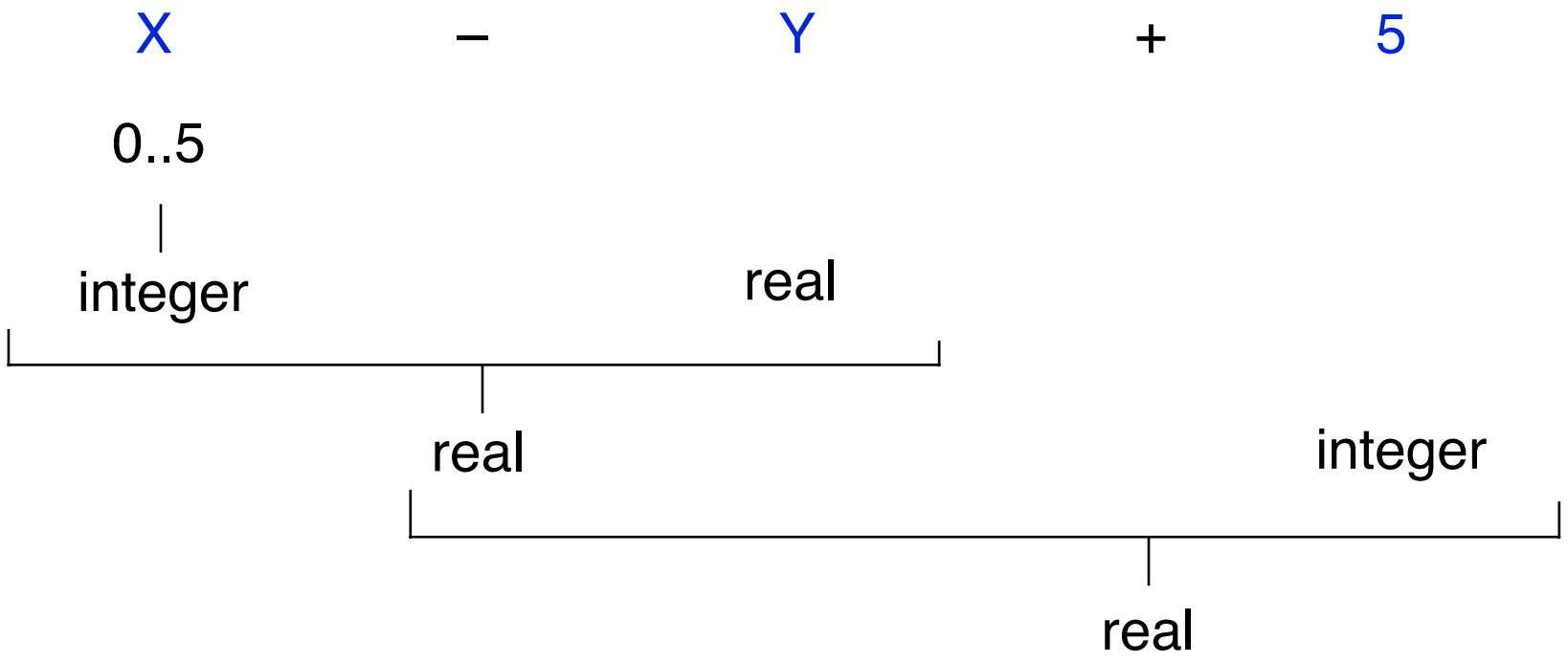
The Type Stack

- The *Type Stack* closely parallels both the run-time Expression Stack and the Symbol Stack
- Like the Symbol Stack, it is used in two ways:
  - to calculate the effective type of an expression, and
  - to accumulate type information in a declaration
- When used in processing expressions, it simulates the Expression Stack, computing effective result types instead of values - this parallels the Symbol Stack's computation of effective access attributes
- When used during a declaration, it parallels the Symbol Stack's accumulation of declaration attributes, building up type information for declared symbols as declarations are processed, which is stored in the Symbol Table when the declaration is completed

Type Stack – Expression Example

X - Y + 5

where X is of type subrange 0 .. 5
Y is of type **real**
5 is an integer literal constant



Type Stack

- The Type Stack simulates the contents of the Expression Stack in our abstract machine, but instead of computing the value of the results, we compute the types

Type Stack

- In our example, one of the operations on the stacks was to *promote* a subtype (0 .. 5) to its parent type (integer) - in Pascal, the subtype concept is limited to subranges

- More generally, this concept applies to static OO languages, in which inheritance is handled in the same way

- In that case, the type on the Type Stack is the *class* of the variable reference - if the operation on that reference is not a member of that class, then the type will be iteratively replaced by the superclass of the class on the stack until the operation is found, or the root of the class tree is reached (dynamic and virtual dispatching are handled at run time)

- The Type Stack also has other uses - in Java, it is used to verify the integrity of the byte code before it is executed by simulating execution of the byte code to check type conformance

Type Stack - Declarations

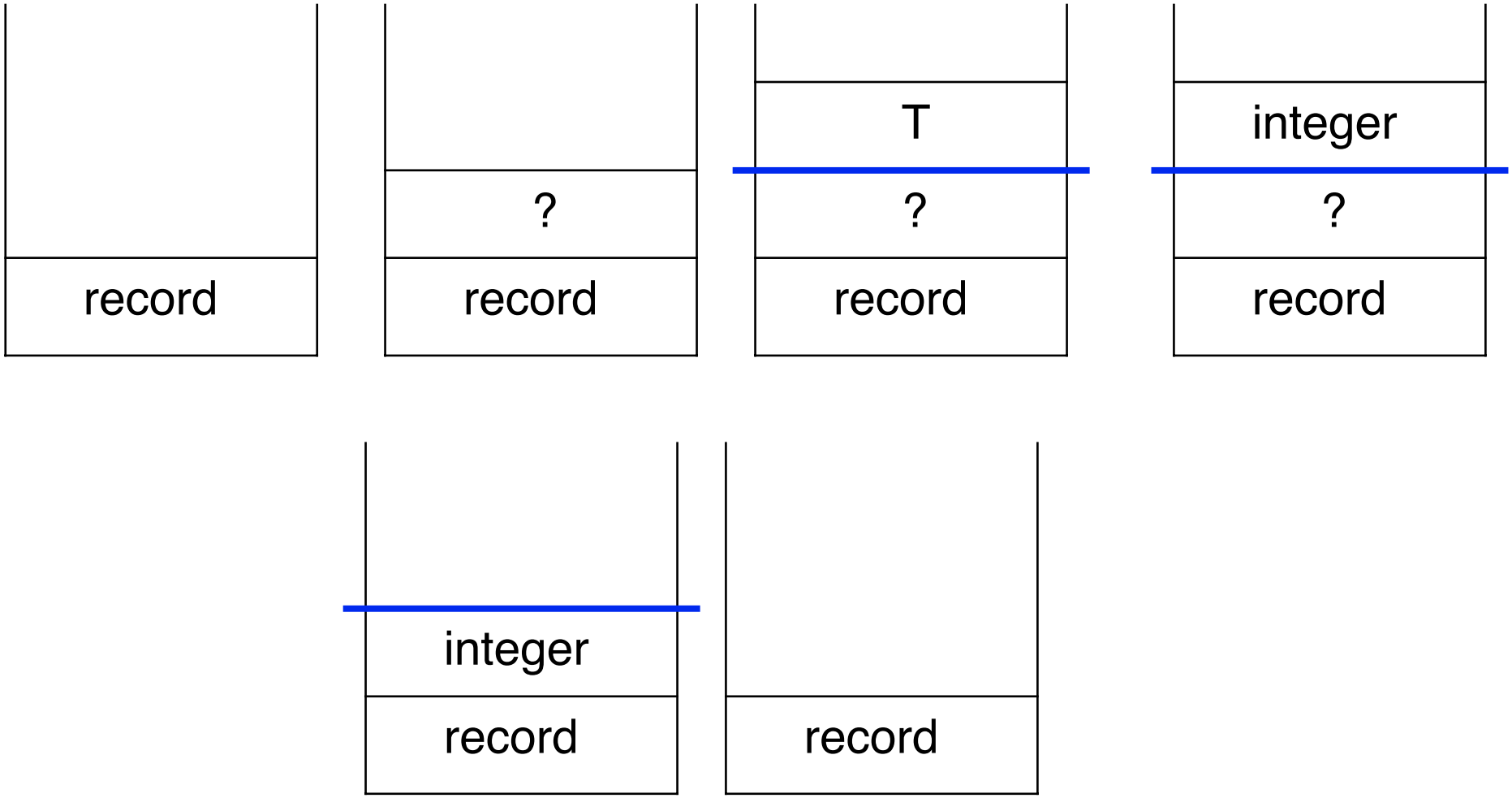
- Like the Symbol Stack, the Type Stack is used to accumulate the type information that will be entered into the Type Table for the declared symbol when the declaration is complete

- Again, the Type Stack conceptually has two parts
  - the *Type Declaration Stack* (TDS), and
  - the *Type Expression Stack* (TES)

- As was the case for the Symbol Stack, these two uses do not interfere because the TES is provably empty whenever the TDS must be pushed or popped

Type Stack - Declaration Example

```
type T = integer
var R : record
            var X : T
        end record
```



Semantic Mechanisms

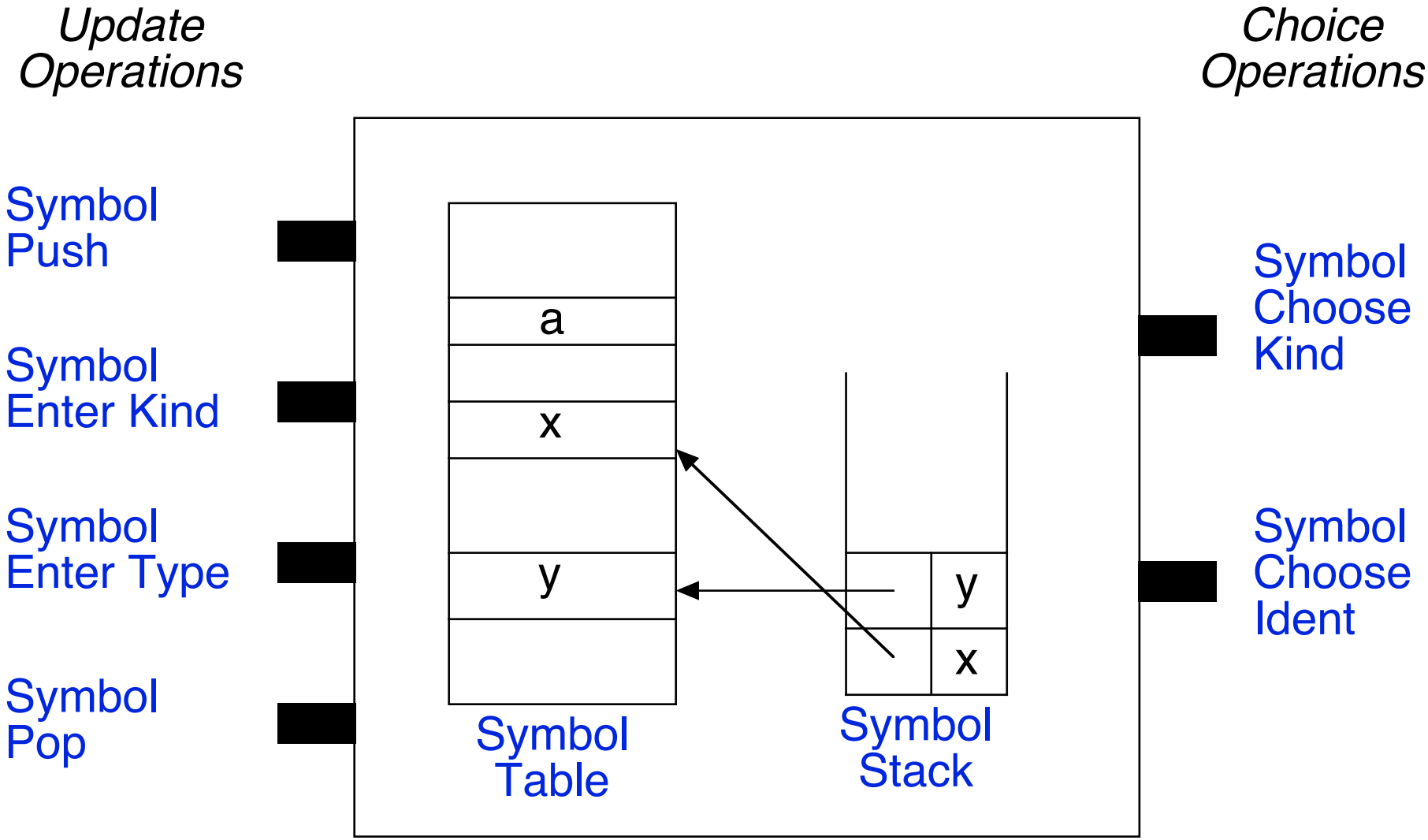
- We now look at how these ideas are implemented in S/SL
- *Semantic mechanisms* are the means by which S/SL keeps track of semantic information and performs actions
- Semantic mechanisms form an *abstract data type* by combining a data structure and the operations on it into a single entity - external access to the details of the data is prohibited, and only the operations may be used (much like a static class in OO)
- Outside of the mechanism (i.e. from the S/SL program or other mechanisms) the data structures are *opaque*
- The choice as to which data structures are combined into a single mechanism is an implementation decision - in PT, the Symbol Table and the Symbol Stack are separate mechanisms, whereas in the Euclid compiler they are combined into a single mechanism

Semantic Mechanisms

- Semantic mechanisms isolate the low-level manipulation of data structures from the high level algorithms that use them (the S/SL program)
- Since the high level algorithm is in a dataless language, the interface between the two is necessarily thin and well behaved
- In one sense, semantic mechanisms are simply a programming discipline that enforces good engineering practice, where the high level algorithm is strictly separated from the low level data manipulation (as in a good OO program)
- Semantic mechanisms provide the only access to their associated data structures, allowing them to be implemented in any way that the implementor chooses
- In particular, PT is not an OO or modular language, thus the ADT concept which is evident at the S/SL level must be enforced manually at the implementation level in PT

Semantic Mechanisms - An Example

- Unified Symbol Mechanism (from the Euclid compiler)



Symbol Mechanism

Semantic Mechanisms - Semantic Operations

- Recall that there are two kinds of semantic operations -
  - *update operations*, that change the mechanism state, and
  - *choice operations*, that return a value from the state

- Each semantic mechanism has one or more *designated elements* which are the implicit operand(s) of all semantic operations
  - in stack structured mechanisms, this is simply the top element

- Previous example:

  SymbolEnterKind –> sets the *kind* attribute of the top element of the stack (presently *y*)

  SymbolChooseKind –> returns the *kind* attribute of the top element of the stack (presently *y*)

Semantic Mechanisms - Conventions

- A number of semantic mechanism conventions have evolved with use of S/SL in many compilers - these help us to understand and maintain S/SL code

1. Operations and values associated with a semantic mechanism are always named using a naming convention that identifies the mechanism (e.g., *oSymbolEnterType*)

2. Update semantic operations change the state of at most one semantic mechanism (the one they are associated with)

   - For example, *oSymbolEnterType* may change the state of the Symbol mechanism, but it should not change the state of the Type mechanism

   - If more than one mechanism must be updated, then more than one operation should be invoked

Semantic Mechanisms - Conventions

3. Choice semantic operations do not change the state of the mechanism
   - For example, *oSymbolChooseType* should not change the state of the Symbol mechanism (or any other)

4. Implementations of mechanisms should not violate other mechanisms
   - For example, the implementation of the *oSymbolEnterType* operation should not read the current type directly from the Type Stack implementation - it should use an *auxiliary function* provided by the Type Stack mechanism to access the type information
   - Remember, PT is not an OO language, and OO discipline must be maintained by hand

```
AdditionType:
    [ oTypeChooseKind                   % of the right operand
       |tyInteger:
          oTypePop
          [ oTypeChooseKind             % of the left operand
              | tyInteger:              % result int, leave stack alone
              | tyReal:                 % result real, leave stack alone
              | *:
                 #eBadAdditionType
                 oTypePop                   % recovery
                 oTypePush(tyInteger)       % assume integer
          ]
       |tyReal:
          oTypePop
          [ oTypeChooseKind             % of the left operand
              | tyInteger:
                 oTypePop
                 oTypePush(tyReal)
              | tyReal:
              | *:
                 #eBadAdditionType
                 oTypePop               % recovery
                 oTypePush(tyReal)      % assume real
          ]
       | *:
          #eBadAdditionType
          oTypePop
          oTypePop
          oTypePush(tyInteger)
    ];
```

```
TypeDeclaration:
    sIdentifier
    oSymbolPushNew(syType)
    [
        |sInteger;
            oTypePush(tyInteger)
        | sIntegerLiteral:                          % subrange
            oValuePushLiteral                       % lower bound
            oTypePushNew(tySubrange)
            sIntegerLiteral                         % upper bound
            oValuePushLiteral
            oTypeEnterRangeValues                   % queries value stack
            oValuePop
            oValuePop
        | sidentifier:                              % named type
            oSymbolPush
            [oSymbolChooseKind
                |syType:
                    oTypePushSymbolType             % queries symbol stack
                |*:
                    #eTypeNameRequired
                    oTypePush(tyInteger)            % assume integer
            ]
            oSymbolPop
    ]
    oSymbolEnterType                        % get type from type stack
                                            % and assign to the symbol
    oTypePop
    oScopeEnterSymbol
    oSymbolPop;
```

Summary

- Simulation in Semantic Analysis, part 2: the Type Stack
  - We simulate execution like an ES, except compute types and attributes rather than values
  - The Symbol Stack computes attributes of symbol declarations and references, and the Type Stack computes types of declarations and expressions
  - Each is implemented by a semantic mechanism

- **Next** :
  - Tabulation