INF5110 – Compiler Construction

Types and type checking

Spring 2016

UNIVERSITAS OSLOENSIS
MDCCCXI

Outline

1. Types and type checking
    - Intro
    - Various types and their representation
    - Equality of types
    - Type checking

Outline

1. Types and type checking
    - Intro
    - Various types and their representation
    - Equality of types
    - Type checking

General remarks and overview

- Goal here:
    - what are *types*?
    - static vs. dynamic typing
    - how to describe types syntactically
    - how to represent and use types in a compiler
- coverage of various types
    - basic types (often predefined/built-in)
    - type constructors
    - values of a type and operators
    - representation at run-time
    - run-time tests and special problems (array, union, record, pointers)
- specification and implementation of type systems/type checkers
- advanced concepts

Why types?

- crucial user-visible *abstraction* describing program behavior.

- one view: type describes a set of (mostly related) *values*
- static typing: checking/enforcing a type discipline at compile time
- dynamic typing: same at run-time, mixtures possible
- completely untyped languages: very rare, types were part of PLs from the start.

Milner's dictum ("type safety")

Well-typed programs cannot go wrong!

- *strong* typing:[1] rigourously prevent "misuse" of data
- types useful for later phases and optimizations
- documentation and partial specification

---

[1] Terminology rather fuzzy, and perhaps changed a bit over time.

Types: in first approximation

Conceptually

- semantic view: A set of values *plus* a set of corresponding operations
- syntactiv view: notation to *construct* basic elements of the type (it's values) *plus* "procedures" operating on them
- compiler implementor's view: data of the same type have same underlying memory representation

further classification:

- built-in/predefined vs. user-defined types
- basic/base/elementary/primitive types vs. compound types
- type constructors: building more compex types from simpler ones
- reference vs. value types

Outline

1. Types and type checking
    - Intro
    - Various types and their representation
    - Equality of types
    - Type checking

Some typical base types

		base types	
`int`	0, 1, ...	$+, -, *, /$	integers
`real`	`5.05E4` ...	`+,-,*`	real numbers
`bool`	`true`, `false`	`and` `or` (|) ...	booleans
`char`	`'a'`		characters
⋮			

- often HW support for some of those (including many of the op's)
- mostly: elements of `int` are not exactly mathematical *integers*, same for `real`
- often variations offered: `int32`, `int64`
- often implicit *conversions* and relations between basic types
  - which the type system has to specify/check for legality
  - which the compiler has to implement

Some compound types

composed types		
`array[0..9] of real`		`a[i+1]`
`list`	`[]`, `[1;2;3]`	concat
`string`	`"text"`	concat ...
`struct` / `record`		`r.x`
...		

- mostly reference types
- when built in, special "easy syntax" (same for basic built-in types)
    - `4 + 5` as opposed to `plus(4,5)`
    - `a[6]` as opposed to `array_access(a, 6)` ...
- parser/lexer aware of built-in types/operators (special precedences, associativity etc)
- cf. functionality "built-in/predefined" via libraries

Abstract data types

- unit of *data* together with *functions/procedures/operations* . . . operating on them
- encapsulation + interface
- often: separation between exported and interal operations
  - for instance `public`, `private` . . .
  - or via separate interfaces
- (static) classes in Java: may be used/seen as ADTs, methods are then the “operations”

```
ADT  begin
    integer i;
    real x;
    int proc total(int a) {
       return i * x + a   // or: ``total = i * x + a''
    }
end
```

Type constructors: building new types

- array type
- record type (also known as struct-types
- union type
- pair/tuple type
- pointer type
  - explict as in C
  - implict distinction between reference and value types, hidden from programmer (e.g. Java)
- *signatures* (specifying methods/procedures/subroutines/functions) as type
- function type constructor, incl. higher-order types (in functional languages)
- (names of) classes and subclasses
- ...

Arrays

Array type

```
array [<indextype >] of <component type>
```

- elements (arrays) = (finite) functions from index-type to component type
- allowed index-types:
  - non-negative (unsigned) integers?, `from ... to ...`?
  - other types?: enumerated types, characters
- things to keep in mind:
  - indexing outside the array bounds?
  - are the array bounds (statically) known to the compiler?
  - *dynamic* arrays (extensible at run-time)?

One and more-dimensional arrays

- one-dimensional: effiencty implementable in standard hardware, (relative memory addressing, known offset)
- two or more dimensions

```
array [1..4] of array [1..3] of real
array [1..4, 1..3]  of real
```

- one can see it as "array of arrays" (Java), an array is typically a reference type
- conceptually "two-dimensional"
- *linear layout* in memory (dependent on the language)

Records ("structs")

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

- values: "labelled tuples" (`real`× `int`)
- constructing elements, e.g.
- access (read or update): *dot-notation* `x.i`
- implemenation: linear memory layout given by the (types of the) attributes
- attributes accessible by statically-fixed *offsets*
- fast access
- cf. objects as in Java

Tuple/product types

- $T_1 \times T_2$ (or in ascii `T_1 * T_2`)
- elements are *tuples*: for instance: `(1, "text")` is element of `int * string`
- generalization to n-tuples:

value	type
`(1, "text", true)`	`int * string * bool`
`(1, ("text", true))`	`int * (string * bool)`

- structs can be seen as "labeled tuples", resp. tuples as "anonymous structs"
- tuple types: common in functional languages,
- in C/Java-like languages: n-ary tuple types often only implicit as *input* types for procedures/methods (part of the "signature")

Union types (C-style again)

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

- related to *sum types* (outside C)
- (more or less) represents *disjoint union* of values of "participating" types
- access in C (confusingly enough): dot-notation `u.i`

Union types in C and type safety

- union types is C: bad example for (safe) type disciplines, as it's simply type-unsafe, basically an unsafe hack …
- the union type (in C):
  - nothing much more than directive to allocate enough memory to hold largest member of the union.
  - in the above example: `real` takes more space than `int`
- role of type here is more: implementor's (= low level) focus and memory allocation need, not "proper usage focus" or assuring strong typing

⇒ bad example of modern use of types

- better (type-safe) implementations known since

⇒ *variant record*, "tagged"/"discriminated" union) or even inductive data types[2]

-

---

[2] Basically: it's union types done right plus possibility of "recursion".

Variant records from Pascal

```
record case isReal: boolean of
  true:  (r:real);
  false: (i:integer);
```

- "variant record"
- non-overlapping memory layout[3]
- type-safety-wise: not really of an improvement
- programmer responsible to set and check the "discriminator" self

```
record case  boolean of
  true:  (r:real);
  false: (i:integer);
```

---

[3] Again, it's a implementor-centric, not user-centric view

Pointer types

- *pointer* type: notation in C: `int*`
- " ∗ ": can be seen as type constructor

```
int* p;
```

- random other languages: `^integer` in Pascal, `int ref` in ML
- value: *address* of (or reference/pointer to) values of the underlying type
- operations: *dereferencing* and determining the address of an data item (and C allows "pointer arithmetic")

```
var a:  ^integer
var b:   integer
...
a := &i   (* i an int var *)
          (* a := new integer ok too *)
b:= ^a + b
```

Implicit dereferencing

- many languages: more or less hide existence of pointers
- cf. reference types vs. value types often: automatic/implicit dereferencing

```
C r;                    //
C r = new C();
```

- "sloppy" speaking: " `r` is an object (which is an instance of class `C` /which is of type `C`)",
- slighly more recise: variable " `r` contains an object... "
- precise: variable " `r` will contain a reference to an object"
- `r.field` corresponds to something like " `(*r).field`, similar in Simula
- programming with pointers:
  - "popular" source of errors
  - test for non-null-ness often required
  - explicit pointers: can lead to problems in block-structured language (when handled non-expertly)
  - watch out for parameter passing
  - aliasing

Function variables

```
program Funcvar;
var pv : Procedure (x: integer);

  Procedure Q();
  var
    a : integer;
    Procedure P(i : integer);
    begin
      a:= a+i;          (* a def'ed outside          *)
    end;
  begin
    pv := @P;           (* ``return'' P,             *)
  end;                  (* "@" dependent on dialect *)
begin
  Q();
  pv(1);
end.
```

Function variables and nested scopes

- tricky part here: nested scope + function definition *escaping* surrounding function/scope.
- here: inner procedure "returned" via assignment to function variable[4]
- think about *stack discipline* of dynamic memory management?
- related also: functions allowed as return value?
  - Pascal: not directly possible (unless one "returns" them via function-typed reference variables like here)
  - C: possible, but *nested* function definitions not allowed
- combination of nested function definitions and functions as official return values (and arguments): *higher-order functions*
- Note: functions as arguments less problematic than as return values.

---

[4] Let's for the sake of the lecture, not distinguish conceptually between functions and procedures. But in Pascal, a procedure does not return a value, functions do.

Function signatures

- define the "header" (also "signature") of a function[5]
- in the discussion: we don't distinguish mostly: functions, procedures, methods, subroutines.
- functional type (independent of the name *f*): `int`→`int`

Modula-2

```
var f: procedure (integer): integer;
```

C

```
int (*f) (int)
```

- *values*: all functions ... with the given signature
- problems with block structure and free use of procedure variables.

---

[5] Actually, an identfier of the function is mentioned as well.

Escaping: function var's outside the block structure

```
 1  program Funcvar;
 2  var pv : Procedure (x: integer);
 3
 4      Procedure Q();
 5      var
 6          a : integer;
 7          Procedure P(i : integer);
 8          begin
 9              a:= a+i;          (* a def'ed outside          *)
10          end;
11      begin
12          pv := @P;             (* ''return'' P,             *)
13      end;                      (* "@" dependent on dialect *)
14  begin
15      Q();
16      pv(1);
17  end.
```

- at line 15: variable a no longer exists
- possible safe usage: only assign to such variables (here pv) a new value (= function) at the same blocklevel the variable is declared
- note: function *parameters* less problematic (stack-discipline

Classes and subclasses

Parent class

```
class A {
  int i;
  void f() {...}
}
```

Subclass B

```
class B extends A {
  int i
  void f() {...}
}
```

Subclass C

```
class C extends A {
  int i
  void f() {...}
}
```

- classes resemble records, and subclasses variant types, but additionally
  - local methods possble (besides fields)
  - subclasses
  - objects mostly created dynamically, *no* references into the stack
  - subtyping and polymorphism (subtype polymorphism): a reference typed by `A` can also point to `B` or `C` objects

- special problem: not really many, nil-pointer still possible

Access to object members: late binding

- notation `rA.i` or `rA.f()`
- dynamic binding, late-binding, virtual access, virtual access, dynamic dispatch . . . : all mean roughly the same
- central mechanism in almost all OO language, in connection with inheritance

Virtual access `rA.f()` (methods)

"deepest" `f` in the run-time class of the *object*, `rA` points to (independent from the *static* class type of `rA`.

- remember: "most-closely nested" access of variables in nested lexical block
- Java:
  - methods "in" objects are only dynamically bound
  - instance variables not, neither static methods "in" classes.

Example

```
public class Shadow {
    public static void main(String[] args){
        C2 c2 = new C2();
        c2.n();
    }
}

class C1 {
    String s = "C1";
    void m () {System.out.print(this.s);}
}


class C2 extends C1 {
    String s = "C2";
    void n () {this.m();}
}
```

Inductive types in ML and similar

- *type-safe* and powerful
- allows pattern matching

```
IsReal of real | IsInteger of int
```

- allows *recursive* definitions ⇒ inductive data types:

```
type int_bintree =
   Node of int * int_bintree * bintree
|   Nil
```

- `Node`, `Leaf`, `IsReal`: *constructors* (cf. languages like Java)
- constructors used as discriminators in “union” types

```
type exp =
     Plus of exp * exp
   | Minus of exp * exp
   | Number of int
   | Var of string
```

Recursive data types in C

does not work

```
struct intBST  {
  int val;
  int isNull;
  struct intBST left, right;
}
```

“indirect” recursion

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

In Java: references implicit

```
class BSTnode {
 int val;
 BSTnode left, right;
```

- note: implementation in ML: also uses pointers (but hidden from the user)
- no nil-pointers in ML (and `NIL` is not a nil-point, it's a cosntructor)

Outline

Example with interfaces

```
interface I1 { int m (int x) ; }
interface I2 { int m (int x); }
class C1 implements I1 {
    public int m(int y) {return y++;  }
}
class C2 implements I2 {
    public int m(int y) {return y++;  }
}

public class Noduck1 {
    public static void main(String[] arg) {
        I1 x1 = new C1();         // I2 not possible
        I2 x2 = new C2();
        x1 = x2;
    }
}
```

analogous effects when using classes in their roles as types

Structural vs. nominal equality

a, b

```
var a, b: record
   int i;
   double d
 end
```

c

```
var c: record
   int i;
   double d
 end
```

typedef

```
typedef idRecord: record
   int i;
   double d
 end
```

```
var d: idRecord;
var e: idRecord;;
```

what's possible?

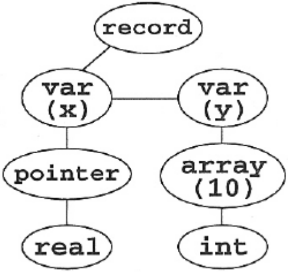
```
a := c;
a := d;

a := b;
d := a;
```

Types in the AST

- types are part of the syntax, as well
- represent: either in a separate symbol table, or part of the AST

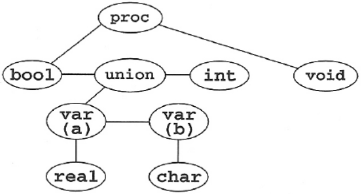
Record type

```
record
  x: pointer to real;
  y: array [10] of int
 end
```



procedure header

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



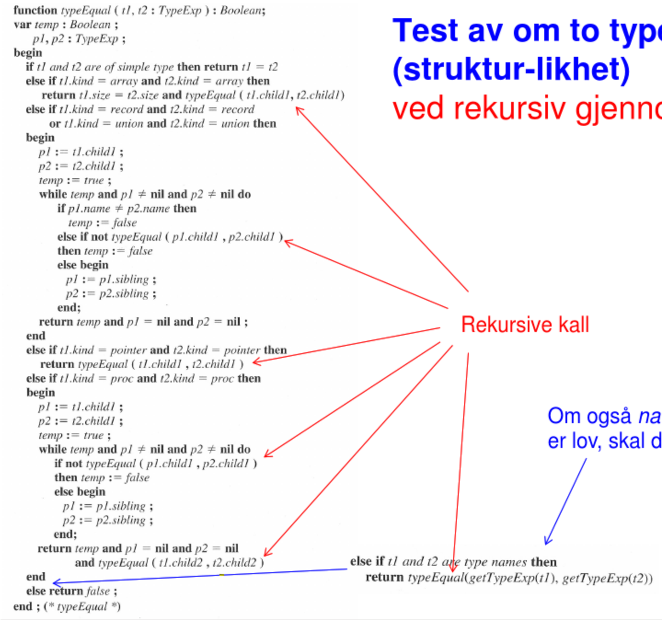
Structured types without names

$$
\begin{array}{rcl}
\textit{var-decls} & \rightarrow & \textit{var-decls}\,\mathbf{;}\,\textit{var-decl} \;\mid\; \textit{var-decl} \\
\textit{var-decl} & \rightarrow & \textbf{\textit{id}}\ \mathbf{:}\ \textit{type-exp} \\
\textit{type-exp} & \rightarrow & \textit{simple-type} \;\mid\; \textit{structured-type} \\
\textit{simple-type} & \rightarrow & \textbf{\textit{int}} \;\mid\; \textbf{\textit{bool}} \;\mid\; \textbf{\textit{real}} \;\mid\; \textbf{\textit{char}} \;\mid\; \textbf{\textit{void}} \\
\textit{structured-type} & \rightarrow & \textbf{\textit{array}}\ \mathbf{[}\ \textit{num}\ \mathbf{]}\ \textbf{\textit{of}}\ \textit{type-exp} \\
 & \mid & \textbf{\textit{record}}\ \textit{var-decls}\ \textbf{end} \\
 & \mid & \textbf{\textit{union}}\ \textit{var-decls}\ \textbf{end} \\
 & \mid & \textbf{\textit{pointerto}}\ \textit{type-exp} \\
 & \mid & \textbf{\textit{proc}}\ \mathbf{(}\ \textit{type-exps}\ \mathbf{)}\ \textit{type-exp} \\
\textit{type-exps} & \rightarrow & \textit{type-exps}\,\mathbf{,}\,\textit{type-exp} \;\mid\; \textit{type-exp}
\end{array}
$$

Structural equality

Test av om to typer er like (struktur-likhet)

ved rekursiv gjennomgang



```
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 ;
    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 )
  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 ; (* typeEqual *)
```

Rekursive kall

Om også *navnelikhet* er lov, skal dette med

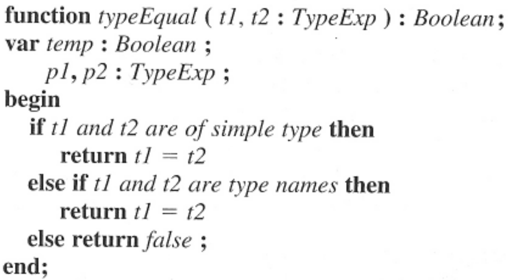
```
else if t1 and t2 are type names then
   return typeEqual(getTypeExp(t1), getTypeExp(t2))
```

Types with names

$$
\begin{array}{rcl}
\mathit{var\text{-}decls} & \rightarrow & \mathit{var\text{-}decls};\mathit{var\text{-}decl} \ \mid \ \mathit{var\text{-}decl} \\
\mathit{var\text{-}decl} & \rightarrow & \boldsymbol{id} : \mathit{simple\text{-}type\text{-}exp} \\
\mathit{type\text{-}decls} & \rightarrow & \mathit{type\text{-}decls};\mathit{type\text{-}decl} \ \mid \ \mathit{type\text{-}decl} \\
\mathit{type\text{-}decl} & \rightarrow & \boldsymbol{id} = \mathit{type\text{-}exp} \\
\mathit{type\text{-}exp} & \rightarrow & \mathit{simple\text{-}type\text{-}exp} \ \mid \ \mathit{structured\text{-}type} \\
\mathit{simple\text{-}type\text{-}exp} & \rightarrow & \mathit{simple\text{-}type} \ \mid \ \boldsymbol{id} \\
\mathit{simple\text{-}type} & \rightarrow & \boldsymbol{int} \ \mid \ \boldsymbol{bool} \ \mid \ \boldsymbol{real} \ \mid \ \boldsymbol{char} \ \mid \ \boldsymbol{void} \\
\mathit{structured\text{-}type} & \rightarrow & \boldsymbol{array}\ \mathbf{[}\ \mathit{num}\ \mathbf{]}\ \boldsymbol{of}\ \mathit{simple\text{-}type\text{-}exp} \\
 & \mid & \boldsymbol{record}\,\mathit{var\text{-}decls}\mathbf{end} \\
 & \mid & \boldsymbol{union}\,\mathit{var\text{-}decls}\mathbf{end} \\
 & \mid & \boldsymbol{pointerto}\,\mathit{simple\text{-}type\text{-}exp} \\
 & \mid & \boldsymbol{proc}\ \mathbf{(}\ \mathit{type\text{-}exps}\ \mathbf{)}\ \mathit{simple\text{-}type\text{-}exp} \\
\mathit{type\text{-}exps} & \rightarrow & \mathit{type\text{-}exps},\mathit{simple\text{-}type\text{-}exp} \ \mid \ \mathit{simple\text{-}type\text{-}exp}
\end{array}
$$

Name equality

- all types have "names", and two types are equal iff their names are equal
- type equality checking: obviously simpler
- of course: type names may have *scopes*. . . .



```
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;
```

Type aliases

- languages with type aliases (type synonyms): C, Pascal, ML ....
- often very convenient (`type Coordinate = float * float`)
- light-weight mechanism

type alias; make `t1` known also under name `t2`

```
t2  = t1    // t2 is the ''same type''.
```

- also here: different choices wrt. *type equality*

Alias if simple types

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

- often: `t1` and `t2` are the "same" type

Alias of structured types

```
t1 = array [10] of int;
t2 = array [10] of int;
t3 = t2
```

- mostly `t3` $\neq$ `t1` $\neq$ `t2`

Outline

1. Types and type checking
    - Intro
    - Various types and their representation
    - Equality of types
    - Type checking

Type checking of expressions (and statements)

- types of subexpressions must "fit" to the expected types the contructs can operate on[6]
- type checking: a *bottom-up* task

⇒ *synthesized* attributes, when using AGs

- Here: using an attribute grammar specification of the type checker
  - type checking conceptually done *while parsing* (as actions of the parser)
  - also common: type checker operates on the AST *after* the parser has done its job[7]
- type system vs. type checker
  - type system: specification of the rules governing the use of types in a language
  - type checker: algorithmic formulation of the type system (resp. implementation thereof)

---

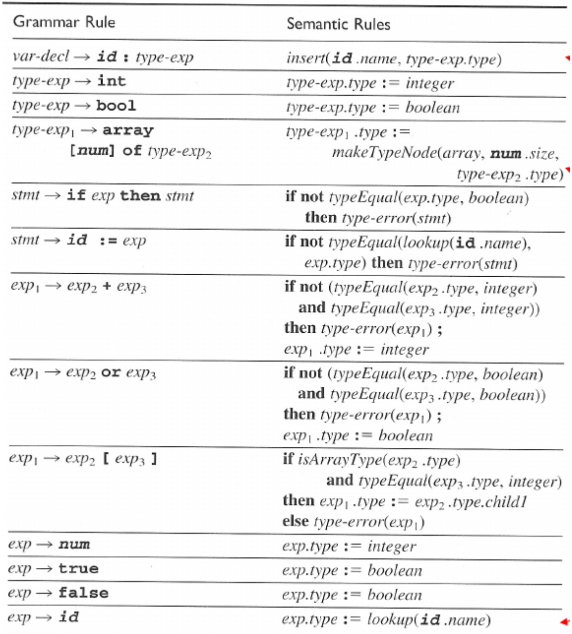
[6] In case (operator) overloading: that may complicate the picture slightly. Operators are selected depending on the type of the subexpressions.

[7] one can, however, use grammars as specification of that *abstract* syntax tree as well, i.e., as a "second" grammar besides the grammar for concrete parsing.

Grammar for statements and expressions

$$
\begin{array}{rcl}
\mathit{program} & \rightarrow & \mathit{var\text{-}decls}\,\mathbf{;}\,\mathit{stmts} \\
\mathit{var\text{-}decls} & \rightarrow & \mathit{var\text{-}decls}\,\mathbf{;}\,\mathit{var\text{-}decl} \;\mid\; \mathit{var\text{-}decl} \\
\mathit{var\text{-}decl} & \rightarrow & \boldsymbol{id}\;\mathbf{:}\;\mathit{type\text{-}exp} \\
\mathit{type\text{-}exp} & \rightarrow & \boldsymbol{int} \;\mid\; \boldsymbol{bool} \;\mid\; \boldsymbol{array}\;\mathbf{[}\,\mathit{num}\,\mathbf{]}\;\boldsymbol{of}\;\mathit{type\text{-}exp} \\
\mathit{stmts} & \rightarrow & \mathit{stmts}\,\mathbf{;}\,\mathit{stmt} \;\mid\; \mathit{stmt} \\
\mathit{stmt} & \rightarrow & \mathbf{if}\;\mathit{exp}\;\mathbf{then}\;\mathit{stmt} \;\mid\; \boldsymbol{id}\;\mathbf{:=}\;\mathit{exp} \\
\mathit{exp} & \rightarrow & \mathit{exp}\;\mathbf{+}\;\mathit{exp} \;\mid\; \mathit{exp}\;\boldsymbol{or}\;\mathit{exp} \;\mid\; \mathit{exp}\;\mathbf{[}\,\mathit{exp}\,\mathbf{]}
\end{array}
$$

Type checking as semantic rules



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)
exp → **num**	*exp.type* := *integer*
exp → **true**	*exp.type* := *boolean*
exp → **false**	*exp.type* := *boolean*
exp → **id**	*exp.type* := *lookup*(**id** .*name*)

Diverse notions

- *Overloading*
    - common for (at least) standard operations
    - also possible for user defined functions/methods . . .
    - disambiguation via (static) types of arguments
    - “ad-hoc” polymorphism
    - implementation:
        - put types of parameters as “part” of the name
        - look-up gives back a set of alternatives
- type-conversions: can be problematic in connection with overloading
- (generic) polymporphism
  `swap(var x,y:  anytype)`