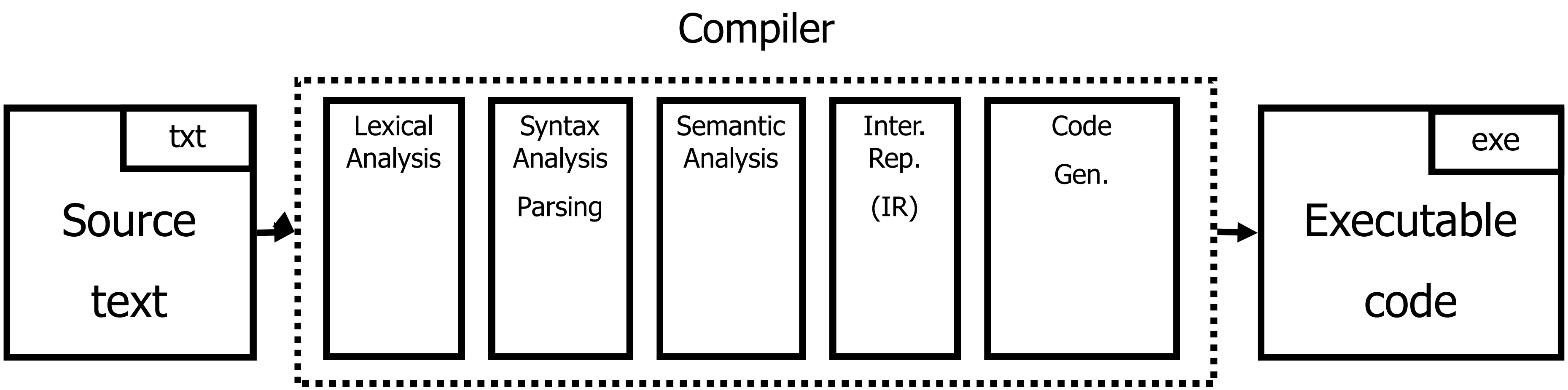
Lecture 06 – Semantic Analysis

THEORY OF COMPILATION

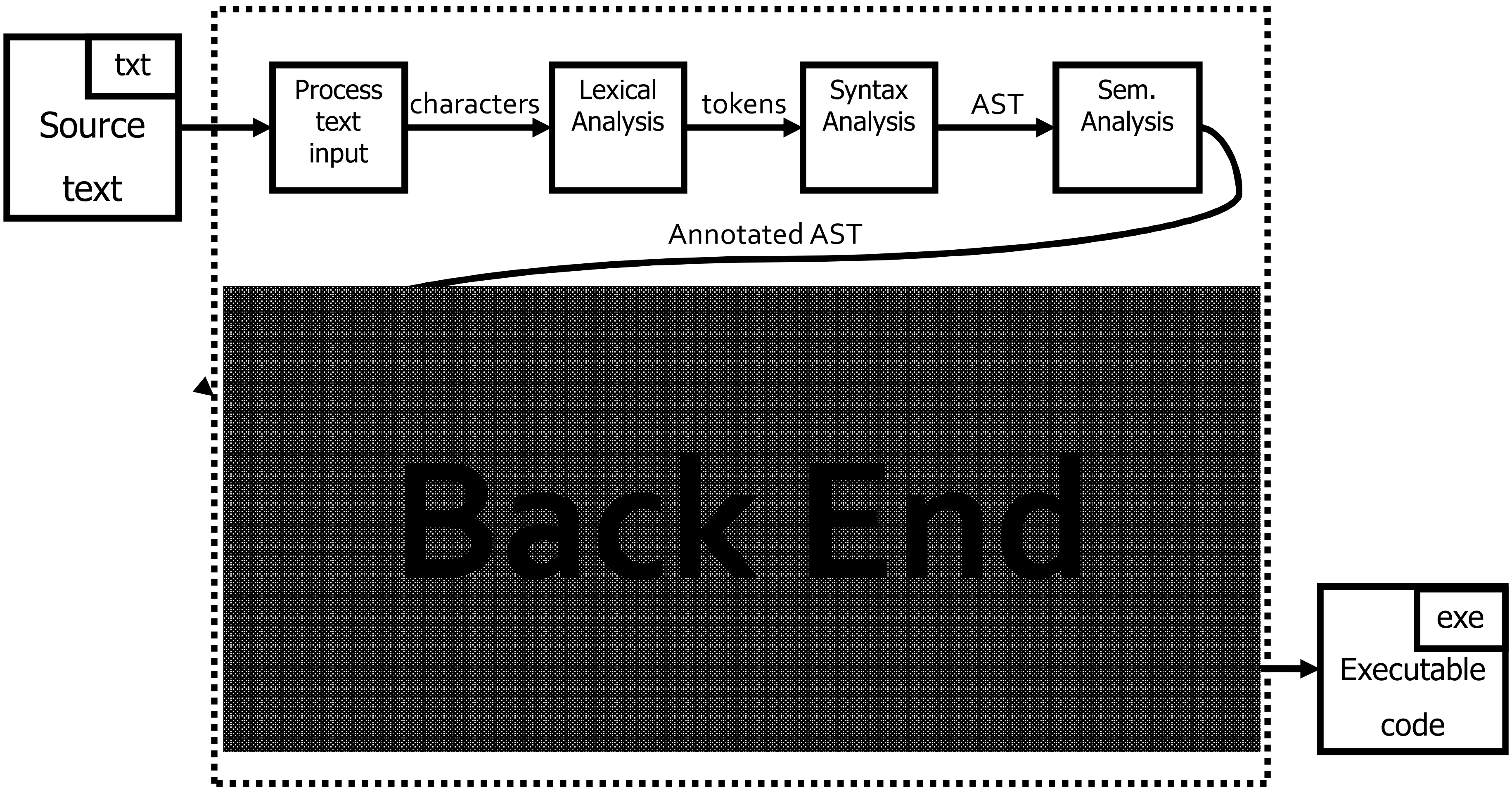
Eran Yahav

You are here



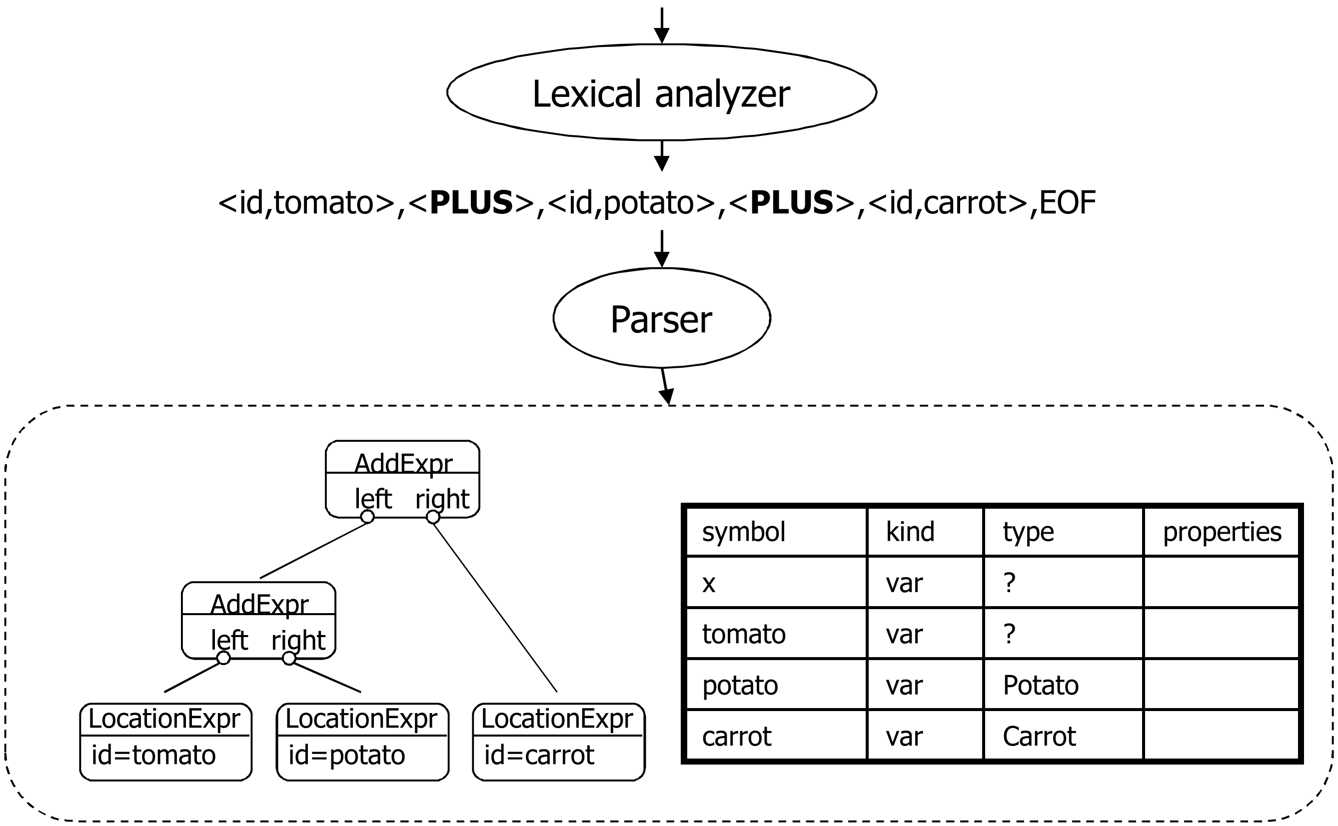
Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
Inter.
Rep.
(IR)
Code
Gen.
exe
Executable
code

You are here…


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

What we want

Potato potato;
Carrot carrot;
x = tomato + potato + carrot



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

tomato is undefined

potato used before initialized

Cannot add Potato and Carrot

Contextual Analysis

- Often called “Semantic analysis”

- Properties that cannot be formulated via CFG
  - Type checking
  - Declare before use
    - Identifying the same word “w” re-appearing – wbw
  - Initialization
  - …

- Properties that are hard to formulate via CFG
  - “break” only appears inside a loop
  - …

- Processing of the AST

Contextual 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;
}
```

- Forward references?
- Languages that don’t require declarations?

Symbol table

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

name	pos	type	...
month	1	RANGE[1..12]	
month_name	…	…	
…			

- 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"

int variable named "i"

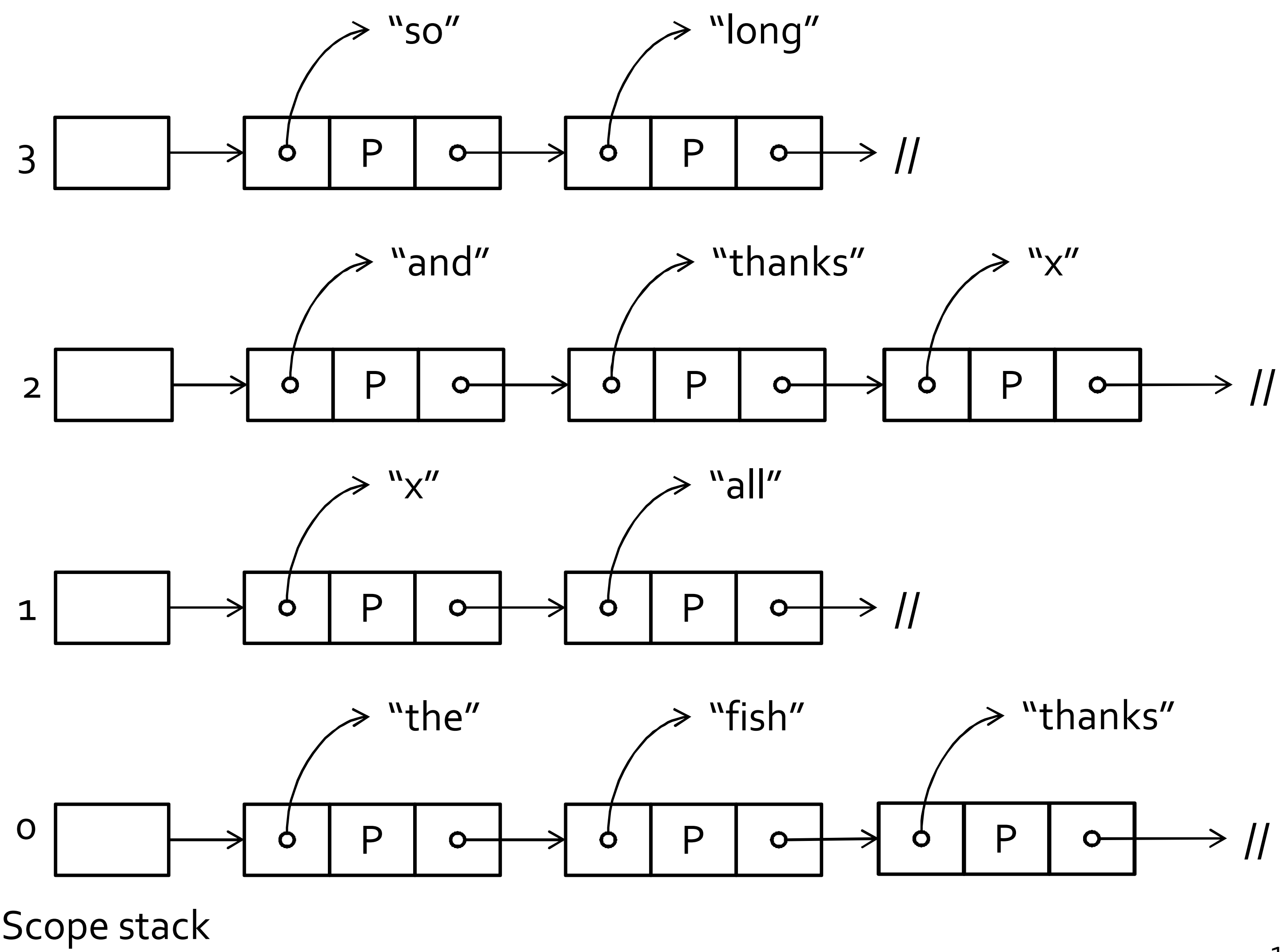
Scopes

- 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 top scope
- Identifier Lookup
  - Search for identifier top-down in scope stack

Scope-structured symbol table

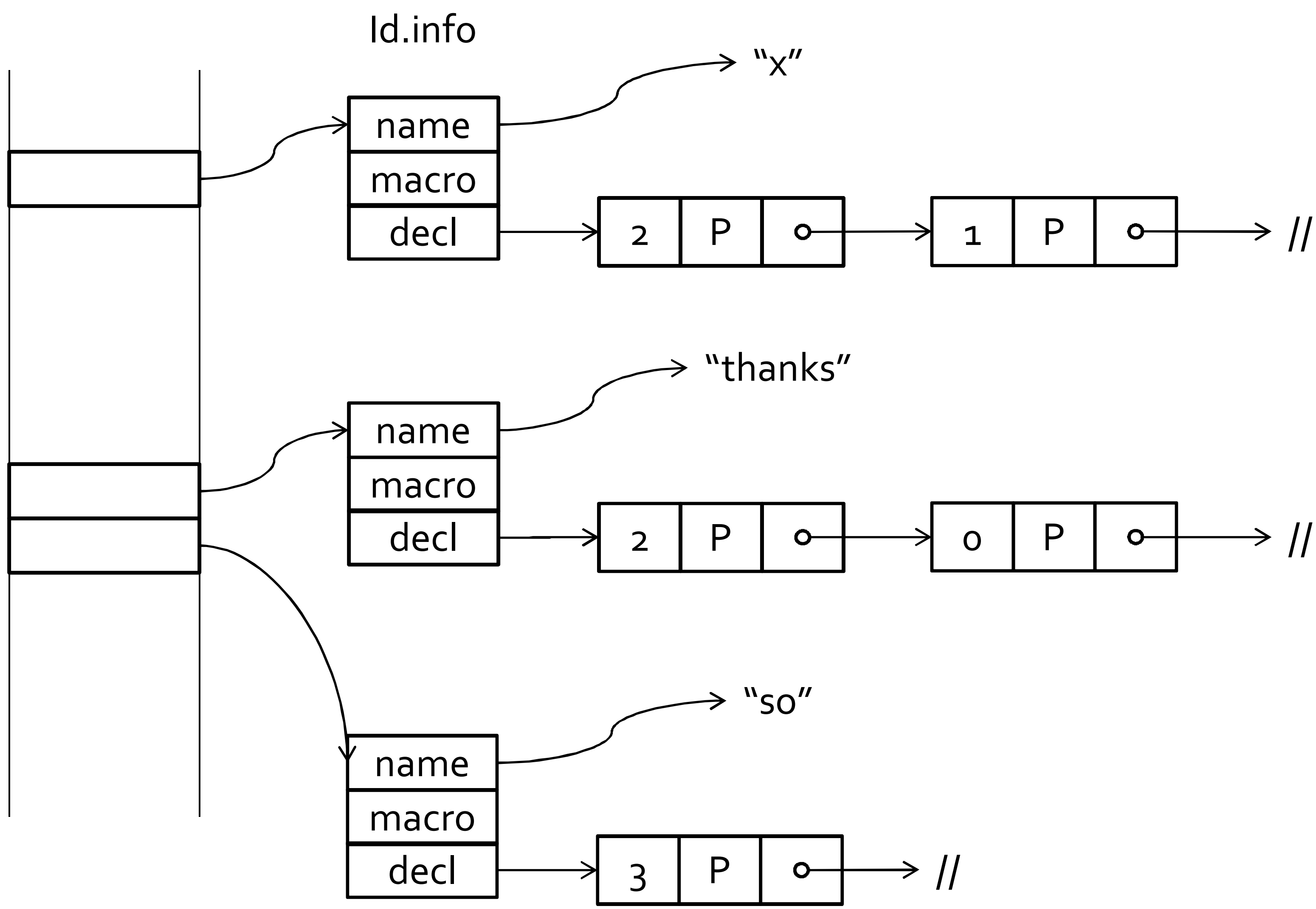
```
{
int the=1;
int fish=2;
Int thanks=3;
{
 int x = 42;
 int all = 73;
  {
…
  }
 }
}
```



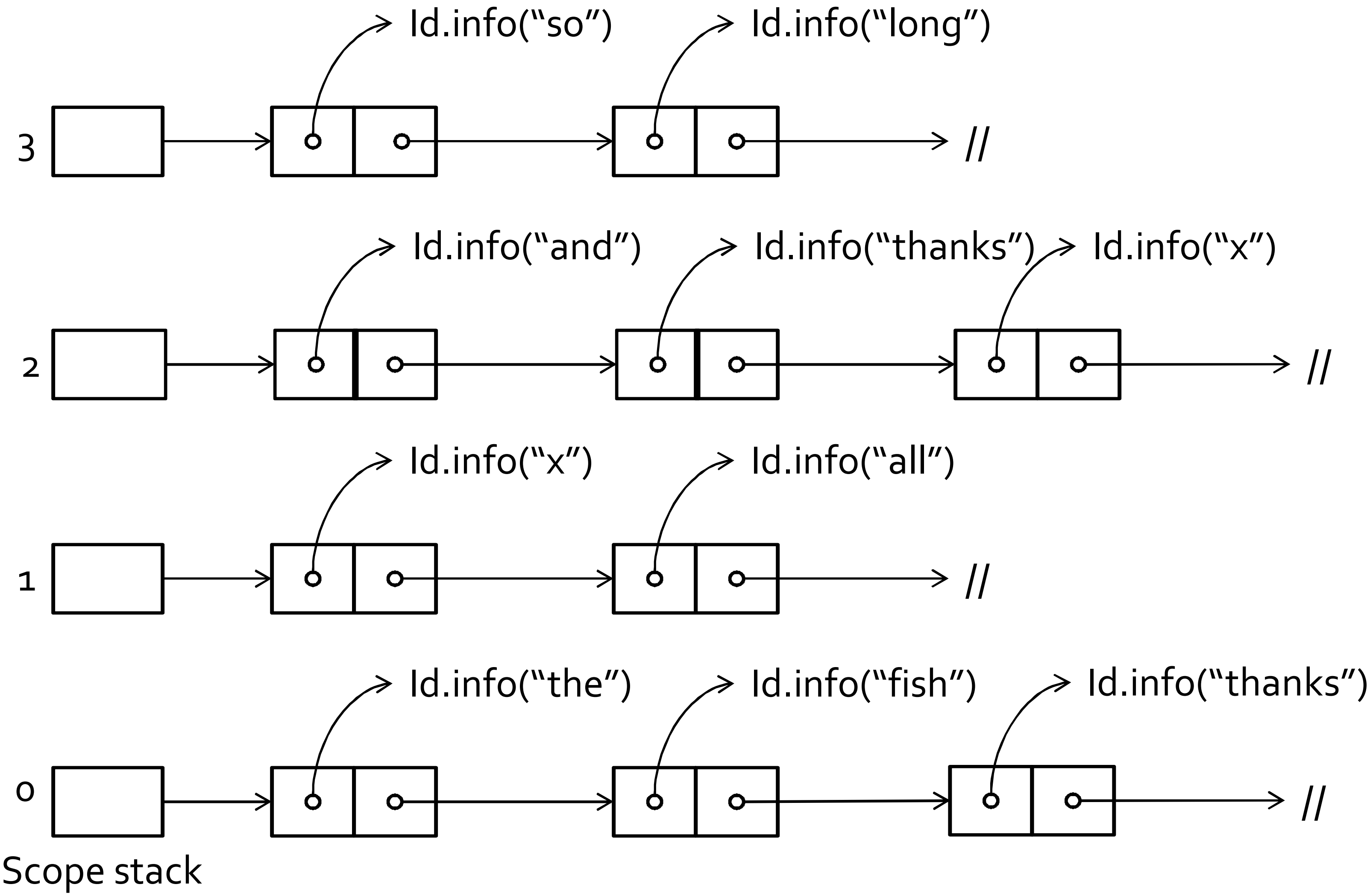
Scope and symbol table

- Scope x Identifier -> properties
  - Expensive lookup
- A better solution
  - hash table over identifiers

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?

Semantic Checks

- Scope rules
  - Use symbol table to check that
    - Identifiers defined before used
    - No multiple definition of same identifier
    - Program conforms to scope rules

- 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 System (textbook definition)

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute"

-- Types and Programming Languages
/ Benjamin C. Pierce

Type System

- A type system of a programming language is a way to define how "good" program behave
  - Good programs = well-typed programs
  - Bad programs = not well typed
- Type checking
  - Static typing – most checking at compile time
  - Dynamic typing – most checking at runtime
- Type inference
  - Automatically infer types for a program (or show that there is no valid typing)

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 as well typed
  - Why?

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

Type Checking

- Type 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 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 E1 has type int and E2 has type int,
then E1 + E2 has type int

$$\frac{E1 : int \qquad\qquad E2 : int}{E1 + E2 : int}$$

(Generally, also use a context A)

More Typing Rules (examples)

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

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

$$\frac{A \vdash E1 : \text{int} \qquad A \vdash E2 : \text{int}}{A \vdash E1 \; \textit{op} \; E2 : \text{int}} \qquad \textit{op} \in \{ +, -, /, *, \% \}$$

$$\frac{A \vdash E1 : \text{int} \qquad A \vdash E2 : \text{int}}{A \vdash E1 \; \textit{rop} \; E2 : \text{boolean}} \qquad \textit{rop} \in \{ <=, <, >, >= \}$$

$$\frac{A \vdash E1 : T \qquad A \vdash E2 : T}{A \vdash E1 \; \textit{rop} \; E2 : \text{boolean}} \qquad \textit{rop} \in \{ ==, != \}$$

And Even More Typing Rules

$$\frac{A \vdash E1 : \text{boolean} \qquad A \vdash E2 : \text{boolean}}{A \vdash E1 \ \textit{lop}\ E2 : \text{boolean}} \qquad \textit{lop} \in \{\ \&\&, ||\ \}$$

$$\frac{A \vdash E1 : \text{int}}{A \vdash \mathbf{-}\ E1 : \text{int}} \qquad \frac{A \vdash E1 : \text{boolean}}{A \vdash \mathbf{!}\ E1 : \text{boolean}}$$

$$\frac{A \vdash E1 : T[]}{A \vdash E1.\text{length} : \text{int}} \qquad \frac{A \vdash E1 : T[] \qquad A \vdash E2 : \text{int}}{A \vdash E1[E2] : T} \qquad \frac{A \vdash E1 : \text{int}}{A \vdash \text{new } T[E1] : T[]}$$

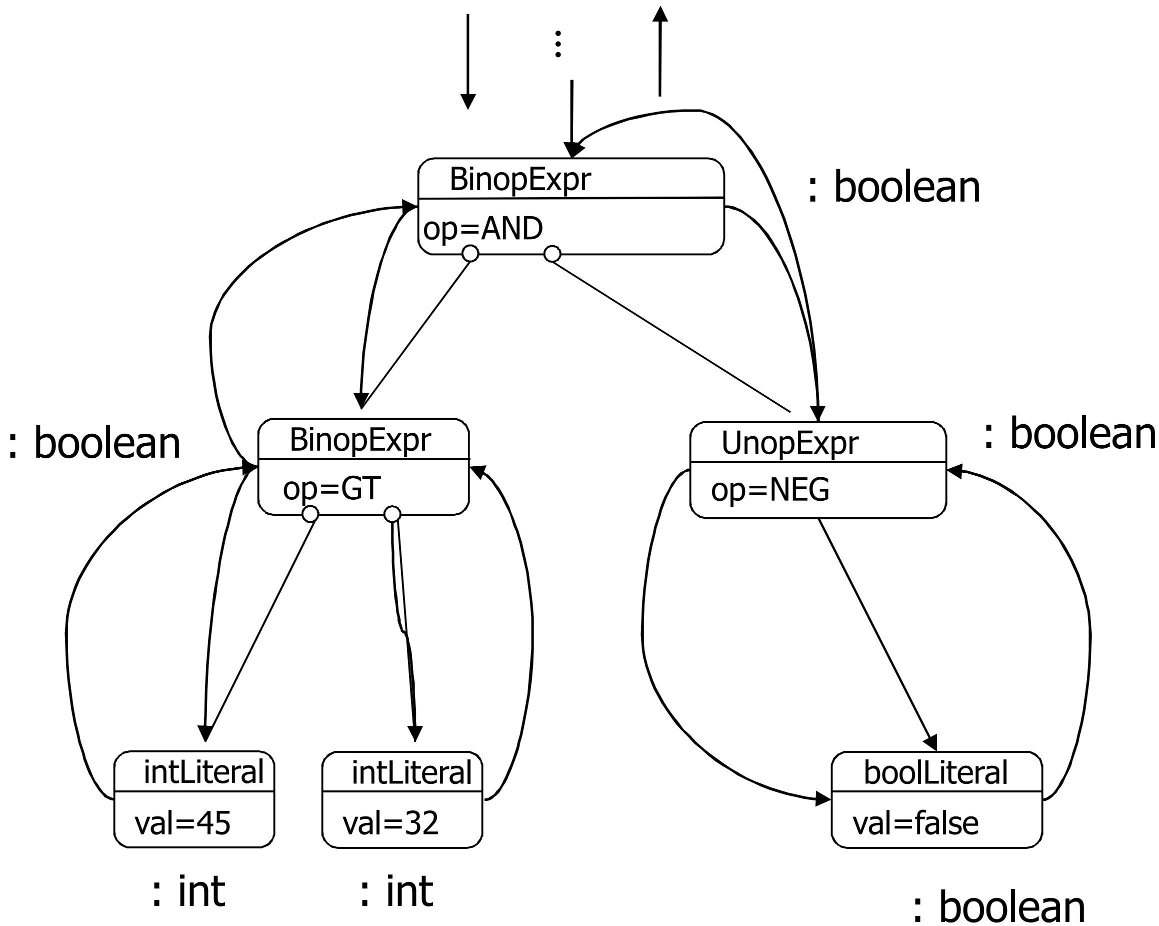
$$\frac{A \vdash T \text{ \textbackslash in } C}{A \vdash \text{new } T() : T} \qquad \frac{\text{id} : T \in A}{A \vdash \text{id} : T}$$

Type Checking

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

- Alternative: type-check during parsing
  - More complicated alternative
  - But naturally also more efficient

Example



$$\frac{A \vdash E1 : \text{boolean} \quad A \vdash E2 : \text{boolean}}{A \vdash E1 \; \textit{lop} \; E2 : \text{boolean}}$$

$\textit{lop} \in \{ \&\&, || \}$

$$\frac{A \vdash E1 : \text{boolean}}{A \vdash !E1 : \text{boolean}}$$

$$\frac{A \vdash E1 : \text{int} \quad A \vdash E2 : \text{int}}{A \vdash E1 \; \textit{rop} \; E2 : \text{boolean}}$$

$\textit{rop} \in \{ <=, <, >, >= \}$

$$\frac{}{A \vdash \text{false} : \text{boolean}}$$

$$\frac{}{A \vdash \textit{int-literal} : \text{int}}$$

45 > 32 && !false

Type Declarations

- So far, we ignored the fact that types can also be declared

TYPE Int_Array = ARRAY [Integer 1..42] OF Integer;   (explicitly)

Var a : ARRAY [Integer 1..42] OF Real;   (anonymously)

Type Declarations

Var a : ARRAY [Integer 1..42] OF Real;

TYPE #type01_in_line_73 = ARRAY [Integer 1..42] OF Real;
Var a : #type01_in_line_73;

Forward References

```
TYPE Ptr_List_Entry = POINTER TO List_Entry;
TYPE List_Entry =
  RECORD
    Element : Integer;
    Next : Ptr_List_Entry;
  END RECORD;
```

- Forward references must be resolved
  - A forward references added to the symbol table as forward reference, and later updated when type declaration is met
  - At the end of scope, must check that all forward references have been resolved
  - Check must be added for circularity

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., types of record fields

Type Equivalence: Name Equivalence

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

t1 not (name) equivalence to t2

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

t3 equivalent to t4

Type Equivalence: Structural Equivalence

```
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 name equivalence
- why?

Coercions

- If we expect a value of type T1 at some point in the program, and find a value of type T2, is that acceptable?

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

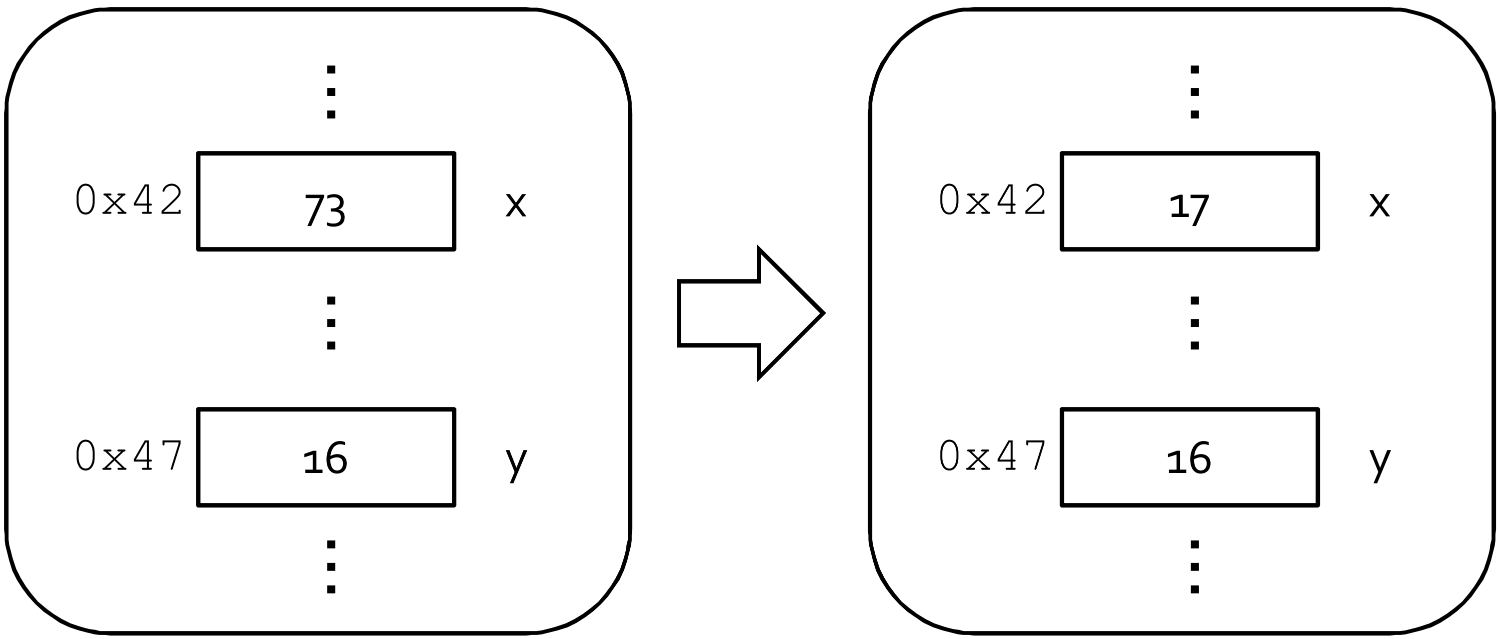
l-values and r-values

```
dst := src
```

- What is dst? What is src?
  - dst is a memory location where the value should be stored
  - src is a value
- "location" on the left of the assignment called an l-value
- "value" on the right of the assignment is called an r-value

l-values and r-values (example)

```
x:= y + 1
```



l-values and r-values (example)

```
x := A[1]

x := A[A[1]]
```

l-values and r-values (examples)

expression construct	**resulting kind**
constant	rvalue
identifier (variable)	lvalue
identifier (otherwise)	rvalue
&lvalue	rvalue
*rvalue	lvalue
V[rvalue]	V
V.selector	V
rvalue+rvalue	rvalue
lvalue := rvalue	rvalue

l-values and r-values

expected

found

	lvalue	**rvalue**
lvalue	-	deref
rvalue	error	-

So far…

- Static correctness checking
  - Identification
  - Type checking
- Identification matches applied occurrences of identifier to its defining occurrence
- 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)

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
  - (already mentioned when we did recursive descent)
  - How to update the attributes

- Attribute grammars

Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type
- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      Expr $\rightarrow$ Expr + Term
      Expr.type = Expr1.type when (Expr1.type == Term.type)
      Error otherwise

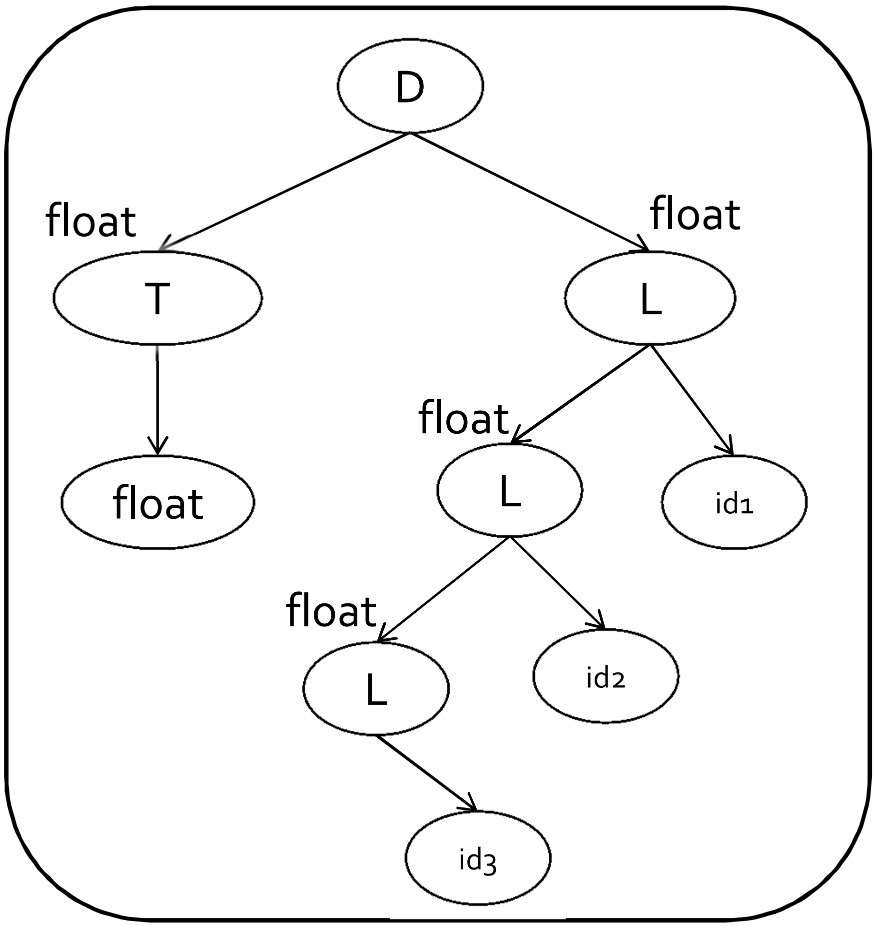
Indexed symbols

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

- Expr $\longrightarrow$ Expr + Term
  Becomes
  Expr $\longrightarrow$ Expr_1 + Term

Example

float x,y,z



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.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

Dependencies

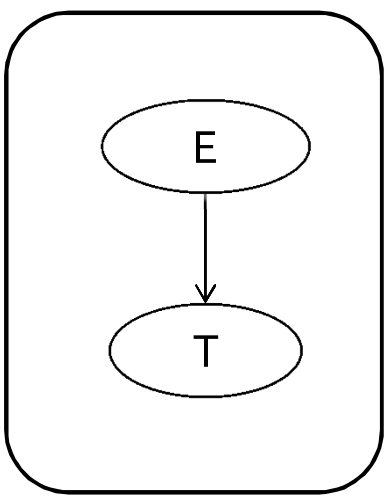
- A semantic equation $a = b_1,\ldots,b_m$ requires computation of $b_1,\ldots,b_m$ to determine the value of a
- The value of a depends on $b_1,\ldots,b_m$
  - We write $a \leftarrow b_i$

Attribute Evaluation

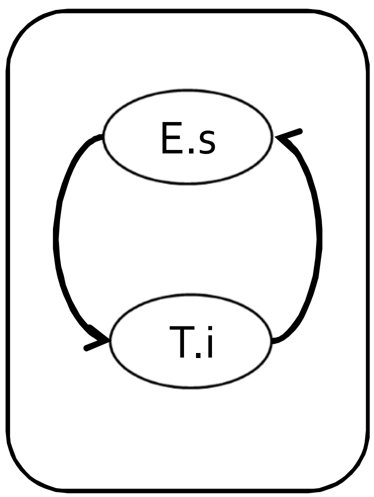
- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once

Cycles

- Cycle in the dependence graph
- May not be able to compute attribute values



AST



Dependence graph

E.S = T.i
T.i = E.s + 1

Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering

- Works as long as there are no cycles

Building Dependency Graph

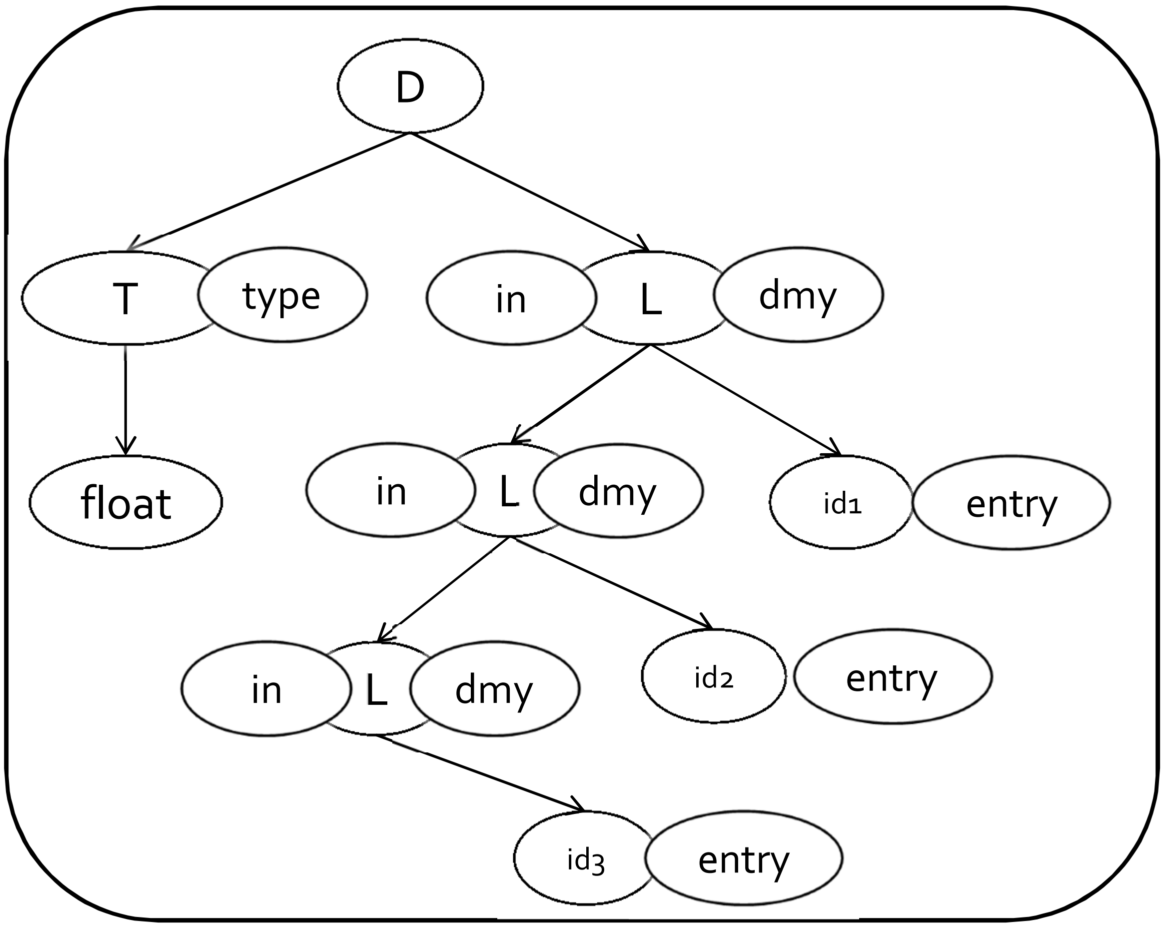
- All semantic equations take the form

  attr1 = func1(attr1.1, attr1.2,…)
  attr2 = func2(attr2.1, attr2.2,…)

- Actions with side effects use a dummy attribute
- Build a directed dependency graph G
  - For every attribute a of a node n in the AST create a node n.a
  - For every node n in the AST and a semantic action of the form b = f(c1,c2,…ck) add edges of the form (ci,b)

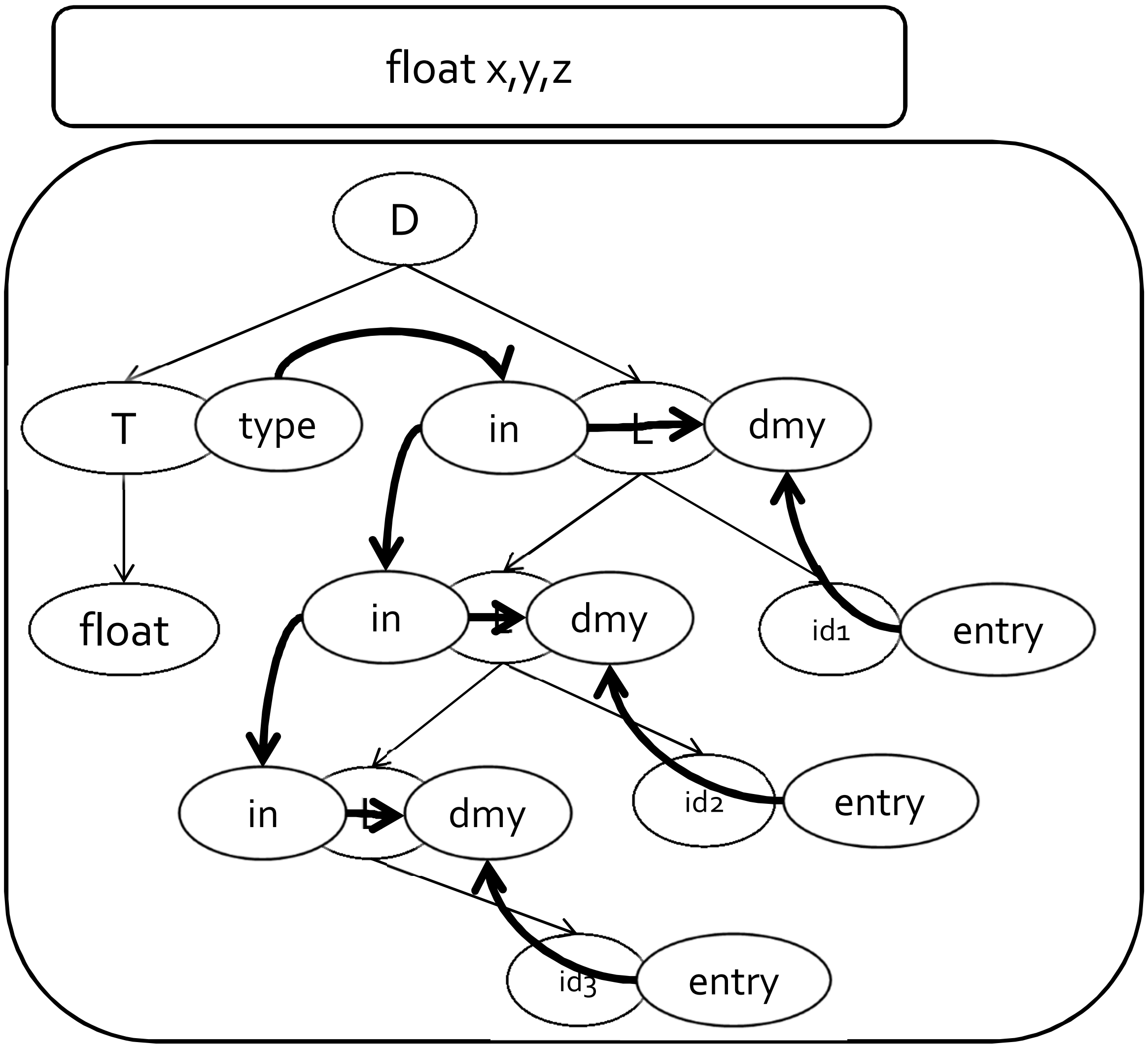
Example

float x,y,z



Prod.	**Semantic Rule**
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

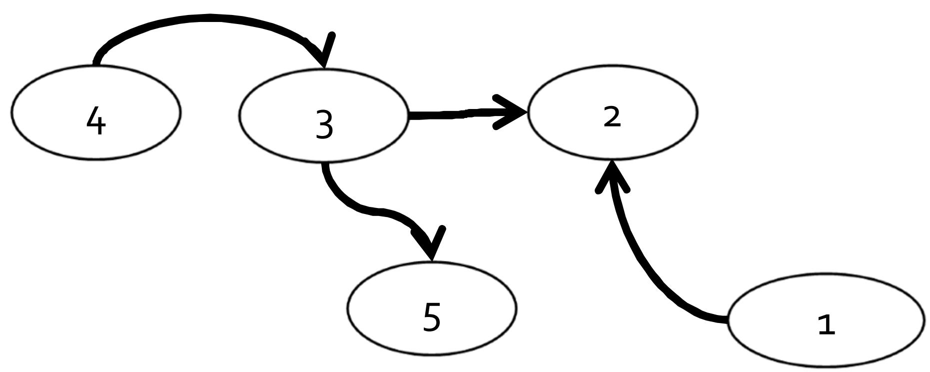
Example



Prod.	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)

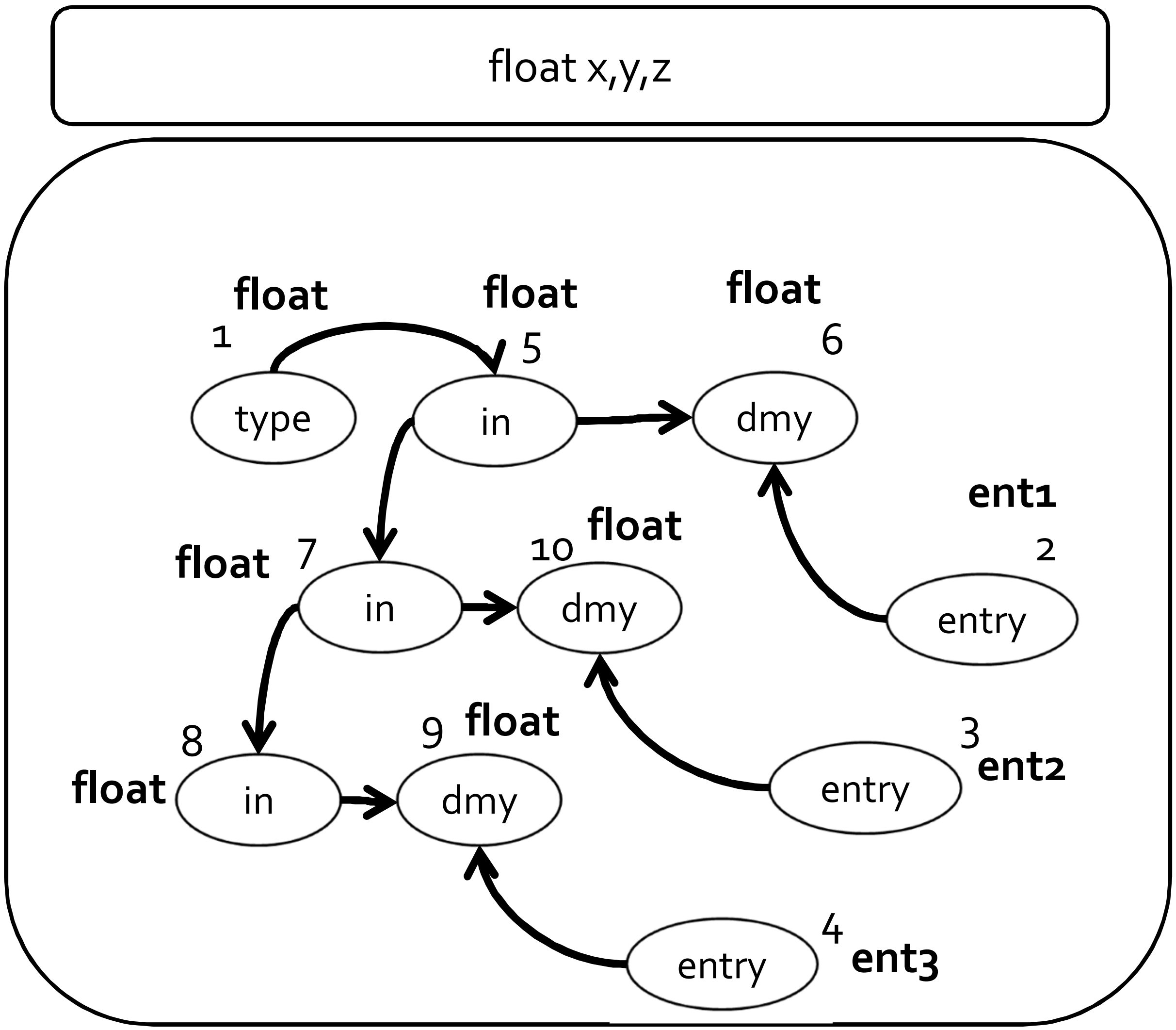
Topological Order

- For a graph G=(V,E), |V|=k
- Ordering of the nodes v1,v2,…vk such that for every edge (vi,vj) ∈ E, i < j



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

Example


float x,y,z
float
1
type
float
5
in
float
6
dmy
ent1
2
entry
float
7
in
10
float
dmy
3
entry
ent2
8
float
in
9
float
dmy
4
entry
ent3

But what about cycles?

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

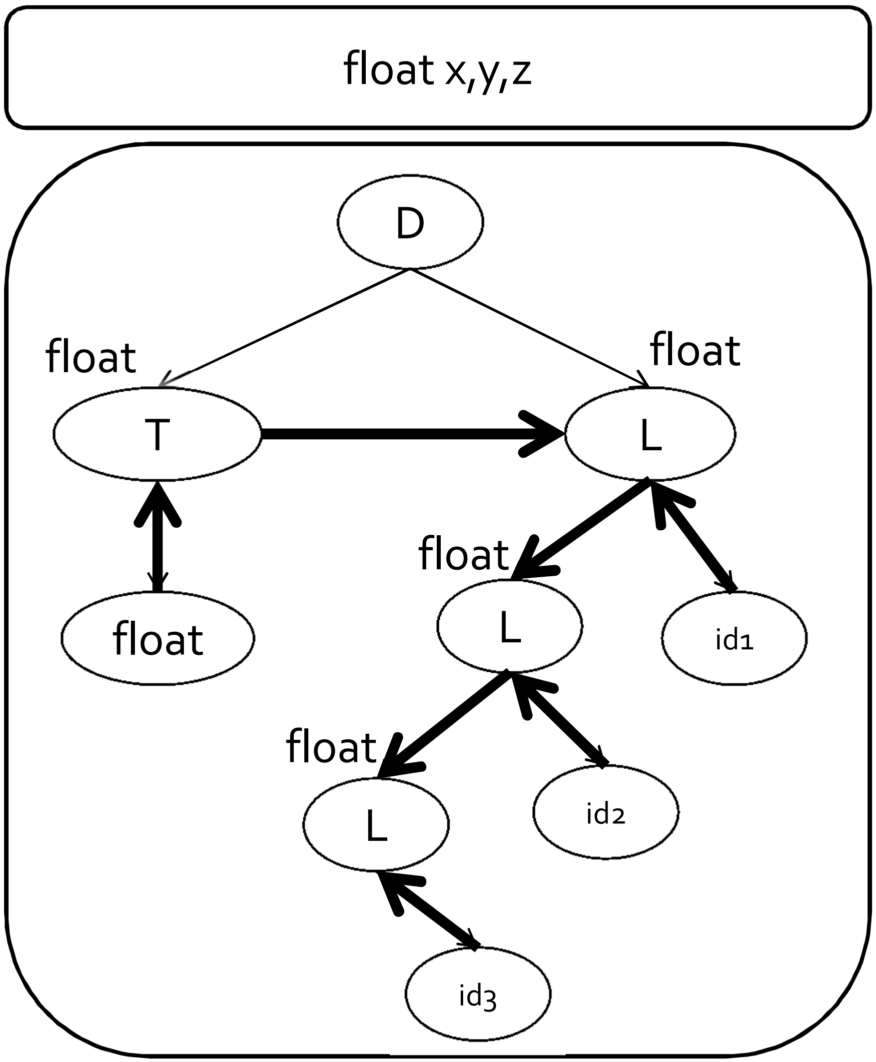
- Special classes of attribute grammars
  - Our “usual trick”
  - sacrifice generality for predictable performance

Inherited vs. Synthesized Attributes

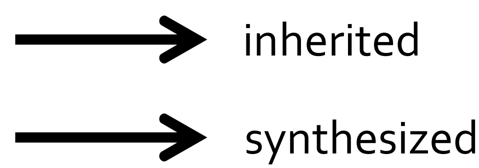
- Synthesized attributes
  - Computed from children of a node
- Inherited attributes
  - Computed from parents and siblings of a node

- Attributes of tokens are technically considered as synthesized attributes

example



Production	Semantic Rule
D → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
L → L1, id	L1.in = L.in addType(id.entry,L.in)
L → id	addType(id.entry,L.in)



S-attributed Grammars

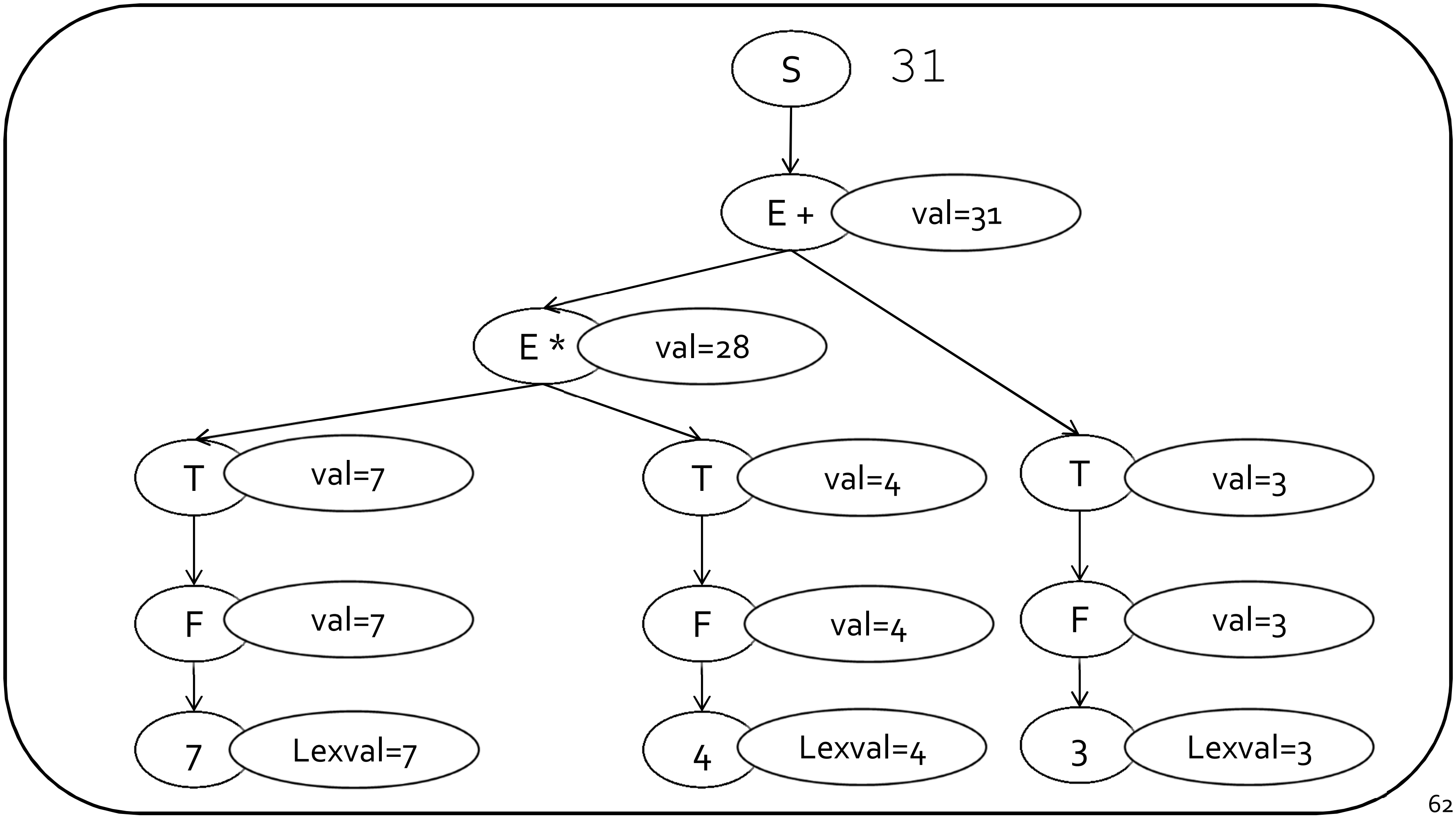
- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes

- Can be computed by any bottom-up parser during parsing
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children

S-attributed Grammar: example

Production	Semantic Rule
$S \longrightarrow E$;	print(E.val)
$E \longrightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \longrightarrow T$	$E.val = T.val$
$T \longrightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \longrightarrow F$	$T.val = F.val$
$F \longrightarrow (E)$	$F.val = E.val$
$F \longrightarrow$ digit	F.val = digit.lexval

example



S
31
E +
val=31
E *
val=28
T
val=7
T
val=4
T
val=3
F
val=7
F
val=4
F
val=3
7
Lexval=7
4
Lexval=4
3
Lexval=3

L-attributed grammars

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

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

The End