Lecture 7-8
The In-depth knowledge Of The Language that We Must Use

prof. Marco Bellia, Dip. Informatica, Università di Pisa

March 11, 2014

In-depth Knowledge of a Programming Language

The In-depth Knowledge of the structure of a language (manly its constructs) occurs through the use of various tools whose availability depends on the specific language.

- **Formal Definition** Syntax, Semantics and Abstract Machine;
- **Execution Tools** Compilers, Interpreters with which to experience and testing programs of language;
- **Language Textbooks**: They are of help in the construction of language programs for algorithms that are both in widespread use, in programming, and in a relevant use, in the specific applications for which the language has been designed
- **Use and Reference Manuals** They contain the Description of: Single constructs, Behavior of an hypothetical Executor, Typical combinations of constructs to build programs.

In-depth Knowledge of a Programming Language/2

- **Where to find** the exact behavior of the following program?

Example

```
int main (int argc, char *argv[]){
      int z = 4;
      int y = z;
   /* int iterationCounter = 0; */
      while((z=y)>0){
            int z;
            --z;
            printf("valore di z=%2d\n", z);
            y = z;
   /*       iterationCounter++; */
      }
   /* printf("Number of iteration of the inner block %2d\n", iterationCounter); */
      return 0;
}
```

- **Is it** a terminating program?
- **Is termination** to be affected by the removal of the comment mark in red colored, statements?
- **Is its use** of control and data resources adequate for the language expressivity?

Programming Paradigms

- The study of paradigms requires the ability to answer such kind of questions
- in order to compare the exact behavior of programs that are written in different languages
- Hence, to obtaining the right in-depth knowledge of languages, we use:
  - **Denotational Semantics** which is expressed in a way that is abstract enough to do not influence the effective implementation of the construct;
  - **Models or Implementation Schema** (e.g. data model, model of memory organization,...) instead of effective implementation of the abstract machine;
  - **Programming Methodologies** in which we study the use and the role of the constructs that the different paradigms offer;
  - **Compilers e/o Interpreters** with which we experience and we will do all the testing of interesting program structures.

Denotational Semantics

Basic Structures are:

- **Syntactic Domains**
- **Semantic Domains**
- **Semantic Functions**

These three entities characterize the language and allow a analytical comparison of the different solutions that the different paradigms may offer

Denotational Semantics- Notational Remarks

- **Function:** $\lambda \mathtt{x}.\mathtt{e}$ defines a function of one argument that computes like $\mathtt{e}$
- **FixPont:** $\mathtt{Y\ F}$ where $\mathtt{F}$ is a functional on $\mathtt{f}$: Example
  $\mathtt{Y\ F} \equiv \mathtt{Y\ Fact}$ where $\mathtt{Fact}$ is the functional:
  $\lambda \mathtt{f}.\lambda \mathtt{n}.\mathtt{if}(\mathtt{n} = 0)\mathtt{then}\ 1\ \mathtt{else}\ \mathtt{n} * \mathtt{f}(\mathtt{n} - 1)$.
  Noting that: $\mathtt{Y\ F}$ is a contraction for: $\mathtt{Yf}.\lambda \mathtt{f}.\mathtt{H}$
- **To compute with FixPoint:**
  intensional use: $\forall \mathtt{x} : \mathtt{Y\ F}(\mathtt{x}) = \mathtt{F}(\mathtt{Y\ F})(\mathtt{x})$
  extensional use: $\mathtt{Y\ F} = \mathtt{Lim}_{\mathtt{n} \in \mathtt{Nat}}(\mathtt{Y\ F})^{\mathtt{n}}$ where:
  $$(\mathtt{Y\ F})^0 = \mathtt{F}(\bot)$$
  $$(\mathtt{Y\ F})^{\mathtt{n}} = \mathtt{F}((\mathtt{Y\ F})^{\mathtt{n}-1})$$

Example

intensional use:

$$\begin{aligned}\mathtt{Y\ Fact}(2) &= \mathtt{if}(2 = 0)\mathtt{then}\ 1\ \mathtt{else}\ 2 * \mathtt{Y\ Fact}(1)\\ &= 2 * (\mathtt{if}(1 = 0)\mathtt{then}\ 1\ \mathtt{else}\ 1 * \mathtt{Y\ Fact}(0))\\ &= 2 * 1 * (\mathtt{if}(0 = 0)\mathtt{then}\ 1\ \mathtt{else}\ 1 * \mathtt{Y\ Fact}(0))\\ &= 2 * 1 * 1\end{aligned}$$

extensional use:

$$\begin{aligned}\mathtt{Y\ Fact}^0 &= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n} = 0)\mathtt{then}\ 1\ \mathtt{else}\ \bot\\ \mathtt{Y\ Fact}^1 &= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n} = 0)\mathtt{then}\ 1\ \mathtt{else}\ \mathtt{n} * (\lambda \mathtt{n}.\mathtt{if}(\mathtt{n} = 0)\mathtt{then}\ 1\ \mathtt{else} \bot)(\mathtt{n} - 1)\\ &= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n} = 0)\mathtt{then}\ 1\ \mathtt{else}\ (\mathtt{if}(\mathtt{n} = 1)\mathtt{then}\ \mathtt{n} * 1\ \mathtt{else}\ \mathtt{n} * \bot)\\ &= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n} = 0)\mathtt{then}\ 1\ \mathtt{else}\ (\mathtt{if}(\mathtt{n} = 1)\mathtt{then}\ 1\ \mathtt{else} \bot)\\ \mathtt{Y\ Fact}^2 &= \ldots\end{aligned}$$

Syntactic Domains

Table 1			
Syntactic Domains			
D	::=	Proc I() C; | D D | ...	(*Declaration*)
C	::=	{D C} | I := E | Call I () | ...	(*Command*)
E	::=	I | VL | ...	(*Expressions*)
VL	::=	...	(*Literal*)

Example

Complete the list below, of the syntax constructors, used in the table, and show the signature of each one:

$\{\_\ \_\} : \mathtt{D} \times \mathtt{C} \rightarrow \mathtt{C}$

Semantic Domains/1

Table 2

Semantic Domains

$\mathtt{Env}, \rho, \delta$	$\equiv$	$\mathtt{I} \rightarrow \mathtt{Den}$	(*Environment*)
		Operations di $\mathtt{Env}$:	
		$\mathtt{bind}: \ \mathtt{I} \times \mathtt{Den} \times \mathtt{Env} \rightarrow \mathtt{Env}$	
		$\mathtt{bind}(\mathtt{I}, \mathtt{d}, \rho) \equiv \lambda \mathtt{x}.\ \mathtt{if}((\mathtt{x}\ eq\ \mathtt{I}), \mathtt{d}, \rho(\mathtt{x}))$	
		$\mathtt{find}: \ \mathtt{I} \times \mathtt{Env} \rightarrow \mathtt{Den}$	
		$\mathtt{find}(\mathtt{I}, \rho) \equiv \rho(\mathtt{I})$	
		$\mathtt{empty}: \mathtt{Env}$	
		$\mathtt{empty} \equiv \lambda \mathtt{x}.\ \mathtt{x}$	

Semantic Domains/2

Table 2.continued			
$\mathtt{Store}, \mathtt{s}$	$\equiv$	$\ldots$	(*Memory*)
		Operations of $\mathtt{Store}$:	
		$\mathtt{upd} : \mathtt{Loc} \times \mathtt{Mem} \times \mathtt{Store} \rightarrow \mathtt{Store}$	
		$\mathtt{look} : \mathtt{Loc} \times \mathtt{Store} \rightarrow \mathtt{Mem}$	
$\mathtt{State}$	$::=$	$\ldots$	(*State*)
Auxiliary Semantic Domains			
$\mathtt{Val}, \mathtt{v}$	$::=$	$\ldots$	(*Language Values*)
$\mathtt{Den}, \mathtt{d}$	$::=$	$\ldots$	(*Language Den*)
$\mathtt{Mem}, \mathtt{d}$	$::=$	$\ldots$	(*Language Mem*)
$\mathtt{Loc}, \mathtt{l}$	$::=$	$\ldots$	(*Locations*)
$\mathtt{Input}$	$::=$	$\ldots$	(*I/O : Input*)
$\mathtt{Output}$	$::=$	$\ldots$	(*I/O : Output*)

Semantic Functions/1

Table 3	
Semantic Function	
$\mathcal{D} : \mathtt{D} \rightarrow \mathtt{Env} \rightarrow \mathtt{Env}$	(*Declarations*)
$\mathcal{M} : \mathtt{C} \rightarrow \mathtt{Env} \rightarrow \mathtt{State} \rightarrow \mathtt{State}$	(*Commands*)
$\mathcal{E} : \mathtt{E} \rightarrow \mathtt{Env} \rightarrow \mathtt{State} \rightarrow \mathtt{Val}$	(*Espressions*)
Auxiliary Domains	
$\mathtt{VL} ::= \mathtt{Int} + \mathtt{Char}$	(*Literals : Disjoint Union*)
$\mathtt{Int} ::= \ldots$	(*Literals for integers*)
$\mathtt{Char} ::= \ldots$	(*Literals for characters*)

Semantic Functions/2

Auxiliary Functions (for Disjoint Union)

(*injections, i.e. are constructors [surjective??]*)

$\mathtt{N} : \mathtt{VL}_{\mathtt{Int}} \to \mathtt{Int}$

$\mathtt{C} : \mathtt{VL}_{\mathtt{Char}} \to \mathtt{Char}$

$\mathtt{V} : \mathtt{Int} \cup \mathtt{Char} \to \mathtt{VL}$

$\mathtt{NtoVL} : \mathtt{Int} \to \mathtt{VL}$

$\mathtt{NtoVL} = \lambda \mathtt{x}.\mathtt{V}(\mathtt{x})$

$\mathtt{CtoVL} : \mathtt{Char} \to \mathtt{VL}$

$\mathtt{CtoVL} = \lambda \mathtt{x}.\mathtt{V}(\mathtt{x})$

$\mathtt{VLtoN} : \mathtt{VL} \to \mathtt{Int}_{\perp}$

$\mathtt{VLtoC} : \mathtt{VL} \to \mathtt{Char}_{\perp}$

$\mathtt{VLtoN} = \lambda \mathtt{x}.\mathtt{if}((\mathtt{x} \in \mathtt{Int}), \mathtt{N}(\mathtt{x}), \perp_{\mathtt{Int}})$

$\in\mathtt{Int} : \mathtt{VL} \to \mathtt{TruthV}$

$\in\mathtt{Char} : \mathtt{VL} \to \mathtt{TruthV}$

$\in\mathtt{Int} = \lambda \mathtt{x}.\mathtt{x}\ \mathit{eq}\ \mathtt{V}(\mathtt{N}(\mathtt{x}))$

$\mathtt{MV} : \mathtt{Mem} \to \mathtt{Val}$

$\mathtt{IntoVal} : \mathtt{VL} \to \mathtt{Val}$

Block Structures Languages: Inline and Procedure/1

Table5 – Imperative Languages : Static Scope	
Semantic Functions	
$\mathcal{D}[\![\mathtt{D}]\!]_\rho : Env$	*(Meaning of the Declarations)*
$\quad \mathcal{D}[\![\mathtt{Proc\ I()\ C}]\!]_\rho = \mathtt{bind}(\mathtt{I}, \mathcal{M}[\![\mathtt{C}]\!]_\rho, \rho)$	
$\mathcal{M}[\![\mathtt{C}]\!]_\rho : \mathtt{State} \to \mathtt{State}$	*(Invocation)*
$\quad \mathcal{M}[\![\mathtt{Call\ I()}]\!]_\rho = \mathtt{find}(\mathtt{I}, \rho)$	
Auxiliary Domains	
$\mathtt{Den} ::= \mathtt{Loc} + \mathtt{ProcFun}$	*(Disjoint Union)*
$\mathtt{ProcFun} ::= \mathtt{State} \to \mathtt{State}$	*(Value Procedure)*
Auxiliary Functions	
$\mathtt{Q} : (\mathtt{State} \to \mathtt{State}) \to \mathtt{ProcFun}$	*(injective, costructor)*

Procedure with Static Scope: Implementation

$$\mathcal{M}[\![\mathtt{C}]\!]_\rho \quad vs. \quad (\mathtt{C}, \rho) = (p_C, p_\rho)$$

- **AR Stack**. An AR template, AR_I, is created for each defined procedure, with the features described in lecture3-6. Then, the 'find' operation is implemented by a:
  - backward visit, or
  - LeBlank-Cook's pair access

  to the frames of static chain of the Activation Records
- What is the relation between Static Chain and the ρ, above?

Block Structures Languages: Inline and Procedure/2

Table6 – Imperative Languages : Dynamic Scope	
Semantic Functions	
$\mathcal{D}[\![\mathtt{D}]\!]_\rho : Env$	(*Declarations*)
$\quad \mathcal{D}[\![\mathtt{Proc\ I()\ C}]\!]_\rho = \mathtt{bind}(\mathtt{I}, \lambda\delta.\mathcal{M}[\![\mathtt{C}]\!]_\delta, \rho)$	
$\mathcal{M}[\![\mathtt{C}]\!]_\rho : \mathtt{State} \rightarrow \mathtt{State}$	(*Invocation*)
$\quad \mathcal{M}[\![\mathtt{Call\ I()}]\!]_\rho = \mathtt{find}(\mathtt{I}, \rho)(\rho)$	

Procedure with Dynamic Scope: Implementation

$$\lambda\delta.\mathcal{M}[\![\texttt{C}]\!]_\delta \quad vs. \quad (\texttt{C},?) = (p_C,?)$$

- **AR Stack**. An AR template is created for each defined procedure, with the features described in lecture3-6. Then, the 'find' operation is implemented by a:
  - backward visit

  to the frames of the dynamic chain of the Activation Records

- is Dynamic Chain of a procedure known at the time of the proc. declaration (i.e. compile time)?
- is Dynamic Chain of a procedure known before the proc. invocation?
- is Dynamic Chain of a procedure the same for all the proc. invocation?

Block: Sequential, Parallel, Mixed, Declarations/1

Table7 – Languages : Sequential Declaration

Syntactic Domain

$\texttt{D} ::= \texttt{Proc I() C;} \mid \texttt{Const I} = \texttt{VL} \mid \texttt{D D}\ldots$

Semantic Functions

$\mathcal{D}[\![\texttt{Const I} = \texttt{VL}]\!]_\rho = \texttt{bind}(\texttt{I}, \texttt{IntoVal}(\texttt{VL}), \rho)$

$$
\begin{aligned}
&\texttt{Let}\ \ \mathcal{F} \in \{\mathcal{M}, \mathcal{E}\}\ \ \texttt{in} \\
&\quad \texttt{Let}\ \mathcal{D}[\![\texttt{D}_1]\!] = \lambda\sigma.\texttt{bind}(\texttt{I}_1, \mathcal{F}[\![d_1]\!]_\sigma, \sigma)\ \ \texttt{and} \\
&\quad\quad\quad \mathcal{D}[\![\texttt{D}_2]\!] = \lambda\sigma.\texttt{bind}(\texttt{I}_2, \mathcal{F}[\![d_2]\!]_\sigma, \sigma) \\
&\quad \texttt{in}\ \mathcal{D}[\![\texttt{D}_1\ \texttt{D}_2]\!]_\rho = \mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)) = \mathcal{D}[\![\texttt{D}_2]\!]_{\mathcal{D}[\![\texttt{D}_1]\!]_\rho}
\end{aligned}
$$

Auxiliary Domains

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{VL}$ (*Disjoint Union*)

Block: Sequential, Parallel, Mixed, Declarations/2

Table8 – Languages : Mutually Recursive Definitions

Syntactic Domain

$\texttt{D} ::= \ldots \mid \texttt{D D} \mid \texttt{Mut } \texttt{D}_1 \ \texttt{D}_2 \ \texttt{Ally} \mid \ldots$

Semantic Functions

$\texttt{Let} \ \ \mathcal{F} \in \{\mathcal{M}, \mathcal{E}\} \ \ \texttt{in}$

$$\begin{aligned}
&\texttt{Let } \mathcal{D}[\![\texttt{D}_1]\!] = \lambda\sigma.\texttt{bind}(\texttt{I}_1, \mathcal{F}[\![d_1]\!]_\sigma, \sigma) \ \ \texttt{and} \\
&\qquad \mathcal{D}[\![\texttt{D}_2]\!] = \lambda\sigma.\texttt{bind}(\texttt{I}_2, \mathcal{F}[\![d_2]\!]_\sigma, \sigma) \\
&\texttt{in } \mathcal{D}[\![\texttt{D}_1 \ \texttt{D}_2]\!]_\rho = \mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)) = \mathcal{D}[\![\texttt{D}_2]\!]_{\mathcal{D}[\![\texttt{D}_1]\!]_\rho}
\end{aligned}$$

$$\begin{aligned}
&\texttt{Let } \mathcal{D}[\![\texttt{D}_1]\!] = \lambda\sigma.\lambda\mu.\texttt{bind}(\texttt{I}_1, \mathcal{F}[\![d_1]\!]_\mu, \sigma) \ \ \texttt{and} \\
&\qquad \mathcal{D}[\![\texttt{D}_2]\!] = \lambda\sigma.\lambda\mu.\texttt{bind}(\texttt{I}_2, \mathcal{F}[\![d_2]\!]_\mu, \sigma) \\
&\texttt{in } \mathcal{D}[\![\texttt{Mut } \texttt{D}_1 \ \texttt{D}_2 \ \texttt{Ally}]\!]_\rho = \texttt{Y}\mu.\mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu))
\end{aligned}$$

Block: Sequential, Parallel, Mixed, Declarations/3

- Apply the definitions to the declaration below, in the example:

Example

Let A and B two identifiers. Show the bindings of A and B that the following fragment defines:

```
      ...
{...
 Mut
   Proc A() {Call B();}
   Proc B() {Call A();}
 Ally
...
```

$\mathtt{g} \equiv \mathtt{Y}\mu.\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_{\mu}, \sigma)(\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{A}, \ldots)(\rho)(\mu))$

Compute the first 3 approximations to the solution of the functional:

$\mathtt{H} \equiv \lambda\mu.\mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_{\mu}, \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\{\mathtt{Call\ B();}\}]\!]_{\mu}, \rho))$

At the starting step:

$$\begin{aligned} \mathtt{Y\ H}^0 &= \mathtt{H}(\bot) \\ &= \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_{\mu}, \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\{\mathtt{Call\ B();}\}]\!]_{\mu}, \rho)) \\ \mathtt{Y\ H}^1 &= \mathtt{H}(\mathtt{Y\ H}^0) \\ &= \ldots \\ \mathtt{Y\ H}^2 &= \mathtt{H}(\mathtt{Y\ H}^1) \\ &= \ldots \end{aligned}$$

Block: Sequential, Parallel, Mixed, Declarations: Use and Implementation

- **Use:** Declarations (of any of the 3 forms) is the elective, transparent mechanism that Programming Languages use to allow *Naming*
  - Sequential. Total ordering of the dependencies (references) among declarations
  - Parallel. Mutually Recursive Definitions (of procedure, functions, ..., types, classes)
  - Mixed. It Combines the two above: Explicitly (construct mutually) or implicitly (it is sequential except for procedures, functions,...)
- **Implementation:** It needs Env (in dynamic semantic) and requires Static Analyzers (in static semantics).

Block: Sequential, Parallel, Mixed, Declarations: Static Semantics

- Static Semantics = Rules that restrict the structures of syntactically correct (program) terms
- Static Analysis checks declarations against the use of circular definitions

Example

Show the environment Env when the semantics of mutually applies to the fragment

```
...
{...
 Mut
   int x=  y;
   int y=  3;
 Ally
...
```

Comment (also in view of the slide about mutually, undefined procedure calls)