

Attribute Grammars

- Pagan Ch. 2.1, 2.2, 2.3, 3.2
- Stansifer Ch. 2.2, 2.3
- Slonneger and Kurtz Ch 3.1, 3.2

Formal Languages

- Basis for the design and implementation of programming languages
- **Alphabet**: finite set Σ of symbols
- **String**: finite sequence of symbols
 - Empty string ε
 - Σ^* - set of all strings over Σ (incl. ε)
 - Σ^+ - set of all non-empty strings over Σ
- **Language**: set of strings $L \subseteq \Sigma^*$

Grammars

- $G = (N, T, S, P)$
 - Finite set of **non-terminal symbols** N
 - Finite set of **terminal symbols** T
 - Starting non-terminal symbol $S \in N$
 - Finite set of **productions** P
- Production: $x \rightarrow y$
 - x is a non-empty sequence of terminals and non-terminals; y is a possibly-empty sequence of terminals and non-terminals
- Applying a production: $uxv \Rightarrow uyv$

Languages and Grammars

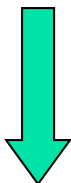
- String derivation
 - $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n$; denoted $w_1 \xRightarrow{*} w_n$
- Language generated by a grammar
 - $L(G) = \{ w \in T^* \mid S \xRightarrow{*} w \}$
- Traditional classification of languages and grammars
 - Regular \subset Context-free \subset Context-sensitive \subset Unrestricted

Regular Languages

- Generated by **regular grammars**
 - All productions are $A \rightarrow wB$ and $A \rightarrow w$
 - $A, B \in N$ and $w \in T^*$
 - Or all productions are $A \rightarrow Bw$ and $A \rightarrow w$
- e.g. $L = \{ a^n b \mid n > 0 \}$ is a regular language
 - $S \rightarrow Ab$ and $A \rightarrow a \mid Aa$
- Alternative equivalent formalisms
 - Regular expressions: e.g. a^*b for $\{ a^n b \mid n \geq 0 \}$
 - Deterministic finite automata (DFA)
 - Nondeterministic finite automata (NFA)

Lexical Analysis in Compilers

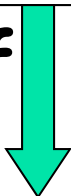
stream of
characters



w,h,i,l,e,(,a,1,5,>,b,b,),d,o,...

Lexical Analyzer (uses a regular grammar)

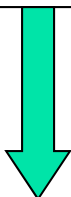
stream of
tokens



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 compiler components

Uses of Regular Languages

- Lexical analysis in compilers
 - e.g., an identifier token is a sequence of characters from the regular language **letter (letter|digit)***
 - each token is a **terminal symbol** for the context-free grammar of the parser
- Pattern matching
 - `stdsun> grep "a\+b" foo.txt`
 - Every line from `foo.txt` that contains a string from the language $L = \{ a^n b \mid n > 0 \}$
 - i.e. the language for reg. expr. a^+b

Context-Free Languages

- Subsume regular languages
 - $L = \{ a^n b^n \mid n > 0 \}$ is c.f. but not regular
- Generated by a **context-free grammar**
 - Each production: $A \rightarrow w$
 - $A \in N, w \in (N \cup T)^*$
- 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

$\langle \text{stmt} \rangle ::= \text{while } \langle \text{exp} \rangle \text{ do } \langle \text{stmt} \rangle$

| $\text{if } \langle \text{exp} \rangle \text{ then } \langle \text{stmt} \rangle$

| $\text{if } \langle \text{exp} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

| $\langle \text{exp} \rangle := \langle \text{exp} \rangle$

| $\langle \text{id} \rangle (\langle \text{exprs} \rangle)$

$\langle \text{exprs} \rangle ::= \langle \text{exp} \rangle \mid \langle \text{exprs} \rangle , \langle \text{exp} \rangle$

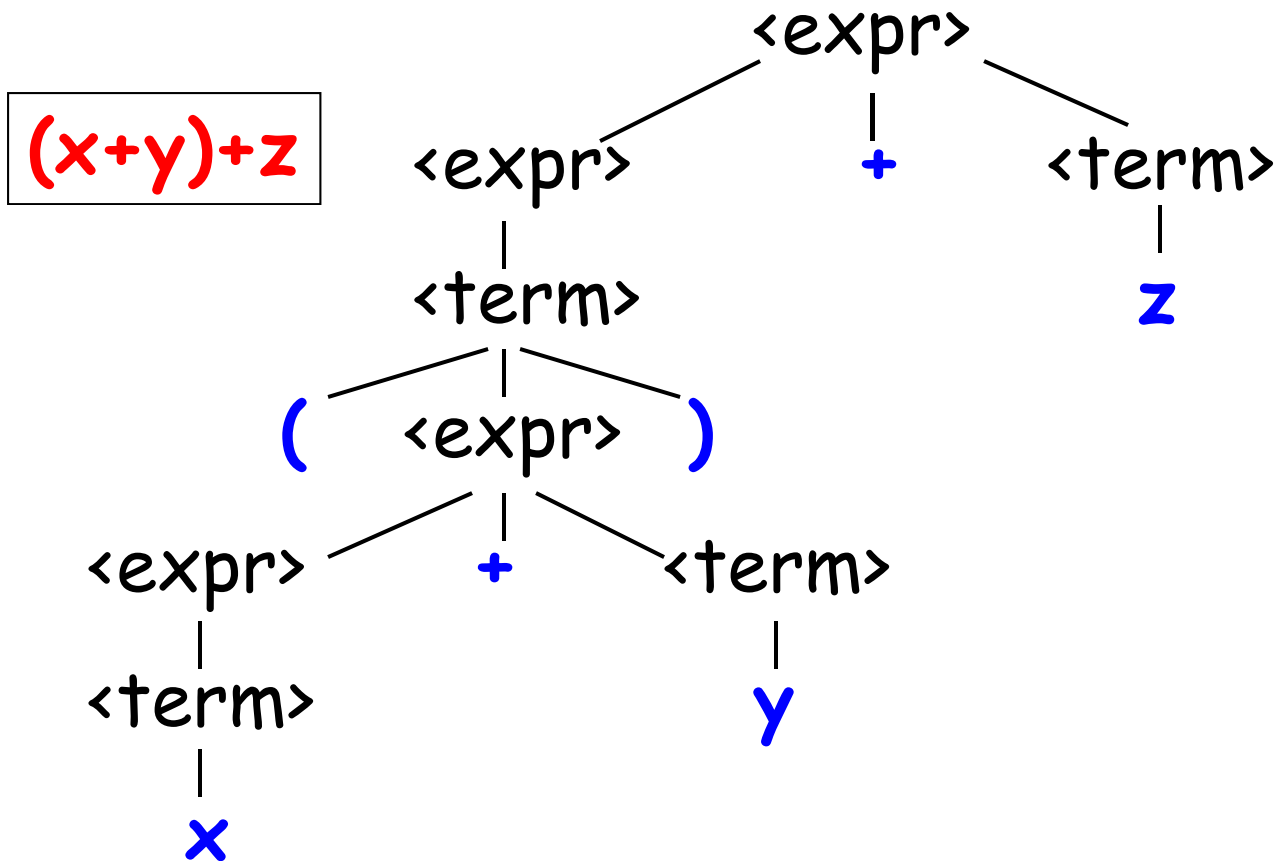
Derivation Tree

- Also called **parse tree** or **concrete syntax tree**
 - Leaf nodes: terminals
 - Inner nodes: non-terminals
 - Root: starting non-terminal of the grammar
- Describes a particular way to derive a string
 - Leaf nodes from left to right are the string
 - to get the string: depth-first traversal, following the leftmost unexplored branch

Example of a Derivation Tree

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{expr} \rangle + \langle \text{term} \rangle$

$\langle \text{term} \rangle ::= x \mid y \mid z \mid (\langle \text{expr} \rangle)$



Equivalent Derivation Sequences

The set of string derivations that are represented by the same parse tree

One derivation:

$$\begin{aligned} \langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \Rightarrow \langle \text{expr} \rangle + z \Rightarrow \\ \langle \text{term} \rangle + z &\Rightarrow (\langle \text{expr} \rangle) + z \Rightarrow \\ (\langle \text{expr} \rangle + \langle \text{term} \rangle) + z &\Rightarrow (\langle \text{expr} \rangle + y) + z \Rightarrow \\ (\langle \text{term} \rangle + y) + z &\Rightarrow (x + y) + z \end{aligned}$$

Another derivation:

$$\begin{aligned} \langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \Rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \Rightarrow \\ (\langle \text{expr} \rangle) + \langle \text{term} \rangle &\Rightarrow (\langle \text{expr} \rangle + \langle \text{term} \rangle) + \langle \text{term} \rangle \Rightarrow \\ (\langle \text{term} \rangle + \langle \text{term} \rangle) + \langle \text{term} \rangle &\Rightarrow (x + \langle \text{term} \rangle) + \langle \text{term} \rangle \Rightarrow \\ (x + y) + \langle \text{term} \rangle &\Rightarrow (x + y) + z \end{aligned}$$

Many more ...

Ambiguous Grammars

- For some string, there are multiple different parse trees
- An ambiguous grammar gives more freedom to the compiler writer
 - e.g. for code optimizations
- For real-world programming languages, we typically have non-ambiguous grammars
 - To remove ambiguity: add non-terminals; or add operator precedence and associativity; or use an attribute grammar (more later ...)

Elimination of Ambiguity

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle \mid$
 $(\langle \text{expr} \rangle) \mid \text{id}$

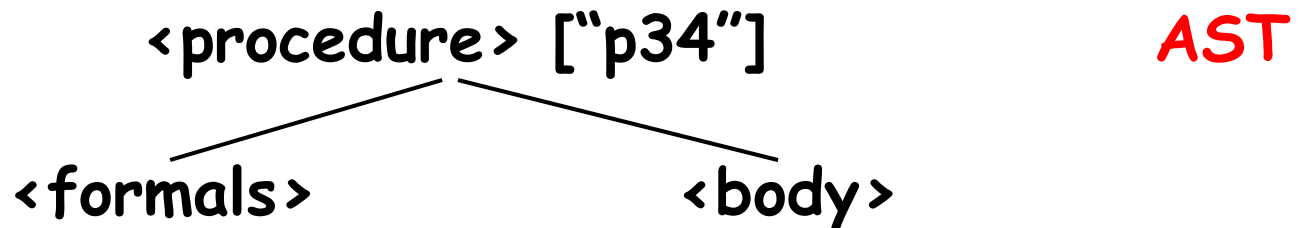
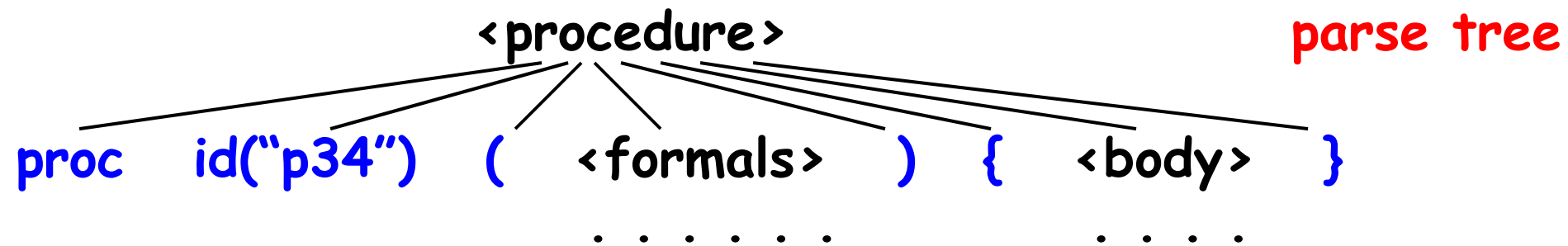
1. Prove that this grammar is ambiguous
2. Create an equivalent non-ambiguous grammar with the appropriate precedence and associativity
 - $*$ has higher precedence than $+$
 - both are left-associative

Example: what is the parse tree for
 $a + b * (c+d) * e$?

Abstract Syntax Trees (AST)

- A "simplified" version of a concrete syntax tree, without loss of information

`<procedure> ::= proc id (<formals>) { <body> }`



Use of Context-Free Grammars

- **Syntax** of a programming language
 - e.g. Java: Chapter 18 of the language specification (JLS) defines a grammar
 - Terminals: identifiers, keywords, literals, separators, operators
 - Starting non-terminal: **CompilationUnit**
- Implementation of a **parser** in a compiler
 - e.g. the JLS grammar (Ch. 18) is used by the parser inside the **javac** compiler

Limitations of Context-Free Grammars

- Cannot represent semantics
 - e.g. "every variable used in a statement should be declared earlier in the code"; or "the use of a variable should conform to its type" (type checking)
 - Need to allow only programs that satisfy certain **context-sensitive conditions**
- Cannot be used to generate things other than parse trees
 - e.g., 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
- Represents a **traversal of the parse tree** and the computation of some information during that traversal

Structure of an Attribute Grammar

- Underlying context-free grammar
- For each terminal and non-terminal: zero, one, or more **attributes**
 - For each attribute: domain of possible values
- Set of **evaluation rules** for each production
- Set of boolean **conditions** for attribute values

Example

- $L = \{ a^n b^n c^n \mid n > 0 \}$; not context-free
- BNF
 - $\langle \text{start} \rangle ::= \langle A \rangle \langle B \rangle \langle C \rangle$ $\langle A \rangle ::= a \mid a \langle A \rangle$
 - $\langle B \rangle ::= b \mid b \langle B \rangle$ $\langle C \rangle ::= c \mid c \langle C \rangle$
- Attributes
 - N_a : associated with $\langle A \rangle$
 - N_b : associated with $\langle B \rangle$
 - N_c : associated with $\langle C \rangle$
 - Value domain for N_a, N_b, N_c : integer values

Example

- Evaluation rules (similar for $\langle B \rangle$, $\langle C \rangle$)

$\langle A \rangle ::= a$

$\langle A \rangle.Na := 1$

| $a\langle A \rangle_2$

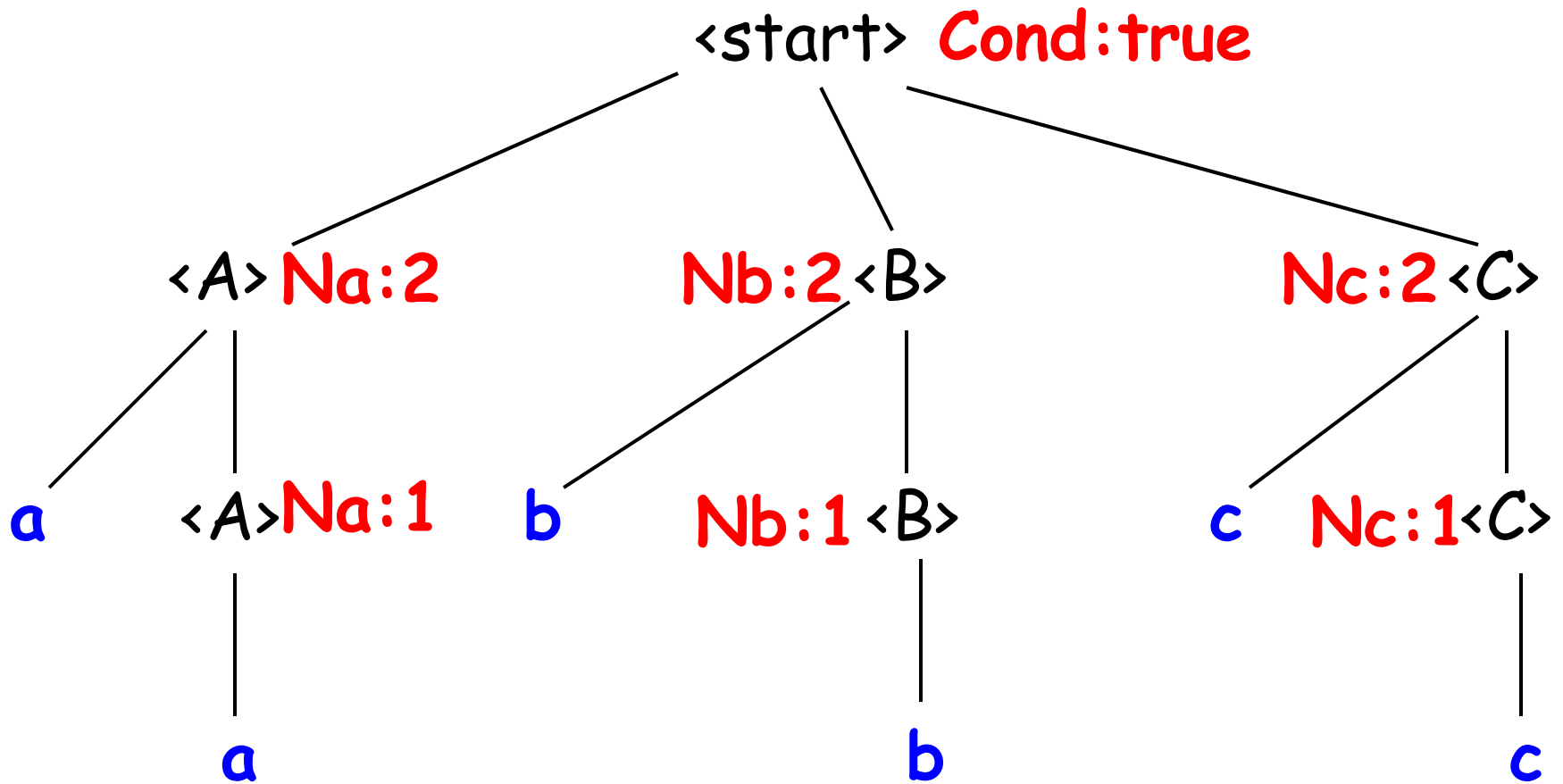
$\langle A \rangle.Na := 1 + \langle A \rangle_2.Na$

- Conditions

$\langle \text{start} \rangle ::= \langle A \rangle \langle B \rangle \langle C \rangle$

Cond: $\langle A \rangle.Na = \langle B \rangle.Nb = \langle C \rangle.Nc$

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 terminal or non-terminal in the node
- Some nodes have boolean conditions
- **Valid** parse tree
 - Attribute values conform to the evaluation rules
 - All boolean conditions are true

Synthesized vs. Inherited Attributes

- Each terminal and non-terminal X : disjoint sets of attributes $Syn(X)$ and $Inh(X)$
 - $Attr(X) = Syn(X) \cup Inh(X)$
- Production $X ::= Y_1 Y_2 \dots Y_n$
- Each attribute in $Syn(X)$ is defined from $Attr(Y_1) \cup \dots \cup Attr(Y_n) \cup Attr(X)$
- Each attribute in $Inh(Y_k)$ is defined from $Attr(X) \cup Attr(Y_1) \cup \dots \cup Attr(Y_n)$

Complete Evaluation Rules

- Synthesized attribute associated with N:
 - Each alternative in " $N ::= \dots$ " should contain a rule for evaluating the attribute
- Inherited attribute associated with N:
 - For every occurrence of N in " $\dots ::= \dots N \dots$ " there must be a rule for evaluating the attribute
- Whenever you create an attribute grammar (in homeworks/exams), make sure it satisfies these requirements

Example: Binary Numbers

- Context-free grammar

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

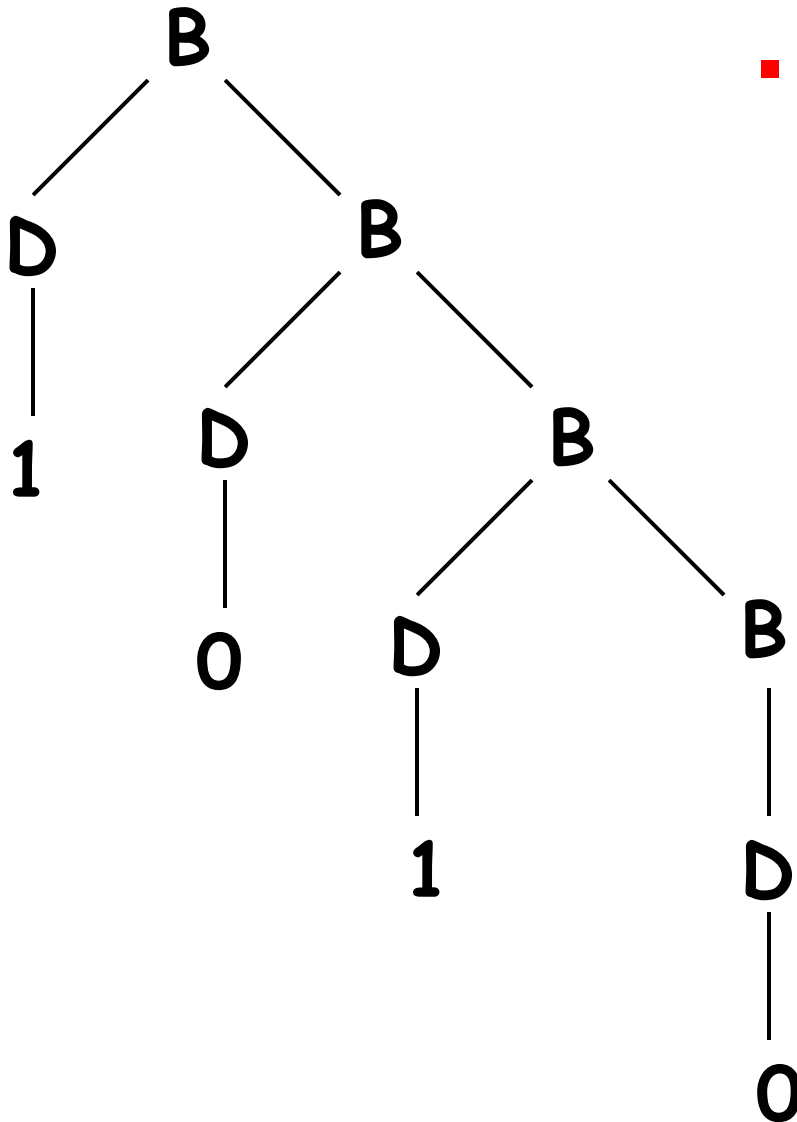
$\langle B \rangle ::= \langle D \rangle \langle B \rangle$

$\langle D \rangle ::= 0$

$\langle D \rangle ::= 1$

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

BNF Parse Tree for Input 1010



- Add attributes
 - $\langle B \rangle$: synthesized **val**
 - $\langle B \rangle$: synthesized **pos**
 - $\langle D \rangle$: inherited **pow**
 - $\langle D \rangle$: **val**

Example: Binary Numbers

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

$\langle B \rangle.\text{pos} := 1$

$\langle B \rangle.\text{val} := \langle D \rangle.\text{val}$

$\langle D \rangle.\text{pow} := 0$

$\langle B \rangle_1 ::= \langle D \rangle \langle B \rangle_2$

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

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

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

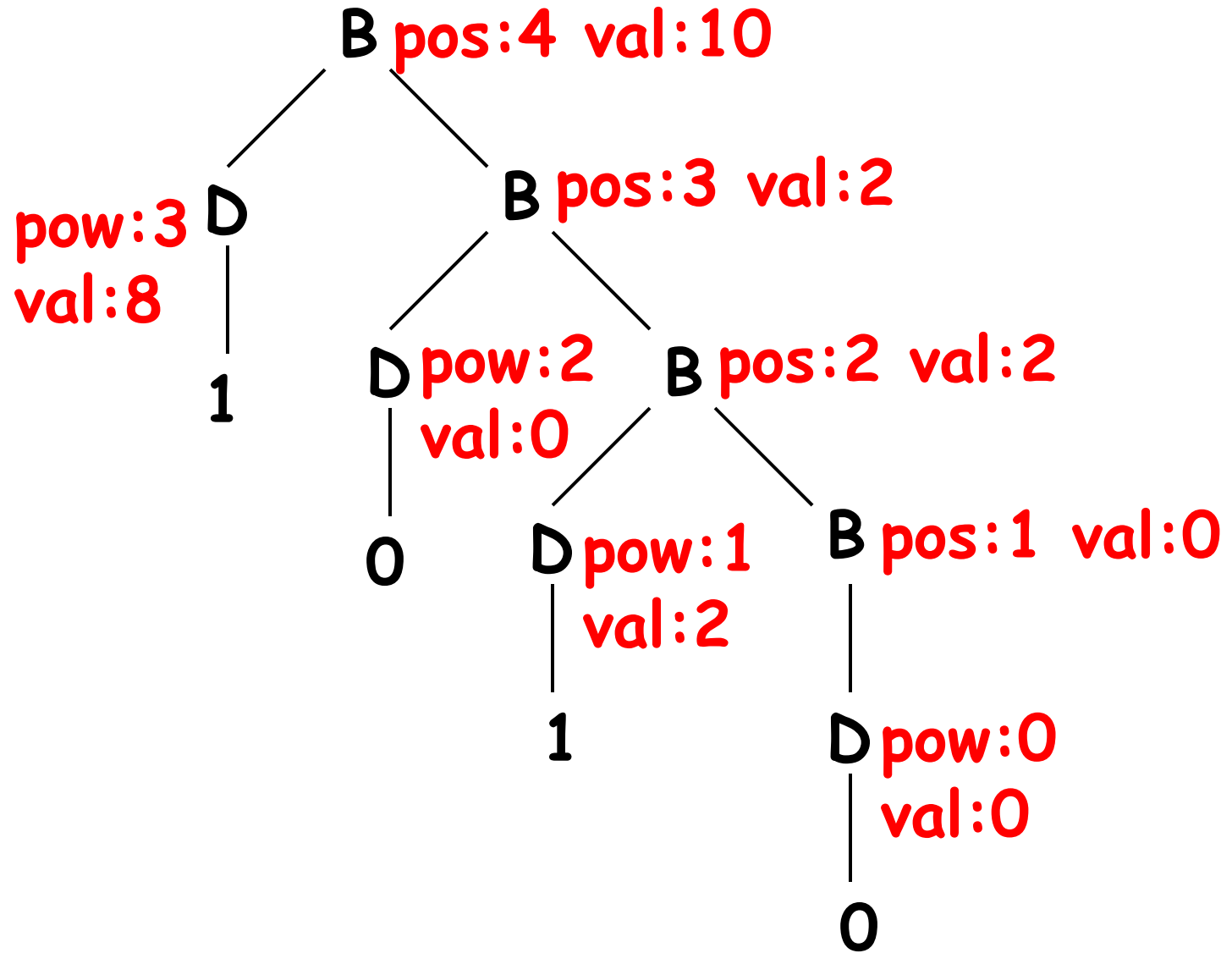
$\langle D \rangle ::= 0$

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

$\langle D \rangle ::= 1$

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

Evaluated Parse Tree



Example: Expression Language

- Problem: evaluate an expression

$\langle S \rangle ::= \langle E \rangle$

$\langle E \rangle ::= 0 \mid 1 \mid \langle I \rangle \mid (\langle E \rangle + \langle E \rangle) \mid$
 $\text{let } \langle I \rangle = \langle E \rangle \text{ in } \langle E \rangle \text{ end}$

$\langle I \rangle ::= \text{id}$

- Attributes
 - $\langle I \rangle$: synthesized **name**
 - $\langle E \rangle$: synthesized **val**, inherited **env**
 - **id**: synthesized **lexval**, represents the string that the lexical analysis puts inside token **id**

Attribute Grammar

$\langle S \rangle ::= \langle E \rangle$

$\langle E \rangle.\text{env} := \emptyset$

$\langle E \rangle ::= 0$

$\langle E \rangle.\text{val} := 0$

$\langle E \rangle ::= 1$

"helper" function

$\langle E \rangle.\text{val} := 1$

$\langle E \rangle ::= \langle I \rangle$

$\langle E \rangle.\text{val} := \text{lookup}(\langle E \rangle.\text{env}, \langle I \rangle.\text{name})$

$\langle I \rangle ::= \text{id}$

$\langle I \rangle.\text{name} := \text{id}.\text{lexval}$

Attribute Grammar

$\langle E \rangle_1 ::= (\langle E \rangle_2 + \langle E \rangle_3)$

$\langle E \rangle_1.val := \langle E \rangle_2.val + \langle E \rangle_3.val$

$\langle E \rangle_2.env := \langle E \rangle_1.env$

$\langle E \rangle_3.env := \langle E \rangle_1.env$

$\langle E \rangle_1 ::= \text{let } \langle I \rangle = \langle E \rangle_2 \text{ in } \langle E \rangle_3 \text{ end}$

$\langle E \rangle_2.env := \langle E \rangle_1.env$

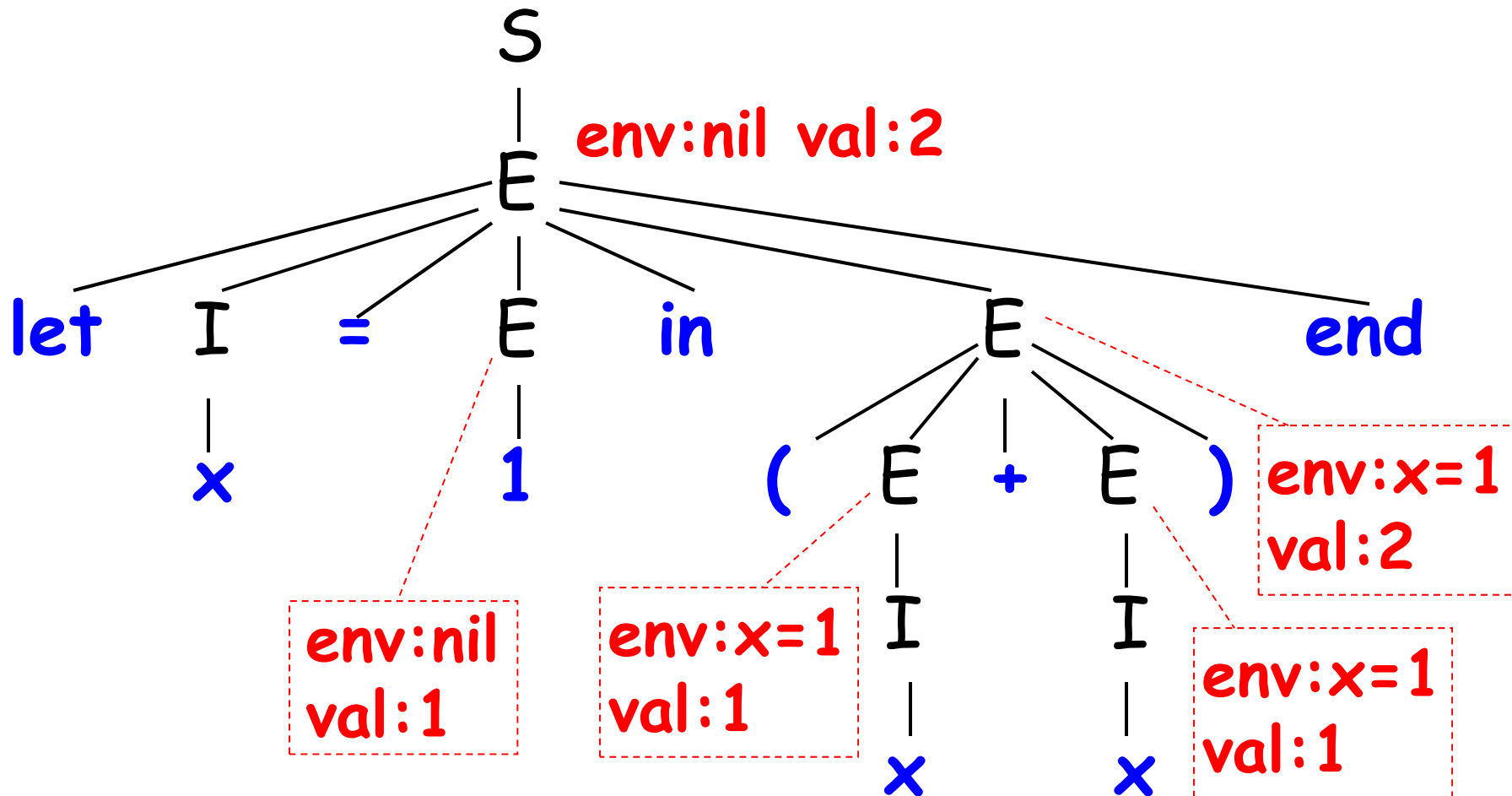
$\langle E \rangle_3.env :=$
 $\text{update}(\langle E \rangle_1.env, \langle I \rangle.name, \langle E \rangle_2.val)$

$\langle E \rangle_1.val := \langle E \rangle_3.val$

creates a
"fresh"
environment;
does not
affect
 $\langle E \rangle_1.env$

Example

- Evaluation of `let x = 1 in (x+x) end`



Another Example

- How about the evaluation of

let $x = 1$ in

let $x = (x+1)$ in

$(x+2)$

end

end

More than Context-Free Power

- $L = \{ a^n b^n c^n \mid n > 0 \}$
 - Unlike $L = \{ a^n b^n \mid n > 0 \}$, here we need explicit counting
- $L = \{ w c w \mid w \in \{a,b\}^* \}$
 - The “flavor” of checking whether identifiers are declared before their uses
 - Cannot be done with a context-free grammar
 - requires context-sensitive conditions
- Syntax analysis (i.e., parser) cannot handle semantic properties

"Fixing" Context-Free Grammars

- $\langle S \rangle_1 ::= a \mid b \mid \langle S \rangle_2 \langle S \rangle_3$
 - Ambiguous: e.g. abab can be parsed as if it were (ab)(ab) or $a(b(a(b)))$ or ...
 - Suppose we want to make the grammar unambiguous, and associate to the right
- One approach: $\langle X \rangle ::= a \mid b$ and $\langle S \rangle ::= \langle X \rangle \langle S \rangle$
- Another approach: attribute **len** for S
 - $\langle S \rangle ::= a \mid b \quad \langle S \rangle.\text{len} := 1$
 - $\langle S \rangle_1 ::= \langle S \rangle_2 \langle S \rangle_3 \quad \langle S \rangle_1.\text{len} := \langle S \rangle_2.\text{len} + \langle S \rangle_3.\text{len}$
Cond: $\langle S \rangle_2.\text{len} = 1$

"Fixing" Context-Free Grammars

- $\langle E \rangle_1 ::= \langle \text{Num} \rangle \mid \langle E \rangle_2 + \langle E \rangle_3$ - ambiguous
 - Want left-associativity:
 - $a+b+c$ as if it were $(a+b)+c$, not $a+(b+c)$
 - Change BNF: $\langle E \rangle_1 ::= \langle \text{Num} \rangle \mid \langle E \rangle_2 + \langle \text{Num} \rangle$
 - Alternative: attribute grammar
 - Attribute n : number of $+$
 - $\langle E \rangle_1 ::= \langle \text{Num} \rangle$ $\langle E \rangle_1.n := 0$
 - $\langle E \rangle_1 ::= \langle E \rangle_2 + \langle E \rangle_3$ $\langle E \rangle_1.n := 1 + \langle E \rangle_2.n + \langle E \rangle_3.n$
- Cond: ?

Attribute Evaluation: Dependence Graph

- $\langle A \rangle.x := \langle B \rangle.y + \langle C \rangle.z$
 - $\langle A \rangle.x$ depends on $\langle B \rangle.y$
 - $A.x \leftarrow B.y$ dependence edge
 - $\langle A \rangle.x$ depends on $\langle C \rangle.z$
 - $A.x \leftarrow C.z$ dependence edge
- $\langle A \rangle_1.x := \langle A \rangle_2.x$
 - $\langle A \rangle_1.x$ depends on $\langle A \rangle_2.x$
 - $A_1.x \leftarrow A_2.x$ dependence edge

Attribute Grammar Evaluation

- Given: parse tree for parsed input with attributes attached to tree nodes
- Find evaluation order of attributes
 - Build **dependence graph**
 - Node: a pair (parse tree node, attribute)
 - Complain about cycles in the graph
 - Topologically sort the graph
- Evaluate the attributes in sorted order

Example: Binary Numbers

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

$\langle B \rangle.\text{pos} := 1$

$\langle B \rangle.\text{val} := \langle D \rangle.\text{val}$

$\langle D \rangle.\text{pow} := 0$

$\langle B \rangle_1 ::= \langle D \rangle \langle B \rangle_2$

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

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

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

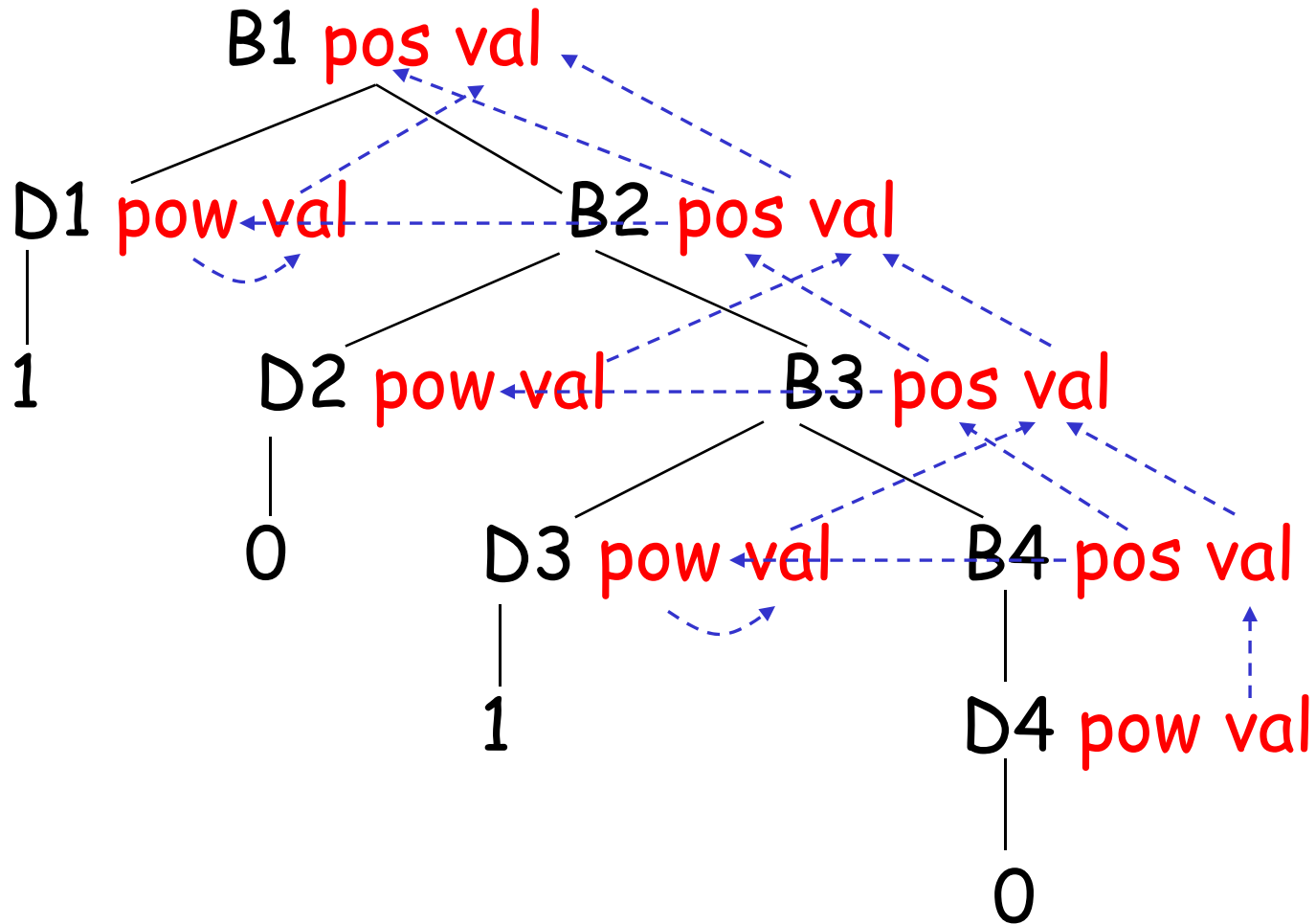
$\langle D \rangle ::= 0$

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

$\langle D \rangle ::= 1$

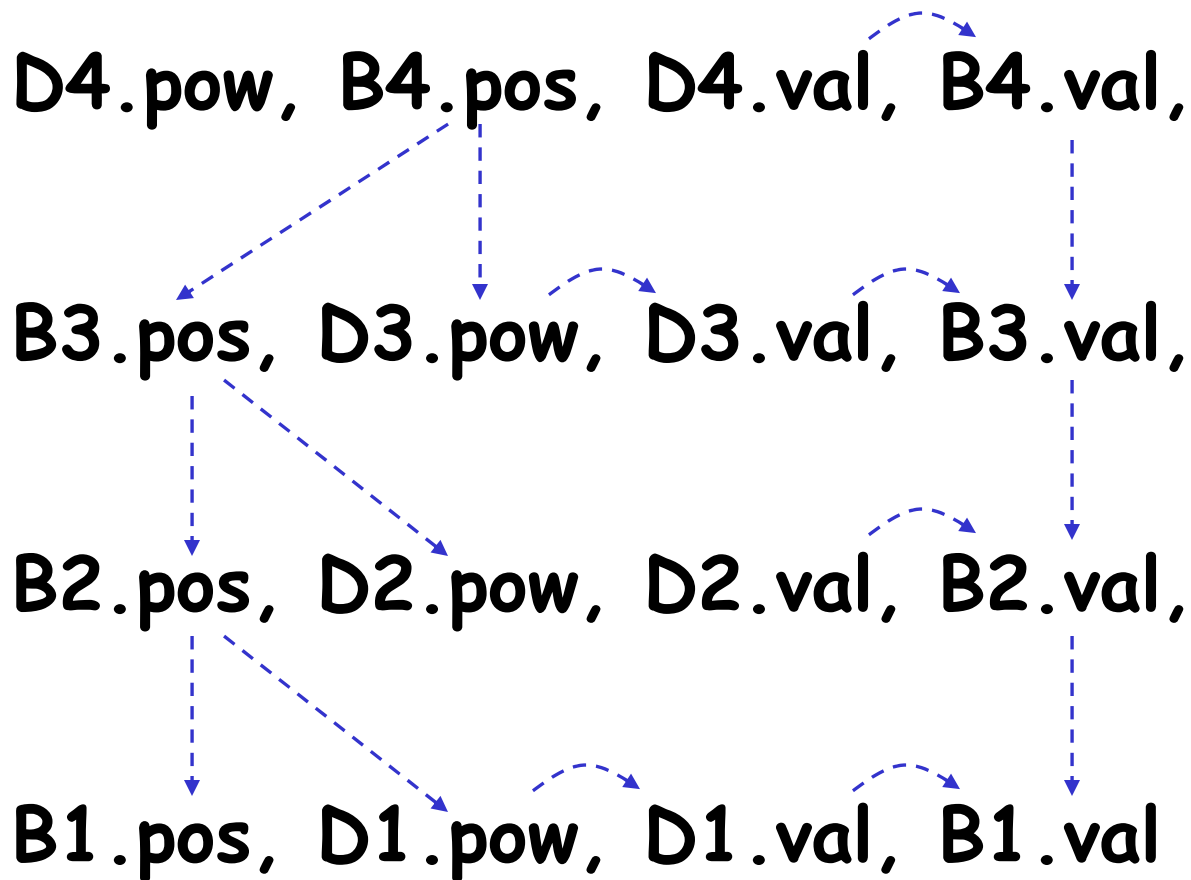
$\langle D \rangle.\text{val} := 2^{\langle D \rangle.\text{pow}}$

Dependence Graph for Binary Numbers



Sort the Graph

- Topological sort: x is "smaller" than y iff $x \rightarrow y$

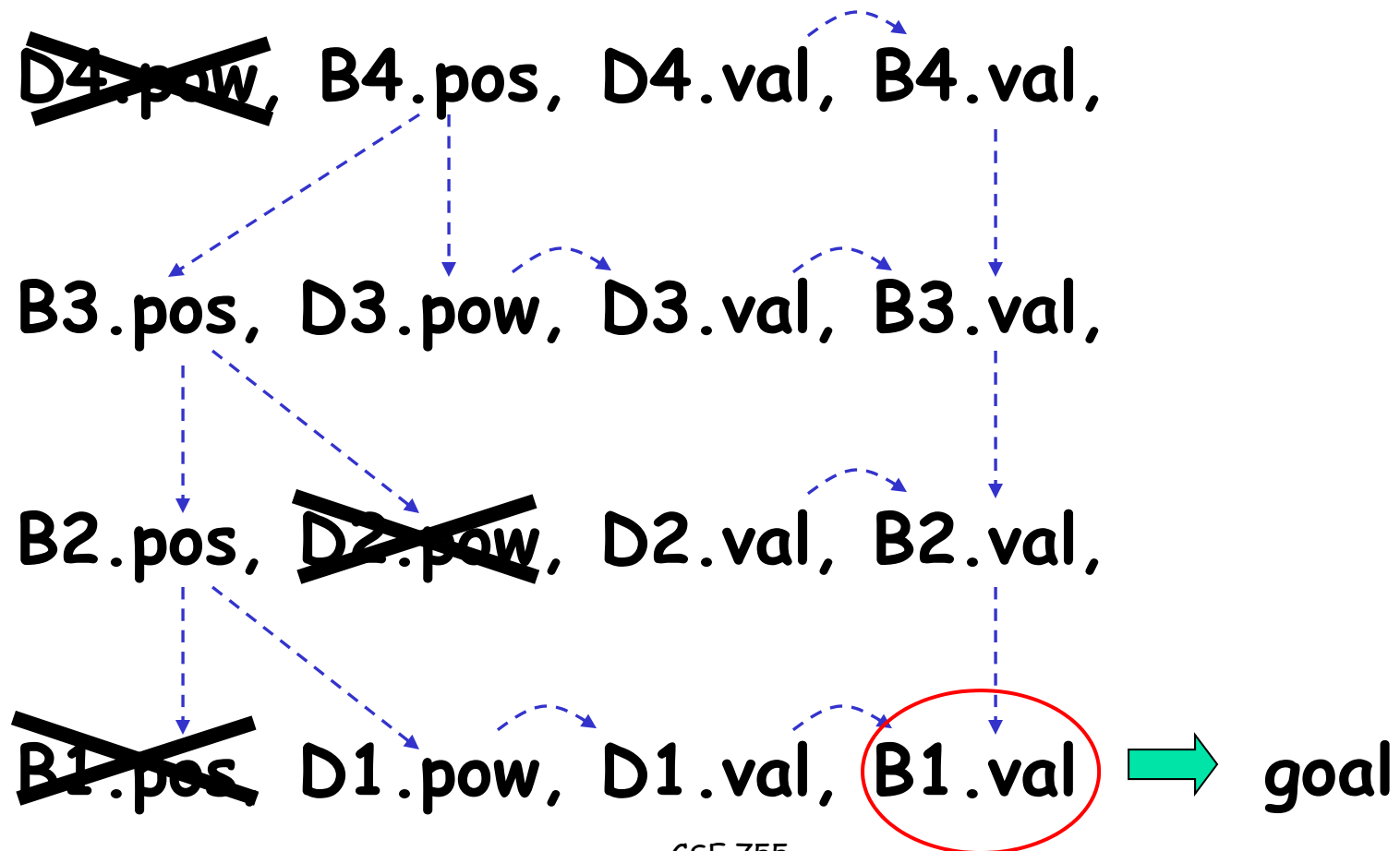


Cycles

- The notion of “topological sort” only makes sense for directed acyclic graphs
- Cycles in the dependence graph: recursive dependencies
 - There are approaches to solve meaningful recursive systems of equations
 - e.g., fixed-point computation
- In this course we will disallow cycles
 - No cyclic dependencies in your solutions in exams and homeworks

Optimizations

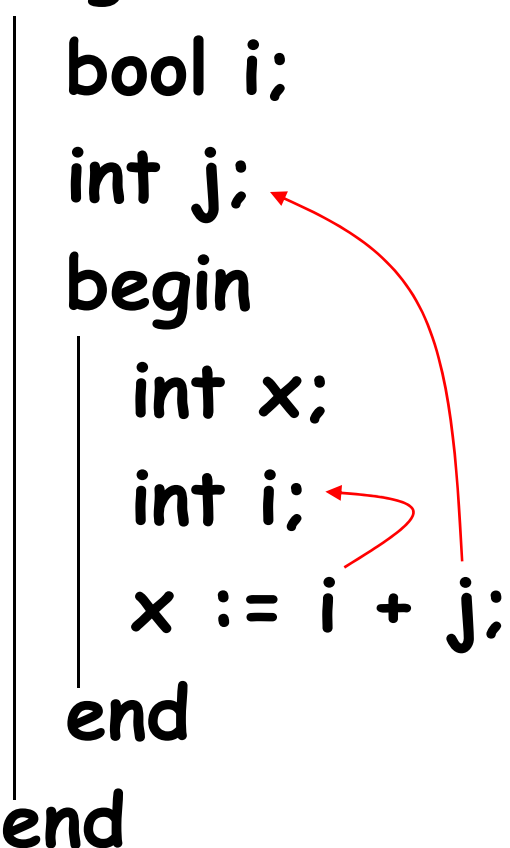
- Remove nodes based on reachability



Long Example 1: Type Checking

- Given: program with declarations
- Check that variables are used correctly

```
begin
  bool i;
  int j;
  begin
    int x;
    int i;
    x := i + j;
  end
end
```



Blocks, Declarations, Statements

$\langle \text{prog} \rangle ::= \langle \text{block} \rangle$

$\langle \text{block} \rangle ::= \text{begin } \langle \text{declist} \rangle ; \langle \text{stmtlist} \rangle \text{end}$

$\langle \text{stmtlist} \rangle ::= \langle \text{stmt} \rangle$

$| \langle \text{stmt} \rangle ; \langle \text{stmtlist} \rangle$

$\langle \text{stmt} \rangle ::= \langle \text{assign} \rangle$

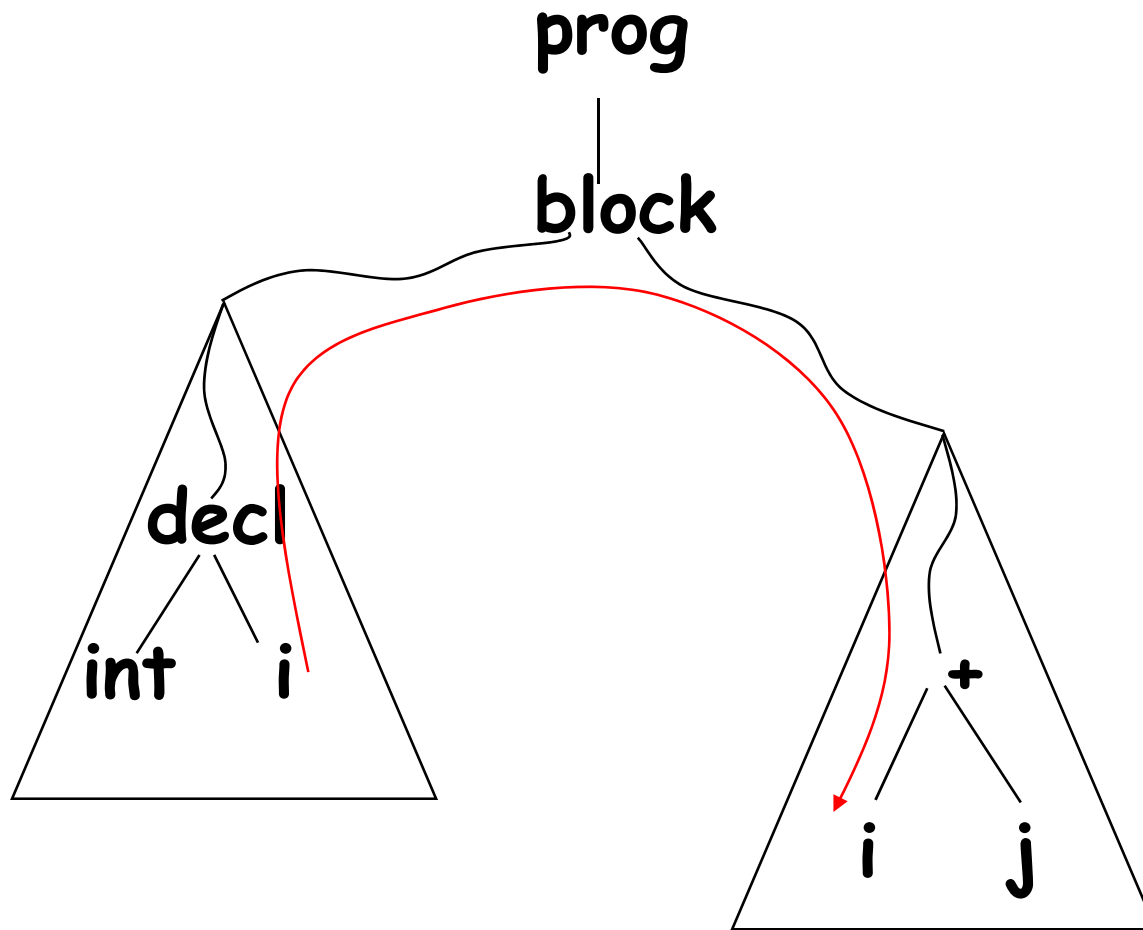
$| \langle \text{block} \rangle$

$| \dots$

Problem

- Check types of variables: int and bool
- For nested blocks, use the innermost declaration
- Check parameter types and return type for functions
 - For simplicity: all functions will return int
- Type checking: a form of semantic checking that a compiler will perform on the parse tree (or the AST)

Parse Tree



Data Structure

- Use a **stack** of **symbol tables**
 - symbol table: set of pairs (name,type)
- Build a symbol table for the declarations in a begin-end block
 - as a synthesized attribute **tbl**
- “Propagate” a stack of symbol tables to statements and expressions
 - propagate downwards on the parse tree as an inherited attribute **alltbl**

Type Checking

- Pass around a stack of symbol tables

```
begin
```

```
  bool i;
```

```
  int j;
```

```
  begin
```

```
    int x;
```

```
    int i;
```

```
    x := i + j;
```

```
  end
```

```
end
```

```
[  
  {"x",INT}, {"i",INT},  
  {"i",BOOL}, {"j",INT}  
]
```

bottom of stack

Context-Free Grammar - Part 1

$\langle \text{prog} \rangle ::= \langle \text{block} \rangle$

$\langle \text{block} \rangle ::= \text{begin } \langle \text{declist} \rangle ; \langle \text{stmtlist} \rangle \text{end}$

$\langle \text{declist} \rangle ::= \langle \text{decl} \rangle \mid \langle \text{decl} \rangle ; \langle \text{declist} \rangle$

$\langle \text{decl} \rangle ::= \text{int id} \mid \text{bool id} \mid$
 $\text{fun id } (\langle \text{formalslist} \rangle) : \text{int} = \langle \text{block} \rangle$

$\langle \text{stmtlist} \rangle ::= \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmtlist} \rangle$

$\langle \text{stmt} \rangle ::= \langle \text{assign} \rangle \mid \langle \text{block} \rangle \mid$
 $\text{if } \langle \text{boolexpr} \rangle$
 $\text{then } \langle \text{stmtlist} \rangle \text{ else } \langle \text{stmtlist} \rangle \text{ fi}$

Context-Free Grammar - Part 2

$\langle \text{assign} \rangle ::= \text{id} := \langle \text{boolexp} \rangle \mid$
 $\text{id} := \langle \text{intexp} \rangle$

$\langle \text{boolexp} \rangle ::= \text{true} \mid \text{false} \mid \text{id}$

$\langle \text{intexp} \rangle ::= \text{intconst} \mid \text{id} \mid$
 $\langle \text{intexp} \rangle + \langle \text{intexp} \rangle \mid$
 $\text{id} (\langle \text{actualslist} \rangle)$

Ignore Functions for Now

$\langle \text{prog} \rangle ::= \langle \text{block} \rangle$

$\langle \text{block} \rangle.\text{alltbl} := \text{emptystack}$

$\langle \text{block} \rangle ::= \text{begin } \langle \text{declist} \rangle ; \langle \text{stmtlist} \rangle \text{ end}$

$\langle \text{stmtlist} \rangle.\text{alltbl} :=$

*creates
a "fresh"
stack*

$\rightarrow \text{push}(\langle \text{declist} \rangle.\text{tbl}, \langle \text{block} \rangle.\text{alltbl})$

$\langle \text{stmtlist} \rangle_1 ::= \langle \text{stmt} \rangle$

$\langle \text{stmt} \rangle.\text{alltbl} := \langle \text{stmtlist} \rangle_1.\text{alltbl}$

$| \langle \text{stmt} \rangle ; \langle \text{stmtlist} \rangle_2$

$\langle \text{stmt} \rangle.\text{alltbl} := \langle \text{stmtlist} \rangle_1.\text{alltbl}$

$\langle \text{stmtlist} \rangle_2.\text{alltbl} := \langle \text{stmtlist} \rangle_1.\text{alltbl}$

Declarations

$\langle \text{declist} \rangle_1 ::= \langle \text{decl} \rangle$

$\langle \text{declist} \rangle_1.\text{tbl} := \langle \text{decl} \rangle.\text{tbl}$

| $\langle \text{decl} \rangle ; \langle \text{declist} \rangle_2$

$\langle \text{declist} \rangle_1.\text{tbl} :=$

$\langle \text{decl} \rangle.\text{tbl} \cup \langle \text{declist} \rangle_2.\text{tbl}$

Cond:

$\text{ids}(\langle \text{decl} \rangle.\text{tbl}) \cap \text{ids}(\langle \text{declist} \rangle_2.\text{tbl}) = \emptyset$

`ids` is a function that returns the set of all names in a table;

used to check for duplicates: e.g. `int x; int x;`

Declarations

$\langle \text{decl} \rangle ::= \text{int id}$

$\langle \text{decl} \rangle.\text{tbl} := \{ (\text{id.lexval}, \text{INT}) \}$

| bool id

$\langle \text{decl} \rangle.\text{tbl} := \{ (\text{id.lexval}, \text{BOOL}) \}$

| $\text{fun id} (\langle \text{formalstlist} \rangle) : \text{int} = \langle \text{block} \rangle$



ignore for now

Statements

$\langle \text{stmtlist} \rangle_1 ::= \langle \text{stmt} \rangle$

$\langle \text{stmt} \rangle.\text{alltbl} := \langle \text{stmtlist} \rangle_1.\text{alltbl}$

| $\langle \text{stmt} \rangle ; \langle \text{stmtlist} \rangle_2$

$\langle \text{stmt} \rangle.\text{alltbl} := \langle \text{stmtlist} \rangle_1.\text{alltbl}$

$\langle \text{stmtlist} \rangle_2.\text{alltbl} := \langle \text{stmtlist} \rangle_1.\text{alltbl}$

$\langle \text{stmt} \rangle ::= \langle \text{assign} \rangle$

$\langle \text{assign} \rangle.\text{alltbl} := \langle \text{stmt} \rangle.\text{alltbl}$

| $\langle \text{block} \rangle$

$\langle \text{block} \rangle.\text{alltbl} := \langle \text{stmt} \rangle.\text{alltbl}$

| **if** $\langle \text{boolexp} \rangle$ **then** $\langle \text{stmtlist} \rangle$ **else** ...

$\langle \text{boolexp} \rangle.\text{alltbl} := \langle \text{stmt} \rangle.\text{alltbl}$

$\langle \text{stmtlist} \rangle.\text{alltbl} := \langle \text{stmt} \rangle.\text{alltbl}$

Statements

$\langle \text{assign} \rangle ::= \text{id} := \langle \text{intexp} \rangle$

$\langle \text{intexp} \rangle.\text{alltbl} := \langle \text{assign} \rangle.\text{alltbl}$

the "last"
type on
the stack

Cond:

$\text{typeof}(\text{id}.\text{lexval}, \langle \text{assign} \rangle.\text{alltbl}) = \text{INT}$

| $\text{id} := \langle \text{boolexp} \rangle$

$\langle \text{boolexp} \rangle.\text{alltbl} := \langle \text{assign} \rangle.\text{alltbl}$

Cond:

$\text{typeof}(\text{id}.\text{lexval}, \langle \text{assign} \rangle.\text{alltbl}) = \text{BOOL}$

Expressions

$\langle \text{intexp} \rangle_1 ::= \text{intconst}$

| id

Cond: $\text{typeof}(\text{id.lexval},$
 $\langle \text{intexp} \rangle_1.\text{alltbl}) = \text{INT}$

| $\langle \text{intexp} \rangle_2 + \langle \text{intexp} \rangle_3$

$\langle \text{intexp} \rangle_2.\text{alltbl} := \langle \text{intexp} \rangle_1.\text{alltbl}$

$\langle \text{intexp} \rangle_3.\text{alltbl} := \langle \text{intexp} \rangle_1.\text{alltbl}$

$\langle \text{boolexp} \rangle ::= \text{true}$

| false

| id

Cond: $\text{typeof}(\text{id.lexval},$
 $\langle \text{boolexp} \rangle.\text{alltbl}) = \text{BOOL}$

Declarations

$\langle \text{decl} \rangle ::= \text{int id}$

$\langle \text{decl} \rangle.\text{tbl} := \{ (\text{id.lexval}, \text{INT}) \}$

| bool id

$\langle \text{decl} \rangle.\text{tbl} := \{ (\text{id.lexval}, \text{BOOL}) \}$

| $\text{fun id} (\langle \text{formalslist} \rangle) : \text{int} = \langle \text{block} \rangle$

$\langle \text{decl} \rangle.\text{tbl} :=$

$\{ (\text{id.lexval},$

$\text{FUN}(\langle \text{formalslist} \rangle.\text{types}, \text{INT})) \}$

synthesized attr:
ordered list of INT/BOOL



$\langle \text{block} \rangle.\text{alltbl} \rightarrow$ what are we going to do here?

More Propagation of Information

- Declarations: add inherited **alltbl**

$\langle \text{block} \rangle ::= \text{begin } \langle \text{declist} \rangle ; \langle \text{stmtlist} \rangle \text{end}$

$\langle \text{stmtlist} \rangle.\text{alltbl} :=$

$\text{push}(\langle \text{declist} \rangle.\text{tbl}, \langle \text{block} \rangle.\text{alltbl})$

$\langle \text{declist} \rangle.\text{alltbl} :=$

$\text{push}(\langle \text{declist} \rangle.\text{tbl}, \langle \text{block} \rangle.\text{alltbl})$

We first gather the declarations in $\langle \text{declist} \rangle$, and then we use them to check the blocks "embedded" in $\langle \text{declist} \rangle$

Declarations

$\langle \text{declist} \rangle_1 ::= \langle \text{decl} \rangle$

$\langle \text{declist} \rangle_1.\text{tbl} := \langle \text{decl} \rangle.\text{tbl}$

$\langle \text{decl} \rangle.\text{alltbl} := \langle \text{declist} \rangle_1.\text{alltbl}$

| $\langle \text{decl} \rangle ; \langle \text{declist} \rangle_2$

$\langle \text{declist} \rangle_1.\text{tbl} :=$

$\langle \text{decl} \rangle.\text{tbl} \cup \langle \text{declist} \rangle_2.\text{tbl}$

$\langle \text{decl} \rangle.\text{alltbl} := \langle \text{declist} \rangle_1.\text{alltbl}$

$\langle \text{declist} \rangle_2.\text{alltbl} := \langle \text{declist} \rangle_1.\text{alltbl}$

Cond:

$\text{ids}(\langle \text{decl} \rangle.\text{tbl}) \cap \text{ids}(\langle \text{declist} \rangle_2.\text{tbl}) = \emptyset$

Declarations Again

$\langle \text{decl} \rangle ::= \text{int id}$

$\langle \text{decl} \rangle.\text{tbl} := \{ (\text{id}.\text{lexval}, \text{INT}) \}$

| bool id

$\langle \text{decl} \rangle.\text{tbl} := \{ (\text{id}.\text{lexval}, \text{BOOL}) \}$

| $\text{fun id} (\langle \text{formalslist} \rangle) : \text{int} = \langle \text{block} \rangle$

$\langle \text{decl} \rangle.\text{tbl} :=$

$\{ (\text{id}.\text{lexval},$

$\text{FUN}(\langle \text{formalslist} \rangle.\text{types}, \text{INT})) \}$

$\langle \text{block} \rangle.\text{alltbl} :=$

$\text{push}(\langle \text{formalslist} \rangle.\text{tbl}, \langle \text{decl} \rangle.\text{alltbl})$

Type-Checking Function Bodies

- Is the following code legal?

```
begin
```

```
  fun f (int i): int =
```

```
    begin
```

```
      ... g(5) ...
```

```
    end;
```

```
  fun g (int j): int =
```

```
    begin
```

```
      ...
```

```
    end; ...
```

```
end
```

Function Call

$\langle \text{intexp} \rangle ::= \text{id} (\langle \text{actualslist} \rangle)$

redundant
if we have
only int return
types

Cond: $\text{typeof}(\text{id.lexval},$
 $\langle \text{intexp} \rangle.\text{alltbl}) = \text{FUN}$

Cond: $\text{rettype}(\text{id.lexval},$
 $\langle \text{intexp} \rangle.\text{alltbl}) = \text{INT}$

list of $\langle \text{intexp} \rangle$
and $\langle \text{boolexp} \rangle$

$\langle \text{actualslist} \rangle.\text{expTypes} :=$
 $\text{paramtypes}(\text{id.lexval},$
 $\langle \text{intexp} \rangle.\text{alltbl})$

$\langle \text{actualslist} \rangle.\text{alltbl} :=$
 $\langle \text{intexp} \rangle.\text{alltbl}$

Long Example 2: Code Generation

- Given: parse tree for a simple program (after type checking)
- Translate to assembly code
- The evaluation rules of the attribute grammar generate the assembly code

Simple Imperative Language (IMP)

$\langle c \rangle_1 ::= \text{skip} \mid \langle \text{assign} \rangle \mid \langle c \rangle_2 ; \langle c \rangle_3$
 $\mid \text{if } \langle \text{be} \rangle \text{ then } \langle c \rangle_2 \text{ else } \langle c \rangle_3$
 $\mid \text{while } \langle \text{be} \rangle \text{ do } \langle c \rangle_2$

$\langle \text{assign} \rangle ::= \text{id} := \langle \text{ae} \rangle$

$\langle \text{ae} \rangle_1 ::= \text{id} \mid \text{intconst} \mid \langle \text{ae} \rangle_2 + \langle \text{ae} \rangle_3$
 $\mid \langle \text{ae} \rangle_2 - \langle \text{ae} \rangle_3 \mid \langle \text{ae} \rangle_2 * \langle \text{ae} \rangle_3$

$\langle \text{be} \rangle_1 ::= \text{true} \mid \text{false}$
 $\mid \langle \text{ae} \rangle_1 = \langle \text{ae} \rangle_2 \mid \langle \text{ae} \rangle_1 < \langle \text{ae} \rangle_2$
 $\mid \neg \langle \text{be} \rangle_2 \mid \langle \text{be} \rangle_2 \wedge \langle \text{be} \rangle_3$
 $\mid \langle \text{be} \rangle_2 \vee \langle \text{be} \rangle_3$

Assembly Language

- Processor with a single register ("accumulator")
- **LOAD x**: copy the value of memory location (variable) x into the accumulator
- **LOAD const**: set the value of the accumulator to an integer constant
- **STO x**: write accumulator to memory location (variable) x

Assembly Language

- **ADD x**: add the value of memory location x to the content of the accumulator
 - The result stays in the accumulator
- **ADD const**: add constant to accumulator
- **BR L**: branch to label L (goto)
- **BZ L**: if accumulator is zero, branch to label L
- **L: NOP**: label L , associated with a "no-operation" instruction NOP

Code Generation Strategy

- Synthesized attribute **code**
 - Contains a sequence of instructions

- Example:

$\langle ae \rangle_1 ::= \langle ae \rangle_2 + \langle ae \rangle_3$
 $\langle ae \rangle_1.\text{code} :=$

leaves its result
in the accumulator

\langle
 $\langle ae \rangle_2.\text{code},$
"STO T1", \rightarrow temp
var
 $\langle ae \rangle_3.\text{code},$
"ADD T1"
 \rangle

Code Generation Strategy

▪ $\langle c \rangle_1 ::= \text{if } \langle be \rangle \text{ then } \langle c \rangle_2 \text{ else } \langle c \rangle_3$

$\langle c \rangle_1.\text{code} := \langle$
 $\langle be \rangle.\text{code},$
 " BZ L1",
 $\langle c \rangle_2.\text{code},$
 " BR L2",
 " L1: NOP",
 $\langle c \rangle_3.\text{code},$
 " L2: NOP"
 \rangle

Problems

- T1 cannot be used in $\langle ae \rangle_3.code$
 - Need to generate temporary names
- Labels L1 and L2 cannot be used in $\langle c \rangle_2.code$, $\langle c \rangle_3.code$, or elsewhere
 - Need to generate label names
- Keep counter for temporary names
 - inherited **temp**
- Keep "global" counter for label names
 - inherited **labin**, synthesized **labout**

Code Generation for Statements

$\langle \text{prog} \rangle ::= \langle c \rangle$

$\langle \text{prog} \rangle.\text{code} := \langle c \rangle.\text{code}$

$\langle c \rangle.\text{labin} := 0$

$\langle C \rangle_1 ::= \langle C \rangle_2 ; \langle C \rangle_3$

$\langle C \rangle_1.\text{code} :=$

$\text{concat}(\langle C \rangle_2.\text{code}, \langle C \rangle_3.\text{code})$

$\langle C \rangle_2.\text{labin} := \langle C \rangle_1.\text{labin}$

$\langle C \rangle_3.\text{labin} := \langle C \rangle_2.\text{labout}$

$\langle C \rangle_1.\text{labout} := \langle C \rangle_3.\text{labout}$

Code Generation for Statements

$\langle c \rangle ::= \text{skip}$

$\langle c \rangle.\text{code} := \langle \text{"NOP"} \rangle$

$\langle c \rangle.\text{labout} := \langle c \rangle.\text{labin}$

| $\langle \text{assign} \rangle$

$\langle c \rangle.\text{code} := \langle \text{assign} \rangle.\text{code}$

$\langle c \rangle.\text{labout} := \langle c \rangle.\text{labin}$

Code Generation for Statements

$\langle c \rangle_1 ::= \text{if } \langle be \rangle \text{ then } \langle c \rangle_2 \text{ else } \langle c \rangle_3$

$\langle c \rangle_2.labin := \langle c \rangle_1.labin + 2$

$\langle c \rangle_3.labin := \langle c \rangle_2.labout$

$\langle c \rangle_1.labout := \langle c \rangle_3.labout$

$\langle c \rangle_1.code := \text{concat}(\langle be \rangle.code,$
 ("BZ" label($\langle c \rangle_1.labin$)),
 $\langle c \rangle_2.code,$
 ("BR" label($\langle c \rangle_1.labin + 1$)),
 (label($\langle c \rangle_1.labin$) "NOP"),
 $\langle c \rangle_3.code,$
 (label($\langle c \rangle_1.labin+1$) "NOP"))

Code Generation for Statements

$\langle c \rangle_1 ::= \text{while } \langle be \rangle \text{ do } \langle c \rangle_2$
 $\langle c \rangle_2.labin := \langle c \rangle_1.labin + 2$
 $\langle c \rangle_1.labout := \langle c \rangle_2.labout$
 $\langle c \rangle_1.code := \text{concat}(\$
 $(\text{label}(\langle c \rangle_1.labin) \text{ "NOP" }),$
 $\langle be \rangle.code,$
 $(\text{"BZ"} \text{ label}(\langle c \rangle_1.labin + 1)),$
 $\langle c \rangle_2.code,$
 $(\text{"BR"} \text{ label}(\langle c \rangle_1.labin))$
 $(\text{label}(\langle c \rangle_1.labin+1) \text{ "NOP" }))$

Code Generation for Statements

`<assign> ::= id := <ae>`

`<ae>.temp := 1`

`<assign>.code := concat(
 <ae>.code,
 ("STO" id.lexval))`

- For recursive languages, we need to access variables on the run-time call stack (multiple instances of `id.lexval`)

Code Generation for Expressions

$\langle ae \rangle_1 ::= \text{intconst}$

$\langle ae \rangle_1.\text{code} := \langle (\text{"LOAD" intconst.lexval}) \rangle$

| **id**

$\langle ae \rangle_1.\text{code} := \langle (\text{"LOAD" id.lexval}) \rangle$

| $\langle ae \rangle_2 + \langle ae \rangle_3$

$\langle ae \rangle_2.\text{temp} := \langle ae \rangle_1.\text{temp}$

$\langle ae \rangle_3.\text{temp} := \langle ae \rangle_1.\text{temp} + 1$

$\langle ae \rangle_1.\text{code} := \text{concat}(\langle ae \rangle_2.\text{code},$
 $(\text{"STO" temp}(\langle ae \rangle_1.\text{temp})),$
 $\langle ae \rangle_3.\text{code},$
 $(\text{"ADD" temp}(\langle ae \rangle_1.\text{temp}))$)

Example for Code Generation

- Source program

if $x = 42$ then

 if $a = b$ then

$y := 1$

 else

$y := 2$

else

$y := 3$

Example for Code Generation

- Generated code for outer if statement

code for (x=42)

BZ L1

code for inner if

BR L2

L1: NOP

code for y := 3

L2: NOP

Inner If Statement

```
# code for x = 42
BZ L1
# code for a = b
BZ L3
# code for y := 1
BR L4
L3: NOP
# code for y := 2
L4: NOP
BR L2
L1: NOP
# code for y := 3
L2: NOP
```

Code for Assignments

```
# code for x = 42
BZ L1
# code for a = b
BZ L3
LOAD 1
STO y
BR L4
L3: NOP
LOAD 2
STO y
L4: NOP
BR L2
L1: NOP
LOAD 3
STO y
L2: NOP
```

Summary: Attribute Grammars

- Useful for expressing arbitrary cycle-free walks over context-free parse trees
- Synthesized and inherited attributes
- Conditions to reject invalid parse trees
- Evaluation order depends on attribute dependencies

Summary: Attribute Grammars

- Realistic applications: for **type checking** and **code generation**
- “Global” data structures must be passed around as attributes
- Any data structure (sets, etc.) can be used as long as we work on paper
- The evaluation rules can call auxiliary functions
 - but the functions cannot have side effects