4.2 Definition module

Central data structure, stores properties of program entities
e. g. *type of a variable, element type of an array type*

A program entity is identified by the **key** of its entry in the data structure.

Operations:

NewKey ( )	yields a new key
ResetP (k, v)	sets the property P to have the value v for key k
SetP (k, v, d)	as ResetP; but the property is set to d if it has been set before
GetP (k, d)	yields the value of the Property P for the key k; yields the default-Wert d, if P has not been set

Operations are called as dependent computations in the tree

Implementation: a property list for every key, for example

Generation of the definition module: From specifications of the form

Property name : property type;

```
ElementNumber: int;
```

functions ResetElementNumber, SetElementNumber, GetElementNumber are generated.

Lecture Compiler I WS 2001/2002 / Slide 79

Objectives:

Properties of program entities

In the lecture:

- Explain the operations,
- explain the generator,
- give examples.

Suggested reading:

Kastens / Übersetzerbau, Section S. 130 unten

Assignments:

- Use the PDL tool of Eli

Questions:

- Give examples where calls of the operations are specified as computations in tree contexts. Describe how they depend on each other.

4.3 Type analysis

Task: Compute and check types of program entities and constructs at compile time

- **defined entities** (e. g. variables)
  have a type property, stored in the definition module

- **program constructs** (e. g. expressions)
  have a type attribute, associated to their symbol resp. tree node
  special task: **resolution of overloaded operators** (functions, methods)

- **types themselves are program entities**
  represented by keys;
  named using type definitions; **unnamed** in complex type notations

- **types have properties**
  e. g. the element type of an array type

- **type checking for program entities and for program constructs**
  a type must / may not have certain properties in certain contexts
  compare expected and given type; **type relations**: equal, compatible;
  compute type **coercion**

Lecture Compiler I WS 2001/2002 / Slide 80

Objectives:

Learn to categorize the tasks

In the lecture:

- Motivate type analysis tasks with typical properties of strongly typed languages;
- give examples

Suggested reading:

Kastens / Übersetzerbau, Section 6.1

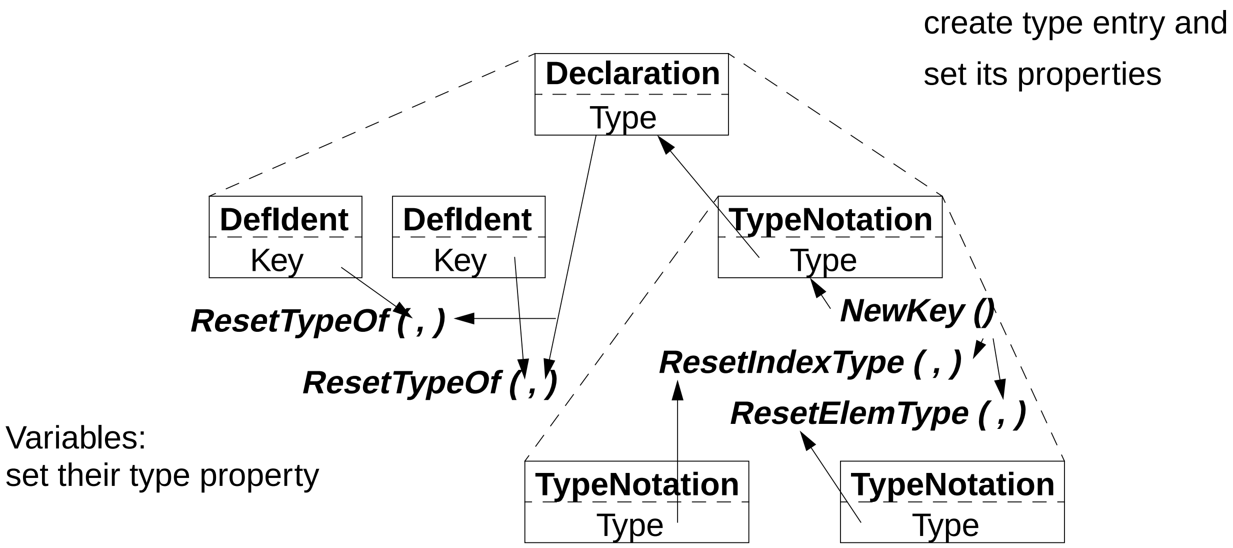
Questions:

- Give examples for program entities that have a type property and for others which don’t.
- Enumerate at least 5 properties of types in Java, C or Pascal.
- Give an example for a recursively defined type, and show its representation using keys.

Declarations and type notations

operations in the tree for the construct:

```
a, b: array [1..10] of real;
```



Lecture Compiler I WS 2001/2002 / Slide 81

Objectives:

Understand type analysis for declarations

In the lecture:

- Types as properties of program entities,
- types as attributes of program constructs,
- explain attributes and computations in the tree,
- explain the dependencies between the computations.

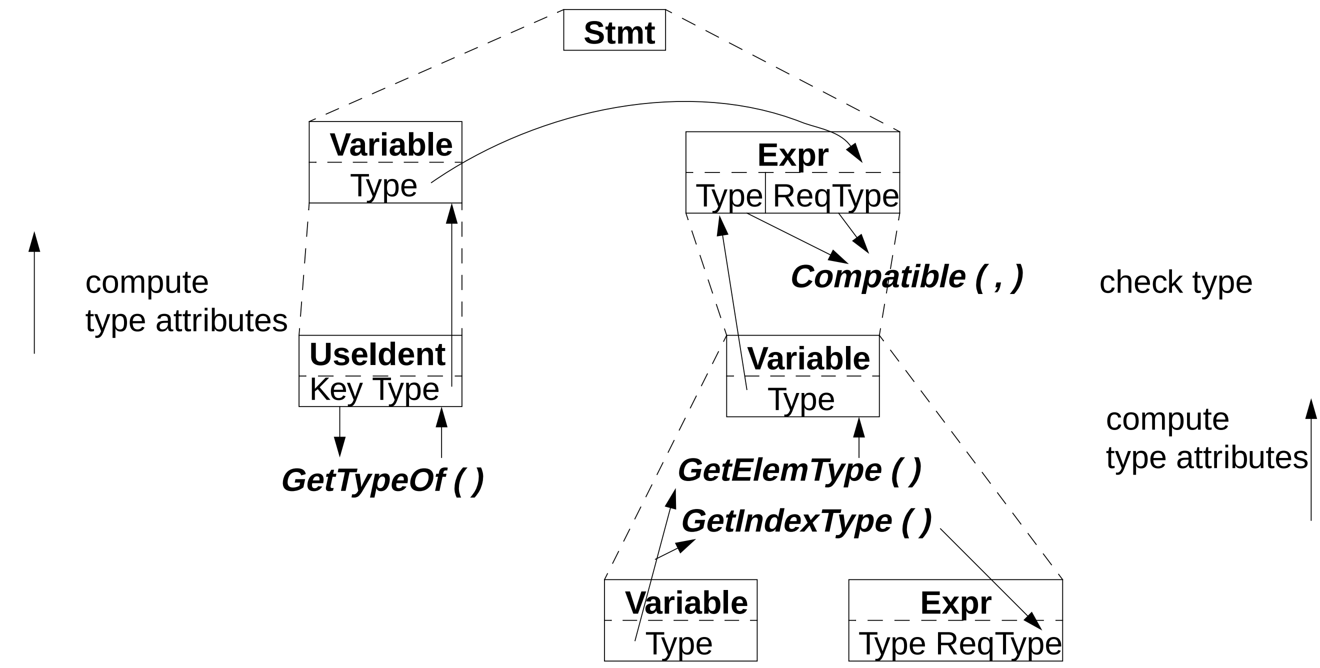
Suggested reading:

Kastens / Übersetzerbau, Section 6.1

Types of expressions required by context

operations in the tree for:

```
x := a[i];
```



Lecture Compiler I WS 2001/2002 / Slide 82

Objectives:

Example for computation and check of types

In the lecture:

- Types as properties of program entities,
- types as attributes of program constructs,
- explain attributes and computations in the tree,
- explain the dependencies between the computations.

Suggested reading:

Kastens / Übersetzerbau, Section 6.1

Assignments:

- Compose the trees of CI-81 and CI-82 into a complete tree. Find an evaluation order for the operations. State for each operation the weakest precondition with respect to the execution of other operations.

(see also Exercise 24)

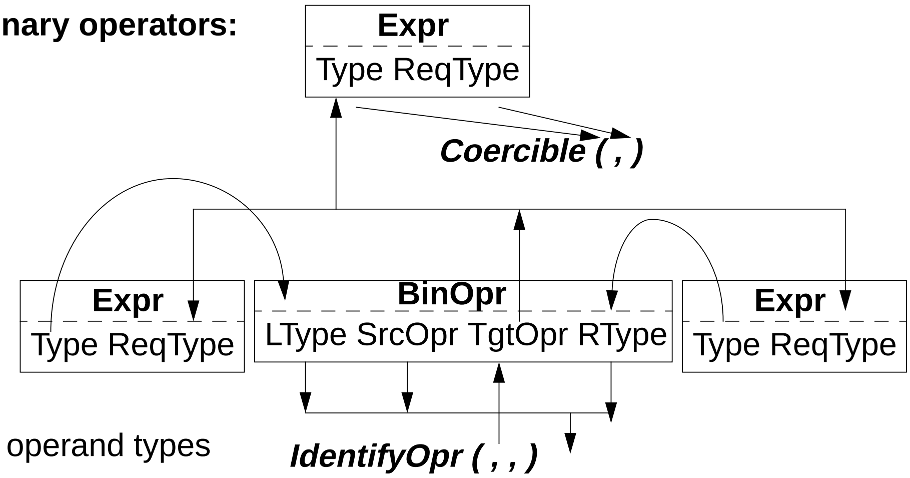
Overloading resolution for operators

Overloading: **same operator symbol** (source operator) is used for **several target operators** having **different signatures** and **different meanings**, e. g. specified by a table like:

symbol	signature	meaning
+	int ✕ int -> int	addition of integral numbers
+	real ✕ real -> real	floating point addition
+	set ✕ set -> set	union of sets
=	t ✕ t -> boolean	comparison for values of type t

Coercion: implicitly applicable type conversion: e. g. int -> real, char -> string, ...

Context of overloaded binary operators:



given: source operator and operand types
find: target operator

Lecture Compiler I WS 2001/2002 / Slide 83

Objectives:

Understand the task of overloading resolution

In the lecture:

Explain

- overloaded operators, functions, and methods,
- attribute computations,
- Eli tool OIL

Suggested reading:

Kastens / Übersetzerbau, Section 6.1

Assignments:

- overloading resolution as in C (Exercise 23)

Type analysis for object-oriented languages

Class hierarchy is a type hierarchy:

implicit type coercion: class -> super class
explicit type cast: class -> subclass

Variable of class type may contain
an object (reference) of its subclass

```
Circle k = new Circle (...);

GeometricShape f = k;

k = (Circle) f;
```

Check signature of overriding methods:

calls must be type safe; Java requires the same signature;
following weaker requirements are sufficient (*contra variant parameters*, language Sather):

call of dynamically bound method: `a = x.m (p);`

Variable: `X x; A a; P p; C c; B b;`

super class `class X { C m  (Q q) {` use of `q;... return c; } }`

subclass `class Y { B m  (R r) {` use of `r;... return b; } }`

Analyse dynamic methode binding; try to decide it statically:

static analysis tries to further restrict the run-time type:

```
GeometricShape f;...; f = new Circle(...);...; a = f.area();
```


Lecture Compiler I WS 2001/2002 / Slide 84

Objectives:

Understand classes as types

In the lecture:

Explain

- class hierarchy - type coercion
- type checking for dynamically bound methods calls
- predict the runtime classs of objects

Questions:

- Why would overridden methods not be type safe if they had "covariant" parameters (all 3 arrows between the classes X and Y would point up)? That is the situation in Eiffel.

Type analysis for functional languages (1)

Static typing and type checking without types in declarations

Type inference: Types of program entities are inferred from the context where they are used

Example in ML:

```
fun choice (cnt, fct) =
  if fct cnt then cnt else cnt - 1;
```

describe the types of entities using type variables:

```
cnt:    'a,
fct:    'b->'c,
choice: ('a * ('b->'c)) -> 'd
```

form equations that describe the uses of typed entities

```
'c  = bool
'b  = 'a
'd  = 'a
'a  = int
```

solve the system of equations:

```
choice: (int * (int->bool)) -> int
```


Lecture Compiler I WS 2001/2002 / Slide 85

Objectives:

Understand type inference

In the lecture:

Explain how types are computed from the operations without having typed declarations

Questions:

- How would type inference find type errors?

Type analysis for functional languages (2)

Parametrically polymorphic types: types having type parameters

Example in ML:

```
fun map (l, f) =
          if null l
          then nil
          else (f (hd l)) :: map (tl l, f)
```

polymorphic signature:

```
map: ('a list * ('a -> 'b)) -> 'b list
```

Type inference yields **most general type** of the function,
such that all uses of entities in operations are correct;

i. e. **as many unbound type parameters as possible**

calls with different concrete types, consistently substituted for the type parameter:

```
map([1,2,3], fn i => i*i)         'a = int, 'b = int
map([1,2,3], even)                'a = int, 'b = bool
map([1,2,3], fn i =(i,i))         'a = int, 'b = ('a*'a)
```


Lecture Compiler I WS 2001/2002 / Slide 86

Objectives:

Understand polymorphic types

In the lecture:

- Explain analysis with polymorphic types.
- Explain the difference of polymorphic types and generic types from the view of type analysis.