Semantic Analysis

Fan Long

University of Toronto

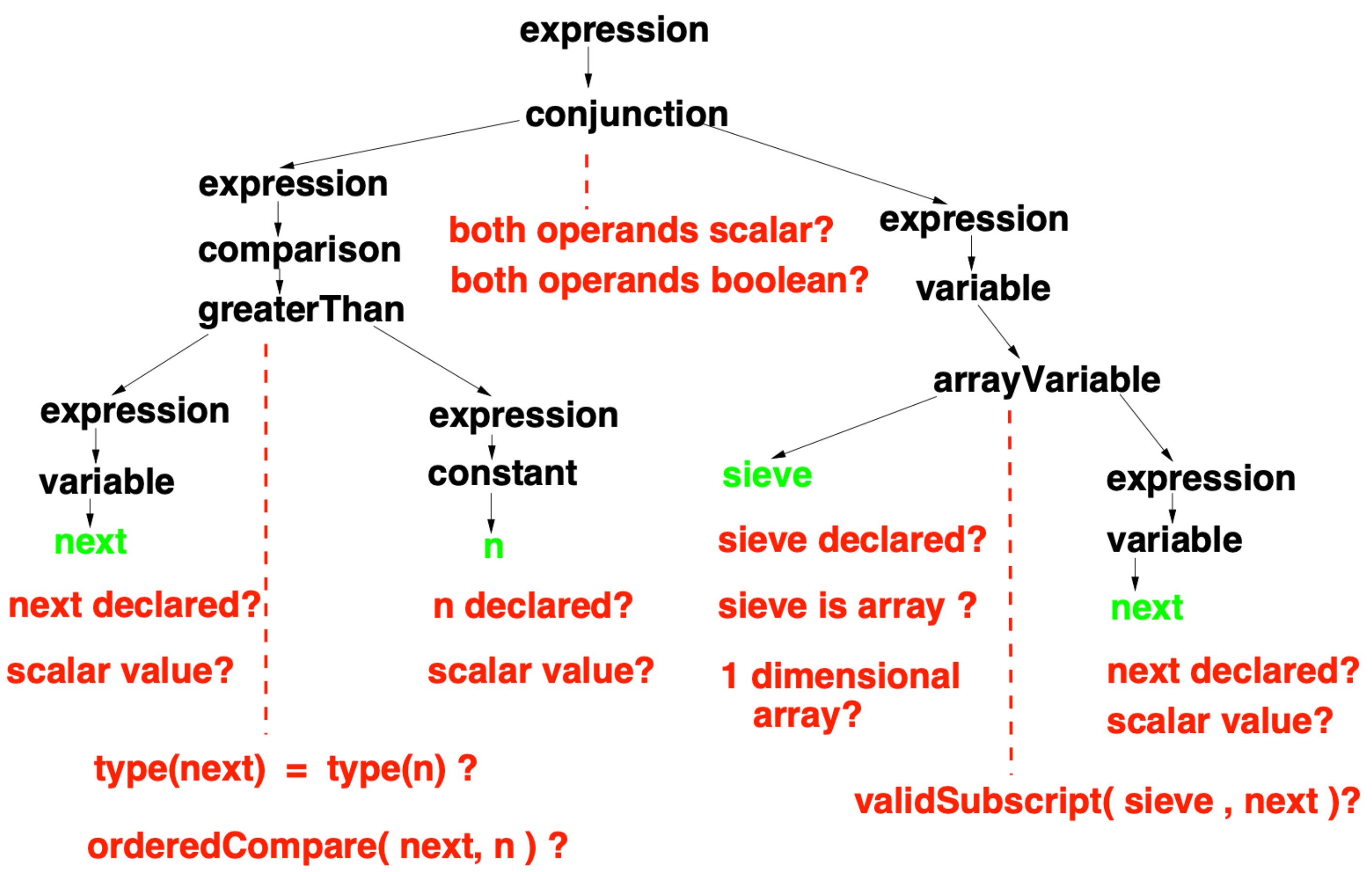
Semantic Analysis

- Validation of *non-syntactic* language constraints
  - Static semantic analysis - during compilation
  - Dynamic semantic analysis - run time checks
- Semantic analysis
  - Visibility and Accessibility Analysis
  - Type Checking
  - Proper Usage Analysis
  - Range and Value Analysis
  - Range and Value Propagation
- The compilers symbol table is usually built during semantic analysis as a side effect of declaration processing.

UNIVERSITY OF TORONTO

Semantic Analysis Example

```
next > n || sieve[ next]
```



UNIVERSITY OF TORONTO

Type Equivalence, Compatibility, Suitability

- Every language definition includes rules about when objects of different types can be used together in various constructs.
- Many languages have several rules concerning types:
  - *Type Equivalence Conditions under which objects of two different types are considered to be* equivalent. *Equivalence is usually required when the* addresses *of data objects are being manipulated.*
  - *Type Compatibility Conditions under which objects of two different types are considered to be* compatible. *Compatibility is usually required for assignments.*
  - *Type suitability Conditions under which objects of two different types are considered* suitable *to be used together. Suitability is usually required for operands in expressions.*
- The two most widely used Type Equivalence Rules are *structural equivalence* and *name equivalence*

UNIVERSITY OF TORONTO

Type Equivalence Rules

- Define: Name Type Equivalence
  - Two types are name equivalent iff they ultimately derive from a common definition.
  - *ultimately derive* allows type renaming, e.g., **type** S : T
  - In implementation terms, two named types are equivalent if they refer to the same type table entry.
- Define: Structural Type Equivalence
  - Type types are structurally equivalent if their definitions have the same structure and corresponding values are equal.
  - Structural equivalence is isomorphism for types.
  - In implementation terms a parallel walk of two type trees is required to establish structural equivalence.

UNIVERSITY OF TORONTO

Algorithm for Structural Equivalence

function isEquivalent(**type** T_1 , **type** T_2) **:** **Boolean** $\{$

/* Test types T_1 and T_2 for structural equivalence */

if T_1 **.** typeKind **not** = T_2 **.** typeKind **then**

return false /* node mismatch */

for each value $field_i$ in T_1 , T_2

if T_1 **.** $field_i$ **not** $=_{lang}$ T_2 **.** $field_i$ **then**

return false /* value mismatch */

for each $subtree_i$ of T_1 , T_2

if not isEquivalent(T_1 **.** $subtree_i$, T_2 **.** $subtree_i$) **then**

return false /* subtree mismatch */

return true /* all values and subtrees match */

end /* isEquivalent */

Type Equivalence Example

```
typedef struct {
    int B;
    int C;
} A;

typedef A D;
```

```
typedef struct {
    int X;
    int Y;
} F;
```

- A and F are structurally equivalent but not name equivalent
- D and A are name equivalent and thus structurally equivalent

Type Equivalence Checking

- Type equivalence checking is used to guarantee that the type of data object that a pointer points at is *the same* as the declared type for the pointer.
- This check is necessary to ensure that when the pointer is used to access parts of the object that access will yield the correct result.
- Type equivalence implies **memory image equivalence**, i.e., the two types have identified memory images.
- Usually type equivalence checking is done in two cases
  - When the address of a data object is assigned to a pointer.
  - When a variable is passed as a reference (i.e. address) parameter.
- Memory image equivalence guarantees that when an address is used to access a type, the internal parts of the type (e.g. fields in a structure) will be accessed correctly

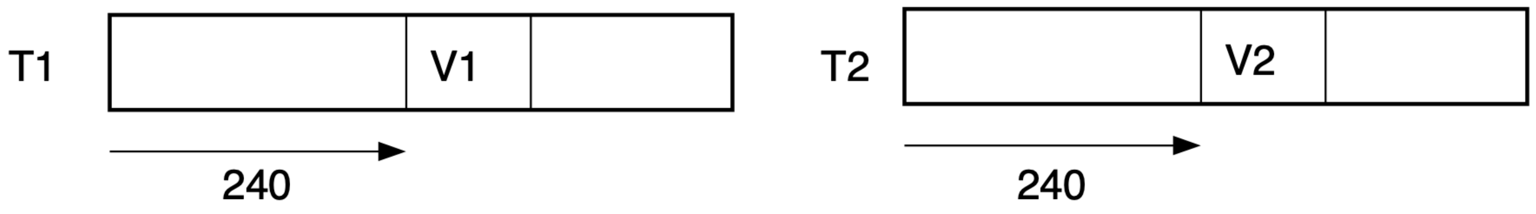
UNIVERSITY OF TORONTO

Memory Equivalence Example

```
typedef struct {
    …
    int V1;
    …
} T1;
```

```
typedef struct {
    …
    int V2;
    …
} T2;
```

- Then structural equivalence guarantees that for all corresponding fields *Fi* in T1 and T2 the address of the field relative to the start of the type is equal.



- Implementing memory equivalence constrains the way that record fields are allocated within a record/structure.

Why Memory Equivalence Matters

```
T1 a;
T2* b;
void foo(T1 x, T1* y) {
     …;
     y.V1 = x.V1;
}
```

Defined only if memory equivalence:

```
foo(a, (T1*)b)
```

UNIVERSITY OF TORONTO

Type Compatibility and Suitability Checking

- Type compatibility checking is used to determine when a value of some given type is compatible with the type of some variable.
- Compatibility checking is typically used to check
  - That the expression on the right side of an assignment statement may be legally assigned to the variable on the left side.
  - That an expression being passed as a value parameter to a function can be legally assigned to the corresponding formal parameter variable.
- Type suitability checking is used to determine when one or more values can be used together or with an operator.
- Typical instances of suitability checking include
  - checking that the operand of a unary operator is of a suitable type for the operator.
  - checking that both operands of a binary operator are of suitable types for the operator and for each other.

UNIVERSITY OF TORONTO

Type Checking in MiniC

- MiniC only has primitive and array types.
- We are going to maintain the type of each expression in the program.

- Types must be identical for parameter passing and assignments.
- The return type and the returned expression must match.
- Arithmetic opcode must have int operand.
- Boolean opcode must have bool operand.
- IfStmt and ForStmt conditional expresions must have boolean type.
- Index expression of an array must have int type.

Visibility and Accessibility

- Visibility analysis determines whether a given reference to an identifier at some point it the program is legal according to the scope rules of the language.
- The perform visibility analysis, the compiler must keep track of declared symbols behaves (logically at least) in a way that is consistent with the scope rules of the language.
- Semantic analyzer must also track the scope structure of the program as it is being processed.
- Accessibility analysis determines whether a visible identifier can be accessed in a given way at some point in a program. Examples:
  - **const** in C prevents from being modifed

UNIVERSITY OF TORONTO

Usage Analysis

- Verify the appropriate use of constants, variables, types, functions.
  - Is an identifier used as a constant actually a constant?
  - Is an identifier used as a variable actually a variable?
  - Is an identifier used as a type actually a type?
  - Is a variable used as a scalar variable actually scalar?
  - Is a variable used as an array actually an array of the right dimensionality?
  - Is an identifiers used as a function actually a function?
  - Are all variables and constants being used in a way that is consistent with their declared type?
  - Are the operands of all operators compatible with the operator?
- Detection of potential runtime faults?

Usage Analysis Examples

```
typedef int T1;
int x, y[10];
int foo(int p);
```

Statement	Error
`T1 = 17;`	Assignment to a type
`f(1) = 1;`	Assignment to a function
`y = x;`	Assign array to scalar variable
`y[0] = x.a;`	Field selection on a scalar variable
`x = foo(1, y[0]);`	Wrong number and type of parameters of foo

Runtime Fault Detection Example

```
void foo(int x, int y) {
  int a, b, c, d[10];
  if (x == 1) a = 0; else a = y;
  b = 0; c = -1;
  …
}
```

Expression	Error
`x / b`	Divide by zero
`d[c]`	Array out of bound
`d[a]`	Maybe array out of bound? How to detect? What to do?

Programming Language Influences

- Definition of the programming language being compiled can have a major influence on the way semantic analysis is implemented.
- Languages that do not require *declaration before use* (e.g. PL/I, C) will usually require a separate semantic analysis pass.
- Language with a weak or non-existent declaration structure (e.g. Lisp, Icon, APL, Prolog) require that most semantic analysis be done dynamically.
- Object-Oriented languages (e.g. Smalltalk, C++, Java) that allow dynamic object binding may require extensive run time checking.
- The presence of dynamically sized objects in a language (e.g. arrays whose bounds are determined at run time) may require more run time checking.

UNIVERSITY OF TORONTO

Semantic Analysis of Declarations

- The canonical declaration associates one or more identifiers with type, structure and size attributes. The syntax of the language determines how these associations are made.

```
C       int i, ia[10], ig(), *ip ;
PL/I    DECLARE ( I, IA(0:9) ) FIXED BINARY(31,0) ;
        DECLARE IP POINTER,
            IG ENTRY RETURNS( FIXED BINARY(31,0) );
```

- Declaration processing involves collecting attribute information and applying defaults for missing attributes.
- Essentially declaration processing involves filling in a symbol table entry for each declared item.

Basic Declaration Processing

- Accumulate list of identifiers being declared

```
int i, j, k, l ;
```

- Lookup each identifier in the current scope to check for multiple declaration in the current scope.
- Enter each symbol in the symbol table for the current scope.
- Associate attributes from declaration with each identifier. Apply language-specific defaults as required.

```
int a, b[10], *c, *d[], **e, f(), *g(), *(h()) ;
```

- Process initial value if one is present.

```
int i, j = 3 ;
```

Expression Processing

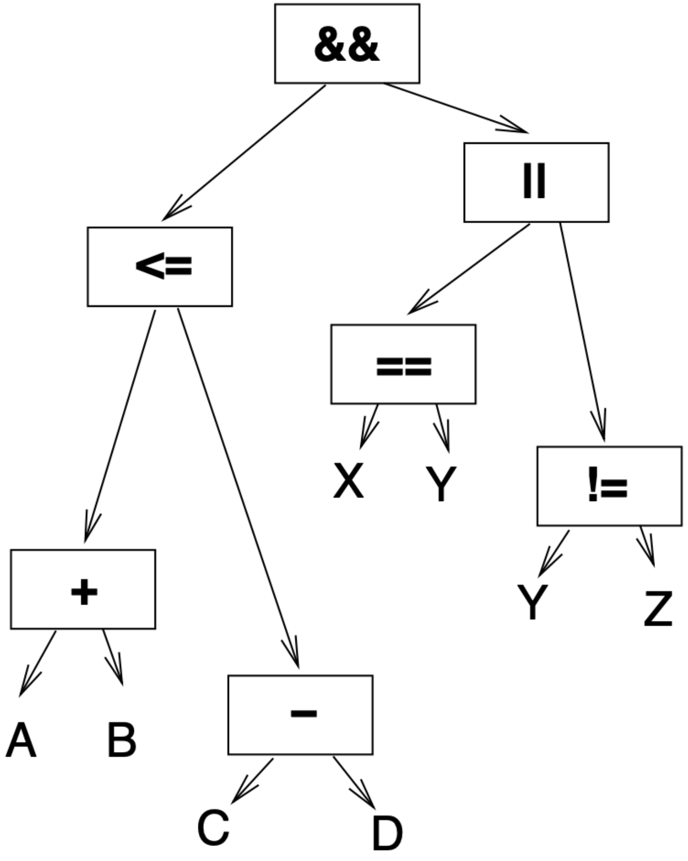
- Semantic analysis expression processing involves type and usage checking of expressions. It assumes that references to identifiers (e.g. variables, named constants, etc.) are processed as described above.
- Due to the embedding of expressions within expressions, stacks are often used to save type and symbol information during expression processing.
- Conceptually expression processing is a depth-first walk of the abstract syntax tree for each expression.
- Frequently expression processing if operator driven, i.e. the arithmetic operator at each node in the expression tree determines what checks are performed on the operands attached to that node.

Expression Processing

Operator(s)	Actions
constant	Tag node with type of constant
variable	Add link to symbol table entry for variable
	Tag node with type of variable
$+, -, *, /$	Verify left and right operands are of a suitable arithmetic type.
	Record arithmetic result type of operator
$<, <=, ==, != , >=, >$	Verify operands are of a comparable type
	Verify comparison is legal
	Record result type is boolean
$++, --$	Check operand is variable compatible with arithmetic
	record result type and non-variable status
and, **or**, **not**	Check operands are boolean type
	Record result type is boolean

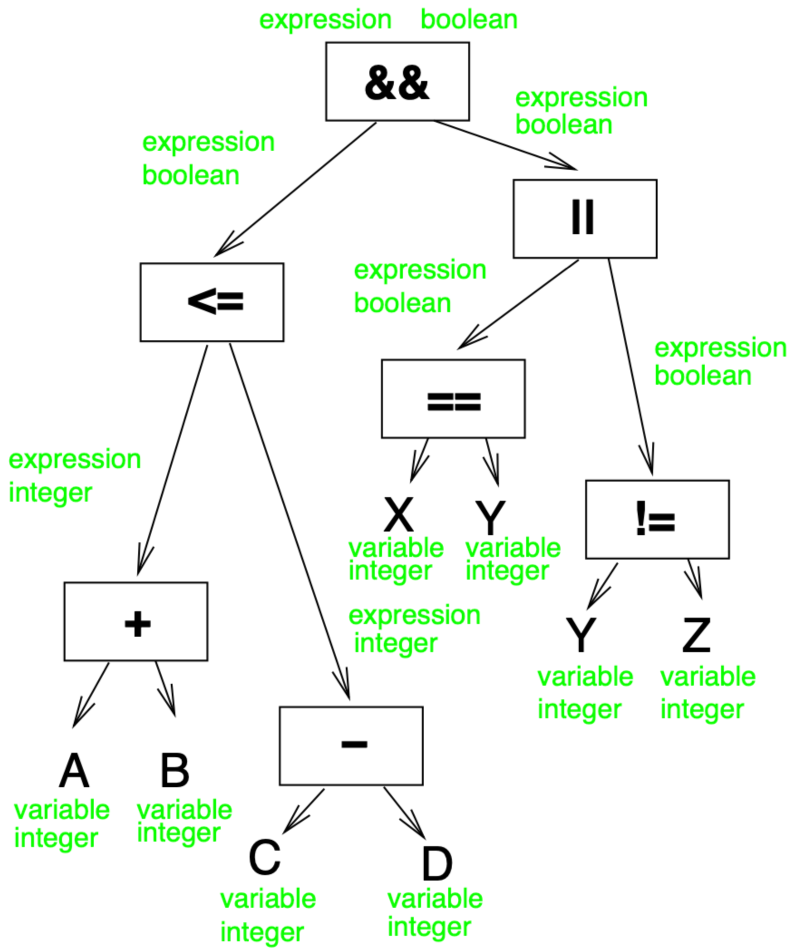
Expression Processing Example

(A + B) <= (C – D) && (X == Y || Y != Z)



Expression Processing Example

(A + B) <= (C – D) && (X == Y || Y != Z)



Q/A?