Compiler Construction

Lecture 8 – 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:
  - **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.)
  - **Label Checking** – Labels references in a program must exist.
  - **Flow control checks** – control structures must be used in their proper fashion (no GOTOs into a FORTRAN DO statement, no breaks outside a loop or switch statement, etc.)

Where Is Semantic Analysis Performed in a Compiler?

- Semantic analysis is not a separate module within a compiler. It is usually a collection of procedures called at appropriate times by the parser as the grammar requires it.
- Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures.
- Implementing the semantic actions in a table-action driven LL(1) parser requires the addition of a third type of variable to the productions and the necessary software routines to process it.

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 will produce an intermediate representation during semantic analysis; most often it will be an abstract syntax tree or quadruples.

Semantic Actions in Top-Down Parsing: An Example

Imagine we're parsing:

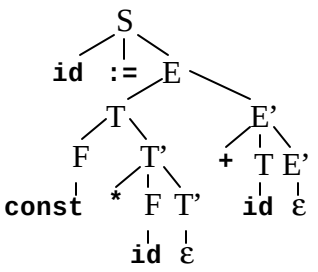
S → id := E
E → T E'
E' → + T E'
E' → ε
T → F T'
T' → * F T'
T' → ε
F → id
F → const
F → (E)

We insert the actions

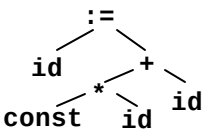
S → id {***pushid***}:= {***pushassn***} E{***buildassn***}
E → T E'
E' → + {***pushop***} T {***buildexpr***} E'
E' → ε
T → F T'
T' → * {***pushop***} F {***buildterm***} T'
T' → ε
F → id ***{pushid}***
F → const ***{pushconst}***
F → (E) ***{pushfactor}***

Example: Parsing An Expression

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



We want to create the AST fragment:



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

Or we can produce a set of quadruples:

```
t1 := 2 * x
t2 := t1 + y
z := t2
```

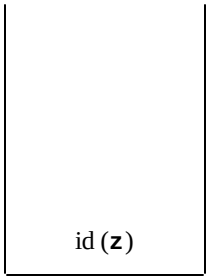
Or we an produce a set of assembly language instructions:

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

Building the AST

`z := 2 * x + y`

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

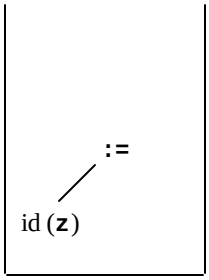


Semantic Stack

Building the AST (continued)

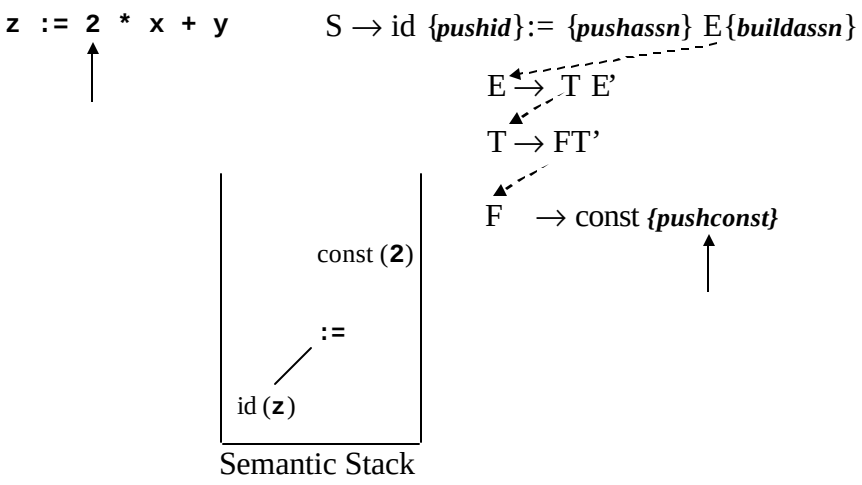
`z := 2 * x + y`

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

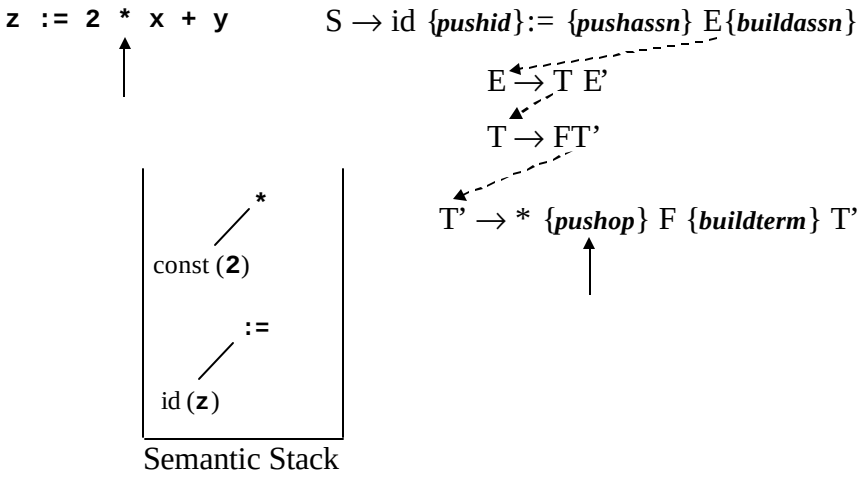


Semantic Stack

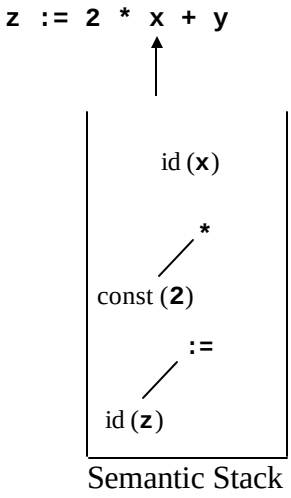
Building the AST (continued)



Building the AST (continued)



Building the AST (continued)



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

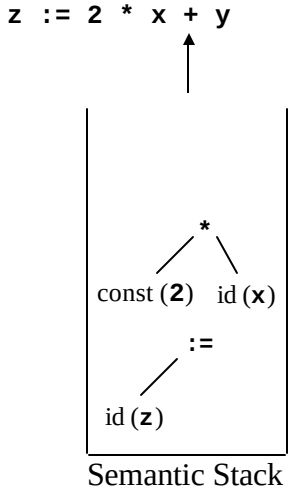
E → T E'

T → FT'

T' → * {*pushop*} F {*buildterm*} T'

F → id {*pushid*}

Building the AST (continued)



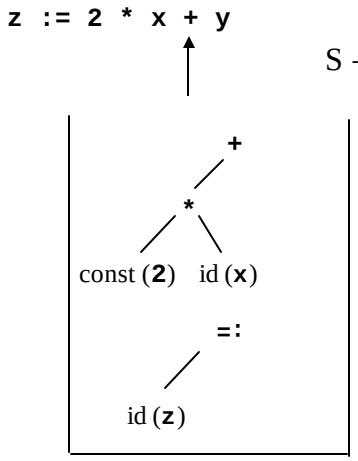
S → id {*pushid*}:= {*pushassn*} E{*buildassn*}

E → T E'

T → FT'

T' → * {*pushop*} F {*buildterm*} T'

Building the AST (continued)

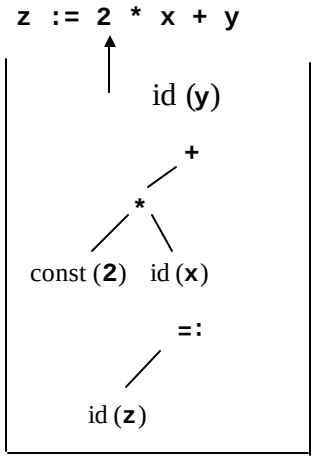


S → id {*pushid*}:= {*pushassn*} E{*buildassn*}

E → T E'

E' → + {*pushop*} T {*buildexpr*} E'

Building the AST (continued)



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

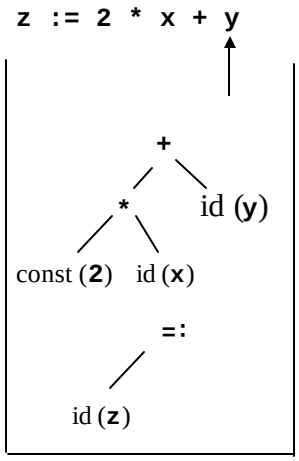
E → T E'

E' → + {*pushop*} T {*buildexpr*} E'

T → FT'

F → id *{pushid}*

Building the AST (continued)



Semantic Stack

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

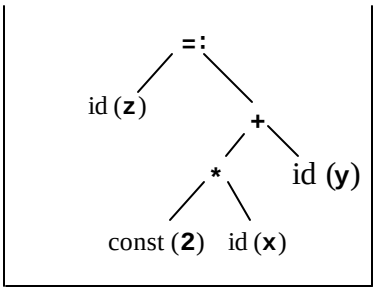
E → T E'

E' → + {*pushop*} T {*buildexpr*} E'

Building the AST (continued)

z := 2 * x + y

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



Semantic Stack

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.

What is an 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.

Static vs. Dynamic Semantics

- The static semantics of a language is indirectly related to the meaning of programs during execution. Its names comes from the fact that these specifications can be checked at compile time.
- Dynamic semantics refers to meaning of expressions, statements and other program units. Unlike static semantics, these cannot be checked at compile time and can only be checked at runtime.

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.

Types of Attributes

The most common types of attributes that we may wish to note for each symbol are:

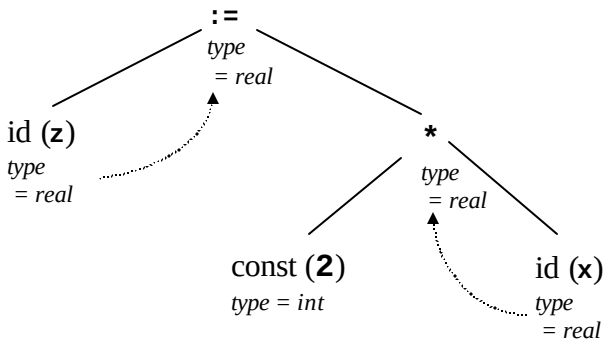
- **Type** – Associates the data object with the allowable set of values.
- **Location** – may be changed by the memory management routine of the operating system.
- **Value** – usually the result of an assignment operation.
- **Name** – can be changed as a result of subprogram calls and returns
- **Component** – data objects may be composed of several data objects. This binding may be represented by a pointer and subsequently changed.

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
  - inherited attributes which are passed down the parse tree
- Each production of the grammar has a set of semantic functions and a set of predicate functions (which may be an empty set).

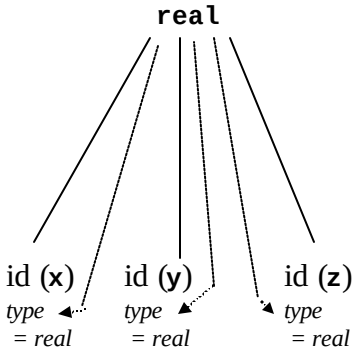
Synthesized Attributes



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;`



Attribute Rules

- Let's rewrite our grammar, assuming that variables are implicitly as in Fortran:

$S \rightarrow$ id := E
Rule: IF *id.reqd-type* = int
AND *E.type* = real
THEN TypeError(*S*, *E*)

$E \rightarrow$ T E'
Rule: IF *T.type* = *E'.type*
THEN *E.type* := *T.type*
ELSE *E.type* = real

$E_1' \rightarrow$ + T E_2'
Rule: IF $E_2'.type$ = *T.type*
THEN $E_1'.type$:= *T.type*
ELSE $E_1'.type$ = real

$E' \rightarrow \varepsilon$

Attribute Rules (continued)

$T \rightarrow$ F T'
Rule: IF *F.type* = *T'.type*
THEN *T.type* := *F.type*
ELSE *T.type* = real

$T_1' \rightarrow$ * F T_2'
Rule: IF *F.type* = $T_2'.type$
THEN $T_1'.type$:= *F.type*
ELSE $T_1'.type$ = real

$T' \rightarrow \varepsilon$

$F \rightarrow$ id
Rule: *F.type* := *id.type*

$F \rightarrow$ const
Rule: *F.type* := *const.type*

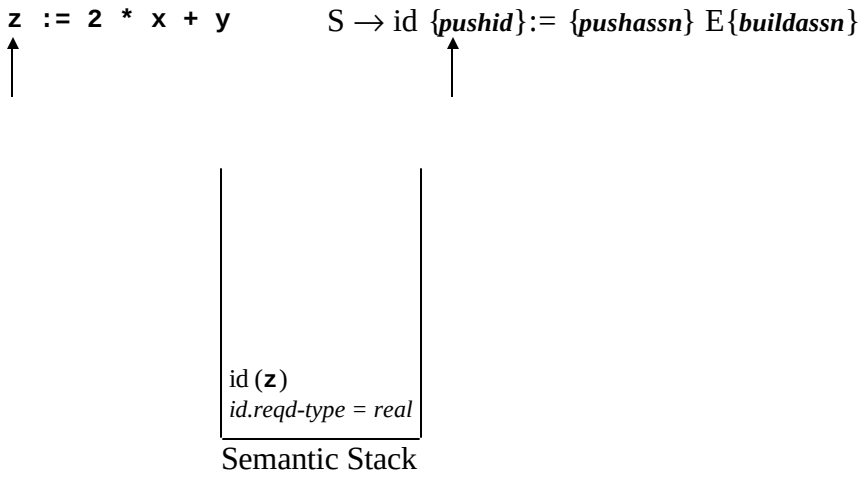
$F \rightarrow$ (E)
Rule: *F.type* := *E.type*

Implementing the Rules

In a semantic actions are to be ***handwritten***, we can incorporate them in the existing actions:

- The rule "IF *id.reqd-type* = *E.type* THEN id.type = *E.type* ELSE TypeError(*S*, *E*)" is incorporated in the procedure ***BuildAssn***
- The rule "IF *T.type* = *E'.type* THEN *E.type* := *T.type* ELSE *E.type* = real" is incorporated in a new procedure called ***SetExprType***, which is placed at the end of the production E → TE'
- We can place the other rules in action procedures placed at the end of their respective productions.

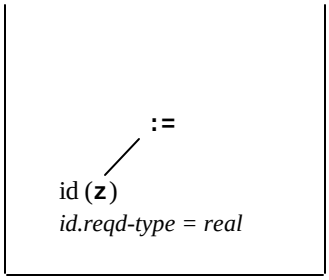
Decorating the AST



Decorating the AST (continued)

`z := 2 * x + y`

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

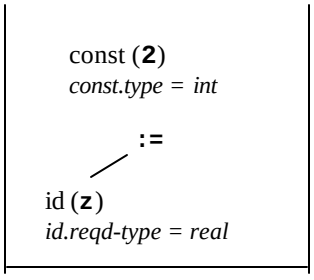


Semantic Stack

Decorating the AST (continued)

`z := 2 * x + y`

F → const ***{pushconst}***

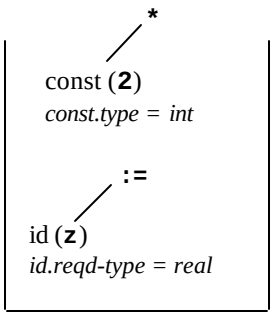


Semantic Stack

Decorating the AST (continued)

z := 2 * x + y

T' → * {***pushop***} F {***buildterm***} T'

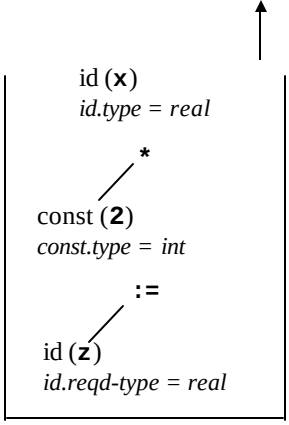


Semantic Stack

Decorating the AST (continued)

z := 2 * x + y

F → id ***{pushid}***

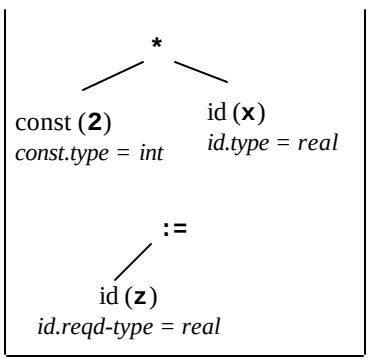


Semantic Stack

Decorating the AST (continued)

z := 2 * x + y

T' → * {***pushop***} F {***buildterm***} T'

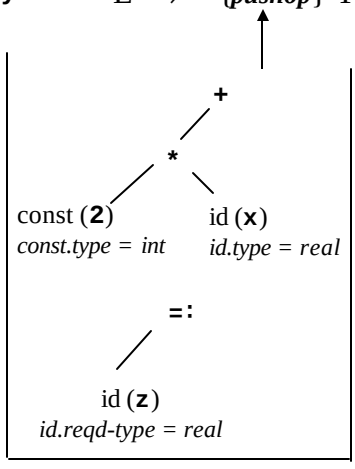


Semantic Stack

Decorating the AST (continued)

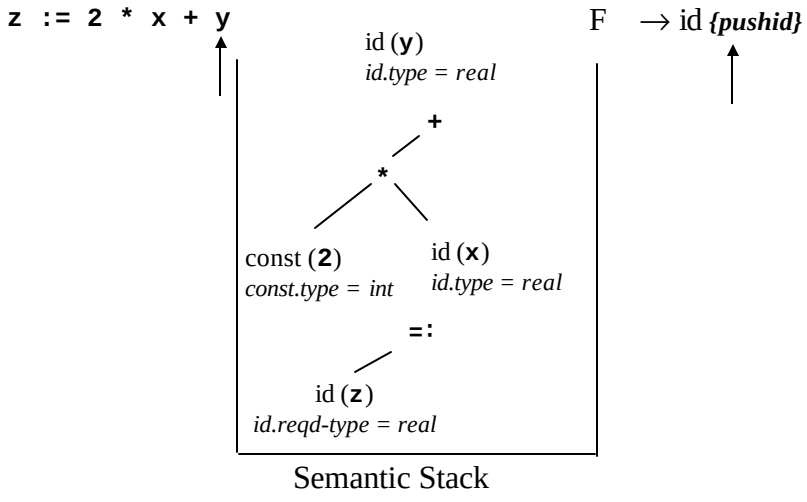
z := 2 * x + y

E' → + {***pushop***} T {***buildexpr***} E'

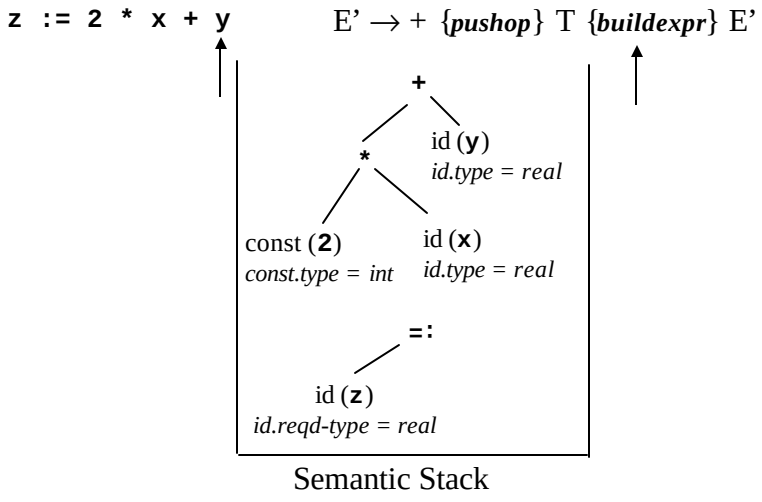


Semantic Stack

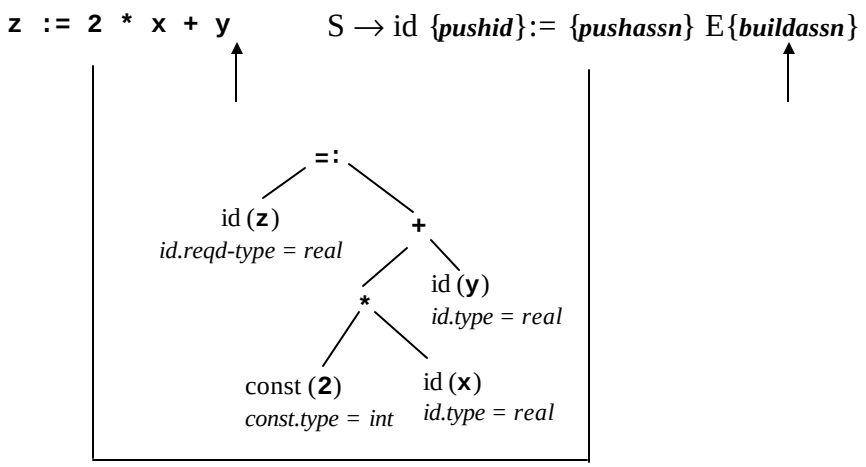
Decorating the AST (continued)



Decorating the AST (continued)



Decorating the AST (continued)



Implementing Semantics Actions In Recursive-Descent Parsing

- In a recursive-descent parser, there is a separate function for each nonterminal in the grammar.
  - The procedures check the lookahead 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

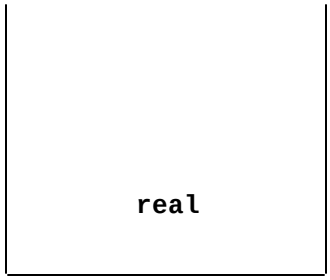
- Before any type checking can be performed, type must be stored in the symbol table. This is done while parsing the declarations.
- When processing the program's header statement:
  - the program's identifier must be assigned the type program
  - the current scope pointer set to point to the main program.

Processing Declarations

- Processing declarations requires several actions:
- If the language allows for user-defined data types, the installation of these data types must have al;ready occurred.
- The data types are installed in the symbol table entries for the declared identifiers.
- The identifiers are added to the abstract syntax tree.

Adding Declarations to the AST

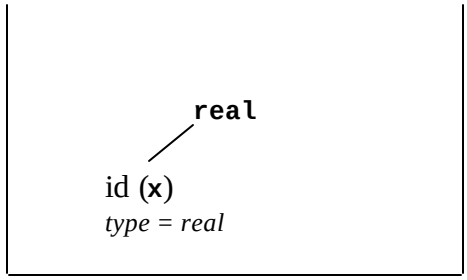
`real x, y, z;` Decl → Type {***pushtype***}IdList;



Semantic Stack

Adding Declarations to the AST (continued)

`real x, y, z;` IdList → id {***addtotree***}IdList'

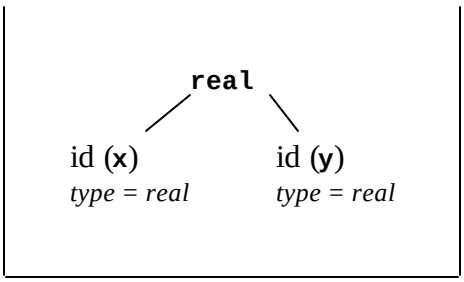


Semantic Stack

Adding Declarations to the AST (continued)

`real x, y, z;`

IdList' → **`,`** id {***addtotree***} IdList'

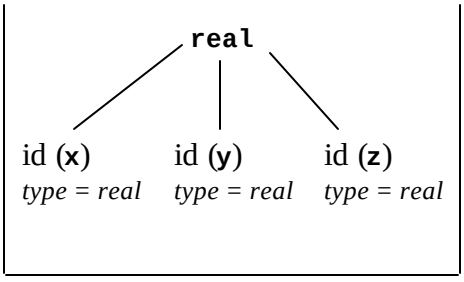


Semantic Stack

Adding Declarations to the AST (continued)

`real x, y, z;`

IdList' → **`,`** id {***addtotree***} IdList'



Semantic Stack