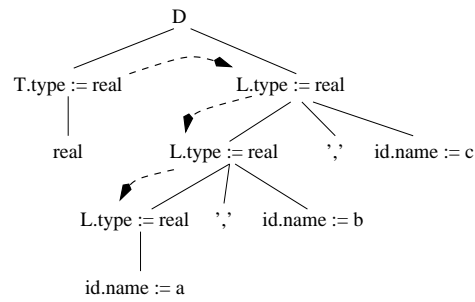


Dependency Graphs

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.

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

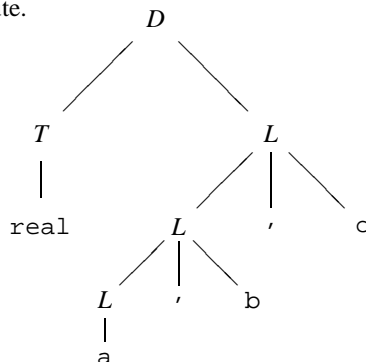
```

D → TL      L.type := T.type
T → int     T.type := integer
T → real   T.type := real
L → L1 , id { L1.type := L.type;
               addsymb(id.name, L.type) }
L → id      addsymb(id.name, L.type)
    
```

where addsymb adds id and its type to symbol table.

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

Parse tree for real a, b, c



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.:

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

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

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

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.

$$\begin{aligned}
 \text{program} &\rightarrow \text{exp} && \text{exp.env} := \text{empty} \\
 \\
 \text{exp} &\rightarrow ID && \text{exp.ok} := \text{lookup}(\text{exp.env}, ID.name) \\
 &\rightarrow NUM && \text{exp.ok} := \text{true} \\
 &\rightarrow \text{exp}_1 '+' \text{exp}_2 && \{ \text{exp}_1.env := \text{exp}_2.env = \text{exp.env}; \\
 &&& \text{exp.ok} := \text{exp}_1.ok \text{ AND } \text{exp}_2.ok \} \\
 &\rightarrow \text{exp}_1 '-' \text{exp}_2 && \{ \text{exp}_1.env := \text{exp}_2.env = \text{exp.env}; \\
 &&& \text{exp.ok} := \text{exp}_1.ok \text{ AND } \text{exp}_2.ok \} \\
 &\rightarrow ID '=' \text{exp}_1 && \{ \text{exp}_1.env := \text{exp.env}; \\
 &&& \text{exp.ok} := \text{lookup}(\text{exp.env}, ID.name) \text{ AND } \text{exp}_1.ok \} \\
 &\rightarrow \text{if0 } \text{exp}_1 \text{ exp}_2 \text{ exp}_3 && \{ \text{exp}_1.env := \text{exp}_2.env := \text{exp}_3.env := \text{exp.env}; \\
 &&& \text{exp.ok} := \text{exp}_1.ok \text{ AND } \text{exp}_2.ok \text{ AND } \text{exp}_3.ok \} \\
 &\rightarrow \{ ' \text{vars } ' ; ' \text{exps } ' \} && \{ \text{exps.env} := \text{extend}(\text{exp.env}, \text{vars}); \\
 &&& \text{exp.ok} := \text{exps.ok} \} \\
 \\
 \text{exps} &\rightarrow \text{exp} && \{ \text{exp.env} := \text{exps.env}; \\
 &&& \text{exps.ok} := \text{exp.ok} \} \\
 &\rightarrow \text{exp } ';' \text{ exps}_1 && \{ \text{exp.env} := \text{exps}_1.env := \text{exps.env}; \\
 &&& \text{exps.ok} := \text{exp.ok AND } \text{exps}_1.ok \}
 \end{aligned}$$

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.)