CS301 W'99 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.)

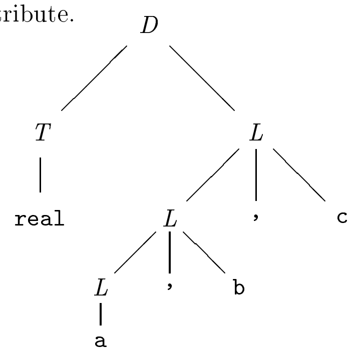
Example: Parsing Declarations

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

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

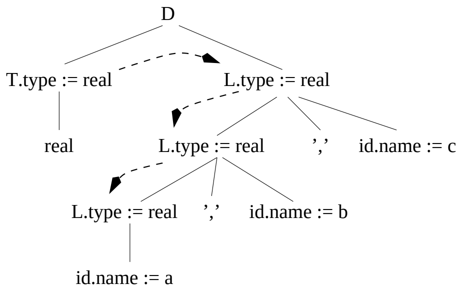
Here *L.type* is **inherited** attribute.

Parse tree for `real a,b,c`



Dependency Graphs

Parse tree for `real a,b,c`



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

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 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):

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


Avoiding Inherited Attributes

When using bottom-up parser (e.g., with `yacc`), 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.:

$D \to T\ L$	*for each* `id` *in L.list* *addsymb(*`id`*.name, T.type)*
$T \to$ `int`	*T.type := integer*
$T \to$ `real`	*T.type := real*
$L \to L_1$ `, id`	*L.list := mk_Ids(*`id`*,*L_1*.list)*
$L \to$ `id`	*L.list := mk_Ids(*`id`*,null)*

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

$D \to T$ `id`	{*D.type := T.type;* *addsymb(*`id`*.name,T.type)* }
$D \to D_1$ `, id`	{*D.type :=* D_1*.type;* *addsymb(*`id`*.name,D.type)* }
$T \to$ `int`	*T.type := integer*
$T \to$ `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 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	$\to$ *exp*	*exp.env := empty*
exp	$\to$ *ID*	*exp.ok := lookup(exp.env,ID.name)*
	$\to$ *NUM*	*exp.ok := true*
	$\to$ exp_1 '`+`' exp_2	{ exp_1*.env :=* exp_2*.env = exp.env;* *exp.ok :=* exp_1*.ok AND* exp_2*.ok* }
	$\to$ exp_1 '`-`' exp_2	{ exp_1*.env :=* exp_2*.env = exp.env;* *exp.ok :=* exp_1*.ok AND* exp_2*.ok* }
	$\to$ *ID* '`=`' exp_1	{ exp_1*.env := exp.env;* *exp.ok := lookup(exp.env,ID.name) AND* exp_1*.ok* }
	$\to$ `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* }
	$\to$ '{' *vars* ';' *exps* '}'	{ *exps.env := extend(exp.env,vars);* *exp.ok := exps.ok* }
exps	$\to$ *exp*	{ *exp.env := exps.env;* *exps.ok := exp.ok* }
	$\to$ *exp* ';' $exps_1$	{ *exp.env :=* $exps_1$*.env := exps.env;* *exps.ok := exp.ok AND* $exps_1$*.ok* }