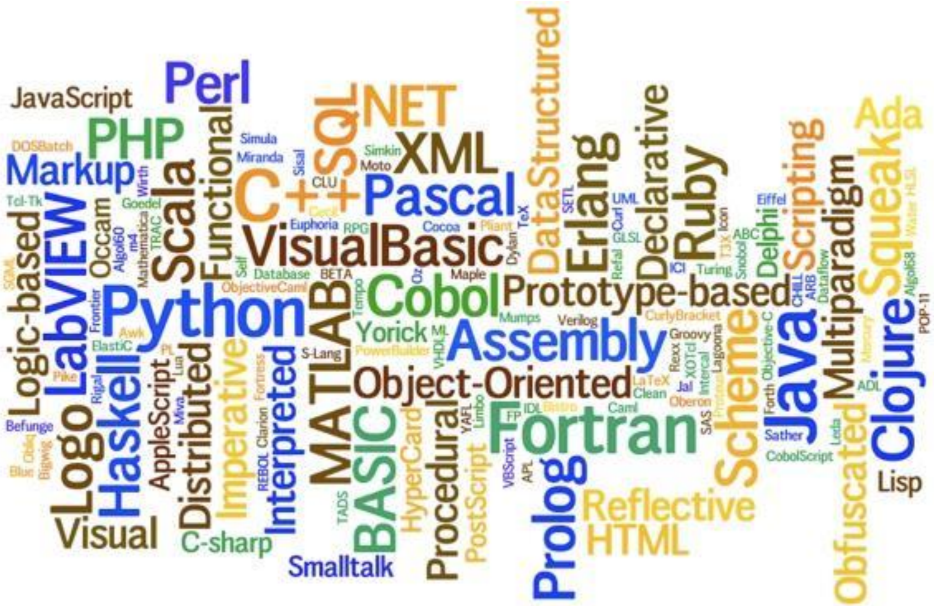


Programming Languages Semantics I

Theory and fundamentals of Programming Languages

Module 4

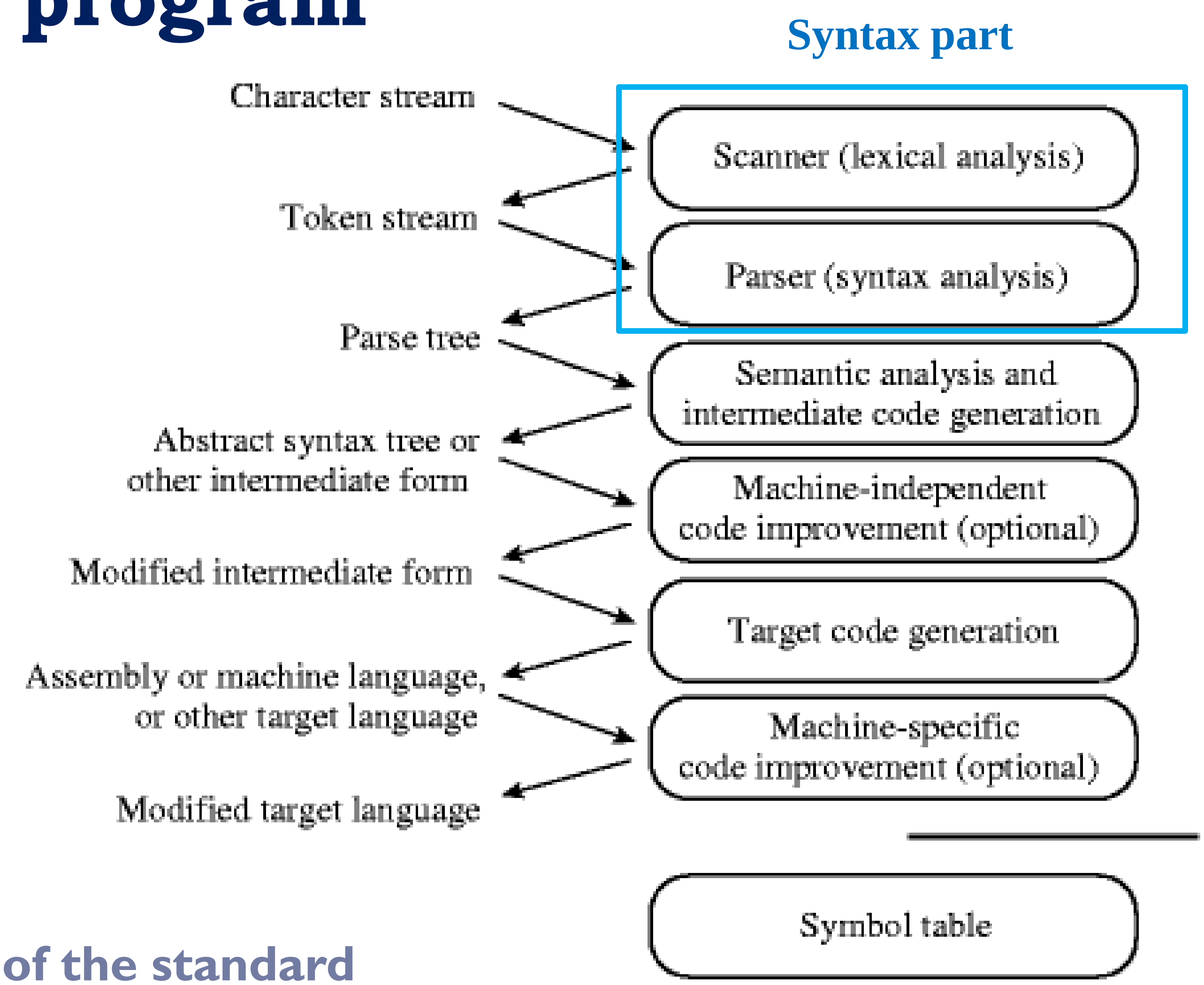
Dr. Tamer ABUHMED

SUNG KYUN KWAN UNIVERSITY

Outline

- Semantics Overview
- Semantics purpose: Program Verification
- Static Semantics
- Attribute grammars
  - Example

Source to program



This is an overview of the standard process of turning a text file into an executable program.

Semantics Overview

- Syntax is about *form* and semantics *meaning*
  - Boundary between syntax & semantics is not always clear
- First we motivate why semantics matters
- Then we look at issues close to the syntax end (e.g., *static semantics*) and *attribute grammars*
- Finally we sketch three approaches to defining "deeper" semantics:
  (1) Operational semantics
  (2) Axiomatic semantics
  (3) Denotational semantics

Motivation

- Capturing what a program in some programming language means is very difficult
- We can't really do it in any practical sense
  - For most work-a-day programming languages (e.g., C, C++, Java, Perl, C#, Python)
  - For large programs
- So, why is worth trying?

Motivation: Some Reasons

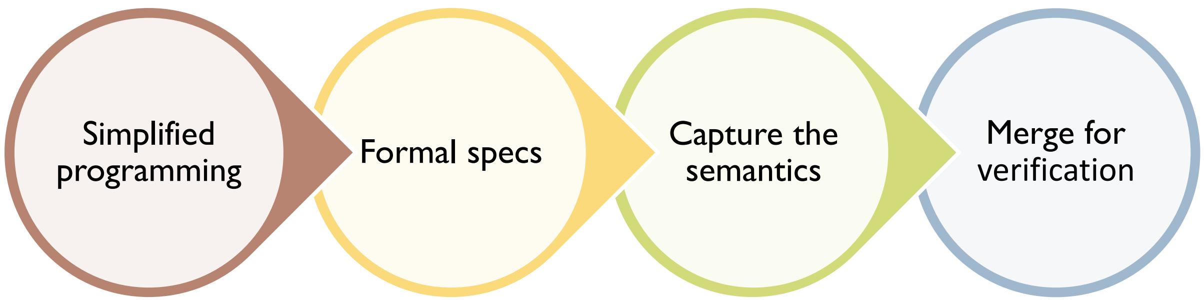
- To inform the programming language compiler/interpreter writer what she should do
  - Natural language may be too ambiguous
- To know that the compiler/interpreter did the *right thing* when it executed our code
  - We can't answer this w/o a solid idea of what the *right thing* is
- To ensure the program satisfies its specification
  - Maybe we can do this automatically if we know what the program means

Program Verification

- Program verification involves formally proving that the computer program does exactly what is stated in the program's specification
- Program verification can be done for simple programming languages and small or moderately sized programs
- Requires a *formal specification* for what the program should do – e.g., its inputs and the actions to take or output to generate
- That's a hard task in itself!

Program Verification

- There are applications where it is worth it to
  (1) use a simplified programming language
  (2) work out formal specs for a program
  (3) capture the semantics of the simplified PL and
  (4) do the hard work of putting it all together and proving program correctness



- What are they?

Program Verification

- There are applications where it is worth it to (1) use a simplified programming language, (2) work out formal specs for a program, (3) capture the semantics of the simplified PL and (4) do the hard work of putting it all together and proving program correctness. Like…
- Security and encryption
- Financial transactions
- Applications on which lives depend (e.g., healthcare, aviation)
- Expensive, one-shot, un-repairable applications (e.g., Martian rover)
- Hardware design (e.g. Pentium chip)

Double Int kills Ariane 5

- The EU Space Agency spent ten years and $7B to produce Ariane 5, a giant rocket capable of putting a pair of three-ton satellites into orbit with each launch and intended to give Europe supremacy in the commercial space business

- All it took to explode the rocket less than a minute into its maiden voyage in 1996 was a small computer program trying to stuff a 64-bit number into a 16-bit space.

Intel Pentium Bug

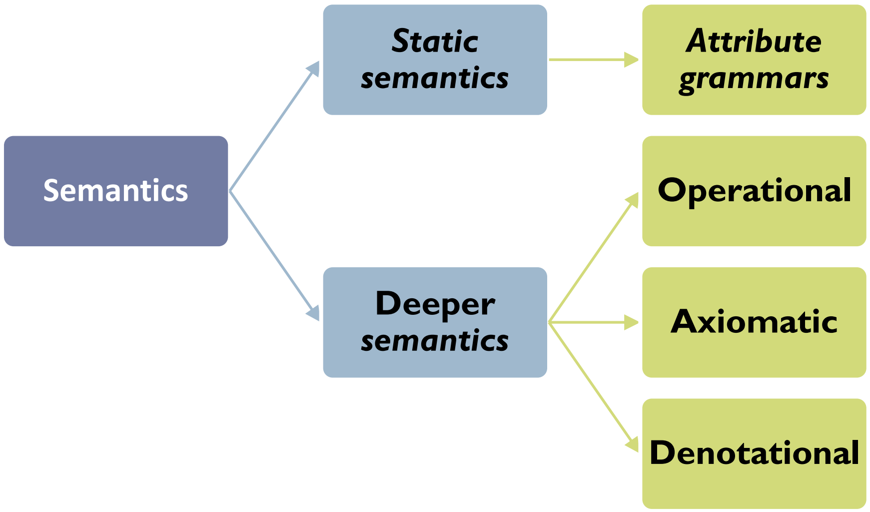
- In the mid 90's a bug was found in the floating point hardware in Intel's latest Pentium microprocessor
- Unfortunately, the bug was only found after many had been made and sold
- The bug was subtle, effecting only the ninth decimal place of *some* computations
- But users cared
- Intel had to recall the chips, taking a $500M write-off

So…

- While automatic program verification is a long range goal …
- Which might be restricted to applications where the extra cost is justified
- We should try to design programming languages that help, rather than hinder, verification
- We should continue research on the semantics of programming languages …
- And the ability to prove program correctness

Semantics in general

- Next we look at issues close to the syntax end, what some calls *static semantics*, and the technique of *attribute grammars*
- Then we sketch three approaches to defining "deeper" semantics
  (1) Operational semantics
  (2) Axiomatic semantics
  (3) Denotational semantics



Static Semantics

- Static: concerned with text of program, not with what changes when the program runs
- Can cover language features impossible or difficult to handle in a CFG
- A mechanism for building a parser producing an abstract syntax tree from its input
- Attribute grammars are a common technique that can handle language features
  - Context-free but cumbersome (e.g., type checking)
  - Non-context-free (e.g., variables must be declared before used)

CFG: context free grammar

Static Semantics: Attribute grammars

Checks of many kinds

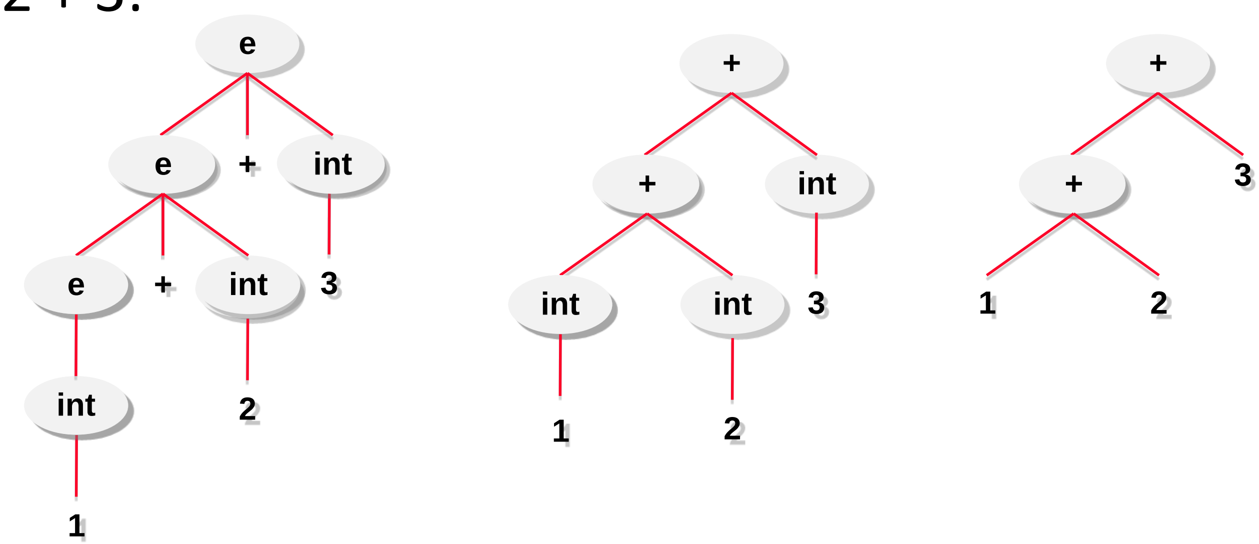
- All identifiers are declared
- Types checking
- Inheritance relationships
- Classes defined only once
- Methods in a class defined only once
- Reserved identifiers are not misused
- etc.

- The requirements depend on the language

Parse tree vs. abstract syntax tree

- Parse trees follow a grammar and usually have many nodes that are artifacts of how the grammar was written
- An abstract syntax tree (AST) eliminates useless structural nodes
- Use nodes corresponding to constructs in the programming language, easing interpretation and compilation
- Consider 1 + 2 + 3:

```
e -> e + e
e -> int
int -> 1
int -> 2
int -> 3
```



parse tree   an AST   another AST

Attribute Grammars

- Attribute Grammars (AGs) were developed by Donald Knuth in ~1968

- Motivation:
  - CFGs can't describe all of the syntax of programming languages
  - Additions to CFGs to annotate the parse tree with some "semantic" info
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

CFG: context free grammar

Attribute Grammar Example

- Ada's rule to describe procedure definitions:

  <proc> => procedure <prName> <prBody> end <prName> ;
- The name after *procedure* must be the same as the name after *end*
- Can't be expressed in a CFG (in practice) because there are too many names
- Solution: annotate parse tree nodes with attributes; add constraints to the syntactic rule in the grammar

rule: <proc> => procedure <prName>[1] <prBody> end <prName>[2] ;

constraint: <prName>[1].string == <prName>[2].string

Attribute Grammars

- Def: An *attribute grammar* is a CFG 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

A Grammar is formally defined by specifying four components.

- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols
- P is a set of productions or rules

Attribute Grammars

- Let $X_0 \Rightarrow X_1 \ldots X_n$ be a grammar rule
- Functions of the form $S(X_0) = f(A(X_1),\ldots A(X_n))$ define *synthesized attributes*
  - i.e., attribute defined by a nodes children
- Functions of the form $I(X_j) = f(A(X_0),\ldots A(X_n))$ for i <= j <= n define *inherited attributes*
  - i.e., attribute defined by parent and siblings
- Initially, there are *intrinsic attributes* on the leaves
  - i.e., attribute predefined

Attribute Grammars: Example 3.6

EXAMPLE 3.6 An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

The look-up function looks up a given variable name in the symbol table and returns the variable's type.

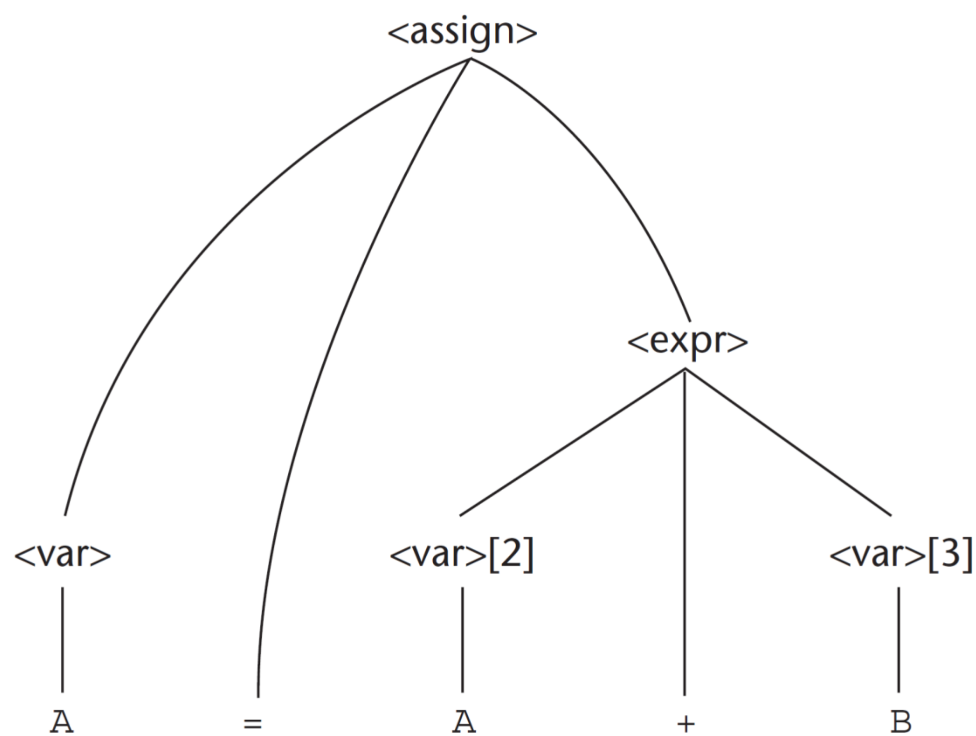
EXAMPLE 3.6 An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

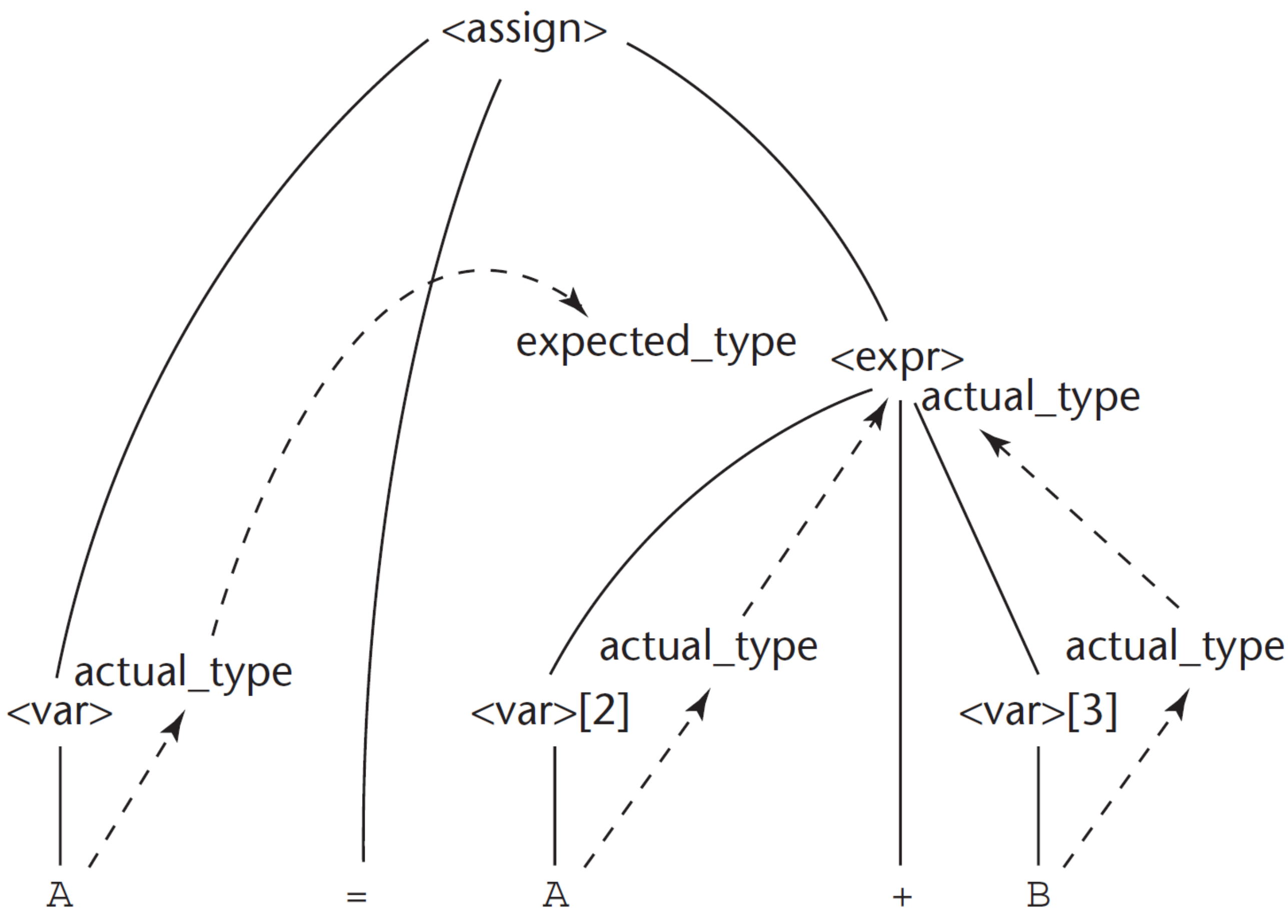
The look-up function looks up a given variable name in the symbol table and returns the variable's type.

Figure 3.6

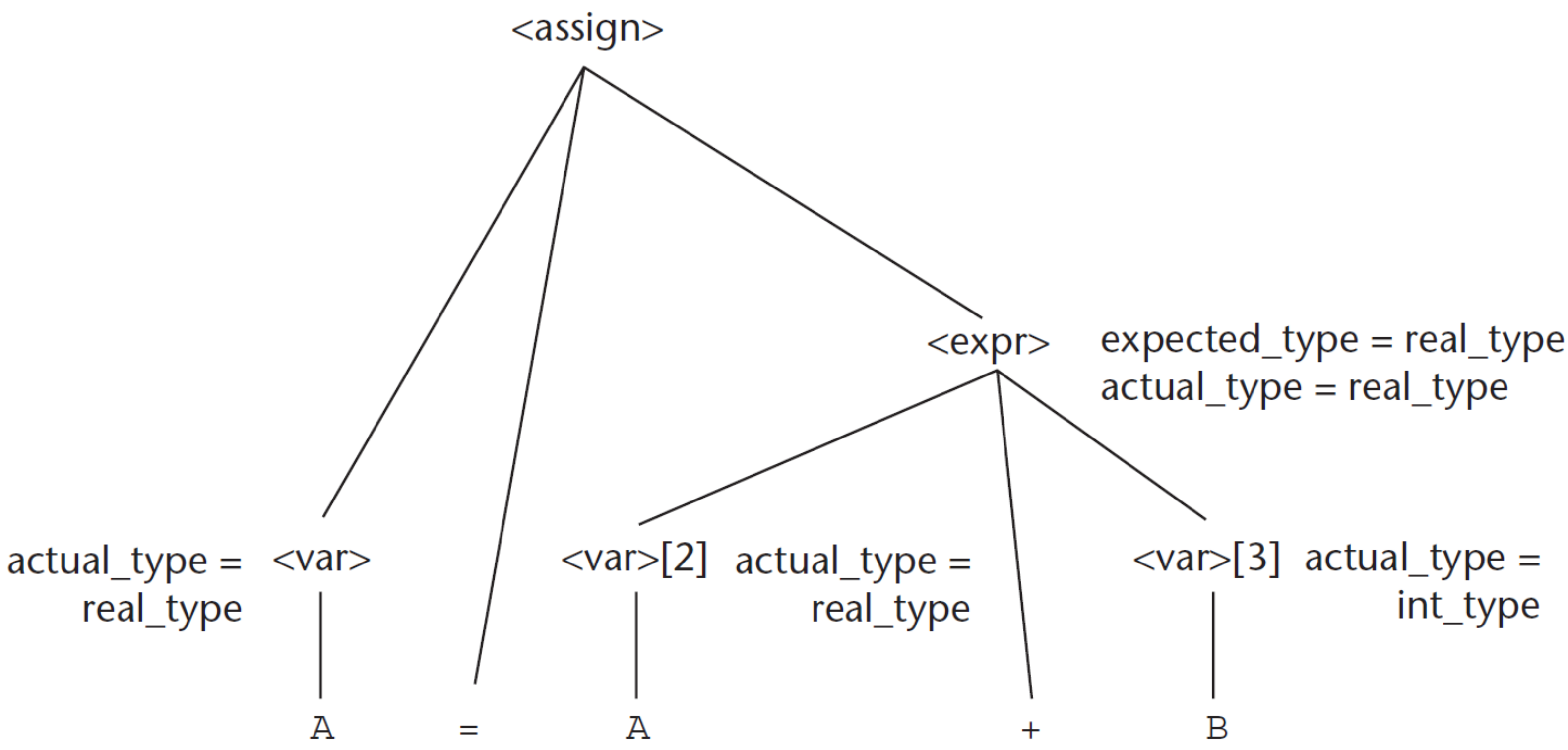
A parse tree for
`A = A + B`



The flow of attributes in the tree



A fully attributed parse tree



Attribute Grammars

- *Example:* expressions of the form id + id
  - id's can be either int_type or real_type
  - types of the two id's must be the same
  - type of the expression must match its expected type
- *BNF:* <expr> -> <var> + <var>

  <var> -> id
- *Attributes:*

actual_type - synthesized for <var> and <expr>

expected_type - inherited for <expr>

Attribute Grammars

- *Attribute Grammar:*

1) 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

2) Syntax rule: <var> -> id

- Semantic rule:
  - <var>.actual_type ←
    lookup_type (id, <var>)

Compilers usually maintain a "symbol table" where they record the names of procedures and variables along with type information. Looking up this information in the symbol table is a common operation.

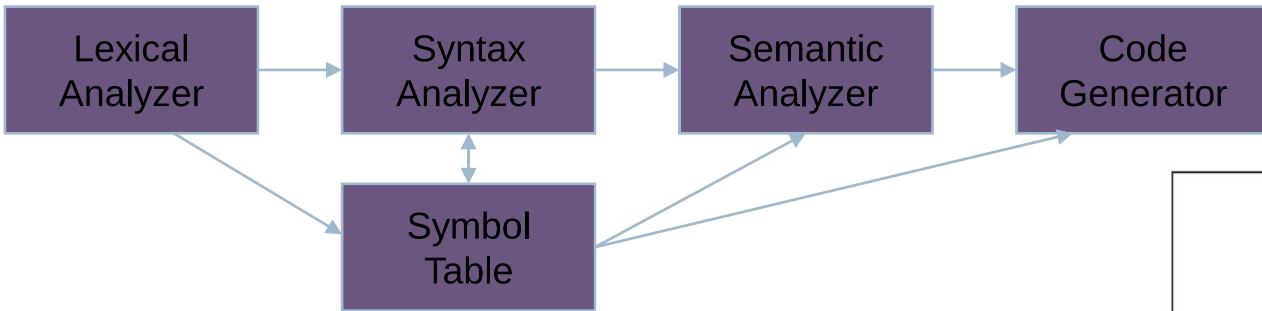
Attribute Grammars (continued)

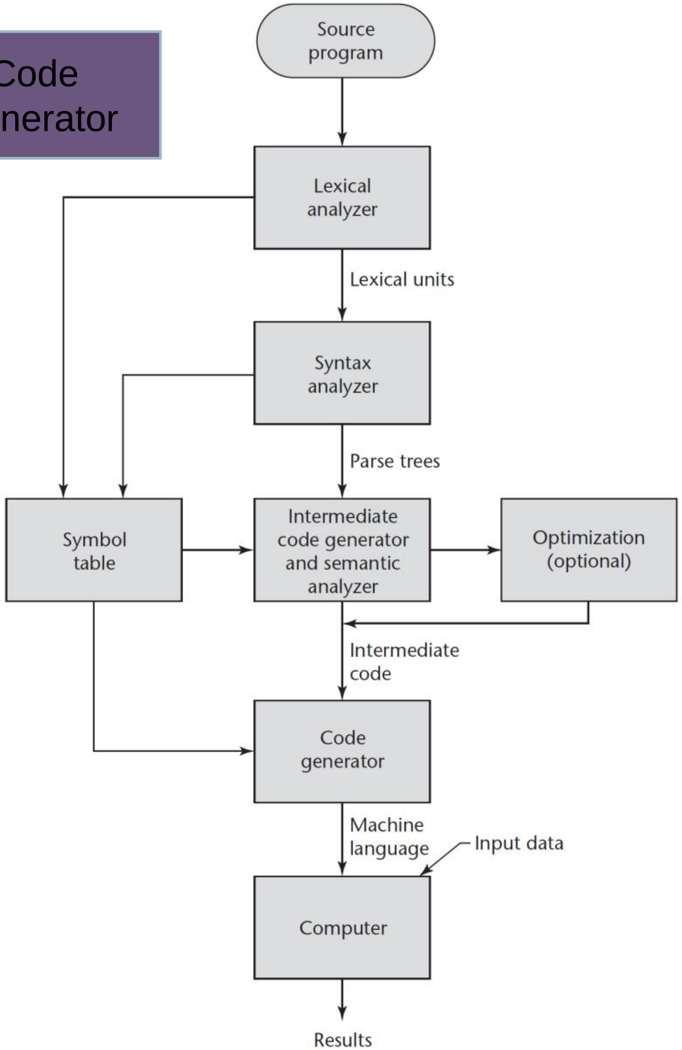
- *How are attribute values computed?*
  - If all attributes were inherited, the tree could be *decorated* in top-down order
  - If all attributes were synthesized, the tree could be *decorated* in bottom-up order
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used

Attribute Grammars (continued)

- Suppose we process the expression A+B using rule <expr> -> <var>[1] + <var>[2]
  - <expr>.expected_type ← inherited from parent
  - <var>[1].actual_type ← lookup (A, <var>[1])
  - <var>[2].actual_type ← lookup (B, <var>[2])
  - <var>[1].actual_type == <var>[2].actual_type
  - <expr>.actual_type ← <var>[1].actual_type
  - <expr>.actual_type == <expr>.expected_type

Symbol tables





Symbol	Kind	Type	Properties
value	field	int	…
test	method	-> int	
setValue	method	int -> void	

Symbol tables

```
class Foo {
  int value;
  int test() {
    int b = 3;
    return value + b;
  }
  void setValue(int c) {
    value = c;
    { int d = c;
      c = c + d;
      value = c;
    }
  }
}
```

`Class Foo symbol table`

Symbol	Kind	Type	Properties
value	field	int	…
test	method	-> int	
setValue	method	int -> void	

Attribute Grammar Summary

- Practical extension to CFGs allowing parse trees annotation with information needed for semantic processing
  - e.g., interpretation or compilation
- The annotated tree is an *abstract syntax tree*
  - It no longer just reflects the derivation
- AGs can move information from anywhere in abstract syntax tree to anywhere else
  - Needed for no-local syntactic dependencies (e.g., Ada example) and for semantics

Summary

- Semantics Overview
- Semantics purpose: Program Verification
- Static Semantics
- Attribute grammars