Department of Computer and Information Sciences

CSC318
COMPILER CONSTRUCTION 1

Lecture: SEMANTIC ANALYSIS

Outline

1. The Phases of Compiler - So Far…
2. The Semantic Analysis
3. Abstract Syntax Tree (AST)
4. Attribute Grammar
5. Decorating AST

1. THE PHASES OF COMPILER - So Far …

1. Lexical Analysis (Scanning)
   - Detects inputs with illegal tokens
   - Incorrect spelling of keywords
     - e.g.: main $ (); devine

2. Syntax Analysis (Parsing)
   - Detects inputs with ill-formed parse trees
     - e.g.: missing semicolons ";" … missing brace " }"

3. Semantic Analysis
   - Last "front end" phase
   - Catches all remaining errors. e.g.: operand is incompatible to operator

The Classes of Errors

1. Lexical Errors
   - Detected by the Scanner
     - Illegal Character Input
     - Illegal Comments
     - Unterminated String constants
     - Infinite Loop

2. Syntax Errors
   - Detected by the Parser
     - Input is not a legal program, because it cannot be parsed by CFG
     - Example: a = * 5

The Classes of Errors (2)

3. Static Semantic Errors

- Detected by a Parser or in a Separate Semantic Analysis Passes
  - Input can be parsed by CFG but some non context-free error

    Examples:

    - Multiple declared variables
    - Undeclared variables
    - Function call with wrong number of arguments
    - Type mismatches

The Classes of Errors (3)

3. Static Semantic Errors:
   - multiple declarations: a variable should be declared (in the same scope) at most once.
   - undeclared variable: a variable should not be used before being declared.
   - type mismatch: type of the left-hand side of an assignment should match the type of the right-hand side.
   - wrong arguments: methods should be called with the right number and types of arguments.

The Classes of Errors (4)

3b. Semantic Errors

- Detected at Compile Time, and also add checks in code for runtime.

  Examples:

  - Division by 0
  - Array Index out-of-bound
  - Dereference of NULL pointer

- Can include code to check for conditions and not allow them to occur

  Advantage:

  - error message instead of error behavior

  Disadvantage:

  - object code is longer and slower

The Classes of Errors (5)

4. Logical Errors
   - Hardest to detect
   - Program is syntactically and semantically correct but does not do the "correct" thing
   - Compiler can sometimes detect this problem.

Examples:

```
if (x = 0) result = 1;
result = 2;
```

- Assignment in the body of the if is "useless" – immediately overwritten by next assignment

2. SEMANTIC ANALYSIS

What is Semantic Analysis?

- Semantic analysis is the task of ensuring that the declarations and statements of a program are ***semantically*** correct, i.e, that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

What Does Semantic Analysis Involve?

- Semantic Analysis typically involves:
  1. Type Checking:
     - Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
     - Error if operands are incompatible with the used operator.
     - Example: 1.2 + 2 (real + int).
  2. Label Checking:
     - Labels references in a program must exist.
  3. Flow control checks:
     - control structures must be used in their proper fashion.
     - E.g. No GOTOs into a FORTRAN DO statement, No breaks outside a loop or switch statement, break needs an enclosing loop, goto label needs a defined label. etc.)

Semantic Analysis – Type Checking

- Semantic Analysis typically involves:
  1. Type Checking:
     - Type Systems
     - Type Equivalence
     - Typing Expressions
       - Coercion
       - Overloading
       - Error Recovery
     - Typing Statements
     - Polymorphic Types

Type Checking (2)

- Type Systems: System rules (Rules of a Language)
  - E.g: Check "If both operands of the arithmetic operators of addition, subtraction and multiplication are of type integer, then the result is of type integer
  - Definition:
    - What are the base/immutable types?
    - What type of constructors are available?
    - Can types be named? Or renamed?
  - Resolutions:
    - What operators can be applied to what type?
    - What forms of coercion are allowed?
    - How are overloading situations resolved?

Type Checking (3)

- **Base/Immutable Types:**
  - Generally objects with a direct machine representation, where the objects can not be further divided
    - Examples: bool, char, int, float, double, long int, unsigned int, …
  - Simple objects have direct types
    - Literals: 3 (int), -5.0 (double)
    - Variables: int x; (int) double y; (double)

Type Checking (4)

- **Type Equivalence**:
  - Name equivalence – objects are considered to be equivalent if they have the same (or in the case of operators – appropriate names):
    - Problem – if a type name is given to a type (Number declared a synonym for int) this may introduce type errors that are not real errors.
  - Structural equivalence – objects are considered equivalent if they have similar structures. This is useful but can allow some mappings which we may not want:

```
class x { int y; float z;};

class position { int angle, float distance; };
```

    - x and position would be considered to be structurally equivalent

Type Checking (5)

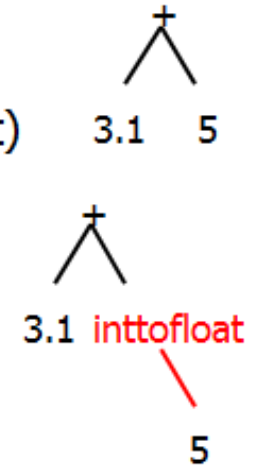
- **Typing Expressions**
  - Deals with expressions, possibly complex expressions where we assume the expressions will result in type. For Example:
    - Type IntLitNode::expr_type() { return int; }
    - Type BoolLitNode::expr_type() { return int; }
    - Type VariableNode::expr_type() { return type of variable; }
- **Typing Statements**
  - Statements checking causes expression type checks.
  - Parts of statement often must return type.

Type Checking (6)

Coercion

- It is reasonable (and in most cases desirable) to allow some automatic coercion (ints to floats for an addition)
- When do we do it? - During expression type checking

```
if (op is +) {
  if (lefttype is float) and (righttype is int)
    insert a inttofloat in right child
}
```



Type Checking (7)

- **Overloading**
  - Allows name for function to be inserted into symbol table multiple times if type for parameter list (product) differs.
  - When looking up function to be called match argument type to each of the possibilities and pick the one that matches
  - Complicating issue – when coercion is allowed matches may not be perfect
    - If two possibilities are close, which one to choose

Example:

- Definitions:

  foo : int X float -> int

  foo : float X int -> float

- Which to choose when matching arguments **int X int**?

Type Checking (8)

- **Error Recovery**
  - As with parsing, often want to find multiple errors
  - What to do when one error detected?
    - Generally, return error as type but don't generate further errors
    - E.g., left argument of + returns error, still want the right argument to be some type that can be added.
  - Error recovery Modes:
    - Panic Mode: Skip symbols until input can by synchronized to a token
    - Phrase-level Recovery: Local error corrections, e.g. replacement of "," by a ";"
    - Error Productions: Extension of grammar to handle common errors.
    - Global correction: Minimal correction of program in order to find a matching derivation (cost intensive)

Type Checking Situations

- Some situations under which type checking is carried out are:
  - Typing Expressions
    - Operand Matching
    - Operand Selection
  - Coercion Types
  - Selecting among Overload Possibilities
  - Polymorphic Types Expansion

Forms Type Checking

- Static Type Checking - type checking done at compile type
  - Used in many strongly typed languages (where all variables/objects must have types).
- Dynamic Type Checking – done at runtime
  - Often used in languages where objects not strongly type (e.g., Lisp).
  - Type checking must be done at runtime (since objects not guaranteed to have a single type)

When To Type Check

- Depending on language
  - but some type checks can often be done in parsing
- Can also be done as a separate process
  - If done as a separate process, performed as a traversal of the Abstract Syntax Trees (AST) from the program
    - Generally two routines:
      - Type of expressions
      - Type of statements

2.1 The Semantic Analyzer

- The Semantic Analyzer traverses the AST generated by the Syntax Analyzer (Parser), two (2) phases:

1. For each scope in the program:
   - process the declarations =
     - add new entries to the symbol table and
     - report any variables that are multiply declared
   - process the statements =
     - find uses of undeclared variables, and
     - update the "ID" nodes of the AST to point to the appropriate symbol-table entry.
2. Process all of the statements in program again:
   - use the symbol-table information to determine the type of each expression, and to find type errors.

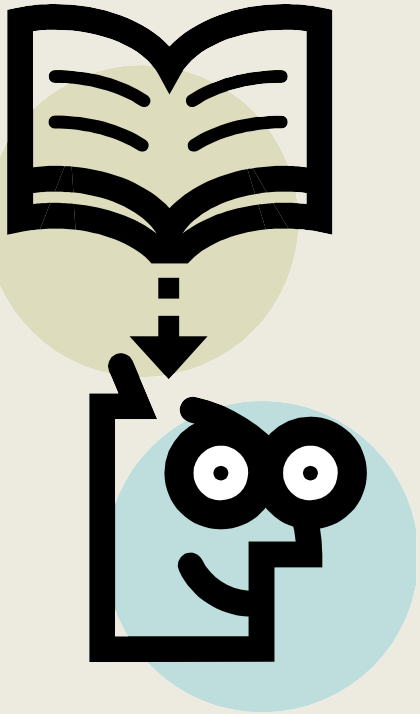
The Symbol Table

- The Symbol Table is the Set of Entries:

1. Keeps track of names declared in the program
2. Stores names of:
   - variables, classes, fields, methods, etc

3. Symbol table entry:
   - associates a name with a set of attributes, e.g.:
     - kind of name (variable, class, field, method, etc)
     - type (int, float, etc)
     - nesting level
     - memory location (i.e., where will it be found at runtime).

The Symbol Table



Please Read Up More On
SYMBOL TABLE
IMPLEMENTATION
(See Upload Note On MOODLE)

What Does Semantic Analysis Produce?

- Part of semantic analysis is producing some sort of representation of the program –
  - either object code or an intermediate representation of the program.
- One-pass compilers will generate object code without using an intermediate representation;
  - code generation is part of the semantic actions performed during parsing.
- Other compilers (e.g. JAVA) will produce an intermediate representation during semantic analysis;
  - most often it will be an AST or quadruples.

Semantic Analysis Example

Semantic Actions in Top-Down Parsing: An Example

Imagine we're parsing:

$S \rightarrow id := E$
$E \rightarrow T\ E'$
$E' \rightarrow + T\ E'$
$E' \rightarrow \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow * F\ T'$
$T' \rightarrow \varepsilon$
$F \rightarrow id$
$F \rightarrow const$
$F \rightarrow (E)$

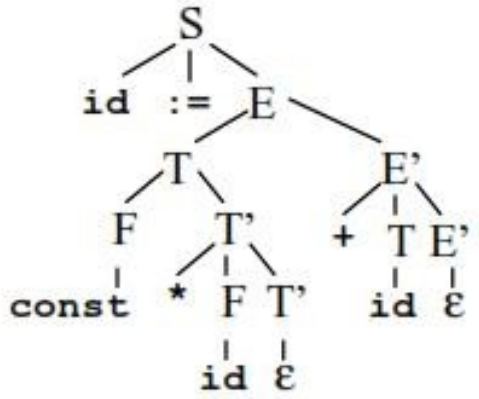
We insert the actions

$S \rightarrow id\ \{pushid\} := \{pushassn\}\ E \{buildassn\}$
$E \rightarrow T\ E'$
$E' \rightarrow +\ \{pushop\}\ T\ \{buildexpr\}\ E'$
$E' \rightarrow \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ \{pushop\}\ F\ \{buildterm\}\ T'$
$T' \rightarrow \varepsilon$
$F \rightarrow id\ \{pushid\}$
$F \rightarrow const\ \{pushconst\}$
$F \rightarrow (E)\ \{pushfactor\}$

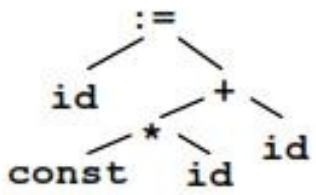
Semantic Analysis Example (2)

Example: Parsing An Expression

In parsing the expression `z := 2 * x + y`, we find this parse structure:



We want to create the AST fragment:



Semantic Analysis Example (3)

Parsing `z := 2*x + y` (continued)

Or we can produce a set of quadruples:

t_1 := 2 * x
t_2 := t_1 + y
z := t_2

Or we an produce a set of assembly language instructions:

```
mov  ax, 2
mov  bx, y
imul bx
mov  z, ax
```

3.0 Building the AST Example

1

Building the AST

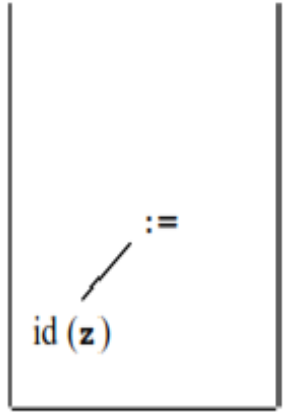
`z := 2 * x + y` S → id {*pushid*} := {*pushassn*} E {*buildassn*}

Semantic Stack

2

Building the AST (continued)

`z := 2 * x + y` S → id {*pushid*} := {*pushassn*} E {*buildassn*}



Semantic Stack

Building the AST Example

3

Building the AST (continued)

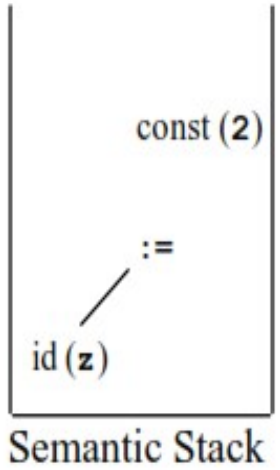
`z := 2 * x + y`

S → id {*pushid*} := {*pushassn*} E {*buildassn*}

E → T E'

T → FT'

F → const {*pushconst*}



Semantic Stack

4

Building the AST (continued)

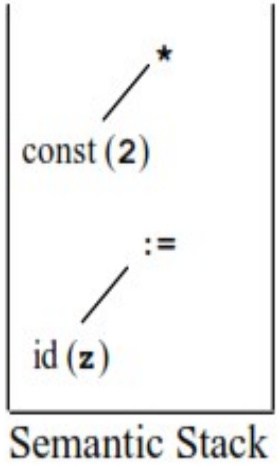
`z := 2 * x + y`

S → id {*pushid*} := {*pushassn*} E {*buildassn*}

E → T E'

T → FT'

T' → * {*pushop*} F {*buildterm*} T'

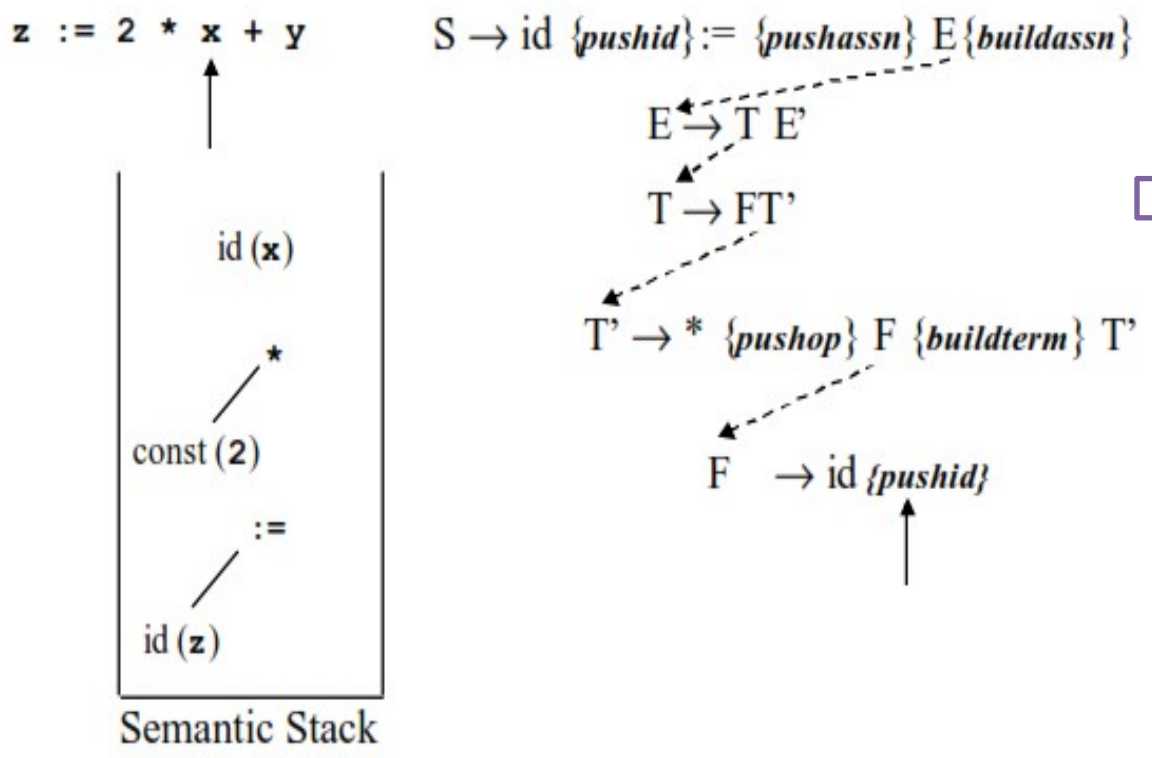


Semantic Stack

Building the AST Example

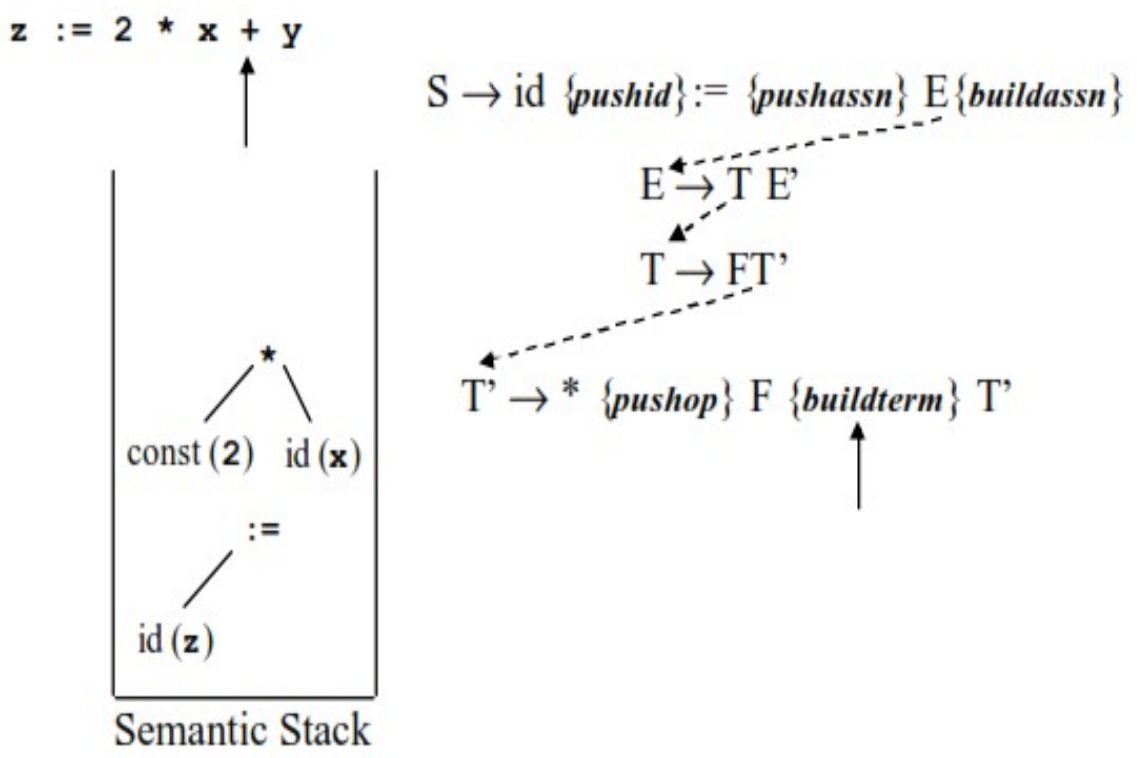
5

Building the AST (continued)



6

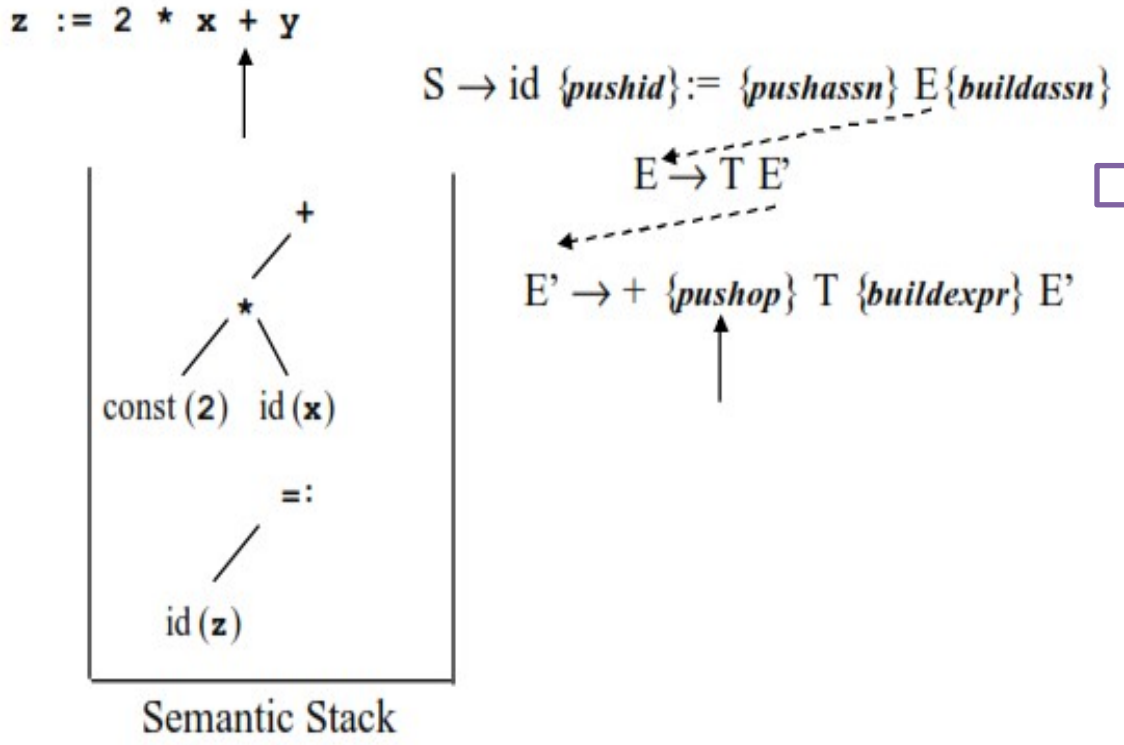
Building the AST (continued)



Building the AST Example

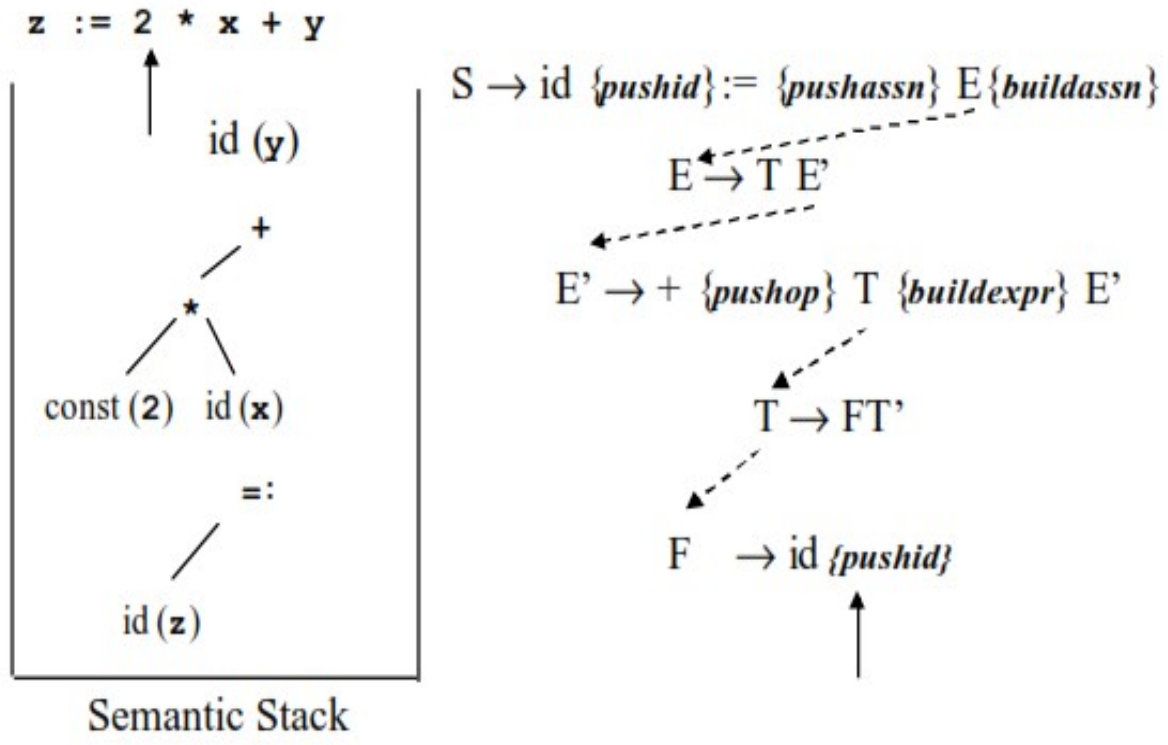
7

Building the AST (continued)



8

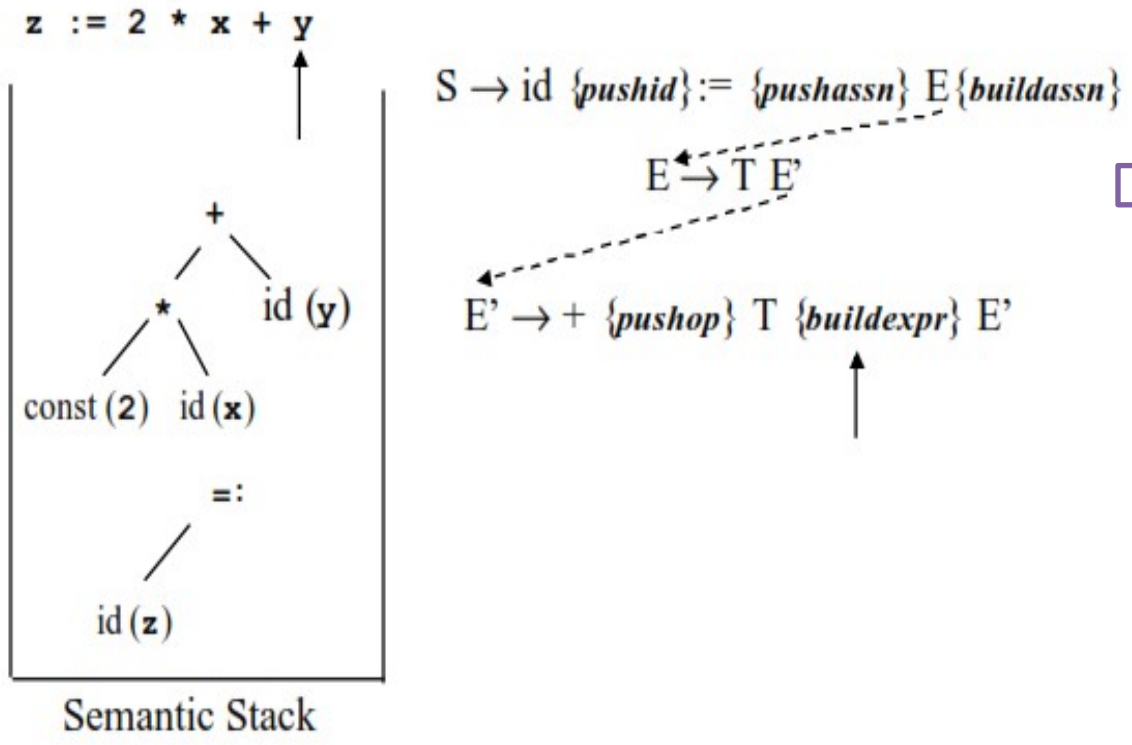
Building the AST (continued)



Building the AST Example

9

Building the AST (continued)



S → id {*pushid*} := {*pushassn*} E {*buildassn*}

E → T E'

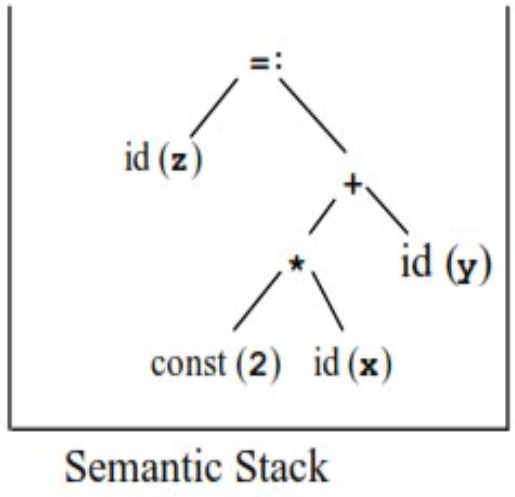
E' → + {*pushop*} T {*buildexpr*} E'

10

Building the AST (continued)

z := 2 * x + y

S → id {*pushid*} := {*pushassn*} E {*buildassn*}



Decorating the AST

- Abstract syntax trees have one enormous advantage over other intermediate representations:
  - they can be “decorated”, i.e., each node on the AST can have their attributes saved in the AST nodes, which can simplify the task of type checking as the parsing process continues.

4.0 Attribute Grammar

- An attribute grammar is an extension to a context- free grammar that is used to describe features of a programming language that cannot be described in BNF or can only be described in BNF with great difficulty.

Examples

- Describing the rule that real variables can be assigned integer values but the reverse is not true is difficult to describe completely in BNF.
- Sebesta says that the rule requiring that all variable must be declared before being used is impossible to describe in BNF.

4.1 What is an Attribute?

- An **Attribute** is a property whose value is assigned to a grammar symbol.
  - **Attribute Computation Functions** (or Semantic Functions) are associated with the productions of a grammar and are used to compute the values of an attribute.
  - **Predicate functions** state some of the syntax and static semantics rules of the grammar.

4.2 Types of Attribute?

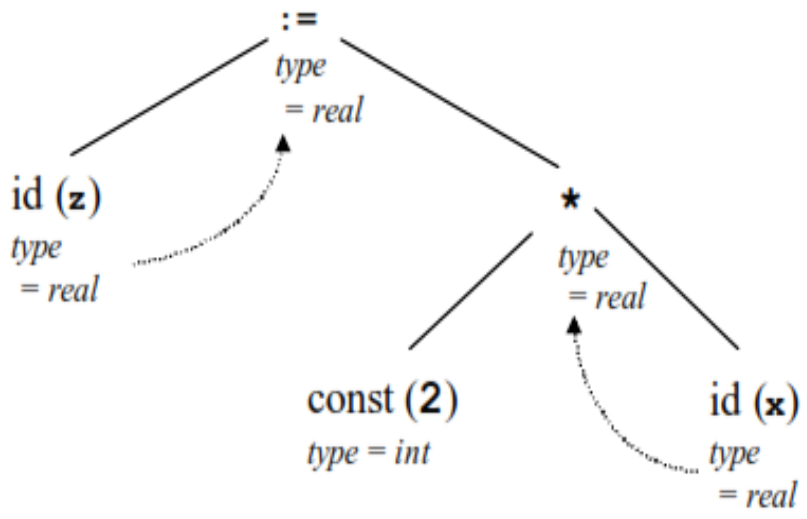
- The most common types of attributes to note for each symbol are:
  - **Types:** associates the data object with the allowable set of values.
  - **Location:** may be changed by the memory management routine of the operating system.
  - **Value:** usually the result of an assignment operation
  - **Name:** can be changed as a result of subprogram calls and returns
  - **Component:** data objects may be composed of several data objects. This binding may be represented by a pointer and subsequently changed.

4.3 Definition of an Attribute Grammar

- An attribute grammar is defined as a grammar with the following added features:
  - Each symbol X has a set of attributes A(X).
  - A(X) can be:
    - **Extrinsic Attributes -** which are obtained from outside the grammar, mostly notably the symbol table.
    - **Synthesized Attributes -** which are passed up the parse tree (from Child to Parent)
    - **Inherited Attributes -** which are passed down the parse tree (Parent to Child)
  - Each production of the grammar has a set of semantic functions and a set of predicate functions (which may be an empty set).

4.3 Definition of an Attribute Grammar (2)

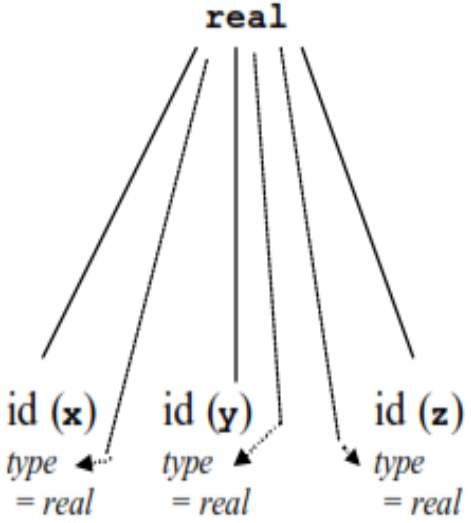
Synthesized Attributes



Inherited Attributes

Decl → Type {*pushtype*}IdList;
Type → **real**
Type → **int**
IdList → id {*addtotree*}IdList'
IdList' → , id {*addtotree*} IdList'
IdList' → ε

Example:
real x, y, z;

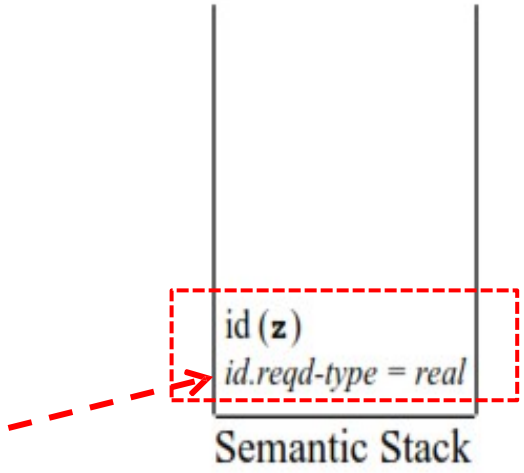


5.0 Decorating the AST

1

Decorating the AST

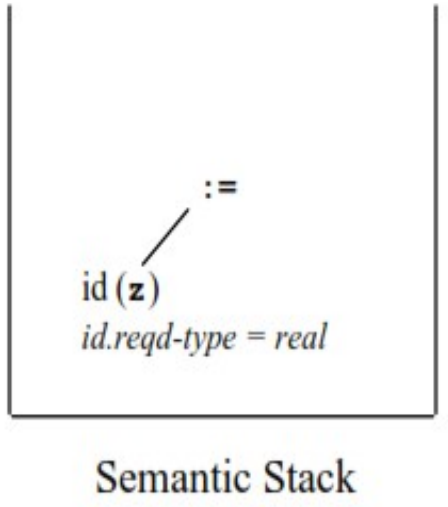
`z := 2 * x + y` S → id {*pushid*} := {*pushassn*} E {*buildassn*}



2

Decorating the AST (continued)

`z := 2 * x + y` S → id {*pushid*} := {*pushassn*} E {*buildassn*}

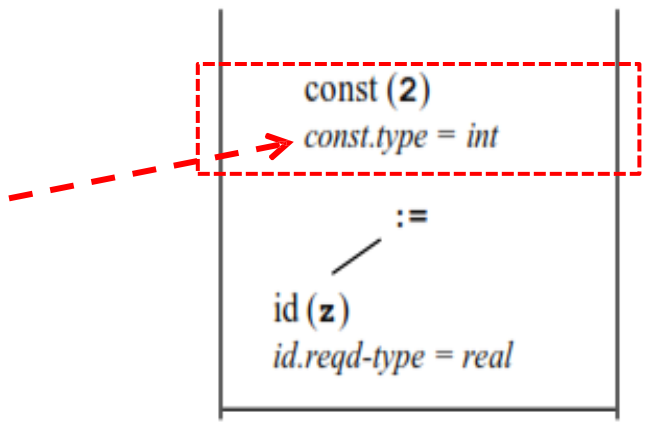


5.0 Decorating the AST

3

Decorating the AST (continued)

`z := 2 * x + y` F → const *{pushconst}*

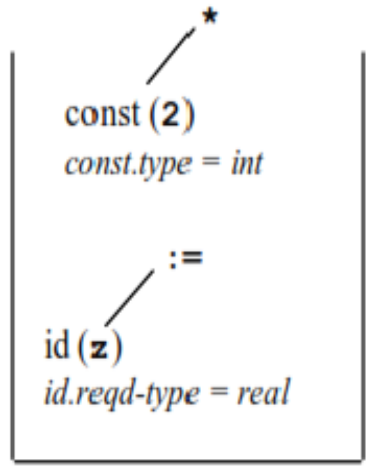


Semantic Stack

4

Decorating the AST (continued)

`z := 2 * x + y` T' → * *{pushop}* F *{buildterm}* T'



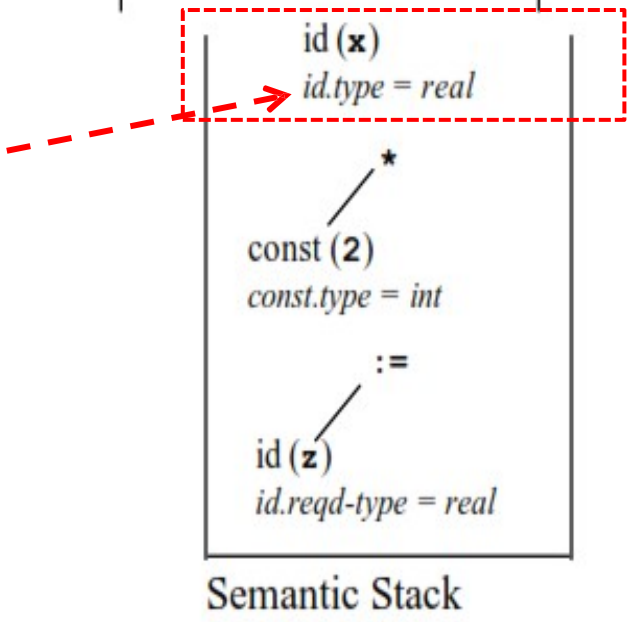
Semantic Stack

5.0 Decorating the AST

5

Decorating the AST (continued)

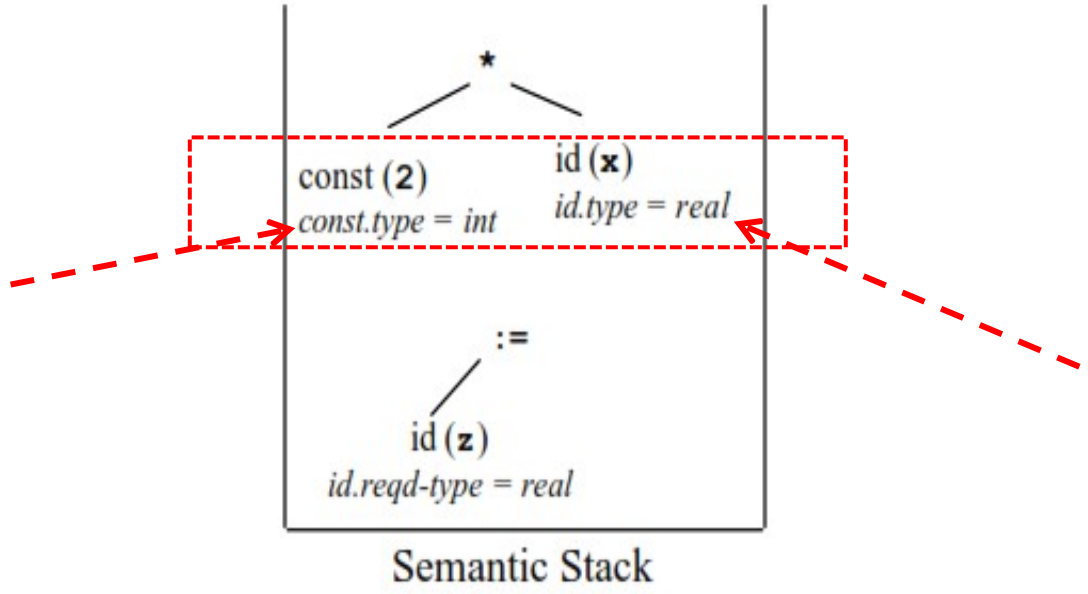
z := 2 * x + y F → id {*pushid*}



6

Decorating the AST (continued)

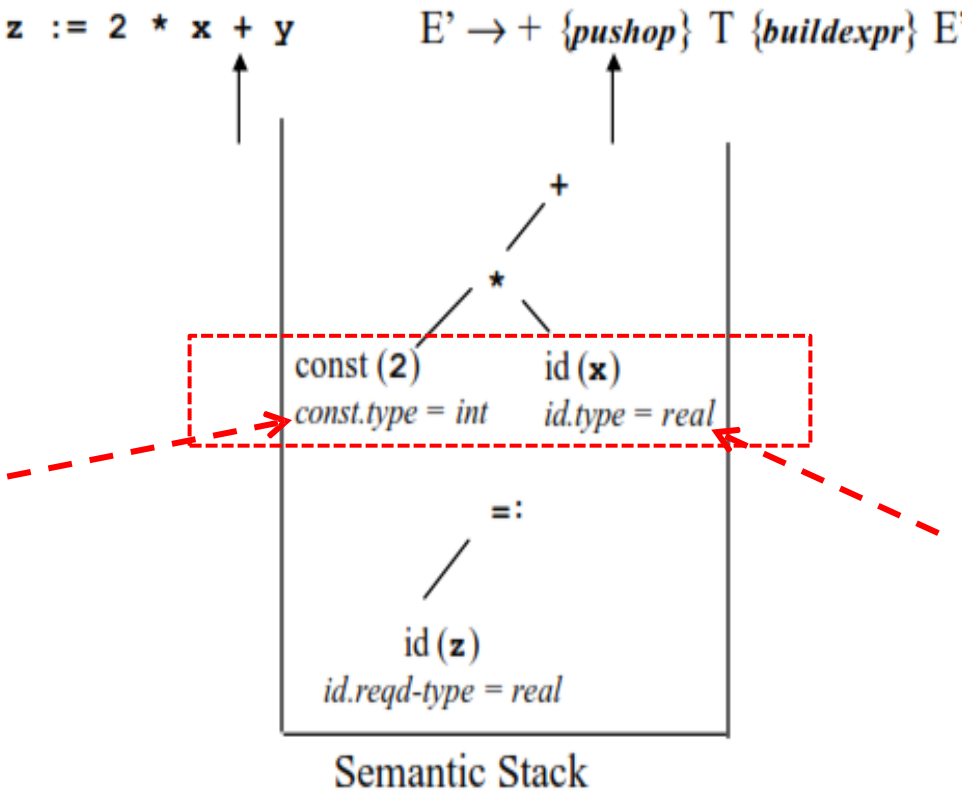
z := 2 * x + y T' → * {*pushop*} F {*buildterm*} T'



5.0 Decorating the AST

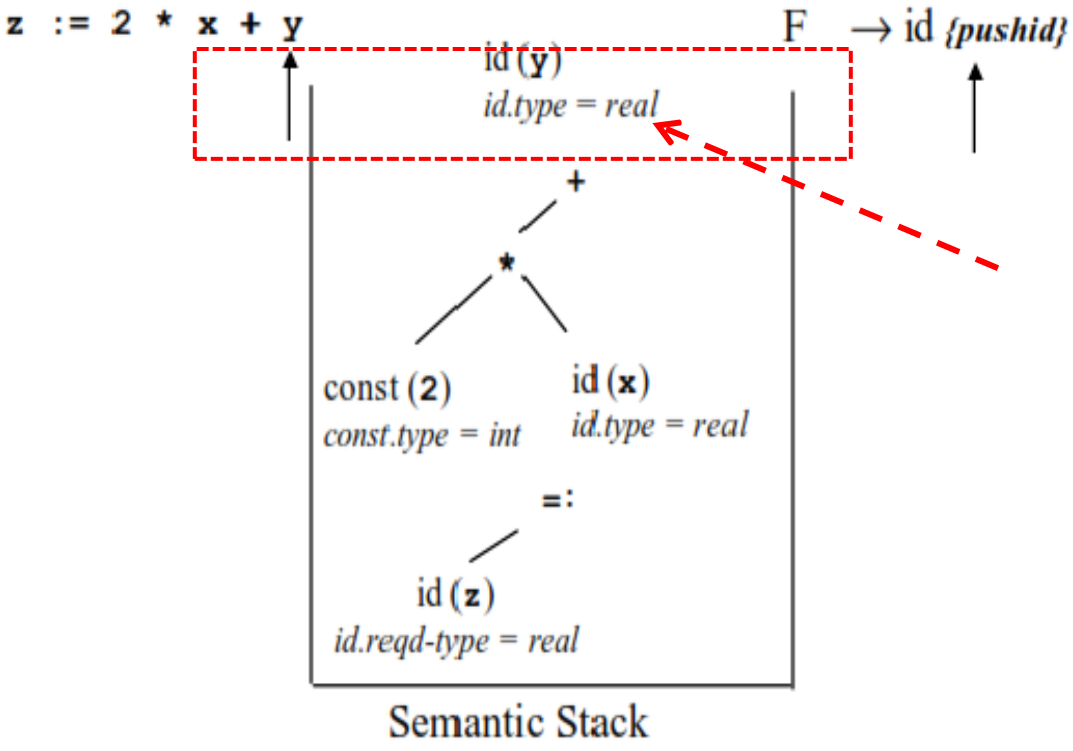
7

Decorating the AST (continued)



8

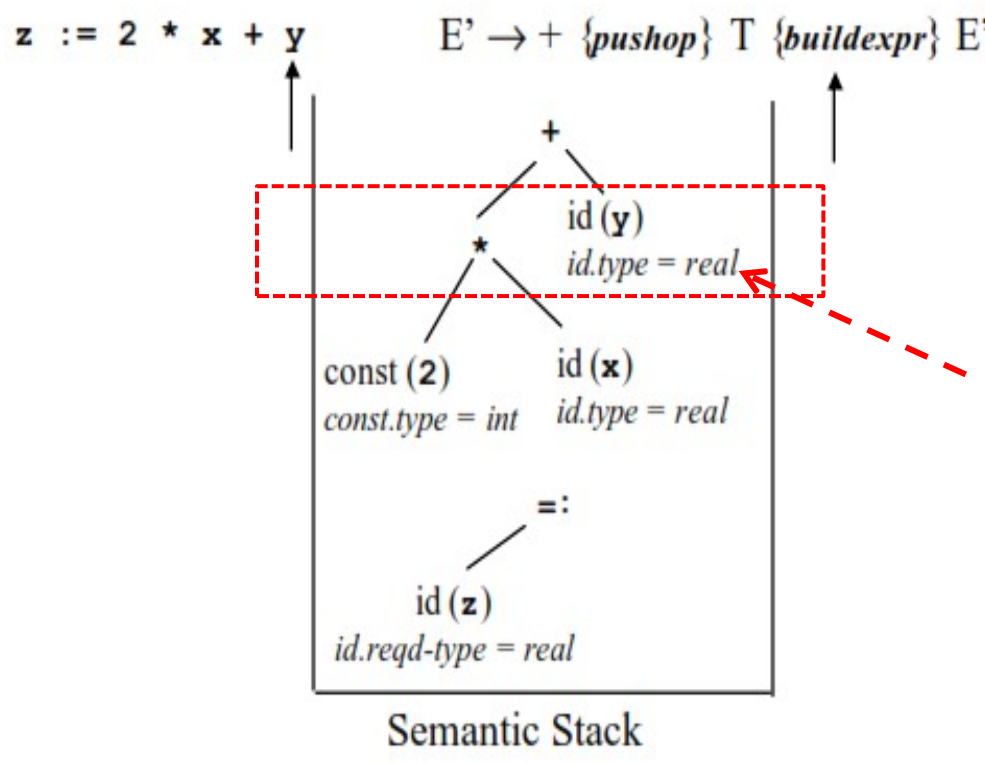
Decorating the AST (continued)



5.0 Decorating the AST

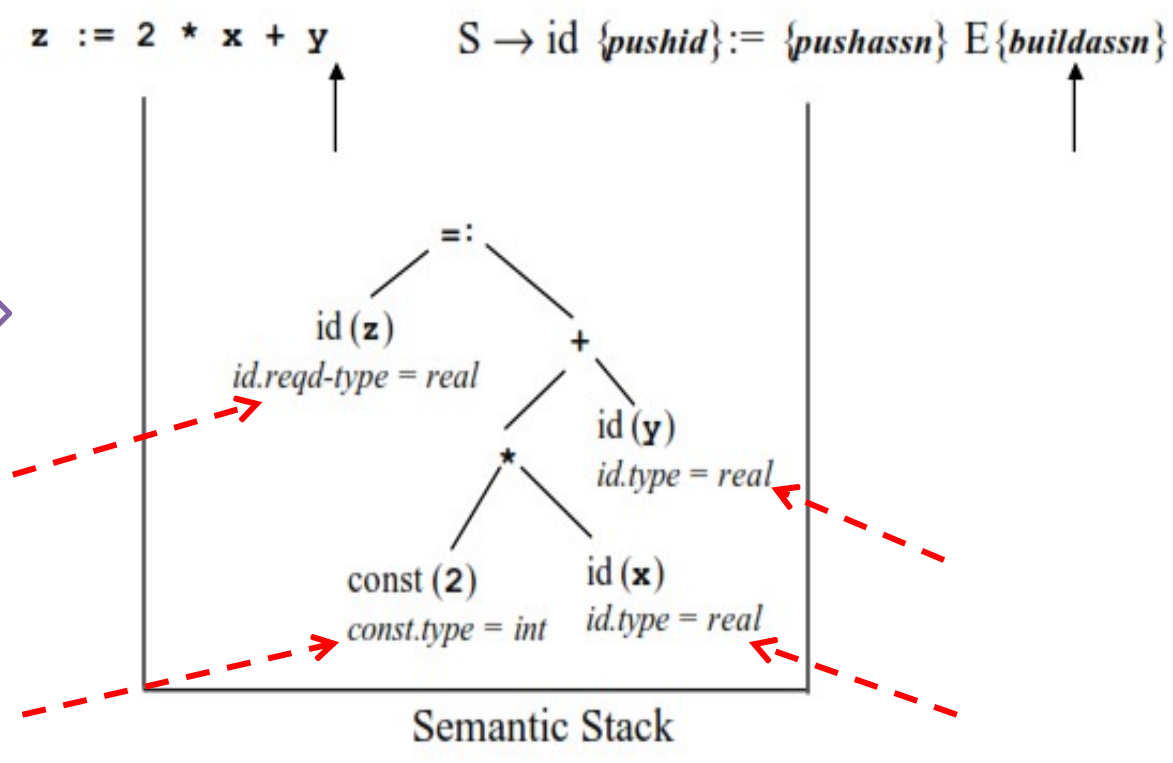
9

Decorating the AST (continued)



10

Decorating the AST (continued)



Implementing Semantics Actions In Recursive-Descent Parsing

- In a Recursive-descent Parser (i.e Top-down Parsing Method), there is a separate function for each nonterminal in the grammar.
  - The procedures check the look-ahead token against the terminals that it expects to find.
  - The procedures recursively call the procedures to parse nonterminals that it expects to find.
  - We now add the appropriate semantic actions that must be performed at certain points in the parsing process.

Processing Declarations

- Before any type checking can be performed,
  - type must be stored in the symbol table.
  - This is done while parsing the declarations.
- When processing the program's header statement:
  - the program's identifier must be assigned the type program.
  - the current scope pointer set to point to the main program.

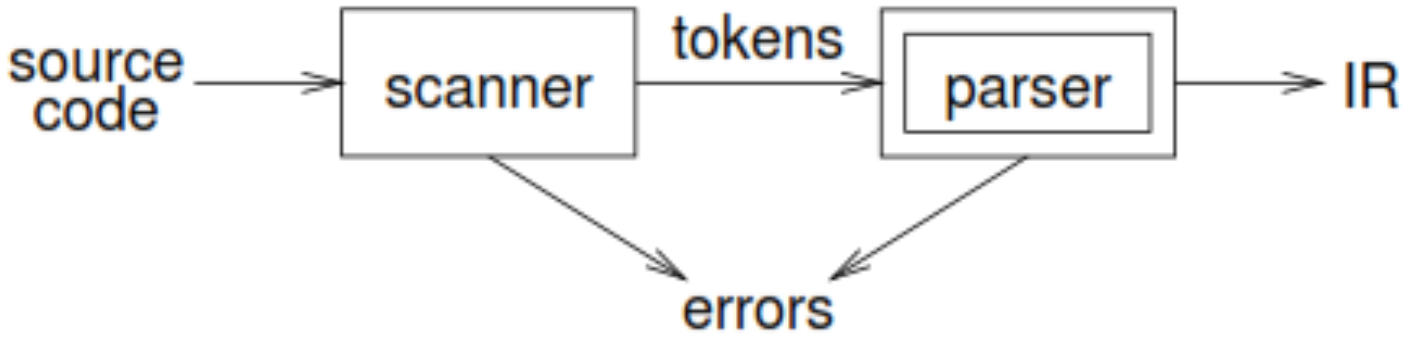
Processing Declarations (2)

- Processing declarations requires several actions:

1. If the language allows for user-defined data types, the installation of these data types must have already occurred.
2. The data types are installed in the symbol table entries for the declared identifiers.
3. The identifiers are added to the abstract syntax tree.

A Rejoinder

The role of the parser



Parser

- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction