6.4 Data types and type checking

- Type inference
- type checking

  the principal tasks of a compiler

6.4 Data types and type checking

- Type checking = set of rules that ensure the type consistency of different constructs in the program
- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -, /) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has boolean type

6.4 Data types and type checking

- **More examples**:
  - For each assignment statement, the type of the updated variable must match the type of the expression being assigned

    match <-> equivalent

  - For each call statement foo(v1, …, vn), the type of each actual argument vi must match the type of the corresponding formal argument fi from the declaration of function foo
  - The type of the return value must match the return type from the declaration of the function

6.4.1 Type expressions and type constructors

- data type forms:

  a set of values with certain operations on those values.

- type information can be explicit and implicit.

  For instance

  var x: array[1..10] of real (explicit)

  const greeting = “Hello”   (implicitly array [1..6] of char)

6.4.1 Type expressions and type constructors

simple types

- such as int ,double, boolean, char.

atomic

- the values exhibit no explicit internal structure, and the typical representation is also simple and predefined.
- void: has no value , represent the empty set.
- new simply type defined such as subrange types and enumerated types.

<u>6.4.1 Type expressions and type constructors</u>

Structured type

- New data types can be created using <u>type constructors</u>.
- Such constructors can be viewed as functions :

  take existing types as parameters .

  return new types with a structure that depends on the constructor.
- Array: Type parameter: There's actually 2:

  <u>index type</u>

  <u>component type.</u>

6.4.1 Type expressions and type constructors

Array

- Arrays are commonly allocated contiguous storage from smaller to larger indexes.
- allow for the use of automatic offset calculations during execution.
- The amount of memory needed is n * size.

6.4.1 Type expressions and type constructors

record

- a record or structure type constructor takes a list of names and associated types and constructs a new type.

```
struct
{double r;
 int i;}
```

- different types may be combined .
- the names are used to access the different components.

6.4.1 Type expressions and type constructors

union

- correspond to the set union operation

```
union
{double r;
 int i;}
```

- disjoint union, each value is viewed as either a real or an integer, but never both.
- Allocate memory in parallel for each component.

6.4.1 Type expressions and type constructors

pointer

- values that are references to values of another type. Most useful in describling recursive types.
- A value of a pointer type is a memory address whose location holds a value of its base type.

  ^integer

  *integer

- allocated space based on the address size of the target machine.

6.4.1 Type expressions and type constructors

function

- an array can be viewed as a function from its index set to its component set.
- Many language have a more general ability to describe function types.
- The allocated space depend on the address size of the target machine. According to the language and the organization of the runtime environment, it should allocate for :

  A code pointer alone

  Environment pointer.

6.4.1 Type expressions and type constructors

class

- similar to a record declaration, except it includes the definition of operations (methods or member functions)
- beyond type system such as inheritance and dynamic binding, must be maintained by separate data structures.

6.4.2 Type names, type declarations and recursive type

- type declarations(type definition): mechanism for a programmer to assign names to type expressions.
- Such as : typedef, = , associated directly with a struct or union constructor.

```
typedef struct
    {double r;
     int i;
    } RealIntRec;    (C)
```

6.4.2 Type names, type declarations and recursive type

- The C language has an additional type naming mechanism in which a name can be associated directly with a **struct** or **union** constructor. Without using a **typedef** directly.

```
struct RealIntRec
    {double r;
     int i;
    };    (C)
```

6.4.2 Type names, type declarations and recursive type

- type declarations cause the declared type names to be entered into the symbol table just as variable declarations.
- Usually the type names can't be reused as variable names.
- The C language has a small *exception* to this rule in that names associated to struct or union declarations can be reused as typedef names.

```
struct RealIntRec
{ double r;
int i;
};
typedef struct RealIntRec RealIntRec;
```

6.4.2 Type names, type declarations and recursive type

- Since type names can appear in type expressions, questions arise about the recursive use of type names.
- Such recursive data types are extremely important in modern programming languages include lists, trees, and many other structures.

6.4.2 Type names, type declarations and recursive type

Two general groups about language:

1、 permit the direct use of recursion in type declarations.

```
datatype intBST = Nil | Node of int*intBST*intBST
```
(ML)

2、 do not permit direct use of recursion in type declarations.

```
struct intBST
{ int val;
struct intBST  *left,  *right;
}
typedef struct intBST * intBST
```
(C)

6.4.3 Type equivalence

- type equivalence: two type expression represent the same type.
- There are many possible ways for type equivalence to be defined by a language.
- We represent type equivalence as it would be in a compiler semantic analyzer.

function *typeEqual (t1,t2:TypeExp): Boolean*;

6.4.3 Type equivalence

A simple grammar for type expressions:

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

structurd-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

6.4.3 Type equivalence

The type expression can be represented by a syntax tree .

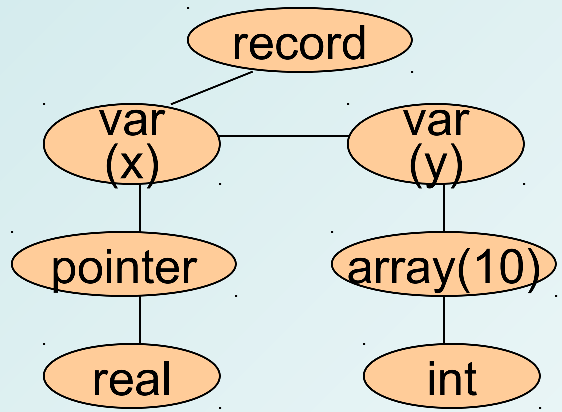
The type expression:

record

    x: pointer to real;

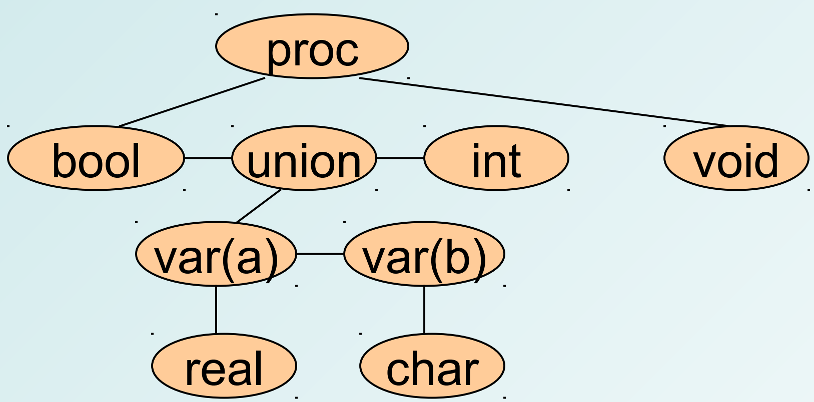
    y: array [10] of int

end



The type expression:

proc (bool, union a:real; b:char end, int): void



6.4.3 Type equivalence

Classification of type equivalence

1、 Structural equivalence
2、 Name equivalence
3、 Declaration equivalence

6.4.3 Type equivalence

Structural equivalence

- two types are the same if and only if they have the same structure.
- two types are the same if and only if they have syntax trees that are identical in structure.

```
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;
```

6.4.3 Type equivalence

Structural equivalence

```
        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
else if t1.kind = pointer and t2.kind = pointer then
     return typeEqual(t1.child1, t2.child1)
```

6.4.3 Type equivalence

Structural equivalence

```
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;
```

6.4.3 Type equivalence

- two arrays are equivalent: the same size and component type.
- two records are equivalent: the same components with the same names and in the same order.

different choices:

The size of the array can be ignored

The components of a structure or union can be in a different order.

6.4.3 Type equivalence

Name equivalence

- restricted variable declarations and type subexpressions to simple types and type names.

```
t1 = array [10]  of  int;
t2 = array [10]  of  int;
t3 = record
    x: t1;
    y: t2
end
```

- two type expressions are equivalent if and only if they are either the same simple type or are the same type name.

```
t1 = int;
t2 = int;
```

t1 and t2 are not equivalent.

6.4.3 Type equivalence

Name equivalence

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

6.4.3 Type equivalence

Name equivalence

```
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;
```

6.4.3 Type equivalence

Name equivalence

One complication in name equivalence:

type expressions can be allowed in variable declarations or subexpressions of type expressions.

a type expression may have no explicit name given to it, a compiler will have to generate an internal name for the type expression that is different from any other names.

x:array [10] of int;
y:array [10] of int;
The variable x and y are assigned different (and unique) type names corresponding to the type expression.

```
if t1 amd t2 are type names then
   return typeEqual( getTypeExp(t1),getTypeExp(t2))
```

6.4.3 Type equivalence

Declaration equivalence

- weaker version of name equivalence

  t2 = t1; are interpreted as establishing type aliases, rather than new types.

- Every type name is equivalent to some base type name, which is either a predefined type or is given by a type expression resulting from the application of a type constructor.

  t1 = array [10] of int;

  t2 = array [10] of int;

  t3 = t1;

type names t1 and t3 are equivalent under declaration equivalence, but neither is equivalent to t2.

6.4.3 Type equivalence

- Pascal uniformly uses declaration equivalence
- C uses declaration equivalence for structures and unions, but structural equivalence for pointers and arrays.
- A language will offer a choice of structural, declaration or name equivalence.

6.4.4 Type inference and type checking

program → var-decls; stmts

var-decls →var-decls; var-decl | var-decl

var-decl→id: type-exp

type-exp→int |bool|array [num] of type-exp

stmts→stmts; stmt|stmt

stmt → if exp then stmt | id :=exp

Table 6.10 (p.330)
Attributes grammar for type checking of this grammar

Grammar Rule	Semantic Rules
var-decl → **id** : *type-exp*	*insert*(**id** .*name*, *type-exp.type*)
type-exp → **int**	*type-exp.type* := *integer*
type-exp → **bool**	*type-exp.type* := *boolean*
$type\text{-}exp_1$ → **array** **[num]** **of** $type\text{-}exp_2$	$type\text{-}exp_1$.*type* := *makeTypeNode*(*array*, **num** .*size*, $type\text{-}exp_2$.*type*)
stmt → **if** *exp* **then** *stmt*	**if not** *typeEqual*(*exp.type*, *boolean*) **then** *type-error*(*stmt*)
stmt → **id** := *exp*	**if not** *typeEqual*(*lookup*(**id** .*name*), *exp.type*) **then** *type-error*(*stmt*)
exp_1 → exp_2 **+** exp_3	**if not** (*typeEqual*(exp_2 .*type*, *integer*) **and** *typeEqual*(exp_3 .*type*, *integer*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *integer*
exp_1 → exp_2 **or** exp_3	**if not** (*typeEqual*(exp_2 .*type*, *boolean*) **and** *typeEqual*(exp_3 .*type*, *boolean*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *boolean*
exp_1 → exp_2 **[** exp_3 **]**	**if** *isArrayType*(exp_2 .*type*) **and** *typeEqual*(exp_3.*type*, *integer*) **then** exp_1 .*type* := exp_2 .*type.child1* **else** *type-error*(exp_1)

6.4.4 Type inference and type checking

1. Declarations: cause the type of an identifier to be entered into the symbol table. Insert (id.name, type-exp.type);
2. Statements: substructures will need to be checked for type correctness.

   if not typeEqual(exp.type,boolean)

   then type-error(stmt)
3. Expression:

6.4.4 Type inference and type checking

- The behavior of such a type checker in the presence of errors:
  - the primary issues are when to generate an error message.
  - how to continue to check types in the presence of errors.

6.4.5 Additional topics in type checking

- Overloading: the same operator name is used for two different operations.

  procedure max(x,y: integer):integer;

  procedure max(x,y: real):real;

  In C and Pascal : illegal (redeclration)

  In Ada and C++ : legal

- type conversion and coercion

  allow arithmetic expressions of mixed type.

  There are two approaches a language can take to such conversions.

  Require the programmer supply a conversion function (Modula-2)

  The type checker supply the conversion automatically. (C)(coercion)

6.4.5 Additional topics in type checking

- Polymorphic typing

  Allow language constructs to have more than one type.

```
procedure swap (var x,y: anytype);

var x, y: integer;
      a, b:  char;
…..
swap(x,y);
swap(a,b);
swap(x,a);
```

A type checker must in every situation where swap is used determine an actual type that matches this type pattern or declare a type error. (involve sophisticated pattern matching techniques)