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

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
  - stdlinux> 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

<stmt> ::= **while** <exp> **do** <stmt>

| **if** <exp> **then** <stmt>

| **if** <exp> **then** <stmt> **else** <stmt>

| <exp> **:=** <exp>

| <id> **(** <exprs> **)**

<exprs> ::= <exp> | <exprs> **,** <exp>

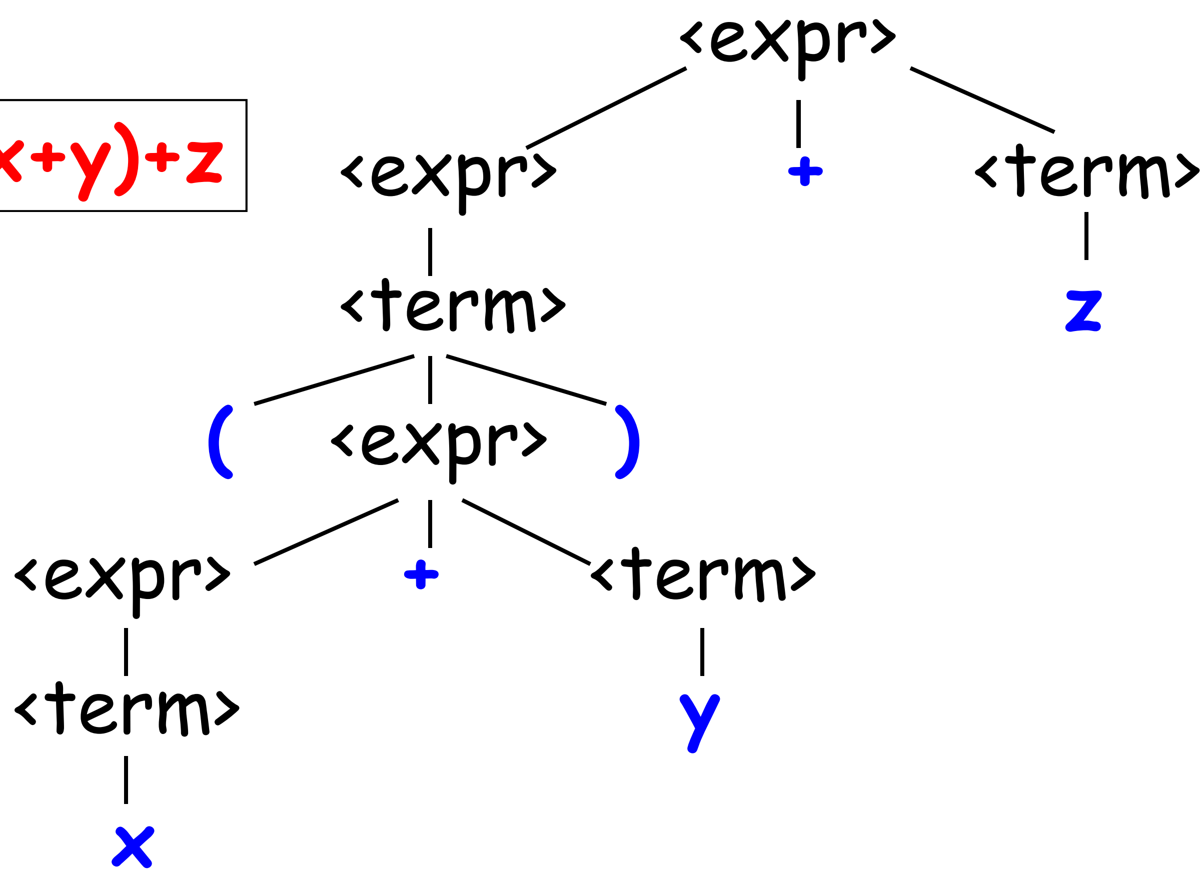
Derivation Tree

- 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

<expr> ::= <term> | <expr> + <term>
<term> ::= x | y | z | (<expr>)

(x+y)+z



Equivalent Derivation Sequences

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

One derivation:

<expr> ⇒ <expr> + <term> ⇒ <expr> + z ⇒
<term> + z ⇒ (<expr>) + z ⇒
(<expr> + <term>) + z ⇒ (<expr> + y) + z ⇒
(<term> + y) + z ⇒ (x + y) + z

Another derivation:

<expr> ⇒ <expr> + <term> ⇒ <term> + <term> ⇒
(<expr>) + <term> ⇒ (<expr> + <term>) + <term> ⇒
(<term> + <term>) + <term> ⇒ (x + <term>) + <term> ⇒
(x + y) + <term> ⇒ (x + y) + z

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

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

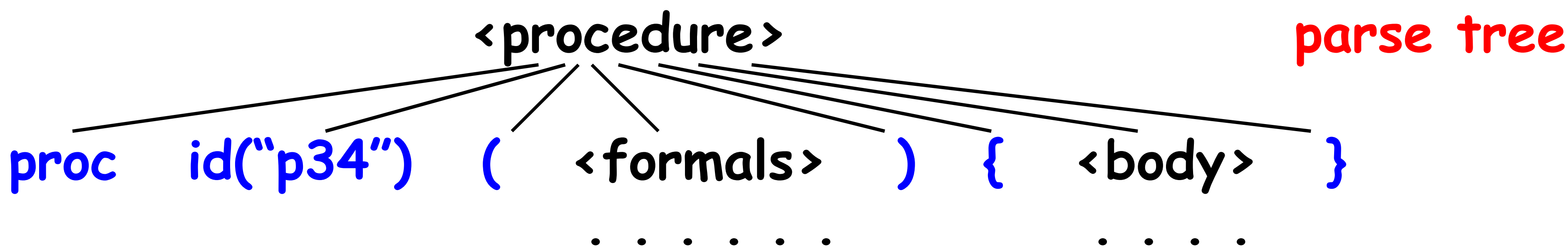
1. Prove that this grammar is ambiguous
2. Create an equivalent non-ambiguous grammar with the appropriate precedence and associativity
   - ***** has higher precedence than **+**
   - both are left-associative

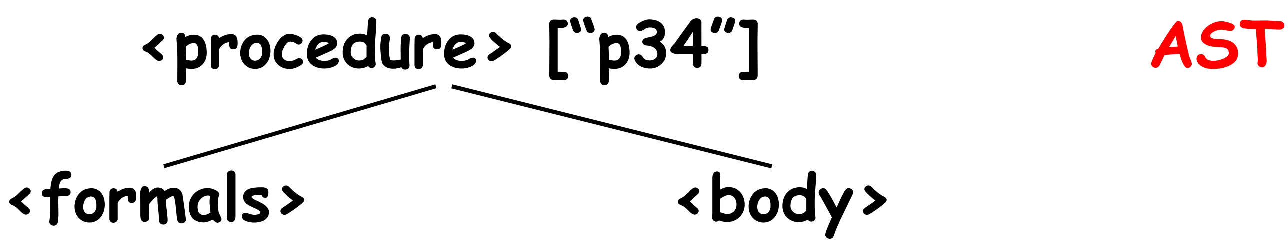
Example: what is the parse tree for
a + b * (c+d) * e ?

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

  <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>
  - Value domain for Na, Nb, Nc: integer values

Example: Alternative

- $L = \{ a^n b^n c^n \mid n > 0 \}$; not context-free
- BNF

  <start> ::= <A><B><C>   <A> ::= **a** | **a**<A>

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

- Attributes
  - **N**: associated with <A>, <B>, <C>
  - Value domain for N: integer values

Example

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

  <A> ::= **a**

  <A>.Na := 1

  | **a**$\langle A\rangle_2$

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

- Conditions

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

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

Example: Alternative

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

  <A> ::= **a**

  <A>.N := 1

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

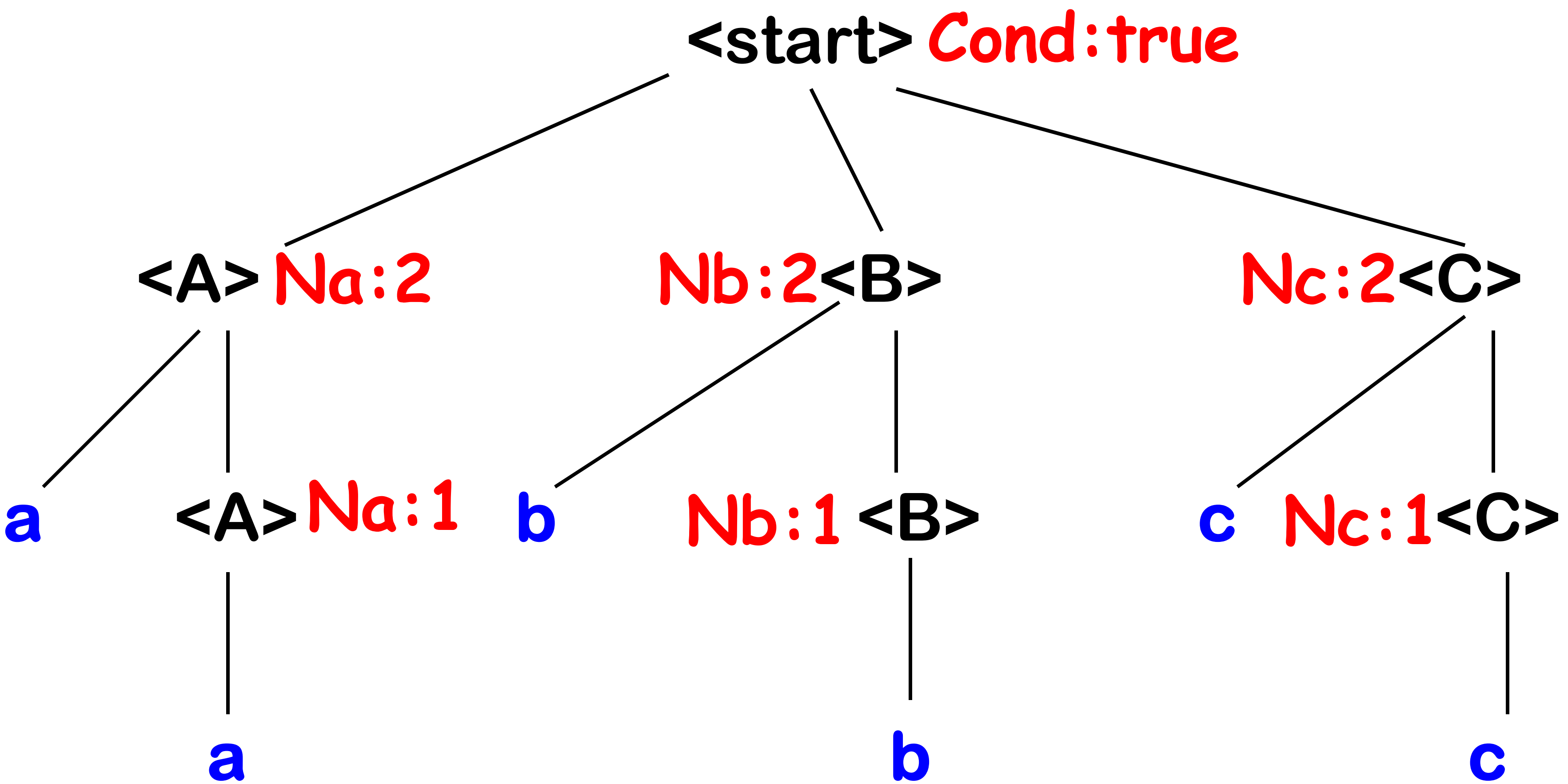
  <A>.N := 1 + <A>$_2$.N

- Conditions

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

  Cond: <A>.N = <B>.N = <C>.N

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)
  - $\mathrm{Attr}(X) = \mathrm{Syn}(X) \cup \mathrm{Inh}(X)$
- Production $X ::= Y_1\ Y_2 \ldots Y_n$
- Each attribute in Syn(X) is defined from $\mathrm{Attr}(Y_1) \cup \ldots \cup \mathrm{Attr}(Y_n) \cup \mathrm{Attr}(X)$
- Each attribute in $\mathrm{Inh}(Y_k)$ is defined from $\mathrm{Attr}(X) \cup \mathrm{Attr}(Y_1) \cup \ldots \cup \mathrm{Attr}(Y_n)$

Complete Evaluation Rules

- **Synthesized attribute associated with N:**
  - **Each alternative in "N ::= …" should contain a rule for evaluating the attribute**
- **Inherited attribute associated with N:**
  - **For every occurrence of N in "… ::= … N …" 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

  <B> ::= <D>

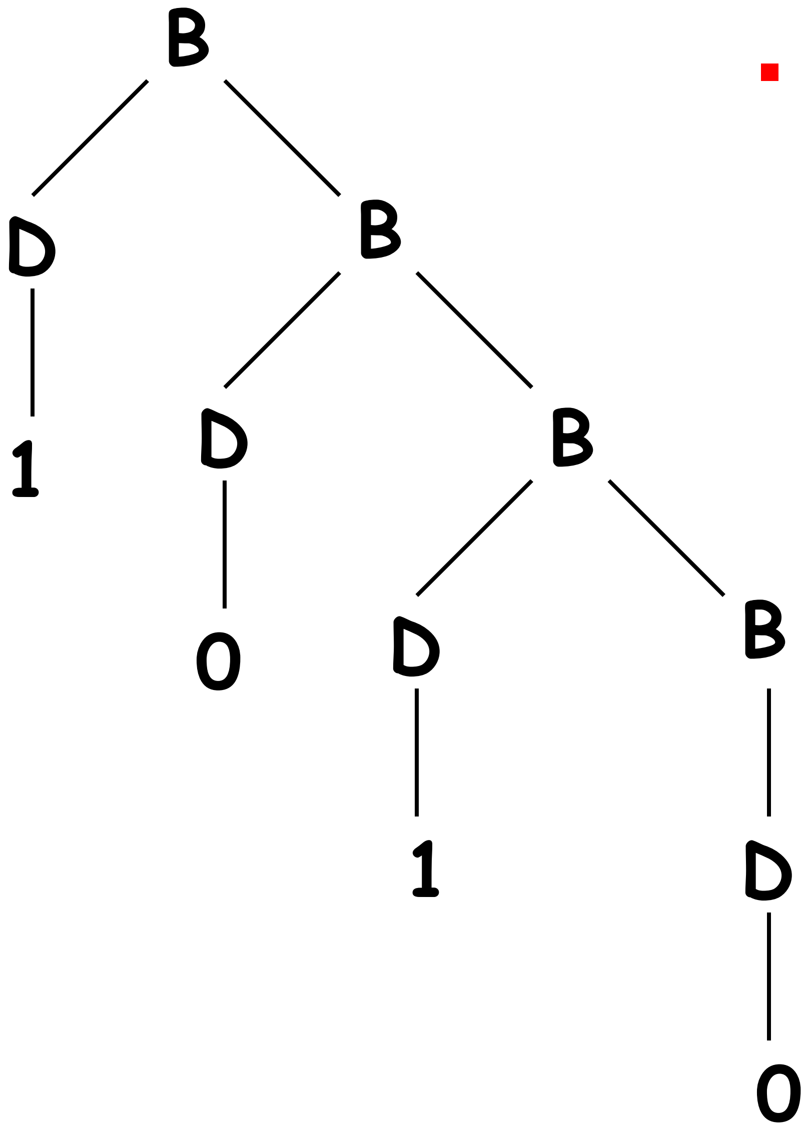
  <B> ::= <D><B>

  <D> ::= 0

  <D> ::= 1

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

BNF Parse Tree for Input 1010



- Add attributes
  - <B>: synthesized **val**
  - <B>: synthesized **pos**
  - <D>: inherited **pow**
  - <D>: **val**

Example: Binary Numbers

<B> ::= <D>

<B>.pos := 1

<B>.val := <D>.val

<D>.pow := 0

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

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

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

<D>.pow := $\langle B\rangle_2$.pos

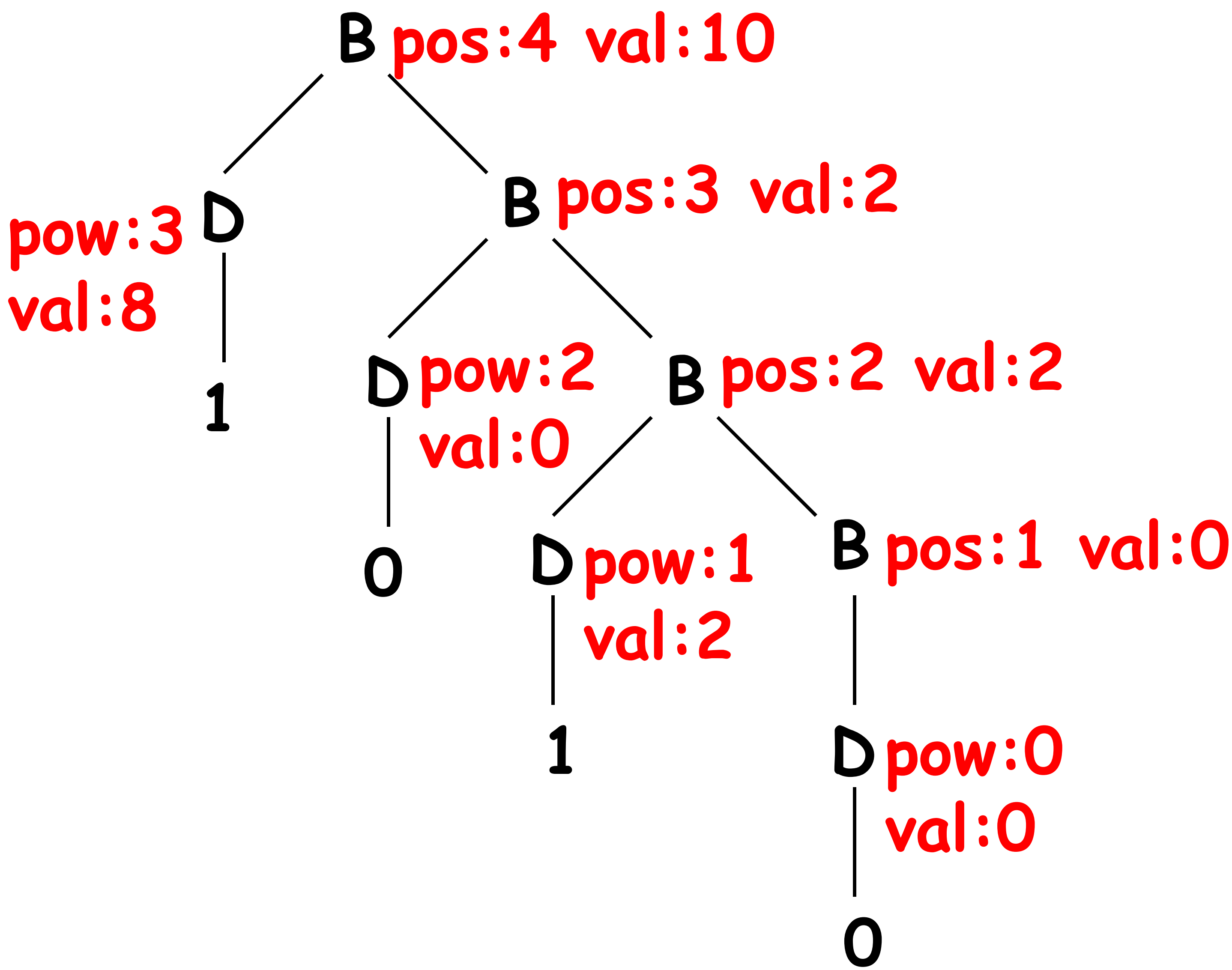
<D> ::= 0

<D>.val := 0

<D> ::= 1

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

Evaluated Parse Tree



Example: Expression Language

- Problem: evaluate an expression

<S> ::= <E>

<E> ::= **0** | **1** | <I> | **(**<E> **+** <E>**)** |
let <I> **=** <E> **in** <E> **end**

<I> ::= **id**

- Attributes
  - <I>: synthesized **name**
  - <E>: synthesized **val**, inherited **env**
  - **id**: synthesized **lexval**, represents the string that the lexical analysis puts inside token **id**

Attribute Grammar

<S> ::= <E>

<E>.env := ∅

<E> ::= 0

<E>.val := 0

<E> ::= 1

"helper" function

<E>.val := 1

<E> ::= <I>

<E>.val := lookup(<E>.env, <I>.name)

<I> ::= id

<I>.name := id.lexval

Attribute Grammar

<E>$_1$ ::= (<E>$_2$ + <E>$_3$)

<E>$_1$.val := <E>$_2$.val + <E>$_3$.val

<E>$_2$.env := <E>$_1$.env

<E>$_3$.env := <E>$_1$.env

<E>$_1$ ::= let <I> = <E>$_2$ in <E>$_3$ end

<E>$_2$.env := <E>$_1$.env

<E>$_3$.env :=

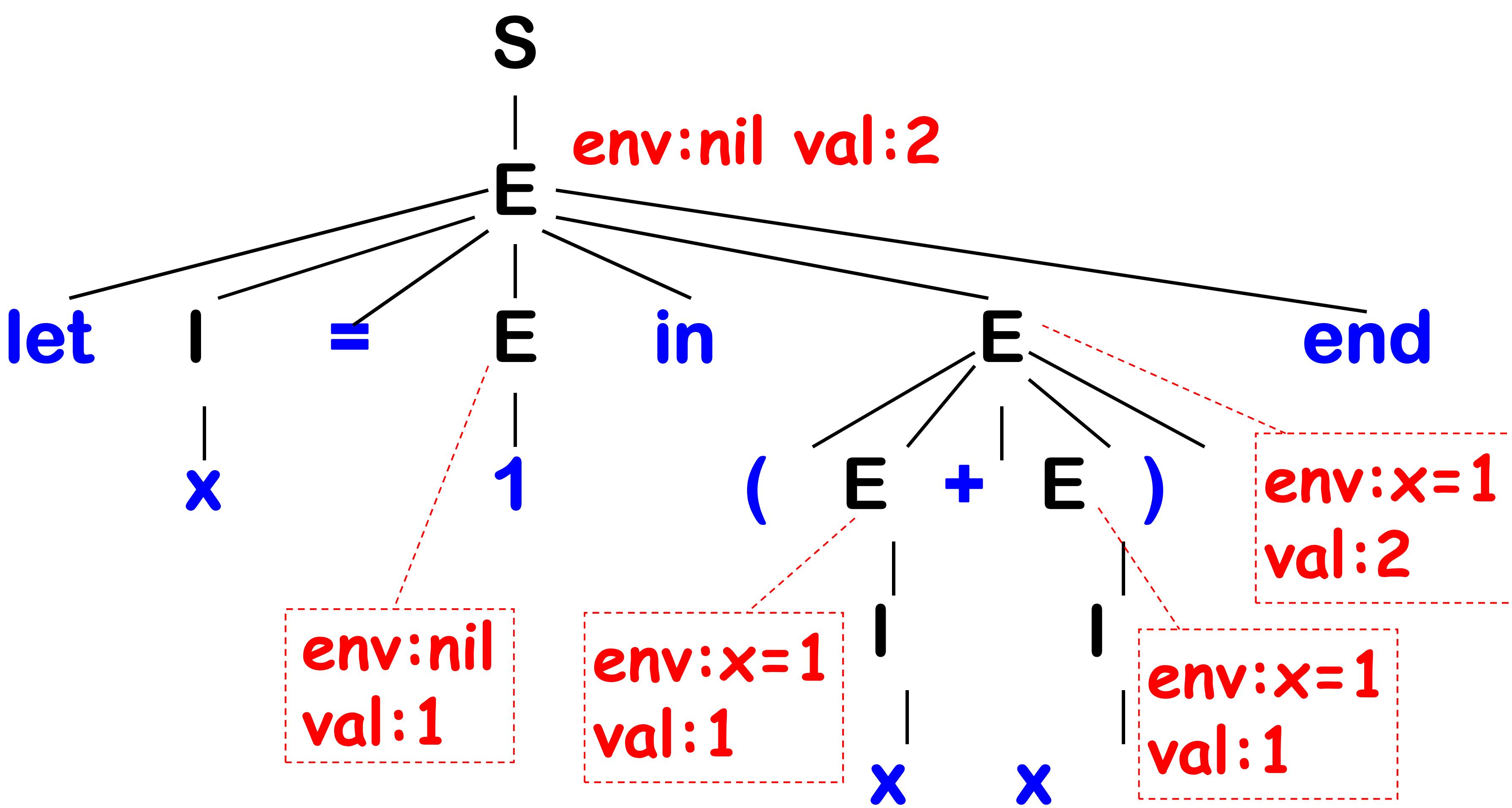
update(<E>$_1$.env,<I>.name,<E>$_2$.val)

<E>$_1$.val := <E>$_3$.val

creates a "fresh" environment; does **not** affect <E>$_1$.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
- $L = \{ wcw \mid w \in \{a,b\}^* \}$
  - 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

- $<S>_1 ::= a \mid b \mid <S>_2<S>_3$
  - 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: $<X> ::= a \mid b$

  $<S> ::= <X><S> \mid <X>$
- Another approach: attribute len for S
  - $<S> ::= a \mid b$ $\quad <S>.len := 1$
  - $<S>_1 ::= <S>_2<S>_3$ $\quad <S>_1.len := <S>_2.len + <S>_3.len$

    Cond: $<S>_2.len = 1$

"Fixing" Context-Free Grammars

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

    Cond: ?

Attribute Evaluation: Dependence Graph

- <A>.x := <B>.y + <C>.z
  - <A>.x depends on <B>.y
    - A.x $\leftarrow$ B.y dependence edge
  - <A>.x depends on <C>.z
    - A.x $\leftarrow$ C.z dependence edge
- <A>$_1$.x := <A>$_2$.x
  - <A>$_1$.x depends on <A>$_2$.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

<B> ::= <D>

<B>.pos := 1
<B>.val := <D>.val
<D>.pow := 0

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

$\langle B\rangle_1$.pos := $\langle B\rangle_2$.pos + 1
$\langle B\rangle_1$.val := $\langle B\rangle_2$.val + <D>.val
<D>.pow := $\langle B\rangle_2$.pos

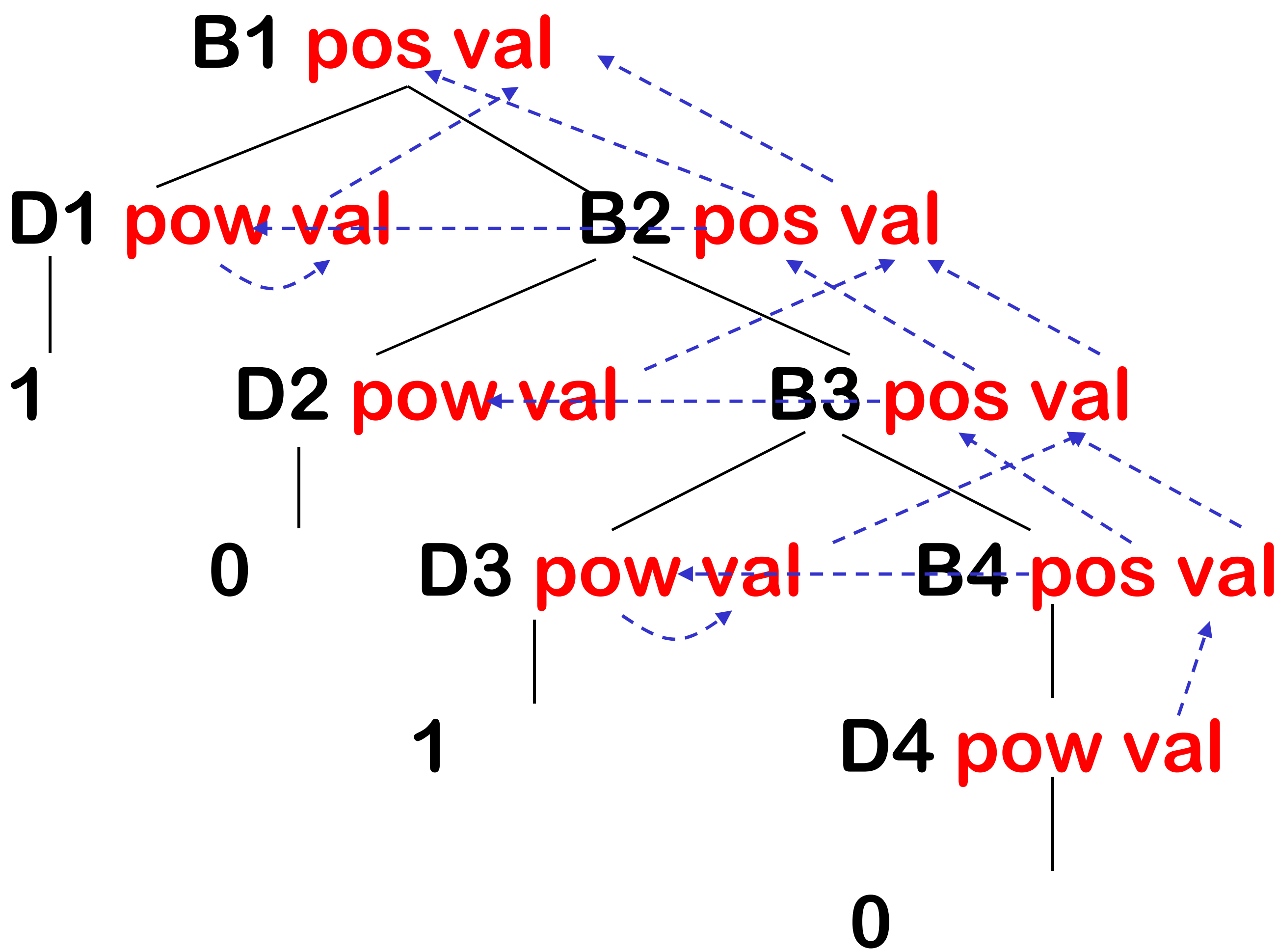
<D> ::= 0

<D>.val := 0

<D> ::= 1

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

Dependence Graph for Binary Numbers



Sort the Graph

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

B4.pos, D4.pow, 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: 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

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

<prog> ::= <block>

<block> ::= **begin** <declist> **;** <stmtlist> **end**

<stmtlist> ::= <stmt>

| <stmt> **;** <stmtlist>

<stmt> ::= <assign>

| <block>

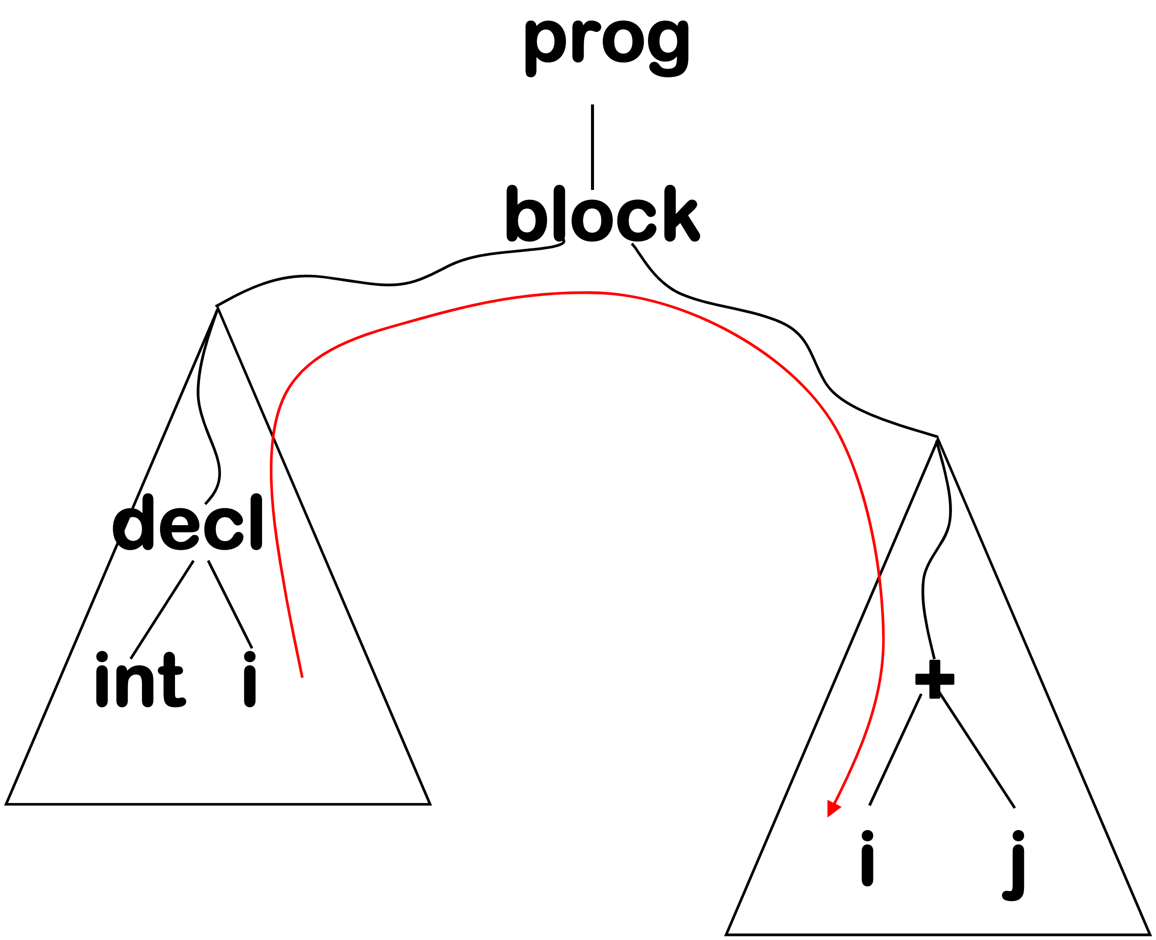
| …

…

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

<prog> ::= <block>
<block> ::= begin <declist> ; <stmtlist> end

<declist> ::= <decl> | <decl> ; <declist>
<decl> ::= int id | bool id |
fun id (<formalslist>) : int = <block>

<stmtlist> ::= <stmt> | <stmt> ; <stmtlist>
<stmt> ::= <assign> | <block> |
if <boolexpr>
then <stmtlist> else <stmtlist> fi

Context-Free Grammar – Part 2

<assign> ::= **id :=** <boolexp> |
id := <intexp>

<boolexp> ::= **true** | **false** | **id**

<intexp> ::= **intconst** | **id** |
<intexp> **+** <intexp> |
id (<actualslist> **)**

Ignore Functions for Now

<prog> ::= <block>

<block>.alltbl := emptystack

<block> ::= **begin** <declist> **;** <stmtlist> **end**

creates a "fresh" stack

<stmtlist>.alltbl :=

push(<declist>.tbl, <block>.alltbl)

<stmtlist>$_1$ ::= <stmt>

<stmt>.alltbl := <stmtlist>$_1$.alltbl

| <stmt> **;** <stmtlist>$_2$

<stmt>.alltbl := <stmtlist>$_1$.alltbl

<stmtlist>$_2$.alltbl := <stmtlist>$_1$.alltbl

Declarations

<declist>$_1$ ::= <decl>

<declist>$_1$.tbl := <decl>.tbl

| <decl> ; <declist>$_2$

<declist>$_1$.tbl :=

<decl>.tbl $\cup$ <declist>$_2$.tbl

Cond:

ids(<decl>.tbl) $\cap$ ids(<declist>$_2$.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

<decl> ::= **int id**

<decl>.tbl := { (id.lexval, INT) }

| **bool id**

<decl>.tbl := { (id.lexval, BOOL) }

| **fun id (** <formalslist> **) : int =** <block>

ignore for now

Statements

<stmtlist>$_1$ ::= <stmt>

<stmt>.alltbl := <stmtlist>$_1$.alltbl

| <stmt> ; <stmtlist>$_2$

<stmt>.alltbl := <stmtlist>$_1$.alltbl

<stmtlist>$_2$.alltbl := <stmtlist>$_1$.alltbl

<stmt> ::= <assign>

<assign>.alltbl := <stmt>.alltbl

| <block>

<block>.alltbl := <stmt>.alltbl

| **if** <boolexp> **then** <stmtlist> **else** ...

<boolexp>.alltbl := <stmt>.alltbl

<stmtlist>.alltbl := <stmt>.alltbl

Statements

<assign> ::= id := <intexp>

<intexp>.alltbl := <assign>.alltbl

the "last" type on the stack

Cond:

typeof(id.lexval,<assign>.alltbl) = INT

| id := <boolexp>

<boolexp>.alltbl := <assign>.alltbl

Cond:

typeof(id.lexval,<assign>.allbtl) = BOOL

Expressions

<intexp>$_1$ ::= **intconst**
| **id**
Cond: typeof(**id**.lexval,
<intexp>$_1$.alltbl) = INT
| <intexp>$_2$ **+** <intexp>$_3$
<intexp>$_2$.alltbl := <intexp>$_1$.alltbl
<intexp>$_3$.alltbl := <intexp>$_1$.alltbl
<boolexp> ::= **true**
| **false**
| **id**
Cond: typeof(**id**.lexval,
<boolexp>.alltbl) = BOOL

Declarations

<decl> ::= **int id**

<decl>.tbl := { (id.lexval, INT) }

| **bool id**

<decl>.tbl := { (id.lexval, BOOL) }

| **fun id (** <formalslist> **) : int =** <block>

<decl>.tbl :=

synthesized attr: ordered list of INT/BOOL

{ (id.lexval,

FUN(<formalslist>.types, INT)) }

<block>.alltbl → what are we going to do here?

More Propagation of Information

- **Declarations: add inherited alltbl**

<block> ::= **begin** <declist> **;** <stmtlist> **end**

<stmtlist>.alltbl :=

push(<declist>.tbl, <block>.alltbl)

<declist>.alltbl :=

push(<declist>.tbl, <block>.alltbl)

We first gather the declarations in <declist>, and then we use them to check the blocks "embedded" in <declist>

Declarations

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

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

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

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

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

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

$$\boxed{\begin{array}{l} \langle\text{decl}\rangle.\text{alltbl} := \langle\text{declist}\rangle_1.\text{alltbl} \\ \langle\text{declist}\rangle_2.\text{alltbl} := \langle\text{declist}\rangle_1.\text{alltbl} \end{array}}$$

Cond:

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

Declarations Again

<decl> ::= int id

<decl>.tbl := { (id.lexval, INT) }

| bool id

<decl>.tbl := { (id.lexval, BOOL) }

| fun id (<formalslist>) : int = <block>

<decl>.tbl :=

{ (id.lexval,

FUN(<formalslist>.types, INT)) }

<block>.alltbl :=

push(<formalslist>.tbl,<decl>.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

<actualslist> is a list of <intexp> and <boolexp>

<intexp> ::= id (<actualslist>)

Cond: typeof(id.lexval, <intexp>.alltbl) = FUN

redundant if we have only int return types

Cond: rettype(id.lexval, <intexp>.alltbl) = INT

<actualslist>.expTypes := paramtypes(id.lexval, <intexp>.alltbl)

<actualslist>.alltbl := <intexp>.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)

$<c>_1 \ ::= \ \textbf{skip} \mid <assign> \mid <c>_2 \ \textbf{;} \ <c>_3$

$\quad \mid \textbf{if} \ <be> \ \textbf{then} \ <c>_2 \ \textbf{else} \ <c>_3$

$\quad \mid \textbf{while} \ <be> \ \textbf{do} \ <c>_2$

$<assign> \ ::= \ \textbf{id} \ \textbf{:=} \ <ae>$

$<ae>_1 \ ::= \ \textbf{id} \mid \textbf{intconst} \mid <ae>_2 + <ae>_3$

$\quad \mid <ae>_2 - <ae>_3 \mid <ae>_2 * <ae>_3$

$<be>_1 \ ::= \ \textbf{true} \mid \textbf{false}$

$\quad \mid <ae>_1 = <ae>_2 \mid <ae>_1 < <ae>_2$

$\quad \mid \neg <be>_2 \mid <be>_2 \wedge <be>_3$

$\quad \mid <be>_2 \vee <be>_3$

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:

$\langle ae \rangle_1 ::= \langle ae \rangle_2 + \langle ae \rangle_3$

$\langle ae \rangle_1.code :=$

<

$\langle ae \rangle_2.code$,

"STO T1",

$\langle ae \rangle_3.code$,

"ADD T1"

>

leaves its result in the accumulator

temp var

Code Generation Strategy

- $<c>_1$ ::= **if** <be> **then** $<c>_2$ **else** $<c>_3$

  $<c>_1$.code := <
  <be>.code,
  "BZ L1",
  $<c>_2$.code,
  "BR L2",
  "L1: NOP",
  $<c>_3$.code,
  "L2: NOP"
  >

Problems

- T1 cannot be used in <ae>$_3$.code
  - Need to generate temporary names
- Labels L1 and L2 cannot be used in <c>$_2$.code, <c>$_3$.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

<prog> ::= <c>

<prog>.code := <c>.code

<c>.labin := 0

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

$\langle c\rangle_1$.code := concat($\langle c\rangle_2$.code,$\langle c\rangle_3$.code)

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

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

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

Code Generation for Statements

<c> ::= **skip**

<c>.code := < "NOP" >

<c>.labout := <c>.labin

| <assign>

<c>.code := <assign>.code

<c>.labout := <c>.labin

Code Generation for Statements

<c>$_1$ ::= if <be> then <c>$_2$ else <c>$_3$

<c>$_2$.labin := <c>$_1$.labin + 2

<c>$_3$.labin := <c>$_2$.labout

<c>$_1$.labout := <c>$_3$.labout

<c>$_1$.code := concat(<be>.code,

("BZ" label(<c>$_1$.labin)),

<c>$_2$.code,

("BR" label(<c>$_1$.labin + 1)),

(label(<c>$_1$.labin) "NOP“),

<c>$_3$.code,

(label(<c>$_1$.labin+1) "NOP“))

Code Generation for Statements

<c>$_1$ ::= while <be> do <c>$_2$

<c>$_2$.labin := <c>$_1$.labin + 2

<c>$_1$.labout := <c>$_2$.labout

<c>$_1$.code := concat(

(label(<c>$_1$.labin) "NOP"),

<be>.code,

("BZ" label(<c>$_1$.labin + 1)),

<c>$_2$.code,

("BR" label(<c>$_1$.labin))

(label(<c>$_1$.labin+1) "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

$\langle ae\rangle_1$ ::= **intconst**

$\langle ae\rangle_1$.code := < ("LOAD" **intconst**.lexval) >

| **id**

$\langle ae\rangle_1$.code := < ("LOAD" **id**.lexval) >

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

$\langle ae\rangle_2$.temp := $\langle ae\rangle_1$.temp

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

$\langle ae\rangle_1$.code := concat(

$\langle ae\rangle_2$.code,

("STO" temp($\langle ae\rangle_1$.temp)),

$\langle ae\rangle_3$.code,

("ADD" temp($\langle ae\rangle_1$.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