COMPILER (CSE 4120)

(Lecture 10: Semantic Analysis 3)

Sungwon Jung, Ph.D.

Mobile Computing and Data Engineering Lab.
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Dept. of Computer Science at Sogang University

Data Type and Type Checking

- Type inference:
  - The computation and maintenance of information on data types
- Type checking:
  - To ensure that each part of a program makes a sense under the type rules of the language
- Data type information may be either static or dynamic, or a mixture of two
  - Dynamic data type: LISP
  - Static data type: Pascal, C, and Ada
    - Used to determine the size of memory needed for the allocation of each variable and the way that memory can be accessed

Data Type and Type Checking

- Data type
  - A set of values with certain operations on those values
  - Usually described by a type expression
    - a type name : e.g., integer
    - A structured expression: e.g., array [1..10] of real
  - Type expressions can occur in several places in a program
    - In variable declarations
      - var x: array [1..10] of real;
    - In type declarations
      - type RealArray = array [1..10] of real;
  - Type information can be
    - Explicit
    - Implicit: e.g., const greeting = "Hello!"; ⇒ array [1..6] of char

Data Type and Type Checking

- Type information in declarations are
  - Maintained in the symbol table
  - Retrieved by the type checker whenever the associated name are referenced
- New types are inferred from the types already defined
  - e.g., a[i]
    - Data types of names **a** and **i** are fetched from the symbol table
      - If **a** has type **array[1..10] of real** and **i** has type **integer**, then the subexpression **a[i]** is determined to be type correct and has type **real**
      - Range checking is not statically determinable
- Kinds of type expressions available in a language and the type rules of the language governing the use of these type expressions determines
  - The way data types represented
  - The way a symbol table maintains type information
  - The rules used by a type checker to infer types

Type Expressions and Type constructors

- Simple types
  - int, double, boolean, char
    - Their values exhibit no explicit internal structure
- Given a set of predefined types, new data types can be created using **type constructors** such as array and record or struct
  - Can be viewed as functions taking existing types as parameters and return new types
  - Often called structured types
  - Classification
    - Array, Record, Union, Pointer, Function, Class

Type Names, Type Declarations, and Recursive Types

- A programmer can assign names to type expressions
  - Type declarations (or type definitions) include the **typedef** mechanism in C
    - Examples
      - typedef struct { double r; int i; } RealIntRec
      - struct RealIntRec { double r; int i; }; struct RealIntRec x;
  - Type declarations cause the declared type names to be entered into the symbol table just as variable declarations cause variable names to be entered
  - Attributes associated with type names in the symbol table
    - Scope
    - Type expression corresponding to the type name

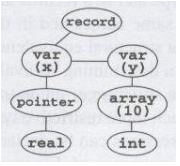
Type Equivalence

- A type checker must frequently answer the question of when two type expressions represent the same type
  - Type equivalence
- Use a syntax tree to represent the type expression
  - record

    x: pointer to real;

    y: array [10] of int

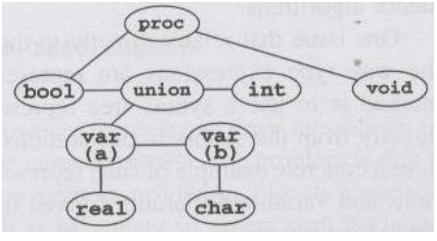
    end



var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id :** *type-exp*
type-exp → *simple-type* | *structured-type*
simple-type → **int** | **bool** | **real** | **char** | **void**
structured-type → **array [num] of** *type-exp* |
record *var-decls* **end** |
union *var-decls* **end** |
pointer to *type-exp* |
proc (*type-exps* **)** *type-exp*
type-exps → *type-exps* **,** *type-exp* | *type-exp*

Type Equivalence

- The type expression
  - **proc(bool, union a: real; b:char end, int):void**



Type Equivalence

- Two kinds of type equivalences:
  - Structural equivalence
    - Two types are the same iff they have the same structure
  - Name equivalence
    - Two types expressions are equivalent iff they are either the same simple type or are the same type name
- Structural equivalence
  - If syntax trees are used to represent types, two types are equivalent iff they have syntax trees that are identical in structure
    - Two arrays are not equivalent unless they have same size and component type
    - Two records are not equivalent unless they have the same components with the same names and in the same order
    - Possible for different choices to be made in a structural equivalence algorithm
      - e.g., the size of the array could be ignored in determining equivalence

Structural Equivalence

- Pseudocode

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 and t2 are of simple type then return t1 = t2
  else if t1.kind = array and t2.kind = array then
     return t1.size = t2.size and typeEqual ( t1.child1, t2.child1)
  else if t1.kind = record and t2.kind = record
       or t1.kind = union and t2.kind = union then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1 ≠ nil and p2 ≠ nil do
       if p1.name ≠ p2.name then
          temp := false
       else if not typeEqual ( p1.child1 , p2.child1 )
       then temp := false
       else begin
         p1 := p1.sibling ;
         p2 := p2.sibling ;
       end;
    return temp and p1 = nil and p2 = nil ;
  end
```

- Pseudocode

```
  else if t1.kind = pointer and t2.kind = pointer then
     return typeEqual ( t1.child1 , t2.child1 )
  else if t1.kind = proc and t2.kind = proc then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1 ≠ nil and p2 ≠ nil do
       if not typeEqual ( p1.child1 , p2.child1 )
       then temp := false
       else begin
         p1 := p1.sibling ;
         p2 := p2.sibling ;
       end;
    return temp and p1 = nil and p2 = nil
           and typeEqual ( t1.child2 , t2.child2 )
  end
  else return false ;
end ; (* typeEqual *)
```

```
var-decls → var-decls ; var-decl | var-decl
var-decl → id : simple-type-exp
type-decls → type-decls ; type-decl | type-decl
type-decl → id = type-exp
type-exp → simple-type-exp | structured-type
simple-type-exp → simple-type | id
simple-type → int | bool | real | char | void
structured-type → array [num] of simple-type-exp |
                  record var-decls end |
                  union var-decls end |
                  pointer to simple-type-exp |
                  proc ( type-exps ) simple-type-exp
type-exps → type-exps , simple-type-exp | simple-type-exp
```

Type Equivalence

- Name equivalence
  - A very strong sort of type equivalence
  - Given the type declarations **t1 = int; t2 = int;**
    - The type **t1** and **t2** are not equivalent and are also not equivalent to **int**

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 and t2 are of simple type then
     return t1 = t2
  else if t1 and t2 are type names then
     return t1 = t2
  else return false ;
end;
```

    - The actual type expressions corresponding to type names must be entered into the symbol table

Type Equivalence

- Name equivalence
  - Problem when type expressions other than simple types or type names continue to be allowed in variable declarations, or as subexpressions of type expressions
    - No explicit name given to a type expression
    - A complier generate an internal name for the type expression different from any other names
      - **x: array [10] of int; y: array [10] of int;**
      - The variables **x** and **y** are assigned different type names

Type Equivalence

- Care must be taken in implementing structural equivalence when recursive type references are possible
  - Can result in an infinite loop
    - Can be avoided by modifying the behavior of the call *typeEqual*(*t1*,*t2*) in the case where t1 and t2 are type names to include the assumption that they are already potentially equal

```
t1 = record
       x: int;
       t: pointer to t2;
     end;

t2 = record
       x: int;
       t: pointer to t1;
     end;
```

    - If the *typeEqual*(*t1*,*t2*) function ever returns to the same call, success can be declared at that point

Type Equivalence

- Declaration equivalence
  - A weaker version of name equivalence used by C and Pascal
  - t2 = t1 are interpreted as establishing type aliases, rather than new types
    - Example: **t1 = int; t2 = int**
      - Both **t1** and **t2** are equivalent to **int**
  - Every type name is equivalent to some base type name, either a predefined type or given by a type expression
    - t1 = array [10] of int; t2 = array [10] of int; t3 = t1;
    - The type name t1 and t3 are equivalent but neither is equivalent to t2
  - To implement declaration equivalence, a new operation getBaseTypeName must be provided by the symbol table
    - It fetches the base type name rather than the associated type expression

Type Equivalence

- Declaration equivalence
  - Pascal:
    - Uniformly uses declaration equivalence
  - C:
    - Uses declaration equivalence for structures and unions
    - Uses structural equivalence for pointers and arrays