CS252:HACD Fundamentals of Relational Databases

Notes for Section 2: Values, Types, Variables, Operators

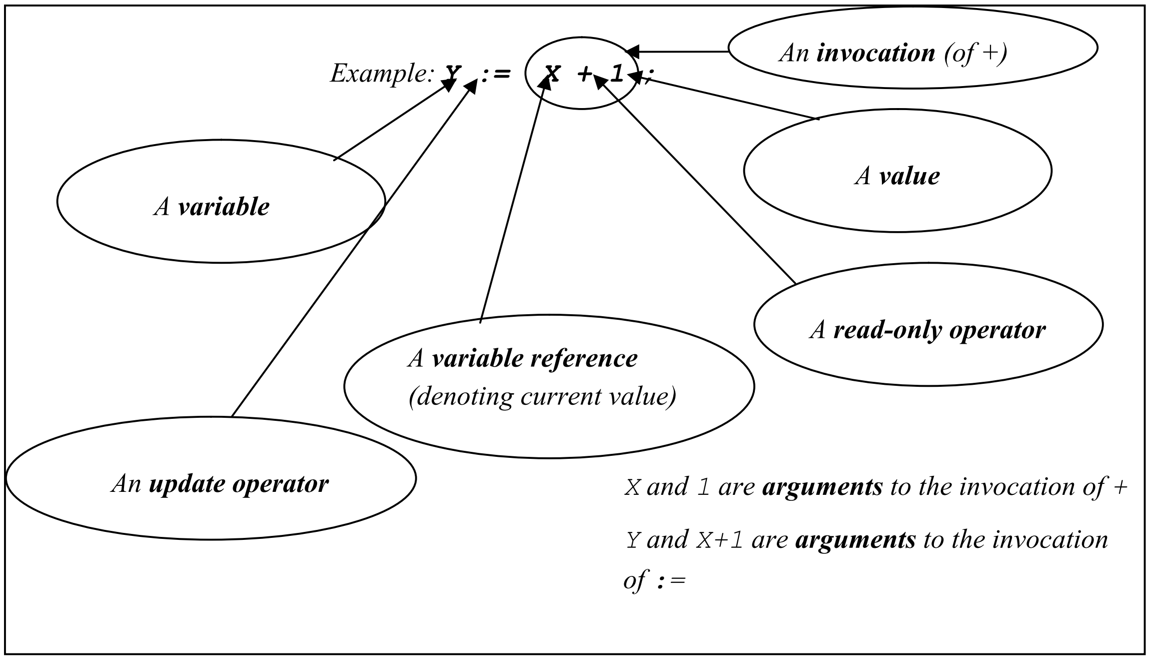
1. Cover slide

In this section we look at the four fundamental concepts on which most computer languages are based. We acquire some useful terminology to help us talk about these concepts in a precise way, and we begin to see how the concepts apply to relational database languages in particular. It is quite possible that you are already very familiar with these concepts—indeed, if you have done any computer programming they cannot be totally new to you—but I urge you to study the chapter carefully anyway, as not everybody uses exactly the same terminology (and not everybody is as careful about their use of terminology as we need to be in the present context). And in any case I also define some special terms, introduced by C.J. Date and myself in the 1990s, which have perhaps not yet achieved wide usage—for example, *selector* and *possrep*.

I wrote "most computer languages" because some languages dispense with variables. Database languages typically do not dispense with variables because it seems to be the very nature of what we call a database, that it varies over time in keeping with changes in the enterprise. Money changes hands, employees come and go, get salary rises, change jobs, and so on. A language that supports variables is said to be an imperative language (and one that does not is a functional language).

2. Anatomy of an Imperative

The slide shows a simple imperative—the assignment, `Y := X + 1`—annotated with the various terms we use for its components:


Example: Y := X + 1 ;
An invocation (of +)
A value
A variable
A read-only operator
A variable reference (denoting current value)
An update operator
X and 1 are arguments to the invocation of +
Y and X+1 are arguments to the invocation of :=

The update operator := is known as assignment. The imperative `Y := X + 1` is an (invocation of) assignment. Its effect is to evaluate the expression `X + 1`, yielding some numerical result *r* and to assign *r* to the variable `Y`. Subsequent references to `Y` therefore yield *r* (until some imperative is given to assign something else to `Y`).

Note the two operands of the assignment: `Y` is the *target*, `X+1` the *source*. The terms *target* and *source* here are names for the *parameters* of the operator. In the example, the argument `Y` is substituted for the parameter *target* and the argument `X+1` is substituted for the parameter *source*. We say that *target* is subject to update, meaning that any argument substituted for it must denote a variable. The other parameter, *source*, is not subject to update, so any argument substituted must denote a value, not a variable. `Y` denotes a variable and `X+1` denotes a value.

Now let's analyse the expression `X+1`. It is an invocation of the read-only operator `+`, which has two parameters, perhaps named *a* and *b*. Neither *a* nor *b* is subject to update. A read-only operator is one that has no parameter that is subject to update. Evaluation of an invocation of a read-only operator yields a value and updates nothing. The arguments to the invocation, in this example denoted by `X` and `1`, are also expressions denoting values. `1` is a *literal*, denoting the numerical value that it always denotes; `X` is a *variable reference*, denoting the value currently assigned to `X`.

A literal, in general, is any expression that denotes a value and does not contain any variable references.

3. Important Distinctions

Each of the distinctions mentioned on Slide 3 is illustrated in Slide 2, as follows:

- **Value versus variable:** `Y` denotes a variable, `X` denotes the current value assigned to the variable `X`. `1` denotes itself (a value).
- **Variable versus variable reference:** `Y` denotes itself, a variable; `X` is a variable reference, denoting its current value.
- **Update operator versus read-only operator:** **:=** (assignment) is an update operator; + (addition) is a read-only operator.
- **Operator versus invocation:** + is an operator; `X + 1` is an invocation of `+`.
- **Parameter versus argument:** The expressions `X` and `1` are arguments to the invocation of `+`; the operator + is defined to have two parameters. When an operator is invoked, an argument must be provided for each of its defined parameters. The term *argument* can refer to the value denoted by the expression as well as to the expression itself.
- **Parameter subject to update versus parameter not subject to update:** The first parameter of **:=** (the one representing the target) is subject to update and must therefore be substituted with a variable when **:=** is invoked; the second parameter of **:=** and both parameters of + are not subject to update and must be substituted, in invocations, by expressions denoting values.

4. A Closer Look at an Operator (+)

A read-only operator is what mathematicians call a *function*, and a function turns out to be just a special case of a relation! Because it is a relation, a function can be depicted in tabular form. The slide shows a picture of part of the function represented by the read-only operator `+`.

a	b	c
1	2	3
2	3	5
2	1	3

and so on (*ad infinitum*)

The relation depicted here represents the predicate $a + b = c$. The relation attributes a and b can be considered as the parameters of the operator +. Each tuple maps a pair of values substituted for a and b to the result of their addition, which is substituted for c. The relation is a function because each unique $<a,b>$ pair maps to exactly one c value—no two tuples with the same a value also have the same b value, so, given an a and a b, so to speak, we know the (only) resulting c.

Notice how the relational perception of an operator neutralises the distinction between arguments and result.

This relation could also represent the predicate $c - b = a$, or $c - a = b$.

You can imagine the invocation 1 + 2 as singling out the tuple with a=1 and b=2 (there is only one such tuple) and yielding the c value (3) in that tuple.

This relation is concerned only with numbers, its *domain of discourse*, some would say. Mathematicians, perceiving it as a **function** mapping pairs of numbers (a,b) to numbers (c), call the (a,b) number-pairs the *domain* of the function and numbers (c) its *range*. Computer scientists, perceiving it as an operator, say that its parameters a and b are of type number, as is the result, c (the type of the result is normally referred to as the type of the operator).

5. An Operator Definition

In computer languages we distinguish between operators that are defined as part of the language and operators that may be defined by uses of the language. Those defined as part of the language are called *built-in* operators and those defined by users are called *user-defined* operators.

The grammar for **Tutorial D** does not include a precise list of built-in operators. It mentions a few particular ones that have been devised for certain special purposes and adds "… plus the usual possibilities", leaving it to the implementation to decide what the usual possibilities are. In this book the matter of whether an operator used in my examples is built-in or user-defined is immaterial, except of course for those operators which an implementation is explicitly required to provide as built-in.

User-defined operator definition in **Tutorial D** is illustrated in Slide 5, which defines an operator named `HIGHER_OF` to give the value of whichever is the higher of two given integers:

```
OPERATOR HIGHER_OF ( A INTEGER, B INTEGER ) RETURNS INTEGER ;
IF A > B THEN RETURN A ;
          ELSE RETURN B ;
END IF ;
END OPERATOR ;
```

Explanation:

- **`OPERATOR HIGHER_OF`** announces that an operator is being defined and its name is `HIGHER_OF`.

- **`A INTEGER, B INTEGER`** specifies two parameters, named `A` and `B` and both of declared type `INTEGER`.
- **`RETURNS INTEGER`** specifies that the value resulting from every invocation of `HIGHER_OF` shall be of type `INTEGER` (which is thus the declared type of `HIGHER_OF`).
- **`IF … END IF ;`** is a single imperative (specifically, an `IF` statement) constituting the program code that implements `HIGHER_OF`. The programming language part of **Tutorial D**, intended for writing implementation code for operators and applications, is really beyond the scope of CS252, but if you are reasonably familiar with programming languages in general you should have no trouble understanding **Tutorial D**, which is deliberately both simple and conventional.

  The `IF` statement contains further imperatives within itself …
- **`IF A > B THEN RETURN A`** … such as `RETURN A` here, which is executed only when the given `IF` condition, `A > B`, evaluates to `TRUE` (i.e., is satisfied by the arguments substituted for `A` and `B` in an invocation of `HIGHER_OF`). The `RETURN` statement terminates the execution of an invocation and causes the result of evaluating the given expression, `A`, to be the result of the invocation.
- **`ELSE RETURN B`** specifies the imperative to be executed when the given `IF` condition is not satisfied.
- **`END IF`** marks the end of the `IF` statement.
- **`END OPERATOR`** marks the end of the program code and in fact the end of the operator definition.

Notes concerning *Rel*:

1. *Rel* provides as built-in all the **Tutorial D** operators used in CS252 except where explicitly stated to the contrary. (User-defined types didn't appear until 2009.)

2. *Rel* supports **Tutorial D** user-defined operators.

3. *Rel* additionally supports user-defined operators with program code written in Java™ (the language in which *Rel* itself is implemented), indicated by the key word `FOREIGN`. Examples of such operators are provided in the download package for *Rel*. Here are two of them (as provided at the time of writing in Version 3.12):

```
OPERATOR SUBSTRING(s CHAR, beginindex INTEGER, endindex
INTEGER) RETURNS CHAR Java FOREIGN
// Substring, 0 based
return new ValueCharacter(s.stringValue().substring(
                          (int)beginindex.longValue(),
                          (int)endindex.longValue()));
END OPERATOR;
```

```
OPERATOR SUBSTRING(s CHAR, index INTEGER) RETURNS CHAR
Java FOREIGN
// Substring, 0 based
return new ValueCharacter(s.stringValue().substring(
                          (int)index.longValue()));
END OPERATOR;
```

Notice that they are both named `SUBSTRING`, the first having three parameters, the second two. Thus, *Rel* can tell which one is being invoked according to the number of arguments to the invocation (and in fact according to the declared types of those arguments). The first, when invoked, yields the string starts at the given `beginindex` position within the given string `s`, and ends at the given `endindex` position, where 0 is the position of the first character in `s`. The second yields the string that starts at the given `index` position in `s` and ends at the end of `s`. Hence, `SUBSTRING('database',2,4) = 'tab'` and `SUBSTRING('database',4) = 'base'`.

I do not offer an explanation of the Java™ code used in these examples, that being beyond the scope of CS252.

6. What Is a Type?

A type is a *named* set of values. Much of the relational database literature, especially the earlier literature, uses the term *domain* for this concept, because that was the term E.F. Codd used. Nowadays we prefer *type* because that is the term most commonly used for the concept in computer science and it is not at all clear that Codd meant anything significantly different when he introduced *domain*. In fact, Codd's term was used to refer specifically to what we now call the *declared type* of an attribute of a relation.

For example, there might be a type named `WEEKDAY` whose values constitute the set `{ Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }`. For another example, type `INTEGER` is commonly available in computer languages, its values being all the integers in some range, such as $-(2^{32})$ to 2^{32}-1. Terms such as `Monday` and `-1` are *literals*.

`Monday` denotes a certain value of type `WEEKDAY` and `-7` denotes a certain value of type `INTEGER`. It is important that every value that can be operated on in a computer language can be denoted by some literal in that language.

In any computer language that supports types (as most of them do), some types are *built-in* (provided as part of the language). In some languages the only supported types are the built-in ones, but the trend in modern languages has been towards inclusion of support for *user-defined types* too.

Tutorial D includes comprehensive support for user-defined types, and so does the latest (2009) version of *Rel*. Previous releases of *Rel* support only the built-in types of **Tutorial D**. These are:

- `CHARACTER` or, synonymously, `CHAR`, for character strings.
- `INTEGER` or, synonymously, `INT`, for integers.
- `RATIONAL` for rational numbers, denoted by numeric literals that include a decimal point, such as `3.25, 1.0, 0.0, -7.935`.
- `TUPLE` types and `RELATION` types as described later in this lecture.

7. What is a Type For?

In general, a type is used for *constraining* the values that are permitted to be used for some purpose. In particular, for constraining:

- the values that can be assigned to a variable
- the values that can be substituted for a parameter
- the values that an operator can yield when invoked
- the values that can appear for a given attribute of a relation

In each of the above cases, the type used for the purpose in question is the *declared type* of the variable, parameter, operator, or attribute, respectively.

8. What is the Type of This?

Now, if every value is of some type, and a relation, as we have previously observed, is a value, then we need to understand to what type a given relation belongs, and we need a name for that type. Here again is our running example of a relation:

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

At this stage it is perhaps tempting to conclude that relations are all of the same type, which we might as well call `RELATION`, just as all integers are of type `INTEGER`. However, it turns out to be much more appropriate, as we shall see, to consider relations of the same heading to be of the same type and relations of different headings to be of different types. But all the types whose values are relations have that very fact—their values are relations—in common, and we call them *relation types*.

If relation types are distinguished by their headings, it is clear that a relation type name must include a specification of its heading. In **Tutorial D**, therefore, the type name for the relation under consideration can be written as

```
    RELATION { StudentId SID, Name NAME,  CourseId CID }
```

or, equivalently (for recall that there is no ordering to the elements of a set),

```
    RELATION { Name NAME, StudentId SID, CourseId CID }
```

—and this is a particular relation type.

This type name is written in **Tutorial D** notation. Braces, `{` and `}`, are used because the order of elements inside the braces is immaterial. In fact, `{ StudentId SID, Name NAME, CourseId CID }` denotes a set of attribute definitions, each consisting of an attribute name followed by a type name. (Note: the type of an attribute is sometimes called its *domain*, the term proposed by E.F. Codd in 1970.)

`RELATION { StudentId SID, Name NAME,  CourseId CID }` might in fact be the declared type of the relation variable `ENROLMENT`, and that variable might be part of the university's database.

Clearly, there is in theory an infinite number of relation types, because there is no limit to the degree of a relation. (Recall that the degree is the number of attributes.)

Here are some more relation types:

- `RELATION { StudentId SID, CourseId CID }`
- `RELATION { a INTEGER, b INTEGER, c INTEGER }`
- `RELATION { n INTEGER, w WEEKDAY }`
- `RELATION { }`

That last one looks a bit special! We'll investigate that one later.

The fact that we call them all relation types suggests that they have things in common about them, even though they are different types. We can see already, for example, that every relation type involves a set of attributes (the empty set in one particular case). Of particular interest are the *operators* they have in common, which are described in lectures HACD.4-6.

9. How to Write This as a Literal?

In Slide 6, "What Is a Type?", we noted the importance of being able to denote every value that can be operated on in a computer language by some literal in that language. We have also noted that a relation, such as our running example, is a value. What might a literal look like, that denotes that value? In other words, how are we to write literals denoting values of type `RELATION { StudentId SID, Name NAME, CourseId CID }`?

Of course we need a common notation for writing literals of any relation type.

10. A Relation Literal in Tutorial D

The slide shows a reasonable-looking attempt that actually isn't good enough:

```
    RELATION {
       TUPLE { StudentId S1, CourseId C1, Name Anne        },
       TUPLE { StudentId S1, CourseId C2, Name Anne      },
       TUPLE { StudentId S2, CourseId C1, Name Boris     },
       TUPLE { StudentId S3, CourseId C3, Name Cindy     },
       TUPLE { StudentId S4, CourseId C1, Name Devinder  }
    }
```

But of course it is not reasonable to expect a computer language to recognise symbols such as `S1,` `C1,` and `Boris`. We need a proper way of writing literals for those student identifiers, course identifiers, and names. And that is the matter we address next …

11. Literals for Student Ids, etc

Recall the declared types of the attributes: `SID, NAME, CID`. These are necessarily user-defined types, for it would not be reasonable to expect them to be built-in.

Suppose that values of type `SID` *are represented by* character strings (values of type `CHAR`). `CHAR` might well be a built-in type, and is indeed built-in in **Tutorial D** and *Rel*. Suppose further that character strings are denoted by text in quotes, like this: `'S1'`, as indeed they are in most computer languages. Then a literal for the student identifier S1 might be: `SID ( 'S1' )`. This literal is an invocation of the operator whose name happens to be the same as that of the type for student identifiers, `SID`. This operator has a single parameter whose declared type, `CHAR`, is that of the representations (character strings) chosen for student identifiers. In this invocation the `CHAR` literal `'S1'` appears in substitution for that parameter. The result of the invocation is not a character string but a value of type `SID`.

We call the operator `SID` a *selector,* because it can be used to "select" *any* value of type `SID`. Now, it is very likely that not every character string can validly represent a student identifier. Perhaps student identifiers must each be the letter S, followed by a maximum of four numeric digits. In that case we can expect the operator `SID`, when it is invoked, to check that the given string conforms to this rule—and raise an error if it doesn't. That is one good reason why type `SID` might be chosen in preference to type `CHAR` for student identifiers.

By the way, you can think of the literal `'S1'` as "selecting" a `CHAR` value. Every `CHAR` value can be denoted by a sequence of characters enclosed in quotes, and every sequence of characters enclosed in quotes does denote a `CHAR` value; so this syntax for literals does satisfy the requirements for being a selector.

Notes concerning *Rel*:

Rel does not (at the time of writing, September 2008) support user-defined types, so in ***Rel*** exercises we will use `CHAR` instead of types like `SID, NAME` and `CID`.*Rel* allows `CHAR` literals to be enclosed either in quotes, as already shown, or in double-quotes, like this: `"S1"`. Thus, `"S1"` and `'S1'` both denote the same `CHAR` value.

12. A Tuple Literal

In Slide 10 we tried to write a relation literal by specifying a collection of tuple literals, and we tried to write a tuple literal by specifying a collection of attribute values. Now that you know how to specify those attribute values properly, you can easily see that the correct way of writing the first of those tuple literals, arising from the foregoing discussion, is like this:

```
TUPLE { StudentId SID('S1'), CourseId CID('C1'),
        Name NAME('Anne')}
```

And here is the entire relation literal in **Tutorial D**:

```
RELATION {
    TUPLE { StudentId SID ('S1' ) , CourseId CID ( 'C1' ),
            Name NAME( 'Anne' ) },
    TUPLE { StudentId SID ( 'S1' ) , CourseId CID ( 'C2' ),
            Name NAME( 'Anne' ) },
    TUPLE { StudentId SID ( 'S2' ) , CourseId CID ( 'C1' ),
            Name NAME( 'Boris' ) },
    TUPLE { StudentId SID ( 'S3' ) , CourseId CID ( 'C3' ),
            Name NAME( 'Cindy' ) },
    TUPLE { StudentId SID ( 'S4' ) , CourseId CID ( 'C1' '),
            Name NAME( 'Devinder' ) }
    }
```

13. Types and Representations

Consider again the invocation `SID('S1')`, a literal of type `SID`. Recall that `SID` is an operator that, when invoked with a suitable character string, returns a value of type `SID`; also that every value of type `SID` can be denoted by some invocation of the operator `SID`. I have explained that we call such an operator a *selector* (for values of the type in question).

Note that the relation literal given on Slide 12 is an invocation of a certain *relation selector* – the specific selector for relations of that specific type. Similarly, a tuple literal is an invocation of a certain *tuple selector*.

The parameters of a selector correspond to components of what we call a possible representation, or *possrep* for short. (I will explain later why we use the word "possible" here.) So, certain "suitable" values of type `CHAR` can be considered to represent values of type `SID`. Which `CHAR` values in particular? Perhaps just those that consist of the upper-case letter S followed by numeric digits up to an agreed maximum length. The next slide shows how the possrep and the format rule can be put together to form a *type definition* for type `SID`.

14. A Type Definition for `SID`

Here is the complete definition, as expressed in **Tutorial D** with the use of operators `LENGTH`, `SUBSTRING`, and `IS_DIGITS`, defined in the script OperatorsChar.d provided by *Rel* in its directory named Scripts:

```
    TYPE SID POSSREP SID { C CHAR
                           CONSTRAINT LENGTH(C) <= 5
                                      AND
                                      SUBSTRING(C,0,1) = 'S'
                                      AND
                                      IS_DIGITS(SUBSTRING(C,1))
                         } ;
```

Recall that `POSSREP` is short for "possible representation". It means that the operators defined for type `SID` behave as if values of type `SID` were represented that way, regardless of how they are *physically* represented "under the covers". That is why we use the word "possible"—the values might possibly be represented internally that way (but they don't have to be and we don't even know if they are).

Note how the definition of a (user-defined) type depends on the existence of the types used as declared types for the components (in this case just one component) of the possrep. That's why the DBMS has to provide some built-in types.

Reminder: Versions of *Rel* prior to 2009 did not support `TYPE` statements.

15. Type Constraint for `SID`

This slide shows the `CONSTRAINT` part of the type definition:

```
CONSTRAINT LENGTH(C) <= 5
           AND
           SUBSTRING(C,0,1) = 'S'
           AND
           IS_DIGITS(SUBSTRING(C,1))
```

Explanation:

- **`CONSTRAINT`** announces that the expression following it (up to but excluding the closing brace) is a condition that must be satisfied by all possrep values that do indeed represented values of type `SID`. Note that the expression itself uses the logical connective `AND`, with its usual meaning, to connect three expressions, two of which are *comparisons* and each of which is a *truth-valued* expression—one that, when evaluated, yields either `TRUE` or `FALSE`.
- **`LENGTH(C) <= 5`** expresses a rule to the effect that the total length of a value for the `C` possrep component must never exceed 5. Here I assume the existence of the operator, `LENGTH`, for the purpose at hand. The definition of **Tutorial D** does not include all of the operators that an implementation might provide as built-in. As already mentioned, *Rel* provides a script for defining this and other useful operators on character strings, in OperatorsChar.d.
- **`SUBSTRING(C,0,1)`** denotes the string consisting of the leftmost character of the value of the `C` possrep component. `SUBSTRING` is defined in OperatorsChar.d. Note that it treats strings as starting at position 0, not 1.
- **`SUBSTRING(C,1)`** uses the other `SUBSTRING` operator defined in OperatorsChar.d and denotes the string consisting of the whole of the value of the `C` possrep component apart from the first character. This is given as the argument to an invocation of `IS_DIGITS`, also provided in OperatorsChar.d. `IS_DIGITS` takes a string and yields `TRUE` if every character in the given string is a numeric digit, otherwise `FALSE`.

The combination of possrep and type constraint defines the entire set of values that constitute the type.

16. Defining a Subtype

As the slide says, the subject of this slide is really beyond the scope of CS252, but you are welcome to use subtyping in *Rel* if you would like to try it out—in the coursework exercises, for example.

Tutorial D's subtyping is different from that found in typical object-oriented languages such as Java, because it uses subtyping by constraint—often known as *specialization by constraint*—rather than subtyping by *extension*.

17. What Is a Variable?

The example shown on the slide is a `VAR` statement:

```
VAR SN SID INIT SID ( 'S1' ) ;
```

Note that the declaration takes the form of an imperative that, when invoked, brings into existence ("creates") a variable.

Explanation:

- **`VAR SN`** announces that what follows defines a variable named `SN`.
- **`SID`**, immediately following `SN`, specifies the declared type of this variable, indicating that only values of type `SID` can be assigned to `SN`.
- **`INIT`** specifies that what follows is an expression whose value is to be immediately assigned to `SN`. You already know what `SID ( 'S1' )` denotes (see Slide 11). The value specified in an `INIT` clause is commonly called the *initial value* of the variable in question. It remains that value until it is replaced by subsequent invocation of an update operator such as assignment.

To answer the question posed in the slide's title, we can now see that a variable is something consisting of a name, a declared type, and a value. The name and declared type remain the same throughout the existence of the variable, but its value can change from time to time, which is of course why it is called a variable. Although the value can change from time to time, the value must always be a value of the declared type of the variable—in this example a value of type `SID`.

18. Updating a Variable

A value is assigned to a variable by invoking some *update operator*. The simplest and most general of such operators is the assignment operator itself—**:=** in **Tutorial D** (and some other languages). The example on the slide shows a simple invocation of assignment to "update" the variable `SN`:

The first example on the slide, the assignment `SN := SID ( 'S2' )`, simply assigns the student identifier S2 to `SN`. If that assignment is given some time after the declaration shown in Slide 16, then the effect will indeed be to change the value of the variable `SN`. It is changed from being the student identifier S1 to the student identifier S2. Here `SID ( 'S2' )` is the *source* for the assignment and `SN` is the *target*. The source does not have to be a literal, of course, as the second example on the slide demonstrates.

`SN := S# ( LEFT ( THE_C (SN), 1 ) || '5' )` assigns the student identifier S5 to `SN` (because the current value of `SN` must begin with S as required by the type constraint). Please note that details of the expression on the right-hand side of this assignment are utterly unimportant as far as CS252 is concerned. In case you need them, though:

> `||` is the "concatenation" operator that joins two strings together to form a single string;
>
> `THE_C(SN)` yields the character string that is the value of the component named `C` of the representation of the `SID` value of SN. (The equivalent in Java would be `SN.C.`)
>
> `LEFT ( s, n )` returns the string consisting of the leftmost *n* characters of the string *s*. The leftmost single character of `THE_C(SN)` is of course the letter S, which is to be concatenated with the string consisting of the numeric digit 5.

The next example on the slide is

```
    CALL SET_DIGITS ( SN , 23 ) ;
```

`SET_DIGITS ( SN , 23 )` is (we assume) equivalent to
`SN := S# ( LEFT ( THE_C (SN), 1 ) || '23' )`. Note that `SN` in the invocation of `SET_DIGITS` is an argument substituted for a parameter that is defined for update, so here stands for the variable itself and not for its current value.

The term *pseudovariable* comes from a once well known programming language called PL/I. It refers to an expression, other than a simple variable name, that is permitted to appear as a target of some update operator invocation (such as on the left-hand side of an assignment).

Tutorial D additionally allows certain special kinds of expression to appear as update targets. Such expressions are called *pseudovariables*—they are not real variables but they can be treated as if they were. A pseudovariable takes the form of an invocation of a read-only operator in which one of the operands is a reference to a variable (or pseudovariable) that is to be updated. The final example on the slide is an assignment to a pseudovariable:

```
    THE_C ( SN ) := 'S2' ;
```

In **Tutorial D**, `THE_C ( SN )` is a pseudovariable implied by the possrep definition, and `THE_C ( SN ) := 'S2'` is equivalent to `SN := SN ( 'S2' )`. A pseudovariable of the form `SUBSTR ( … )` is self-explanatory.

19. Important Distinctions Arising

I conclude this lecture by reminding you of the important distinctions I drew to your attention at its beginning. I repeat them here, with illustrative examples:

- values and variables

  A value such as the integer 3, the character string `'London'`, or the relation `RELATION { TUPLE { A 3, B 'London' } }` is something that exists independently of time or space and is not subject to change. A variable *is* subject to change, by invocation of assignment (other update operators are really shorthands for particular assignments).

- values and representations of values

  The character string value `'S1'` is a *possible representation* of the student identifier S1, a value of type `SID` denoted by `SID('S1')`.

- types and representations

  `POSSREP { C CHAR }` defines a possible representation for all values of type `SID`.

- read-only operators and update operators

  + is a *read-only operator* because, when it is invoked, it returns a value. := is an *update operator* because, when it is invoked, it has the effect of replacing the current value of a variable—and does *not* return a value.

- operators and invocations

  `SID` is an *operator*. Its full name (signature) is `SID(C CHAR)`. `SID( 'S1')` is an *invocation* of `SID`. Similarly, + is an operator, with full name such as `+(A RATIONAL, B RATIONAL)`, and `x+y` is an invocation of `+`.

- parameters and arguments

  `C CHAR` is a *parameter* (and in fact the only parameter) of the operator `SID`. `'S1'` is the argument substituted for `C CHAR` in the invocation `SID('S1')`. Similarly, `x` and `y` are the arguments substituted for `A RATIONAL` and `B RATIONAL` in the invocation `x+y`.

You can now test your understanding of these distinctions by carrying out the accompanying exercises (which also include some revision material for Lecture HACD.1).

20. EXERCISES

Complete sentences 1-10 below, choosing your fillings from the following:

`=`, `:=`, `::=`, argument, arguments, body, bodies, `BOOLEAN`, cardinality, `CHAR`, `CID`, degree, denoted, false, heading, headings, `INTEGER`, list, lists, literal, literals, operator, operators, parameter, parameters, read-only, set, sets, `SID`, true, type, types, update, variable, variables.

In 1-5, consider the expression `X = 1 OR Y = 2`.

1. In the given expression, = and `OR` are __________ whereas `X` and `Y` are __________.
2. `X` and `1` are _________ to an invocation of __________.
3. The value _______ by the given expression is of _____ `BOOLEAN`.
4. `1` and `2` are both _______ of _______ `INTEGER`.
5. The operators used in the given expression are ________ operators.

In 6-10, consider the expression `RELATION { X SID, Y CID } { }`.

6. It denotes a relation whose _______ is zero and whose _______ is two.
7. It is a relation _______.
8. The declared type of `Y` is _______.
9. In general, the heading of a relation is a possibly empty _______ of attributes and its body is a possibly empty _______ of tuples.
10. It is _____ that the assignment `RV` __ `RELATION { X SID, Y CID } { }` is legal if the _______ of `RV` is `{ Y CID, X SID }`, ______ that it is legal if the _______ of `RV` is `{ A SID, B CID }`, ______ that it is legal if the _______ of `RV` is `{ X CID, Y SID }`, and ______ that it is legal if the _______ of `RV` is `{ X CHAR, Y CHAR }`.

End of Notes