CSC 530 Lecture Notes Week 4

Intro Formal Semantics of PLs
Intro to Attribute Grammars

I. Reading: Papers 10 and 11

II. What is Semantics?

A. The *meaning* of program

B. Broadly, two fundamental forms:

1. how a program *behaves*

2. What a program *denotes*

What is semantics, cont'd

C. Also defined as *not syntax*.

1. Syntax expresses structure
2. Semantics expresses meaning

D. Semantic eval in two phases:

1. Static semantics (type chking)
2. Dynamic semantics (exec'n)

III. How to Specify Semantics?

A. Informal approaches:

1. Free-form English

2. Formalized English

3. Output of a compiler

How to Specify Semantics, cont'd

B. Formal approaches

1. Attribute Grammars (Knuth)

2. Axiomatic (Hoare)

3. Denotational (Scott, Strachey)

4. Algebraic (Goguen)

5. Operational (you all)

IV. **Why *Formal* Semantics?**

A. Systematic, machine-independent, rigorous language design

1. A "BNF" for semantics.

2. Formal and concise.

3. Less bulky than operational def.

B. Formal def for translator imple'n

C. Basis for program verification

D. Reference for programmers

V. Common features of semantic definition techniques

A. Notational power and complexity

B. Syntax-directed.

C. Semantic *domains* of *environment* and *store*.

Common features, cont'd

D. Semantic "bootstrapping"

1. Start with grammar

2. *Operational semantics* requires abstract interpreter.

3. *Denotational semantics* requires mathematical logic.

Common features, cont'd

4. Bottom line -- define meaning in terms of what we already understand.

5. We must *trust* the underlying formalisms.

6. Mathematics is more trustworthy than interpreter.

VI. Role of functional PLS

A. Functional PL is mathematical, so can be used for formal semantics.

B. Concepts, notations from functional pgming used extensively.

VII. Overview of Major Techniques

A. For each technique consider:

1. *Language Semantics* -- semantics of a full PL

2. *Program Semantics* -- semantics of a particular program

3. *Orientation* -- practical uses

Overview of techniques, cont'd

B. Attribute grammars

1. *Language semantics* are

   a. CFG

   b. set of attributes

   c. attribute equations assoc'd with grammar rules

Attribute grammars, cont'd

2. *Program semantics* are:

   a. Attribute values associated with nodes of parse tree

   b. Values obtained by well-defined evaluation process

3. *Orientation* -- compilers

Overview of techniques, cont'd

C. Denotational

1. *Language semantics* are

   a. CFG (*abstract syntax*)

   b. Semantic domains

   c. Semantic functions that map syntactic forms into semantic domains.

Denotational overview, cont’d

2. *Program semantics* are results of semantic function eval’n

3. *Orientation* -- language design.

Overview of techniques, cont’d

D. Axiomatic

1. *Language Semantics* are:

   a. CFG

   b. axioms and rules of inference

   c. one axiom per grammar rule

Overview of axiomatic, cont'd

2. *Program Semantics* are:

    a. Formulae asserted to be true within a program

    b. Formula at end is meaning of the entire program.

3. *Orientation* -- program verification.

Overview of techniques, cont'd

E. Operational

   1. *Language Semantics* are:

      a. abstract syntax

      b. execution states of structured values

      c. set of instructions that change state

Overview of operational, cont’d

2. *Program Semantics* are set execution snapshots

3. *Orientation* -- compiler/interpreter writing; pedagogy.

VIII. Example attribute grammar for type checking

A. Defines *static semantics*

B. Components of the def:

1. "term-factor" BNF
2. string-valued *type* attribute
3. global list-valued *env* attribute of (*name*, *type*) pairs.
4. semantic equations defining how *type* is comnputed

Example attribute grammar, cont'd

C. Grammar rules and equations:

E ::= E_1 + T	E.type = (if E_1.type = T.type then E_1.type else "ERROR")
E ::= T	E.type = T.type
T ::= T_1 * F	T.type = (if T_1.type = F.type then T_1.type else "ERROR")
T ::= F	T.type = F.type
F ::= ident	F.type = Lookup(env, ident).type
F ::= real	F.type = "real"
F ::= integer	F.type = "integer"

Example attribute grammar, cont'd

D. Observations

1. Abstractly, "=" is math equality, not var assmnt

2. "=" *can be* interpreted concretely as assmnt

3. Equations appear as Yacc-like "action routines"

Example attribute grammar, cont'd

4. Equations are *abstract* action routines

5. Meaning expressed in *syntax-directed* framework

6. Equations employ *auxiliary functions*

IX. Another example -- expr eval

A. Attribute grammars can convey any aspect language semantics

1. Above defined type checking

2. Next we define expr eval

Another example, cont'd

B. Components of the def:

1. same "term-factor" grammar

2. numeric *val* attribute

3. semantic equations defining how *val* is computed

Another example, cont'd

C. Here are the rules:

$E ::= E_1 + T$	$E.val = E_1.val + T.val$
$E ::= T$	$E.val = T.val$
$T ::= T_1 * F$	$E.val = E_1.val * T.val$
$T ::= F$	$T.val = F.val$
F ::= ident	F.val = GetVal(store, ident)
F ::= real	F.val = read(val)
F ::= integer	F.val = read(val)

Another example, cont'd

D. Observations

1. As before, equations are abstraction of code

2. Use aux function *GetVal*

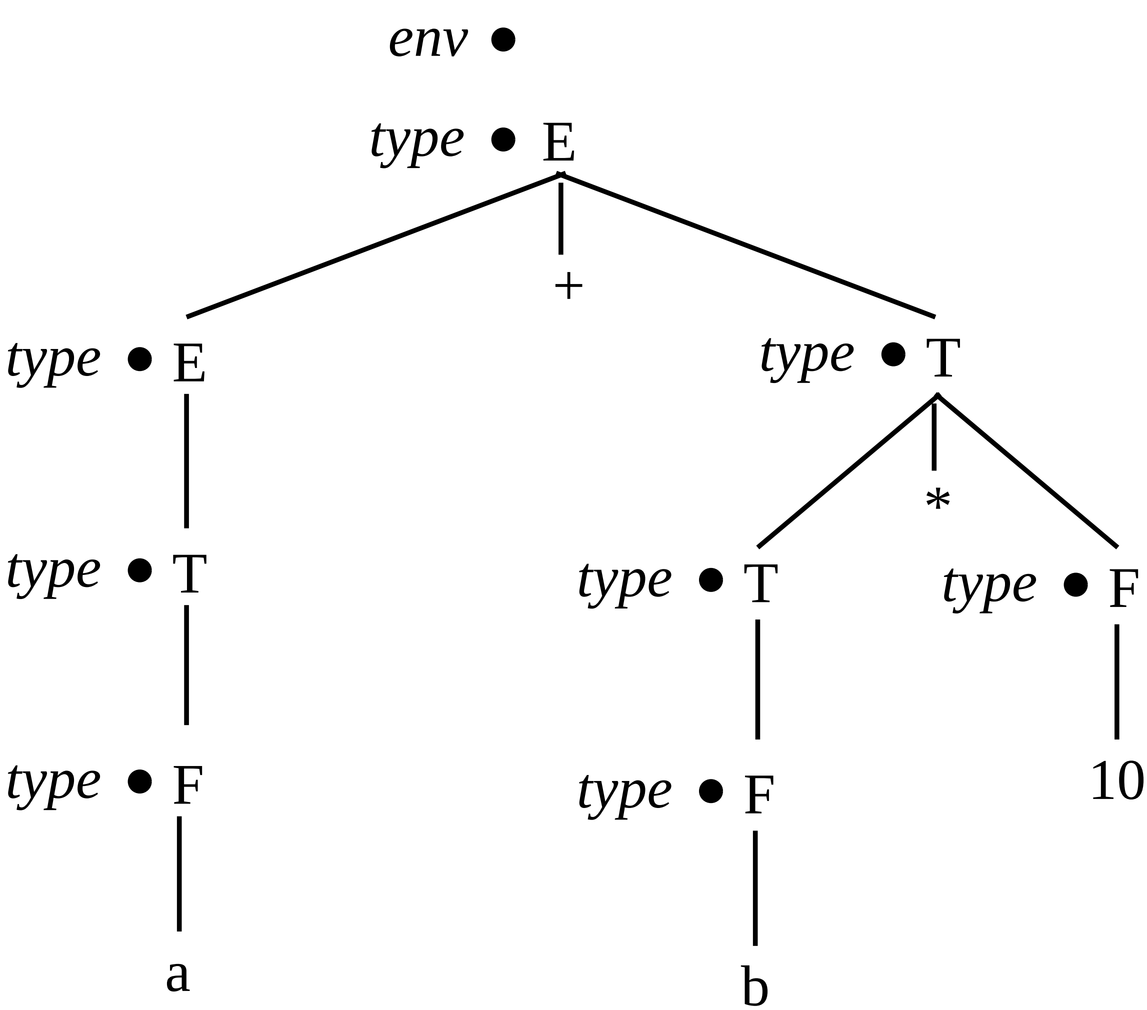
3. Other aux function *read*

X. **Attribute evaluation**

A. Using *attributed parse tree*

B. For example,

Attribute eval, cont'd



Attribute eval, cont’d

1. Labeled bullets mark computed attribute values

2. `env` attribute *global*, accessible at all nodes

Attribute eval, cont'd

C. Eval performed by applying semantic eqns at each tree node

1. Visit nodes in some order

2. Eqns do not specify order, only *attribute dependencies*.

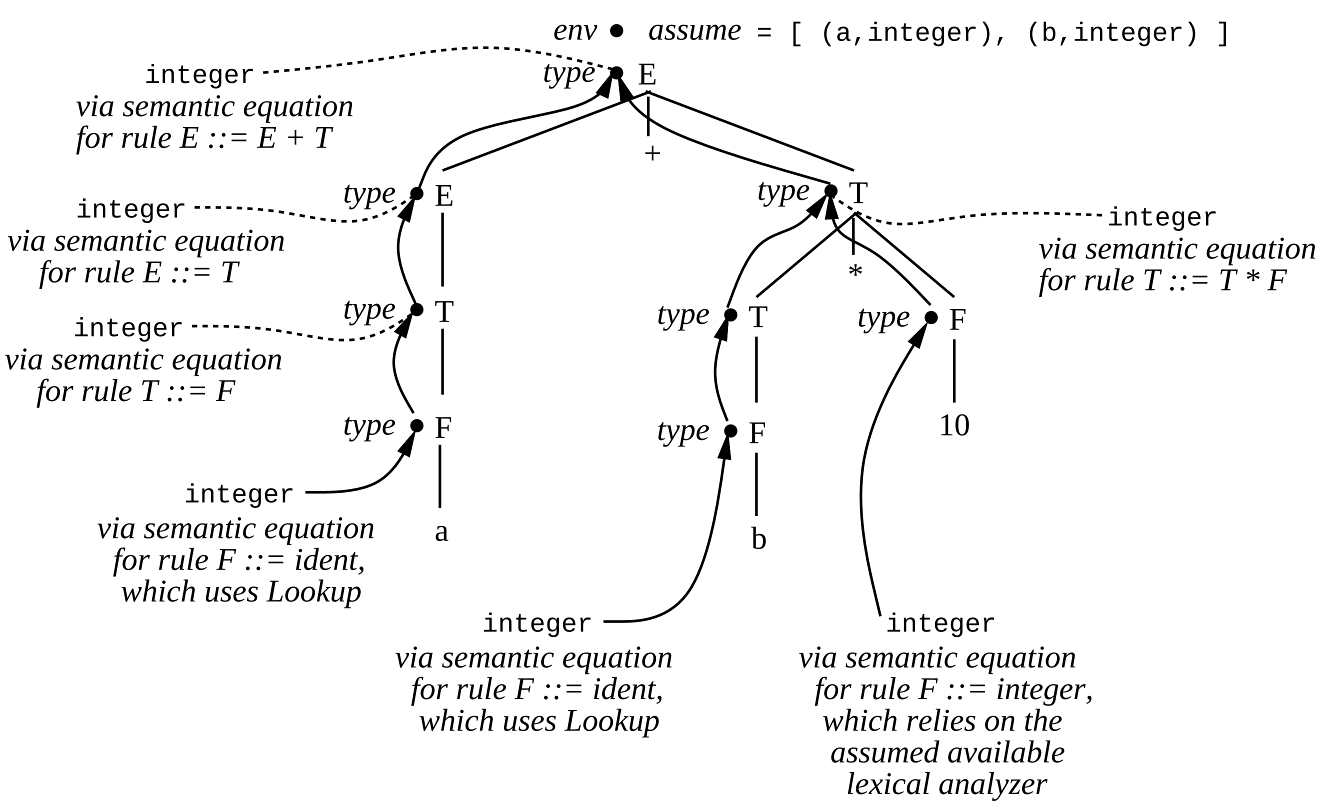
3. Evaluator chooses traversal order based on dependencies; for now postorder

Attribute eval, cont'd

D. Let's now trace

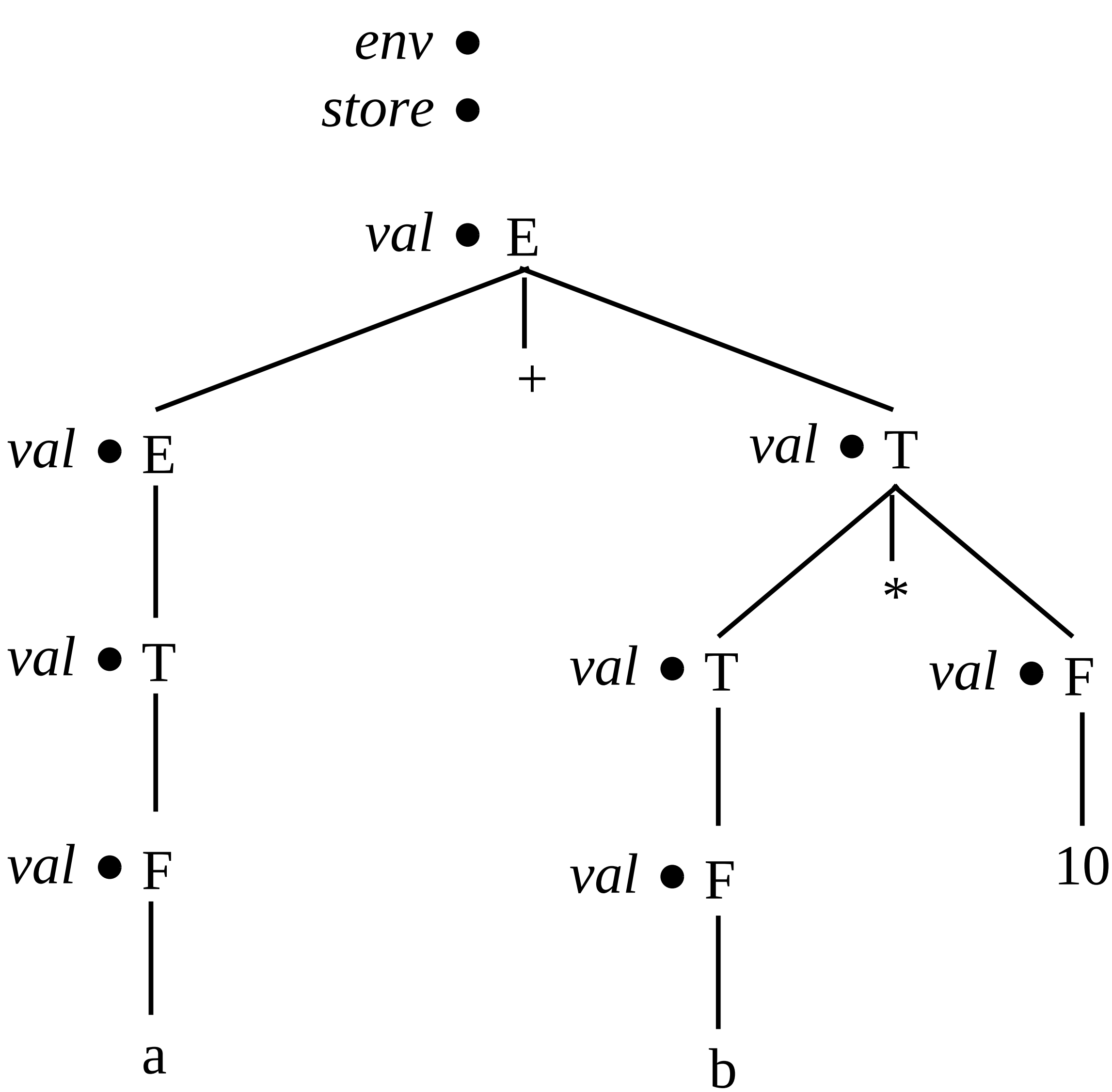
E. Here's the result:

Attribute eval, cont'd



Attribute eval, cont'd

F. Similar trace for expr eval on:



XI. Inherited versus synthesized attributes

A. Equations specify two forms of dependencies:

1. *Synthesized attribute* dependent on attributes *below*.

2. *Inherited attribute* dependent on attributes *above* or *beside*.

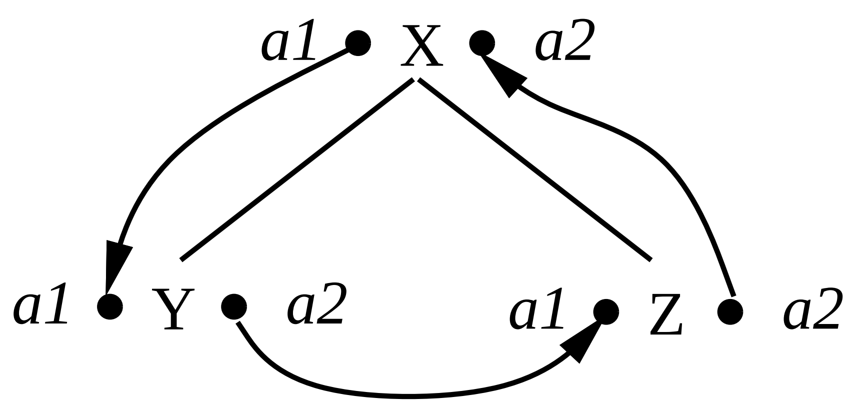
Inherited vs. synthesized, cont'd

B. E.g., consider

$$X ::= Y\ Z \qquad \begin{aligned} Y.a1 &= X.a1 \\ Z.a1 &= Y.a2 \\ X.a2 &= Z.a2 \end{aligned}$$

and the corresponding dependency diagram

Inherited vs. synthesized, cont'd



1. value of *Y.a1* inherited down from *X.a1*

2. value of *Z.a1* inherited across from *Y.a2*

3. value of *X.a2* synthesized up from *Z.a2*

Inherited vs. synthesized, cont'd

C. Dependencies dictate how to traverse for complete eval

1. With only synthesized attributes, eval can be single bottom-up traversal.

2. With inherited attrs, traversal order chosen so values of dependents are known.

3. With real PLs, one to three depth-first passes.

4. Details next time.

Inherited vs. synthesized, cont'd

D. Important to remember -- passes are not explicitly defined by eqns.

1. Equations are *declarative*.

2. Eval in any order, as long as the dependencies satisfied.[1]

---

[1] Unless global attributes are used; more next week.