

CS321 F'04 Lecture Notes
Lecture 12

Inherited Attributes

Sometimes convenient to make node's attributes dependent on **siblings** or **ancestors** in tree.

Useful for expressing dependence on **context**, e.g., relating identifier **uses** to **declarations**. (This is especially important because CFG cannot capture such dependencies.)

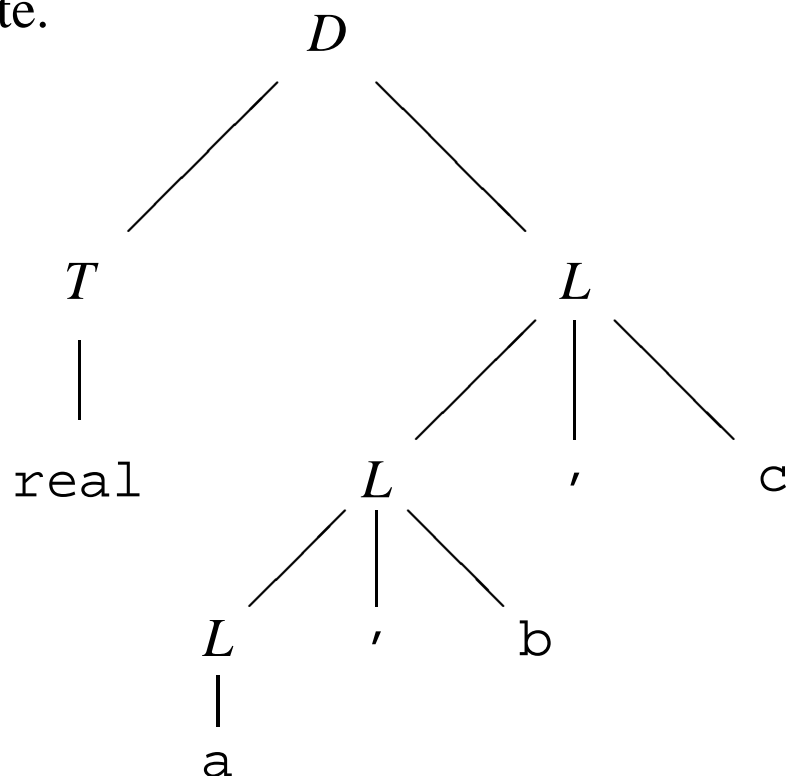
Example: Parsing Declarations

$$\begin{aligned}
 D &\rightarrow T L & L.type &:= T.type \\
 T &\rightarrow \text{int} & T.type &:= \text{integer} \\
 T &\rightarrow \text{real} & T.type &:= \text{real} \\
 L &\rightarrow L_1 , \text{id} & \{ L_1.type &:= L.type; \\
 & & & \text{addsymb}(\text{id.name}, L.type) \} \\
 L &\rightarrow \text{id} & \text{addsymb}(\text{id.name}, & L.type)
 \end{aligned}$$

where `addsymb` adds `id` and its type to symbol table.

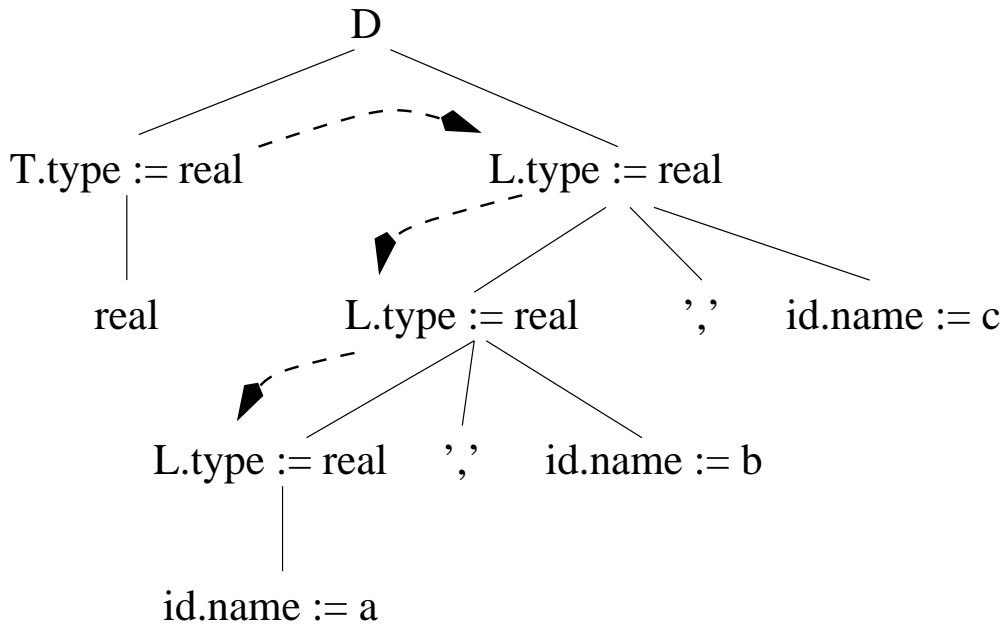
Here *L.type* is **inherited** attribute.

Parse tree for `real a,b,c`



Dependency Graphs

Parse tree for `real a, b, c`



Arrows show **dependency** relation among attributes. Taken together, arrows describe **dependency graph**.

Evaluate attributes in **topological** order of dependency graph.

If attributes are defined on parse tree, may want to evaluate attributes while (or instead of) building the tree. This is **sometimes** possible:

- Saw how to evaluate synthesized attributes during bottom-up parser; this method doesn't work for inherited attributes.
- Top-down parser can easily evaluate **L-attributed** grammars, in which attributes don't depend on their right ancestors. (Bottom-up parsers can sometimes handle these too, though with difficulty.)
- For some attribute grammars, must build entire tree **before** evaluating attributes.

Attribute Evaluation during Recursive Descent

Each non-terminal function is modified to take **inherited** attribute values as **arguments** and return (record of) **synthesized** attribute value(s) as **result**.

All inherited values are known when function is called.

All synthesized values are known when function returns.

Example revisited (with left-recursion removed):

```
class Typ ;
static Typ intTy = new Ty();
static Typ realTy = new Ty();

void D() {
    Typ ty = T();
    L(ty); }
Typ T() {
    if (tok == INT) {
        tok = lex(); return intTy;
    } else if (tok == REAL) {
        tok = lex(); return realTy;
    } else error(); }
void L(Typ ty) {
    if (tok == ID) {
        addsymb(lexeme,ty); tok = lex();
    } else error();
    if (tok == ',') {
        tok = lex();
        L(ty);
    } }
```

Avoiding Inherited Attributes

When using bottom-up parser (e.g., with yacc or CUP), it is desirable to avoid inherited attributes.

There are several approaches:

- Move the activity requiring the attribute to a higher node in the tree, by substituting a synthesized attribute for the inherited one, e.g.:

$$\begin{array}{ll}
 D \rightarrow T L & \text{for each } \text{id} \text{ in } L.\text{list} \\
 & \text{addsymp}(\text{id}.\text{name}, T.\text{type}) \\
 T \rightarrow \text{int} & T.\text{type} := \text{integer} \\
 T \rightarrow \text{real} & T.\text{type} := \text{real} \\
 L \rightarrow L_1 , \text{id} & L.\text{list} := \text{mk_Ids}(\text{id}, L_1.\text{list}) \\
 L \rightarrow \text{id} & L.\text{list} := \text{mk_Ids}(\text{id}, \text{null})
 \end{array}$$

- Can sometimes **rewrite** grammar, e.g.:

$$\begin{array}{ll}
 D \rightarrow T \text{id} & \{ D.\text{type} := T.\text{type}; \\
 & \text{addsymp}(\text{id}.\text{name}, T.\text{type}) \} \\
 D \rightarrow D_1 , \text{id} & \{ D.\text{type} := D_1.\text{type}; \\
 & \text{addsymp}(\text{id}.\text{name}, D.\text{type}) \} \\
 T \rightarrow \text{int} & T.\text{type} := \text{integer} \\
 T \rightarrow \text{real} & T.\text{type} := \text{real}
 \end{array}$$

Attributes on AST's

Attribute grammar method extends to **abstract** grammars (not intended for parsing), e.g., AST grammars.

- Same concept, but evaluation always occurs after whole tree is built.
- Can use recursive descent to evaluate (rather than parse).
- Typical applications: typechecking, code generation, interpretation.

Why attribute grammars?

- **Compact**, convenient formalism.
- **Local** rules describe entire computation.
- Separate **traversal** from **computation**.
- (Purely **functional** rules can be evaluated in any order.)

Checking of E Language (Homework 1)

Can view checking process as evaluation of following attribute grammar, where

- $exp.ok$ and $exps.ok$ are synthesized boolean attributes indicating whether expression has checked successfully; and
- $exp.env$ and $exps.env$ are inherited environment attributes (with operators *empty*, *extend*, and *lookup*) containing entries for all in-scope variables.

$program \rightarrow exp \qquad exp.env := empty$

$exp \rightarrow ID \qquad exp.ok := lookup(exp.env, ID.name)$

$\rightarrow NUM \qquad exp.ok := true$

$\rightarrow exp_1 '+' exp_2 \quad \{ exp_1.env := exp_2.env = exp.env;$
 $exp.ok := exp_1.ok AND exp_2.ok \}$

$\rightarrow exp_1 '-' exp_2 \quad \{ exp_1.env := exp_2.env = exp.env;$
 $exp.ok := exp_1.ok AND exp_2.ok \}$

$\rightarrow ID '=' exp_1 \quad \{ exp_1.env := exp.env;$
 $exp.ok := lookup(exp.env, ID.name) AND exp_1.ok \}$

$\rightarrow if0 exp_1 exp_2 exp_3 \quad \{ exp_1.env := exp_2.env := exp_3.env := exp.env;$
 $exp.ok := exp_1.ok AND exp_2.ok AND exp_3.ok \}$

$\rightarrow '\{ vars ;' exps '\}' \quad \{ exps.env := extend(exp.env, vars);$
 $exp.ok := exps.ok \}$

$exps \rightarrow exp \quad \{ exp.env := exps.env;$
 $exps.ok := exp.ok \}$

$\rightarrow exp ';' exps_1 \quad \{ exp.env := exps_1.env := exps.env;$
 $exps.ok := exp.ok AND exps_1.ok \}$