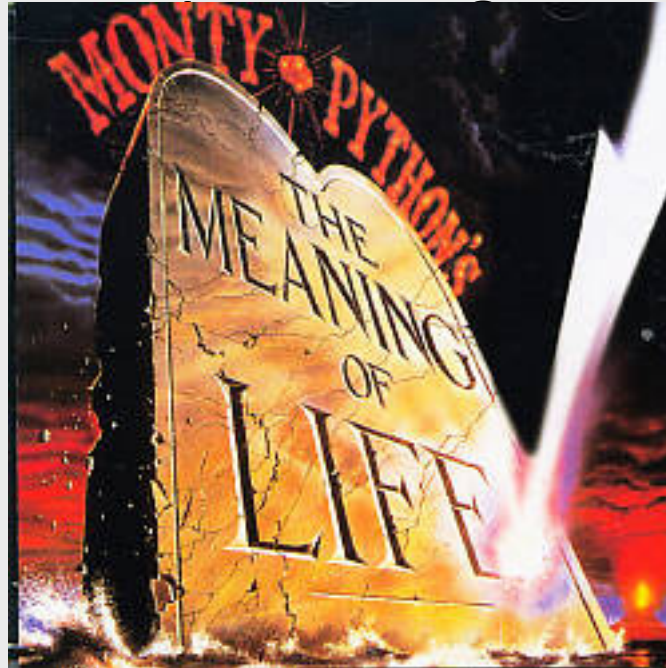


Compilation

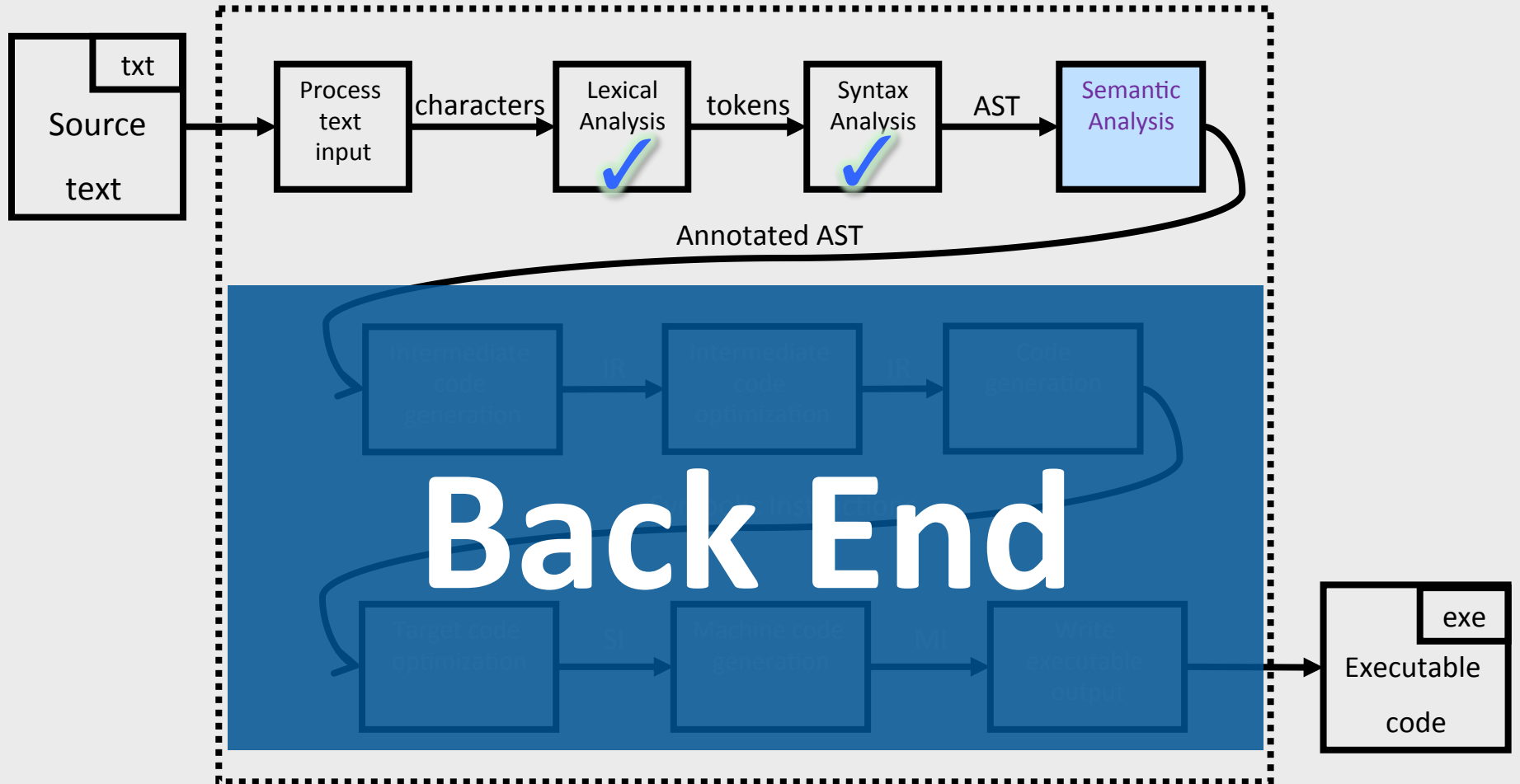
0368-3133 2014/15a



Semantic Analysis

Noam Rinetzky

You are here...



Oops

- `int x; xx = 0;`
- `int x, y; z = x + 1;`
- `main() { break; }`
- `int x; print(x)`



Can the parser help?

0 or 1 – this is the question

```
int x = 0;
```

```
p() { print(x) }
```

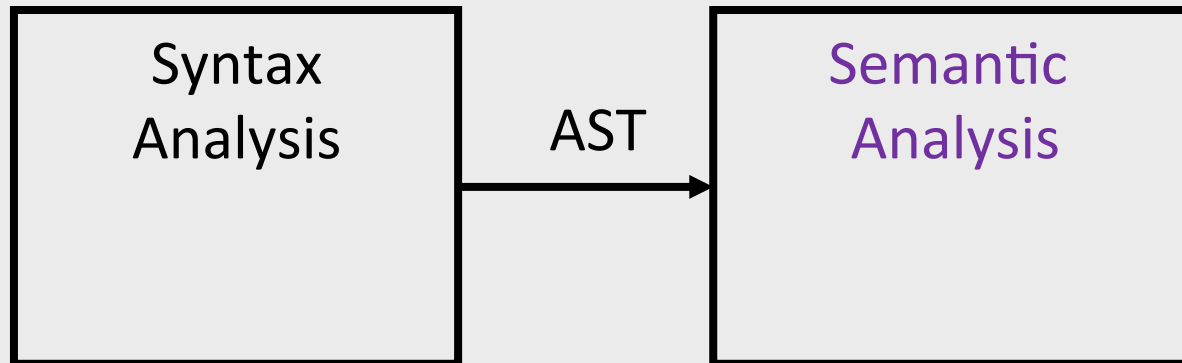
```
q() { int x = 1; p() }
```

Can the parser help?



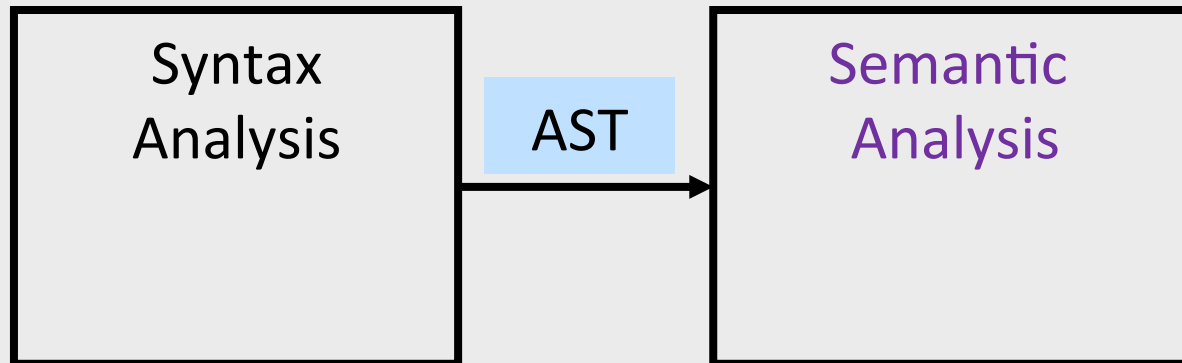
We want help ...

- Semantic (Context) analysis to the rescue



We want help ...

- Semantic (Context) analysis to the rescue



Abstract Syntax Tree

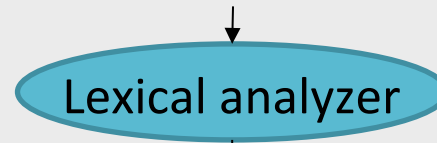
- AST is a simplification of the parse tree
- Can be built by traversing the parse tree
 - E.g., using visitors
- Can be built directly during parsing
 - Add an action to perform on each production rule
 - Similarly to the way a parse tree is constructed

Abstract Syntax Tree

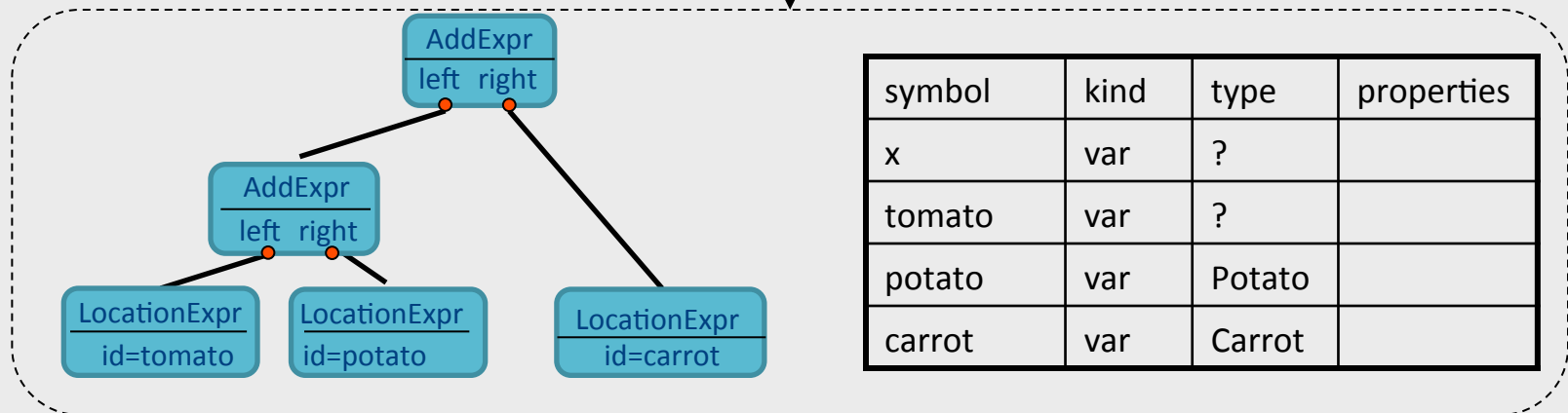
- The interface between the parser and the rest of the compiler
 - Separation of concerns
 - Reusable, modular and extensible
- The AST is defined by a context free grammar
 - The grammar of the AST can be ambiguous!
 - $E \rightarrow E + E$
 - Is this a problem?
- Keep syntactic information
 - Why?

What we want

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



...<id,tomato>,<PLUS>,<id,potato>,<PLUS>,<id,carrot>,EOF



'tomato' is undefined

'potato' used before initialized

Cannot add Potato and Carrot

Context Analysis

- Check properties contexts of in which constructs occur
 - 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

Context 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;  
}
```

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;
```

A struct field named i

A struct variable named i

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

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

Reading the "i" field of struct "i"

Not so fast...

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

A struct field named i

A struct variable named i

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

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

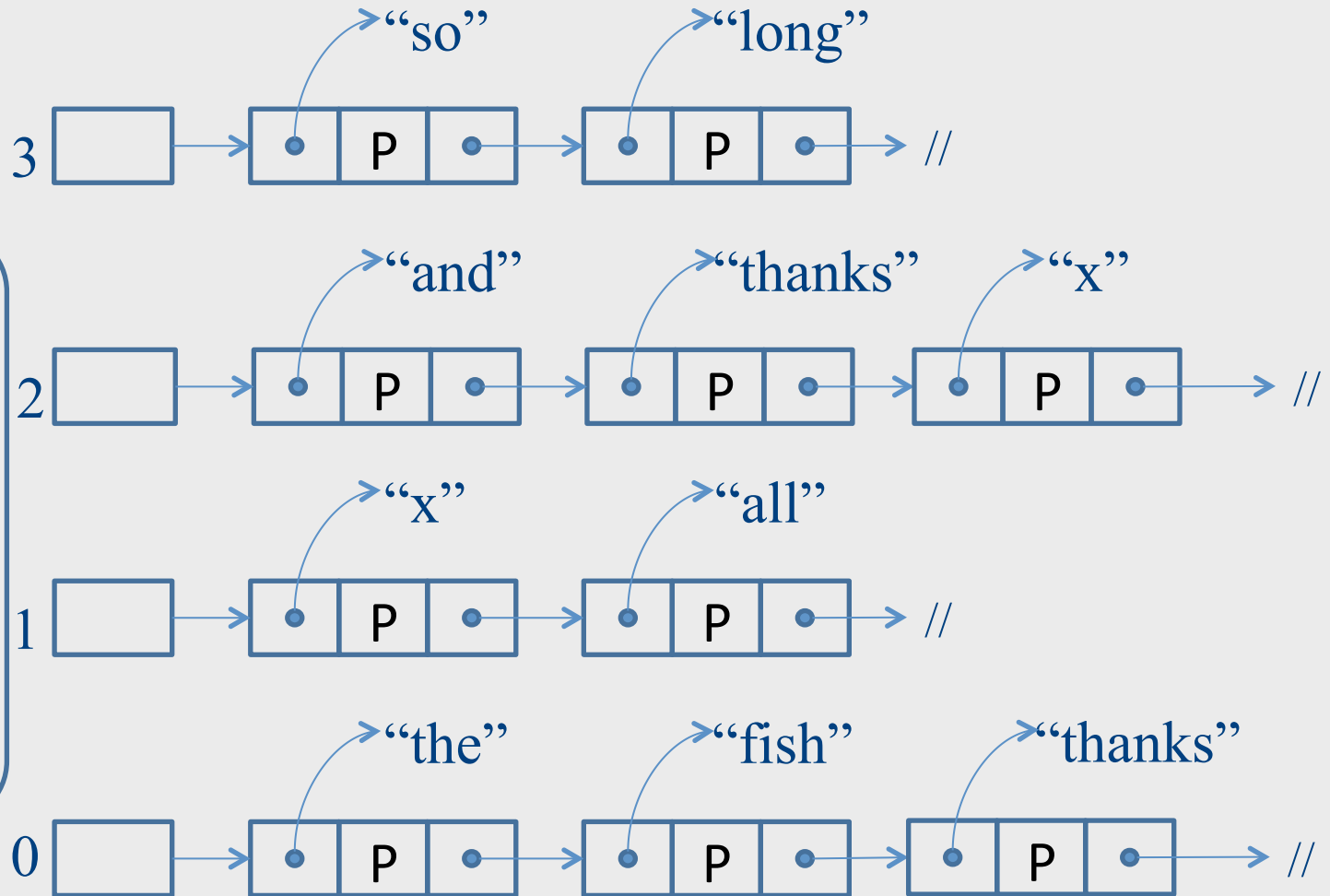
Reading the "i" field of struct "i"

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



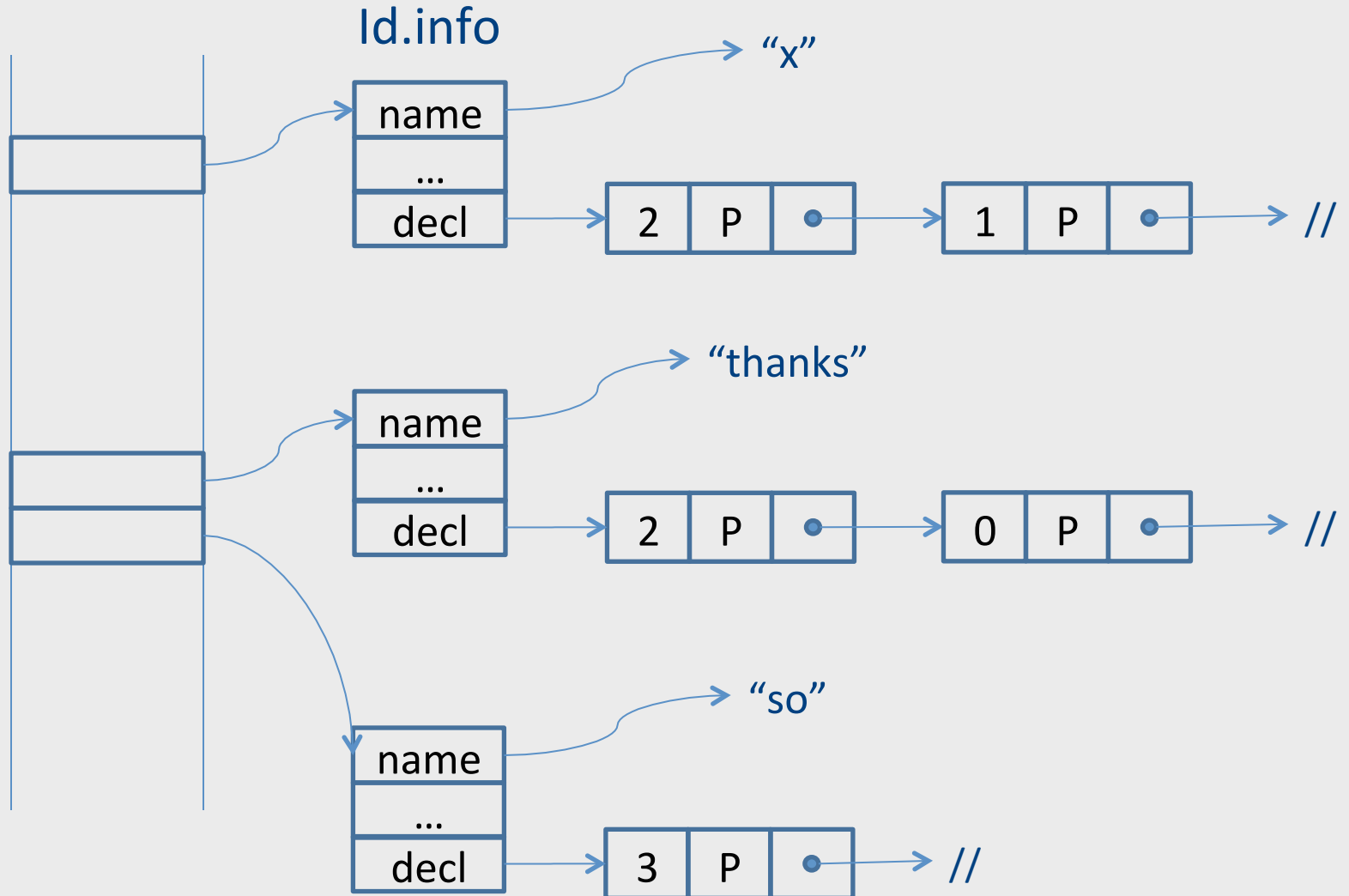
```
{  
  int the=1;  
  int fish=2;  
  int thanks=3;  
  {  
    int x = 42;  
    int all = 73;  
    {  
      ...  
    }  
  }  
}
```

Scope stack

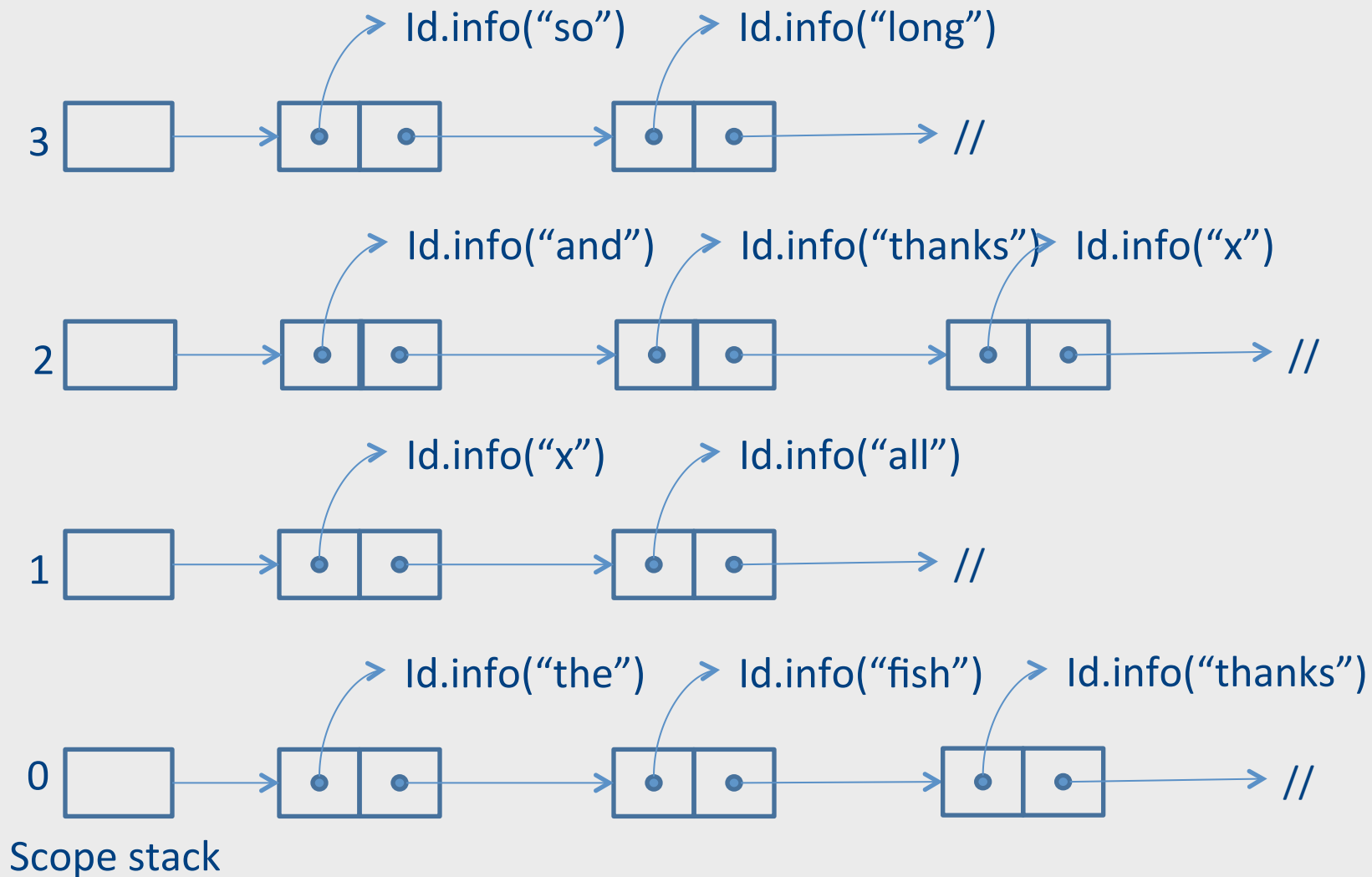
Scope and symbol table

- 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)

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

Semantic Checks

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

Types

- What is a type?
 - Simplest answer: a set of values + allowed operations
 - Integers, real numbers, booleans, ...
- Why do we care?
 - Code generation: $\$1 := \$1 + \$2$
 - 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
by 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
 - Why?

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

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

More Typing Rules (examples)

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

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

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

$\frac{}{\textit{string-literal} : \text{string}}$

$\frac{E1 : \text{int} \quad E2 : \text{int}}{E1 \textit{ op} E2 : \text{int}}$

$\textit{op} \in \{ +, -, /, *, \% \}$

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

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

$\frac{E1 : T \quad E2 : T}{E1 \textit{ rop} E2 : \text{boolean}}$

$\textit{rop} \in \{ ==, != \}$

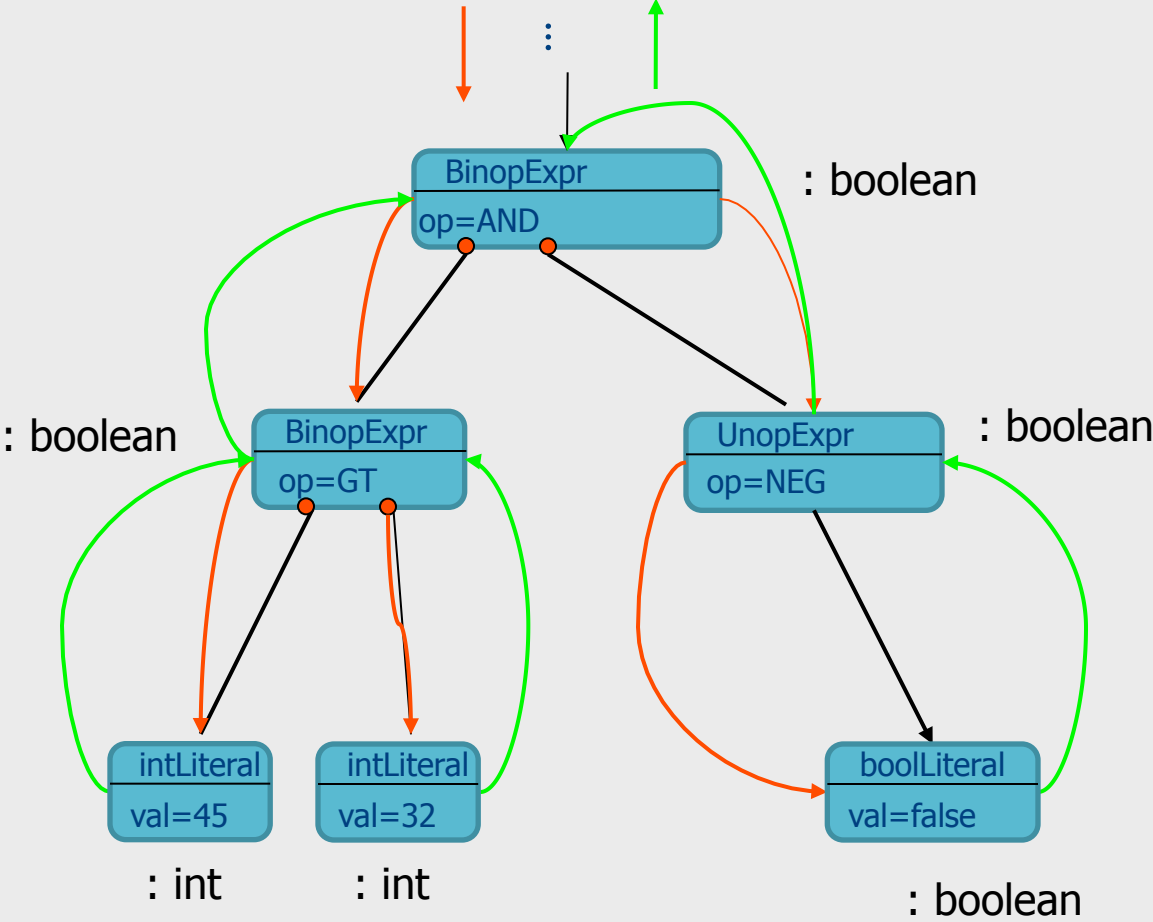
And Even More Typing Rules

$$\frac{E1 : \text{boolean} \quad E2 : \text{boolean}}{E1 \text{ } \textit{lop} \text{ } E2 : \text{boolean}} \quad \textit{lop} \in \{ \&\&, || \}$$
$$\frac{E1 : \text{int}}{- E1 : \text{int}}$$
$$\frac{E1 : \text{boolean}}{! E1 : \text{boolean}}$$
$$\frac{E1 : T[]}{E1.\text{length} : \text{int}}$$
$$\frac{E1 : T[] \quad E2 : \text{int}}{E1[E2] : T}$$
$$\frac{E1 : \text{int}}{\text{new } T[E1] : T[]}$$

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



45 > 32 && !false

$E1 : \text{boolean}$	$E2 : \text{boolean}$
<hr/>	
$E1 \text{ } lop \text{ } E2 : \text{boolean}$	
$lop \in \{ \&\&, \}$	
<hr/>	
$E1 : \text{boolean}$	
<hr/>	
$!E1 : \text{boolean}$	
<hr/>	
$E1 : \text{int}$	$E2 : \text{int}$
<hr/>	
$E1 \text{ } rop \text{ } E2 : \text{boolean}$	
$rop \in \{ <=, <, >, >= \}$	
<hr/>	
$\text{false} : \text{boolean}$	
<hr/>	
$\text{int-literal} : \text{int}$	

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

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;
```

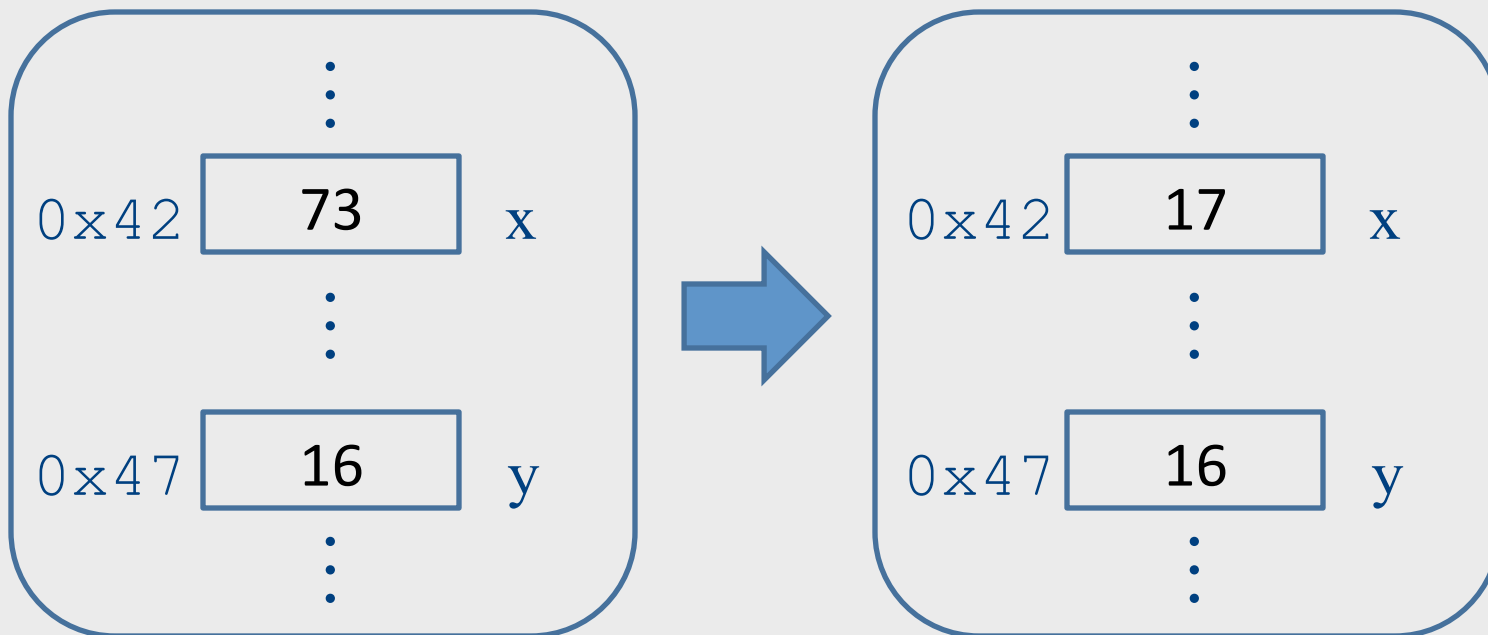
I-values and r-values

```
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

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
 - 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)

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
 - How to update the attributes
- Attribute grammars

Attribute grammars

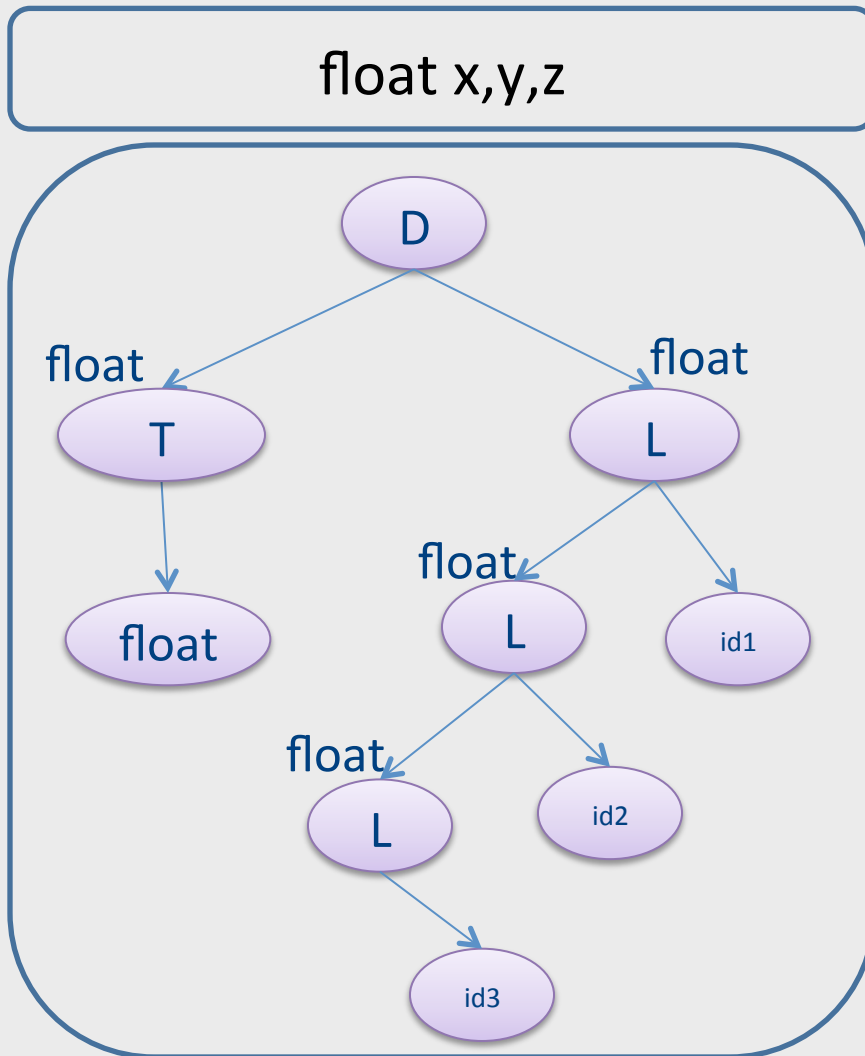
- Attributes
 - Every grammar symbol has attached attributes
 - Example: Expr.type
- Semantic actions
 - Every production rule can define how to assign values to attributes
 - Example:
Expr \rightarrow Expr + Term
Expr.type = Expr1.type when (Expr1.type == Term.type)
Error otherwise

Indexed symbols

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

- $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
Becomes
 $\text{Expr} \rightarrow \text{Expr}1 + \text{Term}$

Example



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

Attribute Evaluation

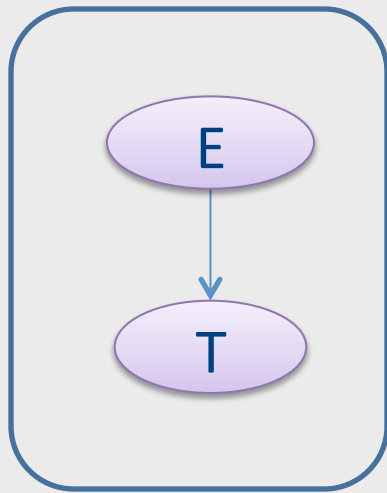
- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
 - In the right order such that
 - No attribute value is used before its available
 - Each attribute will get a value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

Dependencies

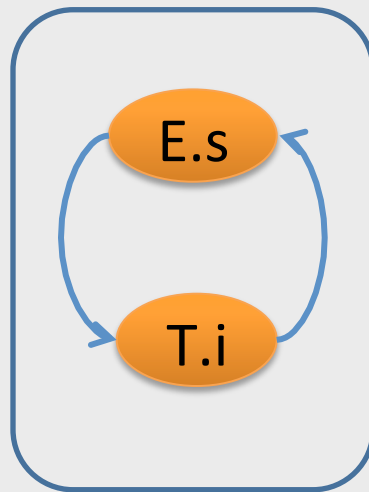
- A semantic equation $a = b_1, \dots, b_m$ requires computation of b_1, \dots, b_m to determine the value of a
- The value of a depends on b_1, \dots, b_m
 - We write $a \leftarrow b_i$

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

$\text{attr1} = \text{func1}(\text{attr1.1}, \text{attr1.2}, \dots)$

$\text{attr2} = \text{func2}(\text{attr2.1}, \text{attr2.2}, \dots)$

- 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 $b = f(c_1, c_2, \dots, c_k)$ add edges of the form (c_i, b)

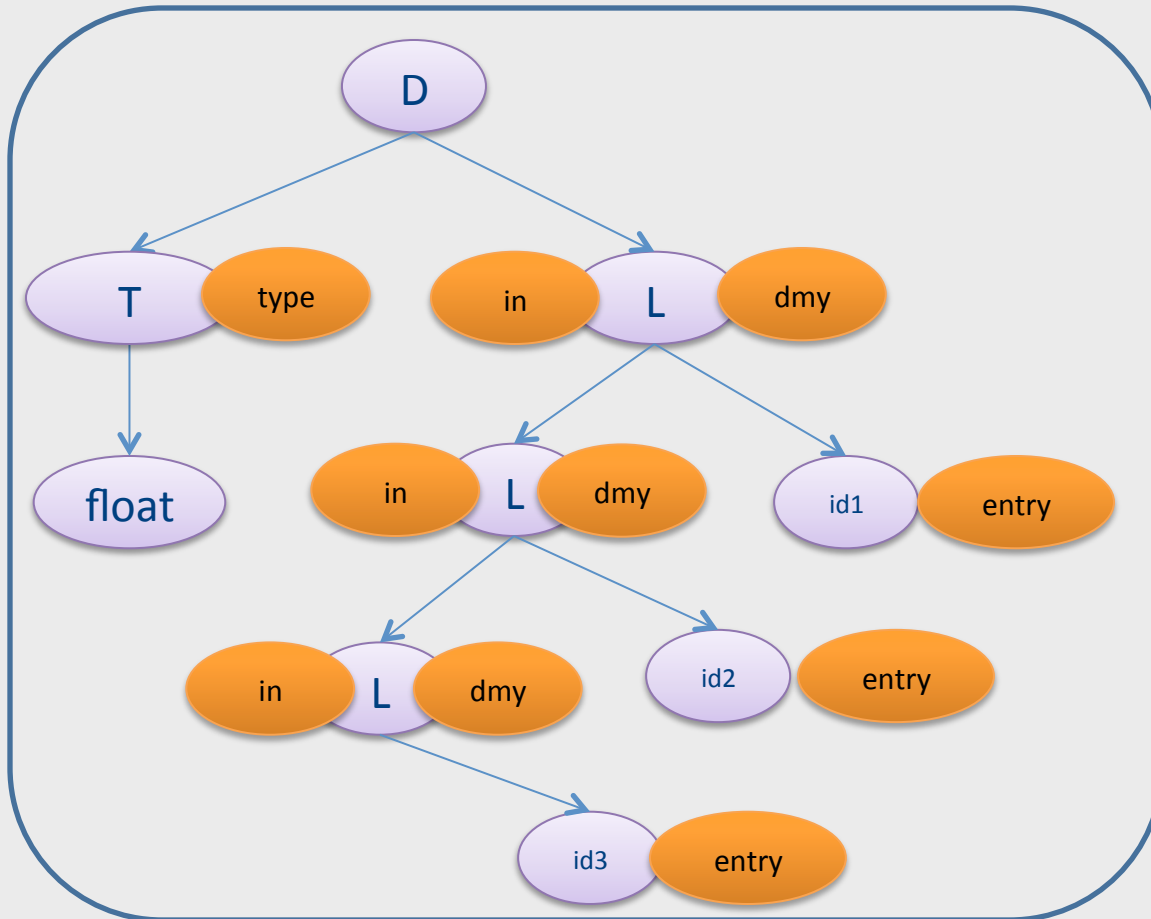
Production	Semantic Rule
$D \rightarrow T L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L1, id$	$L1.in = L.in$ $addType(id.entry, L.in)$
$L \rightarrow id$	$addType(id.entry, L.in)$

Convention:
Add dummy variables
for side effects.

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

Example

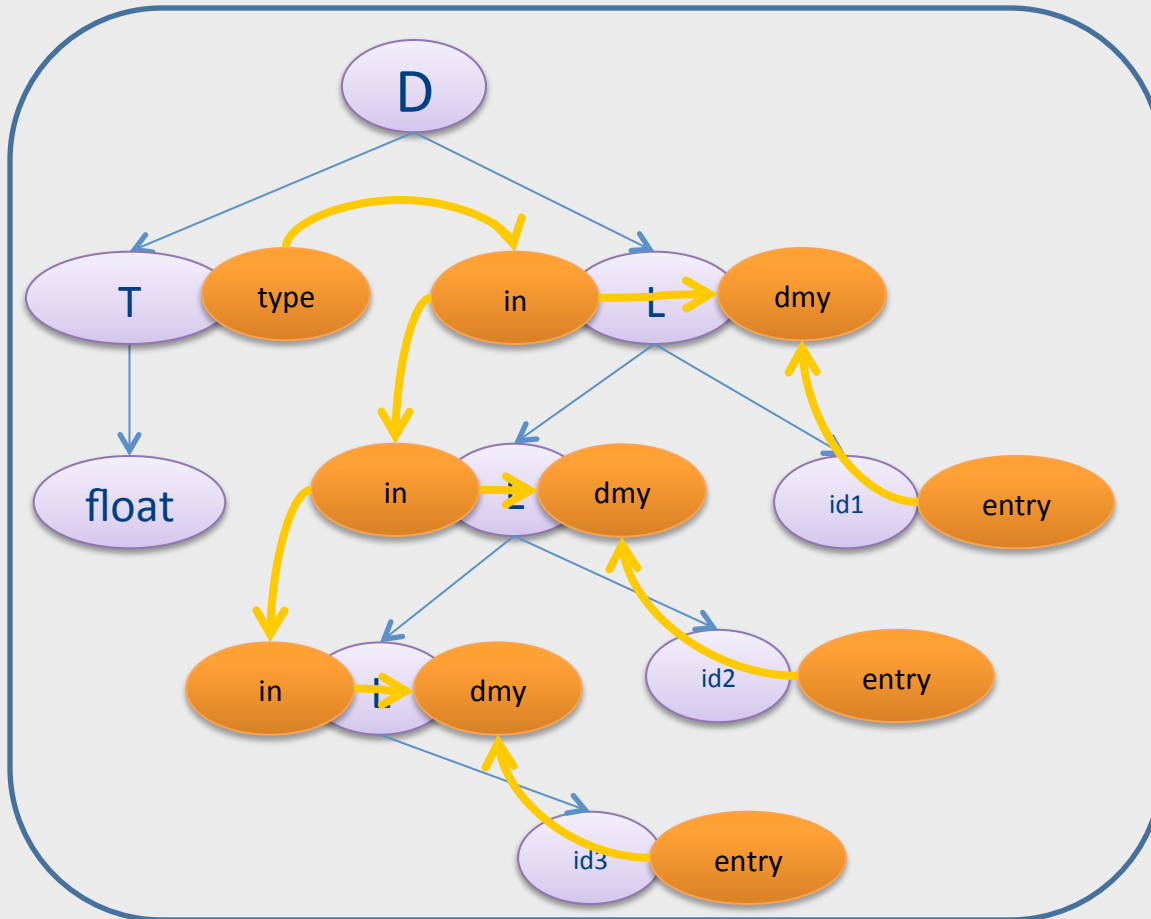
float x,y,z



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

Example

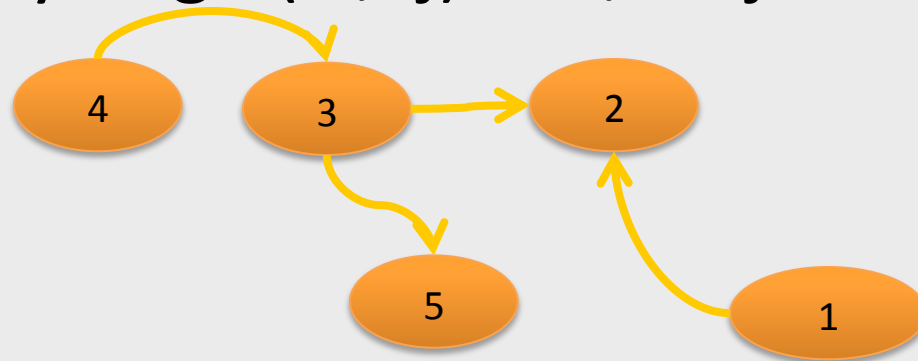
float x,y,z



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

Topological Order

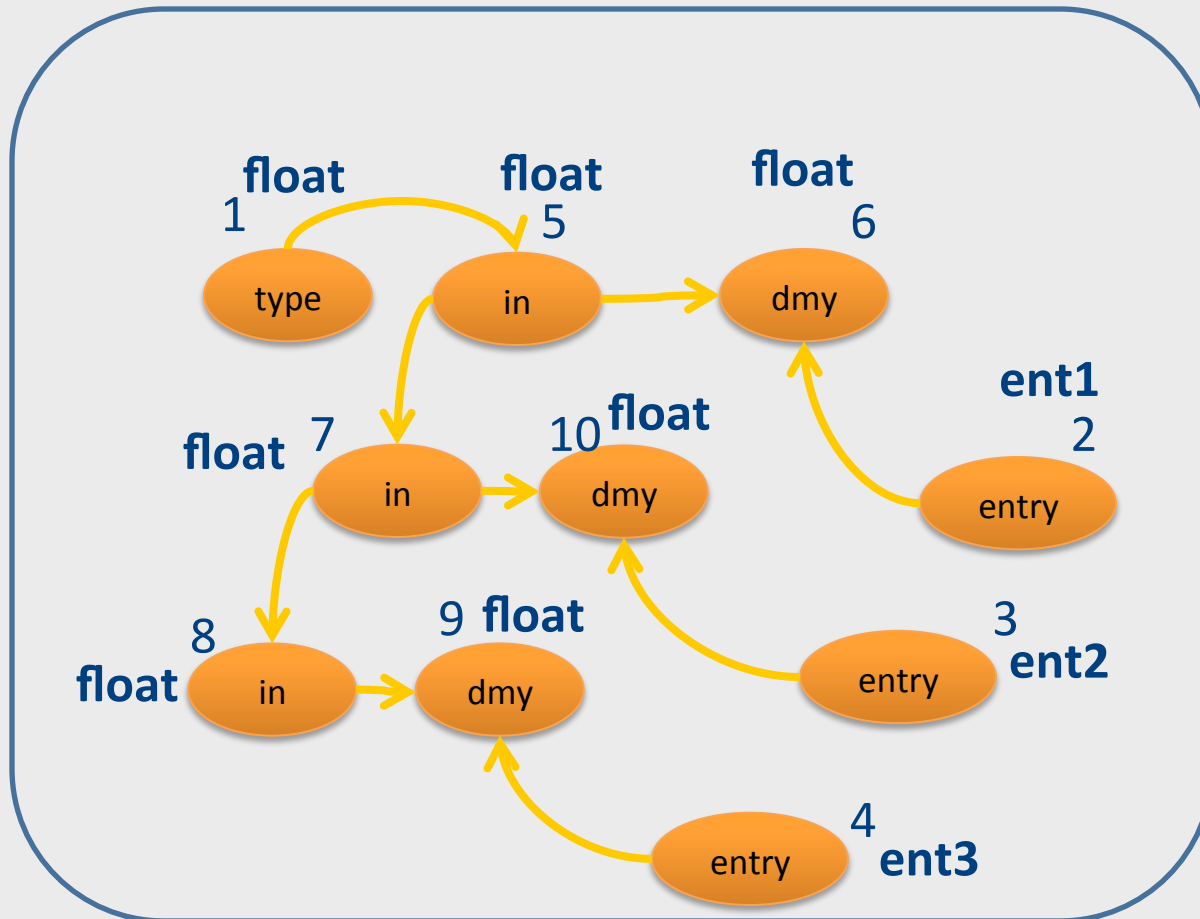
- For a graph $G=(V,E)$, $|V|=k$
- Ordering of the nodes v_1, v_2, \dots, v_k such that for every edge $(v_i, v_j) \in E$, $i < j$



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

Example

float x,y,z



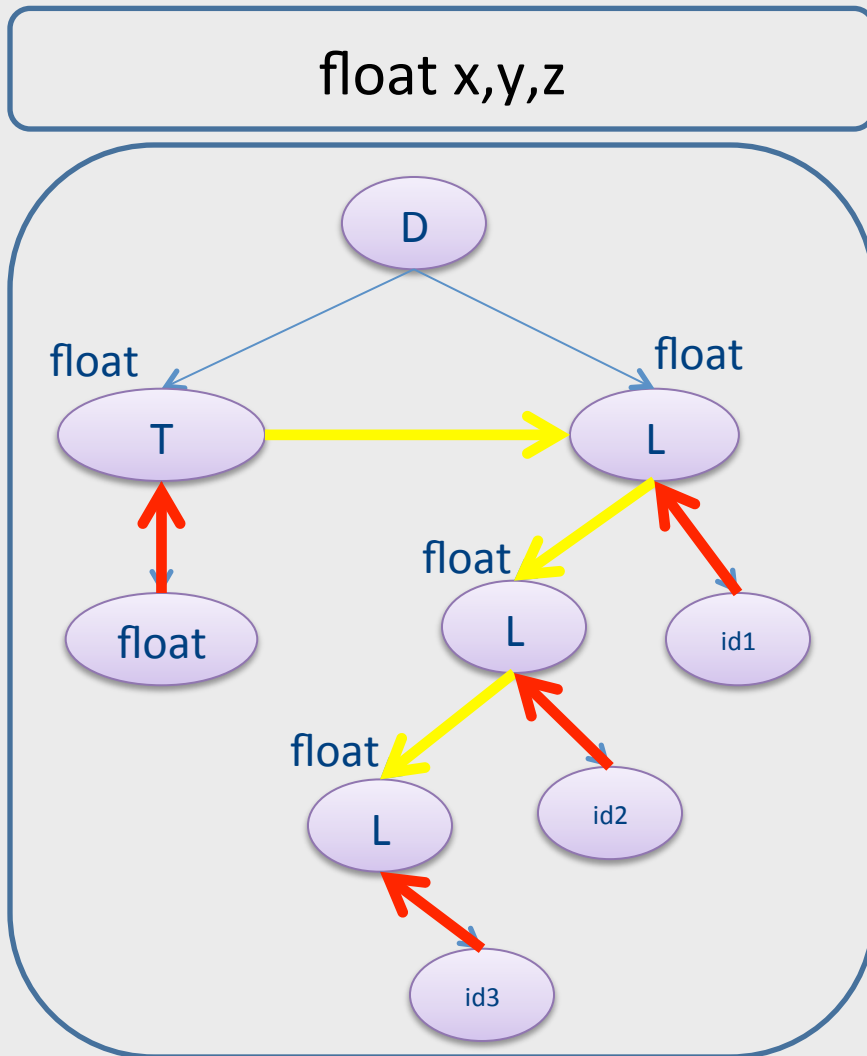
But what about cycles?

- 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
$D \rightarrow T L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L1, id$	$L1.in = L.in$ $addType(id.entry, L.in)$
$L \rightarrow id$	$addType(id.entry, L.in)$

→ inherited
→ synthesized

S-attributed Grammars

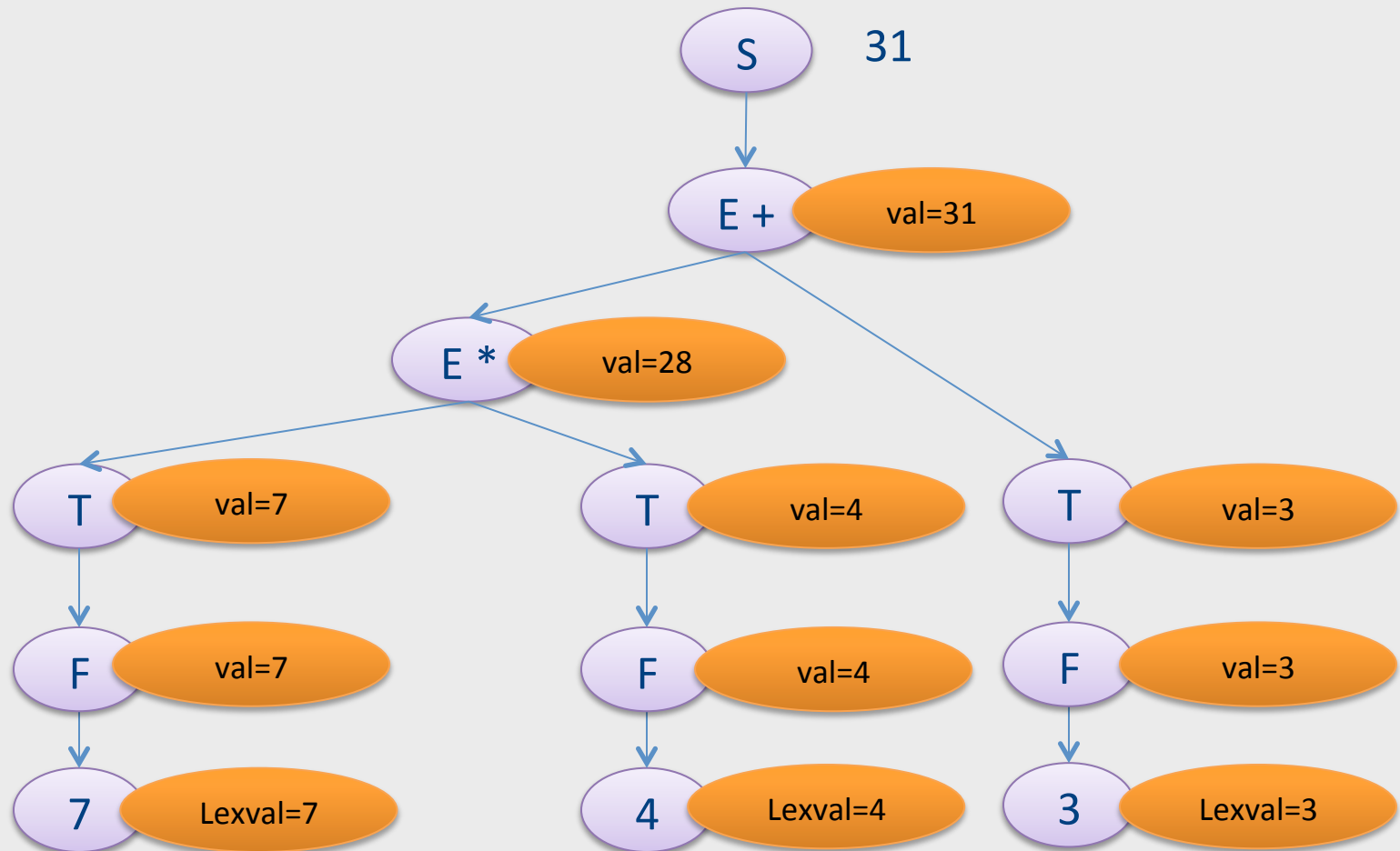
- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes

- 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
$S \rightarrow E ;$	<code>print(E.val)</code>
$E \rightarrow E1 + T$	<code>E.val = E1.val + T.val</code>
$E \rightarrow T$	<code>E.val = T.val</code>
$T \rightarrow T1 * F$	<code>T.val = T1.val * F.val</code>
$T \rightarrow F$	<code>T.val = F.val</code>
$F \rightarrow (E)$	<code>F.val = E.val</code>
$F \rightarrow \text{digit}$	<code>F.val = digit.lexval</code>

example



L-attributed grammars

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

Example: typesetting



- Each box is built from smaller boxes from which it gets the height and depth, and to which it sets the point size.
- pointsize (ps) – size of letters in a box. Subscript text has smaller point size of 0.7p.
- height (ht) – distance from top of the box to the baseline
- depth (dp) – distance from baseline to the bottom of the box.

Example: typesetting

production	semantic rules
$S \rightarrow B$	$B.ps = 10$
$B \rightarrow B1 B2$	$B1.ps = B.ps$ $B2.ps = B.ps$ $B.ht = \max(B1.ht, B2.ht)$ $B.dp = \max(B1.dp, B2.dp)$
$B \rightarrow B1 \text{ sub } B2$	$B1.ps = B.ps$ $B2.ps = 0.7 * B.ps$ $B.ht = \max(B1.ht, B2.ht - 0.25 * B.ps)$ $B.dp = \max(B1.dp, B2.dp - 0.25 * B.ps)$
$B \rightarrow \text{text}$	$B.ht = \text{getHt}(B.ps, \text{text.lexval})$ $B.dp = \text{getDp}(B.ps, \text{text.lexval})$

Computing the attributes from left to right during a DFS traversal

```
procedure dfvisit (n: node);  
begin  
  for each child m of n, from left to right  
    begin  
      evaluate inherited attributes of m;  
      dfvisit (m)  
    end;  
  evaluate synthesized attributes of n  
end
```

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

- Front-end