# Semantic Analysis

Chapter 4

# Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are
  - semantic analysis
  - (intermediate) code generation
- The principal job of the semantic analyzer is to enforce static semantic rules
  - constructs a syntax tree (usually first)
  - information gathered is needed by the code generator

# Role of Semantic Analysis

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved
- A common approach interleaves construction of a syntax tree with parsing (no explicit parse tree), and then follows with separate, sequential phases for semantic analysis and code generation

# Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse or syntax tree
- ATTRIBUTE GRAMMARS provide a formal framework for decorating such a tree
- Consider the following LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity:

# Attribute Grammars

```
E → E + T
E → E − T
E → T
T → T * F
T → T / F
T → F
F → - F
F → (E)
F → const
```

- This says nothing about what the program MEANS

# Attribute Grammars

- We can turn this into an attribute grammar as follows (similar to Figure 4.1):

```
E → E + T    E1.val = Sum(E2.val,T.val)
E → E − T    E1.val = Diff(E2.val,T.val)
E → T        E.val  = T.val
T → T * F    T1.val = Prod(T2.val,F.val)
T → T / F    T1.val = Div(T2.val,F.val)
T → F        T.val  = F.val
F → - F      F1.val = Prod(F2.val,-1)
F → (E)      F.val  = E.val
F → const    F.val  = C.val
```

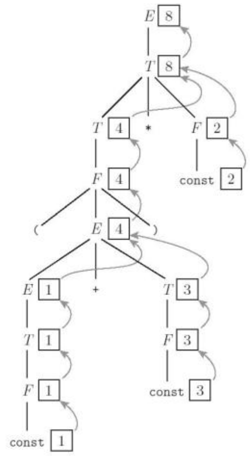
## Attribute Grammars

- The attribute grammar serves to define the semantics of the input program
- Attribute rules are best thought of as definitions, not assignments
- They are not necessarily meant to be evaluated at any particular time, or in any particular order, though they do define their left-hand side in terms of the right-hand side

## Evaluating Attributes

- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree
  - When a parse tree under this grammar is fully decorated, the value of the expression will be in the *val* attribute of the root
- The code fragments for the rules are called SEMANTIC FUNCTIONS
  - Strictly speaking, they should be cast as functions, e.g., E1.val = sum (E2.val, T.val) but often we will use the obvious E1.val = E2.val + T.val

# Evaluating Attributes




```
E → E + T        E1.val = E2.val + T.val
E → E − T        E1.val = E2.val - T.val
E → T            E.val  = T.val
T → T * F        T1.val = T2.val * F.val
T → T / F        T1.val = T2.val / F.val
T → F            T.val  = F.val
F → - F          F1.val = - F2.val
F → (E)          F.val  = E.val
F → const        F.val  = C.val
```

Figure 4.2: **Decoration of a parse tree for** `(1 + 3) * 2`**.** The val attributes of symbols are shown in boxes. Curving arrows represent the attribute flow, which is strictly upward in this case.

# Evaluating Attributes

- This is a very simple attribute grammar:
  - Each symbol has at most one attribute
    - the punctuation marks have no attributes
- These attributes are all so-called SYNTHESIZED attributes:
  - They are calculated only from the attributes of things below them in the parse tree

# Evaluating Attributes

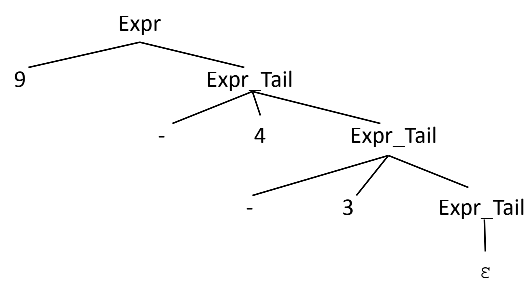
- In general, we are allowed both synthesized and INHERITED attributes:
  - Inherited attributes may depend on things above or to the side of them in the parse tree
  - Tokens have only synthesized attributes, initialized by the scanner (name of an identifier, value of a constant, etc.).
  - Inherited attributes of the start symbol constitute run-time parameters of the compiler

# Inherited Attributes

- LL(1) grammar covering subtraction:

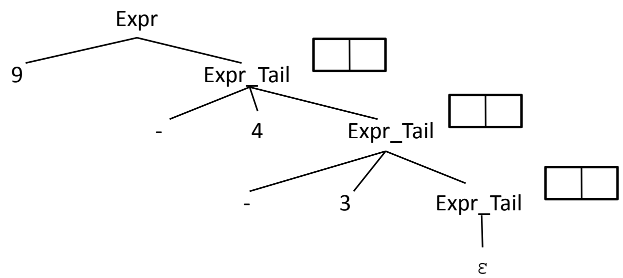
```
Expr        → const Expr_Tail
Expr_Tail   → - const Expr_Tail | ε
```

- For the expression 9 – 4 – 3:



# Inherited Attributes

- If we are allowed to pass attribute values not only bottom-up but also left-to-right then we can pass 9 into the Expr_Tail node for evaluation, and so on for each Expr_Tail




Similar to recursion when the result is accumulated as recursive calls made

# Evaluating Attributes

- The grammar for evaluating expressions is called S-ATTRIBUTED because it uses only synthesized attributes
- Its ATTRIBUTE FLOW (attribute dependence graph) is purely bottom-up
  - It is SLR(1), but not LL(1)
- An equivalent LL(1) grammar requires inherited attributes:

# Evaluating Attributes – Example

- Attribute grammar in Figure 4.3:

| | |
|---|---|
| E → T TT | E.v = TT.v |
| | TT.st = T.v |
| $TT_1$ → + T $TT_2$ | $TT_1$.v = $TT_2$.v |
| | $TT_2$.st = $TT_1$.st + T.v |
| $TT_1$ → - T $TT_2$ | $TT_1$.v = $TT_2$.v |
| | $TT_2$.st = $TT_1$.st - T.v |
| TT → ε | TT.v = TT.st |
| T → F FT | T.v = FT.v |
| | FT.st = F.v |

# Evaluating Attributes– Example

- Attribute grammar in Figure 4.3 (continued):

| | |
|---|---|
| $FT_1$ → * F $FT_2$ | $FT_1$.v = $FT_2$.v |
| | $FT_2$.st = $FT_1$.st * F.v |
| $FT_1$ → / F $FT_2$ | $FT_1$.v = $FT_2$.v |
| | $FT_2$.st = $FT_1$.st / F.v |
| FT → ε | FT.v = FT.st |
| $F_1$ → - $F_2$ | $F_1$.v = - $F_2$.v |
| F → ( E ) | F.v = E.v |
| F → const | F.v = C.v |

- Figure 4.4 – parse tree for (1+3)*2

# Evaluating Attributes– Example

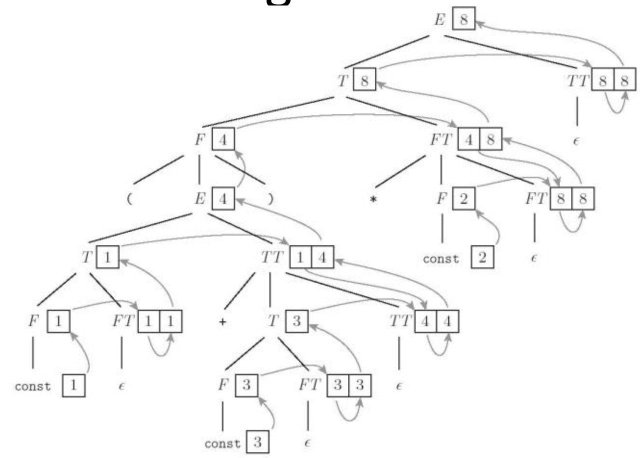



Figure 4.4: **Decoration of a top-down parse tree for** `(1 + 3) * 2`**, using the attribute grammar of Figure 4.3.** Curving arrows again represent attribute flow, which is no longer bottom-up, but is still left-to-right.

# Evaluating Attributes– Example

- Attribute grammar in Figure 4.3:
  - This attribute grammar is a good bit messier than the first one, but it is still L-ATTRIBUTED, which means that the attributes can be evaluated in a single left-to-right pass over the input
  - In fact, they can be evaluated during an LL parse
  - Each synthetic attribute of a LHS symbol (by definition of *synthetic*) depends only on attributes of its RHS symbols

# Evaluating Attributes – Example

- Attribute grammar in Figure 4.3:
  - Each inherited attribute of a RHS symbol (by definition of *L-attributed*) depends only on
    - inherited attributes of the LHS symbol, or
    - synthetic or inherited attributes of symbols to its left in the RHS
  - L-attributed grammars are the most general class of attribute grammars that can be evaluated during an LL parse

# Evaluating Attributes

- There are certain tasks, such as generation of code for short-circuit Boolean expression evaluation, that are easiest to express with non-L-attributed attribute grammars
- Because of the potential cost of complex traversal schemes, however, most real-world compilers insist that the grammar be L-attributed

## Evaluating Attributes - Abstract Syntax

- The Abstract Syntax defines essential syntactic elements without describing how they are concretely constructed
- Consider the following Pascal and C loops

| **Pascal** | **C** |
|---|---|
| `while i<n do begin` | `while (i<n) {` |
| `  i:=i+1` | `      i=i+1;` |
| `end` | `}` |

Small differences in concrete syntax; identical abstract construct

## Abstract Syntax Format

- We can define an abstract syntax using rules of the form
  - LHS = RHS
    - LHS is the name of an abstract syntactic class
    - RHS is a list of essential components that define the class
      - Similar to defining a variable. Data type or abstract syntactic class, and name
- Recursion naturally occurs among the definitions as with BNF
  - Makes it fairly easy to construct programmatically, similar to what we did for the concrete syntax

# Abstract Syntax Example

- Loop

```
Loop = Expression test ; Statement body
```

  – The abstract class Loop has two components, a *test* which is a member of the abstract class Expression, and a *body* which is a member of an abstract class Statement

- Nice by-product: If parsing abstract syntax in a language like Java, it makes sense to actually define a class for each abstract syntactic class, e.g.

```
class Loop extends Statement {
 Expression test;
 Statement body;
}
```

# Abstract Syntax of a C-like Language

```
Program = Declarations decpart;  Statements body;
Declarations = Declaration*
Declaration = VariableDecl   |   ArrayDecl
VariableDecl = Variable v;  Type t
ArrayDecl = Variable v;  Type t;  Integer size
Type = int | bool | float | char
Statements = Statement*
Statement = Skip | Block | Assignment |
            Conditional | Loop
Skip =
Block = Statements
Conditional = Expression test;
             Statement thenbranch, elsebranch
Loop = Expression test;  Statement body
Assignment = VariableRef target;   Expression source
Expression = VariableRef | Value | Binary | Unary
```

# Abstract Syntax of a C-like Language

```
VariableRef = Variable | ArrayRef
Binary = Operator op;  Expression term1, term2
Unary = UnaryOp op;  Expression term
Operator = BooleanOp | RelationalOp | ArithmeticOp
BooleanOp = && | ||
RelationalOp = = | ! | != | < | <= | > | >=
ArithmeticOp = + | - | * | /
UnaryOp = ! | -
Variable = String id
ArrayRef = String id; Expression index
Value = IntValue | BoolValue | FloatValue | CharValue
IntValue = Integer intValue
FloatValue = Float floatValue
BoolValue = Boolean boolValue
CharValue = Character charValue
```

# Java Abstract Syntax for C-Like Language

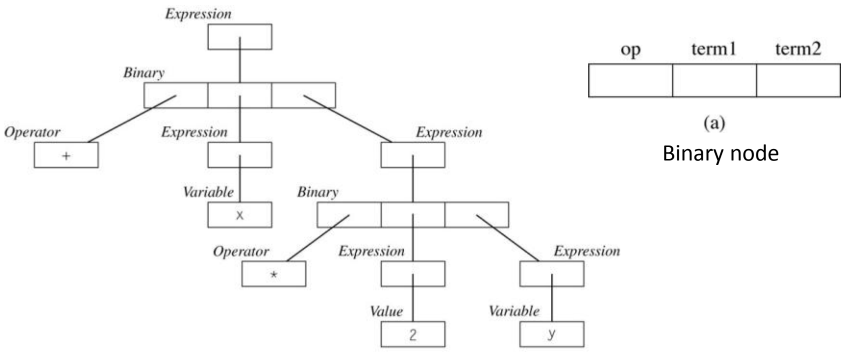
```
class Loop extends Statement {
  Expression test;
  Statement body;
}
class Assignment extends Statement {
  // Assignment = Variable target; Expression source
  Variable target;
  Expression source;
}

…
```

# Abstract Syntax Tree

- Just as we can build a parse tree from a BNF grammar, we can build an abstract syntax tree from an abstract syntax
- Example for: x+2*y

```
Expression = Variable | Value | Binary
Binary = Operator op ; Expression term1, term2
```



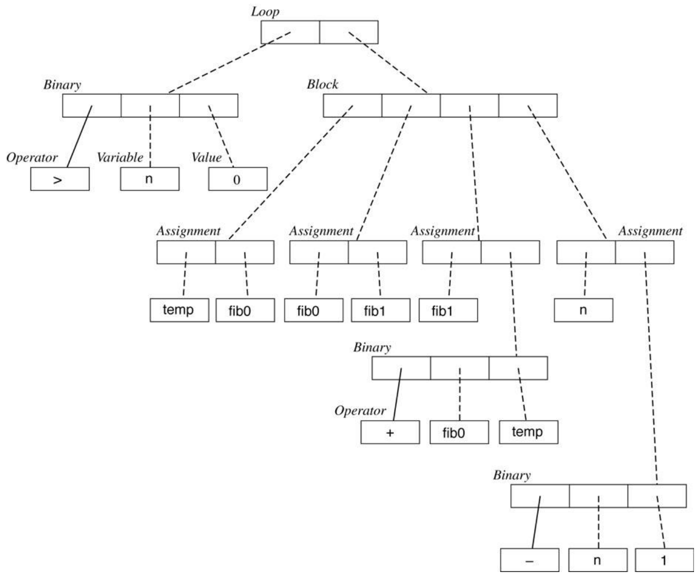

Expr (b)

# Sample C-Like Program

- Compute nth fib number

```
// compute result = the nth Fibonacci number
void main () {
  int n, fib0, fib1, temp, result;
  n = 8;
  fib0 = 0;
  fib1 = 1;
  while (n > 0) {
    temp = fib0;
    fib0 = fib1;
    fib1 = fib0 + temp;
    n = n - 1;
  }
   result = fib0;
}
```

# Abstract Syntax for Loop of C-Like Program




# Concrete and Abstract Syntax

- Aren't the two redundant?
  - A little bit
- The concrete syntax tells the programmer exactly what to write to have a valid program
- The abstract syntax allows valid programs in two different languages to share common abstract representations
  - It is closer to semantics
  - We need both!
- To construct the abstract syntax tree a common approach is a **bottom-up attribute grammar** associated with the concrete syntax

# Evaluating Attributes – Syntax Trees

$E_1 \longrightarrow E_2 + T$
  ▷ $E_1$.ptr := make_bin_op("+", $E_2$.ptr, T.ptr)

$E_1 \longrightarrow E_2 - T$
  ▷ $E_1$.ptr := make_bin_op("−", $E_2$.ptr, T.ptr)

$E \longrightarrow T$
  ▷ E.ptr := T.ptr

$T_1 \longrightarrow T_2 * F$
  ▷ $T_1$.ptr := make_bin_op("×", $T_2$.ptr, F.ptr)

$T_1 \longrightarrow T_2 / F$
  ▷ $T_1$.ptr := make_bin_op("÷", $T_2$.ptr, F.ptr)

$T \longrightarrow F$
  ▷ T.ptr := F.ptr

$F_1 \longrightarrow - F_2$
  ▷ $F_1$.ptr := make_un_op("+/_", $F_2$.ptr)

$F \longrightarrow ( E )$
  ▷ F.ptr := E.ptr

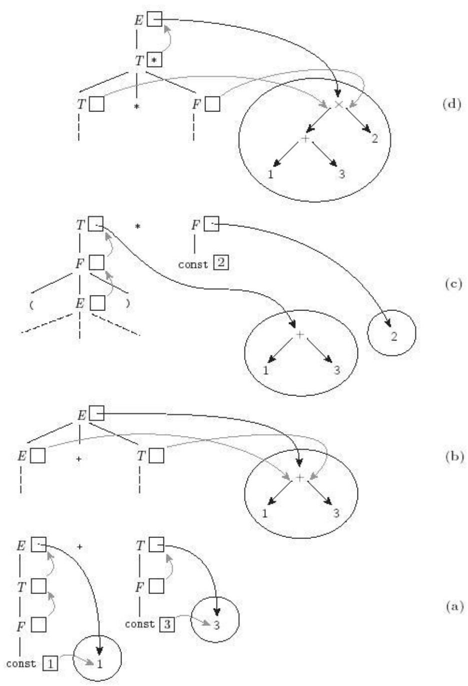
$F \longrightarrow$ const
  ▷ F.ptr := make_leaf(const.val)

Figure 4.5: **Bottom-up attribute grammar to construct a syntax tree.** The symbol +/_ is used (as it is on calculators) to indicate change of sign.

Skipping Top-Down, but it exists too (with inherited attributes)

# Evaluating Attributes – Syntax Trees

(1+3)*2




$E_1 \longrightarrow E_2 + T$
  ▷ $E_1$.ptr := make_bin_op("+", $E_2$.ptr, T.ptr)

$E \longrightarrow T$
  ▷ E.ptr := T.ptr

$T_1 \longrightarrow T_2 * F$
  ▷ $T_1$.ptr := make_bin_op("×", $T_2$.ptr, F.ptr)

$T \longrightarrow F$
  ▷ T.ptr := F.ptr

$F_1 \longrightarrow - F_2$
  ▷ $F_1$.ptr := make_un_op("+/_", $F_2$.ptr)

$F \longrightarrow ( E )$
  ▷ F.ptr := E.ptr

$F \longrightarrow$ const
  ▷ F.ptr := make_leaf(const.val)

Figure 4.7: Construction of a syntax tree via decoration of a bottom-up parse

# Action Routines

- We can tie this discussion back into the earlier issue of separated phases v. on-the-fly semantic analysis and/or code generation
- If semantic analysis and/or code generation are interleaved with parsing, then the TRANSLATION SCHEME we use to evaluate attributes MUST be L-attributed

# Action Routines

- If we break semantic analysis and code generation out into separate phase(s), then the code that builds the parse/syntax tree can still use a left-to-right (L-attributed) translation scheme
- However, the later phases are free to use a fancier translation scheme if they want

# Action Routines

- There are automatic tools that generate translation schemes for context-free grammars or tree grammars (which describe the possible structure of a syntax tree)
  - These tools are heavily used in syntax-based editors and incremental compilers
  - Most ordinary compilers, however, use ad-hoc techniques

# Action Routines

- An ad-hoc translation scheme that is interleaved with parsing takes the
  form of a set of ACTION ROUTINES:
  - An action routine is a semantic function that we tell the compiler to execute at a particular point in the parse
  - Same idea as the previous abstract syntax example (Fig 4.6, 4.7), except the action routines are embedded among the symbols of the right-hand sides; work performed is the same
- For our LL(1) attribute grammar, we could put in explicit action routines as follows:

# Action Routines - Example

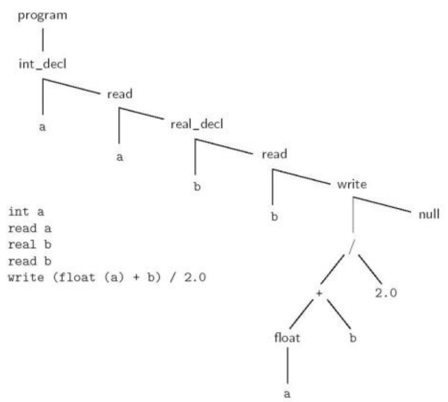
- Action routines (Figure 4.9)

$E \longrightarrow T$ { TT.st := T.ptr } $TT$ { E.ptr := TT.ptr }
$TT_1 \longrightarrow$ + $T$ { $TT_2$.st := make_bin_op ("+", $TT_1$.st, T.ptr) } $TT_2$ { $TT_1$.ptr := $TT_2$.ptr
$TT_1 \longrightarrow$ - $T$ { $TT_2$.st := make_bin_op ("−", $TT_1$.st, T.ptr) } $TT_2$ { $TT_1$.ptr := $TT_2$.ptr
$TT \longrightarrow \epsilon$ { TT.ptr := TT.st }
$T \longrightarrow F$ { FT.st := F.ptr } $FT$ { T.ptr := FT.ptr }
$FT_1 \longrightarrow$ * $F$ { $FT_2$.st := make_bin_op ("×", $FT_1$.st, F.ptr) } $FT_2$ { $FT_1$.ptr := $FT_2$.ptr }
$FT_1 \longrightarrow$ / $F$ { $FT_2$.st := make_bin_op ("÷", $FT_1$.st, F.ptr) } $FT_2$ { $FT_1$.ptr := $FT_2$.ptr }
$FT \longrightarrow \epsilon$ { FT.ptr := FT.st }
$F_1 \longrightarrow$ - $F_2$ { $F_1$.ptr := make_un_op ("+/_", $F_2$.ptr) }
$F \longrightarrow$ ( $E$ ) { F.ptr := E.ptr }
$F \longrightarrow$ const { F.ptr := make_leaf (const.ptr) }

Figure 4.9: **LL(1) grammar with action routines to build a syntax tree.**

# Decorating a Syntax Tree

- Abstract syntax tree for a simple program to print an average of an integer and a real




*program* $\longrightarrow$ *item*
*int_decl : item* $\longrightarrow$ *id item*
*read : item* $\longrightarrow$ *id item*
*real_decl : item* $\longrightarrow$ *id item*
*write : item* $\longrightarrow$ *expr item*
*null : item* $\longrightarrow$ $\epsilon$
*'/' : expr* $\longrightarrow$ *expr expr*
*'+' : expr* $\longrightarrow$ *expr expr*
*float : expr* $\longrightarrow$ *expr*
*id : expr* $\longrightarrow$ $\epsilon$
*real_const : expr* $\longrightarrow$ $\epsilon$

Figure 4.11: **Syntax tree for a simple calculator program.**

*program* ⟶ *item*
▷ item.symtab := nil
▷ program.errors := item.errors_out
▷ item.errors_in := nil

*int_decl* : $item_1$ ⟶ *id* $item_2$
▷ declare_name(id, $item_1$, $item_2$, int)
▷ $item_1$.errors_out := $item_2$.errors_out

*real_decl* : $item_1$ ⟶ *id* $item_2$
▷ declare_name(id, $item_1$, $item_2$, real)
▷ $item_1$.errors_out := $item_2$.errors_out

*read* : $item_1$ ⟶ *id* $item_2$
▷ $item_2$.symtab := $item_1$.symtab
▷ if ⟨id.name, ?⟩ ∈ $item_1$.symtab
    $item_2$.errors_in := $item_1$.errors_in
  else
    $item_2$.errors_in := $item_1$.errors_in + [id.name "undefined at" id.location]
▷ $item_1$.errors_out := $item_2$.errors_out

*write* : $item_1$ ⟶ *expr* $item_2$
▷ expr.symtab := $item_1$.symtab
▷ $item_2$.symtab := $item_1$.symtab
▷ $item_2$.errors_in := $item_1$.errors_in + expr.errors
▷ $item_1$.errors_out := $item_2$.errors_out

':=' : $item_1$ ⟶ *id* *expr* $item_2$
▷ expr.symtab := $item_1$.symtab
▷ $item_2$.symtab := $item_1$.symtab
▷ if ⟨id.name, A⟩ ∈ $item_1$.symtab    -- for some type A
    if A ≠ error and expr.type ≠ error and A ≠ expr.type
      $item_2$.errors_in := $item_1$.errors_in + ["type clash at" $item_1$.location]
    else
      $item_2$.errors_in := $item_1$.errors_in
  else
    $item_2$.errors_in := $item_1$.errors_in + [id.name "undefined at" id.location]
▷ $item_1$.errors_out := $item_2$.errors_out

*null* : *item* ⟶ ϵ
▷ item.errors_out := item.errors_in

Complete Attribute Grammar

*id* : *expr* ⟶ ϵ
▷ if ⟨id.name, A⟩ ∈ expr.symtab    -- for some type A
    expr.errors := nil
    expr.type := A
  else
    expr.errors := [id.name "undefined at" id.location]
    expr.type := error

*int_const* : *expr* ⟶ ϵ
▷ expr.type := int

*real_const* : *expr* ⟶ ϵ
▷ expr.type := real

'+' : $expr_1$ ⟶ $expr_2$ $expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

'−' : $expr_1$ ⟶ $expr_2$ $expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

'×' : $expr_1$ ⟶ $expr_2$ $expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

'÷' : $expr_1$ ⟶ $expr_2$ $expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

*float* : $expr_1$ ⟶ $expr_2$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ convert_type($expr_2$, $expr_1$, int, real, "float of non-int")

*trunc* : $expr_1$ ⟶ $expr_2$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ convert_type($expr_2$, $expr_1$, real, int, "trunc of non-real")

```
macro declare_name(id, cur_item, next_item : syntax_tree_node; t : type)
    if ⟨id.name, ?⟩ ∈ cur_item.symtab
        next_item.errors_in := cur_item.errors_in + ["redefinition of" id.name "at" cur_item.location]
        next_item.symtab := cur_item.symtab − ⟨id.name, ?⟩ + ⟨id.name, error⟩
    else
        next_item.errors_in := cur_item.errors_in
        next_item.symtab := cur_item.symtab + ⟨id.name, t⟩

macro check_types(result, operand1, operand2)
    if operand1.type = error or operand2.type = error
        result.type := error
        result.errors := operand1.errors + operand2.errors
    else if operand1.type ≠ operand2.type
        result.type := error
        result.errors := operand1.errors + operand2.errors + ["type clash at" result.location]
    else
        result.type := operand1.type
        result.errors := operand1.errors + operand2.errors

macro convert_type(old_expr, new_expr : syntax_tree_node; from_t, to_t : type; msg : string)
    if old_expr.type = from_t or old_expr.type = error
        new_expr.errors := old_expr.errors
        new_expr.type := to_t
    else
        new_expr.errors := old_expr.errors + [msg "at" old_expr.location]
        new_expr.type := error
```