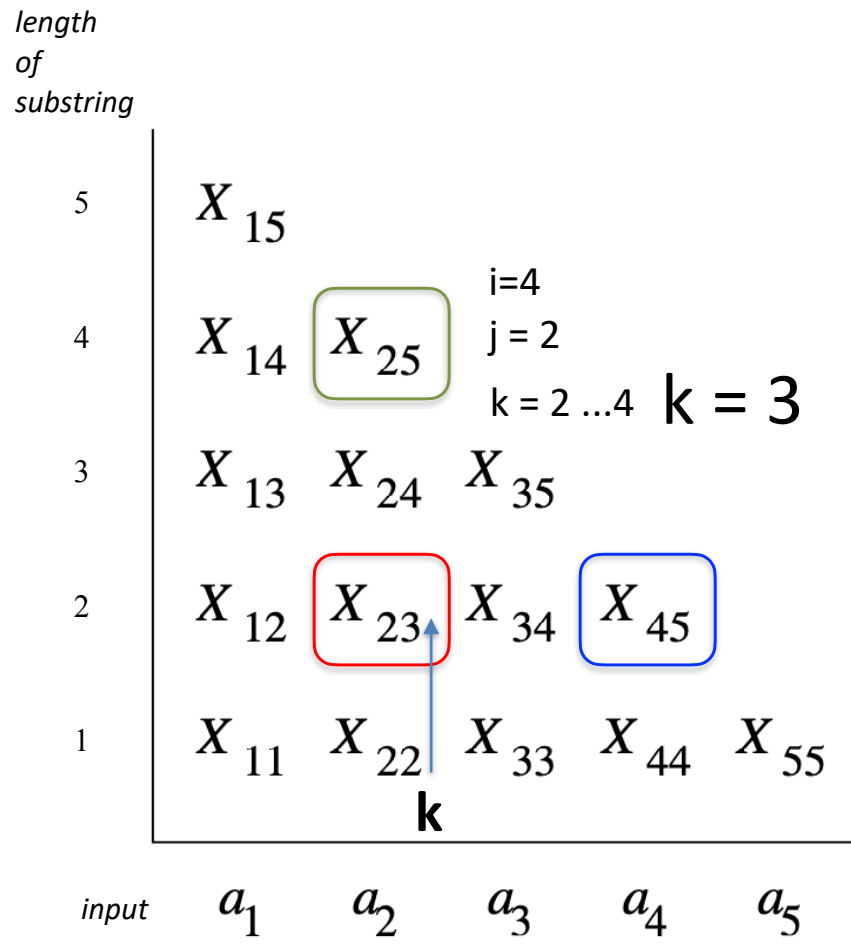
for j = 1 ... n-(i-1): // start position of span

for k = j ... j+(i-1): // partition of span

$X_{j,j+(i-1)} = X_{j,j+(i-1)} \cup \{A \in V \mid A \rightarrow BC \in P \wedge B \in X_{j,k} \wedge C \in X_{k+1,j+(i-1)}\}$

$X_{2,5}$

if $S \in X_{1,n}$ return ACCECT else return REJECT



$X_{2,5} = \{ A \in V \mid A \rightarrow BC \in P \wedge B \in X_{2,2} \wedge C \in X_{3,5} \}$

$\cup \{ A \in V \mid A \rightarrow BC \in P \wedge B \in X_{2,3} \wedge C \in X_{4,5} \}$

CYK Algorithm

$G = \langle V,T,P,S \rangle$

input = array[1..n]

// assume $X_{j,i} = \{\}$ for $1 \le j \le i \le n$

for m = 1 ... n:

$X_{m,m} = \{ A \in V \mid A \rightarrow input[m] \in P \}$

for i = 2 ... n: // length of span

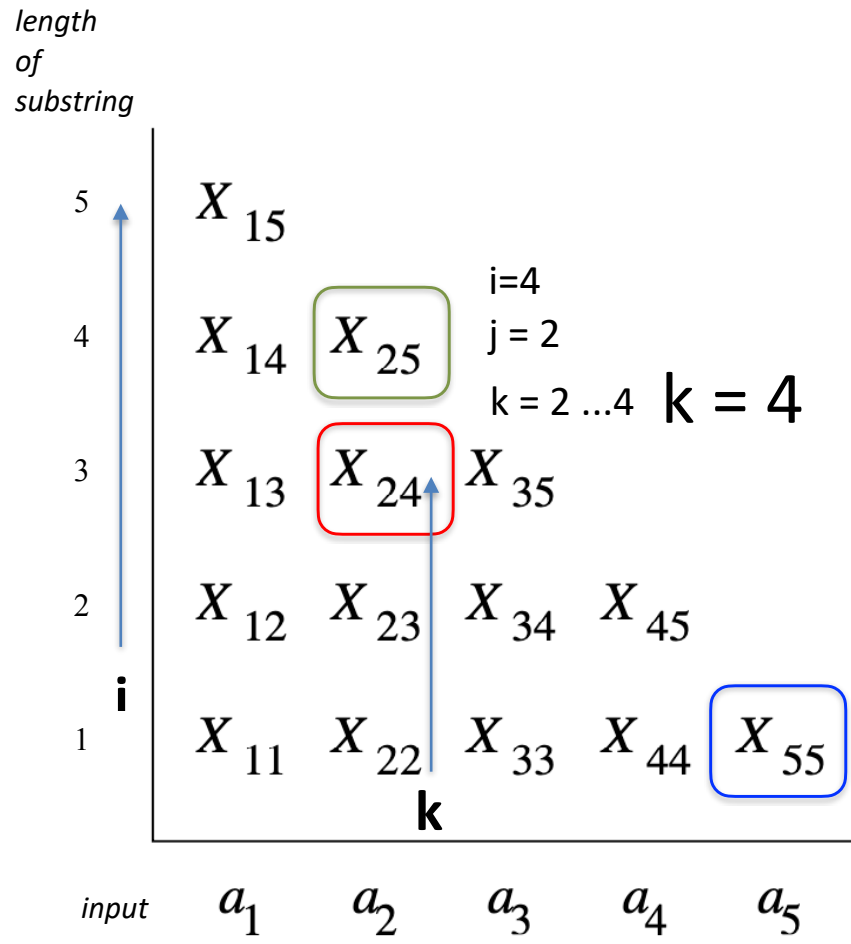
for j = 1 ... n-(i-1): // start position of span

for k = j ... j+(i-1): // partition of span

$X_{j,j+(i-1)} = X_{j,j+(i-1)} \cup \{A \in V \mid A \rightarrow BC \in P \wedge B \in X_{j,k} \wedge C \in X_{k+1,j+(i-1)} \}$

$X_{2,5}$

if $S \in X_{1,n}$ return ACCECT else return REJECT



n = 5

$X_{2,5} = \{ A \in V \mid A \rightarrow BC \in P \wedge B \in X_{2,2} \wedge C \in X_{3,5} \}$

$\cup \{ A \in V \mid A \rightarrow BC \in P \wedge B \in X_{2,3} \wedge C \in X_{4,5} \}$

$\cup \{ A \in V \mid A \rightarrow BC \in P \wedge B \in X_{2,4} \wedge C \in X_{5,5} \}$

Syntax Analysis: Concluding Words

Summary: Top-Down Parsing

- ✓ LL(k) grammars
  - ✓ FIRST & FOLLOW sets
- ✓ Top-down analysis (finds a leftmost derivation)
- ✓ LL(k) parsers
  - ✓ LL(k) parsing with recursive descent (manual)
  - ✓ LL(k) parsing with pushdown automata (generated)
- ✓ Deficiencies of LL(k) parsing
  - ✓ Cannot deal with common prefixes and left recursion
  - ✓ Mitigations: left-factoring, substitution, left recursion removal (might result in a complicated grammar)
- ✓ Building the AST
- ✓ Error handling

Summary: Bottom Up-Parsing

- ✓ LR(k) grammars
- ✓ Bottom-up analysis (finds a rightmost derivation "in reverse")
- ✓ LR(k) parsers (generated)
  - ✓ Use a table and a stack to find a derivation (by reductions)
  - ✓ Definition of LR Items and the automaton.
  - ✓ Creating the table from the automaton.
  - ✓ LR(0), SLR, LR(1), LALR – different LR items, same basic algorithm
    - Same ideas for LR(k>1), but not used in practice

LR(2) is More Powerful than LR(1), But…

- ✓ LR(2) uses the same algorithm as LR(1) but with LR(2) items
- Tables to large to be used in practice
  - LR(k) tables are exponential in k

What would LR(2) look like?

	ACTION															GOTO		
	id,id	*,id	=,id	$,id	id,*	*,*	=,*	$,*	id,=	*,=	=,=	=,$	id,$	*,$	=,$	$,$	S	R	L
0		s5	s5	s5	s4	s4											r1	r3	r2
1																acc			

q_0

S' → ● S , $$
S → ● L = R , $$
S → ● R , $$
L → ● * R , =*
L → ● * R , =id
L → ● id , = *
L → ● id , = id
R → ● L , $$
L → ● id , $$
L → ● * R , $$

id

*

(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L

LR(2) is More Powerful than LR(1), But…

- ✓ LR(2) uses the same algorithm as LR(1) but with LR(2) items
- Tables to large to be used in practice
  - LR(k) tables are exponential in k

- **Theorem**: Any language that has an LR(*k*) grammar also has an LR(*1*) grammar
  - Though with a **much** larger grammar

Summary: Bottom Up-Parsing

- ✓ LR(k) grammars
- ✓ Bottom-up analysis (finds a rightmost derivation "in reverse")
- ✓ LR(k) parsers (generated)
  - ✓ Use a table and a stack to find a derivation (by reductions)
  - ✓ Definition of LR Items and the automaton.
  - ✓ Creating the table from the automaton.
  - ✓ LR(0), SLR, LR(1), LALR – different LR items, same basic algorithm
    - Same ideas for LR(k>1), but not used in practice
- ✓ Error handling via panic mode/error derivation
- ✓ Creating the parse tree & AST
- ✓ Automatic creation of parser via CUP/Bison

LR is More Powerful than LL, But…

- Theroem: Any LL(*k*) language is also in LR(*k*) (and not vice versa); *i.e.*, LL(*k*) ⊂ LR(*k*)
- **But** the lookahead is counted differently in the two cases:

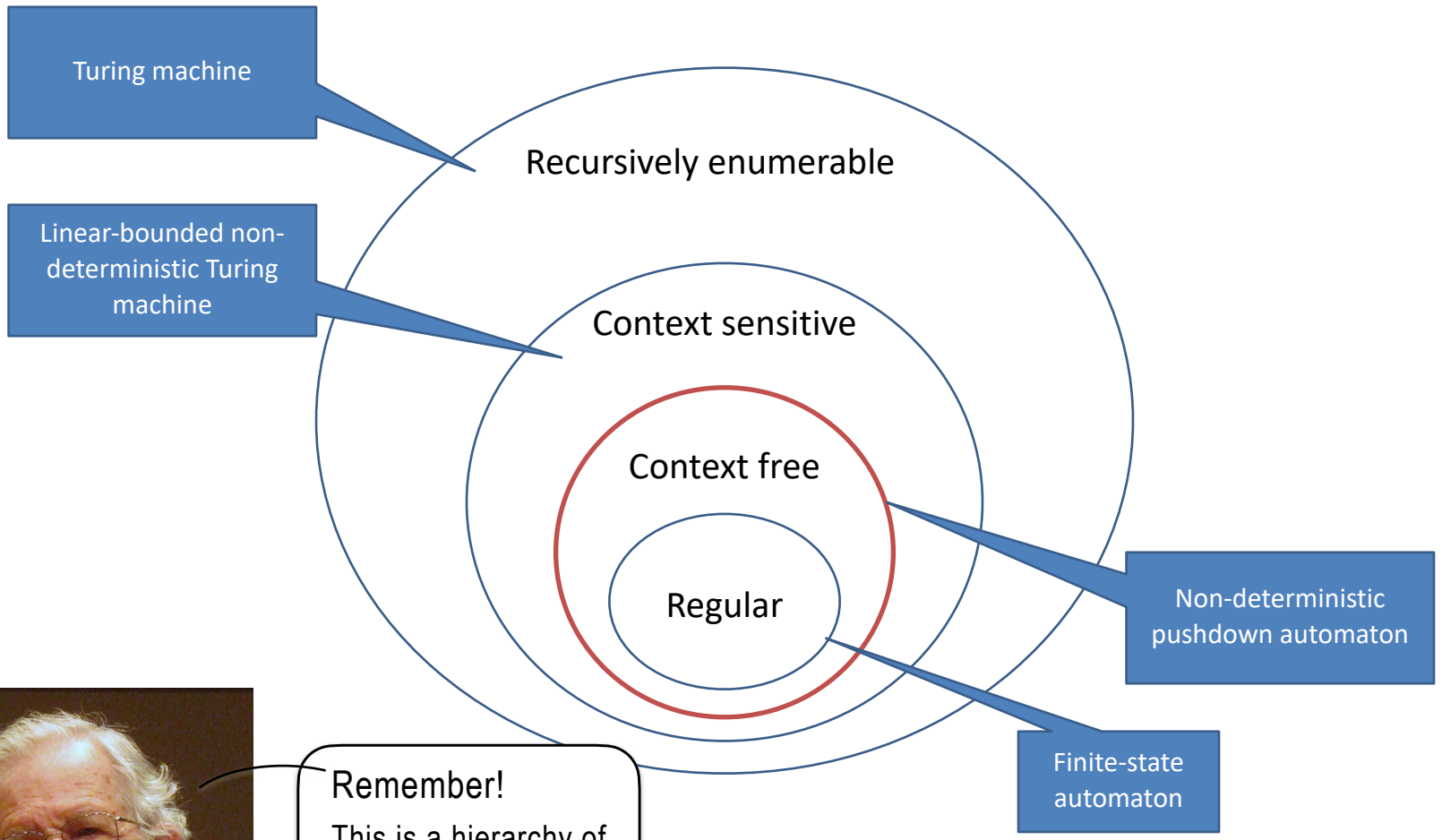
With LL(*k*), the algorithm sees *k* tokens of the input and selects a derivation rule based on (i) the expanded non-terminal and (ii) the first k tokens derivable from the rule's right-hand side

With LR(*k*), the algorithm sees the entire right-hand side of the derivation rule plus *k* more input tokens

- LR(0) sees the *entire right-side*

- The LR family of parsers is more popularly used today
  - *…but LL fights back (lookup ANTLR)*

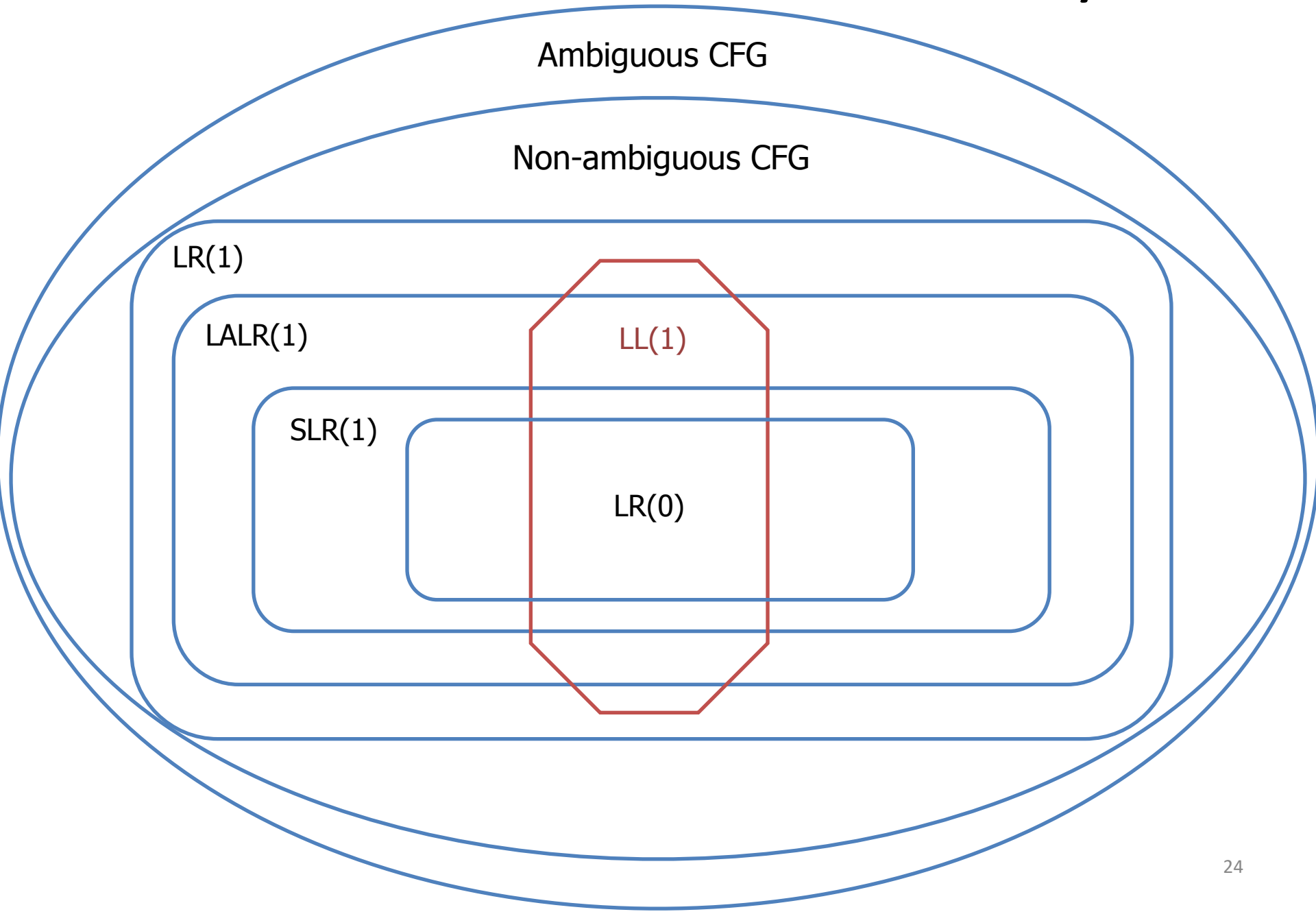
Chomsky Hierarchy



Remember!

This is a hierarchy of *languages*.

"Useful" Grammar Hierarchy



Home Exercise

- Not graded
- Can help you see if you understand syntax analysis
  - At least good enough to succeed in some previous exams
- Exams file contains **some** solutions

Question from Exam 2021/22 (B)

שאלה 1 (20 נקודות)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (3-1 שורות)

1. האם הוא מתרחש במהלך בניית הקומפיילר / הקומפילציה / linking / loading / ריצת התוכנית.
2. אם בחרתם בזמן קומפילציה ,הסבירו מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלבנטיים.
3. אם מאורע יכול להתרחש במספר שלבים, ציינו את כולם.

א. (4 נקודות) התגלה כי x הוא שם של Identifier בתוכנית.
ב. (4 נקודות) נמצא כי ניתן להימנע מקונפליקט shift/reduce ע"י הגדרת אסוציאטיביות שמאלית של אופרטור +.
ג. (4 נקודות) הוחלט כי המשתנה הלוקאלי y ימוקם בכתובת 0x00000800.
ד. (4 נקודות) הוחלט כי המשתנה הגלובאלי g ימוקם בכתובת 0x00000800.
ה. (4 נקודות) התגלה כי p הוא שם של פרוצדורה בתוכנית.

Question from Exam 2023/24 (A)

שאלה 2 (25 נקודות) (השאלה ממשיכה גם בעמוד הבא)

נתון הדקדוק המתאר מספר רציונלי בגודל לא חסום הרשום על פי השיטה העשרונית:

S → N.N$
N → N dig | dig

כאשר S ו N הם ה non-terminals (S הוא ה non-terminal הראשי). ה terminals הם dig (ספרה יחידה 0..9) ונקודה (.) כרגיל, $ מציין את סוף הקלט.

א. (4 נקודות) האם השפה שמתאר הדקדוק הינה רגולרית או חסרת הקשר? נמקו.
ב. (3 נקודות) האם הדקדוק הינו LL(1)? נמקו.
ג. (3 נקודות) האם הדקדוק הינו LR(0)? נמקו.
ד. (5 נקודות) נחליף את החוק של N בחוק הבא N→ N dig | dig | ε. האם הדקדוק החדש שהתקבל הינו LR(0), SLR, או LR(1)? נמקו.
ה. (10 נקודות) עבור הדקדוק המקורי, נרצה לגלות כתכונה סמנטית של S את ערכו השלם של המספר (whole), כאשר העיגול תמיד מתבצע כלפי מטה. כלומר, הערך השלם של 4.3 ו 4.7 הינו 4. בכדי למנוע overflow , במידה ומתקבל כי ערך זה גדול מ 1,000,000 ערכה של התכונה הסמנטית ייקבע ל 1,000,001 . האם ניתן לחשב את התכונה הסמנטית whole באמצעות Attribute Grammar? (ניתן להוסיף תכונות סמנטיות כרצונכם.) אם כן – רשמו דקדוק Attribute Grammar מתאים. אם לא – נמקו.

Semantic Analysis

Question from Exam 2024/25 (B)

שאלה 2 (22 נקודות)
נתון דקדוק הבא.

S → M $
M → M ^ M | n

כאשר S ו M הם ה non-terminals (S הוא ה non-terminal הראשי). ה terminals הם n ו ^. כרגיל, $ מציין את סוף הקלט.

א. (3 נקודות) מה היא השפה L המתקבלת ע"י הדקדוק. האם ניתן לקבלה ע"י אוטומט סופי? נמקו.
ב. (3 נקודות) האם הדקדוק רב משמעי? נמקו.
ג. (4 נקודות) כתבו דקדוק LR(0) המקבל את L. הראו כי הדקדוק LR(0). נמקו מדוע הדקדוק מקבל את L.
ד. (4 נקודות) כתבו דקדוק SLR שאינו LR(0) המקבל את L. הראו כי הדקדוק SLR אך אינו LR(0). נמקו מדוע הדקדוק מקבל את L.
ה. (8 נקודות) נניח כי ל n יש תכונה סמנטית val מסוג מספר שלם אי שלילי וכי הסימן ^ מייצג העלאה בחזקה. נרצה לחשב תחת הבנה זו את ערכם הסמנטי של מילים בשפה כאשר, כרגיל, להעלאה בחזקה יש אסוציאטיביות ימנית. לדוגמא, ערכה הסמנטי של המילה 2^3^2 הינו 512 (2 בחזקת 9) ושל המילה 2^3^0 הינו 2. האם ניתן לחשב ערך זה באמצעות Attribute Grammars עבור הדקדוקים שכתבתם בסעיפים ג' ו ד'? אם כן – רשמו Attribute Grammars מתאימים (אתם רשאים להגדיר Attribute Grammars שונים

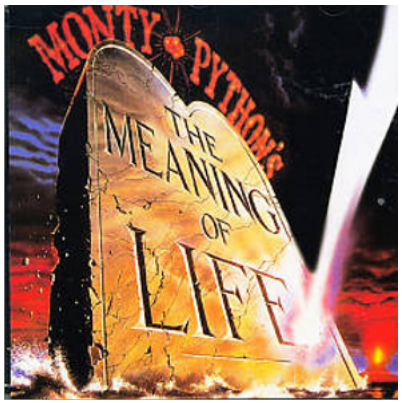
Semantic Analysis

Compilation

0368-3133

Lecture 5b:

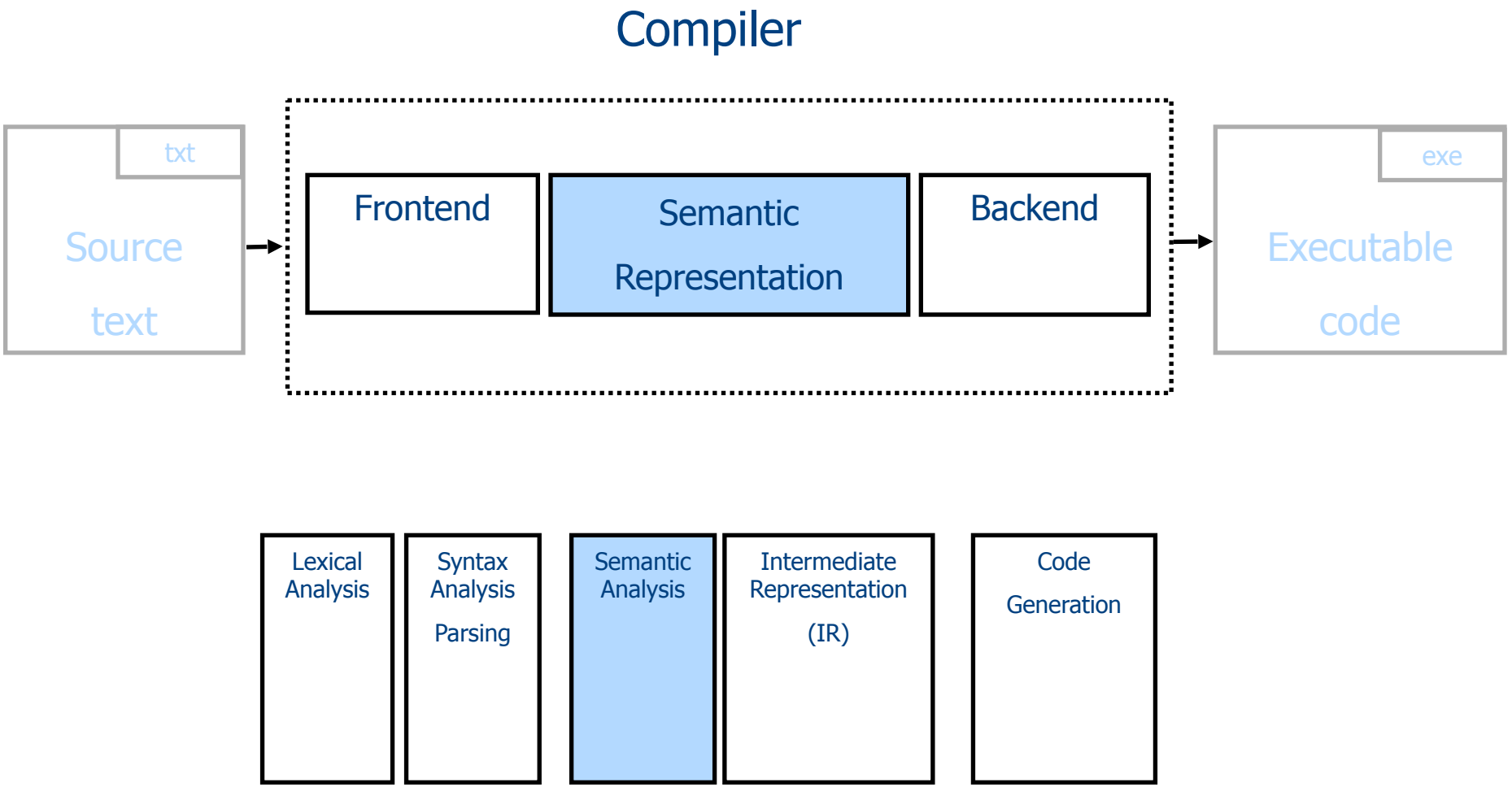
Semantic Analysis



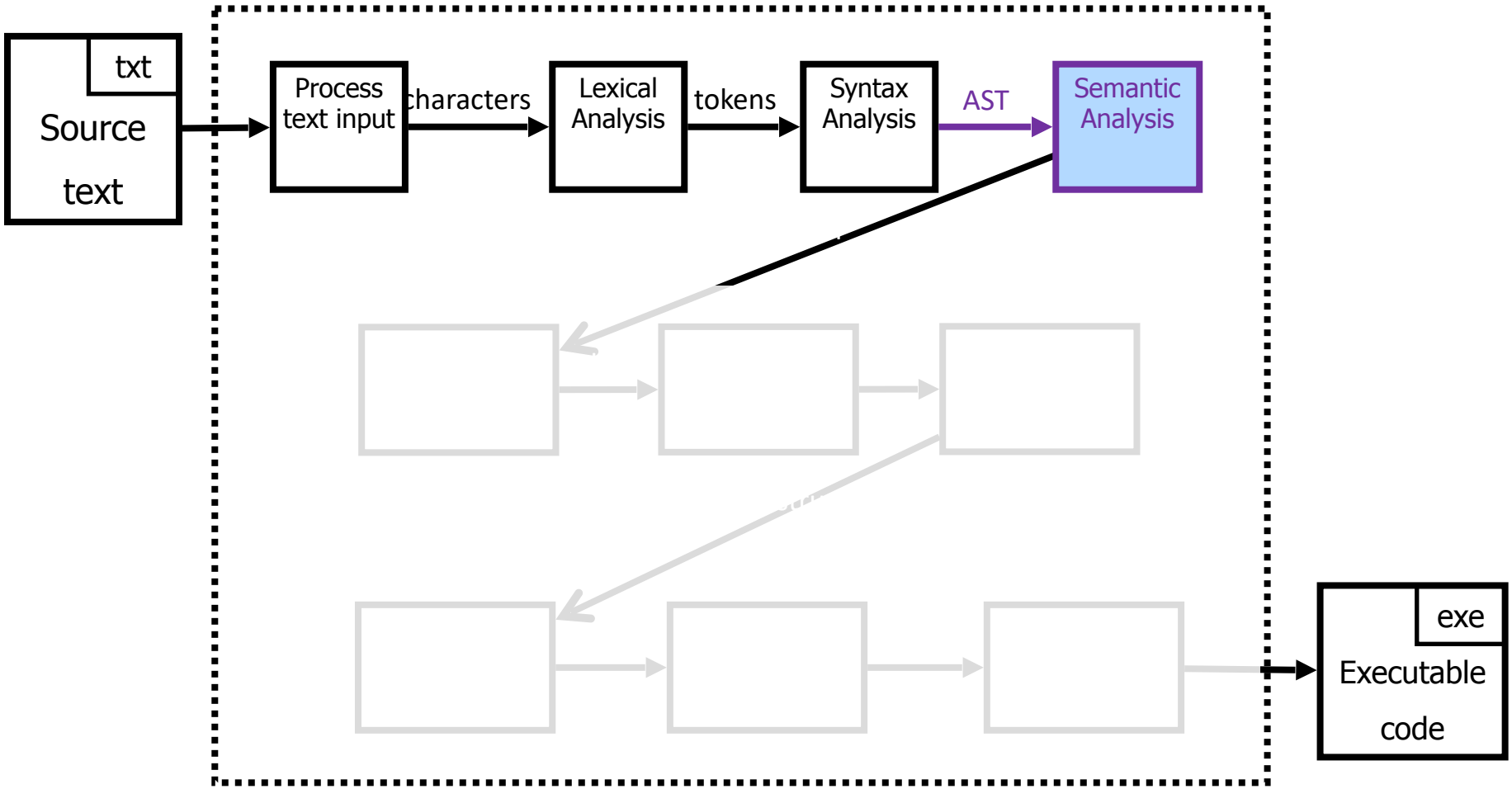
Slides' credit: Eran Yahav, Shachar Itzhaky, Mooly Sagiv

Grune
Chapter 4,11

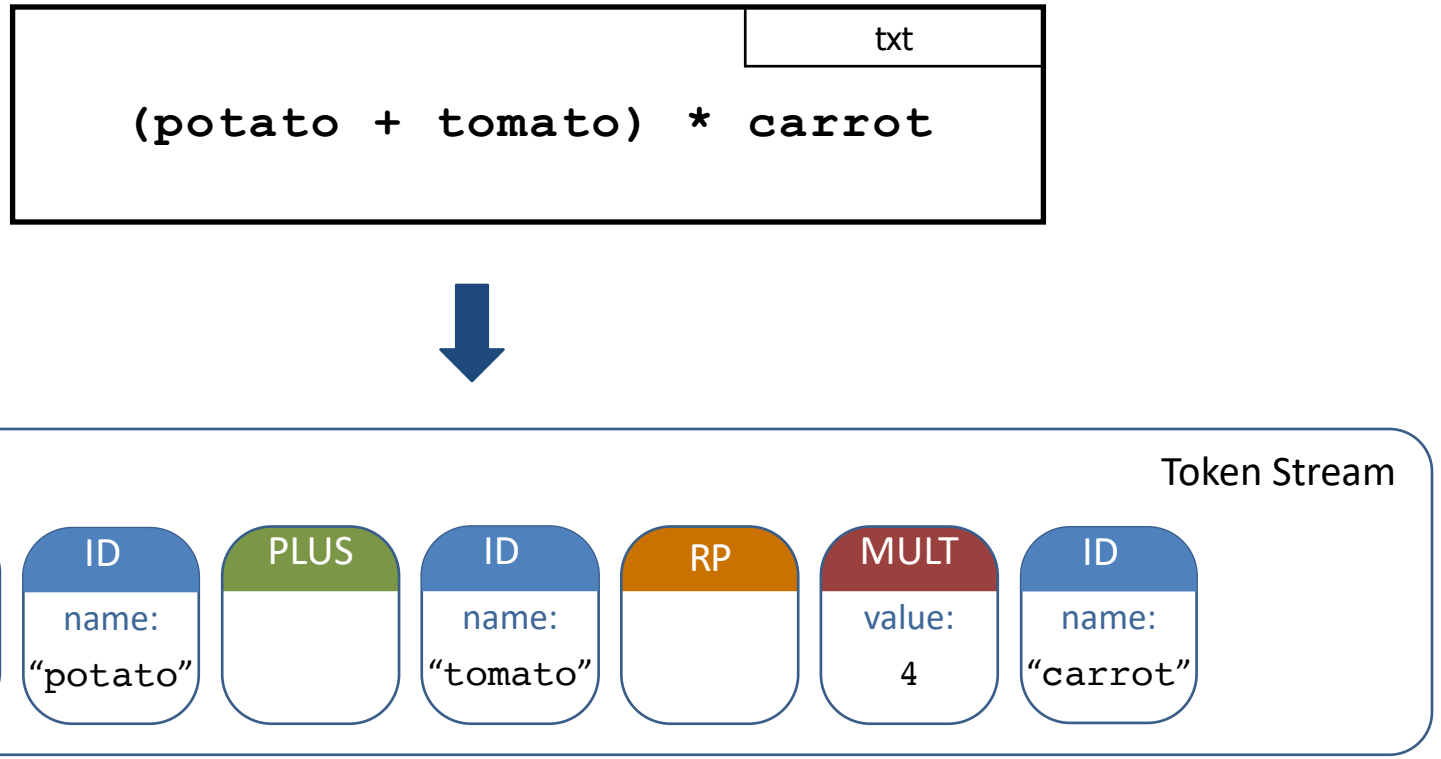
Conceptual Structure of a Compiler



You are here


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

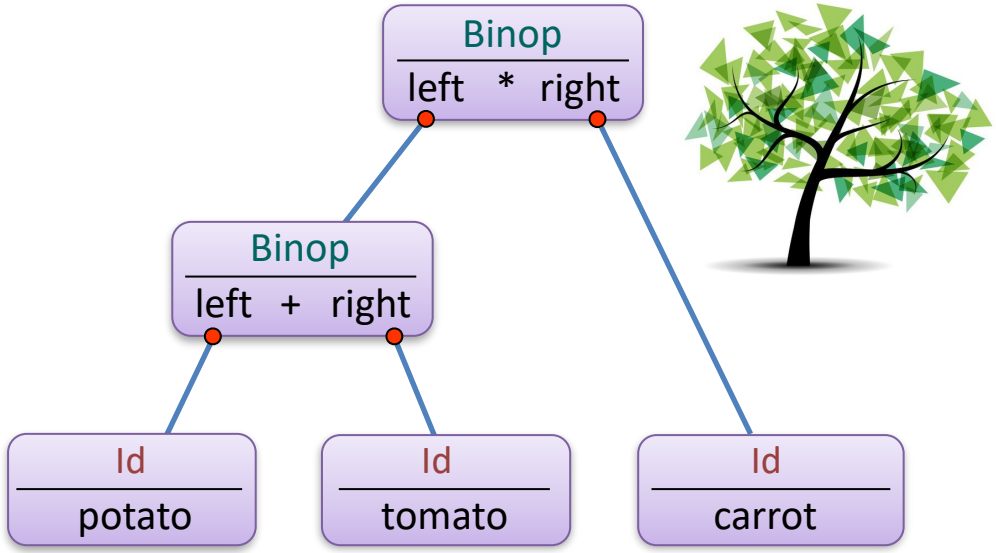
Lexing: from Characters to Tokens



Parsing: from Tokens to ~~Syntax Tree~~ AST

Syntax Tree

Abstract Syntax Tree




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

Semantic Analysis

- Processes the AST

- Goals:
  - Identification
  - Context checking

- A rich (user-friendly?) programming language

  ➡

  A challenging semantics analysis

Identification

```
int x = 0;

p() { print(x) }

q() { int x = 1; p() }
```

0 or 1?

Can the parser help?

Context Checking

✗ int x; xx = 0;

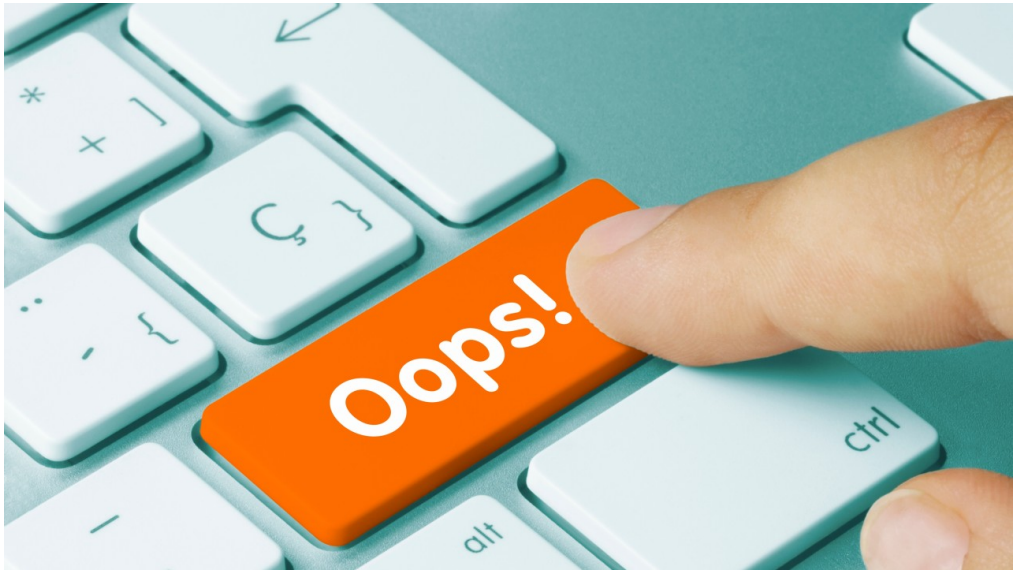
✗ int x, y, x;

✗ int x = “1”;

✗ int x, y; x = y+1;

✓ int x=1, y; y = x + 1;

Can the parser help?



Semantic Analysis

- Sometimes called “Contextual analysis”
  - As opposed to our syntax analysis — which was “context free”

- Properties that cannot be formulated via CFG
  - Declare before use
  - Type checking
  - Multiple definitions/declarations
  - …

Mostly related to the definition and use of identifiers

Semantic Analysis

- Sometimes called “Contextual analysis”
  - As opposed to our syntax analysis — which was “context free”
- Properties that cannot be formu
  - Declare before use
  - Type checking
  - Multiple definitions/declarations
  - …
- Properties that are clumsy to fo
  - “break” only appears inside a loo
    1. Convoluted grammar
    2. Process the AST
  - …

stmts → ε | stmt ; stmts

stmt → `if` expr `{` stmts `}` s_else

| `while` expr `{` stmts `}`

| `let` id `:` type = expr

| id = expr

s_else → ε | `else` `{` stmts `}`

stmt$^{\ell}$ → `if` expr `{` stmts$^{\ell}$ `}` s_else$^{\ell}$

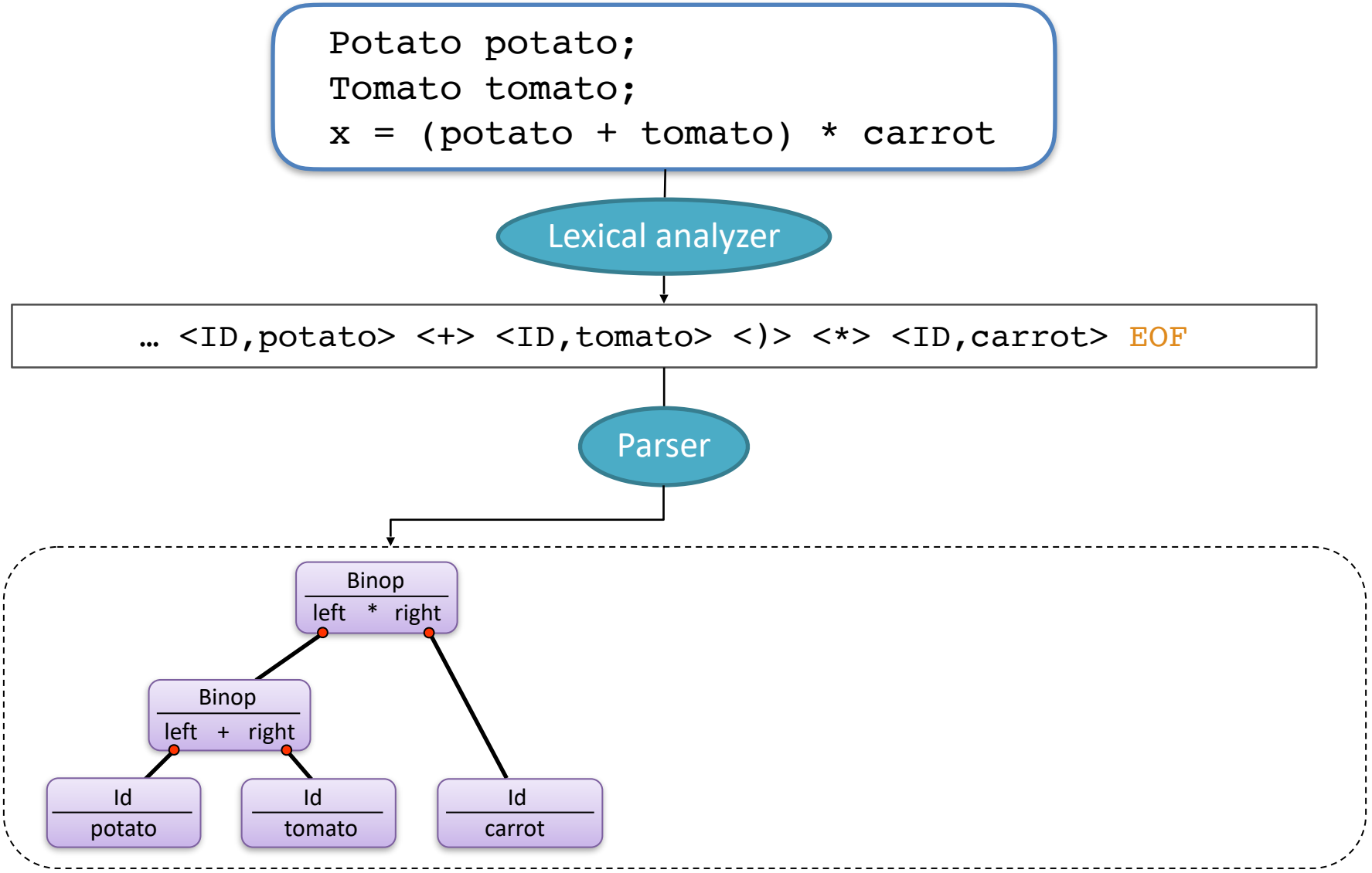
| `while` expr `{` stmts$^{\ell}$ `}`

| `let` id `:` type = expr

| id = expr | **`break`**

s_else$^{\ell}$ → ε | `else` `{` stmts$^{\ell}$ `}`

Semantic Analysis: What We Want



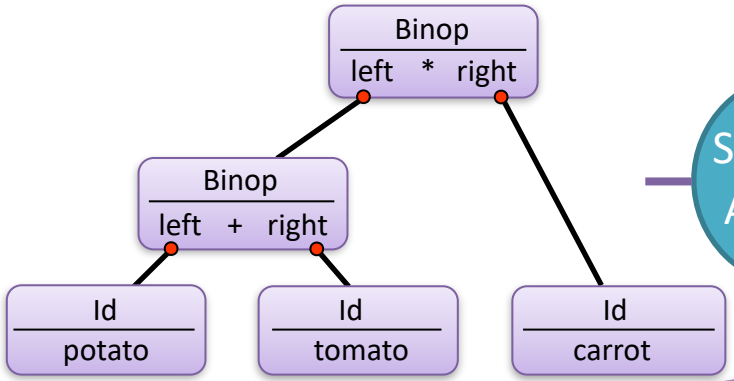
Semantic Analysis: What We Want

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

Lexical analyzer

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

Parser



Semantic Analysis

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

‘carrot’ is undefined

‘potato’ used before initialized

Cannot add ‘Potato’ and ‘Tomato’

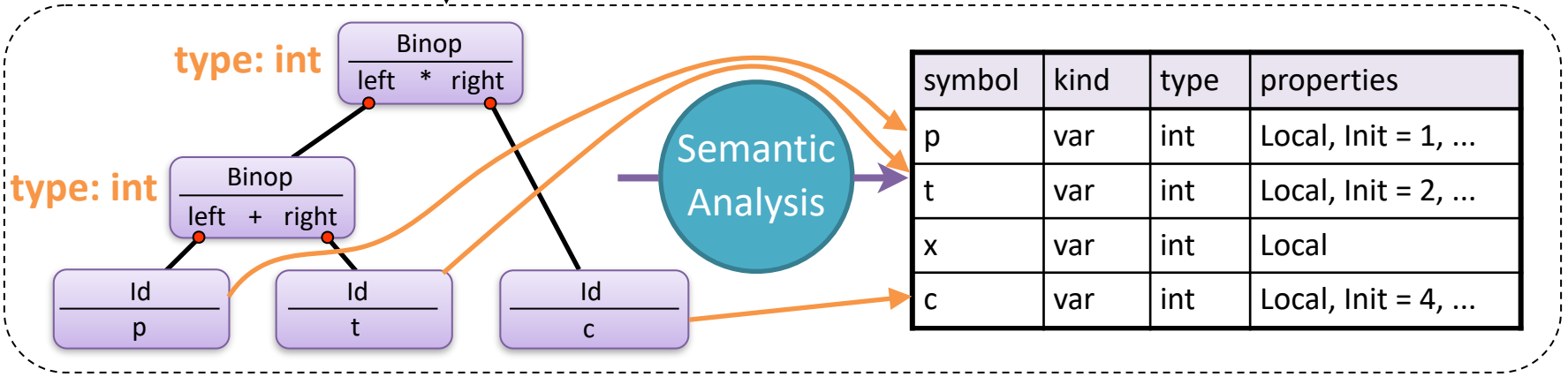
Semantic Analysis: What We Want

```
int p = 1, t = 2, c = 4;
int x;
x = (p + t) * c
```

Lexical analyzer

```
… <ID,p> <+> <ID,t> <)> <*> <ID,c> EOF
```

Parser

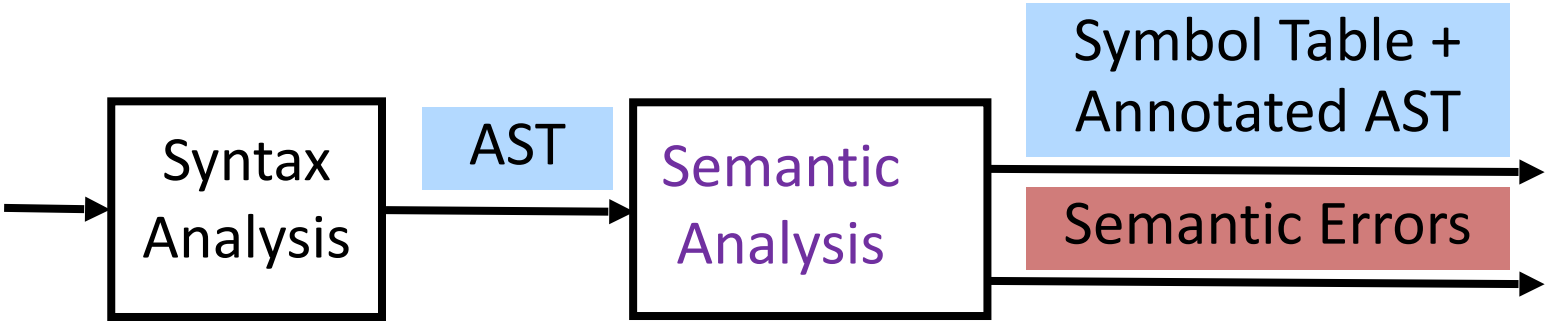


symbol	kind	type	properties
p	var	int	Local, Init = 1, ...
t	var	int	Local, Init = 2, ...
x	var	int	Local
c	var	int	Local, Init = 4, ...

Annotated AST ✔ Symbol Table

Semantic Analysis

- Processes the AST



- Goals:
  - Identification
  - Context checking

Semantic Analysis

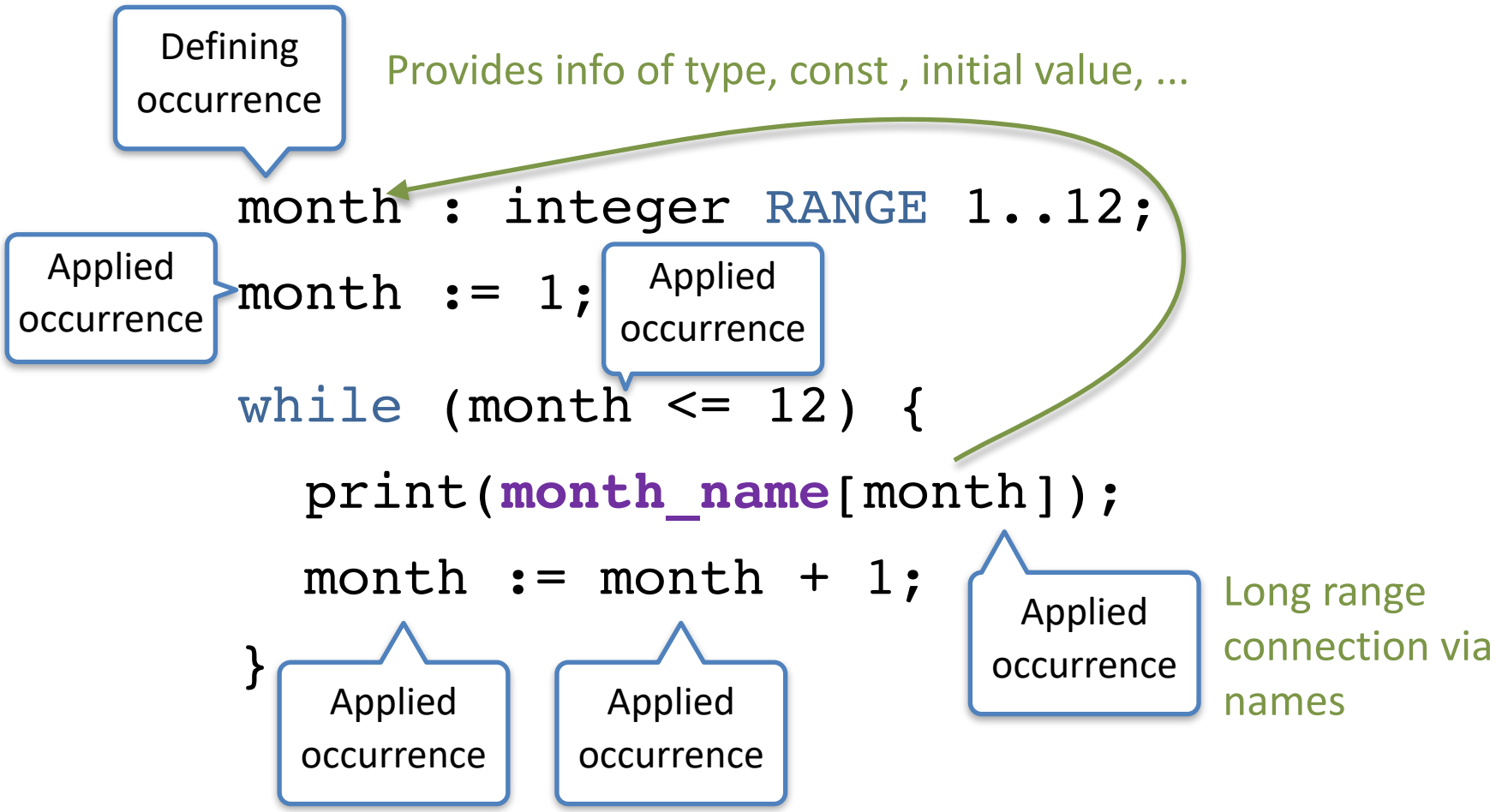
- Identification
  - Gather information about each named item in the program
    - *e.g.*, what is the declaration for each usage
    - *e.g.*, what is the type of the item
  - Information to be used in semantic checks & processing

- Context checking
  - Type checking
    - *e.g.*, the condition in an if-statement is a Boolean
  - Other “simple errors”: Unused vars, trivial conditions

Detecting "clumsy" CF properties, e.g., break

Conceptually, Identification before context checking. Practically, can be done together

Identification



Complications:

- Languages with forward declarations
- Languagaes with no declarations

Symbol Table

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

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

- A table containing information about identifiers in the program
- Single entry for each named item ← Even if there are no/fwd. declarations
- Identification connects all usages to the proper entry
  - Declaration: Add item + properties to table
  - Use: Look-up table.

Symbol Table: Functions Also

```
int month;
int foo(int x) { ... }
  ⋮
month := 1;
while (month <= 12) {
  foo(month + 1);
  month := month + 1;
}
```

name	pos	type	…
month	1	String	
foo	…	**int → int**	
…			

including functions

- A table containing information about identifiers in the program
- Single entry for each named item
- Identification connects all usages to the proper entry
  - Declaration: Add item + properties to table
  - Use: Look-up table.

Not so fast…

```
int month;
int foo(int x) { ... }
int foo(int x, int y) { }
  ⋮
month := 1;
while (month <= 12) {
  foo(month + 1);
  month := month + 1;
}
```

?

?

name	pos	type	…
month	1	String	
foo	…	int → int	
foo	…	int,int → int	
…			

- A table containing information about identifiers in the program
- Single entry for each **named item**
- **Overloading** (if supported by the PL) allows for multiple functions with the same name but different parameters
  - "Name" and "Lookup" must consider the parameters' types
  - Subtyping ("inheritance") adds complications

Not so fast…

```
int month;
int foo(int x) { ... }
bool foo(int x) { }

  ⋮

month := 1;
while (month <= 12) {
  foo(month + 1);
  month := month + 1;
}
```

?

?

name	pos	type	…
month	1	String	
foo	…	int → int	
foo	…	int → bool	
…			

- A table containing information about identifiers in the program
- Single entry for each named item
- **Overloading** (even if supported by the PL) does not allow for multiple functions with the same name and parameters, but different typed of return value

Not so fast…

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

int i = 15;

main() {
 int i = 20;
 i.i = 42;
 int t = i.i;
 printf("%d",t);
}
```

A struct field named "i"

A struct variable named "i"

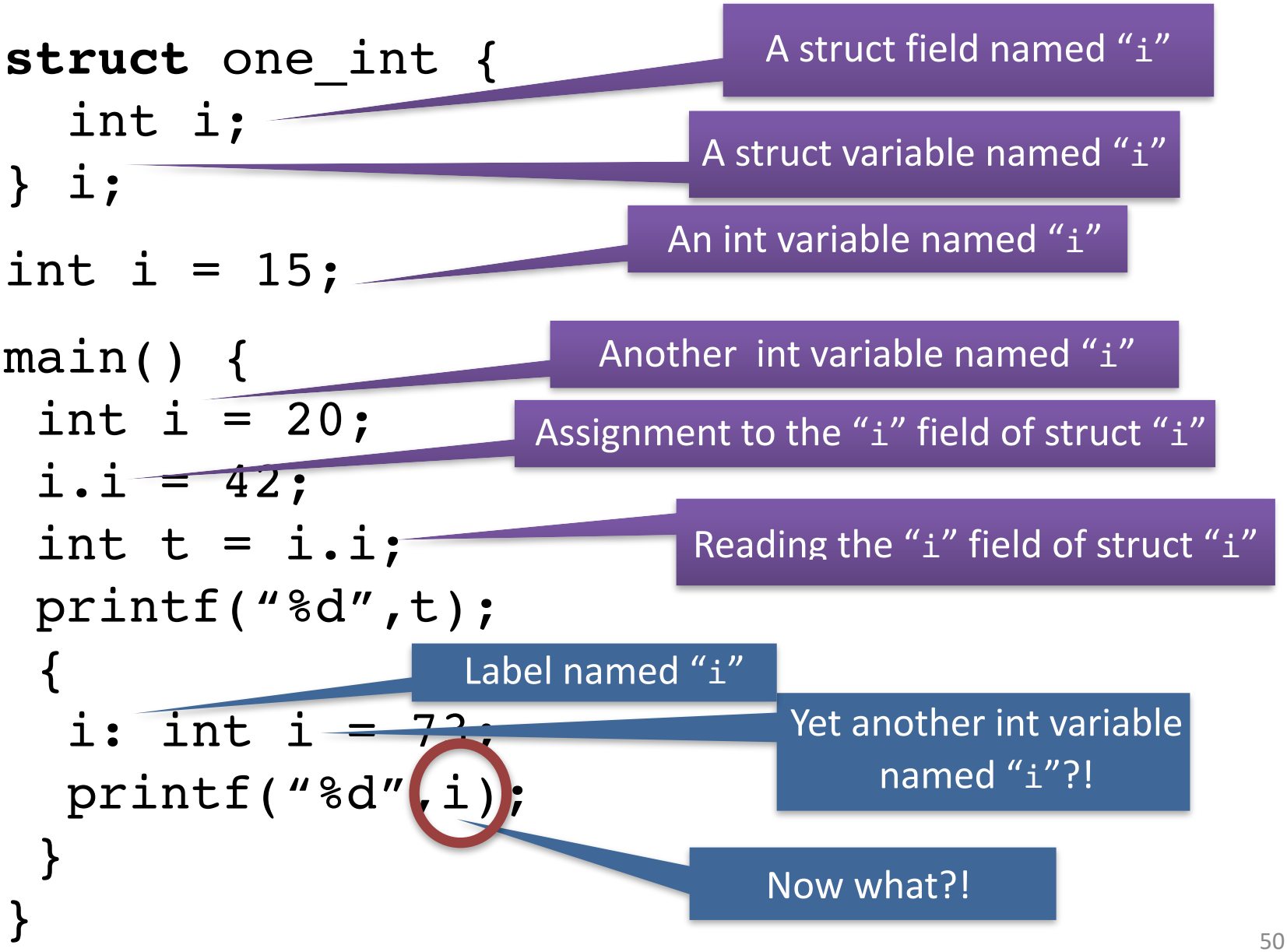
An int variable named "i"

Another int 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;
int i = 15;
main() {
int i = 20;
i.i = 42;
int t = i.i;
printf("%d",t);
{
i: int i = 73;
printf("%d",i);
}
}
A struct field named "i"
A struct variable named "i"
An int variable named "i"
Another int variable named "i"
Assignment to the "i" field of struct "i"
Reading the "i" field of struct "i"
Label named "i"
Yet another int variable named "i"?!
Now what?!

Multiple Namespaces

- Identifiers can live in different namespaces
  - Depends on the language design
  - Namespace of identifier determined by syntactic position

- E.g., C has 3 main namespaces
  - Labels
  - enum, struct & union names
  - Variables, functions, type identifiers

  + Namespace for each struct & union
    - Containing names and types of fields selectors

Not so fast…

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

int i = 15;

main() {
 int i = 20;
 i.i = 42;
 int t = i.i;
 printf("%d",t);
 {
  i: int i = 73;
   printf("%d",i);
 }
}
```

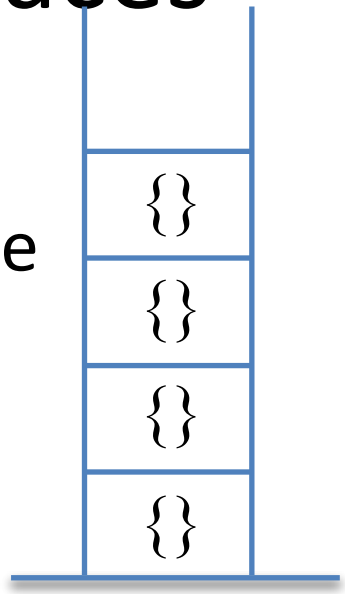
name	pos	type	…
i		label	
i	2	int	
i	3	one_int	
i	5	int	
i	7	int	
t	9	int	
i	13	int	
…	…		

Block-Structured Programming Languages

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

int i = 15;

main() {
 int i = 20;
 i.i = 42;
 int t = i.i;
 printf("%d",t);
 {
  i: int i = 73;
   printf("%d",i);
 }
}
```

name	pos	type	…
i		label	
i	2	int	
i	3	one_int	
i	5	int	
i	7	int	
t	9	int	
i	13	int	
…	…		

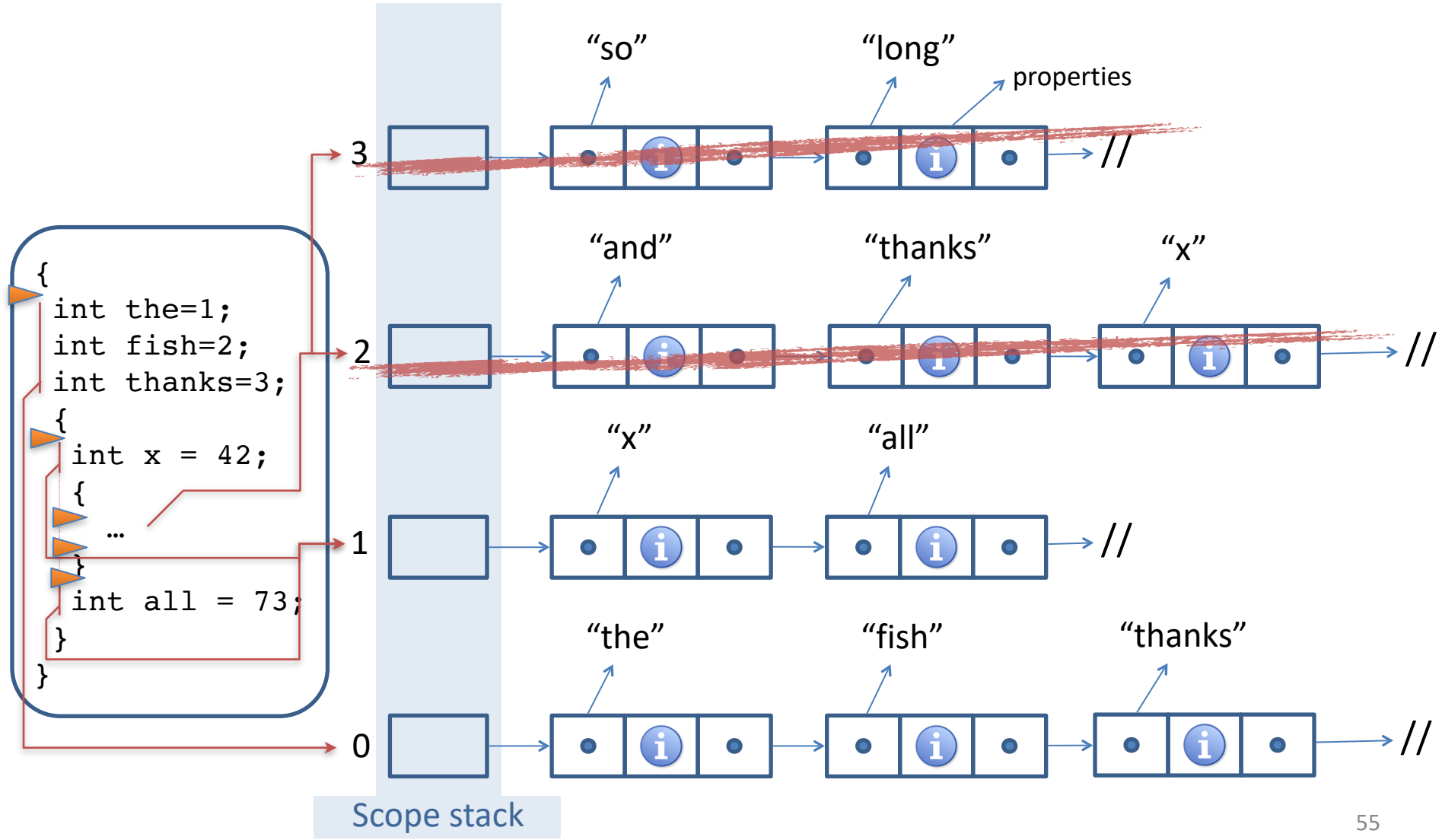
Scope-structured Namespaces



- Typically, for block-structured language we use 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

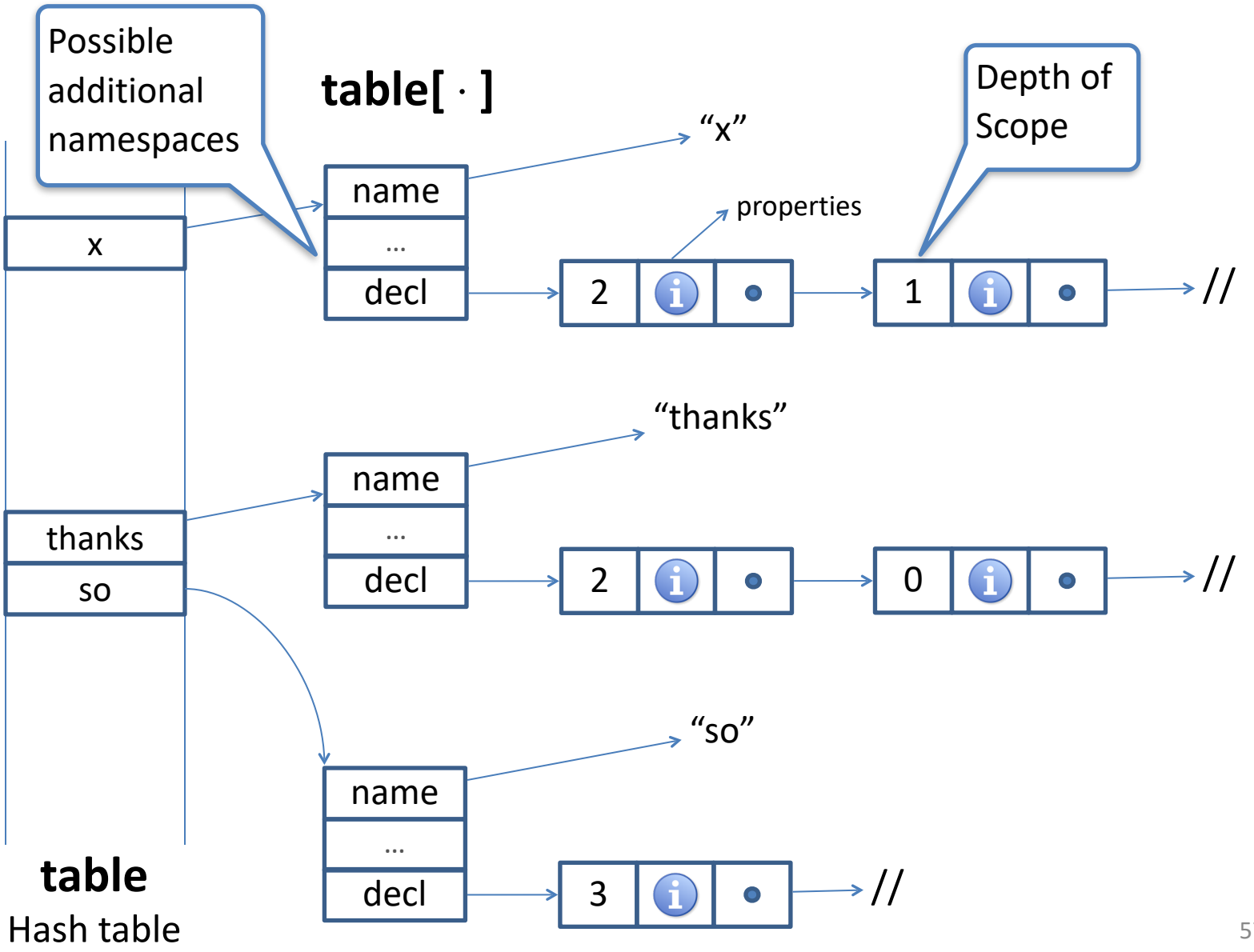


Scope and Symbol Table

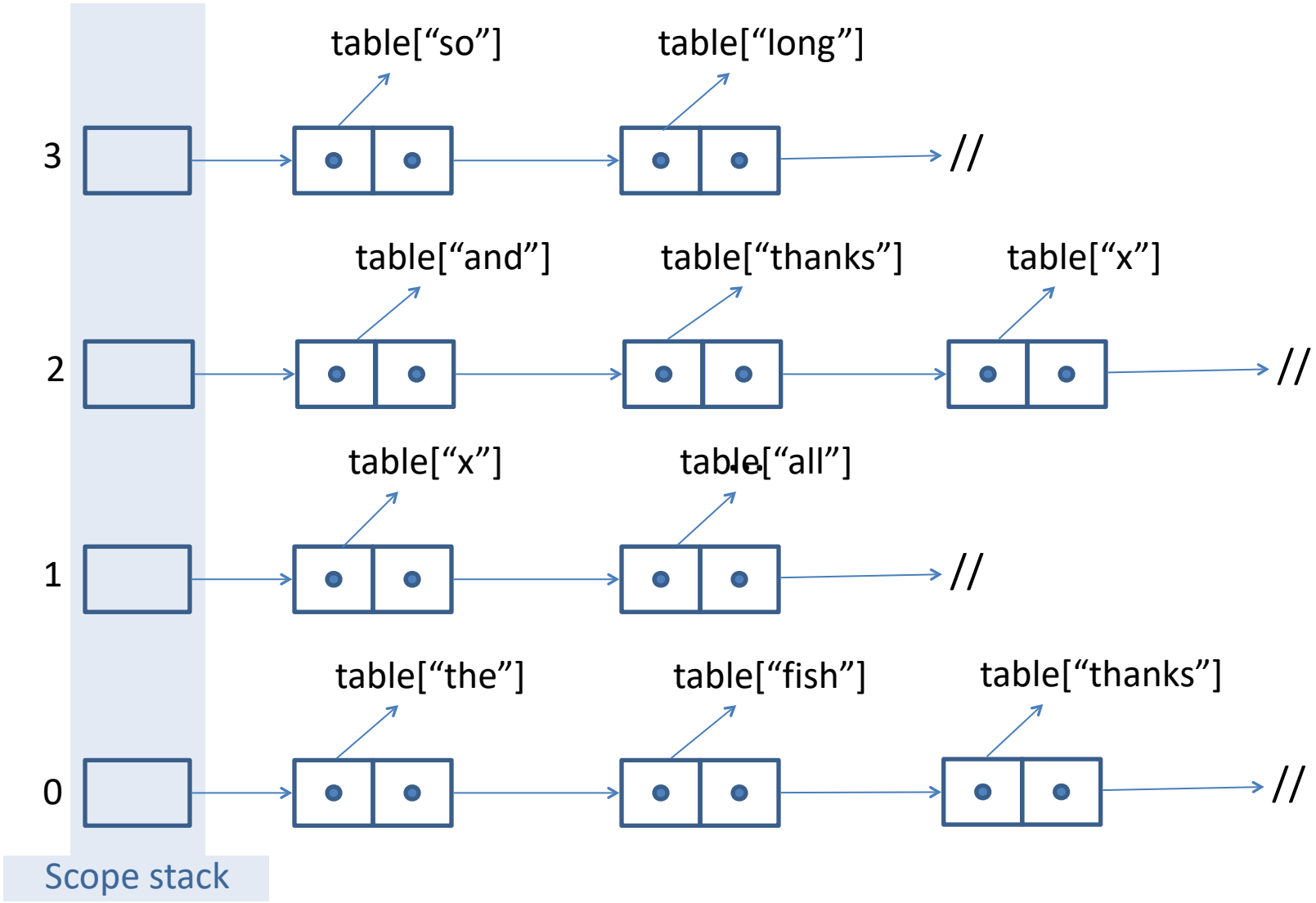
- Scope × Identifier ⟶ properties
  - Expensive lookup: when a reference to x is encountered, the entire table needs to be searched

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

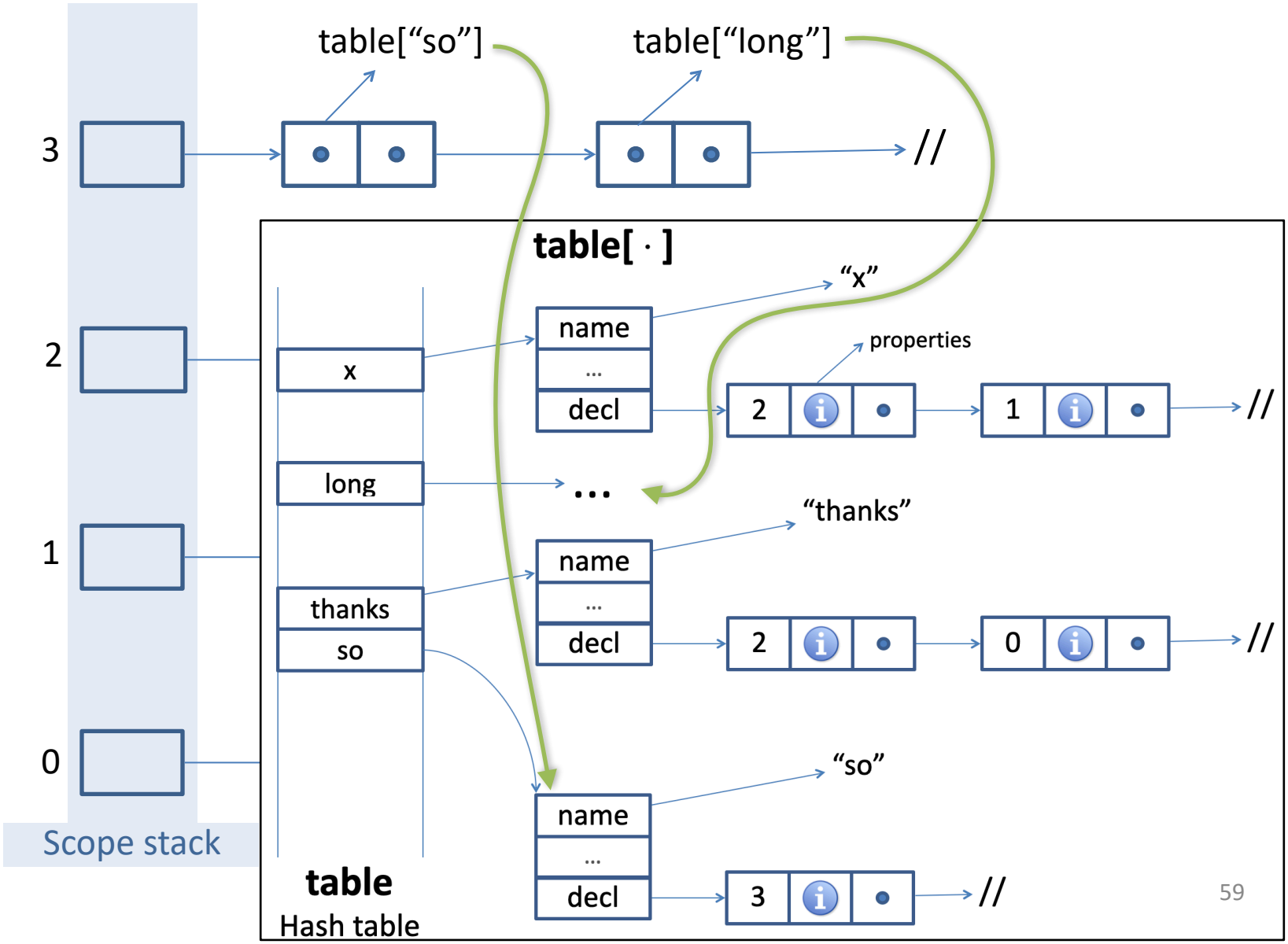
Hash Table-based Symbol Table



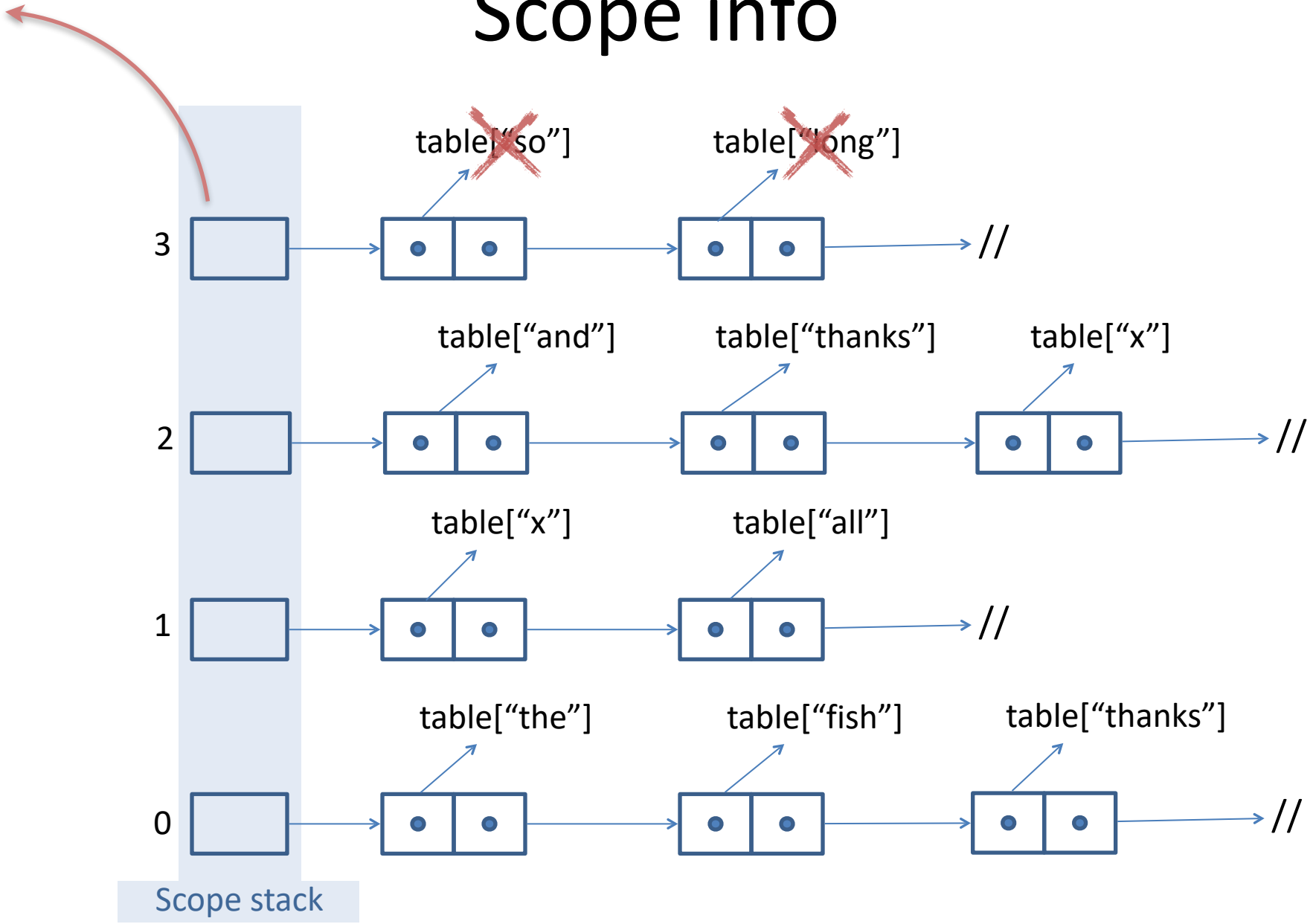
Scope info


table["so"]
table["long"]
3
//
table["and"]
table["thanks"]
table["x"]
2
//
table["x"]
table["all"]
1
//
table["the"]
table["fish"]
table["thanks"]
0
//
Scope stack

Scope info

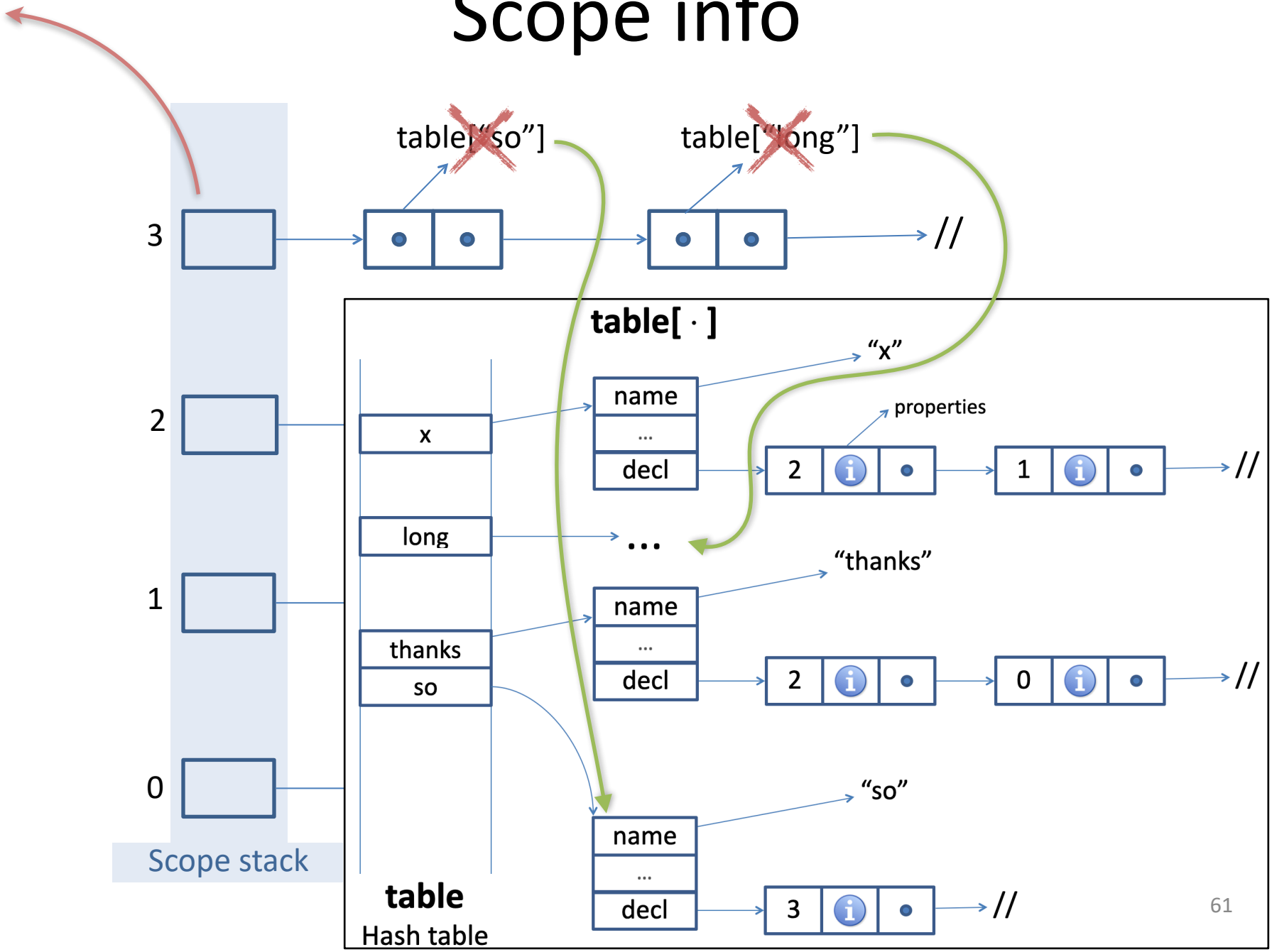

table["so"]
table["long"]
3
2
1
0
//
table[·]
"x"
name
...
decl
properties
x
2
1
//
long
...
"thanks"
name
...
decl
thanks
so
2
0
//
"so"
name
...
decl
3
//
Scope stack
table
Hash table

Scope info

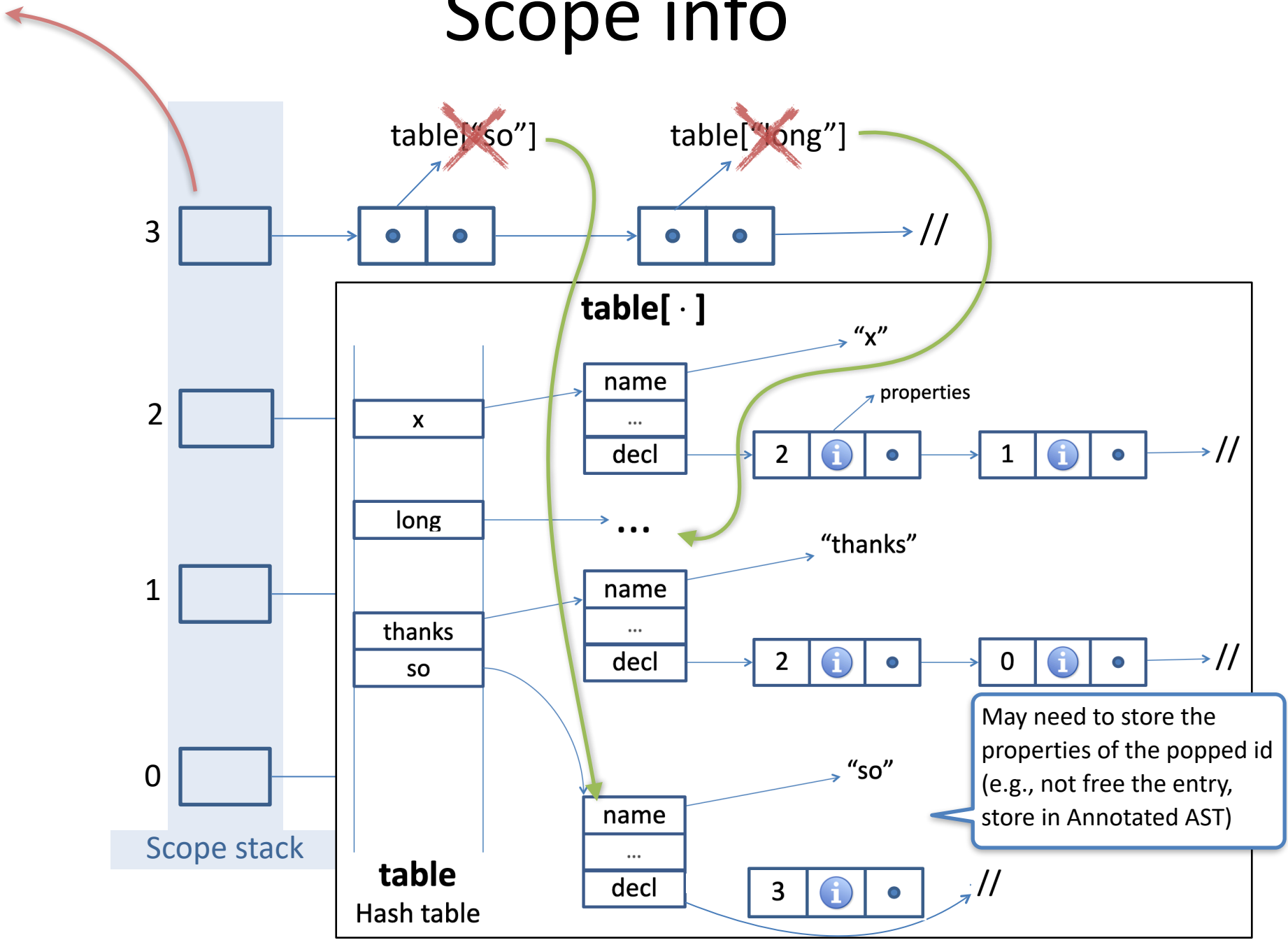


(now just pointers to the corresponding record in the symbol table)

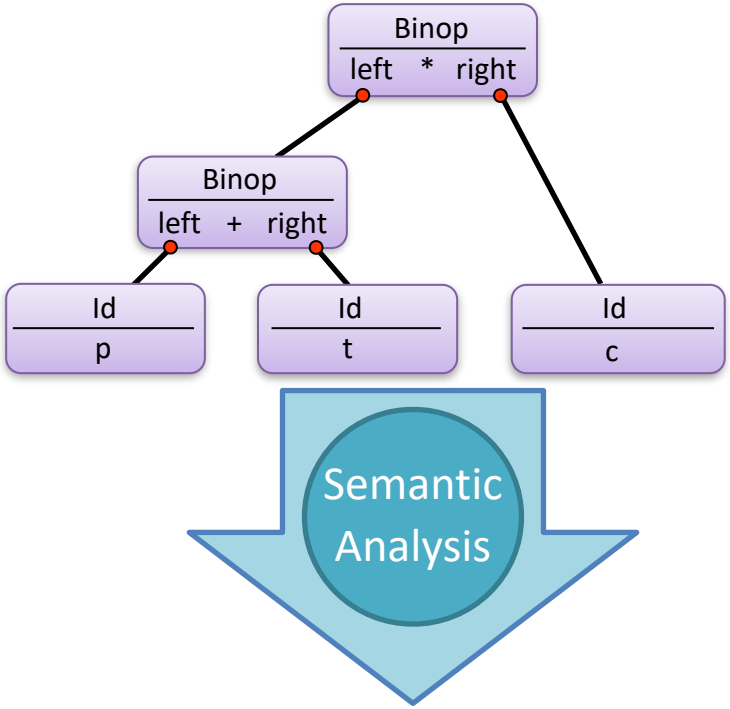
Scope info

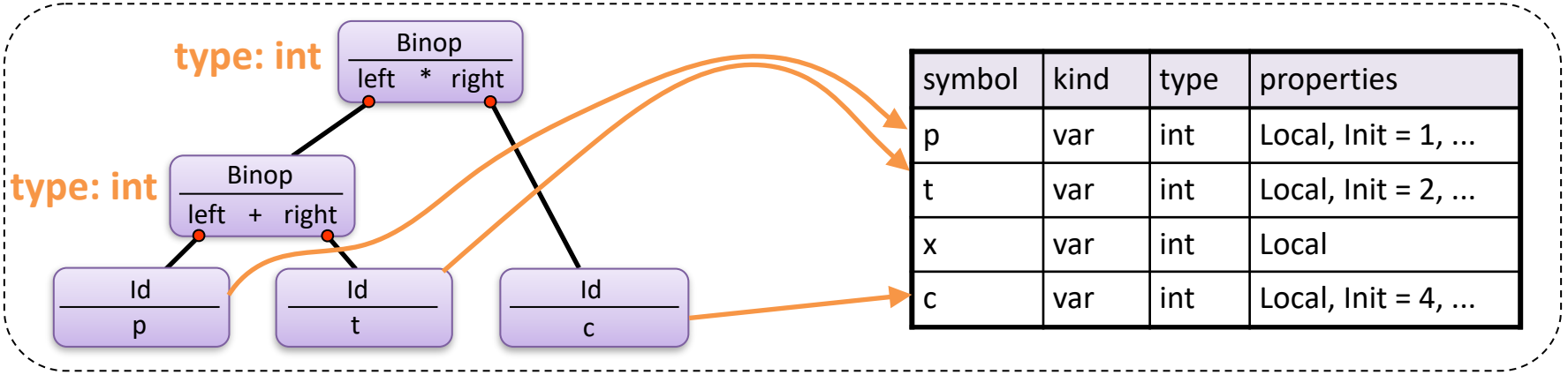

table["so"]
table["long"]
3
2
1
0
//
table[·]
"x"
name
...
decl
properties
x
2
1
//
long
...
"thanks"
thanks
so
name
...
decl
2
0
//
"so"
name
...
decl
3
//
Scope stack
table
Hash table

Scope info


table["so"]
table["long"]
3
2
1
0
//
table[·]
"x"
name
...
decl
properties
x
long
thanks
so
2
1
"thanks"
0
"so"
3
Scope stack
table
Hash table
May need to store the properties of the popped id (e.g., not free the entry, store in Annotated AST)

How does this magic happen?





symbol	kind	type	properties
p	var	int	Local, Init = 1, ...
t	var	int	Local, Init = 2, ...
x	var	int	Local
c	var	int	Local, Init = 4, ...

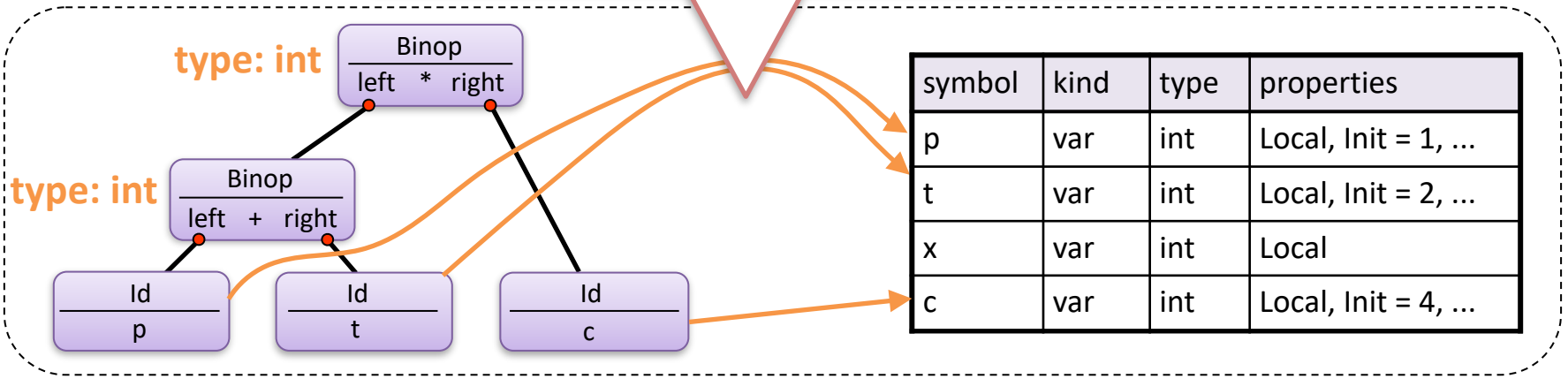
Annotated AST

Symbol Table

How does this magic happen?

BTW. Remember Lexing+Parsing?

- How did we know to always map tokens to the same identifier?
  - We didn't! Each id token is a new occurence
  - Now is the first time where we match them up



symbol	kind	type	properties
p	var	int	Local, Init = 1, ...
t	var	int	Local, Init = 2, ...
x	var	int	Local
c	var	int	Local, Init = 4, ...

Annotated AST

Symbol Table

How does this magic happen?

- **Ad-hoc:** **Traverse** the AST top-down and maintain the symbol table

- **Syntax Directed Translation:**
  - Attach semantics attributes to grammar symbols
  - Define how to evaluate them
  - An attribute grammar "engine" does the evaluation

  ➡ Similarly to the way we built the AST bottom-up

Recall: Building the AST

```
class Node {}

class BinOp extends Node {
        Node lhs;
        Node rhs;
        char op;
}

class Num extends Node {
        int value;
}
```

$E \rightarrow e1{:}E + e2{:}E$ {

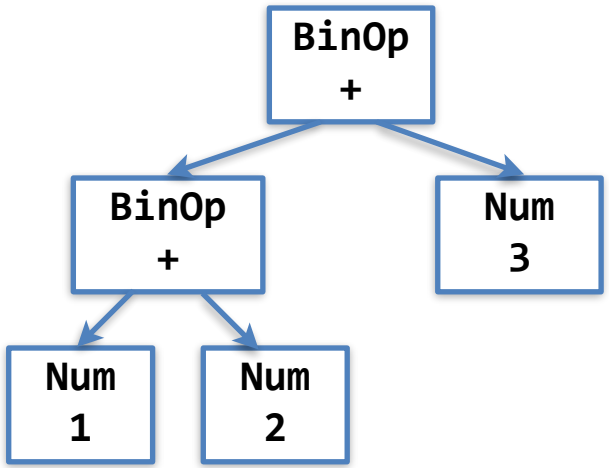
```
  RESULT = new BinOp();
  RESULT.lhs = r1;
  RESULT.rhs = e2;
  RESULT.op = '+'
```

}

$E \rightarrow n{:}NUM$ {

```
  RESULT = new Num();
  RESULT.value = n.value;
```

}



(1 + 2) + 3

How does this magic happen?

- **Ad-hoc:** **Traverse** the AST top-down and maintain the symbol table

- **Syntax Directed Translation:**
  - Attach semantics attributes to grammar symbols
  - Define how to evaluate them
  - An attribute grammar "engine" does the evaluation

  ➡ Similarly to the way we built the AST bottom-up

How does this magic happen?

- **Ad-hoc:** **Traverse** the AST top-down and maintain the symbol table
  - Block entry: enter a new scope
  - Block exit: exit top scope
  - Identifier declaration: create an entry in the top scope
    - Add properties from the declaration (e.g., types)
  - Identifier use: lookup identifier
    - **Annotate** AST identifier node with pointer to entry

▶ One Option: Using the Visitor Pattern

Semantic Analysis using Visitors

```
class Node {}

class BinOp extends Node {
  Node lhs;
  Node rhs;
  char op;
}

class Num extends Node {
  int value;
}
```

```
class ASTVisitor {
  void visit(BinOp node) {
    //do specific things to BinOp
  }
  void visit(Num node) {
    //do specific things to Num
  }
  ...
}

 int main() {
   Parser p = new Parser();
   Node n = p.parse();
   ASTVisitor v =
         new ASTVisitor();
   n.accept(v);
 }
```

Semantic Analysis using Visitors

```
class Node {
  void accept(ASTVisitor v);
}

class BinOp extends Node {
  void accept(ASTVisitor v) {
    v.visit(this);
  }
  Node lhs;
  Node rhs;
  char op;
}

class Num extends Node {
  void accept(ASTVisitor v) {
   v.visit(this);
  }
  int value;
}
```

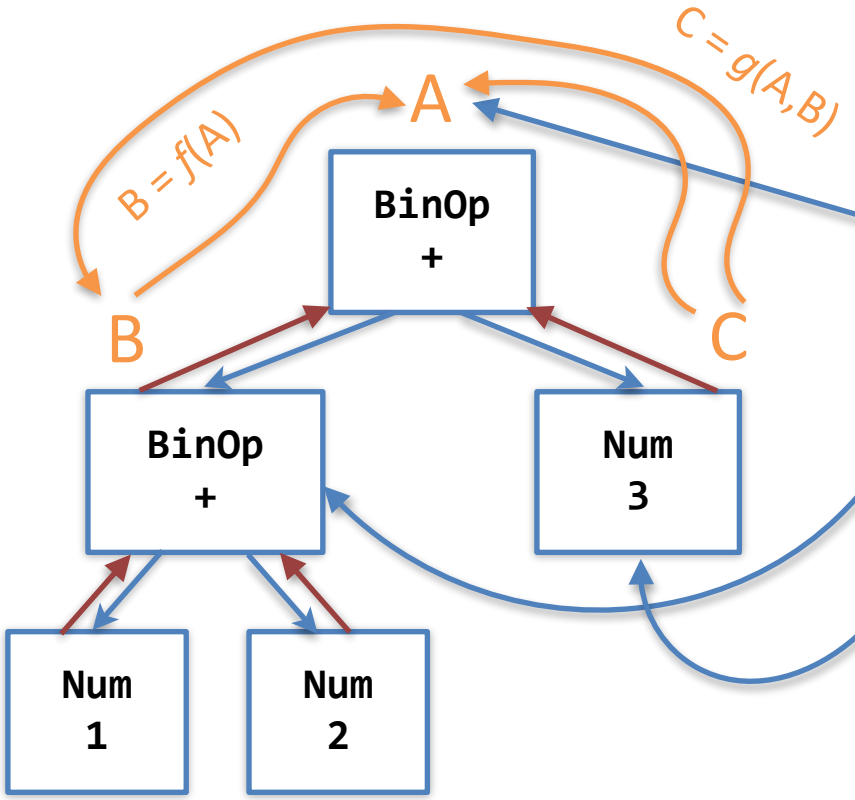
```
class ASTVisitor {
  void visit(BinOp node) {
    //do specific things to BinOp
  }
  void visit(Num node) {
    //do specific things to Num
  }
  ...
}

int main() {
  Parser p = new Parser();
  Node n = p.parse();
  ASTVisitor v =
        new ASTVisitor();
  n.accept(v);
}
```

Semantic Analysis using Visitors

```
class Node {
  void accept(ASTVisitor v);
}

class BinOp extends Node {
  void accept(ASTVisitor v) {
    v.visit(this);
  }
  Node lhs;
  Node rhs;
  char op;
}

class Num extends Node {
  void accept(ASTVisitor v)
  {
  void accept(ASTVisitor v) {
    v.visit(this);
  }
  int value;
}
```

```
class ASTVisitor {
  void visit(BinOp node) {
    //do specific things to BinOp
    lhs.accept(this);
    //do specific things to BinOp
    rhs.accept(this);
    //do things using node, lhs, rhs
  }
  ...
}

int main() {
  Parser p = new Parser();
  Node n = p.parse();
  ASTVisitor v =
        new ASTVisitor();
  n.accept(v);
}
```

Semantic Analysis using Visitors




```
class ASTVisitor {
  void visit(BinOp node) {
    //do specific things to BinOp
    lhs.accept(this);
    //do specific things to BinOp
    rhs.accept(this);
    //do things using node, lhs, rhs
  }
  ...
}

int main() {
  Parser p = new Parser();
  Node n = p.parse();
  ASTVisitor v =
        new ASTVisitor();
  n.accept(v);
}
```

A, B, C attributes of every (expression) node

Semantic Analysis using Visitors

```
class Node {
  void accept(ASTVisitor v);
}

class Block extends Node {
  void accept(ASTVisitor v) {
    v.visit(this);
  }
  StmtList sl;
}
```

```
class ASTVisitor {
  void visit(Block node) {
    //do specific things to Block
    symTab.openScope();
    sl.accept(this);
    //do things using node, sl
    symTab.closeScope();
  }
  ...
}
int main() {
  Parser p = new Parser();
  Node n = p.parse();
  ASTVisitor v =
        new ASTVisitor();
  n.accept(v);
}
```

Semantic Analysis

- Identification
  - Gather information about each named item in the program
    - *e.g.*, what is the declaration for each usage
  - Information to be used in semantic chacks & processing

- Context checking
  - Things that are difficult to do during parsing
  - Type checking
    - *e.g.*, the condition in an if-statement is a Boolean
  - Other “simple errors”: Unused vars, trivial conditions

Handling Break: A Possible Abstract Syntax

	// Abstract Syntax		**// Concrete Syntax**
Stmt ->	Stmt Stmt	(SeqStmt)	// Stmt **;** Stmt
|	Exp Stmt Stmt	(IfStmt)	// **if (**Exp**)** Stmt **else** Stmt
|	Exp Stmt	(WhileStmt)	// **while (**Exp**)** Stmt
|	break	(BreakSt)	// **break**

Handling Break: A Possible AST

```
class Node {};
class Exp extends Node { };

class SeqStmt extends Stmt {
    public Stmt fstSt, secondSt;
      ...
}

class IfStmt extends Stmt {
    public Exp exp;
    public Stmt thenSt, elseSt;
     ...
}
```

```
class Stmt extends Node {} ;

class WhileStmt extends Stmt {
    public Exp exp;
    public Stmt body;
    ...
}

class BreakSt extends Stmt {};
```

	// Abstract Syntax		// Concrete Syntax
Stmt ->	Stmt Stmt	(SeqStmt)	// Stmt **;** Stmt
|	Exp Stmt Stmt	(IfStmt)	// **if (**Exp**)** Stmt **else** Stmt
|	Exp Stmt	(WhileStmt)	// **while (**Exp**)** Stmt
|	break	(BreakSt)	// **break**

Handling Break: A Possible AST

```
void checkBreak(Stmt st)
{
      if (st instanceof SeqSt) {
         SeqSt seqst = (SeqSt) st;
         checkBreak(seqst.fstSt);
         checkBreak(seqst.secondSt);
      }
      else if (st instanceof  IfSt) {
         IfSt ifst = (IfSt) st;
         checkBreak(ifst.thenSt);
         checkBreak(ifst elseSt);
      }
      else if (st instanceof  WhileSt) ; // skip
      else if (st instanceof  BreakeSt) {
         System.error.println(
              "Break must be enclosed within a loop"); }
}
```

Can be done with Visitors (Try!)

Context Checking

- ✓ Things that are difficult to do during parsing
- Type checking
  - *e.g.*, the condition in an if-statement is a Boolean
- Other “simple errors”: Unused vars, trivial conditions

Semantic Checks

- Scope rules
  - Use symbol table (while building it) to check:
    - No multiple definition of the 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, …
  - Allowed operations

- Why do we care?
  - Safety
    - Guarantee that certain errors cannot occur at runtime
  - Abstraction
    - Hide implementation details
  - Documentation
  - Optimization

Type Systems

- A language's type system specifies which operations are valid for which types

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

string

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

ERROR

Why do We Need Type Systems?

- Consider assembly code
  - add $r1, $r2, $r3
- What are the types of $r1, $r2, $r3?

Types and Operations

- Certain operations are legal for values of each type
  - It does not make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types

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

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

- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values because nothing else will!
    - Certainly not an (untyped) assembly language

Type Systems

- 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

Typing Checking via Typing Rules

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

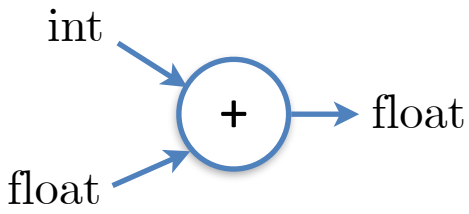
- Examples

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

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

Typing Checking via Typing 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

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

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

$\diamond \in \{ +, -, /, *, \% \}$

Binary Arithmetic Operators

$$\frac{E_1 : \text{int} \qquad E_2 : \text{int}}{E_1 \ominus E_2 : \text{boolean}}$$

$\ominus \in \{ <=, <, >, >= \}$

Comparison Operators

T = any type

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

$\ominus \in \{ ==, != \}$

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

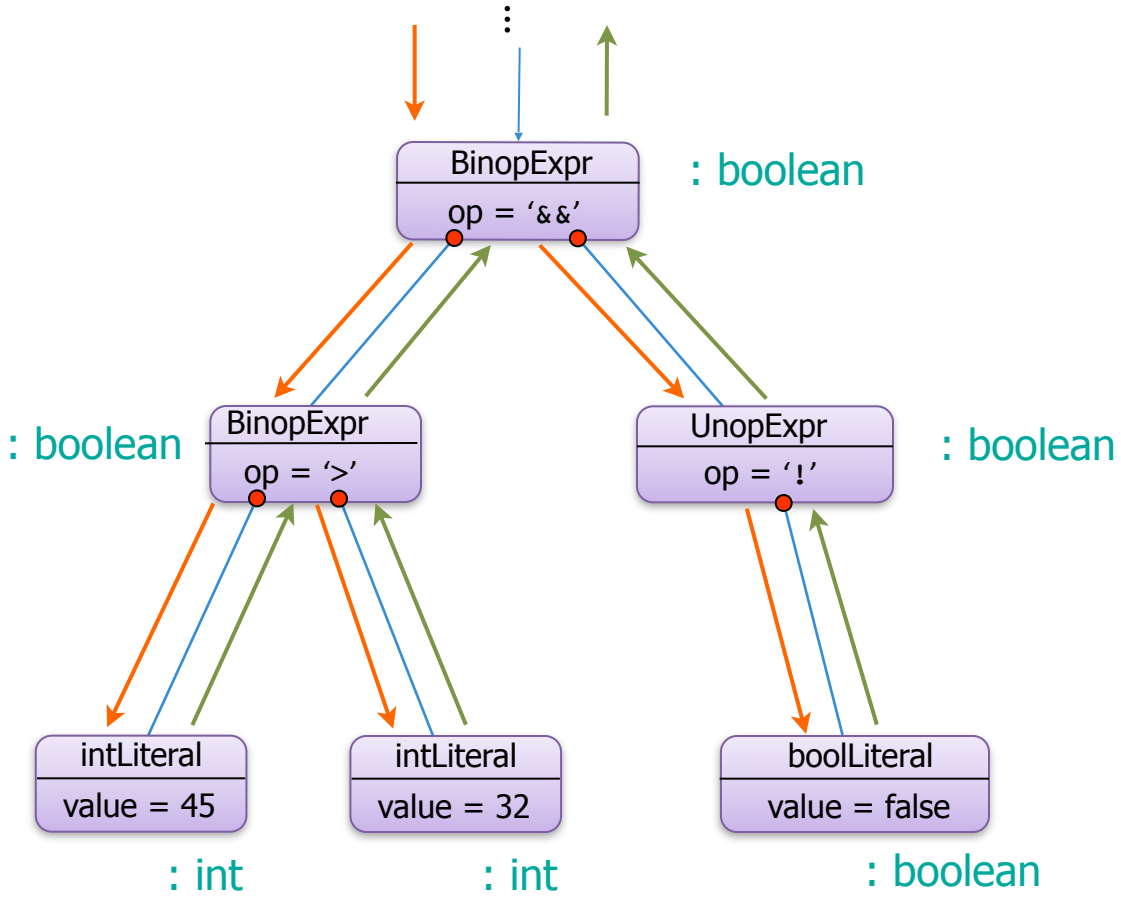
$$\frac{E_1 : T[] \qquad E_2 : \text{int}}{E_1[E_2] : T} \qquad \frac{E_1 : \text{int}}{\text{new } T[E_1] : T[]}$$

How is Type Checking done?

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

- Alternative: identify + type-check during parsing
  - (Slightly?) more complicated
  - But naturally also more efficient
    - Narrow compilers
  - (We'll see that later)

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
```

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

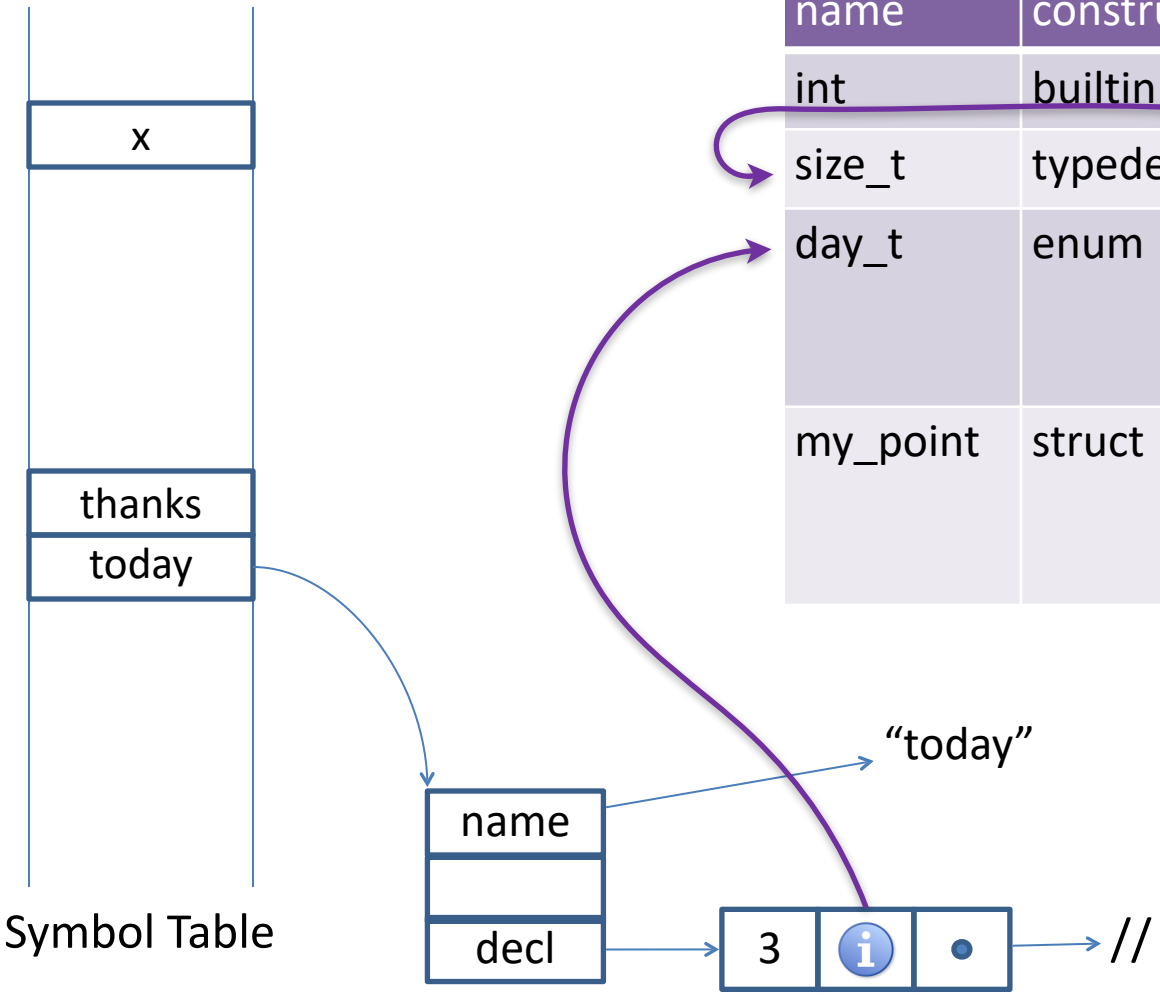
- All types in a compilation unit are collected in a type table

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



Recording Types of Functions and Methods

```
int foo(float x, int y) {
⋮
}
```

```
class A {
  int foo(float x, int y) {
  ⋮
  }
};
```

Symbol Table

name	pos	type	…
⋮			
foo	42	(float,int)→int	
⋮			
⋮			

Type Table

name	constructor	Structure	…
int	builtin	–	
⋮			
A	class	foo: (float,int)→int	
⋮			

Recording Types of Functions and Methods

```
int foo(float x, int y) {
⋮
}
```

```
class A {
  int foo(float x, int y) {
  ⋮
  }
};
```

Symbol Table

name	pos	type	…
⋮			
foo	42	(float,int)→int	
⋮			
⋮			

Type Table

name	constructor	Structure	…
int	builtin	–	
⋮			
A	class		
⋮			

name	type	…
⋮		
foo	(float,int)→int	
⋮		

Alt., have a type table for A

Forward Type References


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

(sometimes symbol)

- 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 (“resolved”)
- At the end of scope, check that all forward refs have been resolved

Circular Dependencies Between Types

```
struct A;
⋮

struct B {
    A element_A;
}

struct A {
    B element_B;
}
```

Object of type A:

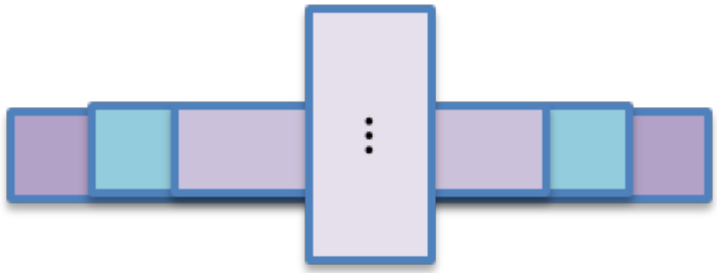
B element_B :

A element_A :

B element_B :

A element_A :

...



Forward references can get circular!
If we allow forward references, we must explicitly disallow cyclic usage!

Circular Dependencies Between Types

```
TYPE x = y
TYPE y = x
```

Trivial example of circular dependency

- (Probably) illegal -- Should be disallowed
- C & co. do not allow forward declaration of types
  - Allow forward reference to union/structure tags, which is sufficient in practice

Nominal vs. Structutal Type Systems

Type Checks

From symbol table: type τ_1

From typing rules: type τ_2

```
x = E;
```

does τ_1 *match* τ_2?

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

Type Conversions

- 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++

Type Conversions

- 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
```

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

Cast user-defined (explicit) type conversion

Coercion Compiler generated automatic (implicit) type conversion

Corections

- If we expect a value of type T_1 at some point in the program, and find a value of type T_2, what to do?
- C: type conversion depends on context

```
int   x;
float y;
x = y;   // y→int
y + x;   // x→float
```

- Multi-step coercions: `int` → `float` → `complex`
- A difficult problem if the languages has arbitrary types, operators, coercion and

Strongly Typed *vs.* Weakly Typed

- **How are coercions handled?**
- **Strongly typed**
  - C, C++, Java
  - Limited "sane" coercions (int → float)
  - Explicit "dangerous" type conversions (int x = ... ; int* p = (int*) x;)
    - Possibly checked around runtime
  - Less flexible, more type safe
- **Weakly typed**
  - JavaScript, Perl, PHP
  - Possibly "insane" conversions ("42x" → 42)
  - More type safe, less flexible

(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

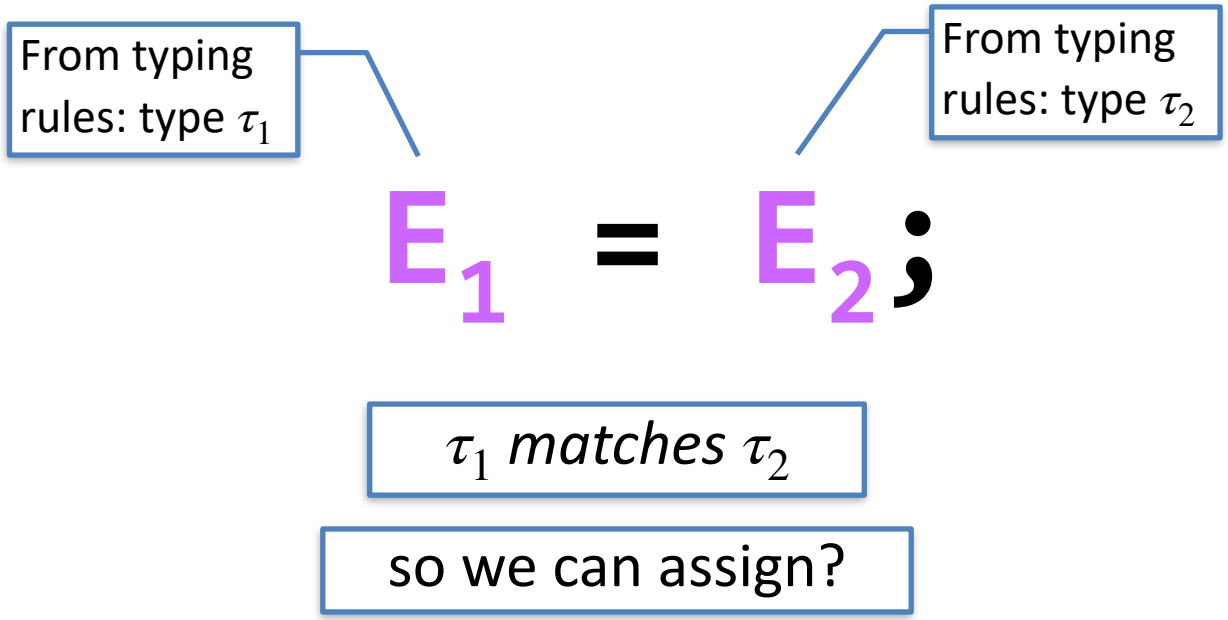
Java

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

Kind Checking

- What is the "kind" of the identifier
  - Variable
  - Constant
  - Function
  - Method
  - Module
  - ...
- Mostly trivial, except
  - Left-values (*l*-values): locations/addresses
  - Right-values (*r*-values): (normal/actual) values

l-values and r-values



l-values and r-values

- Assignment dst := src is compiled into:
  - Compute the address of dst
  - Compute the value of src
  - Store the value of src into the address of dst

l-values and r-values

```
dst := src
```

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

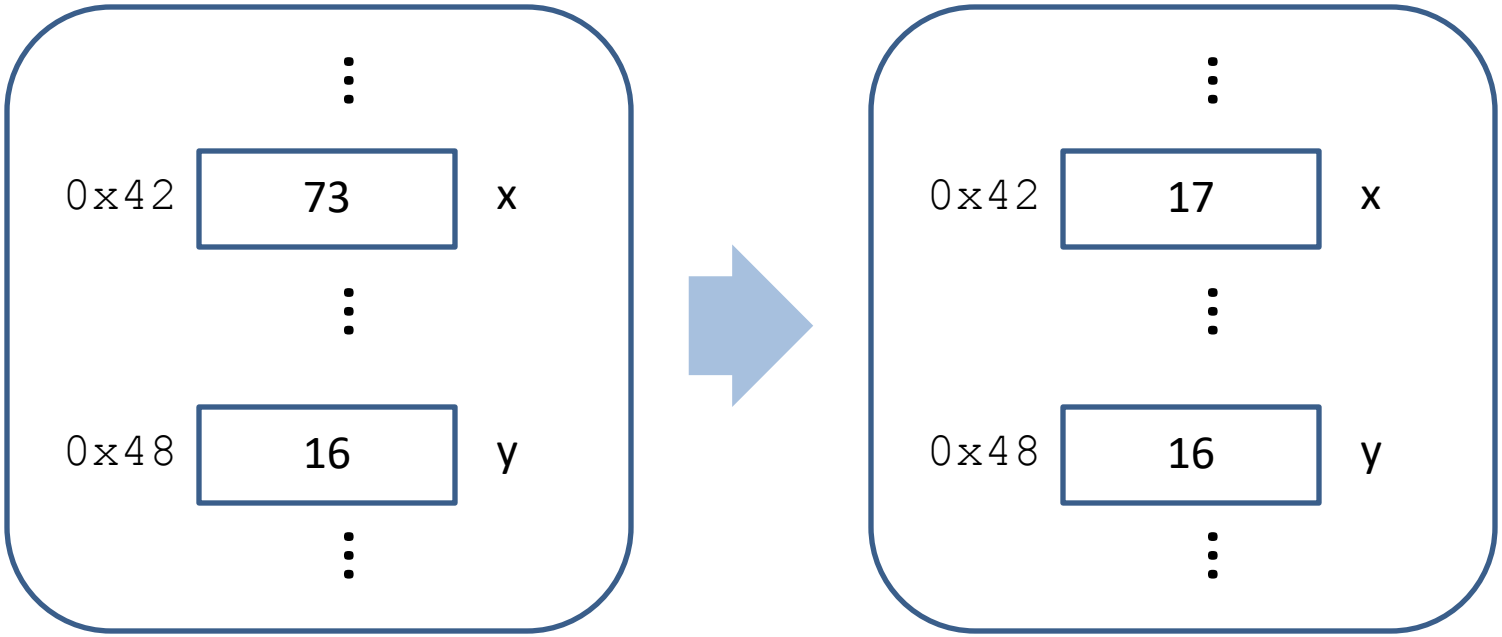
  `dst` is a memory location where data should be stored

  `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 (example)

```
x := A[1]
```
✓

```
x := A[A[1]]
```
✓

```
A[A[1]] := x + 1
```
✓

```
x + 1 := A[1]
```
✗

Example: Partial rules for L-value in C

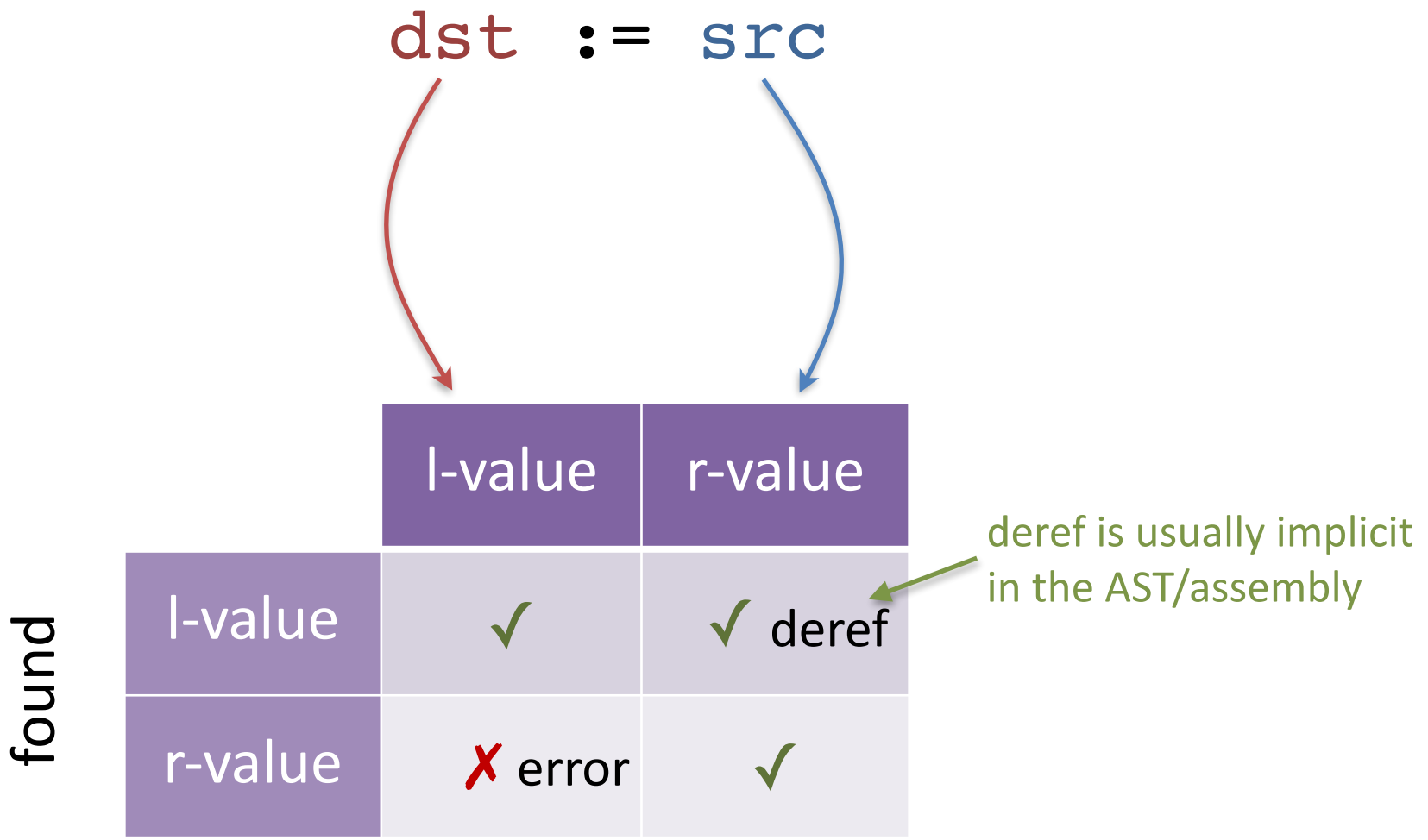
Type of e is pointer to T

l-value(e1) ≠⊥

Type of e2 is pointer to T
Type of e3 is integer

exp	**lvalue**	**rvalue**
id	location(id)	content(location(id))
const	⊥	value(const)
***e**	rvalue(e)	content(rvalue(e))
&e1	⊥	lvalue(e1)
e2 + e3	⊥	rvalue(e2) + sizeof(T)*rvalue(e3)

l-values and r-values



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
```

Subtyping

From typing rules: type τ_1

From typing rules: type τ_2

x = E;

x is l-value, E is l-value

τ_1 *is a superclass of* τ_2

can we assign?

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"
- Handled with corresponding typing rule

T <: S

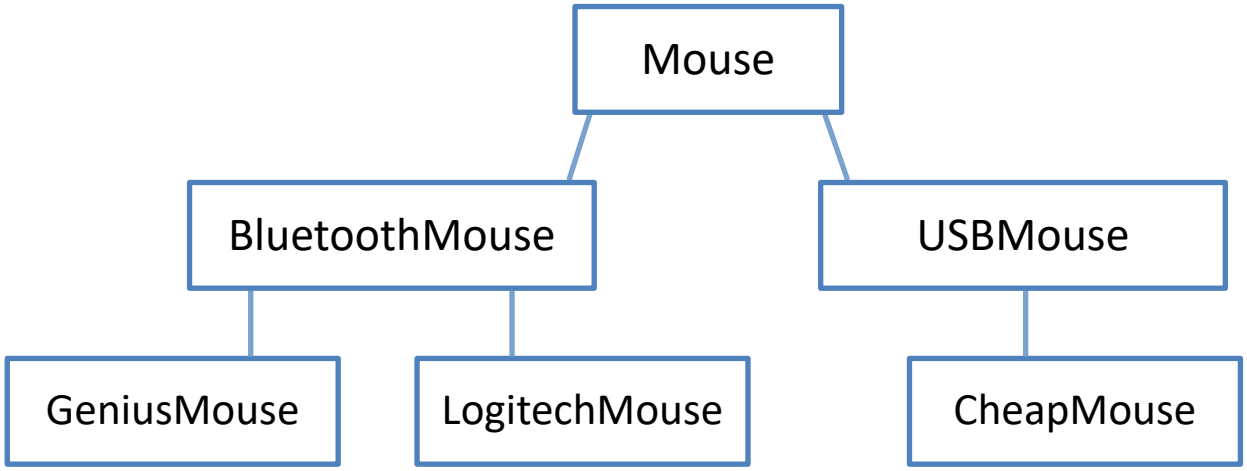
$$\frac{E_1 : T \qquad T <: S}{E_1 : S}$$

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

Usually can be expressed using a minimal type

Subtyping: Class Hierarchy

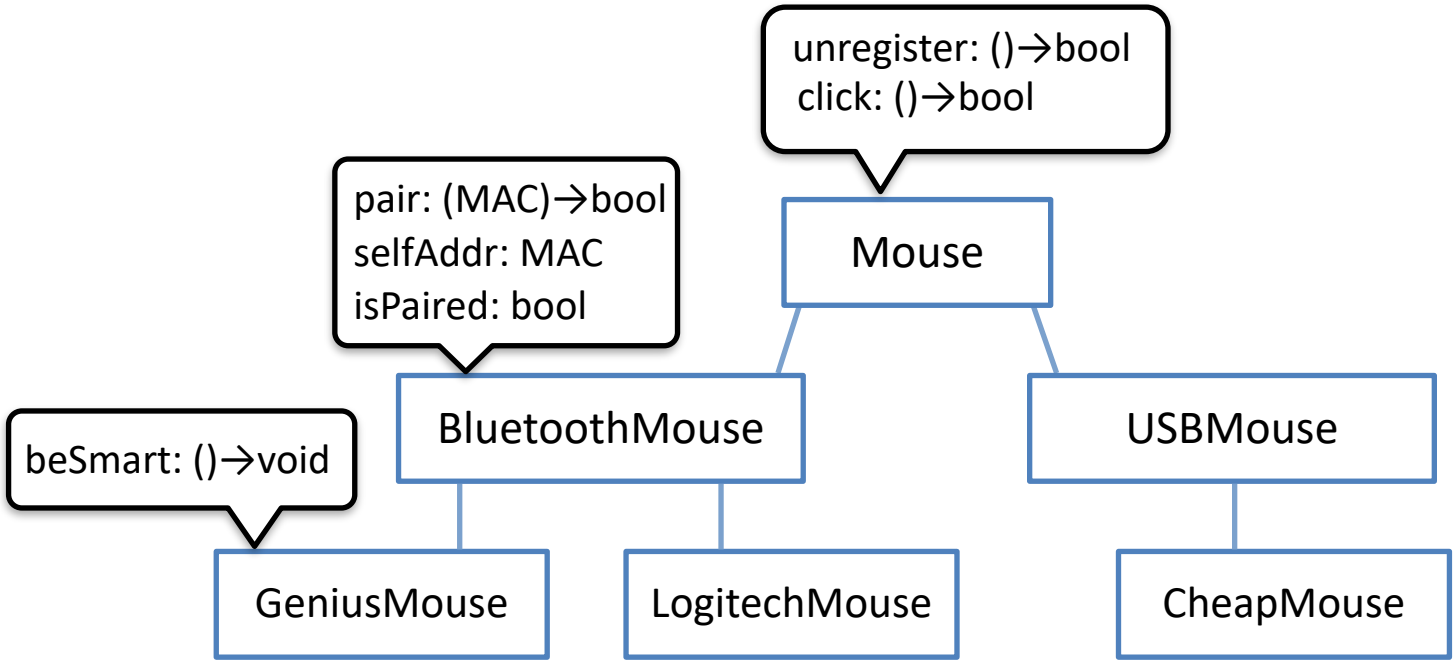
- The subtyping relationan induces a class hierarchy



Inheritance Tree

Subtyping: Inherited Members

- A subclass can be the target of a method of its superclass

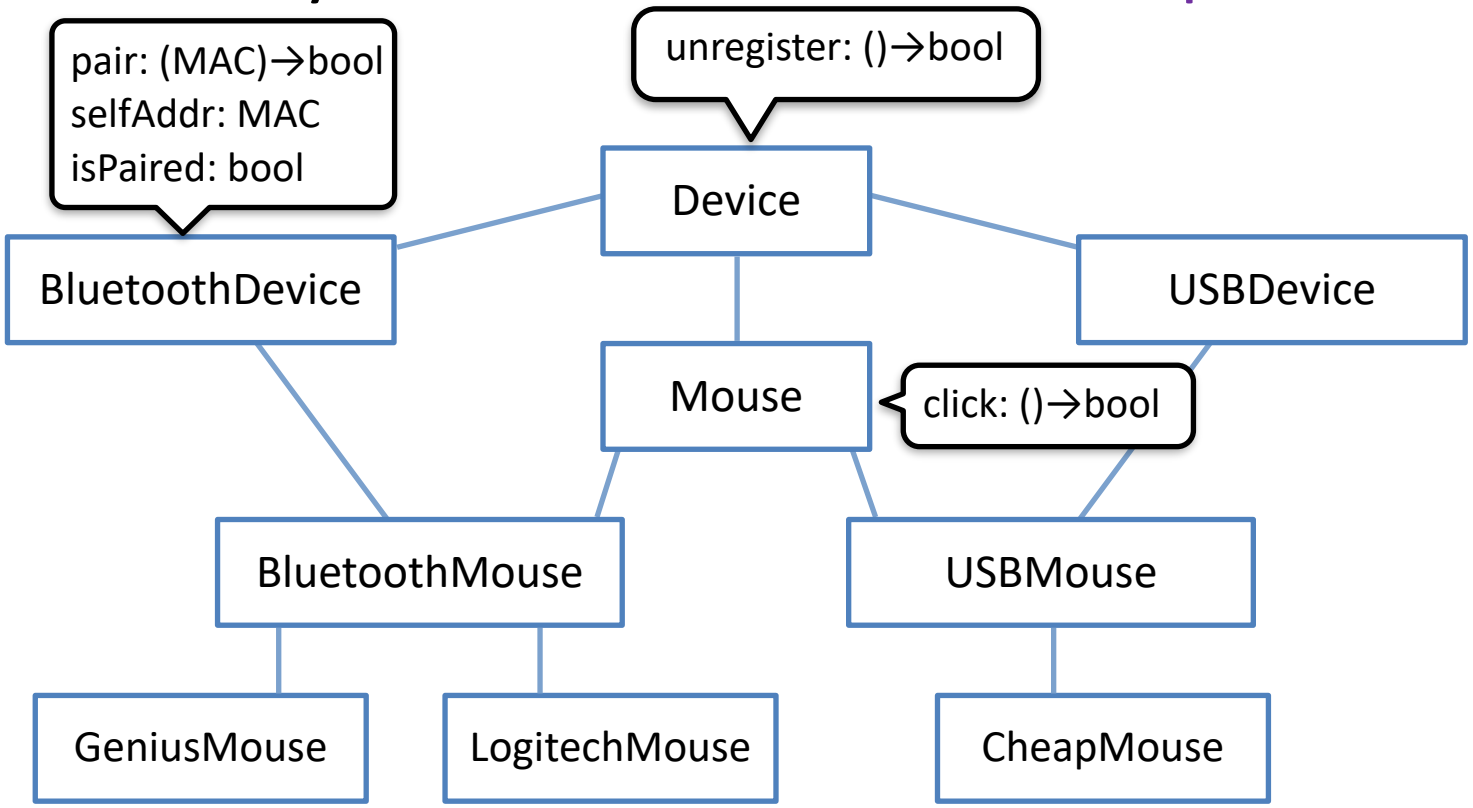



```
GeniusMouse m = getMouse();
m.unregister();
```

ance Tree

Subtyping: Multiple Inheritance

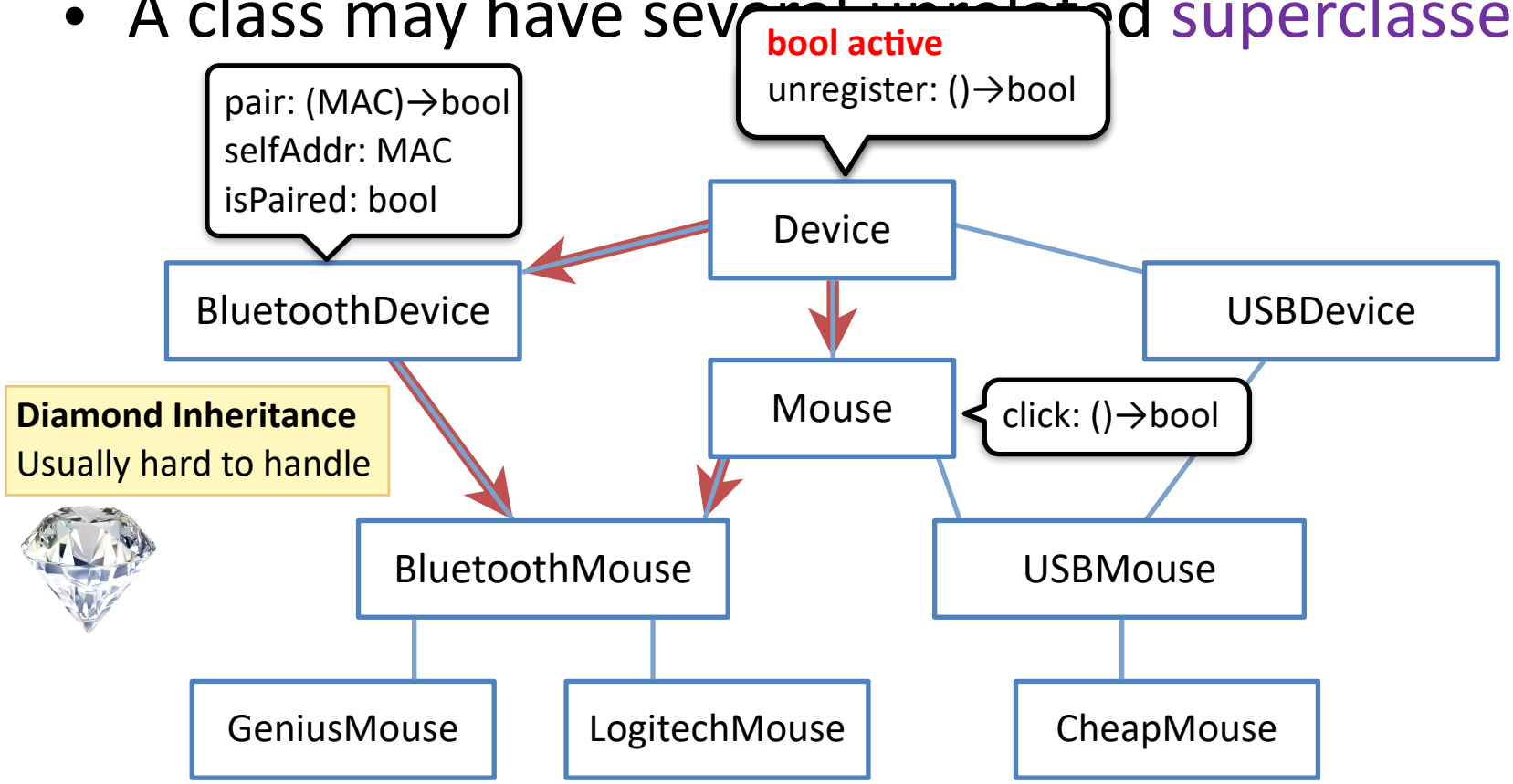
- A class may have several unrelated superclasses



Inheritance ~~Tree~~
DAG

Subtyping: Multiple Inheritance

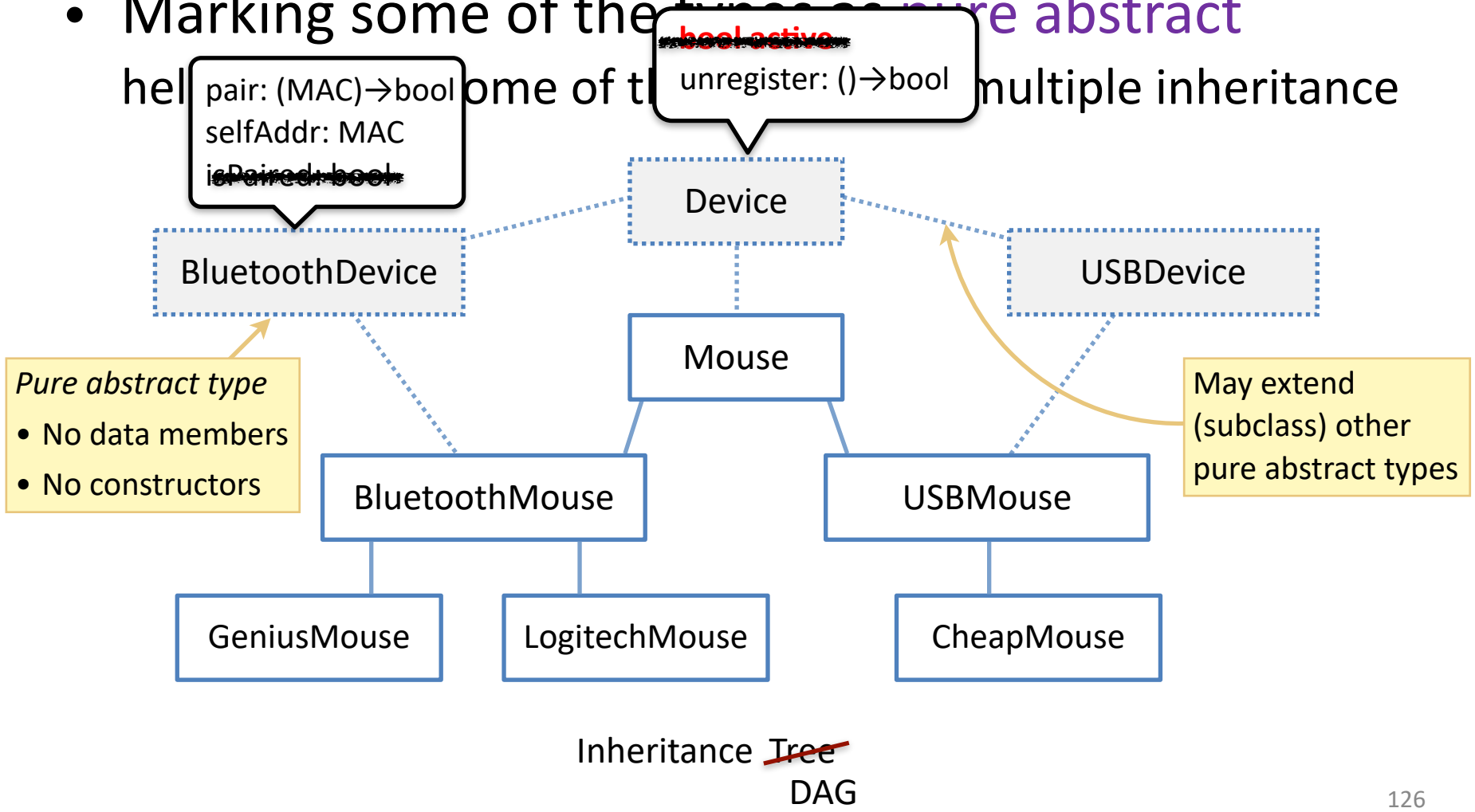
- A class may have several unrelated superclasses



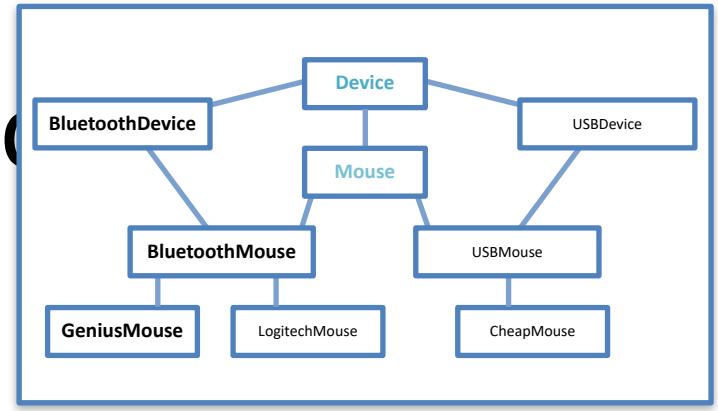
Inheritance ~~Tree~~ DAG

Subtyping: Pure Abstract Classes

- Marking some of the types as pure abstract helpers some of the multiple inheritance



Subtyping: Upc



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

Types: GeniusMouse, BluetoothMouse, Mouse, BluetoothDevice, Device

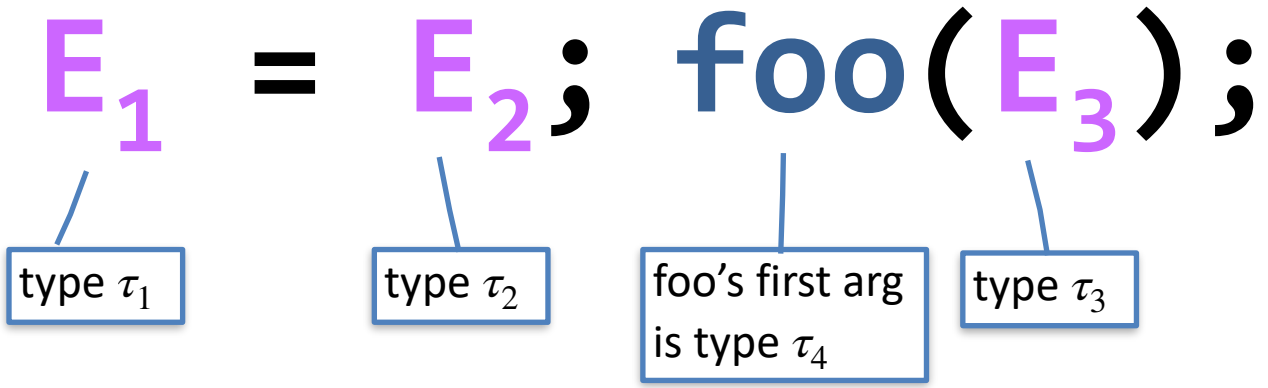
void registerDevice(Mouse m);

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

Types: Mouse, Device

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

Assignable



T_2 <u>is assignable to</u> T_1:

$T_1 = T_2$ *or* T_2 can be coerced to T_1 *or* T_2 is a subtype of T_1

Dynamic Checks

- Certain consistencies need to be checked at runtime in general
  - But can be statically checked in many cases
- Examples
  - Out-of-bound array access
  - Downcasting (from supertype to subtype)

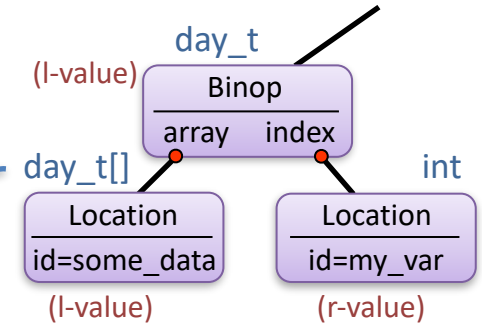
Summary (So Far)

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



Attribute Grammars

Semantic Analysis

- Identification
- Context checking

How does this magic happen?

- **Ad-hoc:** Traverse the AST top-down and maintain the symbol table

- **Syntax Directed Translation:**
  - Attach semantics attributes to grammar symbols
  - Define how to evaluate them
  - Let an attribute grammar "engine" do the evaluation

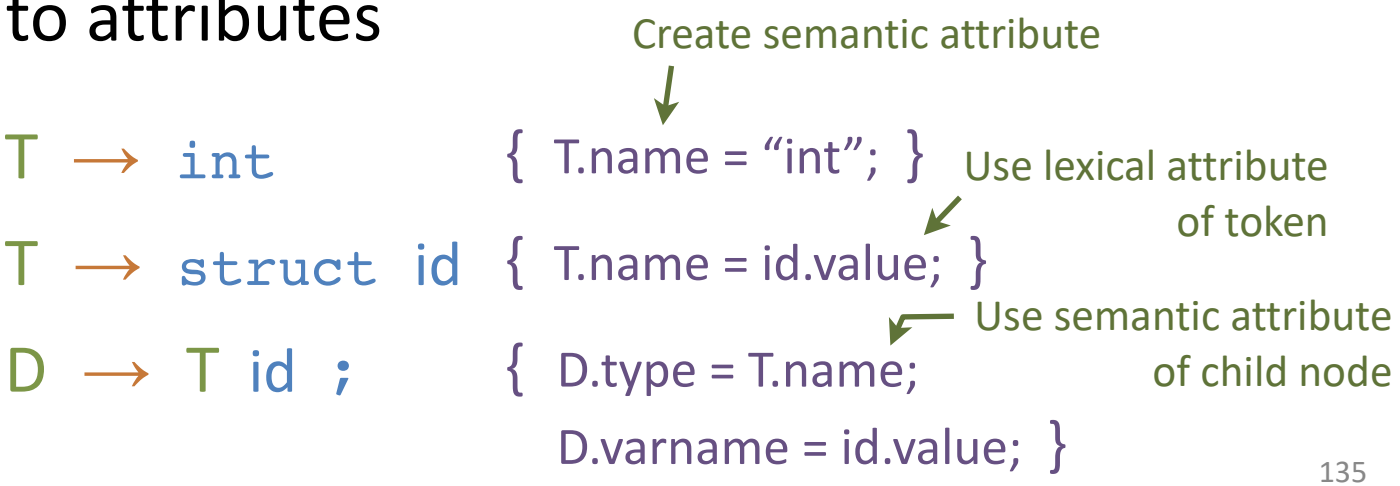
Can help design ad-hoc processing

Syntax Directed Translation

- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - Define how to update the attributes
- Attribute grammars [Knuth 68]

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



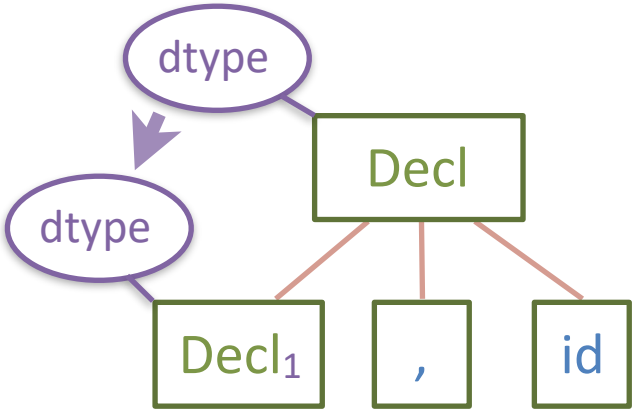
Indexed symbols

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

Decl → Decl , id

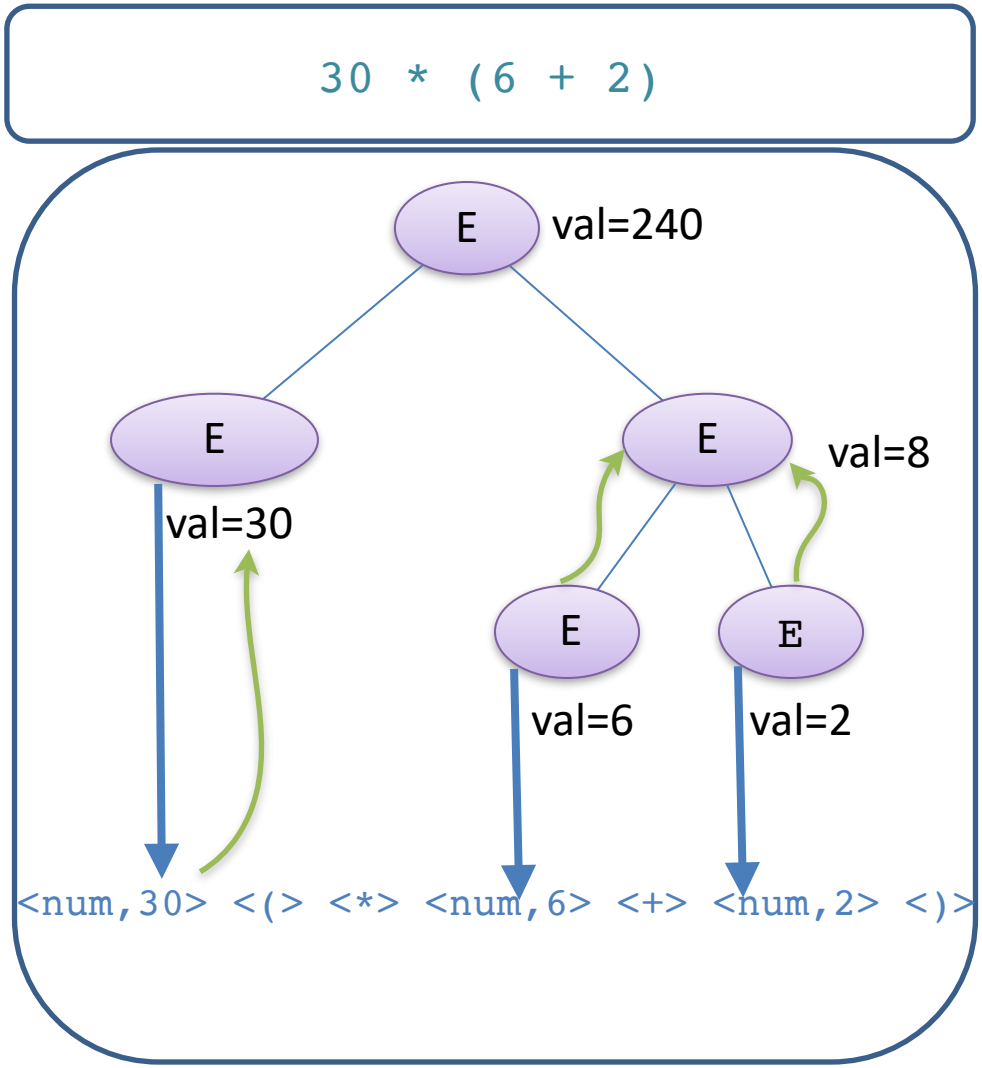
Decl → $Decl_1$, id

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



Example — Evaluating Arithmetic Expressions

(During Bottom-Up Parsing)



Production	Semantic Rule
$E \rightarrow num$	E.val = num.val
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E \rightarrow E_1 * E_2$	$E.val = E_1.val * E_2.val$

Semantic Analysis — Reminder

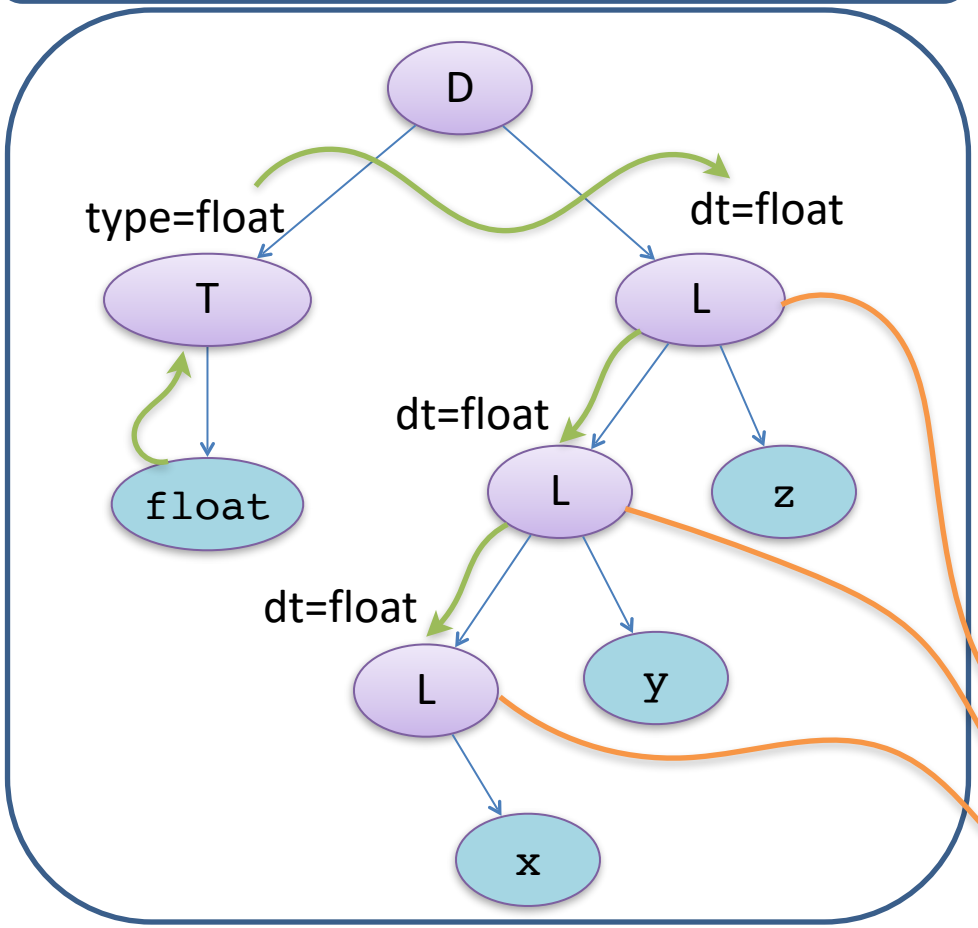
- Identification
  - Read declarations, build symbol table & type table
  - Associate declaration and uses

- Context checking
  - Type checking: Check that the program is type-safe
    - *e.g.*, the condition in an if-statement is a Boolean

Example — Evaluating Declarations

(on the AST) + (Side Effects)

```
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.name, L.dt)
L → id	addType(id.name, L.dt)

name	pos	type
z	…	float
y	…	float
x	…	float

Attribute Evaluation

- Build the AST/Parse tree
  - Sometimes can be done during parsing
- 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

Side Effects

Production	Semantic Rule
D → T L	L.in = 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)

Convention:
Add dummy variables for operations that change the state of the checker (*e.g.* modify symbol table).

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

Who're you calling dummy?!

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.

Attribute Dependencies

- All semantic actions take the form

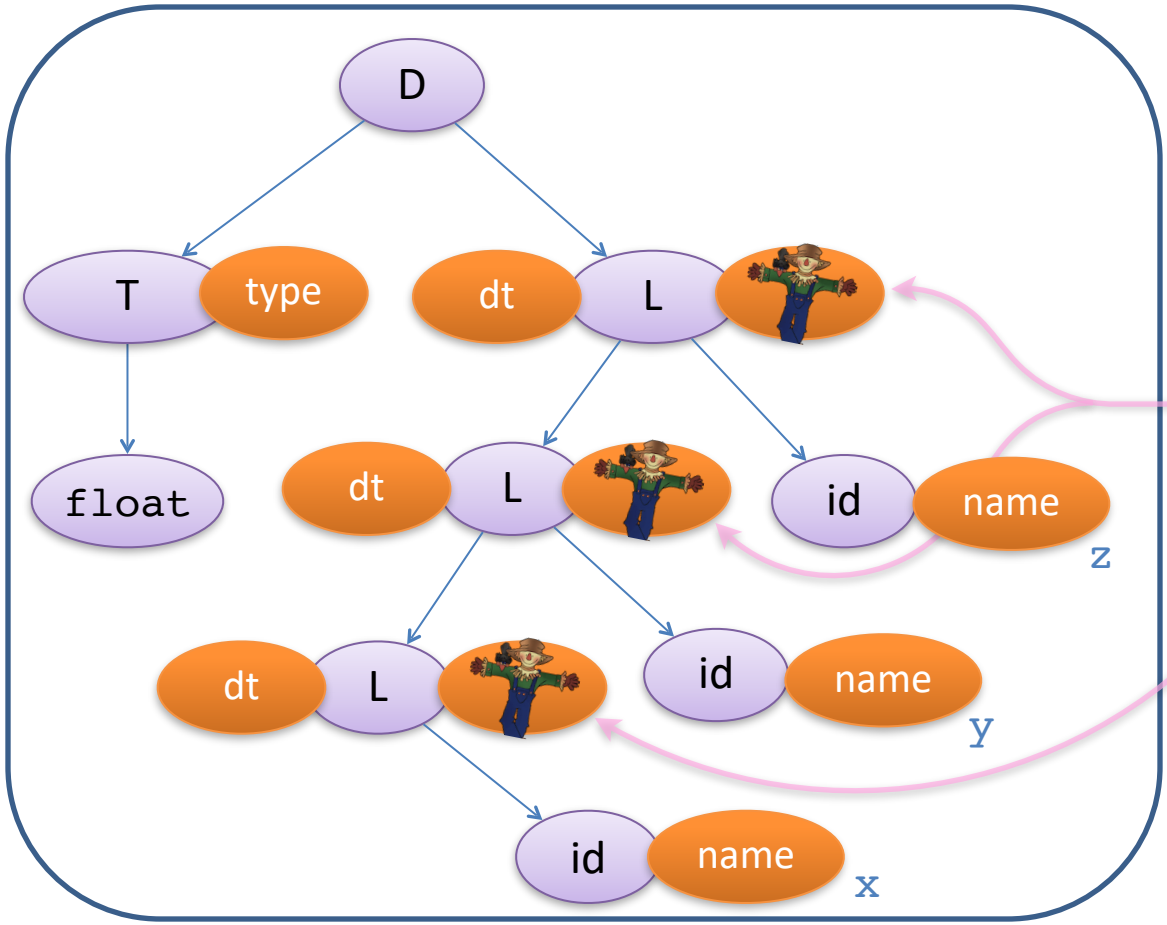
$$a_1 = f_1(b_{1\text{-}1}, b_{1\text{-}2}, \ldots)$$
$$a_2 = f_2(b_{2\text{-}1}, b_{2\text{-}2}, \ldots)$$

  - 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

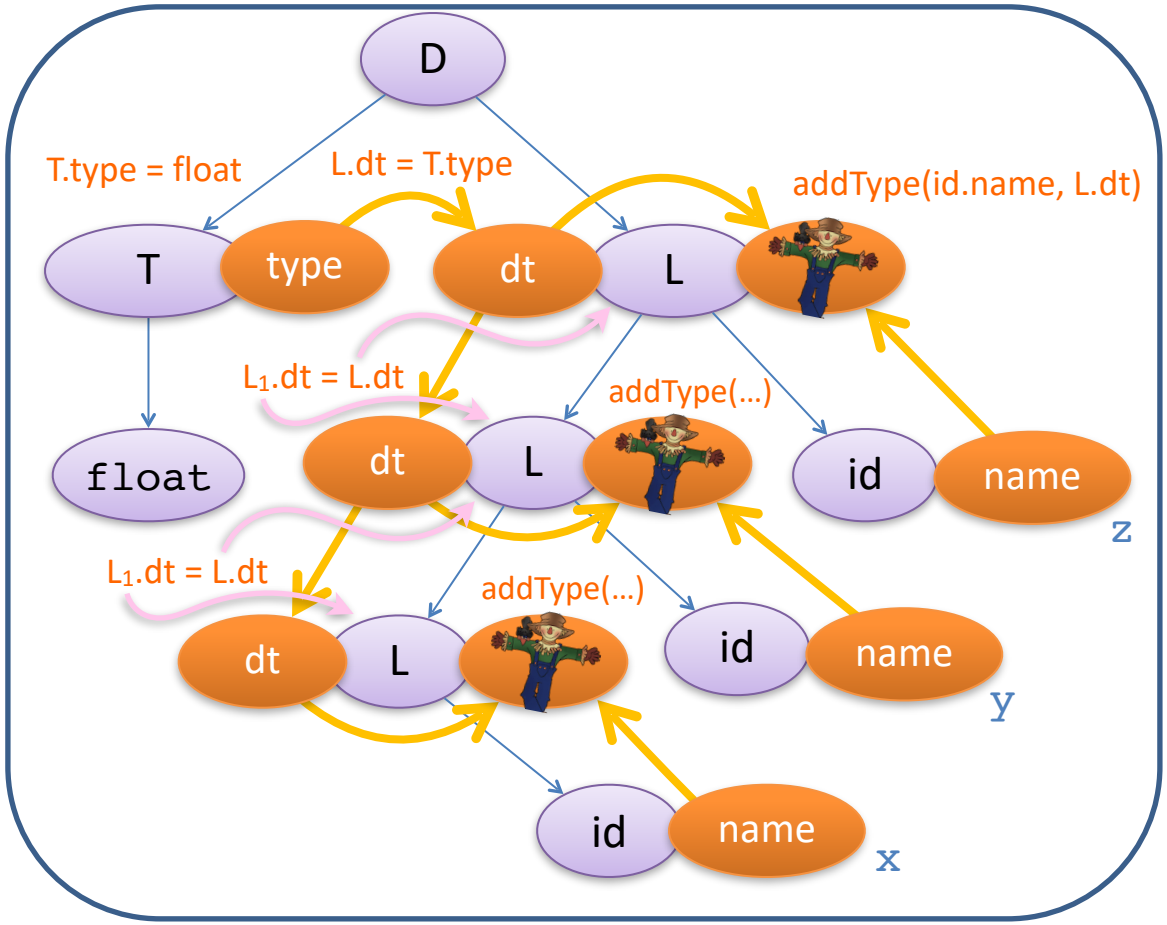
```
float x, y, z
```



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

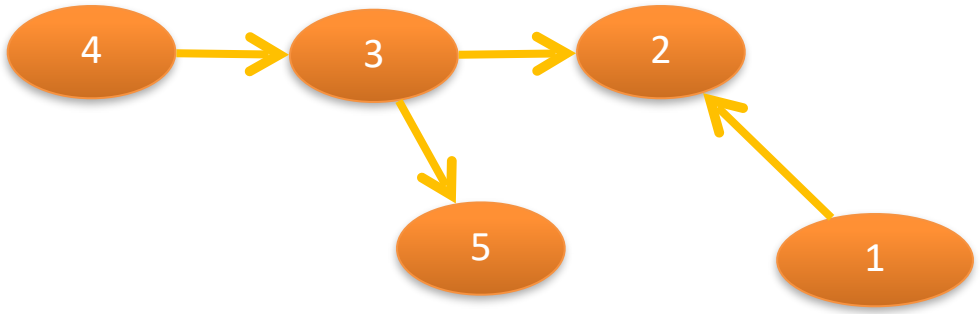
```
float x, y, z
```



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

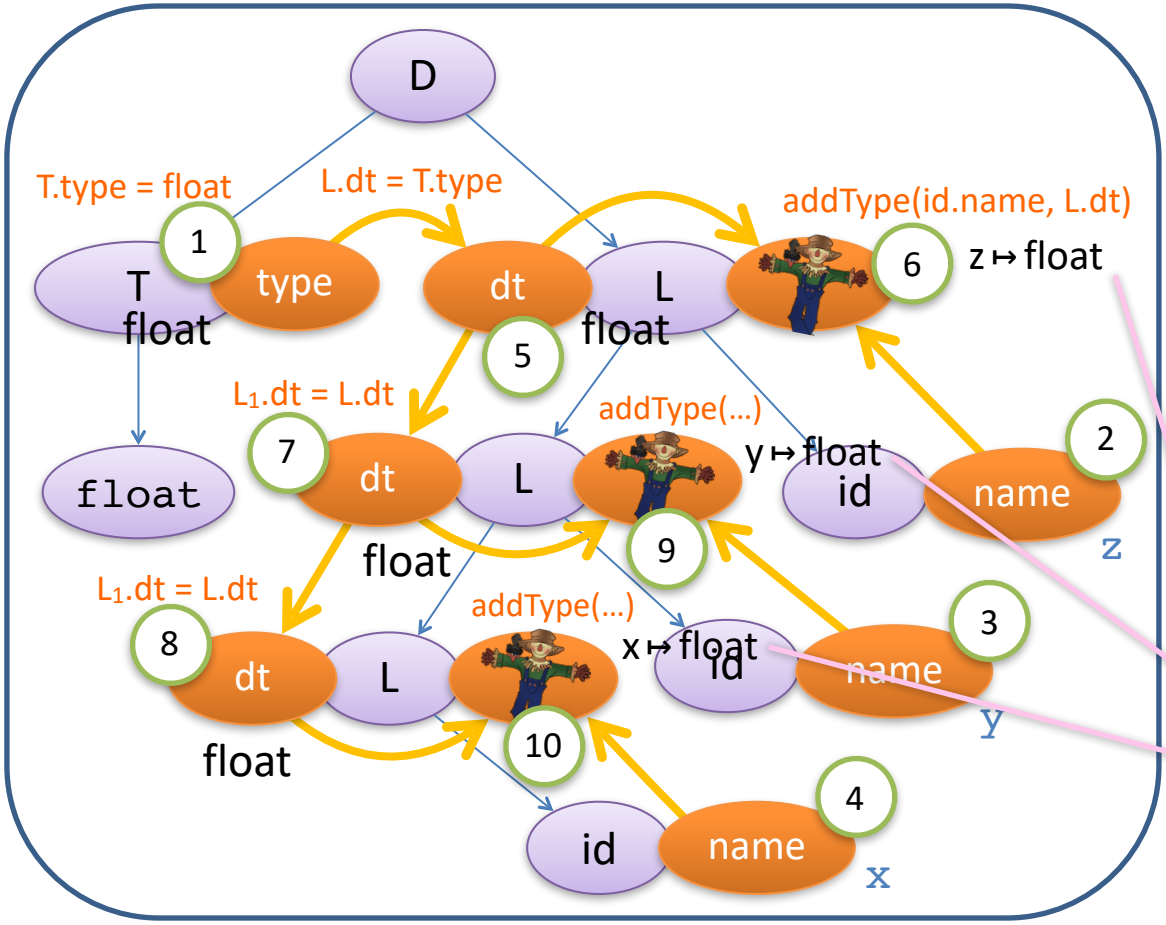
- For a graph G=(V, E), $|V|=k$
- Ordering of the nodes as $\langle v_1, v_2, \ldots, v_k \rangle$ such that for every edge $(v_i, v_j) \in E$, $i < j$



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

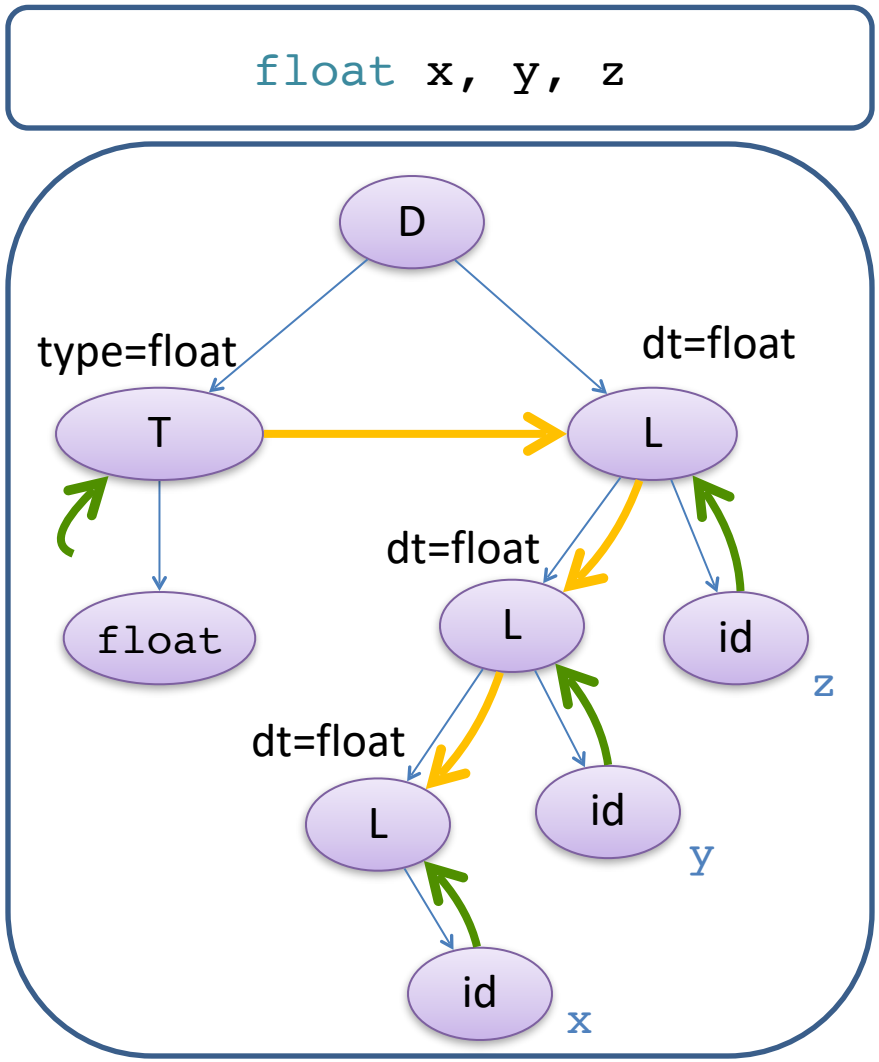
- *Synthesized* attributes
  - Attributes whose values at a given node depend ***only*** on the attributes of its children
- *Inherited* attributes
  - Attributes whose values at a given node depend ***only*** on the attributes of its parent and siblings

Synthesized vs. Inherited Attributes

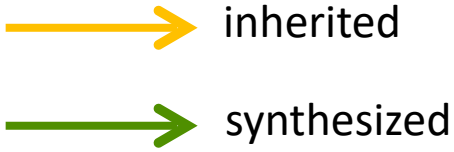
- *Synthesized* attributes
  - Attributes whose values at a given node depend ***only*** on the attributes of its children (and itself)
- *Inherited* attributes
  - Attributes whose values at a given node depend ***only*** on the attributes of its parent and siblings

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

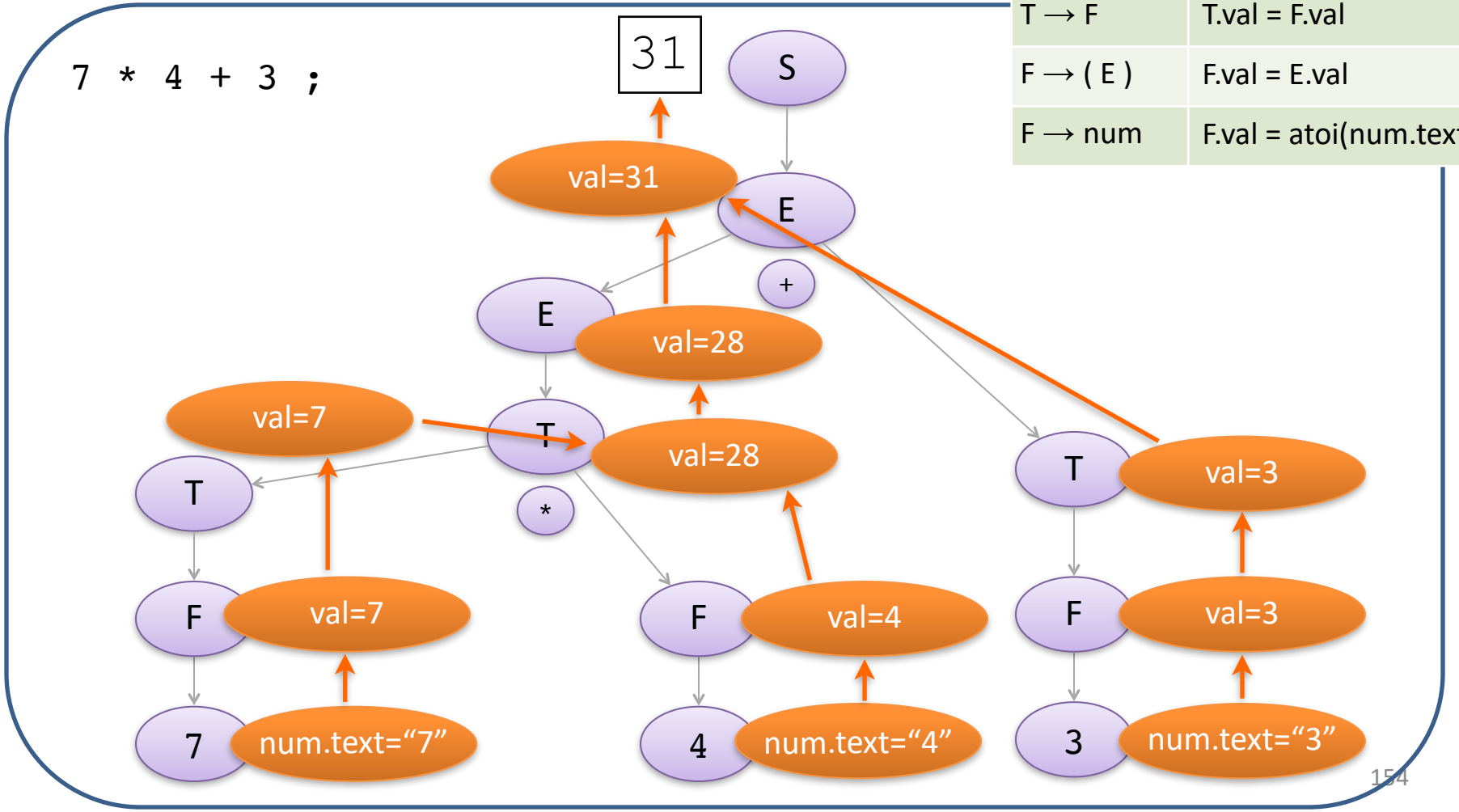
Production	Semantic Rule
$S \rightarrow E$;	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow num$	F.val = atoi(num.text)

This is an interpreter, not a compiler

S-attributed Grammars

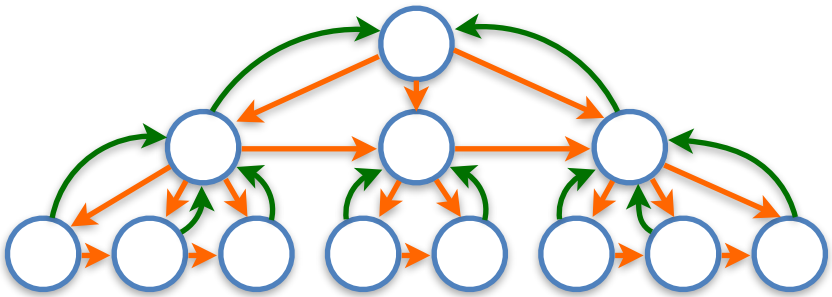
Production	Semantic Rule
S → E ;	print(E.val)
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → num	F.val = atoi(num.text)

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

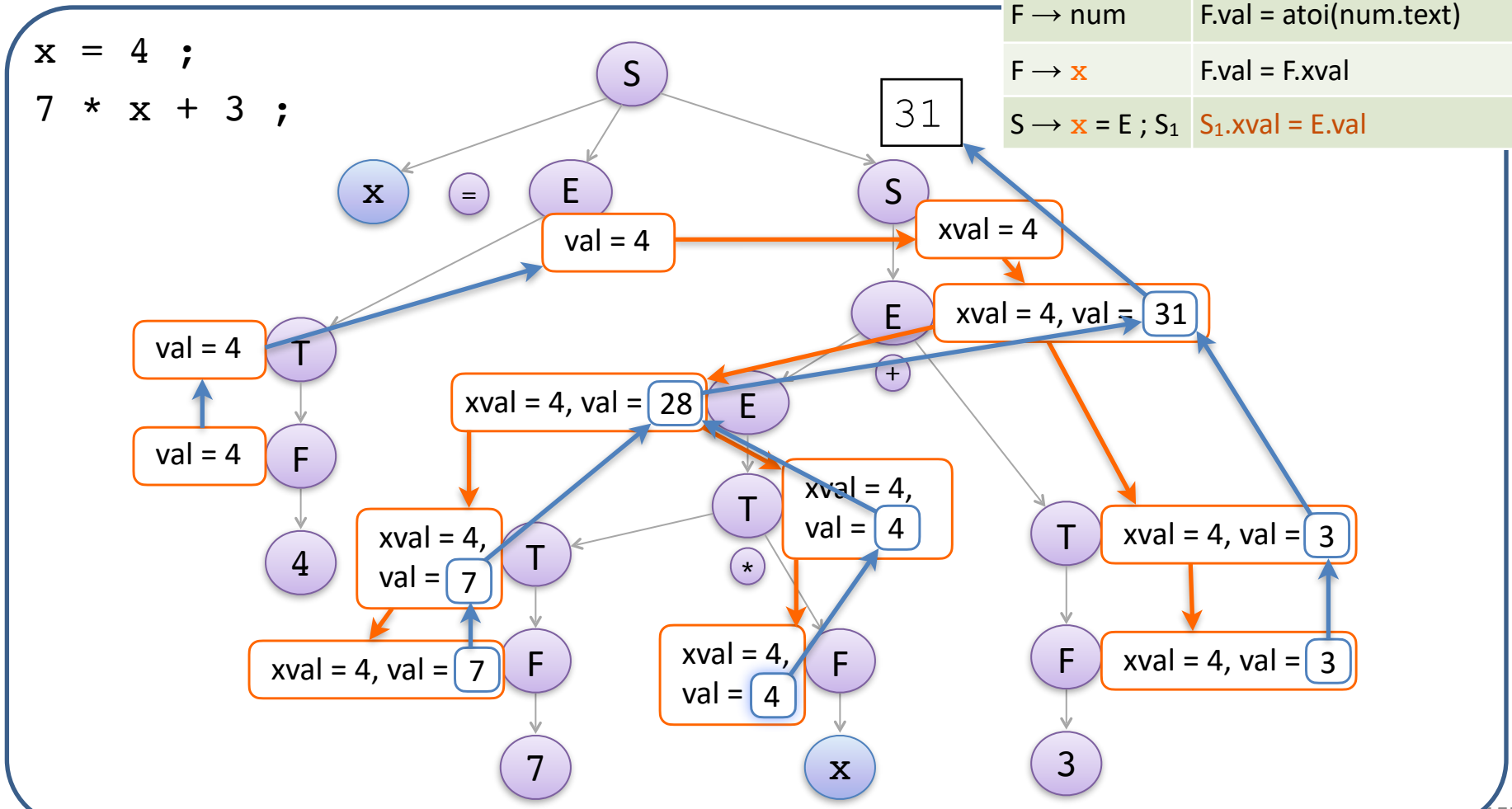
e.g., `x = 4 ; 7 * x + 3 ;`

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	T.xval = E.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	

?

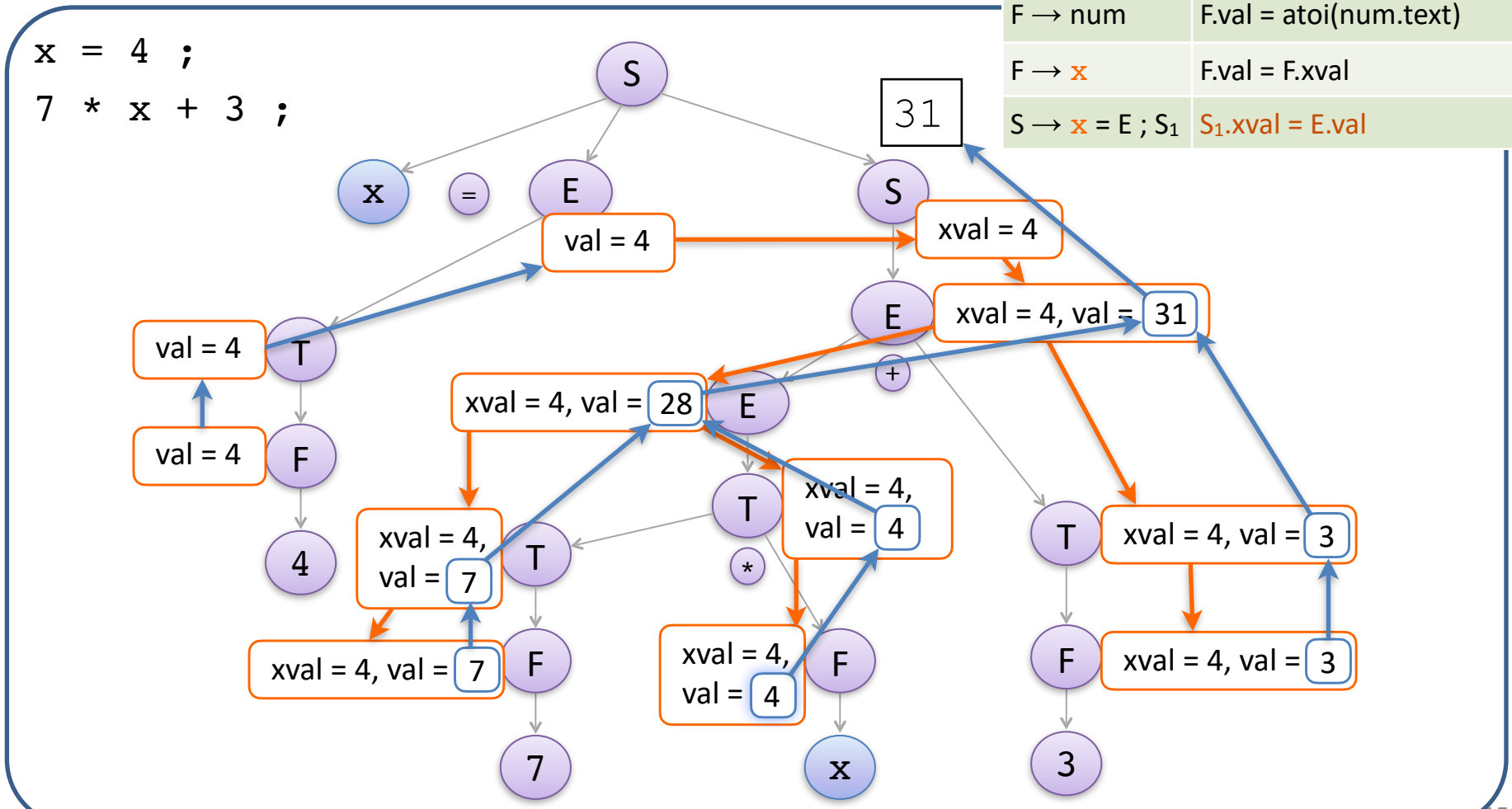
L-attributed Grammars

S → E ;	print(E.val)	E.xval = S.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	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

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	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.name, 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.name, ih.dt);
    A({dt ↦ ih.in});
}

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

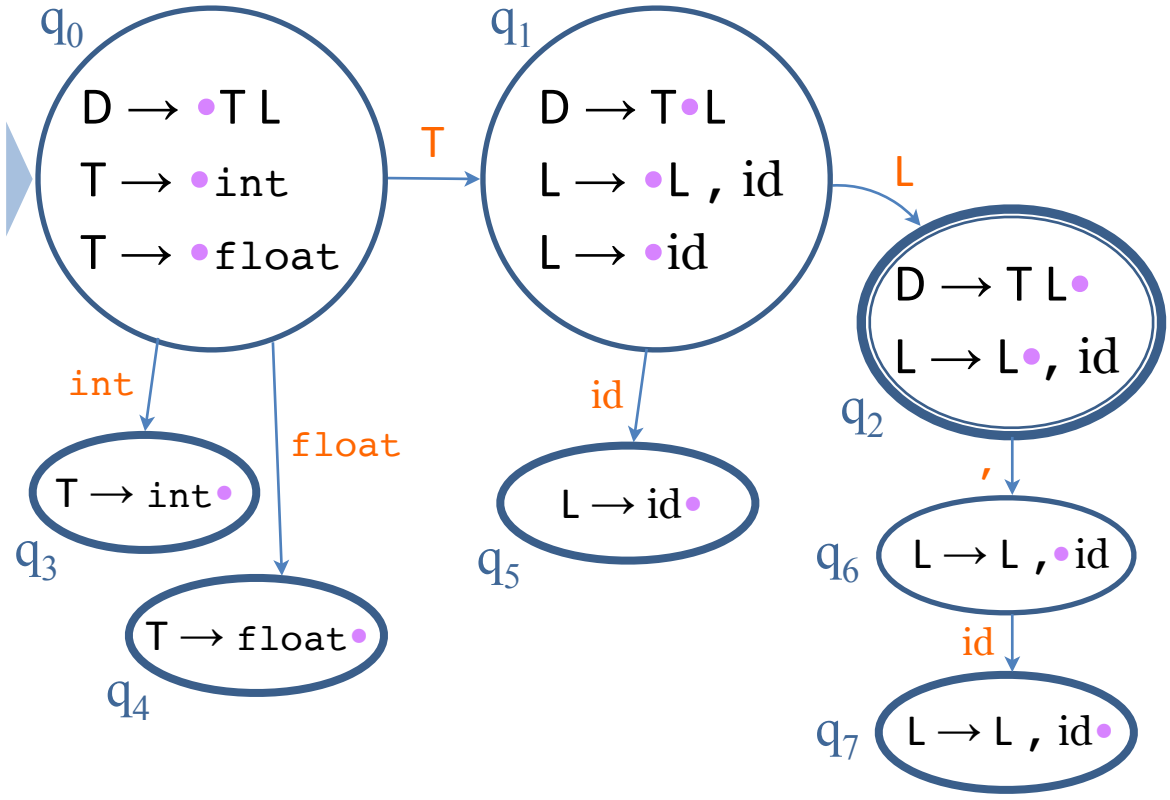
```
T() {
    if (current == INT) {
        match(INT);
        return {type ↦ int};
    } else if (current == FLOAT) {
        match(float);
        return {type ↦ float};
    } else error();
}
```

L-attributed Grammars

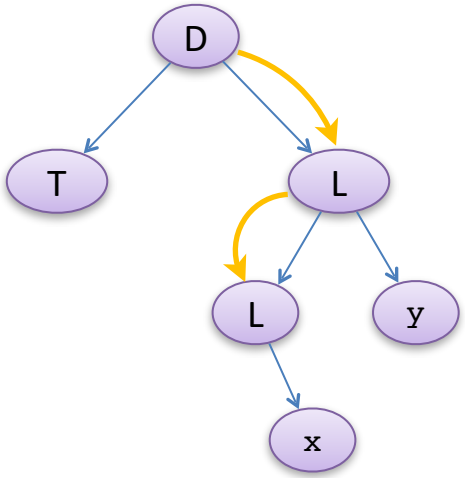
- In shift-reduce parsers:

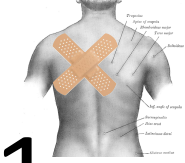
  Semantic actions are performed during reduce.

  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)





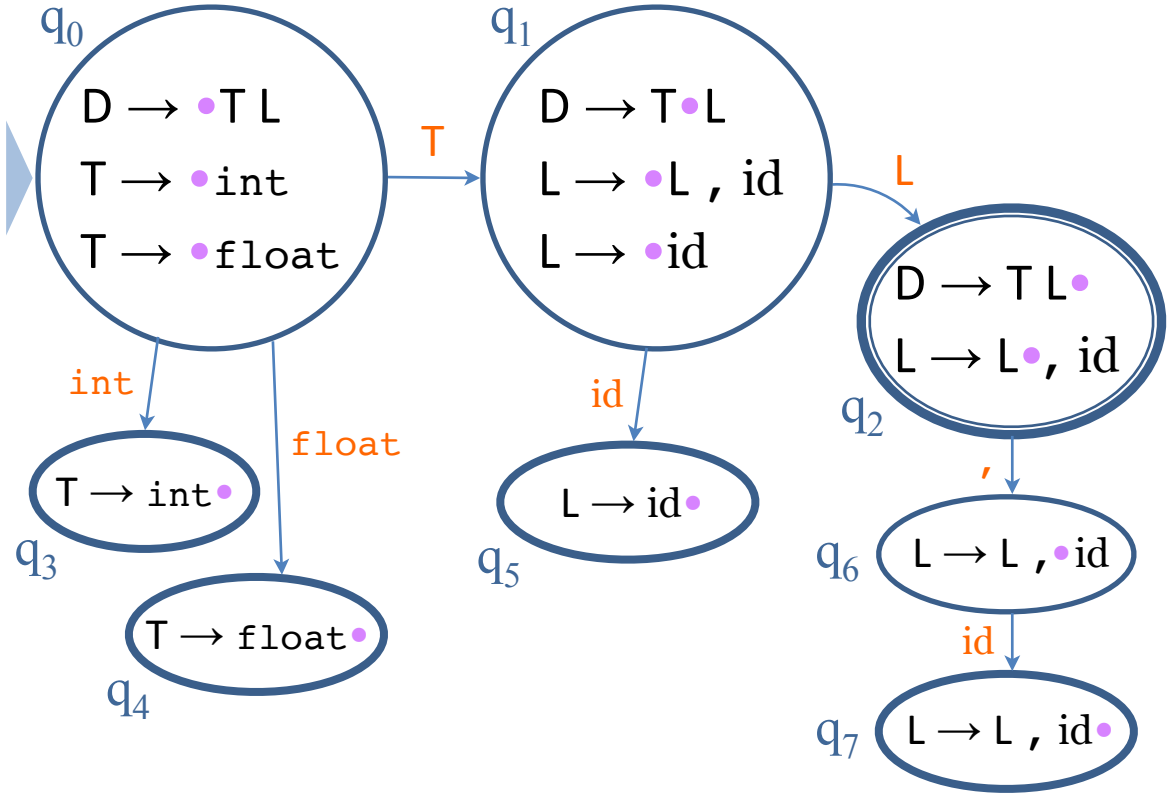
L- ➔ "S"-attributed Grammars - v.1

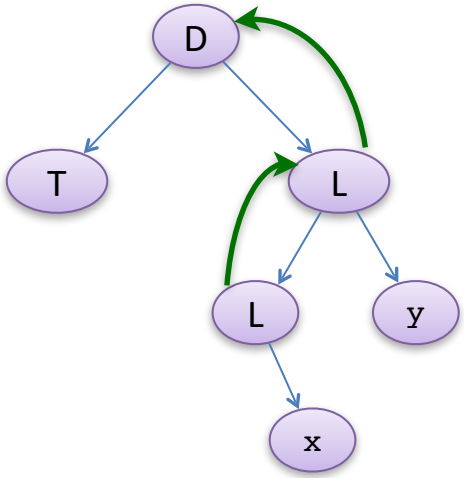
- In shift-reduce parsers:

  Semantic actions are performed during reduce.

  We have a problem — tree is built bottom-up

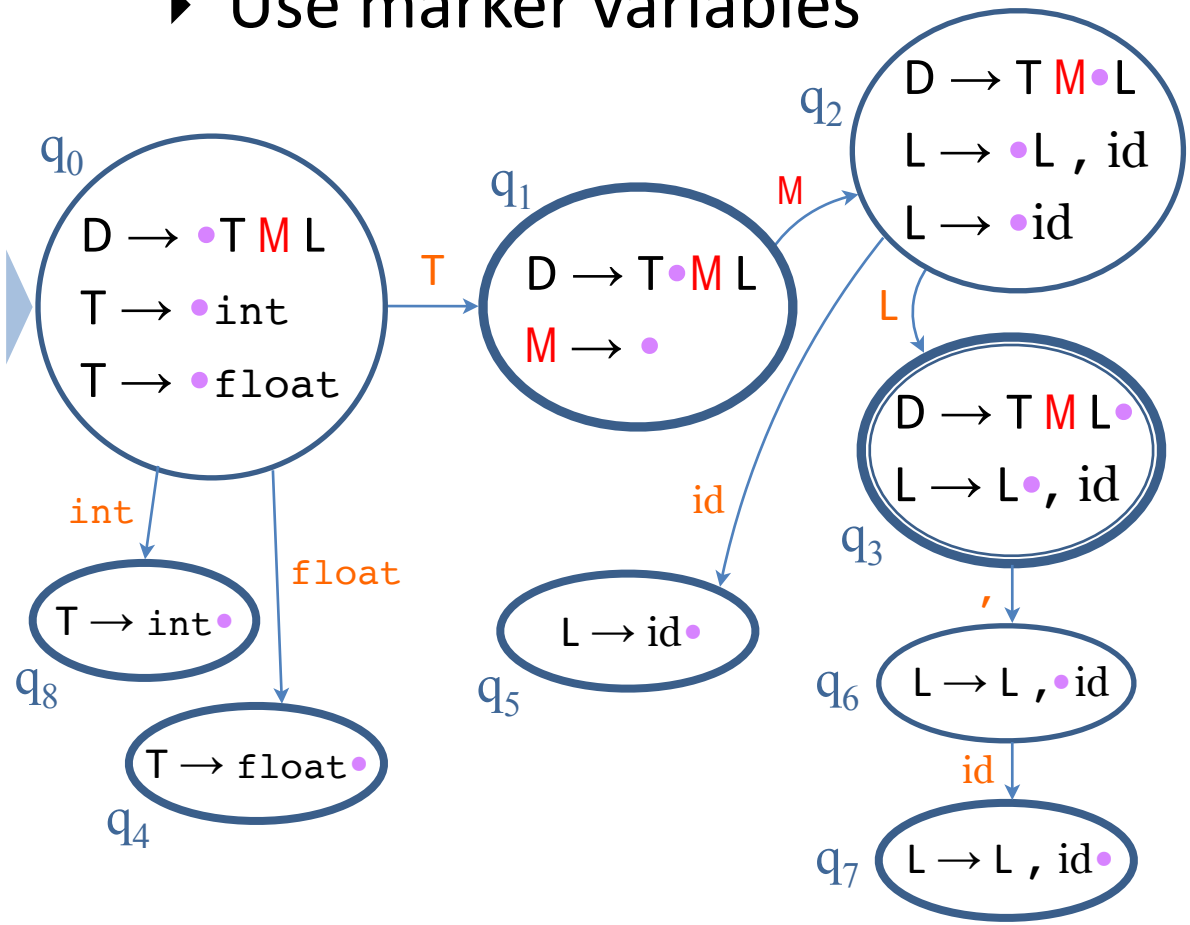
Prod.	Semantic Rule
D → T L	foreach id in L.list addType(ent,T.type)
T → int	T.type = integer
T → float	T.type = float
L → L_1 , id	L.list = L_1.list + [id.entry]
L → id	L=[id.entry]



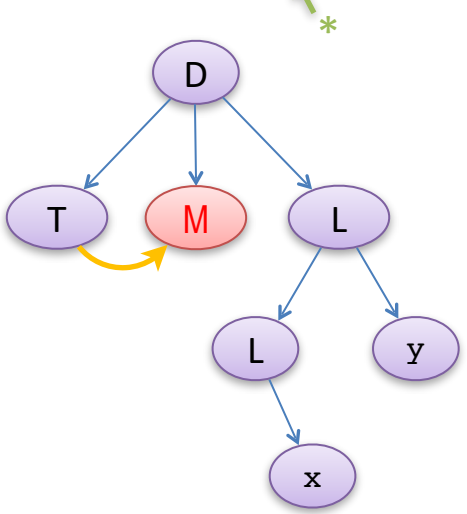


L- ➔ "S"-attributed Grammars - v.2

- 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



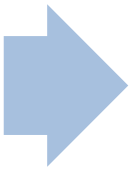
Handling Break: L-Attribute Grammar

```
parser code {:
public int loop_count = 0 ;
:}

stm       : :=
              IF '(' exp ')' stm
            | IF '(' exp ')' stm ELSE stm
            | WHILE  '(' exp ')' m stm {:  loop_count--;   :}
            | '{' stmList '}'
            |   BREAK ';'   {:  if  (loop_count == 0)  system.error.println(
                                          "Break must be enclosed within a loop");
                              :}
          ;
stmList ::=  stmList stm
            | /* empty */
            ;
m         ::= /* empty */  {: loop_count++ ;  :}
            ;
```

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 SLR(1)

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.name, dtype)
M → ε	dtype = T.type
N → ε	dtype = array(T.type)

This grammar is not even LR(1)

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;
}
```

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
    - Markers, delayed computation, global variables

The (Front) End