CS321 Languages and Compiler Design I

Fall 2010

Lecture 12

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.)

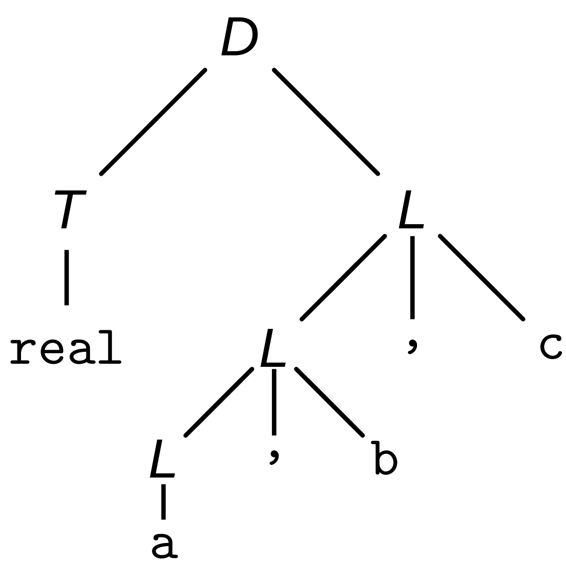
Example: Simple C-like Variable Declarations

$D \rightarrow T\ L$

$T \rightarrow \texttt{int} \mid \texttt{real}$

$L \rightarrow L_1 \texttt{,id} \mid \texttt{id}$

Parse tree for `real a,b,c`:

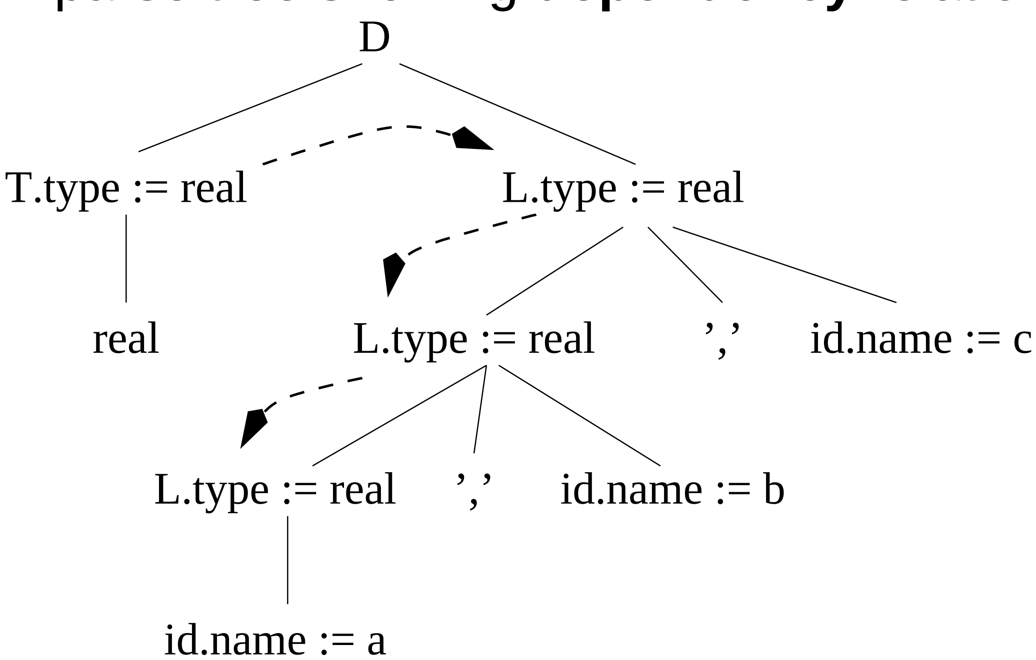


INHERITED ATTRIBUTE GRAMMAR

$D \rightarrow T\ L$	*L.type := T.type*
$T \rightarrow$ `int`	*T.type := integer*
$T \rightarrow$ `real`	*T.type := real*
$L \rightarrow L_1$ `,` `id`	{ L_1*.type := L.type; addsymb(*`id`*.name, L.type)* }
$L \rightarrow$ `id`	*addsymb(*`id`*.name, L.type)*

Here addsymb adds `id` and its type to symbol table, and *L.type* is an **inherited** attribute.

A parse tree showing **dependency** relations among attributes:



ATTRIBUTE EVALUATION

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 wich 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.

ATTRIBUTE EVALUATION DURING RECURSIVE DESCENT

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):

```
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);} }
```

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.:

$D \rightarrow T\ L$	*for each* `id` *in L.list*
	addsymb(`id`*.name, T.type)*
$T \rightarrow$ `int`	*T.type := integer*
$T \rightarrow$ `real`	*T.type := real*
$L \rightarrow L_1$ `,id`	*L.list := append-list(*`id`*,*L_1*.list)*
$L \rightarrow$ `id`	*L.list := singleton-list(*`id`*)*

Avoiding Inherited Attributes (2)

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

$D \rightarrow T$ `id`	$\{$*D.type := T.type;*
	addsymb(`id`*.name,T.type)* $\}$
$D \rightarrow D_1$, `id`	$\{$*D.type :=* D_1*.type;*
	addsymb(`id`*.name,D.type)* $\}$
$T \rightarrow$ `int`	*T.type := integer*
$T \rightarrow$ `real`	*T.type := real*

ATTRIBUTES ON AST'S

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	$\longrightarrow$	*exp*	*exp.env := empty*
exp	$\longrightarrow$	*ID*	*exp.ok := lookup(exp.env,ID.name)*
	$\longrightarrow$	*NUM*	*exp.ok := true*
	$\longrightarrow$	exp_1 '+' exp_2	{ exp_1.env := exp_2.env = exp.env; exp.ok := exp_1.ok AND exp_2.ok }
	$\longrightarrow$	exp_1 '−' exp_2	{ exp_1.env := exp_2.env = exp.env; exp.ok := exp_1.ok AND exp_2.ok }
	$\longrightarrow$	ID '=' exp_1	{ exp_1.env := exp.env; exp.ok := lookup(exp.env,ID.name) AND exp_1.ok }
	$\longrightarrow$	`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 }
	$\longrightarrow$	'{' vars '`;`' exps '}'	{ exps.env := extend(exp.env,vars); exp.ok := exps.ok }
exps	$\longrightarrow$	*exp*	{ exp.env := exps.env; exps.ok := exp.ok }
	$\longrightarrow$	exp '`;`' $exps_1$	{ exp.env := $exps_1$.env := exps.env; exps.ok := exp.ok AND $exps_1$.ok }