§3.4 Attribute Grammars — for describing/checking static semantics

- Describes P.L. more than CFG can describe.
- extension to a CFG

ex) type compatibility

int ← float ; illegal in Java

- Static Semantics of a Language (or, static semantic rules) — type constraints; Can be checked at Compile time (static)

It's difficult to describe static semantics with BNF.

⇒ attribute grammar

- designed by Knuth (1968)
- describes both syntax and semantics
- describes/checks static semantic rules

- Dynamic semantics — for describing meanings of expressions, statements, prog. units, etc.

3 methods:
- operational semantics
- denotational "
- Axiometic "

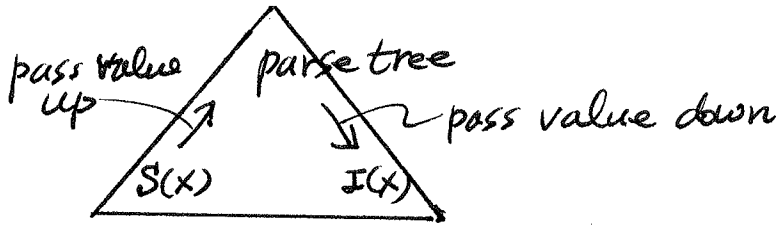
— Attribute grammar

CFG +
- attributes
- attribute computation functions (semantic funcs)
- predicate functions

- for each G symbol X,

  A(X) — a set of attributes

  [S(X) — synthesized attributes
  I(X) — Inherited "] disjoint sets



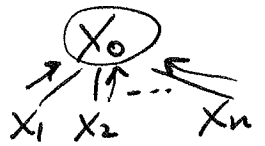
- for each G rule (production),

  ∃ set of semantic functions

  ex) rule (production): $X_0 \to X_1\ X_2 \cdots X_n$

  synthesized attributes of X_0 are computed from children.

  $S(X_0) \Leftarrow f(A(X_1), A(X_2), \cdots, A(X_n))$;



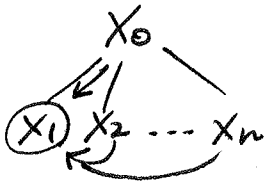
synthesized from children

  Inherited attributes of X_j $(1 \le j \le n)$:

  $I(X_j) \Leftarrow f(A(X_0), \cdots, A(X_n))$;

  or; $f(A(X_0), \cdots, A(X_{j-1}))$;

  to avoid circular comp.



— inherited from (parent and siblings)

- a predicate function has the form of a Boolean expr. on the union of the attribute set $\{A(X_0), A(X_1), \ldots, A(X_n)\}$ and a set of literal attribute values.

  - every predicate associated with every nonT is True
    ⇒ derivation allowed
  - a false predicate function value
    ⇒ violation of the syntax or static semantic rules

- a parse tree of an attribute G
  ≡ a parse tree based on the BNF G + (a set of attribute values attached to each node)
  (possibly empty)

- intrinsic attributes
  Synthesized attributes of leaf nodes whose values are determined outside the parse tree.

ex) Ada procedure definition

```
procedure Name
   (  ≡≡≡  )   } must match
end Name
```

a kind of static semantic
(can not be described with BNF)

↓

(ex) ↓

syntax rule:

<proc_def> → procedure <proc_name>[1] <proc_body> end <proc_name>[2];

(not a part of G)

predicate(rule):

<proc_name>[1].string == <proc_name>[2].string

ex) Checking type rules of Assignment statement

Assume: int and real types only

Attribute G:

- Syntax:
  1. <assign> → <var> = <expr>
  2. <expr> → <var> + <var>
  3. | <var>
  4. <var> → A | B | C

  must be same type

  ex) A = B + C

  A = B

  Assume coercion: int + real ⇒ real, real + int ⇒ real

- attributes for non-T's:
  - "actual_type" — synthesized (associated with <var>, <expr>)
  - "expected_type" — Inherited attribute (associated with <expr>)
- semantic rules — computes attribute values
- predicate (assertions) — semantic error checking

↓ complete attribute G.

1. syntax rule: (inherited)

<assign> → <var>[1] = <expr>

semantic rule:

<expr>.expected_type ← <var>.actual_type

2. syntax rule: (synthesized)

<expr> → <var>[2] + <var>[3]

semantic rule:

<expr>.actual_type ← if((<var>[2].act_T = int) and (<var>[3].act_T = int)) then int else real

predicate:

<expr>.act_T == <expr>.expected_T

3. syntax rule: (synthesized)

<exp> → <var>

Semantic rule:

<exp>.act_T ← <var>.act_T

predicate:

<expr>.act_T == <expr>.expected_T

4. syntax rule:

<var> → A | B | C

semantic rule:

<var>.actual_T ← lookup(<var>.string)

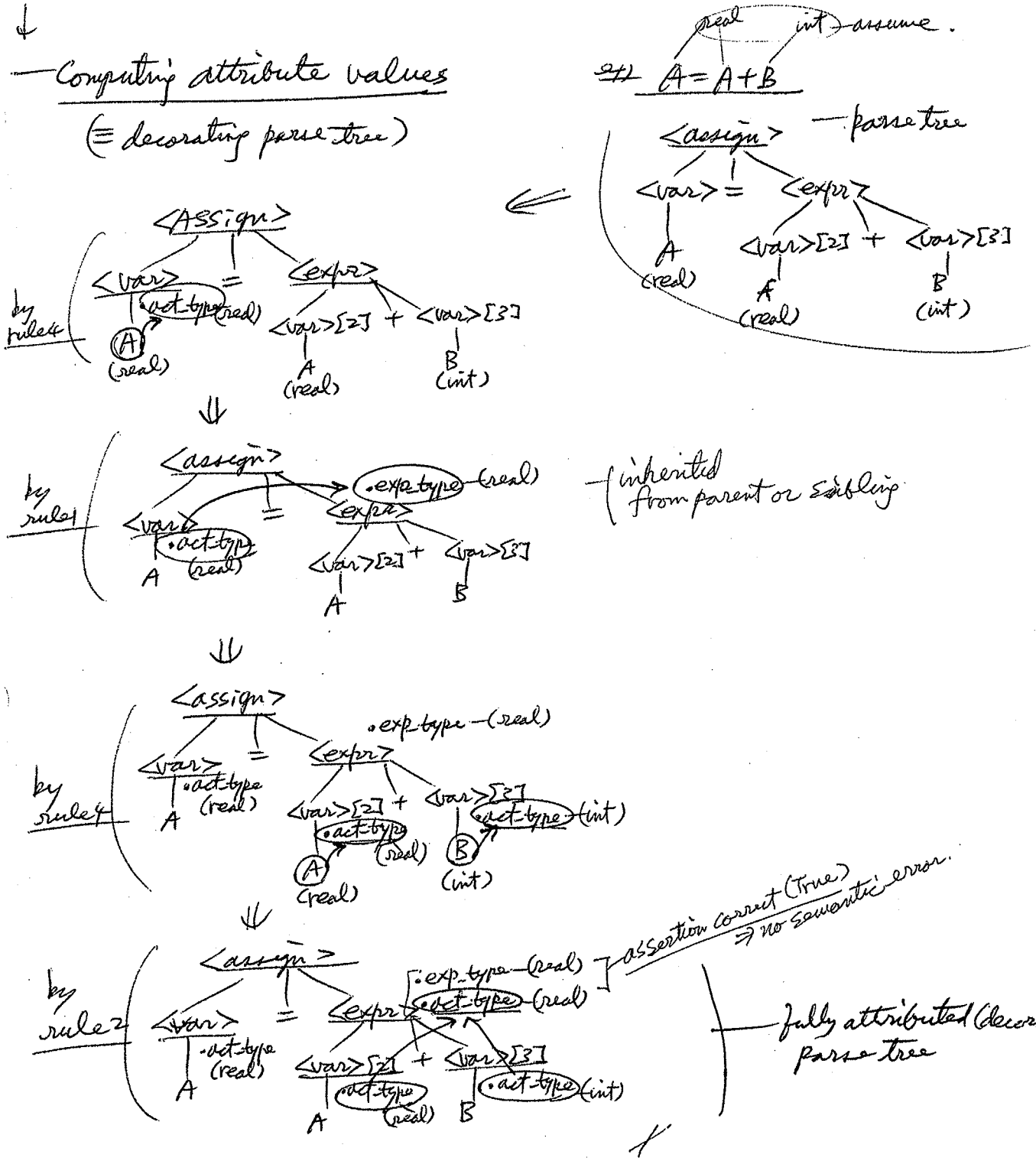
Symbol table

— Computing attribute values

(≡ decorating parse tree)

real int — assume.

eg. A = A + B



— checking the static semantic rules — a part of compilation.

static semantics — and attribute grammar to check it.