Motivating Inherited Attributes

Sometimes it’s convenient to make a 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 CF grammar cannot capture such dependencies.)

Example: Simple C-like Variable Declarations

\[
D \rightarrow T \ L \\
T \rightarrow \text{int} \mid \text{real} \\
L \rightarrow L_1, \text{id} \mid \text{id}
\]

Parse tree for real a,b,c:

```
  D
 /\  \\
T  L
|  /\  \\
| L  L
|   |  \\
real , b , c
 /\  \\
L  L
|  /\  \\
| L  a
```

INHERITED ATTRIBUTE GRAMMAR

\[
\begin{align*}
D & \rightarrow TL & L\.type & := T\.type \\
T & \rightarrow \text{int} & T\.type & := \text{integer} \\
T & \rightarrow \text{real} & T\.type & := \text{real} \\
L & \rightarrow L_1, id & \{ & L_1\.type := L\.type; \text{add symb}(id\.name, L\.type) \} \\
L & \rightarrow id & \text{add symb}(id\.name, L\.type)
\end{align*}
\]

Here \text{add symb} adds \text{id} and its type to symbol table, and \text{L.type} is an \textit{inherited} attribute.

A parse tree showing \textit{dependency} relations among attributes:

\[
\begin{align*}
D & \quad \rightarrow \quad TL \\
T & \quad \rightarrow \quad \text{int} \\
T & \quad \rightarrow \quad \text{real} \\
L & \quad \rightarrow \quad L_1, id \quad \{ \quad L_1\.type := L\.type; \text{add symb}(id\.name, L\.type) \} \\
L & \quad \rightarrow \quad id \quad \text{add symb}(id\.name, L\.type)
\end{align*}
\]
Dependency arrows for a dependency graph; we must 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 S-attributed grammar, in which all attributes are synthesized, during bottom-up parsing; 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.) Example follows.

- For more complicated attribute grammars, might have to build some or all of tree before evaluating attributes.
Each non-terminal function now takes **inherited** attribute values as **arguments** and return (record of) **synthesized** attribute value(s) as result.

Example revisited (with left-recursion removed):

```java
class Ty {}
static Ty intTy = new Ty(); static Ty realTy = new Ty();

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

\[
\begin{align*}
D \rightarrow T L & \quad \text{for each } \text{id} \text{ in } L\text{.list} \\
& \quad \text{addsym}(\text{id}\text{.name, }T\text{.type}) \\
T \rightarrow \text{int} & \quad T\text{.type} := \text{integer} \\
T \rightarrow \text{real} & \quad T\text{.type} := \text{real} \\
L \rightarrow L_1, \text{id} & \quad L\text{.list} := \text{append-list}(\text{id}, L_1\text{.list}) \\
L \rightarrow \text{id} & \quad L\text{.list} := \text{singleton-list}(\text{id})
\end{align*}
\]
Avoiding Inherited Attributes (2)

- Can sometimes rewrite grammar, e.g.:

\[
D \rightarrow T \ id \quad \{ \begin{array}{l}
D.\text{type} := T.\text{type}; \\
\text{addsym}(id.\text{name}, T.\text{type})
\end{array} \}
\]

\[
D \rightarrow D_1 \ , \ id \quad \{ \begin{array}{l}
D.\text{type} := D_1.\text{type}; \\
\text{addsym}(id.\text{name}, D.\text{type})
\end{array} \}
\]

\[
T \rightarrow \text{int} \quad T.\text{type} := \text{integer}
\]

\[
T \rightarrow \text{real} \quad T.\text{type} := \text{real}
\]
Attribute grammar method extends to **abstract** grammars (not intended for parsing), e.g., AST grammars.

- Same concept, but attribute evaluation always occurs after whole tree is built.
- Can use recursive descent as an attribute evaluation technique (regardless of how parsing was performed).
- 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 → exp

exp → ID
→ NUM
→ exp_1 ‘+’ exp_2 { exp_1.env := exp_2.env = exp.env;
  exp.ok := exp_1.ok AND exp_2.ok }
→ exp_1 ‘-’ exp_2 { exp_1.env := exp_2.env = exp.env;
  exp.ok := exp_1.ok AND exp_2.ok }
→ ID ‘=’ exp_1 { exp_1.env := exp.env;
  exp.ok := lookup(exp.env,ID.name) AND exp_1.ok }
→ if0 exp_1 exp_2 exp_3
  { exp_1.env := exp_2.env := exp_3.env := exp.env;
  exp.ok := exp_1.ok AND exp_2.ok AND exp_3.ok }
→ ‘{’ vars ‘;’ exps ‘}’ { exps.env := extend(exp.env,vars);
  exp.ok := exps.ok }

exps → exp { exp.env := exps.env;
  exps.ok := exp.ok }
→ exp ‘;’ exps_1 { exp.env := exps_1.env := exps.env;
  exps.ok := exp.ok AND exps_1.ok }