Programming Languages and Compiler Design

Programming Language Semantics
Compiler Design Techniques

Yassine Lakhnech & Laurent Mounier
{lakhnech,mounier}@imag.fr
http://www-verimag.imag.fr/~lakhnech
http://www-verimag.imag.fr/~mounier.

Master of Sciences in Informatics at Grenoble (MoSIG)
Grenoble Universités
(Université Joseph Fourier, Grenoble INP)
Static Semantic Analysis
**Static semantic analysis**

**Input:** Abstract Syntax Tree (AST)

**Output:** enriched AST (with type information and/or type conversion indications)

Two main purposes:

- name identification: bind use-def occurrences
- type verification and/or type inference
Overview

1. Types in programming languages
2. How to formalize a type system?
3. Example 1: an imperative language
4. Example 2: a functional language
5. Some implementation issues...
What is a type?

- it defines the set of values an expression can take at run-time
- it defines the set of operations that can be applied to an identifier, and the resulting type
What are types useful for?

- **Pgm correctness**
  
  ```
  var x : kilometers ;
  var y : miles ;
  x := x + y ; -- typing error
  ```

- **Pgm readability**
  
  ```
  var e : energy := ... ; -- partition over the variables
  var m : mass := ... ;
  var v : speed := ... ;
  e := 0.5 * (m*v*v) ;
  ```

- **Pgm optimization**
  
  ```
  var x, y, z : integer ; -- and not real
  x := y + z ; -- integer operations are used
  ```
Typed and untyped languages

• **Typed languages:**
  → a dedicated type is associated to each identifier (and hence to each expression)

  examples: Java, Ada, C, Pascal, CAML, etc.
  Rk: strongly typed vs weakly typed languages . . .

• **Untyped languages:**
  → A single (universal) type is associated to each identifier (and hence to each expression)

  examples: Assembly language, shell-script, Lisp, etc.
Typed languages vs and safe languages

“Well-typed programs never go wrong . . .”

(Robin Milner)

Trapped errors vs untrapped errors

Safe language = untrapped errors are not possible

Using types in programming languages is a way to ensure safety but:

- it is not the only one (Lisp is considered safe)
- it is not sufficient (C is considered unsafe)
Types and type constructions

- Basic types: integers, boolean, characters, etc.
- Type constructions
  - cartesian product (structure)
  - disjoint union
  - arrays
  - functions
  - pointers
  - recursive types
  - ...

- But also:
  subtyping, polymorphism, overloading, inheritance, coercion, overriding, etc.

[see http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf]
Subtyping

Subtyping is a preorder relation $\leq_T$ between types.

It defines a notion of substitutability:

if $T_1 \leq_T T_2$, then elements of type $T_2$ may be replaced with elements of type $T_1$.

Examples:

• class inheritance in OO languages ;
• Integer $\leq_T$ Real (in several languages) ;
• Ada :

  type Month is Integer range 1..12 ;
  -- Month is a subtype of Integer
Type checking vs type inference

In a typed language, the set of “correct typing rules” is called a type system. The static semantic analysis phase uses this type system in two ways:

**Type checking**: check whether “type annotations” are used in a consistent way throughout the pgm

**Type inference**: compute a consistent type for each pgm fragments

**Rk**: in some languages (e.g., CAML, Haskel), there are no type annotations at all (all types are inferred).
**Static checking vs dynamic checking**

- **static checking**: verification performed at compile-time
- **dynamic checking**: verification performed at run-time

→ necessary to correctly handle:
  - dynamic binding for variables or procedures
  - polymorphism
  - array bounds
  - subtyping
  - etc.

⇒ For most programming languages, both kinds of checks are used . . .
How to formalize a type system? (1)

- “2 + 3 = 6” is well-typed
- “2 + true = false” is not well-typed
- “x = false” is well-typed if x is a (visible) boolean variable
- “2 + x = y” is well-typed if x and y are (visible) integer/real variables
- “let x = 3 in x + y” is well-typed if y is a (visible) integer/real variable

⇒ a term $t$ can be type-checked under assumptions on its free variables...
How to formalize a type system? (2)

- **Abstract syntax** describes terms (representing AST)
- **Environment** $\Gamma: Name \rightarrow Types$ (partial)
- **Judgement** $\Gamma \vdash t : \tau$
  
  In the environment $\Gamma$, the term $t$ is well-typed and has type $\tau$.
  
  (free variables of $t$ belong to the domain of $\Gamma$)
- **Type system**

  Inference rules
  
  $\frac{}{\Gamma_1 \vdash A_1 \quad \cdots \quad \Gamma_n \vdash A_n} \Gamma \vdash \Lambda$

  Axioms
  
  $\Gamma \vdash A$
Example: natural numbers

\[ a \ := \ n \ | \ x \ | \ a_1 + a_2 \]

Syntax

\[ \Gamma \vdash x : \text{Nat} \quad \text{if} \ \Gamma(x) = \text{Nat} \]

\[ x \text{ is of type } \text{Nat} \text{ in the environment } \Gamma \text{ if } \Gamma(x) = \text{Nat}. \]

\[ \Gamma \vdash n : \text{Nat} \]

The denotation \( n \) is of type \( \text{Nat} \)

\[ \Gamma \vdash a_1 : \text{Nat} \quad \Gamma \vdash a_2 : \text{Nat} \]

\[ \Gamma \vdash a_1 + a_2 : \text{Nat} \]

\( a_1 + a_2 \) is of type \( \text{Nat} \) assuming that \( a_1 \) and \( a_2 \) are of type \( \text{Nat} \).
Derivations in a type system

A type-check is a proof in the type system, i.e., a derivation tree where:

- leaves are axioms
- nodes are obtained by application of inference rules

A judgment is valid iff it is the root of a derivation tree

example:

\[
\emptyset \vdash 1 : \text{Nat} \quad \emptyset \vdash 2 : \text{Nat} \\
\emptyset \vdash 1 + 2 : \text{Nat}
\]

exo: prove that \([x \rightarrow \text{Nat}, y \rightarrow \text{Nat}] \vdash x + 2 : \text{Nat}\)
Type system for the while language
Syntax of the while language

Expressions

- same syntax for boolean and integer expressions ($a$)
- 3 kinds of (syntactically) distinct binary operators: arithmetic ($opa$), boolean ($opb$) and relational ($oprel$)

$$a ::= \text{true} | \text{false} | n | x | a \ opa \ a | a \ oprel \ a | a \ opb \ a$$

Statements

$$S ::= x := a | \text{skip} | S; S | \text{if } a \text{ then } S \text{ else } S | \text{while } a \text{ do } S$$
Judgments

• $\Gamma \vdash S$
  “in the environment $\Gamma$ the statement $S$ is well-typed”.

• $\Gamma \vdash a : t$
  “in the environment $\Gamma$ the expression $a$ is of type $t$.”
**Type system for expressions**

<table>
<thead>
<tr>
<th>bool. constant</th>
<th>int. constant</th>
<th>int opbin</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash \text{true} : \text{Bool} )</td>
<td>( \Gamma \vdash \text{n} : \text{Int} )</td>
<td>( \Gamma \vdash \text{a}_1 \text{ opa } \text{a}_2 : \text{Int} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>variables</th>
<th>bool. opbin</th>
<th>relational operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma(x) = t )</td>
<td>( \Gamma \vdash \text{b}_1 : \text{Bool} )</td>
<td>( \Gamma \vdash \text{a}_1 : t )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma \vdash \text{b}_1 : \text{Bool} )</td>
<td>( \Gamma \vdash \text{a}_2 : t )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma \vdash \text{b}_1 \text{ opb } \text{b}_2 : \text{Bool} )</td>
<td>( \Gamma \vdash \text{a}_1 \text{ oprel } \text{a}_2 : \text{Bool} )</td>
</tr>
</tbody>
</table>
Type system for statements

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Skip</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash a : t \quad \Gamma \vdash x : t )</td>
<td>( \Gamma \vdash x := a )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash S_1 \quad \Gamma \vdash S_2 )</td>
<td>( \Gamma \vdash a : \text{Bool} \quad \Gamma \vdash S )</td>
</tr>
</tbody>
</table>

Exercise: conditional statement?
Exercise

Extend the type system for the expressions assuming that arithmetic types can be now either integer (Int) or real (Real). Several solutions are possible:

- type conversions are never allowed
- only explicit conversions (with a cast operator) are allowed
- (implicit) conversions are allowed
Extension 1: Blocks

A new syntactic rule for the statements:

\[ S ::= \cdots \mid \text{begin } D_V \; ; \; S \; \text{end} \]

And for the declarations:

\[ D_V ::= \text{var } x := a \; ; \; D_V \mid \epsilon \]
Type system

Notations

• \(DV(D_v)\) denotes the set of variables declared in \(D_v\).

• \(\Gamma[y \mapsto \tau]\) denotes the environment \(\Gamma'\) such that:
  - \(\Gamma'(x) = \Gamma(x)\) if \(x \neq y\)
  - \(\Gamma'(y) = \tau\)

Judgments

• \(\Gamma \vdash DV | \Gamma_l\) means
  declarations \(DV\) update environment \(\Gamma\) into \(\Gamma_l\)

• \(\Gamma \vdash S\) means
  statement \(S\) is well-typed within environment \(\Gamma\)
Inference rule for Blocks

\[
\Gamma \vdash \begin{array}{c}
D_V \\
\mid
\end{array} \Gamma_l \quad \Gamma_l \vdash S
\]

\[
\Gamma \vdash \text{begin } D_V \text{ ; } S \text{ end}
\]
**Inference rules for declarations**

**Sequential evaluation**

\[
\frac{\Gamma \vdash a : t \quad \Gamma[x \mapsto t] \vdash D_V | \Gamma_l \quad x \not\in DV(D_V)}{\Gamma \vdash \text{var } x := a ; D_V | \Gamma_l[x \mapsto t]}
\]

**Collateral evaluation**

\[
\frac{\Gamma \vdash a : t \quad \Gamma \vdash D_V | \Gamma_l \quad x \not\in DV(D_V)}{\Gamma \vdash \text{var } x := a ; D_V | \Gamma_l[x \mapsto t]}
\]
Possible variations for variable declarations

• explicitly typed variables:
  \[ \text{var} \ x \ := \ e \ : \ t \]

• uninitialized variables:
  \[ \text{var} \ x \ := \ t \]

• untyped variables ?
  \[ \text{var} \ x \ := \ e \]

• uninitialized and untyped variables ???
  \[ \text{var} \ x \]
Extension 2: Procedures

Syntactic rule for the statements:

\[ S ::= \cdots \mid \text{begin } D_V \ ; \ D_P \ ; \ S \text{ end } \mid \text{call } p \]

and for the declarations:

\[ D_P ::= \text{proc } p \text{ is } S \ ; \ D_P \mid \epsilon \]

\( DP(D_P) \) denotes the set of procedures declared in \( D_P \)

Reminder: the semantics depends on the kind of binding (static vs dynamic) you consider . . .
Judgements

• procedure environment $\Gamma_P : Name \rightarrow \{proc\}$ (partial)

• $\Gamma_V \vdash D_V | \Gamma'_V$ means
  variable declarations $D_V$ update variable environment $\Gamma_V$ into $\Gamma'_V$

• $(\Gamma_V, \Gamma_P) \vdash D_P$ means
  procedure declarations $D_P$ is well-typed within variable and procedure environments $(\Gamma_V, \Gamma_P)$

• $(\Gamma_V, \Gamma_P) \vdash S$ means
  statement $S$ is well-typed within variable and procedure environments $(\Gamma_V, \Gamma_P)
**Static binding for proc. and var.**

**Block**

\[
\frac{\Gamma_V \vdash D_V \mid \Gamma'_V \quad (\Gamma'_V, \Gamma_P) \vdash D_P \quad (\Gamma'_V, \Gamma'_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \text{begin } D_V ; D_P ; S \text{ end}}
\]

**DP**

\[
(\Gamma_V, \Gamma_P) \vdash S \quad (\Gamma_V, \Gamma_P[p \mapsto \text{proc}]) \vdash D_P \quad p \notin DP(D_P)
\]

\[
(\Gamma_V, \Gamma_P) \vdash \text{proc } p \text{ is } S ; D_P
\]

**Call**

\[
(\Gamma_V, \Gamma_P) \vdash \text{call } p \quad \Gamma_P(p) = \text{proc}
\]

where \( \Gamma'_P = \text{upd}(\Gamma_P, D_P) \)

with:

\[
\text{upd}(\Gamma_P, \text{proc } p \text{ is } S ; D_P) = \text{upd}(\Gamma_P[p \mapsto \text{proc}], D_P)
\]

\[
\text{upd}(\Gamma_P, \varepsilon) = \Gamma_P
\]
Dynamic binding for proc. and var.

Block
\[ \frac{\Gamma_V \vdash D_V \mid \Gamma'_V \quad (\Gamma'_V, \Gamma'_P) \vdash S \quad \text{udef}(D_P)}{(\Gamma_V, \Gamma_P) \vdash \text{begin } D_V \mid D_P \mid S \text{ end}} \]

Call
\[ \frac{(\Gamma_V, \Gamma_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \text{call } p \Gamma_P(p) = S} \]

where \( \Gamma'_P = \text{upd}(\Gamma_P, D_P) \)

with:
\begin{align*}
\text{upd}(\Gamma_P, \textproc{proc } p \text{ is } S ; D_P) &= \text{upd}(\Gamma_P[p \mapsto S], D_P) \\
\text{upd}(\Gamma_P, \varepsilon) &= \Gamma_P \\
\text{udef}(\textproc{proc } p \text{ is } S ; D_P)) &= \text{udef}(D_P) \land p \notin DP(D_P) \\
\text{udef}(\varepsilon) &= \text{true}
\end{align*}

Remark:
procedure environment \( \Gamma_P : \text{Name} \rightarrow \text{Stm} \) (partial)
Procedures: static ; variables: dynamic

Block

\[
\begin{array}{c}
(\Gamma_V, D_V) \rightarrow \Gamma'_V \\
(\Gamma'_V, \Gamma_P) \vdash D_P \\
(\Gamma_V, \Gamma_P) \vdash \text{begin} \; D_V; \; D_P; \; S \; \text{end}
\end{array}
\]

Call

\[
(\Gamma_V, \Gamma'_P) \vdash S \\
(\Gamma_V, \Gamma_P) \vdash \text{call } p \Gamma_P(p) = (\Gamma'_P, S)
\]

where \( \Gamma'_P = \text{upd}(\Gamma_P, D_P) \)

with:

\[
\text{upd}(\Gamma_P, \text{proc } p \text{ is } S; D_P) = \text{upd}(\Gamma_P[p \mapsto (\Gamma_P, S)], D_P)
\]

\[
\text{upd}(\Gamma_P, \varepsilon) = \Gamma_P
\]

Remark:

\[\text{ProcEnv} : \text{Name} \rightarrow \text{ProcEnv} \times \text{Stm} \text{ (partial)}\]

\[\Gamma_P \in \text{ProcEnv}\]
Exercices

What about recursive procedures?
Type system for a (small) functional language
A small functional language

Syntax

\[ e ::= n \mid r \mid \text{true} \mid x \mid \text{fun } x : \tau.e \mid (e\ e) \mid (e,\ e) \]

\[ \tau ::= \text{ Bool } \mid \text{ Int } \mid \text{ Real } \mid \tau \rightarrow \tau \mid \tau \times \tau \]

Examples

- 42
- \((x\ 12.5)\)
- \((x,\ \text{true})\)
- \text{fun } x : \text{Bool}.\ x\)
- \(((\text{fun } x : \text{Bool}.\ x)\ 12)\)
- \text{fun } x : \text{Int }\rightarrow\ \text{Real}.\ (x\ 12)\)
Version 1: no polymorhism, explicit type annotations

Judgement

\( \Gamma \vdash e : \tau \) means “in environment \( \Gamma \), \( e \) is well-typed and of type \( \tau \)”

Type System

\( \Gamma \vdash n : \text{Int} \quad \Gamma \vdash r : \text{Real} \quad \Gamma \vdash \text{true} : \text{Bool} \)

\[ \Gamma \vdash x : \Gamma(x) \quad \Gamma \vdash \text{fun } x : \tau_1 . e : \tau_1 \mapsto \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash (e_1 , e_2) : \tau_1 \times \tau_2 \quad \Gamma \vdash (e_1 e_2) : \tau_2 \]
Extension

We add a new construct:

\[
\text{let } x = e_1 : \tau_1 \text{ in } e_2
\]

Informal semantics:

within \(e_2\), each occurrence of \(x\) is replaced by \(e_1\)

Type System

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 : \tau_1 \text{ in } e_2 : \tau_2}
\]
Version 2: no polymorphism, no type annotations

New Syntax

\[ e ::= \ldots \mid \text{fun } x. e \mid \text{let } x = e_1 \text{ in } e_2 \]

Modified rules

\[
\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \text{fun } x. e : \tau_1 \mapsto \tau_2}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}
\]

⇒ a unique value for type \( \tau_1 \) has to be inferred \ldots
Examples

Expressions that can be typed:

- \(((\text{fun } x.x) \ 1)\)
- \(((\text{fun } x.x) \ \text{true})\)
- \(\text{let } x = 1 \ \text{in } ((\text{fun } y.y) \ x)\)
- \(\text{let } f = \text{fun } x.x \ \text{in } (f \ 2)\)

Expressions that cannot be typed: \(\not\exists (\Gamma, \tau) \text{ such that } \Gamma \vdash e : \tau\)

- \((1 \ 2)\)
- \(\text{fun } x.(x \ x)\)
- \(\text{let } f = \text{fun } x.x \ \text{in } ((f \ 1) , (f \ \text{true}))\)
Polymorphism?

We introduce:

- type variable $\alpha$
- $\forall \alpha. \tau$ means “$\alpha$ can take any type within type expression $\tau$”

example: fun $x.x$ is of type $\forall \alpha. \alpha \rightarrow \alpha$

Set of free type variables of an environment $\Gamma$:

$$
\begin{align*}
\mathcal{D}(\text{Bool}) &= \mathcal{D}(\text{Int}) = \mathcal{D}(\text{Real}) = \emptyset \\
\mathcal{D}(\alpha) &= \{\alpha\} \\
\mathcal{D}(\tau_1 \rightarrow \tau_2) &= \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2) \\
\mathcal{D}(\forall \alpha \cdot \tau) &= \mathcal{D}(\tau) \setminus \{\alpha\} \\
\mathcal{D}(\Gamma) &= \bigcup_{x \in \text{dom}(\Gamma)} \mathcal{D}(\Gamma(x))
\end{align*}
$$
Polymorphism: the F system

2 new rules

\[ \frac{\Gamma \vdash e : \tau \quad \alpha \not\in D(\Gamma)}{\Gamma \vdash e : \forall \alpha \cdot \tau} \] (generalization)

\[ \frac{\Gamma \vdash e : \forall \alpha \cdot \tau}{\Gamma \vdash e : \tau[\tau' \mapsto \alpha]} \] (instanciation)

examples:

- let \( f = \text{fun } x.x \text{ in } ((f \ 1), (f \ \text{true})) \)
- \( \text{fun } x.(x \ x) \)

Rk: type inference is no longer decidable in this type system ...
Polymorphism: Hindley-Milner system

Type quantifiers may only appear “in front” of type expressions

**Types**
\[ \tau ::= \text{Bool} \mid \text{Int} \mid \text{Real} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \alpha \]

**Type patterns**
\[ \sigma ::= \tau \mid \forall \alpha \cdot \sigma. \]

3 rules are modified:

\[
\begin{array}{c}
\Gamma \vdash e : \sigma \\
\alpha \not\in D(\Gamma)
\end{array}
\]

\[
\Gamma \vdash e : \forall \alpha \cdot \sigma
\]

\[
\Gamma \vdash e : \sigma[\tau \mapsto \alpha]
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \sigma_1 \\
\Gamma \vdash e_2 : \sigma_2
\end{array}
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma_2
\]

example: \text{let } f = \text{fun } x.x \text{ in } ((f \ 1), (f \ true))
Some implementation issues
Several issues to be handled during static semantic analysis:

1. type-check the input AST
   - formal specification = a type system
   - notion of environment (name binding), to be computed:
     \[ \Gamma_V : Name \rightarrow Type \]
     \[ \Gamma_P : Name \rightarrow \{proc\} \]

2. decorate this AST to prepare code generation
   - give a type to intermediate nodes
   - indicate implicit type conversions

⇒ from type system to algorithms ?
Example (1)

begin
    var x : Int ;
    var y : Real ;
    y := 2 * x + y * (3 + x) ;
end

Initial AST
Example (2)

begin
  var x : Int ;
  var y : Real ;
  y := 2 * x + y * (3 + x) ;
end
From type system to algorithms?

⇒ recursive traversal of the AST . . .

AST representation:

```c
typedef struct tnode {
    String string; // lexical representation
    kind elem; // category (idf, binaop, while, etc.)
    struct tnode *left, *right; // children
    Type type; // type (Int, Real, Void, Bad, etc.)
    ...
} Node;
```

Type-checking function:

```c
Type TypeCheck(*node);
// checks the correctness of node, returns the result Type
// and inserts type conversions when necessary
```
**Type checking algorithm for arithmetic expressions**

<table>
<thead>
<tr>
<th>DENOT</th>
<th>BINAOP</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ n: Int</td>
<td>Γ ⊢ e_l : T_l, Γ ⊢ e_r : T_r, T = resType(T_r, T_l)</td>
<td>Γ(x) = t</td>
</tr>
<tr>
<td></td>
<td>Γ ⊢ e_l binaop e_r : T</td>
<td>Γ ⊢ x : t</td>
</tr>
</tbody>
</table>

function Type typeCheck(Node *node) {
    switch node->elem {
        case DENOT: break; // lexical analysis
        case IDF: node->type=Gamma(node->string); break; // environment
        case BINAOP: // type-checking
            T_l=typeCheck(node->left);
            T_r=typeCheck(node->right);
            node->type=resType(T_l, T_r);
            if (node->type != T_l) insConversion(node->left, node->type);
            if (node->type != T_r) insConversion(node->right, node->type);
            break;
    }
    return node->type;
}

function Type resType(Type t1, Type t2) {
    if (t1==Boolean) or (t2==Boolean) return Bad;
    else return Max(t1, t2);
}
Type checking algorithm for statements

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Iteration</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash S_1$</td>
<td>$\Gamma \vdash b : \text{Bool}$</td>
<td>$\Gamma \vdash x : t$</td>
</tr>
<tr>
<td>$\Gamma \vdash S_2$</td>
<td>$\Gamma \vdash S$</td>
<td>$\Gamma \vdash e : t$</td>
</tr>
<tr>
<td>$\Gamma \vdash S_1;S_2$</td>
<td>$\Gamma \vdash \text{while } b \text{ do } S$</td>
<td>$\Gamma \vdash x := e$</td>
</tr>
</tbody>
</table>

function Type typeCheck(Node *node) {
    switch node->elem {
        case SEQUENCE:
            if (typeCheck(node->left) != Void) return BAD;
            return typeCheck(node->right);
        case WHILE:
            if (typeCheck(node->left) != BOOL) return BAD;
            return typeCheck(node->right);
        case ASSIGN:
            Tl = typeCheck(node->left);
            Tr = typeCheck(node->right);
            if (Tl != Tr) return BAD else return VOID;
    }
}
Environment implementation and name binding?

- associate a Type to each identifier
  - each use occurrence $\mapsto$ decl occurrence
  - info should be retrieved efficiently (no AST traversal)

- handle nested declarations:

begin
  var x : Int ; var y : Real ;
  begin
    var x : Boolean ;
    x = y > 2.5 ;
  end
end
Usual solution: symbol table

- store all information associated to an identifier: type, kind (var, param, proc), address (for code gen), etc.

- built during traversals of the declaration parts of the AST

- efficient search procedure: binary tree, hash table, etc.

- two solutions for handling nested blocks ($\Gamma[x \rightarrow \text{Bool}]$)
  
  - a global table, with a unique id is associated to each idf:
    \[
    \{(x, 1) : \text{Int}, (y, 1) : \text{Real}, ((x, 1.1) : \text{Bool})\}
    \]
    → based on a unique (hierarchical) numbering of blocks
  
  - a dynamic stack of local tables, one local table per block:
    \[
    \{x : \text{Int}, y : \text{Real}\} \rightarrow \{x : \text{Bool}\}\]