Type Checking

Chapter 6, Section 6.3, 6.5

Inside the Compiler: **Front End**

- Lexical analyzer (aka scanner)
  - Converts ASCII or Unicode to a stream of tokens
- Syntax analyzer (aka parser)
  - Creates a parse tree (or AST) from the token stream
- Semantic analyzer
  - Type checking and conversions; other semantic checks
- Generator of intermediate code
  - Creates lower-level **intermediate representation** (IR): e.g., three-address code

Types in Compilers

- Type checking: at compile time, guarantee that the run-time behavior of the program will be correct
  - The type of the operands match the type of the operator (e.g., in Java && requires boolean operands)
  - The types of actual parameters in a function call match the types of the formal parameters
  - Many other examples based on the type system of the language
- Code generation
  - Allocation of memory based on types (e.g., how many bytes do we need for a struct with an int and a float?)
  - Insert explicit type conversions

Outline

- Useful machinery: attribute grammars
- Analysis of declarations
  - Representation of types
- Type checking
  - What is the type of an expression, given the types of its subexpressions? (synthesized attributes)
  - Is there a type error in the program?
- Implicit type conversions: not in the source code, but must be accounted for during type checking and code generation
  - E.g., **int** can be "silently promoted" to **double**

Attribute Grammars

- Given a context-free grammar: for each non-terminal, define zero, one, or more attributes
  - Called "syntax-directed definitions" in the Dragon Book
- An evaluation rule for each production
- Example: value of an expression with constants only

$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$\mid T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$\mid F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$\mid$ **const**	$F.val =$ **const**.*lexval*

  - Attribute *val* for each *E*, *T*, and *F* node
  - Attribute *lexval* for each **const** code

Attribute Grammars

- An attribute of a non-terminal X can be either **synthesized** or **inherited** (but not both)
  - Synthesized attribute X.a: computed from attributes of X's children (this is an oversimplification)
  - Inherited attribute X.a: computed from attributes of X's parent (this is an oversimplification)
- A *lexval* attribute for a terminal (i.e., leaf node)
  - Not computed by evaluation rules, but just provided by the lexical analyzer (e.g., *lexval* for each **const** code)

Back to Types: Type Expressions

- What is a type and how do we represent it inside a compiler? We will use type expressions for this
- A primitive type is a type expression (e.g., *boolean*, *char*, *byte*, *integer*, *long, float, double*, *void*)
- An **array** type constructor, applied to
  - non-array type (for the array elements)
  - sequence of integers (for sizes of array dimensions) and a non-array type expression
  - E.g., *array(integer,10,20)* to represent the type of array **x** with declaration **int x[10][20];**
- In our projects:
  - Types.INT and Types.DOUBLE for primitive types
  - No representation for array types; you need to add it

Type Expressions

- A **record** type constructor, applied to a list of pairs (field name, type expression), is a type expression
  - E.g., *record { x:float, y:float, rgb:array(byte,3) }* could be the type expression for a C struct with fields *x, y* for point coordinates and field *rgb* for RGB point color
- A **function** type constructor →, applied to two type expressions, is a type expression
  - E.g., suppose we have a function that takes an array of 10 floats and returns their sum

    array(float,10) → *float*

Type Expressions

- A **tuple** type constructor ×, applied to a list of type expressions
  - E.g., *record { x:float, y:float, rgb:array(byte,3) }* × *float* → *record { x:float, y:float, rgb:array(byte,3) }* is a function taking two parameters: a record and a float
- Type expressions can naturally be represented with trees or DAGs (details in Dragon Book)
- From the type expression, we can determine how many bytes will be needed in the generated code
  - Note: there may be hardware alignment constraints – e.g., each integer must start at an address divisible by 4; so, for type *record { integer, boolean, integer }* padding may be needed between the second and the third field (unused 3 bytes)

Declarations in Our Projects

decl → **int id** *arrayDecl* **;** | **double id** *arrayDecl* **;**

arrayDecl → **[int_const]** *arrayDecl* **;** | **ε**

AST representation:

class Decl with fields String **id**, int **type**, List<Integer> **dims**

Project 3: create a symbol table and use for type checking

- Create representation for array types
- After parsing, examine all declarations and populate the symbol table with each **id** and its type
- Semantic check: (as in C) re-declarations are not allowed
- Then, examine all expressions and check them

Type Checking

- Look at expressions to see if declared types are consistent with variable usage
- Many checks of the form if **(type expression 1 == type expression 2)** OK otherwise report type error
- Checking: (1) types of subexpressions OK?
  (2) decide the type of the whole expression

Example (subset of the language for the project)

$E \rightarrow$ **id** | **int_const** | **double_const**

$E \rightarrow$ **id [**E_1**]** for simplicity, here we discuss only 1-dimensional arrays

$E \rightarrow E_1 + E_2 \mid E_1 < E_2 \mid E_1 = E_2$

We will use a synthesized attribute *E.type*

First version of checking: strict matching of types

Second version (for the project): allow type conversions, similarly to C

Attribute Grammar for Strict Type Checking

$E \rightarrow$ **id**

- Error if the variable is not declared
- *E*.*type* = *getType*(**id**.*lexval*)   // get from symbol table

$E \rightarrow$ **int_const**

- *E*.*type* = *int*

$E \rightarrow$ **double_const**

- *E*.*type* = *double*

Attribute Grammar for Strict Type Checking

$E \rightarrow$ **id [** E_1 **]**

- Error if the variable is not declared
- If (*getType*(**id**.*lexval*) is not *array(X,Y)*) error
- If (E_1.*type* is not *int*) error
- *E*.*type* = *X*

$E \rightarrow E_1$ **=** E_2

- If (E_1.*type* is not *int* or *double*) error
- If (E_2.*type* is not *int* or *double*) error
- If (E_1.*type* is not the same as E_2.*type*) error
- *E*.*type* = E_1.*type*

Project 3: Also need to check that the left-hand-side of an assignment operator has an l-value: it can only be **id** or **id [**E_1**]**

Attribute Grammar for Strict Type Checking

$E \rightarrow E_1 + E_2$

- If (E_1.*type* is not *int* or *double*) error
- If (E_2.*type* is not *int* or *double*) error
- If (E_1.*type* is not the same as E_2.*type*) error
- E.*type* = E_1.*type*

$E \rightarrow E_1 < E_2$

- If (E_1.*type* is not *int* or *double*) error
- If (E_2.*type* is not *int* or *double*) error
- If (E_1.*type* is not the same as E_2.*type*) error
- E.*type* = *int*

In C there are no boolean types; the result of **<** is an integer

Implicit Type Conversions

- Values of one type are converted to another type
  - E.g. addition: 3.0 + 4 : silently converts 4 to 4.0
  - E.g. our earlier typechecking rules imply that operator **+** has types $int \times int \rightarrow int$ and $double \times double \rightarrow double$
  - But now we also allow $double \times int \rightarrow double$ and $int \times double \rightarrow double$
- In general, whenever the type of an expression is not appropriate
  - The compiler silently converts it to another type
  - Or, if not possible: compile-time error

Example: Conversions in Java [no need to remember this]

- Widening: converting a value into a “larger” type; performed silently by the compiler
- Widening primitive conversions in Java
  - *byte* to *short*, *int*, *long*, *float*, or *double*
  - *short* to *int*, *long*, *float*, or *double*
  - *char* to *int*, *long*, *float*, or *double*
  - *int* to *long*, *float*, or *double*
  - *long* to *float* or *double*
  - *float* to *double*

Some Examples: Conversions in Java

- Assignment conversion: when the value of an expression is assigned to a variable, convert the expr. value to the type of the variable
- Call conversion: applied to each argument of a call
  - The type of the argument expression is converted to the type of the corresponding formal parameter
- Binary numeric conversion: for **+**, **-**, *****, etc.
  - If either operand is *double*, the other is converted to *double*
  - Otherwise, if either operand is *float*, the other is converted to *float*
  - Otherwise, if either operand is *long*, the other is converted to *long*
  - Otherwise, both are converted to *int*

Back to Our Simplified Language

- Let us allow implicit widening conversions from int to double. What will be affected?
- For all binary operators: remove "If (E_1.*type* is not the same as E_2.*type*) error"
- Old rule for $E \rightarrow E_1 + E_2$
  - If (E_1.*type* is not *int* or *double*) error
  - If (E_2.*type* is not *int* or *double*) error
  - If (E_1.*type* is not the same as E_2.*type*) error
  - *E.type* = E_1.*type*
- New rule
  - First two checks are the same
  - *E.type* = E_1.*type*, if E_2.*type* is *integer*
  - *E.type* = *double,* otherwise

How About Assignments?

- New rule for $E \rightarrow E_1$ **=** E_2 (assignment conversion, as in C: right-hand-side value will be converted to the type of the left-hand side expression, if possible)
  - If (E_1.*type* is not *int* or *double*) error
  - If (E_2.*type* is not *int* or *double*) error
  - If (E_1.*type* is *int* and E_2.*type* is *double*) error
  - *E.type* = E_1.*type*

Project 3

- Type checking based on this approach
- For each AST node representing an expression, remember its type
  - E.g., add a field in class Expr and set it to *E.type*
- In preparation for Project 4: for each binary expression, create a temporary variable of the corresponding type
  - E.g., for **a = b + c + d;** Project 4 will create code

    _t1 = b + c;

    a = _t1 + d;

    For this, we will need to determine the type of **_t1**, which is the same at the type of expression **b + c**