Programming Language Semantics and Compiler Design / Sémantique des Langages de Programmation et Compilation

Lexical, Syntactic, and Type Analysis

Yliès Falcone
ylies.falcone@univ-grenoble-alpes.fr — www.ylies.fr
Univ. Grenoble Alpes, and LIG-Inria team CORSE

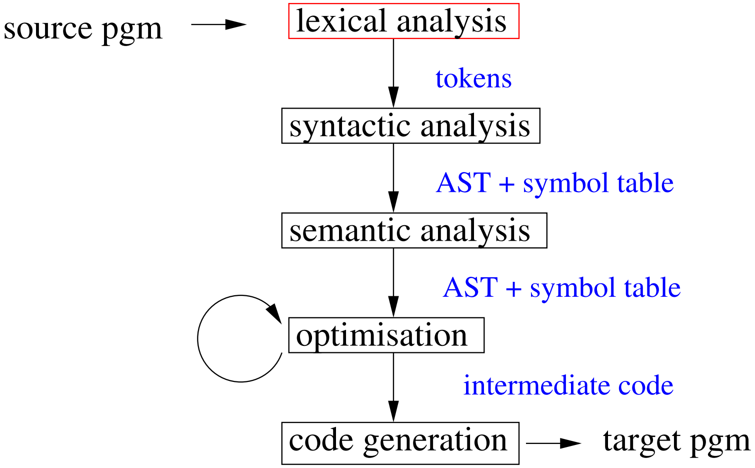
Master of Sciences in Informatics at Grenoble (MoSIG)
Master 1 info

Univ. Grenoble Alpes - UFR IM^{2}AG
www.univ-grenoble-alpes.fr - im2ag.univ-grenoble-alpes.fr

Academic Year 2017 - 2018

Outline - Lexical, Syntactic, and Type Analysis

Compiler architecture



Lexical Analysis

Regular languages

- regular Expressions – *language description*
- (Non-) Deterministic Finite State Automata – *language recognition*
- regular grammars – *language generation/description*

Thus, a lexical analyzer may be

- specified by regular expressions,
- implemented by a Deterministic Finite State Automaton.

Lexical Analyzer Generator

LeX : from Regular expression to Finite State Automaton

LeX description

```
declarations
%%
rules
%%
procedures
```

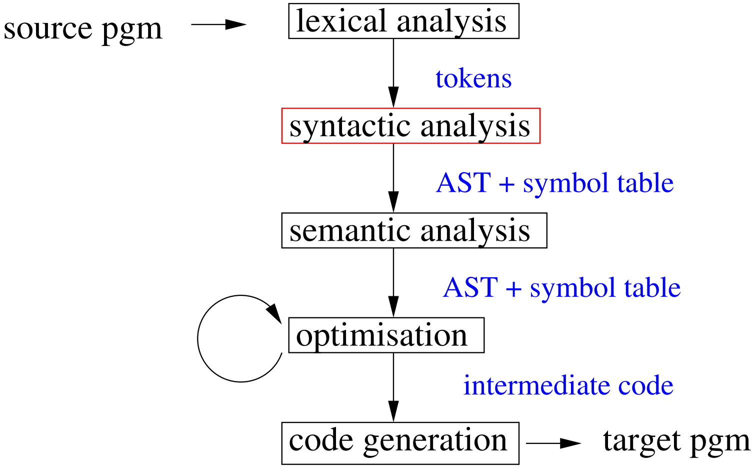
Example of declaration :

```
digit [0-9]
integer {digit}+
```

Example of rule description :

```
{integer} {val=atoi(yytext);return(Integer);}
```

Compiler architecture



Syntactic Analysis

Context-free languages

- Push-down automata – *language recognition*
- Context-free grammar – *language generation/description*

Thus, an LR parser can be

- specified by a LR grammars
- implemented by a deterministic push-down automata

Parser Generator

Yacc/Bison : from HC grammar to push-down automata

Yacc/Bison description

```
declarations
%%
rules
%%
procedures
```

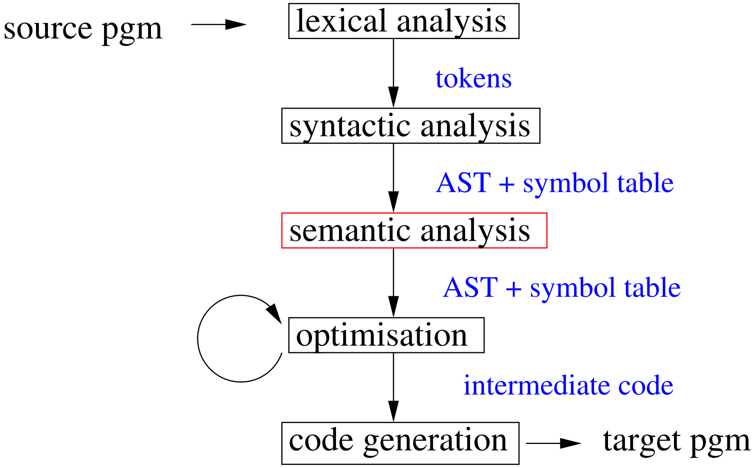
Example of declaration :

```
%type <u_node> program
%type <u_node> e
```

Example of rule description :

```
e :  e '+' t
  {$$=m_node(PLUS,$1,$3);}
 | t
  {$$=$1;}
 ;
```

Compiler architecture



Static Semantic Analysis

Principles and purposes

Input: : Abstract Syntax Tree (AST)

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

Two main purposes:

- name identification: $\rightarrow$ bind **use-def** occurrences
- type verification and/or type inference

Outline: Type Analysis

Outline: Type Analysis

About Types

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.
- It defines the resulting type of an expression after applying an operation.

Objectives: anticipate runtime errors.

Example (Types)

int, float, unsigned int, signed int, string, array, list, . . .

What are Types Useful for?

Program correctness

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

Program readability

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

Program 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).

Example (Typed languages)

Java, Ada, C, Pascal, CAML, etc.

Remark strongly typed vs weakly typed languages...

Untyped languages

A single (universal) type is associated to each identifier
(and hence to each expression).

Example (Untyped languages)

Assembly language, shell-script, Lisp, etc.

Typed languages 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.

Sub-typing

- 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 the 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 program.

Type inference

Compute a consistent type for each program fragments.

Remark In some languages (e.g., Haskel, CAML), there are/can be no type annotations at all (all types are/can be infered). □

Static checking vs dynamic checking

Static checking

Verification performed at *compile-time*.

Dynamic checking

Verification performed at *run-time*.
$\rightarrow$ necessary to correctly handle:

- dynamic binding for variables or procedures
- polymorphism
- array bounds
- subtyping
- etc.

$\Rightarrow$ For most programming languages, both kinds of checks are used...

Outline: Type Analysis

Getting the Intuition on Examples

- "$2 + 3 = 6$" is well-typed
- "$2 + \text{true} = \text{false}$" is not well-typed
- "$x = \text{false}$" 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
- "$\text{let } x = 3 \text{ in } x + y$" is well-typed
  if y is a (visible) integer/real variable

$\Rightarrow$ a term t can be type-checked
under assumptions on its **free variables** ...

How to Formalize a Type System?

- Abstract syntax describes terms (represented by ASTs).
- Environment Γ: *Name* $\stackrel{\text{part.}}{\to}$ *Types*.
- Judgment $\Gamma \vdash t : \tau$.

  "In environment Γ, term t is well-typed and has type τ."

  (free variables of t belong to the domain of Γ)
- Type system

Inference rules	**Axioms**
$\dfrac{\Gamma_1 \vdash \mathcal{A}_1 \quad \cdots \quad \Gamma_n \vdash \mathcal{A}_n}{\Gamma \vdash \mathcal{A}}$	$\Gamma \vdash \mathcal{A}$

Remark A type system is an inference system. □

Example: natural numbers

$e \quad := \quad n \mid x \mid e_1 + e_2$

Syntax

$$\frac{\Gamma(x) = \textbf{Nat}}{\Gamma \vdash x : \textbf{Nat}}$$

x is of type **Nat** in environment Γ if $\Gamma(x) = \textbf{Nat}$.

$$\frac{}{\Gamma \vdash n : \textbf{Nat}}$$

The denotation n is of type **Nat**.

$$\frac{\Gamma \vdash e_1 : \textbf{Nat} \quad \Gamma \vdash e_2 : \textbf{Nat}}{\Gamma \vdash e_1 + e_2 : \textbf{Nat}}$$

$e_1 + e_2$ is of type **Nat** assuming that e_1 and e_2 are of type **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

$$\frac{\emptyset \vdash 1 : \textbf{Nat} \qquad \emptyset \vdash 2 : \textbf{Nat}}{\emptyset \vdash 1\ +\ 2 : \textbf{Nat}}$$

Exercise

Prove that $[x \rightarrow \textbf{Nat}, y \rightarrow \textbf{Nat}] \vdash x + 2 : \textbf{Nat}$.

Outline: Type Analysis

Outline: Type Analysis

Syntax of Language **While**

Expressions

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

$$\texttt{e} \quad ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{n} \mid \texttt{x} \mid \texttt{e opa e} \mid \texttt{e oprel e} \mid \texttt{e opb e}$$

Statements

$$\begin{array}{lll} \mathsf{S} & ::= & \mathsf{x := e} \mid \mathsf{skip} \mid \mathsf{S}\ ;\ \mathsf{S} \mid \\ & & \mathsf{if}\ e\ \mathsf{then}\ S\ \mathsf{else}\ S\ \mathsf{fi} \mid \mathsf{while}\ e\ \ \mathsf{do}\ S\ \mathsf{od} \end{array}$$

Judgments

- $\Gamma \vdash S$
  "In environment Γ, statement S is well-typed".

- $\Gamma \vdash e : t$
  "In environment Γ, expression e is of type t".

Type System for Expressions

bool. constant	int. constant	int opbin
$\dfrac{}{\Gamma \vdash \texttt{true} : \textbf{Bool}}$ $\dfrac{}{\Gamma \vdash \texttt{false} : \textbf{Bool}}$	$\dfrac{}{\Gamma \vdash \texttt{n} : \textbf{Int}}$	$\dfrac{\Gamma \vdash e_1 : \textbf{Int} \quad \Gamma \vdash e_2 : \textbf{Int}}{\Gamma \vdash e_1 \texttt{ opa } e_2 : \textbf{Int}}$

variables	bool. opbin	relational operators
$\dfrac{\Gamma(x) = t}{\Gamma \vdash x : t}$	$\dfrac{\Gamma \vdash e_1 : \textbf{Bool} \quad \Gamma \vdash e_1 : \textbf{Bool}}{\Gamma \vdash e_1 \texttt{ opb } e_2 : \textbf{Bool}}$	$\dfrac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \texttt{ oprel } e_2 : \textbf{Bool}}$

Type system for Statements

Assignment	**Skip**
$\dfrac{\Gamma \vdash e : t \quad \Gamma \vdash x : t}{\Gamma \vdash x := e}$	$\dfrac{}{\Gamma \vdash \texttt{skip}}$

Sequence	**Iteration**
$\dfrac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1 ; S_2}$	$\dfrac{\Gamma \vdash e : \textbf{Bool} \quad \Gamma \vdash S}{\Gamma \vdash \text{while } e \text{ do } S \text{ od}}$

Exercises

Exercise: conditional statement

Complete the type system by providing a rule for *conditional statements*.

Exercise: introducing reals and type conversion

Extend the type system for the expressions assuming that arithmetic types can be now either integer (**Int**) or real (**Real**).

Several solutions are possible:

1. Type conversions are never allowed.
2. Only explicit conversions (with a cast operator) are allowed.
3. (implicit) conversions are allowed.

Outline: Type Analysis

Language **Block**

Reminder

A new syntactic rule for statements:

$$S ::= \cdots \mid \textbf{begin } D_V \; ; \; S \textbf{ end}$$

And for declarations:

$$D_V ::= \textbf{var } x := e \; ; \; D_V \mid \epsilon$$

The semantics is such that:

- one executes S in the state updated after evaluating variable declarations;
- (values of) variables are restored after the execution of S.

Extending the Type System

Notations

- $\texttt{DV}(D_v)$ denotes the set of variables **declared** in D_v.
- $\Gamma[y \mapsto \tau]$ denotes the environment Γ' such that:
  - $\Gamma'(x) = \Gamma(x)$ if $x \neq y$
  - $\Gamma'(y) = \tau$

Judgments

- $\Gamma \vdash \; D_V \mid \Gamma_l$ means

  "declarations D_V update environment Γ into Γ_l"

- $\Gamma \vdash \; S$ means

  "statement S is well-typed within environment Γ"

Extending the Type System

Inference rule for Blocks

$$\frac{\Gamma \vdash D_V \mid \Gamma_l \quad \Gamma_l \vdash S}{\Gamma \vdash \textbf{begin } D_V \ ; \ S \textbf{ end}}$$

Inference rules for declarations

Sequential evaluation

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma \vdash e : t \quad \Gamma[x \mapsto t] \vdash D_V \mid \Gamma_l \quad x \notin \mathtt{DV}(D_V)}{\Gamma \vdash \textbf{var } x := e \ ; \ D_V \mid \Gamma_l}$$

Collateral evaluation

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash D_V \mid \Gamma_l \quad x \notin \mathtt{DV}(D_V)}{\Gamma \vdash \textbf{var } x := e; D_V \mid \Gamma_l[x \mapsto t]}$$

A variable should be declared at most once

Some Alternatives for Variable Declarations

Inference rules for declarations allowing re-declarations

Sequential evaluation

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma \vdash e : t \quad \Gamma[x \mapsto t] \vdash D_V \mid \Gamma_l}{\Gamma \vdash \textbf{var } x := e\ ;\ D_V \mid \Gamma_l[x \mapsto t]}$$

Collateral evaluation

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash D_V \mid \Gamma_l}{\Gamma \vdash \textbf{var } x := e; D_V \mid \Gamma_l[x \mapsto t]}$$

This gives priority to the first declarations.

Alternative declarations

- explicitely typed variables: `var x := e : t`
- uninitialized variables: `var x : t`
- untyped variables(?): `var x := e`
- uninitialized and untyped variables(???): `var x`

We will study these alternatives during the tutorial

Outline: Type Analysis

Language **Proc**

Syntactic rules for statements:

$$S ::= \cdots \mid \textbf{begin } D_V \; ; D_P \; ; \; S \textbf{ end} \mid \textbf{call } p$$

and for declarations:

$$D_P ::= \textbf{proc } p \textbf{ is } S \; ; \; D_P \mid \epsilon$$

$DP(D_P)$ denotes the set of procedures **declared** in D_P.

The semantics depends on the kind of binding (static vs dynamic) one considers. . .

Judgments

- Procedure environment $\Gamma_P : \mathit{Name} \rightarrow \{\mathit{proc}\}$ (partial)

- $\Gamma_V \vdash D_V \mid \Gamma'_V$ means

  "Variable declarations D_V update variable environment Γ_V into Γ'_V".

- $(\Gamma_V, \Gamma_P) \vdash D_P$ means

  "Procedure declarations D_P is well-typed within variable and procedure environments (Γ_V, Γ_P)."

- $(\Gamma_V, \Gamma_P) \vdash S$ means

  "Statement S is well-typed within variable and procedure environments (Γ_V, Γ_P).

Example : Static Binding for Procedures and Variables

Example (Static binding for variables and procedures)

$$
\begin{array}{ll}
\text{begin} & \text{var } x := 0; \\
& \text{proc } p \text{ is } x := x * 2; \\
& \text{proc } q \text{ is call } p; \\
& \text{begin var } x := 5; \\
& \quad \text{proc } p \text{ is } x := x + 1; \\
& \quad \text{call } q; y := x; \\
& \text{end}; \\
\text{end} &
\end{array}
$$

We need to:

- have some "memorization" of the current "procedure mapping" that "remembers the current procedure definitions when it has been defined"
- know the "memory location" currently designated by a variable name

$\hookrightarrow$ when we call q we call p and modify x

Static Binding for Procedures and Variables

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

$$D_P\quad \frac{(\Gamma_V, \Gamma_P) \vdash S \quad (\Gamma_V, \Gamma_P[p \mapsto \textbf{proc}]) \vdash\ D_P \quad p \notin DP(D_P)}{(\Gamma_V, \Gamma_P) \vdash \textbf{proc}\ p\ \textbf{is}\ S\ ;\ D_P}$$

$$\text{Call}\quad \frac{\Gamma_P(p) = \texttt{proc}}{(\Gamma_V, \Gamma_P) \vdash\ \textbf{call}\ p}$$

- where $\Gamma'_P = \mathtt{upd}(\Gamma_P, D_P)$
- with :

$$\begin{array}{rcl} \mathtt{upd}(\Gamma_P, \textbf{proc}\ p\ \textbf{is}\ S\ ;\ D_P) & = & \mathtt{upd}(\Gamma_P[p \mapsto \textbf{proc}], D_P) \\ \mathtt{upd}(\Gamma_P, \varepsilon) & = & \Gamma_P \end{array}$$

Example: Dynamic Binding for Procedures and Variables

Example (Dynamic binding for variables and procedures)

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call q; y := x;
        end;
end
```

We need to have some "memorization" of the current "procedure mapping"
$\hookrightarrow$ when we call q we call p

Dynamic Binding for Procedures and Variables

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

$$\text{Call} \quad \frac{(\Gamma_V, \Gamma_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \textbf{ call } p} \ \Gamma_P(p) = S$$

- where $\Gamma'_P = \mathrm{upd}(\Gamma_P, D_P)$
- with:

$$\begin{array}{rcl} \mathrm{upd}(\Gamma_P, \textbf{proc } p \textbf{ is } S \ ; \ D_P) & = & \mathrm{upd}(\Gamma_P[p \mapsto S], D_P) \\ \mathrm{upd}(\Gamma_P, \varepsilon) & = & \Gamma_P \\ \mathtt{udef}(\textbf{proc } p \textbf{ is } S \ ; \ D_P)) & = & \mathtt{udef}(D_P) \wedge p \notin DP(D_P) \\ \mathtt{udef}(\varepsilon) & = & \text{true} \end{array}$$

Remark procedure environment $\Gamma_P : \mathit{Name} \to \mathit{Stm}$ (partial)

Illustrating the difference between static and dynamic binding for vars/procs

```
begin
   var x := 3;
   proc p is x := x + 1;
   begin
      var x := true ;
      call p
   end
end
```

- correct w.r.t. static binding for variables and procedures
- incorrect w.r.t. dynamic binding for variables and procedures

Illustrating the difference between static and dynamic binding for vars/procs (ctd)

```
begin
   var x := true;
   proc p is x := x + 1;
   begin
      var x := 3;
      call p
   end
end
```

- incorrect w.r.t. static binding for variables and procedures
- correct w.r.t. dynamic binding for variables and procedures

Outline: Type Analysis

A Small Functional Language

Syntax of the language

$$\begin{array}{rcl} e & ::= & n \mid r \mid \textbf{true} \mid \textbf{false} \mid x \mid \textbf{fun}\ x : \tau . e \mid (e\ e) \mid (e\ ,\ e) \\ \tau & ::= & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \tau \rightarrow \tau \mid \tau \times \tau \end{array}$$

Example (Programs)

- 42
- $(x\ 12.5)$
- $(x\ ,\ \mathit{true})$
- $\textbf{fun}\ x : \textbf{Bool}.\ x$
- $((\textbf{fun}\ x : \textbf{Bool}.\ x)\ 12)$
- $\textbf{fun}\ x : \textbf{Int} \rightarrow \textbf{Real}.\ (x\ 12)$

Version 1: no polymorhism, explicit type annotations

Judgment

$\Gamma \vdash e : \tau$ means "In environment Γ, e is well-typed and of type τ."

Type System

$$\frac{}{\Gamma \vdash n : \mathbf{Int}} \quad \frac{}{\Gamma \vdash r : \mathbf{Real}} \quad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \quad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\ x : \tau_1 . e : \tau_1 \mapsto \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ ,\ e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \mapsto \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

Extension: definition of identifiers

We add a new construct:

$$\textbf{let } x = e_1 : \tau_1 \textbf{ in } e_2$$

Informal semantics:

within e_2, each occurrence of x is replaced by e_1

Extending the type system to handle identifiers

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

Version 2: no polymorphism, no type annotations

Syntax of the language

$$e ::= \cdots \mid \textbf{fun } x.e \mid \textbf{let } x = e_1 \textbf{ in } e_2$$

Modified type system

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

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

$\Rightarrow$ a unique value for type τ_1 has to be infered ...

Examples

Expressions that can be typed:

- $((\textbf{fun}\ x.x)\ 1) : \textbf{Int}$
- $((\textbf{fun}\ x.x)\ \textbf{true}) : \textbf{Bool}$
- $\textbf{let}\ x = 1\ \textbf{in}\ ((\textbf{fun}\ y.y)\ x) : \textbf{Int}$
- $\textbf{let}\ f = \textbf{fun}\ x.x\ \textbf{in}\ (f\ 2) : \textbf{Int}$

Expressions that cannot be typed

$\nexists(\Gamma, \tau)$ such that $\Gamma \vdash e : \tau$

- $(1\ 2)$
- $\textbf{fun}\ x.(x\ x)$
- $\textbf{let}\ f = \textbf{fun}\ x.x\ \textbf{in}\ ((f\ 1)\ ,\ (f\ \textbf{true}))$

Polymorphism?

We introduce:

- type variable α
- $\forall\alpha.\tau$ means "α can take any type within type expression τ"

Example (Polymorphic expression)

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

Definition (Set of free type variables)

Given an environment Γ:

$$\mathcal{D}(\textbf{Bool}) = \mathcal{D}(\textbf{Int}) = \mathcal{D}(\textbf{Real}) = \emptyset$$

$$\begin{array}{rcl}
\mathcal{D}(\alpha) & = & \{\alpha\} \\
\mathcal{D}(\tau_1 \longrightarrow \tau_2) & = & \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2) \\
\mathcal{D}(\forall\alpha \cdot \tau) & = & \mathcal{D}(\tau) \setminus \{\alpha\} \\
\mathcal{D}(\Gamma) & = & \displaystyle\bigcup_{x \in \textbf{dom}(\Gamma)} \mathcal{D}(\Gamma(x))
\end{array}$$

Polymorphism: the F system

Definition (Rules for system F)

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{D}(\Gamma)}{\Gamma \vdash e : \forall \alpha \cdot \tau} \quad \text{(generalization)}$$

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

Example (Programs)

- **let** f = **fun** $x.x$ **in** ((f 1) , (f **true**))
- **fun** $x.(x\ x)$

Remark Type inference is no longer decidable in this type system. . . □

Polymorphism: the Hindley-Milner system

Type quantifiers may only appear "in front" of type expressions.

Definition (New Syntax)

$$\begin{array}{rcll} \textbf{Types} & \tau & ::= & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \tau \longrightarrow \tau \mid \tau \times \tau \mid \alpha \\ \textbf{Type patterns} & \sigma & ::= & \tau \mid \forall\alpha \cdot \sigma. \end{array}$$

Definition (New Rules for the Hindley-Milner system)

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{D}(\Gamma)}{\Gamma \vdash e : \forall\alpha \cdot \sigma} \qquad \text{(generalization)}$$

$$\frac{\Gamma \vdash e : \forall\alpha \cdot \sigma}{\Gamma \vdash e : \sigma[\tau \mapsto \alpha]} \qquad \text{(instanciation)}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma[x \mapsto \sigma_1] \vdash e_2 : \sigma_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \sigma_2} \qquad \text{(polymorph ``let'')}$$

Example

let f = **fun** $x.x$ **in** ((f 1) , (f **true**))

Outline: Type Analysis

Reminder

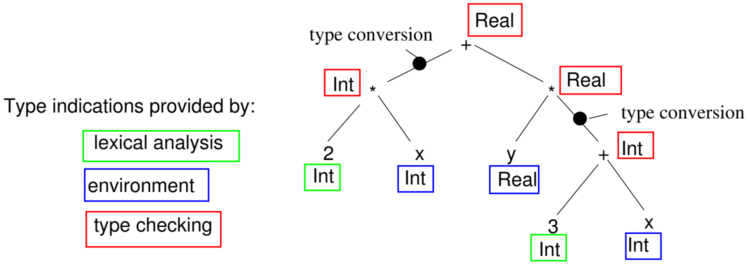
Several issues to be handled during static semantic analysis:

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

$\Rightarrow$ How to go from type system to algorithms?

Example

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



Final AST

From a Type System to Algorithms?

⇒ recursive traversal of the AST...

AST representation:

```
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:

```
Type TypeCheck(* node) ;
// checks the correctness of node, returns the result Type
// and inserts type conversions when necessary
```

Type Checking Algorithm for Arithmetic Expressions

DENOT	BINAOP	IDF
$\dfrac{}{\Gamma \vdash \mathtt{n} : \textbf{Int}}$	$\dfrac{\Gamma \vdash e_l : Tl \quad \Gamma \vdash e_r : Tr \quad T = resType(Tr, Tl)}{\Gamma \vdash e_l \ \mathtt{binaop} \ e_r : T}$	$\dfrac{\Gamma(x) = t}{\Gamma \vdash x : t}$

```
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
     Tl=typeCheck(node->left);
     Tr=typeCheck(node->right);
     node->type=resType(Tl, Tr);
     if (node->type != Tl) insConversion(node->left, node->type);
     if (node->type != Tr) 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

Sequence	Iteration	Assignment
$\dfrac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1 ; S_2}$	$\dfrac{\Gamma \vdash e : \mathbf{Bool} \quad \Gamma \vdash S}{\Gamma \vdash \texttt{while} \quad e \text{ do } S}$	$\dfrac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x := e}$

```
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)
- How can we 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 ($\Gamma[x \to \texttt{Bool}]$)
  - a global table, with a unique id associated to each idf:
    $\{((\texttt{x}, 1) : \texttt{Int}), ((\texttt{y}, 1) : \texttt{Real}), ((\texttt{x}, 1.1) : \texttt{Bool})\}$
    $\to$ based on a unique (hierarchical) numbering of blocks
  - a dynamic stack of local tables, one local table per block:
    $\{\texttt{x:Int, y:Real}\} \longrightarrow \{\texttt{x:Bool}\}$