ATTRIBUTE GRAMMARS

An attribute grammar is a formal way to supplement a formal grammar with semantic information processing. Semantic information is stored in attributes associated with terminal and nonterminal symbols of the grammar. The values of attributes are result of attribute evaluation rules associated with productions of the grammar. Attributes allow to transfer information from anywhere in the abstract syntax tree to anywhere else, in a controlled and formal way

Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse. In simple applications, such as evaluation of arithmetic expressions, attribute grammar may be used to describe the entire task to be performed besides parsing in straightforward way; in complicated systems, for instance, when constructing a language translation tool, such as a compiler, it may be used to validate semantic checks associated with a grammar, representing the rules of a language not explicitly imparted by the syntax definition. It may be also used by parsers or compilers to translate the syntax tree directly into code for some specific machine, or into some intermediate language.

Definition:

An attribute grammar is a context-free grammar G = (S, N, T, P) with the following additions:

- For each grammar symbol x there is a set A(x) of attribute values
- Each rule has a set of functions that define certain attributes of the non-terminals in the rule
- Each rule has a (possibly empty) set of predicates to check for attribute consistency

- Let X0 → X1 ... Xn be a rule
- Functions of the form S(X0) = f(A(X1), ... , A(Xn)) define synthesized attributes
- Functions of the form I(Xj) = f(A(X0), ... , A(Xn)), for i <= j <= n, define inherited attributes Initially, there are intrinsic attributes on the Leaves

Syntax rule:

<proc_def> → procedure <proc_name>[1]

<proc_body> end <proc_name>[2];

Predicate:

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

Syntax

<assign> → <var> = <expr>

<expr> → <var> + <var> | <var>

<var> → A | B | C

•actual_type: synthesized for <var> and <expr>

•expected_type: inherited for <expr>

Syntax rule: <expr> → <var>[1] + <var>[2]

Semantic rules:

<expr>.actual_type ← <var>[1].actual_type

Predicate:

<var>[1].actual_type == <var>[2].actual_type

<expr>.expected_type == <expr>.actual_type

Syntax rule:

<var> → id Semantic rule: <var>.actual_type ← lookup (<var>.string)

How are attribute values computed?

– If all attributes were inherited, the tree could be decorated in top-down order.

– If all attributes were synthesized, the tree could be decorated in bottom-up order.

– In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)

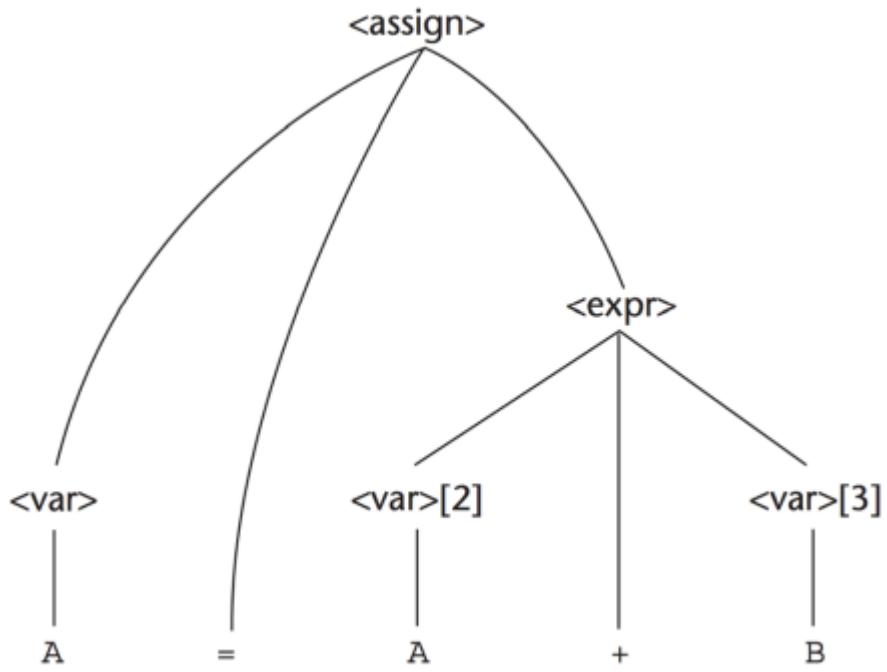
<var>[2].actual_type ← lookup (B)

<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type

<expr>.actual_type =?<expr>.expected_type

Parse Tree



Computing Attribute Values

1. <var>.actual_type ← look-up(A) (Rule 4)

2. <expr>.expected_type ← <var>.actual_type (Rule 1)

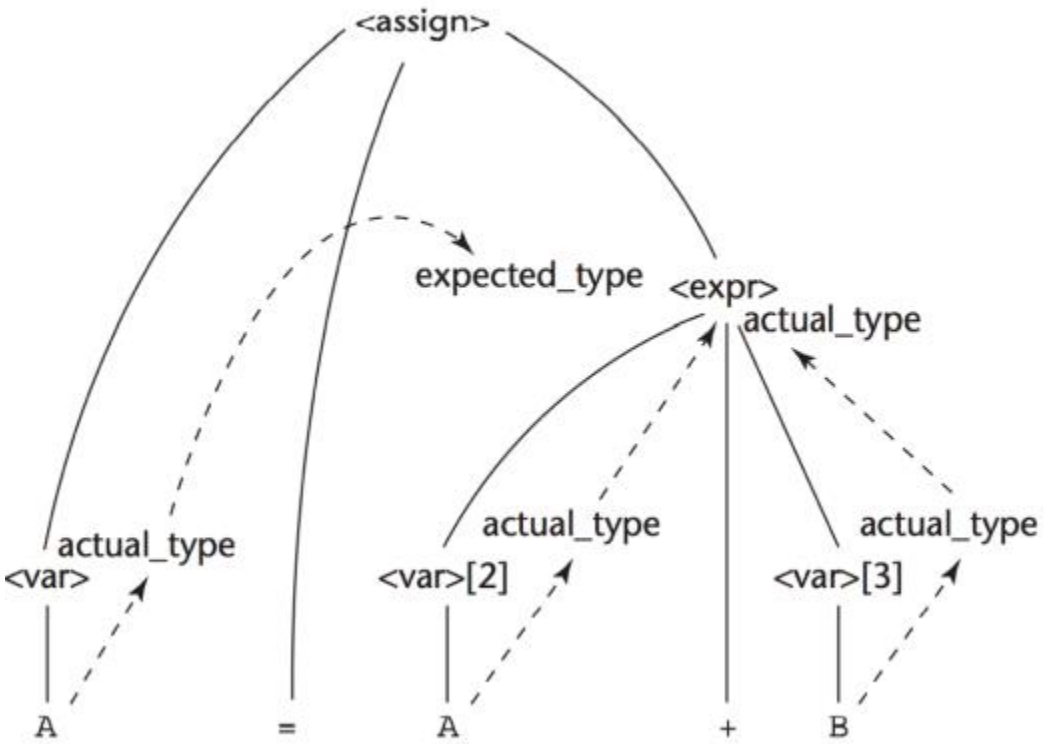
3. <var>[2].actual_type ← look-up(A) (Rule 4)

<var>[3].actual_type ← look-up(B) (Rule 4)

4. <expr>.actual_type ← either int or real (Rule 2)

5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE (Rule 2)

Flow of Attributes in the Tree



A Fully Attributed Parse Tree