Attribute Grammars

Start with BNF grammars and augment them with data and computation.

We may associate with the left and right sides of any BNF rule, some data values and computation.

The data values are called *attributes*. Computation is expressed as so-called *evaluation rules* or *constraint tests*.

Recall that symbols in BNF rules correspond to nodes in parse trees.

We can think of each node in a parse tree as if it were a procedure having:

- Inherited attributes (input parameters)
- Synthesized attributes (output parameters)
- (possibly) Evaluation rules / condidtions
  (computation with attribute values)

Example: Binary integers

The syntax is trivial (we don’t worry about leading 0’s)

We use {a,b} as the binary digits in our language to distinguish them from attribute values 0 and 1.

In BNF: <bin> → <d> | <bin> <d>

<d> → a | b

The grammar is unambiguous; every parse tree is ‘‘left-skewed’’.

In particular, the parse trees for all n-bit strings have identical structure, and only the leaf nodes differ. But there is no information about the actual integer represented.

Let us give an attribute grammar for this language in which an attribute value is synthesized that corresponds to the binary integer represented.

<bin>↑v → <d>↑v

| <bin>↑v′ <d>↑v″ eval ↓v′↓v″↑v

<d>↑v → a↑0 | b↑1

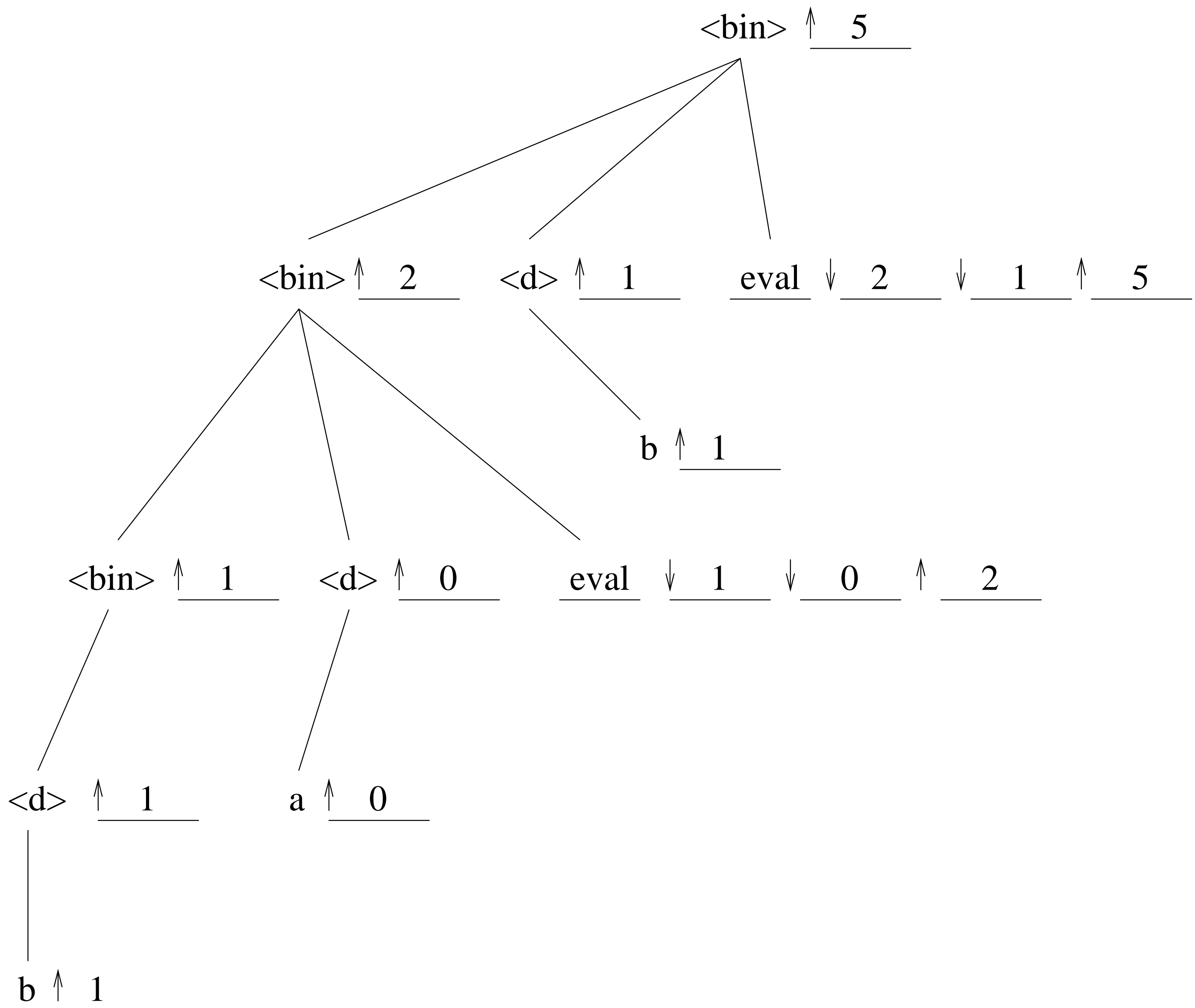
eval↓v′↓v″↑v → {v ← 2*v′ + v″}

Let us construct a parse tree for ‘‘bab’’, i.e., binary 5.

<bin>↑v → <d>↑v
| <bin>↑v′ <d>↑v″ eval ↓v′↓v″↑v

<d>↑v → a↑0 | b↑1

eval↓v′↓v″↑v → {v ← 2*v′ + v″}



We wish to define an assignment statement that allows expressions of "mixed type." Consider the following attribute grammar:

<exp>↑t → <id>↑t
| <id>↑t_1 * <exp>↑t_2 <u>combinetypes:</u> ↓t_1 ↓t_2 ↑t
| <id>↑t_1 / <exp>↑t_2 <u>combinetypes:</u> ↓t_1 ↓t_2 ↑t

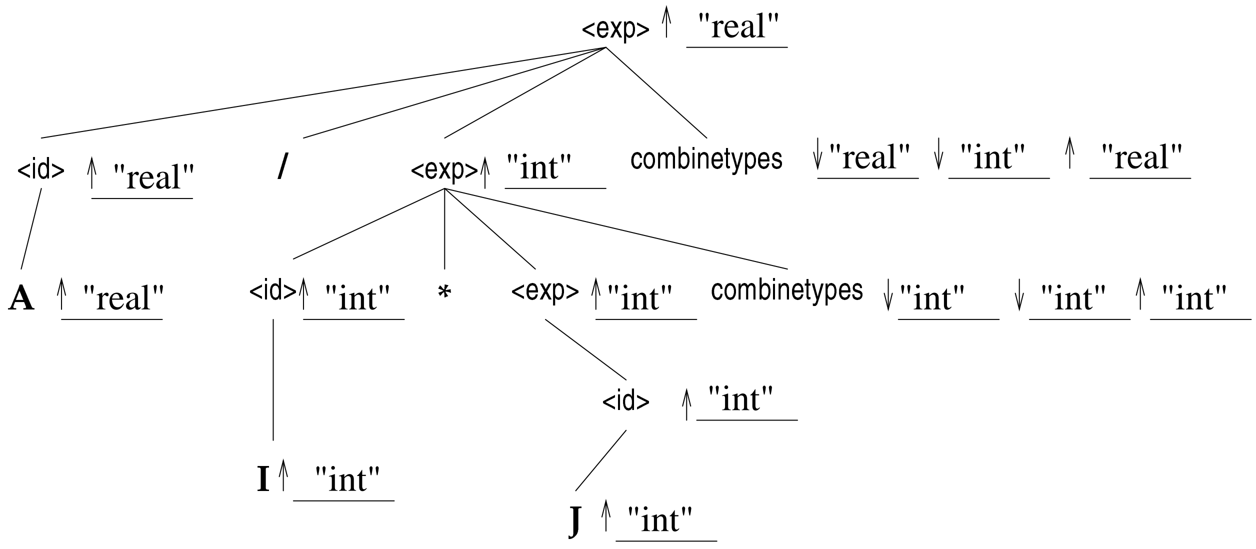
<id>↑t → I ↑"int" | J ↑"int" | · · · | N ↑"int"
| A ↑"real" | B ↑"real" | · · · | H ↑"real"

Evaluation rule:

<u>combinetypes</u> ↓arg_1 ↓arg_2 ↑result →
if arg_1 = "real" then result ← "real" else
if arg_1 = "int" then result ← arg_2

(We use '←' for assignment in the evaluation rule.)

Can we draw a parse tree for "A / I * J" ?



Example: An attribute grammar for doublewords

<dword> → <word>↑w <word2>↓w

<word>↑w → a <word>↑w′ concat↓"a" ↓w′ ↑w
| b <word>↑w′ concat↓"b" ↓w′ ↑w
| a↑"a" | b↑"b"

<word2>↓w → a cond w="a"
| a cond left(w) = "a" assign↑w′ ↓rest(w) <word2>↓w′

<word2>↓w → b cond w="b"
| b cond left(w) = "b" assign↑w′ ↓rest(w) <word2>↓w′

EvaluationRules:

assign↑v ↓exp → {set v to value of exp}

concat↓s_1 ↓s_2 ↑s → {set s to $s_1 s_2$}

left(w) → {1st character in w}

rest(w) → {all but the 1st character in w}

Derivation of "aabaab"

<dword>

<word>↑___<word2>↓___

a <word>↑___ concat↓"a"↓___↑___ <word2>↓___

a a <word>↑___concat↓"a"↓___↑___
concat↓"a"↓___↑___ <word2>↓___

a a b↑"b" concat↓"a"↓"b"↑"ab"
concat↓"a"↓___↑___ <word2>↓___

a a b↑"b" concat↓"a"↓"b"↑"ab"
concat↓"a"↓"ab"↑"aab" <word2>↓"aab"

Now the first half of the doubleword is derived.
We continue with the derivation from <word2>.

<word2>↓"aab"

<word2>↓"aab"

a cond left ("aab")="a" assign↑w′ ↓rest("aab") <word2>↓w′

a cond left ("aab")="a" assign↑"ab" ↓rest("aab") <word2>↓"ab"

The condition and assign rules are complete, and so are omitted.
Remember that aab was derived from <word>, so we have
aab a <word2>↓"ab"

aab a a cond left ("ab")="a"
assign↑w′ ↓rest("ab") <word2>↓w′

aab a a cond left ("ab")="a"
assign↑"b" ↓rest("ab") <word2>↓"b"

aab a a cond left ("ab")="a"
assign↑"b" ↓rest("ab") b cond "b" = "b"

Parse tree for aabaab

<dword>

<word>↑___

a <word>↑___ concat ↓___ ↓___ ↑___

a <word>↑___ concat ↓___ ↓___ ↑___

b ↑___

<word2>↓___

a cond left ("aab")="a" assign ↑___ ↓___ <word2>↓___

a cond left ("ab")_="a" assign ↑___ ↓___ <word2>↓___

b cond "b"=

Parse tree for aabaab