The first component of the phone system specification is the basic type declaration

[Person, Phone]

In the Miranda animation, we must select a representation for each of these abstract types. They are defined in Miranda (since initial upeer-case letters are reserved for another use) as the type synonyms

```
person == string
phone == string
```

The second component of the phone system specification is a **state schema**

PhoneDB

members: $\mathbb{P}$ Person
telephones: Person $\leftrightarrow$ Phone

---

dom telephones $\subseteq$ members

The schema name is PhoneDB, and appears in the “top border”. The body of the schema is divided into two parts. In the top part (above the line) there are two variables declared — 'members' denotes a subset of the abstract type Person, and 'telephones' is a binary relation relating values of type Person and Phone (i.e., a subset of S $\times$ T). These declarations restrict the type of values that are to be associated with a variable, but do not prescribe any specific value. The collection of potential values retained in a specification is referred to as its **state space**.

In the bottom part of the state schema (below the line), an **invariant** property of the state is given. An invariant asserts a condition on declared variable values that is always true. In this case, the telephones relation is only true for a Person who belongs to the set members (but need not be true for all such Persons).

In the Miranda animation, set and relation types are unavailable. However, lists with no duplicates provide a natural representation for sets, and a Boolean-valued function is effectively identical to a relation. Thus the state space in the Miranda animation is given as the type synonym declaration

```
    phonedb == ([person], [(person,phone)])
```

That is, states are constituted as 2-tuples where the first component is a list of person (the members set), and the second component is a list of ordered pairs (the telephones relation).

Then the invariant is implemented as the Boolean function using Miranda's list comprehension as

```
    invar :: phonedb -> bool
    invar (mem,tel) = and [member mem n | (n,a) <- tel]
```

That is, for mem :: [person] and tel :: [(person,phone)], each pair (n,a) is extracted from the relation tel using list comprehension, and the test for n in the set mem is performed to construct a list of bool values. Only if all these values are True is the invariant computed to be True. We have yet to see how the animation ensures that the invariant holds for every state taken on in a test run. We will see a little later that the function 'phdb' is a command interpreter, invoked to animate each operation, and it always verifies the invariant before proceeding.

An **operation schema** has two states associated with it — a **pre-state** (before the operation) and a **post-state** (after the operation). The pre-state is designated by referring to the state variables. The post-state is designated by adding the decoration **'** to the state variables. An operation schema may also involve **input** and **output** variables (arguments and return value, respectively). Last, but not least, an operation schema will prescribe **pre-conditions** and **post-conditions** for the operation.

```
AddEntry ______________________________________
| □PhoneDB
| name?: Person
| newnumber?: Phone
|______________________
| name?□members
| name? ↦ newnumber? □telephones
| telephones' = telephones □ {name? ↦ newnumber?}
| members' = members
|_______________________________________________
```

The □PhoneDB declaration imports all the variable declarations from the PhoneDB state schema, and foretells that the AddEntry operation will make a change in this state. The invariants of the named state schema are also imported and understood as being in conjunction to those in the conditions part of the schema.

In addition to the imported state variables, this operation schema declares two argument variables — the decoration '?' signifies such variables. The condition portion of this schema express pre-conditions on the state and input variables required for the correct application of the operation, written with the undecorated variable names. There are also post-conditions written using the variables decorated with **'**.