Brief Overview

Also see document from Education Board of ACM SIGPLAN (“Motivation” on web page)

Main Questions in PL

Q1: Is this a valid program?

Compile-time and run-time checking (in 6341: attribute grammars and type systems)

Q2: What is this program supposed to do?

Precise language semantics (in 6341: operational semantics)

Q3: How do we execute this program correctly and efficiently?

Implementation of compilers and interpreters (in 6341: projects to build an interpreter; attribute grammars for code generation in a compiler; static analysis for performance optimization)

Why Study Foundations of PL?

Understand your tools better

- Compilers, interpreters, virtual machines, code checking tools, debuggers, assemblers, linkers

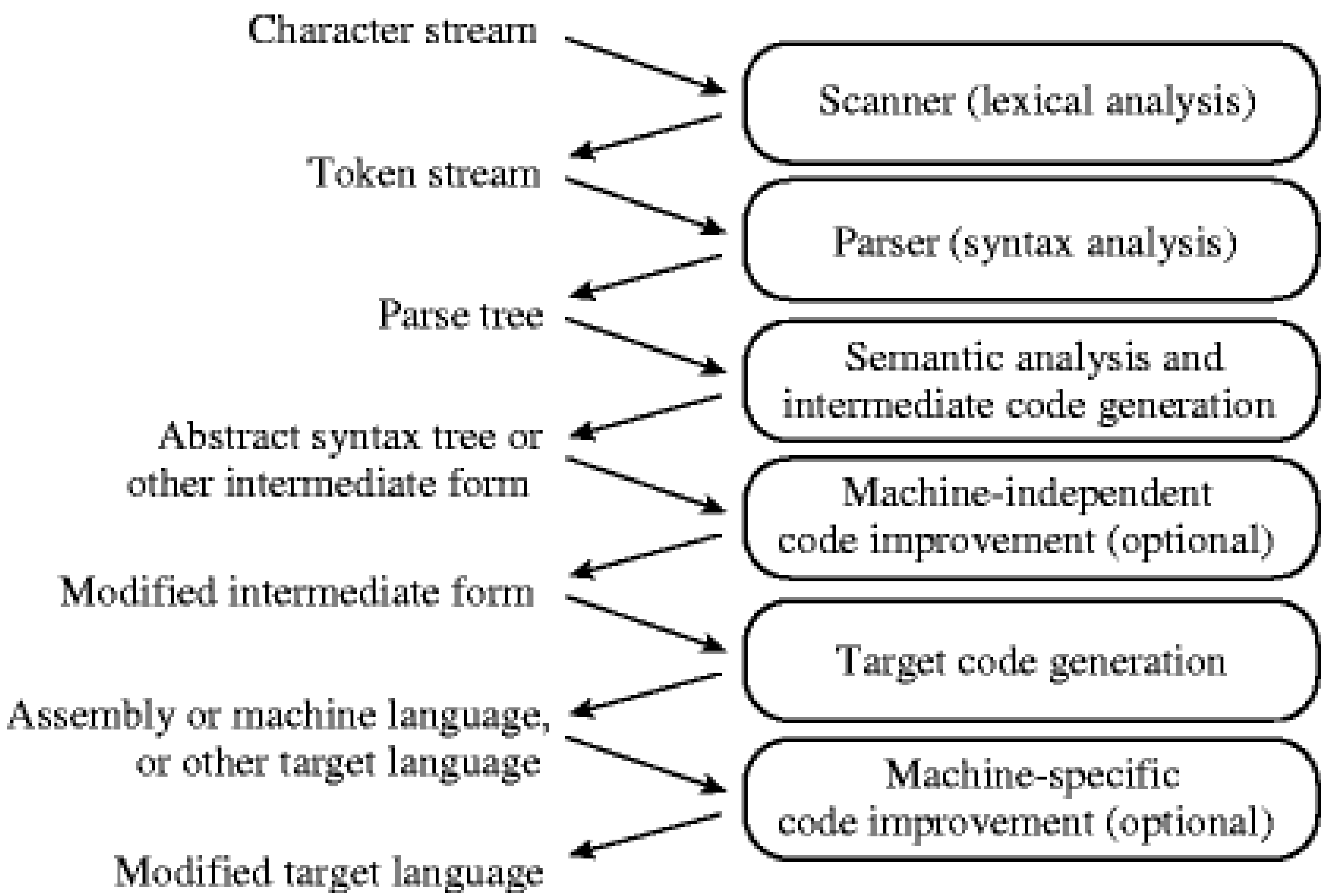
Write your own languages, compilers, analyzers, …

- Happens more often than you'd think [example: Forma]

To fix bugs & make programs fast, often you need to understand what's happening "under the hood"

Most importantly: PLs are the foundations of software; we need to be clear on what they mean and how to support their users with useful tools

Example: Inside a Compiler



Regular grammars & context-free grammars (expected background)

Attribute grammars, type checking, PL semantics

PL semantics

Attribute Grammars

Pagan Ch. 2.1, 2.2, 2.3, 3.2

Slonneger and Kurtz Ch 3.1, 3.2 [online; under Resources on course web page]

Dragon Book Ch. 5.1, 5.2

Outline

Review context-free grammars [expected background]

Introduce attribute grammars

Use scenario: simple type checking

Two flavors of attribute grammars: (1) pure and (2) with limited side effects

Use scenario: more complex type checking

Use scenario: generation of assembly code

Formal Languages

Theoretical basis for the design and implementation of programming languages

Alphabet: finite set **T** of symbols

String: finite sequence of symbols

- Empty string ε (i.e., sequence of length 0)
- **T*** **-** set of all strings over **T** (incl. ε)
- $\mathbf{T^+}$ **-** set of all non-empty strings over **T**

Language: set of strings $\mathbf{L} \subseteq \mathbf{T^*}$

Grammars

G = (N, T, S, P)

- Finite set of **non-terminal symbols** **N**
- Finite set of **terminal symbols** **T** (this is our alphabet)
- Starting non-terminal symbol **S** $\in$ **N**
- Finite set of **productions** **P**
- Goal: define a language **L** $\subseteq$ **T***

Production: **x** $\rightarrow$ **y**

- **x**: non-empty sequence of terminals and non-terminals
- **y**: possibly-empty sequence of terminals/non-terminals

Applying a production: u**x**v $\Rightarrow$ u**y**v

Languages and Grammars

Derivation of a string

$\mathbf{w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n}$; denoted $\mathbf{w_1 \overset{*}{\Rightarrow} w_n}$

Language generated by a grammar

$\mathbf{L(G) = \{ w \in T^* \mid S \overset{*}{\Rightarrow} w \}}$

Traditional classification of languages and grammars

Regular $\subset$ Context-free $\subset$ Context-sensitive $\subset$ Unrestricted

Use in Compilers and Interpreters

stream of characters

w,h,i,l,e,(,a,1,5,>,b,b,),d,o,…

Lexical Analyzer (uses a **regular** grammar)

stream of tokens

keyword[while],leftparen,id[a15],op[>], id[bb],rightparen,keyword[do], …

Parser (uses a **context-free** grammar)

parse tree

each token is a leaf in the parse tree

… more components

Context-Free Languages

Strict superset of regular languages

- Example: $L = \{ a^n b^n \mid n > 0 \}$ is context-free but not regular

Generated by a **context-free grammar**

- Each production: **A** $\rightarrow$ **w**
- **A** is a non-terminal, **w** is a (possibly empty) sequence of terminals and non-terminals

BNF: alternative notation for context-free grammars

- Backus-Naur form: John Backus and Peter Naur, for ALGOL60 (both have received the ACM Turing Award)

BNF Example (related to the language for the project)

<program> ::= <stmtList>

<stmtList> ::= <stmt> <stmtList>

| <stmt>

<stmt> ::= <varDecl> **=** <expr> **;**

| **ident =** <expr> **;**

<varDecl> ::= **int ident**

<expr> ::= **intconst**

| **ident**

| <expr> **+** <expr>

If there are several productions

<X> ::= …

for convenience we write them as a single production

<X> ::= … | … | …

We say "the i^{th} production alternative"

String Derivation

Example of a string from the language

[next slide shows the leftmost derivation sequence (always expands the leftmost non-terminal)]

[try this at home: the rightmost derivation sequence (always expands the rightmost non-terminal)]

int x = 1; y = x + 2;

Example of a string not from the language

x+1 = y;

<program> ::= <stmtList>

<stmtList> ::= <stmt> <stmtList> | <stmt>

<stmt> ::= <varDecl> **=** <expr> **;** | **ident =** <expr> **;**

<varDecl> ::= **int ident**

<expr> ::= **intconst** | **ident** | <expr> **+** <expr>

int x = 1 ; y = x + 2 ;

<program> => <stmtList> =>

<stmt> <stmtList> =>

<varDecl> **=** <expr> **;** <stmtList> =>

int ident$_x$ **=** <expr> **;** <stmtList> =>

int ident$_x$ **= intconst**$_1$ **;** <stmtList> =>

int ident$_x$ **= intconst**$_1$ **;** <stmt> =>

int ident$_x$ **= intconst**$_1$ **; ident**$_y$ **=** <expr> **;** =>

int ident$_x$ **= intconst**$_1$ **; ident**$_y$ **=** <expr> **+** <expr> **;** =>

int ident$_x$ **= intconst**$_1$ **; ident**$_y$ **= ident**$_x$ **+** <expr> **;** =>

int ident$_x$ **= intconst**$_1$ **; ident**$_y$ **= ident**$_x$ **+ intconst**$_2$ **;**

Parse Tree

Also called **derivation tree** or **concrete syntax tree**

- Leaf nodes: terminals
- Inner nodes: non-terminals
- Root: starting non-terminal of the grammar
- Leaf nodes, from left to right, define the string

Each non-leaf node X has children that correspond to some production <X> ::= …

- Children are ordered as they appear in the production

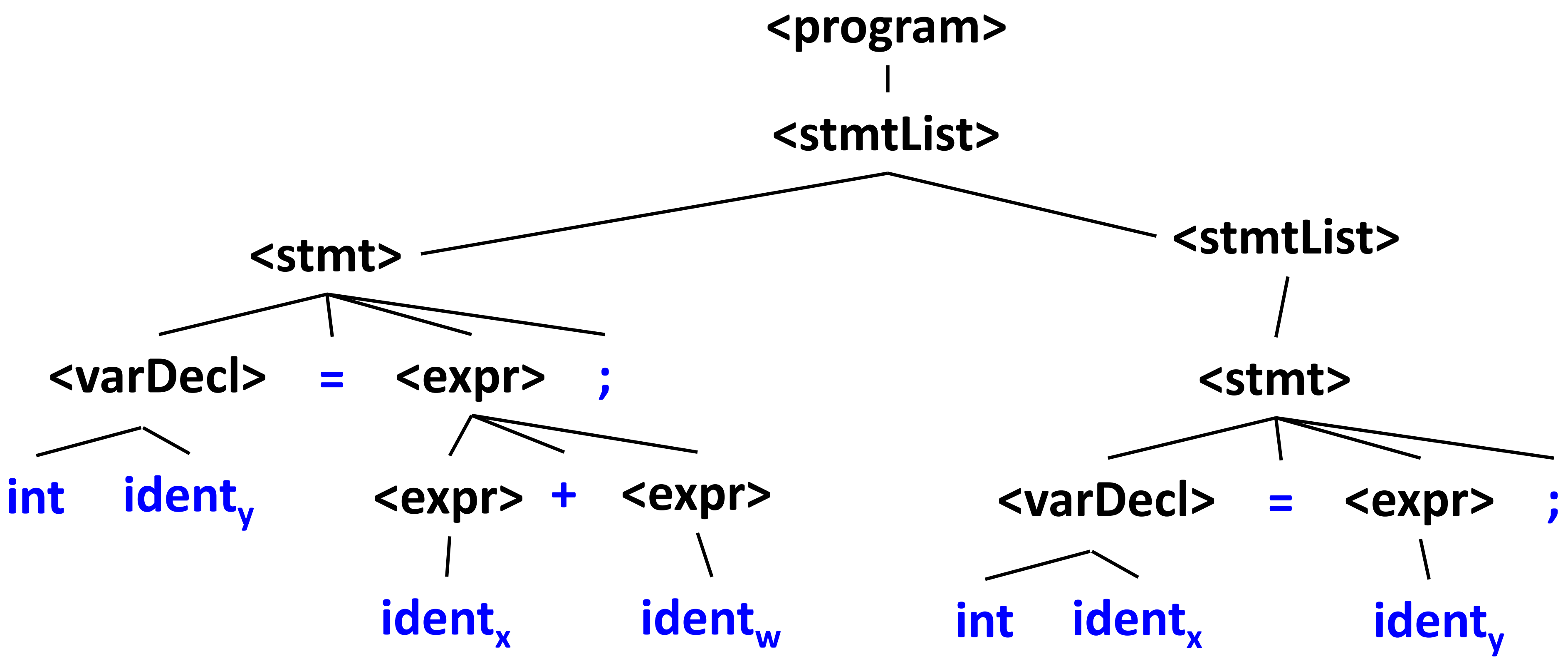
Parse Tree Examples

Example 1: **int x = 1; int y = x + 2;**

Example 2: **int y = x + w; int x = y;**

Example 3: **w = x + y + z;**

int y = x + w; int x = y;



<program> ::= <stmtList>

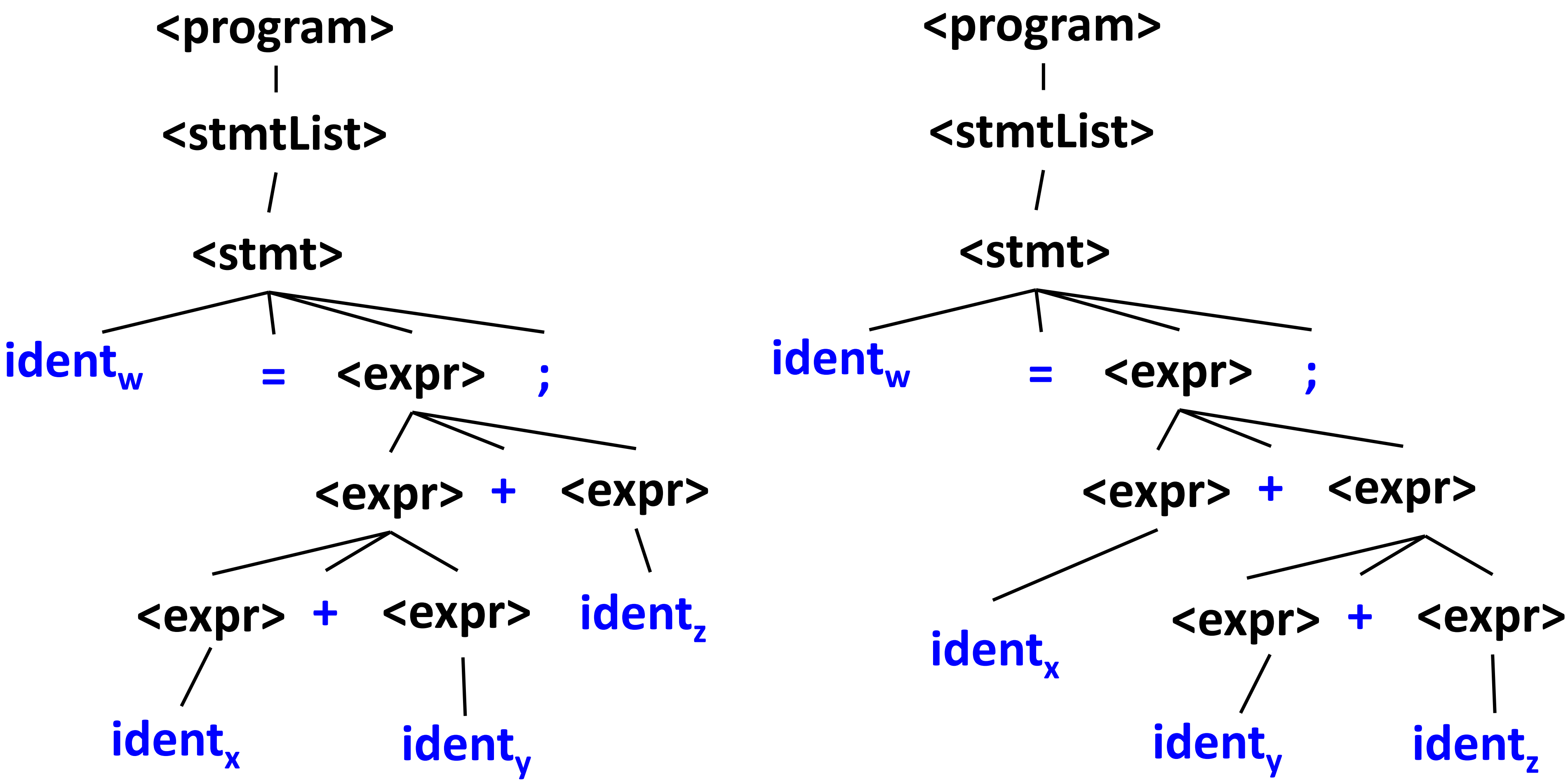
<stmtList> ::= <stmt> <stmtList> | <stmt>

<stmt> ::= <varDecl> **=** <expr> **;** | **ident =** <expr> **;**

<varDecl> ::= **int ident**

<expr> ::= **intconst** | **ident** | <expr> **+** <expr>

w = x + y + z;



<program> ::= <stmtList>

<stmtList> ::= <stmt> <stmtList> | <stmt>

<stmt> ::= <varDecl> **=** <expr> **;** | **ident =** <expr> **;**

<varDecl> ::= **int ident**

<expr> ::= **intconst** | **ident** | <expr> **+** <expr>

Ambiguous Grammar

For some string, there are multiple parse trees

An **ambiguous** grammar

- Gives more freedom to the compiler writer: e.g., for code optimizations (several possible translations)
- Allows under-specification of irrelevant details [we will see this later when we discuss operational semantics and abstract interpretation]
- Must be disambiguated when we build a real parser

To **remove ambiguity**

- Change the grammar, or
- Keep it ambiguous, but tell the parser how to resolve ambiguity so that we have only one possible parse tree [this is the approach used in the programming projects]

Classic Examples of Ambiguity

<expr> ::= <expr> **+** <expr> | **ident**

Two different parse trees for **x + y + z**

<expr> ::= <expr> **+** <expr> | <expr> ***** <expr> | **ident**

Two different parse trees for **x + y * z**

We will illustrate the importance of ambiguity in one specific scenario: associativity and precedence of binary operators in programming languages

Binary Operators in Math

Commutativity: a **op** $b = b$ **op** a

Example: + is commutative and – is not commutative

Associativity [same **op***]:* $(a$ **op** $b)$ **op** $c = a$ **op** $(b$ **op** $c)$

+ is associative; we can write $a + b + c$ since the location of parentheses does not matter

– is not associative; for convenience, we write $a - b - c$ to mean $(a - b) - c$ [i.e., left-to-right reading of $a - b - c$]

Precedence [two different **op***s]:* $a\ \mathbf{op_1}\ b\ \mathbf{op_2}\ c$

Does it mean $(a\ \mathbf{op_1}\ b)\ \mathbf{op_2}\ c$ or is it $a\ \mathbf{op_1}\ (b\ \mathbf{op_2}\ c)$?

If the precedence is defined, we can omit parentheses

+ vs × : higher precedence for ×, so $a + b \times c$ is $a + (b \times c)$

Operator Associativity in PL

Associativity [same **op***]:* how should *a* **op** *b* **op** *c* be evaluated when we execute the program?

- Left-associative operator: *a* **op** *b* **op** *c* should be evaluated as "first compute the value of the left subexpression *a* **op** *b*; then compute the result **op** *c*" [i.e., treat it as (*a* **op** *b*) **op** *c*]
- Right-associative operator: *a* **op** *b* **op** *c* should be evaluated as "first compute the value of the right subexpression *b* **op** *c*; then compute *a* **op** the result" [i.e., treat it as *a* **op** (*b* **op** *c*)]

Why Does Ambiguity Matter?

<expr> ::= <expr> **+** <expr> | **ident**

w = x + y + z;

Parse tree version 1 leads to assembly code version 1

ADD R1, x, y
ADD R2, R1, z
STORE w, R2

Parse tree version 2 leads to assembly code version 2

ADD R1, y, z
ADD R2, x, R1
STORE w, R2

Left or right associativity? Same as asking "how should **x + y + z** be parsed"? It does matter …

// non-associative math operations
int p = 1 - 2 - 3;

SUB R1, 1, 2		**SUB R1, 2, 3**
SUB R2, R1, 3	vs	**SUB R2, 1, R1**
STORE p, R2		**STORE p, R2**

// floating-point computations

```
double x = (0.1 + 0.2) + 0.3;
double y = 0.1 + (0.2 + 0.3);
System.out.println(x==y); // what will be printed here?
```

Why Does Ambiguity Matter?

<expr> ::= <expr> **+** <expr> | <expr> ***** <expr> | **ident**

w = x + y * z;

Assembly code version 1

ADD R1, x, y
MUL R2, R1, z
STORE w, R2

Assembly code version 2

MUL R1, y, z
ADD R2, x, R1
STORE w, R2

Precedence: how should **x + y * z** be parsed? The shape of the parse tree matters …

Exercise: how many different parse trees are possible for **x * y + z * w**

In C++

<table>
<tr><th>Level</th><th>Operators</th><th>Description</th><th>Associativity</th></tr>
<tr><td>15</td><td>()
[]
-> .
++ --</td><td>Function Call
Array Subscript
Member Selectors
Postfix Increment/Decrement</td><td>Left to Right</td></tr>
<tr><td>14</td><td>++ --
+ -
! ~
(type)
*
&
sizeof</td><td>Prefix Increment / Decrement
Unary plus / minus
Logical negation / bitwise complement
Casting
Dereferencing
Address of
Find size in bytes</td><td>Right to Left</td></tr>
<tr><td>13</td><td>*
/
%</td><td>Multiplication
Division
Modulo</td><td>Left to Right</td></tr>
<tr><td>12</td><td>+ -</td><td>Addition / Subtraction</td><td>Left to Right</td></tr>
<tr><td>11</td><td>>>
<<</td><td>Bitwise Right Shift
Bitwise Left Shift</td><td>Left to Right</td></tr>
<tr><td>10</td><td>< <=
> >=</td><td>Relational Less Than / Less than Equal To
Relational Greater / Greater than Equal To</td><td>Left to Right</td></tr>
<tr><td>9</td><td>==
!=</td><td>Equality
Inequality</td><td>Left to Right</td></tr>
<tr><td>8</td><td>&</td><td>Bitwise AND</td><td>Left to Right</td></tr>
<tr><td>7</td><td>^</td><td>Bitwise XOR</td><td>Left to Right</td></tr>
<tr><td>6</td><td>|</td><td>Bitwise OR</td><td>Left to Right</td></tr>
<tr><td>5</td><td>&&</td><td>Logical AND</td><td>Left to Right</td></tr>
<tr><td>4</td><td>||</td><td>Logical OR</td><td>Left to Right</td></tr>
<tr><td>3</td><td>?:</td><td>Conditional Operator</td><td>Right to Left</td></tr>
<tr><td>2</td><td>=
+= -=
*= /= %=
&= ^= |=
<<= >>=</td><td>Assignment Operators</td><td>Right to Left</td></tr>
<tr><td>1</td><td>,</td><td>Comma Operator</td><td>Left to Right</td></tr>
</table>

Elimination of Ambiguity

<expr> ::= <expr> **+** <expr> | <expr> ***** <expr> | **ident**

Simple solution: change the language to force the programmer to write all parentheses

<expr> ::= **(** <expr> **+** <expr> **)** | **(** <expr> ***** <expr> **)** | **ident**

Exercise: convince yourself that this grammar is **not** ambiguous

Problem: too much work for the programmer – e.g., cannot just write **x + y + z** but must write **((x + y) + z)**

Better solution: let's not force the programmer to write all these **(** and **)**, but rather change the grammar accordingly

Elimination of Ambiguity

<expr> ::= <expr> **+** <expr> | <expr> ***** <expr> | **ident** | **const**

Note: added **const** to make the grammar more interesting

Goal: Create an equivalent non-ambiguous grammar with the appropriate precedence and associativity:

- ***** has higher precedence than **+**
- both are left-associative

Solution: two new non-terminals

<expr> ::= <expr> **+** <term> | <term>

<term> ::= <term> ***** <factor> | <factor>

<factor> ::= **ident** | **const**

Exercise: construct parse trees for **x + y + z** and **x + y * z** and imagine what the generated assembly code may look like

Adding Parentheses

Goal: extend the language to allow for parenthesized subexpressions – e.g., **x * (y + z)**

Solution:

<expr> ::= <expr> **+** <term> | <term>

<term> ::= <term> ***** <factor> | <factor>

<factor> ::= **ident** | **const** | **(** <expr> **)**

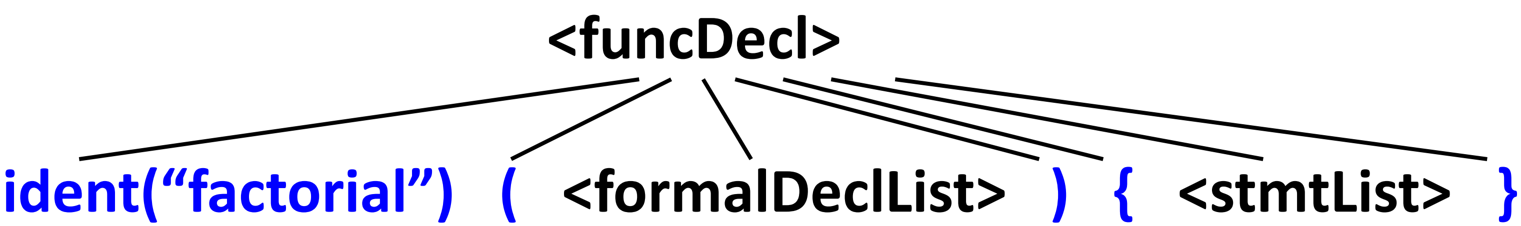
Exercise: construct the parse tree for **x * (y + z)** and imagine what the generated assembly code may look like

Exercise 2: look at the grammar definition in the CUP file for Project 1, convince yourself it is ambiguous, and see the extra “hints” to the parser about precedence and associativity (to resolve ambiguity)

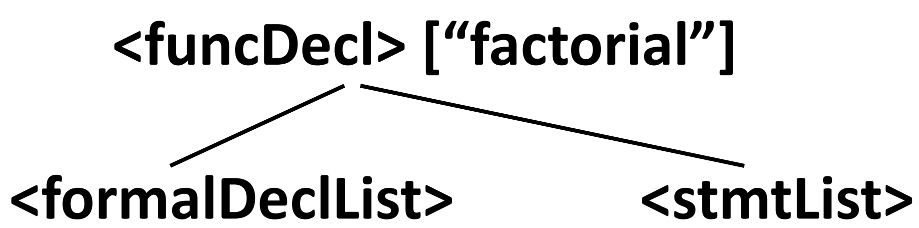
Abstract Syntax Trees (AST)

A simplified version of a **concrete** syntax tree, without loss of information [we will use ASTs in the programming projects]

<funcDef> ::= ident (<formalDeclList>) { < stmtList> }



parse tree



AST

Use of Context-Free Grammars

Syntax of a programming language

Java: Chapter 19 of the language specification (JLS) defines a grammar [under Resources on the web page]

Terminals: identifiers, keywords, literals, separators, operators

Starting non-terminal: **CompilationUnit**

Implementation of a **parser** in a compiler

e.g. the JLS grammar (Ch. 19) is used by the parser inside the **javac** compiler

Limitations of Context-Free Grammars

Cannot represent semantics

Example: "*every variable used in a statement should be declared earlier in the code*" or "*the use of a variable should conform to its type declaration*" (type checking)

Need to allow only programs that satisfy certain context-sensitive conditions

An example of a context: "an earlier declaration of **x** must exist, and it must declare an **int** type"

Cannot generate things other than parse trees

Example: what if we wanted to generate assembly code for the given program?

Attribute Grammars

Generalization of context-free grammars

Used for semantic checking and other compile-time analyses

e.g. type checking in a compiler

Used for translation

e.g. parse tree → assembly code

Implicitly represents a traversal of the parse tree and the computation of information during traversal

Structure of an Attribute Grammar

1. Underlying context-free grammar
2. For a terminal or non-terminal: some **attributes**
3. For each attribute: type of its possible values
   e.g., *integer* or *string* or *map(string* → *list(integer))*
4. Set of **evaluation rules** for each production
5. Set of boolean **conditions** for attribute values

Example

L = { $a^n b^n c^n$ | n > 0 }; not a context-free language

BNF

<start> ::= <A><B><C>

<A> ::= **a** | **a**<A>

<B> ::= **b** | **b**<B>

<C> ::= **c** | **c**<C>

Attributes

Na: associated with <A>

Nb: associated with <B>

Nc: associated with <C>

Type of possible values for Na, Nb, Nc: integer values

Example

Evaluation rules (similar for <B>, <C>)

<A> ::= **a**

<A>.Na := 1

| **a**<A>$_2$

<A>.Na := 1 + <A>$_2$.Na

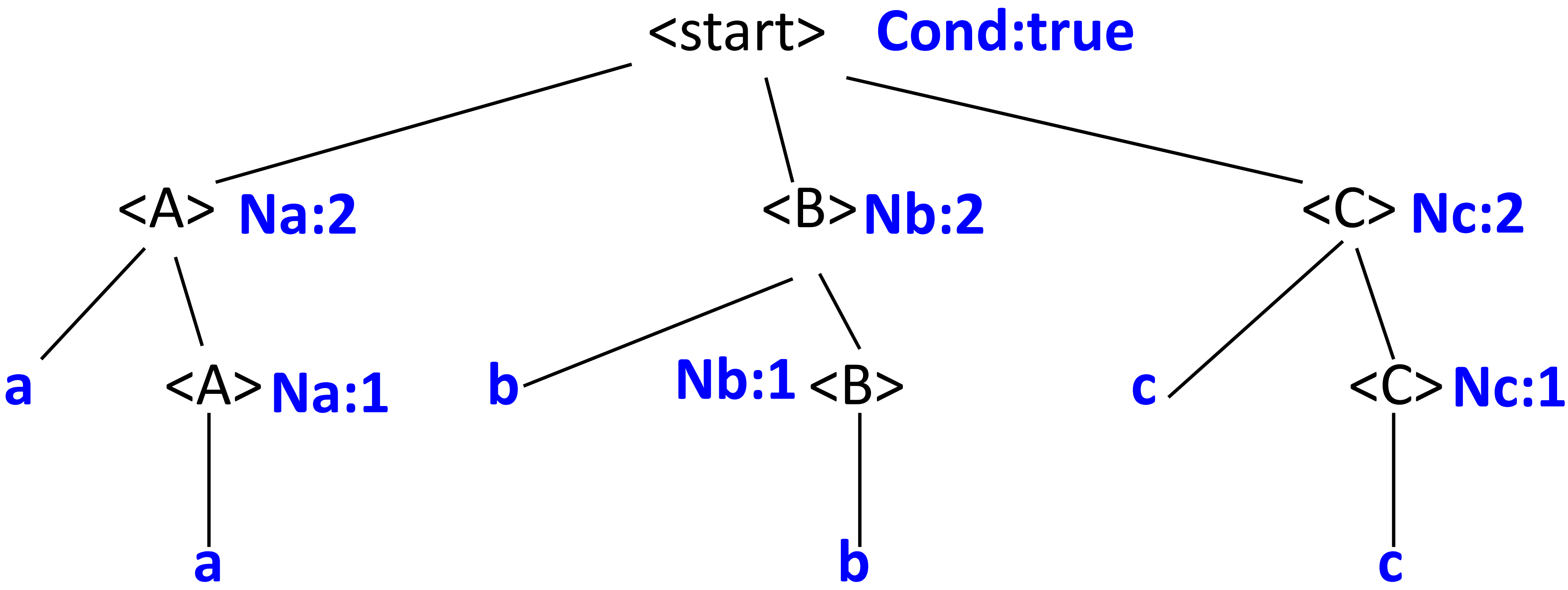
Conditions

<start> ::= <A><B><C>

Cond: [<A>.Na = <B>.Nb = <C>.Nc]

a string belongs to the language defined by this attribute grammar if and only if the parse tree satisfies the condition

Parse Tree



Parse Tree for an Attribute Grammar

Valid tree for the underlying BNF

Each node has (attribute,value) pairs

- One pair for each attribute associated with the node

Some nodes have boolean conditions

- If there is a corresponding Cond: … rule

Valid parse tree

- Attribute values are consistent with the evaluation rules
- **All** boolean conditions are true

Modified Example

Same evaluation rules as before e.g.

<A> ::= **a**

<A>.Na := 1

| **a**<A>$_2$

<A>.Na := 1 + <A>$_2$.Na

Different conditions

<start> ::= <A><B><C>

Cond: [<A>.Na = 3]

Cond: [<A>.Na > <B>.Nb]

Cond: [<B>.Nb > <C>.Nc]

How many valid parse trees exist for this attribute grammar?

Comments

If non-terminal X has an attribute A, each occurrence of X in the parse tree must have a value for A. The evaluation rules should define exactly one value for A for a particular X node.

- Attributes are not like program variables; cannot have:
  <Z>.A := 1 + <Z>.A

In rules/conditions, can only refer to attributes of non-terminals and terminals in the current production alternative

- Cannot look at "grandparent"/"grandchild" parse tree nodes, or even further away up/down the tree

Synthesized vs. Inherited Attributes

Each non-terminal X: disjoint sets of **synthesized** attributes and **inherited** attributes

An attribute A for X cannot be both

For each synthesized attribute A: each production alternative in **X ::= …** should have exactly one evaluation rule for X.A

For each inherited attribute A: each occurrence of X in **… ::= … X … X … X …** should have exactly one evaluation rule for X.A

Synthesized vs. Inherited Attributes

Synthesized attributes convey information about the subtree rooted at the node

Inherited attributes convey context conditions

- E.g., information about variable declarations that have appeared earlier in the program
- The starting non-terminal does not have inherited attributes

For convenience: assume each **terminal** symbol has one attribute lexval with a pre-defined value

- The lexical analyzer sets these values (e.g., some *int* value for a token representing an integer constant)

Example (revisited)

<start> ::= <A><B><C>

<B>.expectedNb := <A>.Na

<C>.expectedNc := <A>.Na

<A> ::= **a**

<A>.Na := 1

| **a**<A>$_2$

<A>.Na := 1 + <A>$_2$.Na

<B> ::= **b**

Cond: [<B>.expectedNb = 1]

| **b**<B>$_2$

<B>$_2$.expectedNb := <B>.expectedNb – 1

similarly for <C>

Na is synthesized, **expectedNb/Nc** are inherited

Example: Binary Numbers

Context-free grammar

<B> ::= <D>

<B> ::= <D><B>

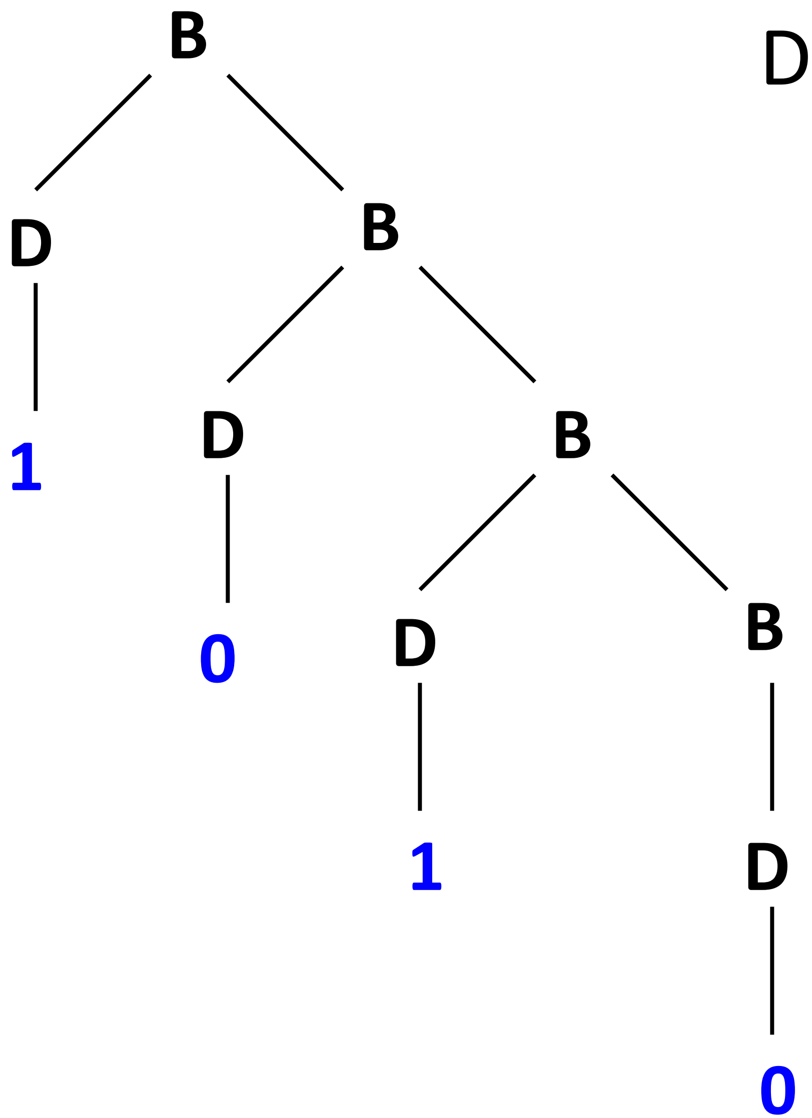
<D> ::= **0**

<D> ::= **1**

Goal: compute the **value** of the binary number

Needed, for example, in compilers during code translation

BNF Parse Tree for Input 1010



Define integer attributes

<B>: synthesized val

<B>: synthesized pos

<D>: inherited pow

<D>: synthesized val

Example: Binary Numbers

<B> ::= <D>

$\langle B\rangle.pos := 1$

$\langle B\rangle.val := \langle D\rangle.val$

$\langle D\rangle.pow := 0$

<B>$_1$ ::= <D><B>$_2$

$\langle B\rangle_1.pos := \langle B\rangle_2.pos + 1$

$\langle B\rangle_1.val := \langle B\rangle_2.val + \langle D\rangle.val$

$\langle D\rangle.pow := \langle B\rangle_2.pos$

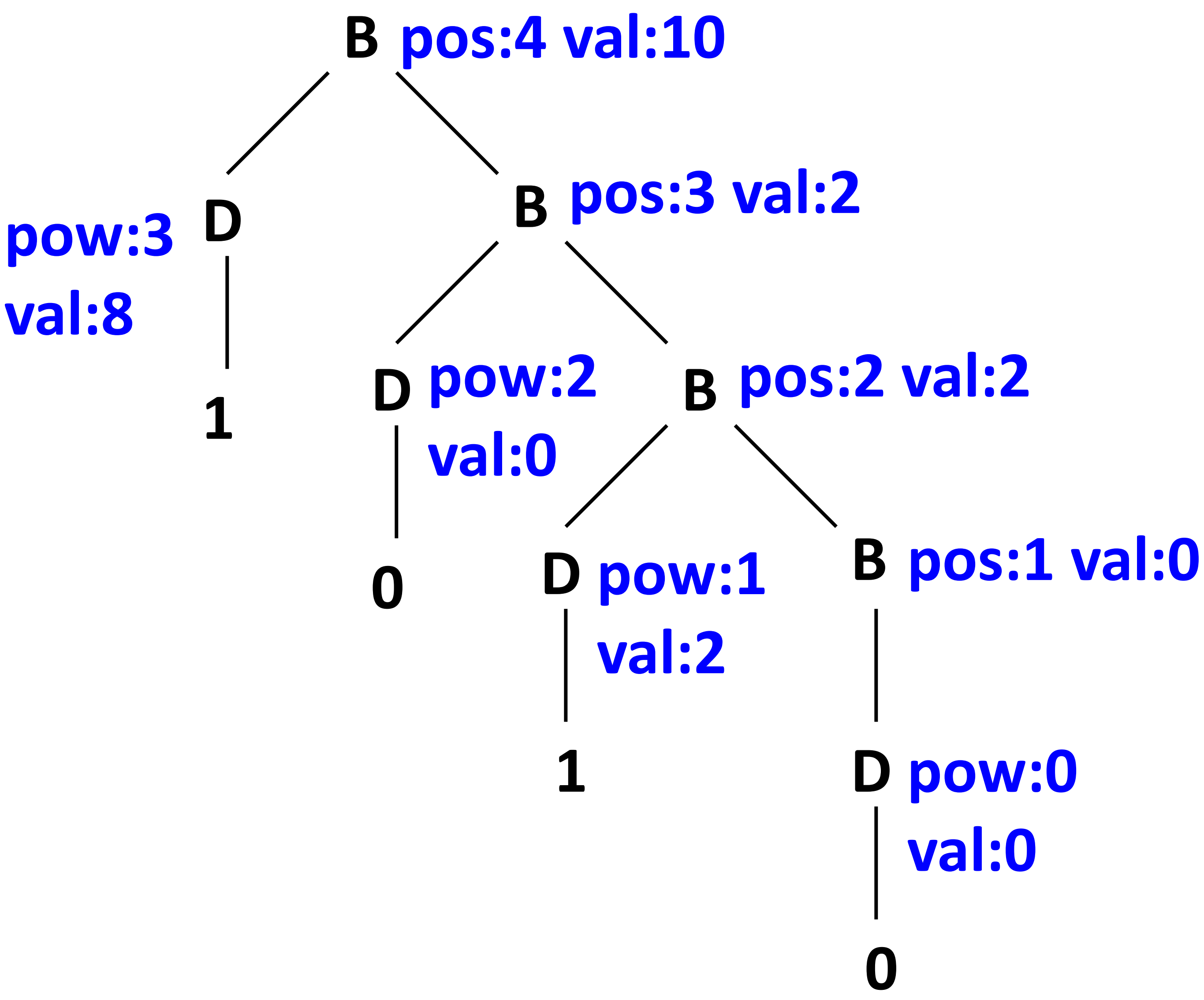
<D> ::= **0**

$\langle D\rangle.val := 0$

<D> ::= **1**

$\langle D\rangle.val := 2^{\langle D\rangle.pow}$

Evaluated Parse Tree



Complex Evaluation Rules

<X>.A := … could be rather complex – e.g. with helper functions, conditional expressions, etc.

Example:

<X>.A := if (<Y>.B = <Z>.C) then f1(<Y>.D) else f2(<Z>.E)

- Must be if-then-else; cannot be if-then. Why?
- Helper functions such as f1 and f2 can use basic algorithms and data structures/operations
- Can only use attributes of non-terminals and terminals that appear in this production alternative

Attribute Evaluation: Dependence Graph

<X>.A := <Y>.B + <Z>.C

Since the value of <X>.A depends on <Y>.B:

Y.B → X.A dependence edge

Since the value of <X>.A depends on <Z>.C:

Z.C → X.A dependence edge

<X>$_1$.A := <X>$_2$.A two different X nodes in the parse tree

Since the value of <X>$_1$.A depends on <X>$_2$.A:

X_2.A → X_1.A dependence edge

Algorithm for Attribute Evaluation

Given a parse tree with attributes attached to tree nodes, how do we compute the attribute values?

Step 1: find evaluation order of attributes

a) Build dependence graph where a node is a pair (parse tree node, attribute)
b) Complain about cycles in the graph: cannot evaluate
c) Topologically sort the graph

Step 2: evaluate the attributes in sorted order

Example: Binary Numbers

<B> ::= <D>

$$\langle B\rangle.\textit{pos} := 1$$
$$\langle B\rangle.\textit{val} := \langle D\rangle.\textit{val}$$
$$\langle D\rangle.\textit{pow} := 0$$

$\langle B\rangle_1$ ::= <D>$\langle B\rangle_2$

$$\langle B\rangle_1.\textit{pos} := \langle B\rangle_2.\textit{pos} + 1$$
$$\langle B\rangle_1.\textit{val} := \langle B\rangle_2.\textit{val} + \langle D\rangle.\textit{val}$$
$$\langle D\rangle.\textit{pow} := \langle B\rangle_2.\textit{pos}$$

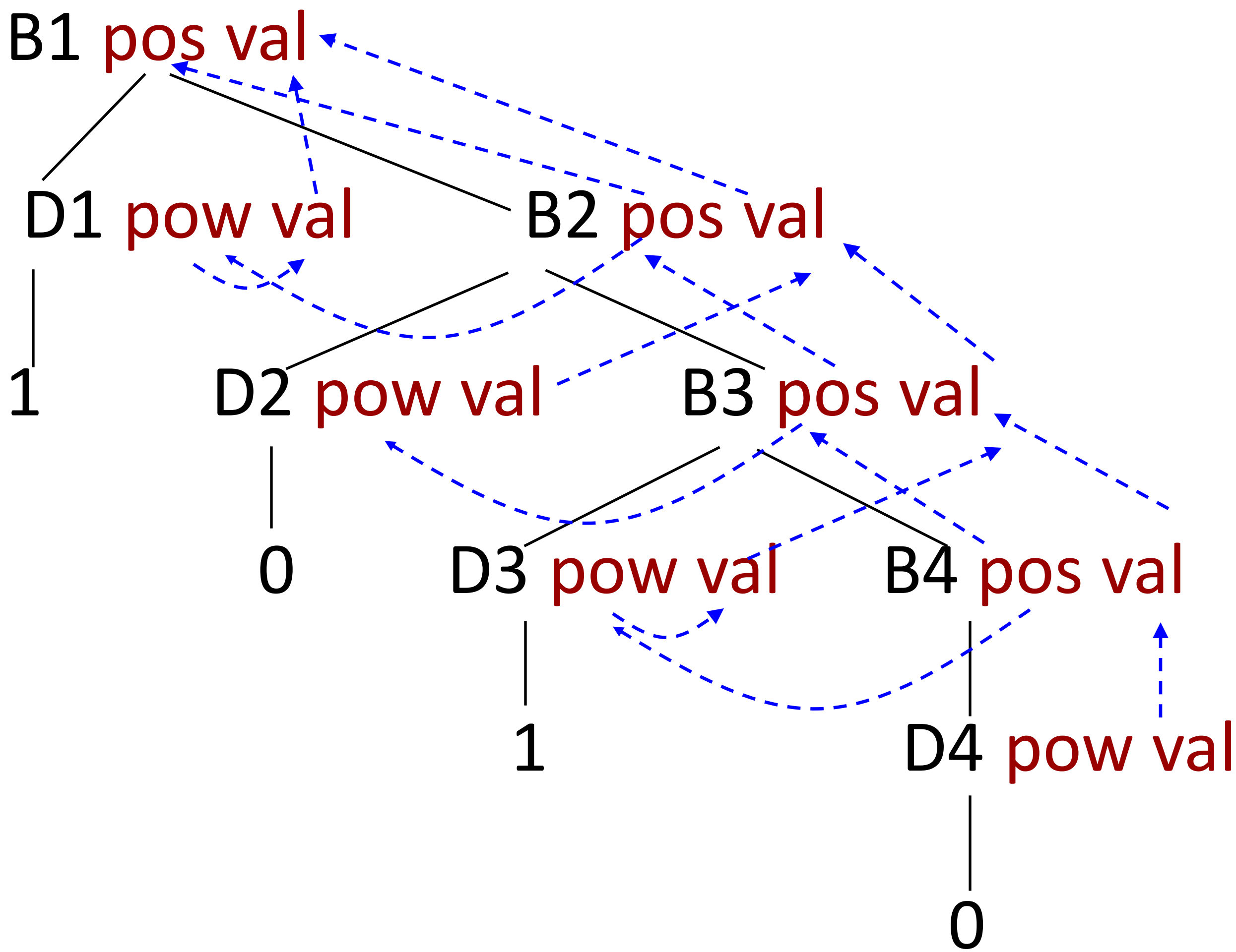
<D> ::= **0**

$$\langle D\rangle.\textit{val} := 0$$

<D> ::= **1**

$$\langle D\rangle.\textit{val} := 2^{\langle D\rangle.\textit{pow}}$$

Dependence Graph for Binary Numbers



Sort the Graph

Topological sort: **x** is "smaller" than **y** iff **x → y**

D4.pow, B4.pos, D4.val, B4.val,

B3.pos, D3.pow, D3.val, B3.val,

B2.pos, D2.pow, D2.val, B2.val,

B1.pos, D1.pow, D1.val, B1.val

Cycles

The notion of “topological sort” only makes sense for directed **acyclic** graphs

Cycles in the dependence graph means we have recursive dependencies

- In general, there are approaches to solve meaningful recursive systems of equations

But, in this course we will disallow cycles

- No cyclic dependencies in exams and homeworks

Use Scenario 1: Simple Type Checking

<program> ::= <stmtList>

<stmtList> ::= <stmt> <stmtList> | <stmt>

<stmt> ::= <varDecl> **=** <expr> **;**

| **ident =** <expr> **;**

<varDecl> ::= **int ident** | **float ident**

<expr> ::= **intconst** | **floatconst** | **ident**

| <expr> **+** <expr>

| **(** <expr> **)**

[grammar is ambiguous; assume the parser resolved this somehow]

Type Checking: Simple Examples

Example 1:

int y = x + w; int x = y; vs **int y = 5 + 3; int x = y;**

Discussed in the next few slides

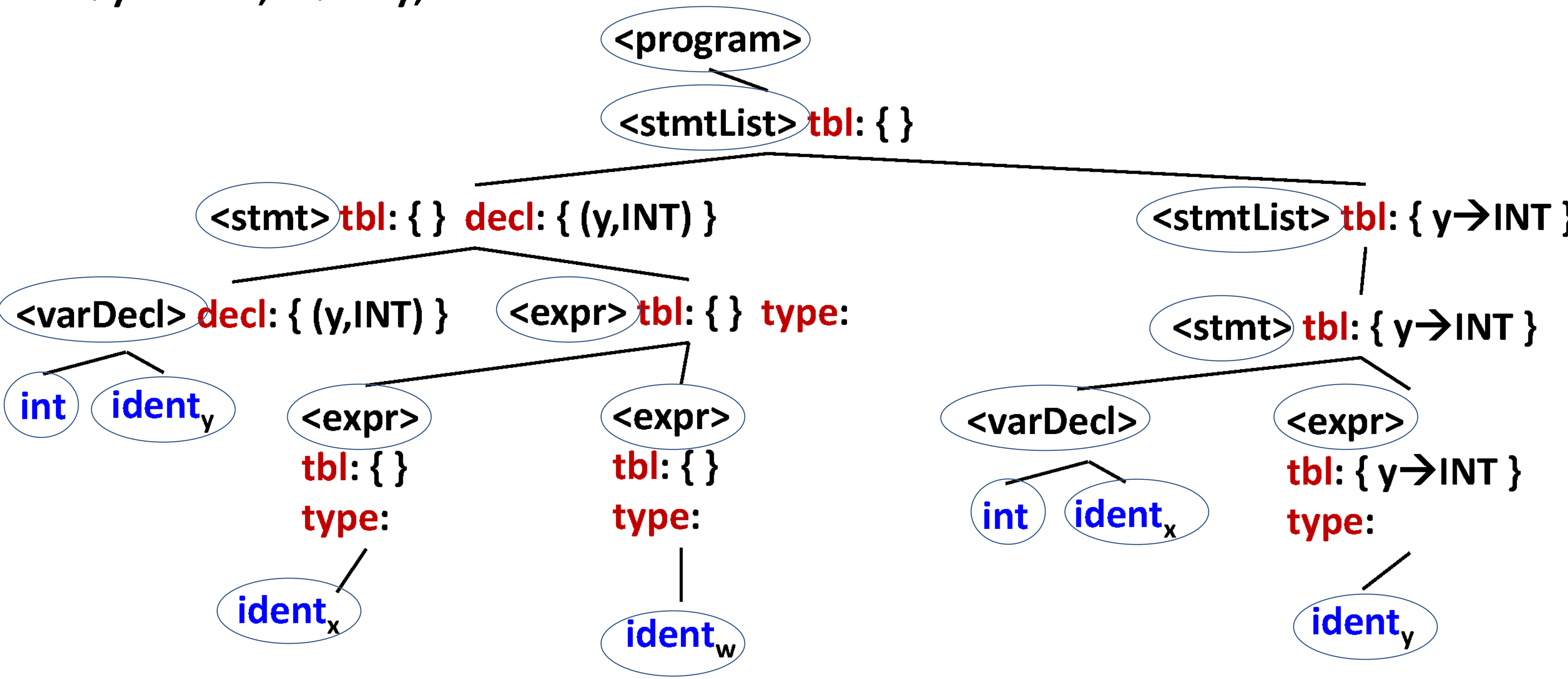
Example 2: for practice

float x = 5.0; float y = x + 1.0; int z = x + y;

Will this type check in Java?

Should it type check in our language? It's up to us. We will choose "No".

int y = x + w; int x = y;



Note: not showing some of the "uninteresting" terminal symbols, to simplify the picture (they are still in the tree)

Our Type Checking Goals

Goal 1: Any variable in an <expr> must have a corresponding declaration in an earlier <stmt>

Example: do not allow **int x = 1; int y = x + w;**

Example: do not allow **int x = x+1;**

Note: in the programming project will also check that **no variable is declared more than once**; in class we will not discuss this check, but you should think how the solution should be changed to perform such checking

Goal 2: Both operands of **+** must be of the same type

Example: do not allow **int x = 1; float y = x + 3.14;**

Attributes for Type Checking Solution

Inherited attribute tbl (short for “symbol table”). The attribute is a map from strings to INT/FLOAT. Each <stmtList>, <stmt>, and <expr> has its tbl.

Synthesized attribute type for <expr>: INT/FLOAT

> When the <expr> is an **ident** (just a variable name), need to look inside <expr>.tbl to figure out if the variable was already declared and with what type

Type Checking: Expressions

<expr> ::= **intconst** *<expr>.type := INT*

| **floatconst** *<expr>.type := FLOAT*

| **ident**

*Cond: [**ident**.lexval has a type in <expr>.tbl]*

*<expr>.type := <expr>.tbl.lookupId(**ident**.lexval)*

| **(** $\langle expr \rangle_2$ **)**

$\langle expr \rangle_2$.tbl := <expr>.tbl.clone() Copies the entire table

<expr>.type := $\langle expr \rangle_2$.type

Type Checking: Expressions

<expr> ::= <expr>$_2$ **+** <expr>$_3$

<expr>$_2$.tbl := <expr>.tbl.clone()

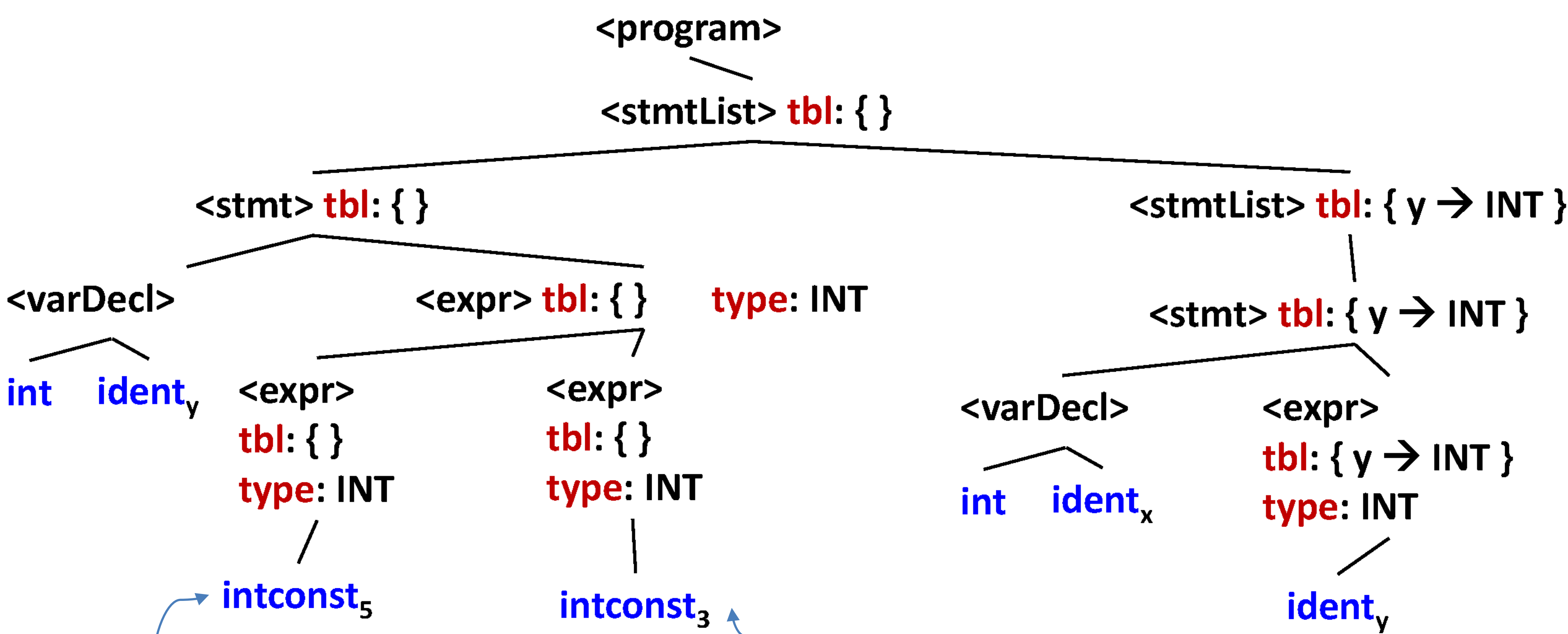
<expr>$_3$.tbl := <expr>.tbl.clone()

Cond: [<expr>$_2$.type = <expr>$_3$.type]

<expr>.type := <expr>$_2$.type

Note: this would disallow code such as **int x = 1; float y = x + 3.14;**

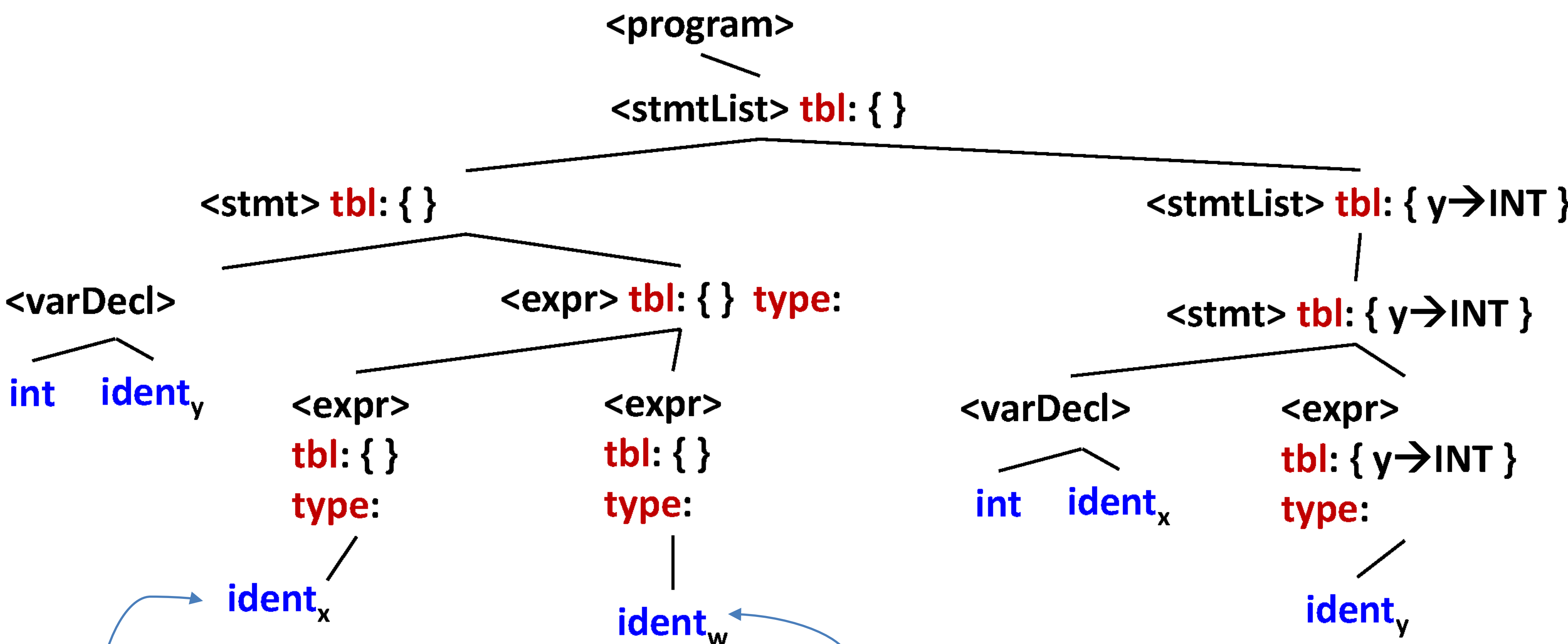
int y = 5 + 3; int x = y;



Cond: [<expr>$_2$.type = <expr>$_3$.type]

What will happen here with this check?

int y = x + w; int x = y;



Cond: [ident.lexval has a type in <expr>.tbl]

What will happen here with this check?

Attributes for Type Checking Solution

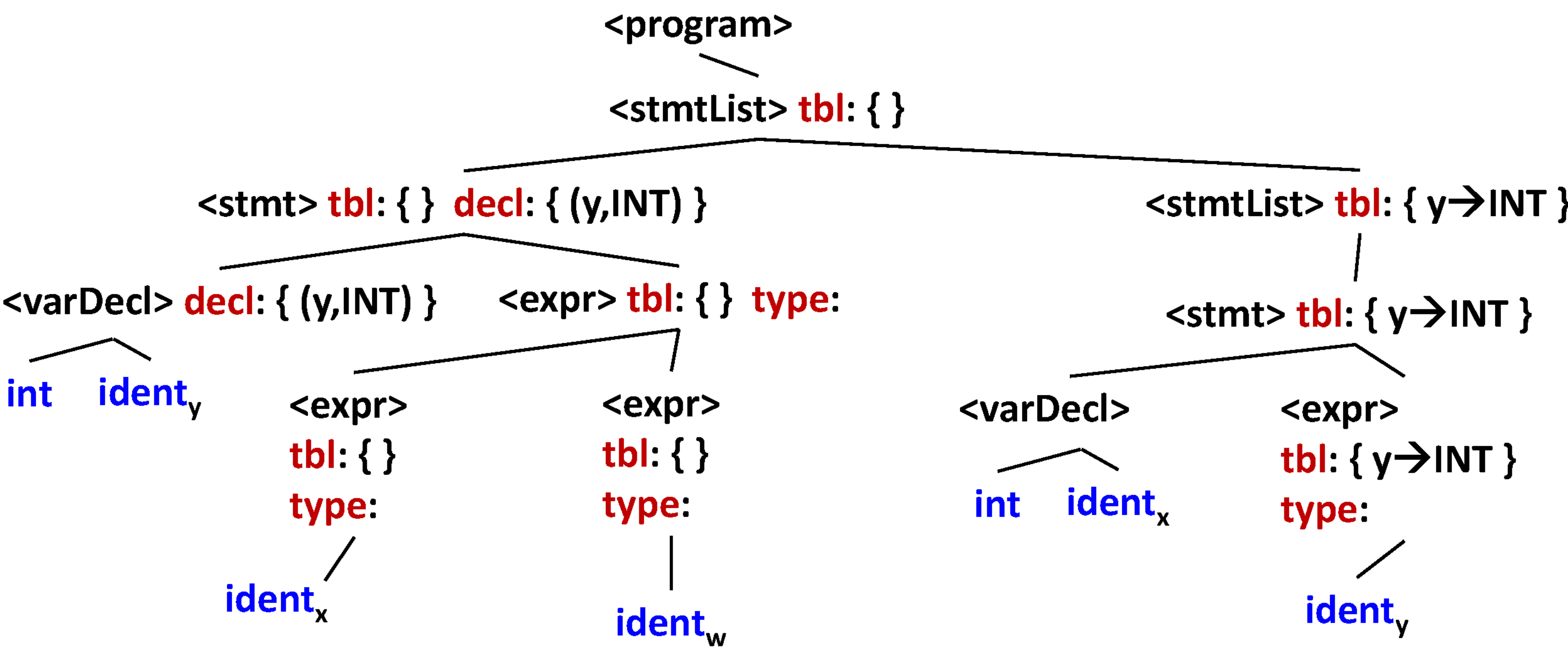
Inherited attribute tbl (short for "symbol table"). The attribute is a map from strings to INT/FLOAT. Each <stmtList>, <stmt>, and <expr> has its tbl.

Synthesized attribute type for <expr>: INT/FLOAT

When the <expr> is an **ident** (just a variable name), need to look inside <expr>.tbl to figure out if the variable was already declared and with what type

Synthesized attribute decl for <varDecl> and <stmt>: a set containing zero or one pair (string,INT/FLOAT)

int y = x + w; int x = y;



Note: not showing **decl** in this part of the tree (but it is there)

Type Checking: Symbol Tables

<program> ::= <stmtList>

<stmtList>.tbl := newTable() empty table

<stmtList> ::= <stmt> <stmtList>$_2$

<stmt>.tbl := <stmtList>.tbl.clone()

<stmtList>$_2$.tbl := <stmtList>.tbl.clone(<stmt>.decl)

Creates a copy of <stmtList>.tbl and adds to it <stmt>.decl

| <stmt>

<stmt>.tbl := <stmtList>.tbl.clone()

Type Checking: Symbol Tables

<varDecl> ::= **int ident**

*<varDecl>.decl := newSet(**ident**.lexval,INT)* Set with one element: a pair (string,INT)

| **float ident** Similarly here

<stmt> ::= <varDecl> **=** <expr> **;**

<stmt>.decl := <varDecl>.decl.clone()

<expr>.tbl := <stmt>.tbl.clone()

| **ident =** <expr> **;**

<stmt>.decl := newSet() *empty set*

<expr>.tbl := <stmt>.tbl.clone()

Type Checking: Assignments

Goal 1: Any variable in an <expr> must have a corresponding declaration in an earlier <stmt>

Example: do not allow **int x = 1; int y = x + w;**

Example: do not allow **int x = x+1;**

Goal 2: Both operands of **+** must be of the same type

Example: do not allow **int x = 1; float y = x + 3.14;**

Goal 3: Both sides of an assignment must be of the same type

Example: do not allow **int x = 1; float y = x;**

Type Checking: Assignments

<stmt> ::= <varDecl> **=** <expr> **;**

Cond: [<expr>.type = type in <varDecl>.decl]

| **ident =** <expr> **;**

*Cond: [**ident**.lexval has a type in <stmt>.tbl]*

*Cond: [<expr>.type = <stmt>.tbl.lookupId(**ident**.lexval)]*

Example

Consider again Example 1:

int y = x + w; int x = y; vs **int y = 5 + 3; int x = y;**

Already saw parse tree and attributes tbl, type, and decl

Where in the tree do the type checks occur?

<expr> ::= **indent** *Cond: [ident.lexval has a type in <expr>.tbl]*

<expr> ::= <expr>$_2$ **+** <expr>$_3$ *Cond: [<expr>$_2$.type = <expr>$_3$.type]*

<stmt> ::= <varDecl> = <expr> ; *Cond: [<expr>.type = type in <varDecl>.decl]*

<stmt> ::= ident = <expr> ; *Cond: [ident.lexval has a type in <stmt>.tbl]*

Cond:[<expr>.type=<stmt>.tbl.lookupId(ident.lexval)]

Efficiency Of Type Checking

Inherited attribute tbl: each <stmtList>, <stmt>, and <expr> has its own table, which is inefficient

- Consider a list of *n* variable declarations. What is the total size of all tbl attributes?

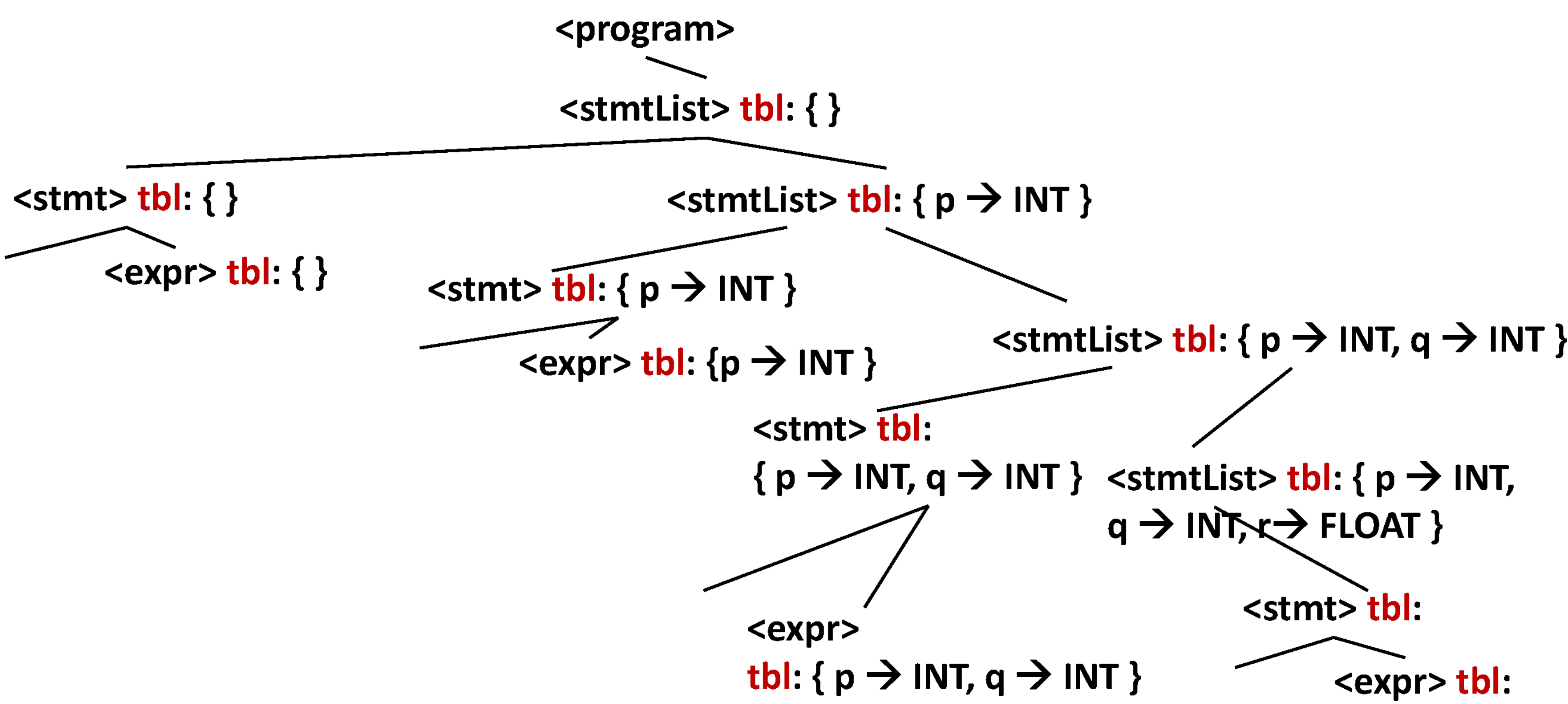
Let's just have one single "global" table

- *Advantage*: more efficient use of space; no need for clone() operations
- *Disadvantage*: need to be very careful in which order attributes are evaluated and how this affects the table

Modified solution: at each <stmtList>, <stmt>, and <expr>, tbl is a **pointer** to a single global table

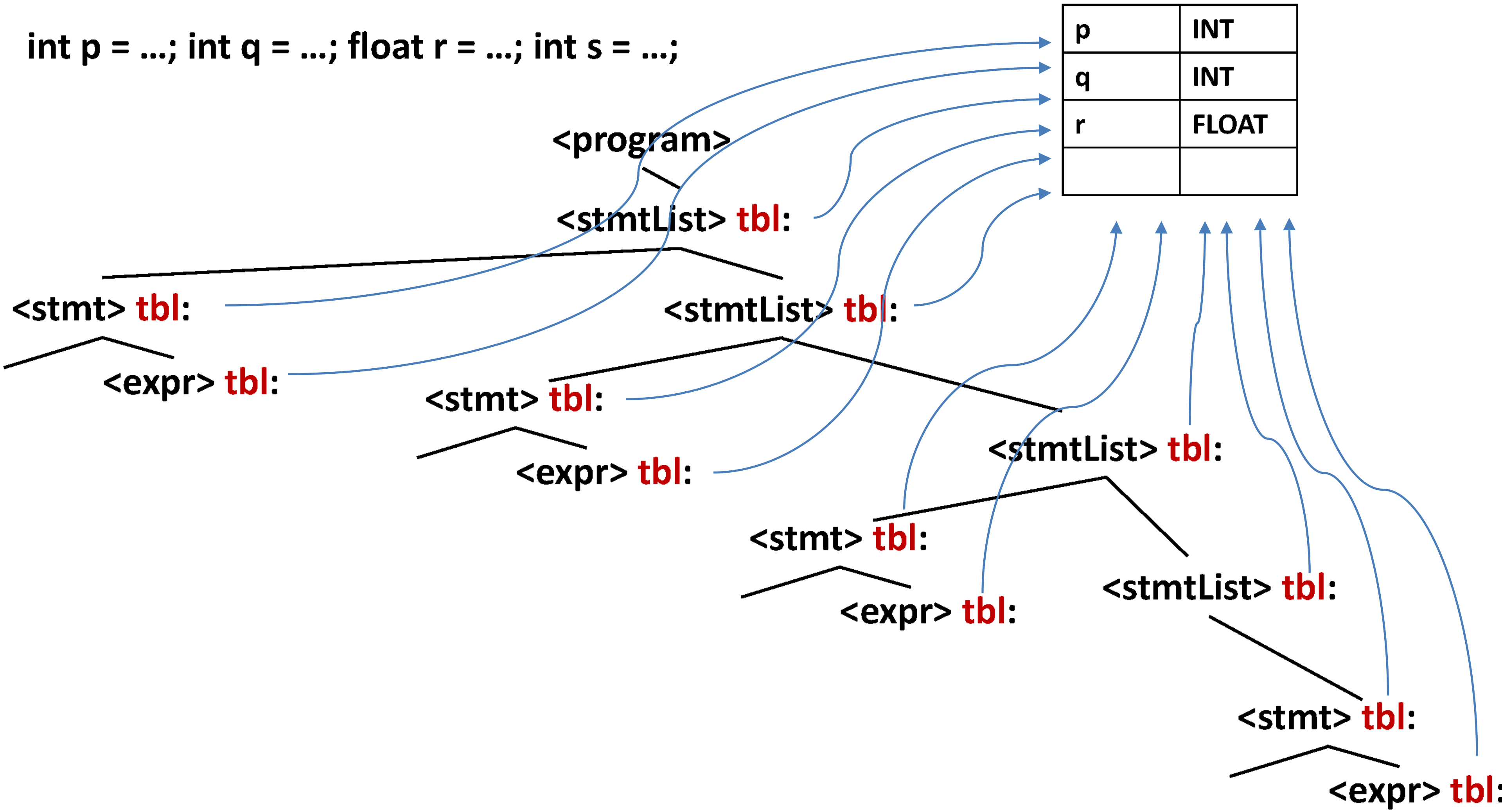
Inefficient

int p = …; int q = …; float r = …; int s = …;



Efficient

int p = …; int q = …; float r = …; int s = …;



Typechecking: Expressions

<expr> ::= **intconst** *<expr>.type := INT*

| **floatconst** *<expr>.type := FLOAT*

| **ident**

*Cond: [**ident**.lexval has a type in <expr>.tbl]*

*<expr>.type := <expr>.tbl.lookupId(**ident**.lexval)*

| **(** $\langle expr \rangle_2$ **)**

$\langle expr \rangle_2$.tbl := <expr>.tbl Copies the pointer; both point to the same global table

<expr>.type := $\langle expr \rangle_2$.type

Typechecking: Expressions

<expr> ::= <expr>$_2$ **+** <expr>$_3$

<expr>$_2$.tbl := <expr>.tbl was *<expr>.tbl.clone()*

<expr>$_3$.tbl := <expr>.tbl was *<expr>.tbl.clone()*

Cond: [<expr>$_2$.type = <expr>$_3$.type]

<expr>.type := <expr>$_2$.type

Typechecking: Symbol Tables

<program> ::= <stmtList>

<stmtList>.tbl := newTable() empty table

<stmtList> ::= <stmt> <stmtList>_2

<stmt>.tbl := <stmtList>.tbl

{ <stmtList>_2.tbl := <stmtList>.tbl;

<stmtList>_2.tbl.insertId(<stmt>.decl) "side effect" of the evaluation }

| <stmt>

<stmt>.tbl := <stmtList>.tbl

Typechecking: Symbol Tables

<varDecl> ::= **int ident**

*<varDecl>.decl := newSet(**ident**.lexval,INT)* Set with one element: a pair (string,INT)

| **float ident** Similarly here

<stmt> ::= <varDecl> **=** <expr> **;**

<stmt>.decl := <varDecl>.decl

<expr>.tbl := <stmt>.tbl

| **ident =** <expr> **;**

<stmt>.decl := newSet() empty set

<expr>.tbl := <stmt>.tbl

Example

Consider

int y = 5 + 3; int x = y;

All tbl are now pointers to the same table

It matters when the checks are performed relative to the *insertId* side effects

Specifically, for <stmtList> ::= <stmt> <stmtList>$_2$:

checks inside <stmt> should happen before the side effect of *<stmtList>$_2$.tbl.insertId(<stmt>.decl)* but after insertId side effects for <stmtList> nodes that are higher in the parse tree

Attribute Grammars with Side Effects

More generally, can we have "global" data structures, i.e., data shared among tree nodes?

Pure attribute grammars: nothing is shared; each node has its own local data, computed once and unchangeable (for example, the first version of type checking)

Advantage: easy to decide the order of evaluation of attributes as we don't have to worry about order of updates to shared data

Attribute grammars with side effects: some shared data and limited side effects on it (Dragon book, Sec 5.1 and 5.2: also known as "syntax-directed definitions")

Advantage: efficiency

Side Effects and Order of Evaluation

Pure attribute grammars: any topological sort order is a valid evaluation order

Attribute grammars with side effects: we need to define additional restrictions on the evaluation (e.g., as we did for insertId for the type checking attribute grammar)

Think of these restrictions as additional artificial edges in the dependence graph (Dragon book, Sec 5.2.5)

Use Scenario 2: More Type Checking

In general, type checking is a form of semantic checking that a compiler will perform after parsing, on the parse tree (or, more likely, on the AST)

- An attribute grammar specifies both the goals of typechecking and (implicitly) the actual algorithm

A generalization of our earlier example: given program with declarations, check types of identifiers (integers, floats, functions)

- For type checking inside a nested block, use the innermost variable declarations
- Will not discuss the complete grammar, just key ideas

Context-Free Grammar

<program> ::= <funcDefList>

<funcDefList> ::= <funcDef> <functDefList> | <funcDef>

<funcDef> ::= <varDecl> **(** <formalDeclList> **)** **{** <stmtList> **}**

<varDecl> ::= **int ident** | **float ident**

<stmt> ::= … | **{** <stmtList> **}** | **while**, **if**, **return** statements (not shown)

<expr> ::= … | **ident (** <exprList> **)** function call

Example:

int f (int x, int y) { int z = x+y; return z; }

int g(int x) { int z = 5; { int t = x+z; return t; } }

int main (int w) { return f(w+1,w+2) + g(8); }

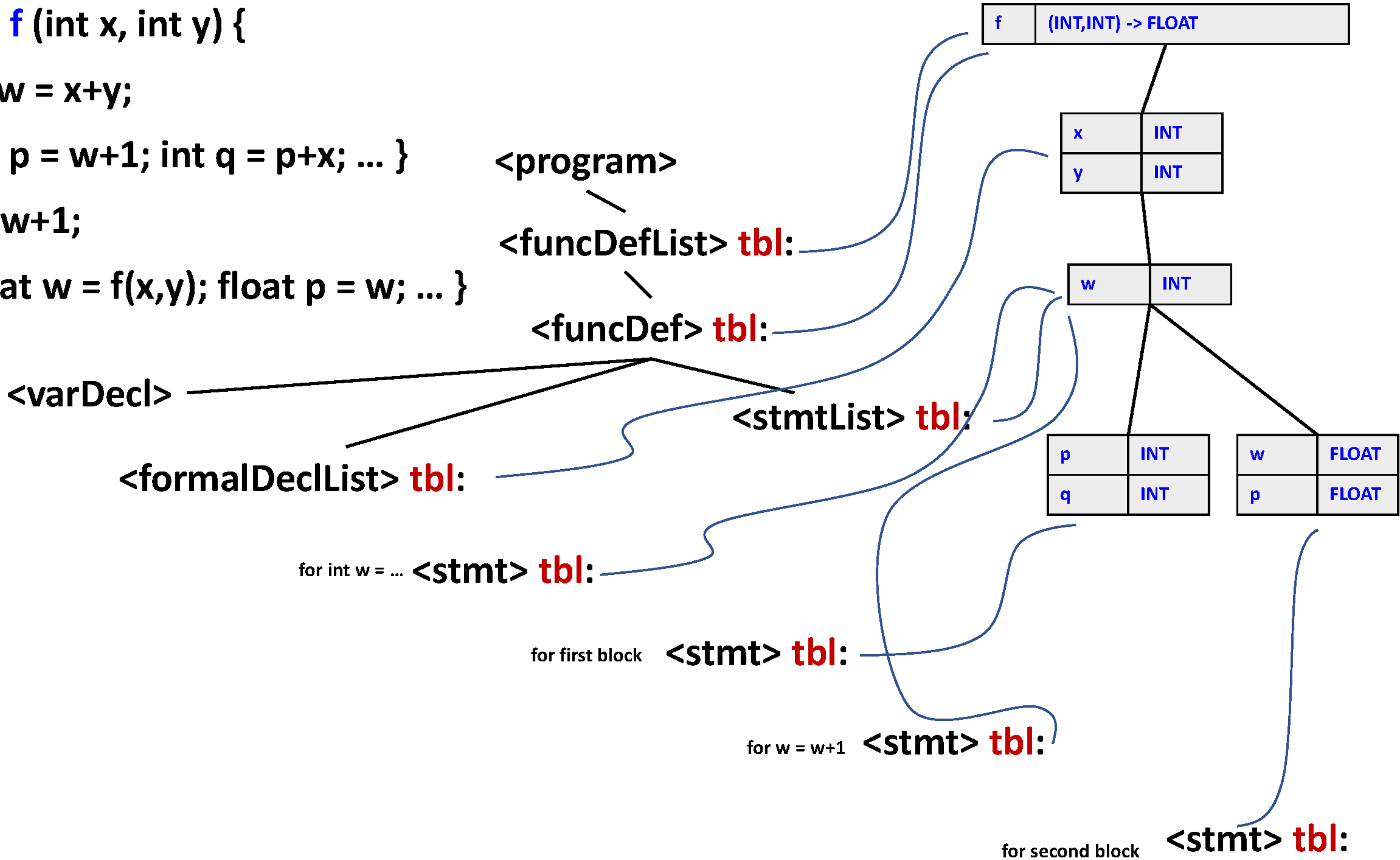
Type Checking Goals

Goal 1: Any variable in an <expr> must have an earlier corresponding declaration, including (1) vars from surrounding blocks and (2) function parameters

Goal 2: Any function name in a call must have a corresponding definition somewhere in the program; types of actual parameters at the call should match types of formal parameters at the definition; similarly for the return type

Idea: use a tree of symbol tables

```
float f (int x, int y) {
 int w = x+y;
 { int p = w+1; int q = p+x; ... }
 w = w+1;
 { float w = f(x,y); float p = w; ... }
}
```



Type Checking: Expressions

<expr> ::= … | **ident**

Cond: [***ident****.lexval has a type in <expr>.tbl]*

*<expr>.type := <expr>.tbl.lookupId(****ident****.lexval)*

Note: lookupId checks the table, its parent table, the grandparent table, etc. until a match is found

Type Checking: Symbol Tables

<stmt> ::= … | **{** <stmtList> **}** nested block

<stmtList>.tbl := <stmt>.tbl.newChildTable()

int f (int x, float y) { int z … { float w … } { int v …} }

Root table T_1: (f, (INT,FLOAT)→INT)

T_2, child of T_1, table for formals: (x,INT) and (y,FLOAT)

T_3, child of T_2, table for locals: (z,INT)

T_4, child of T_3, table for first nested block: (w,FLOAT)

T_5, child of T_3, table for second nested block: (v,INT)

Type Checking: Function Calls

<expr> ::= … | **ident (** <exprList> **)**

*Cond: [**ident**.lexval has a function type in <expr>.tbl]*

*Cond: [formal types in <expr>.tbl.lookupId(**ident**.lexval) match actual types in <exprList>]*

<expr>.type := return type from <expr>.tbl.lookupId(…)

Use Scenario 3: Code Generation

Given: parse tree for a simple program (after type checking)

Goal: translate to assembly code

The evaluation rules of the attribute grammar generate the assembly code

Note: in a real compiler, the parse tree (or AST) will be translated to a machine-independent simplified representation (e.g., three-address code) which is then optimized and translated to machine-specific assembly code. Details in CSE 5343 “Compilers”.

Code Generation for Expressions

<expr> ::= **intconst** | **ident** | **(** <expr> **)**

| <expr> **+** <expr> | <expr> ***** <expr>

Output language

Assembly language for a machine with an infinite number of registers R_1, R_2, … and instruction set as follows

LOAD R_i, x: copy the value of variable x into R_i

LOAD R_i, const: set the value of R_i to an integer constant

STORE x, R_i: write R_i to variable x

ADD R_i, R_j, R_k: add R_j and R_k and store in R_i (R_i could be same as R_j or R_k)

MUL R_i, R_j, R_k: multiply R_j and R_k and store in R_i

Code Generation Strategy

Synthesized attribute code contains a sequence of instructions: concatenation of subsequences from its children, plus new instructions

$$\langle expr \rangle ::= \langle expr \rangle_2 + \langle expr \rangle_3$$

$$\langle expr \rangle.code :=$$

$$\langle expr \rangle_2.code$$

$$\langle expr \rangle_3.code$$

$$\text{"ADD"}\ R_{[for \langle expr \rangle]}, R_{[for \langle expr \rangle_2]}, R_{[for \langle expr \rangle_3]}$$

Simple Code Generation

<expr> ::= **intconst**

<expr>.reg := newReg() *// create a new register name*

*<expr>.code := newInstr(**LOAD**, <expr>.reg, **intconst**.lexval)*

| **ident**

<expr>.reg := newReg()

*<expr>.code := newInstr(**LOAD**, <expr>.reg, **ident**.lexval)*

| <expr>$_2$ **+** <expr>$_3$ // similarly for *****

<expr>.reg := newReg()

<expr>.code := concat(<expr>$_2$.code, <expr>$_3$.code,
*newInstr(**ADD**, <expr>.reg, <expr>$_2$.reg, <expr>$_3$.reg))*

| **(** <expr>$_2$ **)**

<expr>.reg := <expr>$_2$.reg

<expr>.code := <expr>$_2$.code

Observations

newReg(): creates a unique register name. This is a side effect, but the order of these side effects does not matter

We are assuming an infinite number of “abstract” registers, but in reality there is a limit; in compilers, a register allocation pass re-maps the abstract registers to a finite number of real registers

Example

(x+99)*z + v*w

```
LOAD  R1, x
LOAD R2, 99
ADD R3, R1, R2
LOAD R4, z
MUL R5, R3, R4
LOAD R6, v
LOAD R7, w
MUL R8, R6, R7
ADD R9, R5, R8
```

Code Generation for Statements

<stmtList> ::= <stmt> <stmtList> | <stmt>

<stmt> ::= **ident =** <expr> **;** | **if (**<cond>**)** <stmt> **else** <stmt>

| **while (**<cond>**)** <stmt> | **{** <stmtList> **}**

Output language

Labels for some instructions; jump instructions BR and BZ

BR L_i: branch unconditionally to instruction with label L_i

BZ R_i, L_k: branch to instruction with label L_k but only if the value in register R_i is zero (BZ = branch on zero); in many machine languages, zero is a way to represent "false"

L_i : : label L_i; target of BR/BZ

Code Generation for Statements

<stmtList> ::= <stmt> <stmtList>$_2$

<stmtList>.code := concat(<stmt>.code,<stmtList>$_2$.code)

| <stmt>

<stmtList>.code := <stmt>.code

<stmt> ::= **ident =** <expr> **;**

<stmt>.code := concat(<expr>.code,

*newInstr(**STORE**, **ident**.lexval, <expr>.reg))*

| **{** <stmtList> **}**

<stmt>.code := <stmtList>.code

Code Generation for Statements

<stmt> ::= **if (**<cond>**)** <stmt>$_2$ **else** <stmt>$_3$

<stmt>.elseLabel := newLabel()

<stmt>.exitLabel := newLabel()

<stmt>.code := concat(

<cond>.code, // leaves 0 in <cond>.reg if condition is "false"

*newInstr(**BZ**, <cond>.reg, <stmt>.elseLabel),*

<stmt>$_2$.code,

*newInstr(**BR**, <stmt>.exitLabel),*

<stmt>.elseLabel,

<stmt>$_3$.code,

<stmt>.exitLabel)

Example

if (...) x = y+5; else x = 8;

```
code for ... // leaves 0 in R8 if condition is "false"
BZ R8, L33 // jump to "else" if condition is "false"
LOAD R1, y
LOAD R2, 5
ADD R3, R1, R2
STORE x, R3
BR L34
L33: // else label
LOAD R4, 8
STORE x, R4
L34: // exit label
```

Code Generation for Statements

<stmt> ::= **while (**<cond>**)** $\langle stmt\rangle_2$

<stmt>.startLabel := newLabel()

<stmt>.exitLabel := newLabel()

<stmt>.code := concat(

<stmt>.startLabel,

<cond>.code, *// leaves 0 in <cond>.reg if condition is "false"*

*newInstr(**BZ**, <cond>.reg, <stmt>.exitLabel),*

$\langle stmt\rangle_2$.code,

*newInstr(**BR**, <stmt>.startLabel),*

<stmt>.exitLabel)

Example

while (...) x = x+1;

```
L15: // start label
code for ... // leaves 0 in R8 if condition is “false”
BZ R8, L16 // jump to “exit” if condition is “false”
LOAD R1, x
LOAD R2, 1
ADD R3, R1, R2
STORE x, R3
BR L15
L16: // exit label
```

Summary: Attribute Grammars

- Useful for expressing arbitrary cycle-free traversals over context-free parse trees
  - Synthesized and inherited attributes
  - Conditions to reject invalid parse trees
  - Evaluation order depends on attribute dependencies
- Uses: type checking and code generation
- Basic data structures (sets, maps, etc.) can be used
- The evaluation rules can call helper functions
- If functions have global effects (“side effects”), need to define when these effects happen