Semantic Analysis Example: Type Checking

- Most programming language specifications include a **type hierarchy**
  - which compares the language's types in terms of their generality
- Example:
  A float type is considered **wider** (i.e., more general) than an integer (Java, C, and C++)
  - Every integer can be represented as a float
  - On the other hand, **narrowing** a float to an integer loses precision for some float values

Type Checking

- Most languages allow **automatic widening of type**
  - →E.g., an integer can be converted to a float without the programmer having to specify this conversion explicitly

```
int a;
float b, c;
c = a+b; //(automatically type casting for a)
```

- On the other hand, a float cannot become an integer in most languages
  - – unless the programmer explicitly calls for this conversion

Type Checking for *ac*

- Two types defined in ***ac***
  - I.e., **integer** and **float**, and
  - all identifiers must be type-declared in a program before they can be used
- Once the symbol table has been constructed,
  - the declared type of each identifier is known, and
  - the executable statements of the program can be **checked for type consistency**

  → **Type checking**

  Refers to the process that walks the AST **bottom-up**, from its leaves toward its root

Type Analysis for *ac*

- At each AST node, VISIT() is called:
  1. For **constants and symbol** references, the visitor methods simple set the supplied node's type based on the node's contents
  2. For **nodes that compute value**, such as plus and minus, the appropriate type is computed by calling the utility methods
  3. For an **assignment operation**, the visitor makes certain that the value computed by the second child is of the same type as the assigned identifier (the first child)

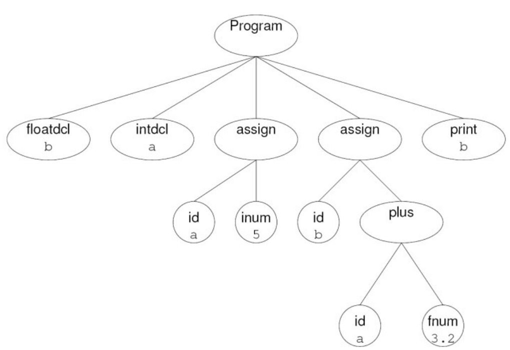

Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

```
/*  Visitor methods                                              */
procedure VISIT( Computing n )
    n.type ← CONSISTENT( n.child1, n.child2 )
end
procedure VISIT( Assigning n )
    n.type ← CONVERT( n.child2, n.child1.type )
end
procedure VISIT( SymReferencing n )
    n.type ← LOOKUPSYMBOL( n.id )
end
procedure VISIT( IntConsting n )
    n.type ← integer
end
procedure VISIT( FloatConsting n )
    n.type ← float
end
/*  Type-checking utilities                                      */
function CONSISTENT( c1, c2 ) returns type
    m ← GENERALIZE( c1.type, c2.type )
    call CONVERT( c1, m )
    call CONVERT( c2, m )
    return (m)
end
function GENERALIZE( t1, t2 ) returns type
    if t1 = float or t2 = float
    then ans ← float
    else  ans ← integer
    return (ans)
end
procedure CONVERT( n, t )
    if n.type = float and t = integer
    then call ERROR( "Illegal type conversion" )
    else
        if n.type = integer and t = float
        then
            /*  replace node n by convert-to-float of node n         */ (13)
        else  /* nothing needed */
end
```

Figure 2.12: Type analysis for ac.

Type Analysis for *ac*

- **CONSISTENT()** is responsible for reconciling the type of a pair of AST nodes with the following steps:
  1. The **GENERALIZE()** function determines the least general (i.e., simplest) type that encompasses its supplied pair of types
     For ac, if either type is float, then float is the appropriate type; otherwise, integer will do
  2. The **CONVERT()** procedure checks whether conversion is necessary, possible, or impossible
- An important consequence occurs at **Marker 13** in Figure 2.12
  - If conversion is attempted from integer to float, then the **AST is transformed to represent this type conversion explicitly**
  - Subsequent compiler passes (particularly code generation) can then assume a type-consistent AST in which all operations are explicit

```
/*   Type-checking utilities                                        */
function CONSISTENT(c1, c2) returns type
    m ← GENERALIZE(c1.type, c2.type)
    call CONVERT(c1, m)
    call CONVERT(c2, m)
    return (m)
end
function GENERALIZE(t1, t2) returns type
    if t1 = float or t2 = float
    then  ans ← float
    else  ans ← integer
    return (ans)
end
procedure CONVERT(n, t)
    if n.type = float and t = integer
    then  call ERROR("Illegal type conversion")
    else
        if n.type = integer and t = float
        then
            /*   replace node n by convert-to-float of node n           */ (13)
        else   /* nothing needed */
end
```

Figure 2.12: Type analysis for ac.

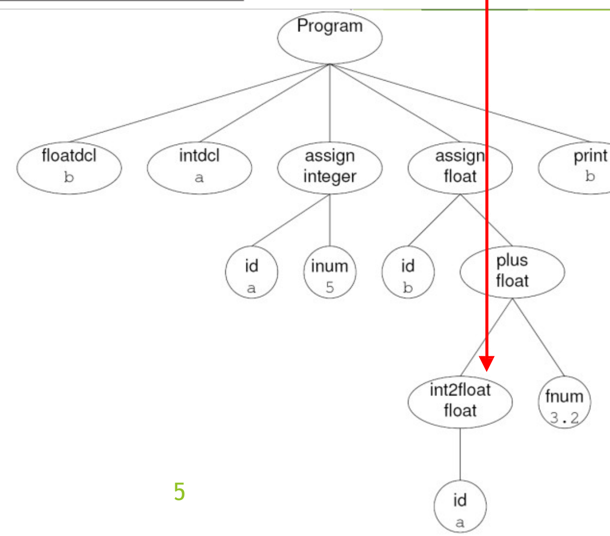

Figure 2.13: AST after semantic analysis.

Code Generation

- The final task undertaken by a compiler

  → The formulation of **target-machine instructions** that faithfully represent the semantics (i.e.,meaning) of the source program

  - Translation exercise of the textbook consists of generating source code that is suitable for the dc program, which is a simple calculator based on a stack machine model

- In a **stack machine**, most instructions receive their input from the contents at or near the **top of an operand stack**
  - The result of most instructions is pushed on the stack
  - Programming languages such as C# and Java are frequently translated into a portable, stack machine representation

Code Generation (Cont'd)

- The AST was transformed and decorated with **type information** during semantic analysis
  - Such information is required for **selecting the proper instructions**
- The instruction set on most computers distinguishes between **float** and **integer** data types
  - ARM processors have the instructions
    - **VADD** for Floating-point Add
    - **VDIV** for Floating-point Divide
    - **ADD** for Integer Add
    - **SDIV** for Signed Divide

Generating Code for *ac*

- Traverse the AST
  - starting at its root and working toward its leaves
- The code generator is called recursively
  - to generate code for the left and right subtrees
  - The resulting values will be at top-of-stack
- **VISIT(Computing n)**
  - generates code for **plus** and **minus**
  - The appropriate operator is then emitted (**Marker 15**) to perform the operation

```
procedure VISIT( Assigning n )
    call CODEGEN( n.child2 )
    call EMIT( "s" )
    call EMIT( n.child1.id )
    call EMIT( "0 k" )                                  (14)
end
procedure VISIT( Computing n )
    call CODEGEN( n.child1 )
    call CODEGEN( n.child2 )
    call EMIT( n.operation )                            (15)
end
procedure VISIT( SymReferencing n )
    call EMIT( "l" )
    call EMIT( n.id )
end
procedure VISIT( Printing n )
    call EMIT( "l" )
    call EMIT( n.id )
    call EMIT( "p" )
    call EMIT( "si" )                                   (16)
end
procedure VISIT( Converting n )
    call CODEGEN( n.child )
    call EMIT( "5 k" )                                  (17)
end
procedure VISIT( Consting n )
    call EMIT( n.val )
end
```

Figure 2.14: Code generation for ac

Generating Code for *ac* (Cont'd)

- **VISIT(Assigning n)**
  - causes the expression to be evaluated
  - Code is then emitted to **store** the value in the appropriate **dc register**
  - The calculator's **precision** is then **reset** to integer by setting the fractional precision to zero **Marker 14**

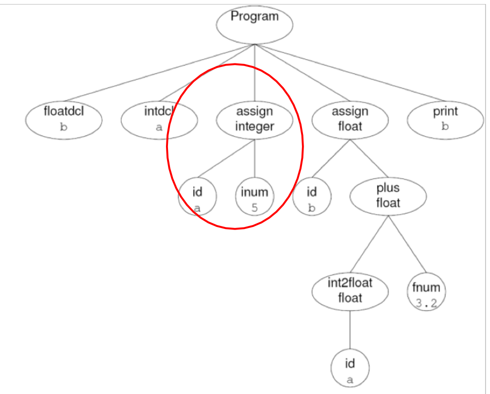

Figure 2.13: AST after semantic analysis.

Code
5
sa
0 k

```
procedure visit( Assigning n )
    call CodeGen(n.child2)            5
    call Emit("s")                    s
    call Emit(n.child1.id)            a
    call Emit("0 k")                  0k
end
```

(14)

Figure 2.14: Code generation for ac

Generating Code for *ac*

- VISIT(Computing n)
  - generates code for **plus** and **minus**
    - The appropriate operator is then emitted (**Marker 15**) to perform the operation

```
procedure VISIT( Assigning n )
    call CODEGEN( n.child2 )
    call EMIT( "s" )
    call EMIT( n.child1.id )
    call EMIT( "0 k" )                                   (14)
end
procedure VISIT( Computing n )
    call CODEGEN( n.child1 )
    call CODEGEN( n.child2 )
    call EMIT( n.operation )                             (15)
end
procedure VISIT( SymReferencing n )
    call EMIT( "l" )
    call EMIT( n.id )
end
procedure VISIT( Printing n )
    call EMIT( "l" )
    call EMIT( n.id )
    call EMIT( "p" )
    call EMIT( "si" )                                    (16)
end
procedure VISIT( Converting n )
    call CODEGEN( n.child )
    call EMIT( "5 k" )                                   (17)
end
procedure VISIT( Consting n )
    call EMIT( n.val )
end
```

Figure 2.14: Code generation for ac

Generating Code for *ac* (Cont'd)

- **VISIT(SymReferencing n)**
  - causes a value to be retrieved from the appropriate **dc register** and **pushed onto the stack**
  - Push register
    → Load (**l**) symbol (**a**) to dc register
    → `**la**'

```
procedure visit( Assigning n )
    call CodeGen( n.child2 )
    call Emit( "s" )
    call Emit( n.child1.id )
    call Emit( "0 k" )                                   (14)
end
procedure visit( Computing n )
    call CodeGen( n.child1 )
    call CodeGen( n.child2 )
    call Emit( n.operation )                             (15)
end
procedure visit( SymReferencing n )
    call Emit( "l" )
    call Emit( n.id )
end
procedure visit( Printing n )
    call Emit( "l" )
    call Emit( n.id )
    call Emit( "p" )
    call Emit( "si" )                                    (16)
end
procedure visit( Converting n )
    call CodeGen( n.child )
    call Emit( "5 k" )                                   (17)
end
procedure visit( Consting n )
    call Emit( n.val )
end
```

Figure 2.14: Code generation for ac

Generating Code for *ac* (Cont'd)

- **VISIT(Printing n)**
  - is tricky because dc does not discard the value on top-of-stack after it is printed
  - The instruction sequence `**si**' is generated at **Marker 16**,
  - thereby popping the stack and storing the value in dc's **i** register
  - Conveniently, the ac language precludes a program from using this register because the **i** token is reserved for spelling the terminal symbol integer

```
procedure visit( Assigning n )
    call CodeGen( n.child2 )
    call Emit( "s" )
    call Emit( n.child1.id )
    call Emit( "0 k" )                    (14)
end
procedure visit( Computing n )
    call CodeGen( n.child1 )
    call CodeGen( n.child2 )
    call Emit( n.operation )              (15)
end
procedure visit( SymReferencing n )
    call Emit( "l" )
    call Emit( n.id )
end
procedure visit( Printing n )
    call Emit( "l" )
    call Emit( n.id )
    call Emit( "p" )
    call Emit( "si" )                     (16)
end
procedure visit( Converting n )
    call CodeGen( n.child )
    call Emit( "5 k" )                    (17)
end
procedure visit( Consting n )
    call Emit( n.val )
end
```

Figure 2.14: Code generation for ac

Generating Code for *ac* (Cont'd)

- **VISIT(Converting n)**
  - causes a change of type from integer to float at **Marker 17,**
  - which accomplished by setting dc's precision to five fractional decimal digits → `**5 k**'

```
procedure VISIT( Assigning n )
    call CODEGEN( n.child2 )
    call EMIT( "s" )
    call EMIT( n.child1.id )
    call EMIT( "0 k" )                                  (14)
end
procedure VISIT( Computing n )
    call CODEGEN( n.child1 )
    call CODEGEN( n.child2 )
    call EMIT( n.operation )                            (15)
end
procedure VISIT( SymReferencing n )
    call EMIT( "l" )
    call EMIT( n.id )
end
procedure VISIT( Printing n )
    call EMIT( "l" )
    call EMIT( n.id )
    call EMIT( "p" )
    call EMIT( "si" )                                   (16)
end
procedure VISIT( Converting n )
    call CODEGEN( n.child )
    call EMIT( "5 k" )                                  (17)
end
procedure VISIT( Consting n )
    call EMIT( n.val )
end
```

Figure 2.14: Code generation for ac

Generating Code for *ac* (Cont’d)

Code	Source	Comments
`5`	`a = 5`	Push 5 on stack
`sa`		Pop the stack, storing (s) the popped value in register a
`0 k`		Reset precision to integer
`la`	`b = a + 3.2`	Load (l) register a, pushing its value on stack
`5 k`		Set precision to float
`3.2`		Push `3.2` on stack
`+`		Add: `5` and `3.2` are popped from the stack and their sum is pushed
`sb`		Pop the stack, storing the result in register `b`
`0 k`		Reset precision to integer
`lb`	`p b`	Push the value of the `b` register
`p`		Print the top-of-stack value
`si`		Pop the stack by storing into the `i` register

Figure 2.15: Code generated for the AST shown in Figure 2.9.