A Virtual Machine Framework for Domain-Specific Languages

David Fick

Supervisor: Prof. B. Watson

Co-supervisor: Prof. D. Kourie

Domains

- A domain is characterized by a set of related concepts, which need to be understood, before they can be applied to real world problems. For example, Chemistry is a domain in which the concepts of elements, compounds, and their respective reactions, are among those that are to be understood by chemists if they are to be considered experts in the domain.

- In essence, a domain, from a developer's perspective, is simply a home where related concepts can live happily together.

Domain Specific Languages (DSLs)

- **Domain**
  - A set of classes, all connected via relationships of some multiplicity. Classes do not overlap between domains. A domain can be dependant on, or make use of services of other domains (namespaces).
- **DSL**
  - A Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to a particular domain.
- **DSL-program**
  - A program, written in the language of the DSL, i.e. the actual DSL program source code.

Why use DSLs?

- An expert already understands his/her domain and just needs a simple means to express solutions
- Intuitive coding style
- Far fewer lines of code than a generic language like, say C++

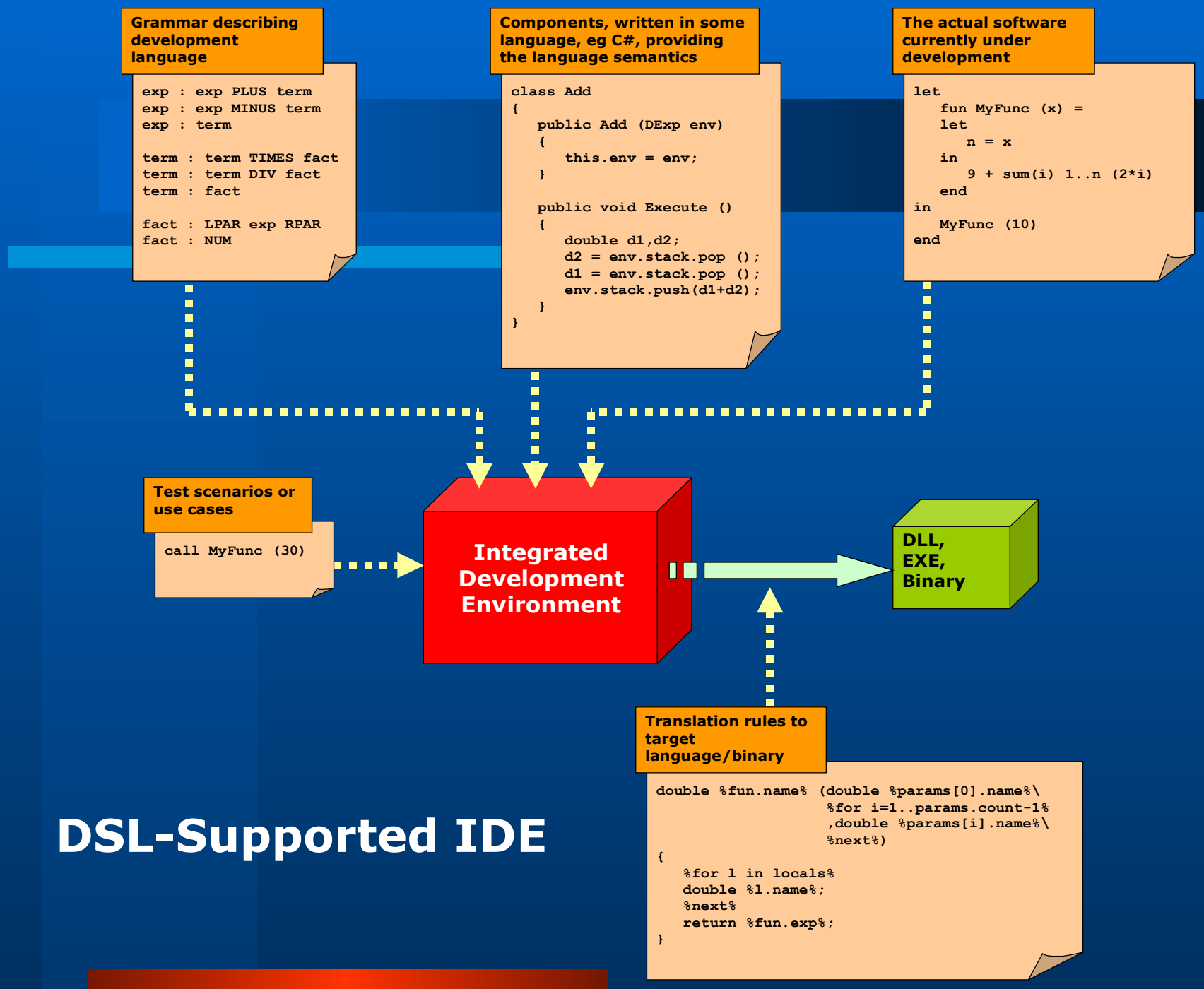
Techniques and criteria for each technique

- **DSL techniques**
  - Interpreter
  - Hard-coded VM
  - VM Framework
- **Criteria for Evaluation**
  - Scalability
  - Performance
  - Modularity

Computer-Aided Software Engineering (CASE)

- **CASE Tools provide sophisticated IDEs (Integrated Development Environments)**
- **Examples include BridgePoint, Borland Delphi, Borland C++Builder, Borland JBuilder, Microsoft's Visual C++ and Eclipse.**
- **Visualize design-time structure, UML, state machines, and even visualize runtime structure, such as stack traces, running threads.**

DSL-Supported IDE



DSL-Supported IDE

Grammar describing development language

```
exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM
```

Components, written in some language, eg C#, providing the language semantics

```
class Add
{
   public Add (DExp env)
   {
      this.env = env;
   }

   public void Execute ()
   {
      double d1,d2;
      d2 = env.stack.pop ();
      d1 = env.stack.pop ();
      env.stack.push(d1+d2);
   }
}
```

The actual software currently under development

```
let
   fun MyFunc (x) =
   let
      n = x
   in
      9 + sum(i) 1..n (2*i)
   end
in
   MyFunc (10)
end
```

Test scenarios or use cases

```
call MyFunc (30)
```

Integrated Development Environment

DLL, EXE, Binary

Translation rules to target language/binary

```
double %fun.name% (double %params[0].name%\
                   %for i=1..params.count-1%
                   ,double %params[i].name%\
                   %next%)
{
   %for l in locals%
   double %l.name%;
   %next%
   return %fun.exp%;
}
```

DSL-Supported IDE

Grammar describing development language

```
exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM
```

Components, written in some language, eg C#, providing the language semantics

```
class Add
{
   public Add (DExp env)
   {
      this.env = env;
   }

   public void Execute ()
   {
      double d1,d2;
      d2 = env.stack.pop ();
      d1 = env.stack.pop ();
      env.stack.push(d1+d2);
   }
}
```

The actual software currently under development

```
let
   fun MyFunc (x) =
   let
      n = x
   in
      9 + sum(i) 1..n (2*i)
   end
in
   MyFunc (10)
end
```

Test scenarios or use cases

```
call MyFunc (30)
```

Integrated Development Environment

DLL, EXE, Binary

Translation rules to target language/binary

```
double %fun.name% (double %params[0].name%\
                   %for i=1..params.count-1%
                   ,double %params[i].name%\
                   %next%)
{
   %for l in locals%
   double %l.name%;
   %next%
   return %fun.exp%;
}
```

DSL-Supported IDE

Grammar describing development language

```
exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM
```

Components, written in some language, eg C#, providing the language semantics

```
class Add
{
   public Add (DExp env)
   {
      this.env = env;
   }

   public void Execute ()
   {
      double d1,d2;
      d2 = env.stack.pop ();
      d1 = env.stack.pop ();
      env.stack.push(d1+d2);
   }
}
```

The actual software currently under development

```
let
   fun MyFunc (x) =
   let
      n = x
   in
      9 + sum(i) 1..n (2*i)
   end
in
   MyFunc (10)
end
```

Test scenarios or use cases

```
call MyFunc (30)
```

Integrated Development Environment

DLL, EXE, Binary

Translation rules to target language/binary

```
double %fun.name% (double %params[0].name%\
                   %for i=1..params.count-1%
                   ,double %params[i].name%\
                   %next%)
{
   %for l in locals%
   double %l.name%;
   %next%
   return %fun.exp%;
}
```

DSL-Supported IDE

Grammar describing development language

```
exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM
```

Components, written in some language, eg C#, providing the language semantics

```
class Add
{
   public Add (DExp env)
   {
      this.env = env;
   }

   public void Execute ()
   {
      double d1,d2;
      d2 = env.stack.pop ();
      d1 = env.stack.pop ();
      env.stack.push(d1+d2);
   }
}
```

The actual software currently under development

```
let
   fun MyFunc (x) =
   let
      n = x
   in
      9 + sum(i) 1..n (2*i)
   end
in
   MyFunc (10)
end
```

Test scenarios or use cases

```
call MyFunc (30)
```

Integrated Development Environment

DLL, EXE, Binary

Translation rules to target language/binary

```
double %fun.name% (double %params[0].name%\
                   %for i=1..params.count-1%
                   ,double %params[i].name%\
                   %next%)
{
   %for l in locals%
   double %l.name%;
   %next%
   return %fun.exp%;
}
```

DSL-Supported IDE

Grammar describing development language

```
exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM
```

Components, written in some language, eg C#, providing the language semantics

```
class Add
{
   public Add (DExp env)
   {
      this.env = env;
   }

   public void Execute ()
   {
      double d1,d2;
      d2 = env.stack.pop ();
      d1 = env.stack.pop ();
      env.stack.push(d1+d2);
   }
}
```

The actual software currently under development

```
let
   fun MyFunc (x) =
   let
      n = x
   in
      9 + sum(i) 1..n (2*i)
   end
in
   MyFunc (10)
end
```

Test scenarios or use cases

```
call MyFunc (30)
```

Integrated Development Environment

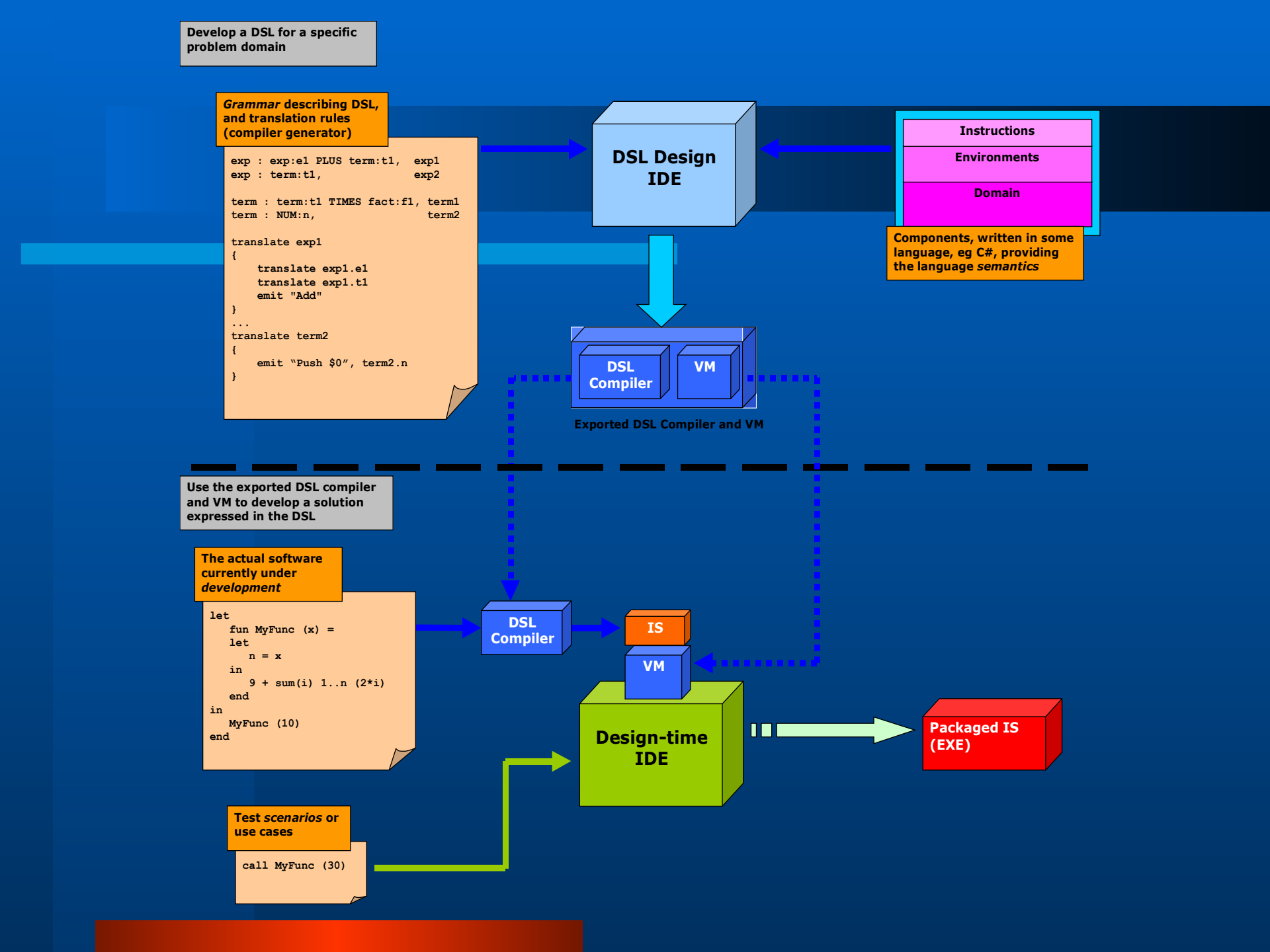
DLL, EXE, Binary

Translation rules to target language/binary

```
double %fun.name% (double %params[0].name%\
                   %for i=1..params.count-1%
                   ,double %params[i].name%\
                   %next%)
{
   %for l in locals%
   double %l.name%;
   %next%
   return %fun.exp%;
}
```

A DSL Development IDE

- **Featuring a VM as part of the CASE Software, and has 2 parts:**
  - **Custom build a DSL, with syntax and semantics, and allow linking into a namespace written in a platform such as the .NET CLR.**
  - **The second part will allow the developer to write programs using the DSL that has been specifically customized for his or her particular need.**


Develop a DSL for a specific problem domain
Grammar describing DSL, and translation rules (compiler generator)
exp : exp:e1 PLUS term:t1, exp1
exp : term:t1, exp2
term : term:t1 TIMES fact:f1, term1
term : NUM:n, term2
translate exp1
{
translate exp1.e1
translate exp1.t1
emit "Add"
}
...
translate term2
{
emit "Push $0", term2.n
}
DSL Design IDE
Instructions
Environments
Domain
Components, written in some language, eg C#, providing the language *semantics*
DSL Compiler
VM
Exported DSL Compiler and VM
Use the exported DSL compiler and VM to develop a solution expressed in the DSL
The actual software currently under *development*
let
fun MyFunc (x) =
let
n = x
in
9 + sum(i) 1..n (2*i)
end
in
MyFunc (10)
end
DSL Compiler
IS
VM
Design-time IDE
Packaged IS (EXE)
Test *scenarios* or use cases
call MyFunc (30)

Some example DSLs

- *EXP* – a simple functional expression language
- *RT* – a ray-tracer scene description language
- *LQ* – Little Quilt. A functional language of quilts

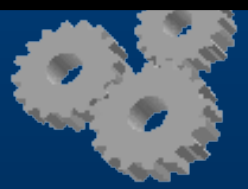
DSL Interpreter

- Conceptual view of a DSL interpreter

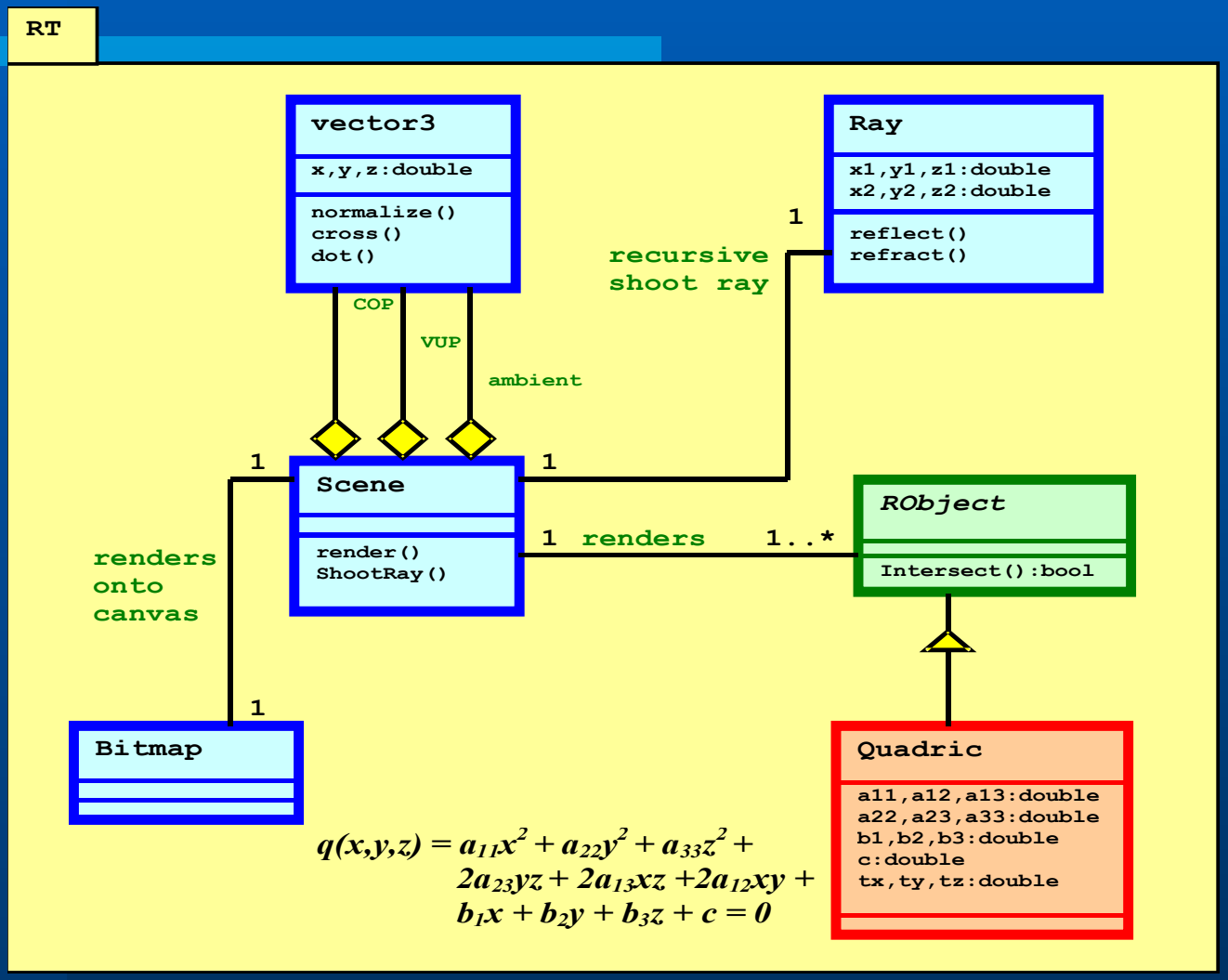
Source DSL

```
scene
{
    sphere 0.5
    {
        translate 3 3 0
        specular 0 0 1
    }
}
```

Statically Interpret DSL



The Domain of RT

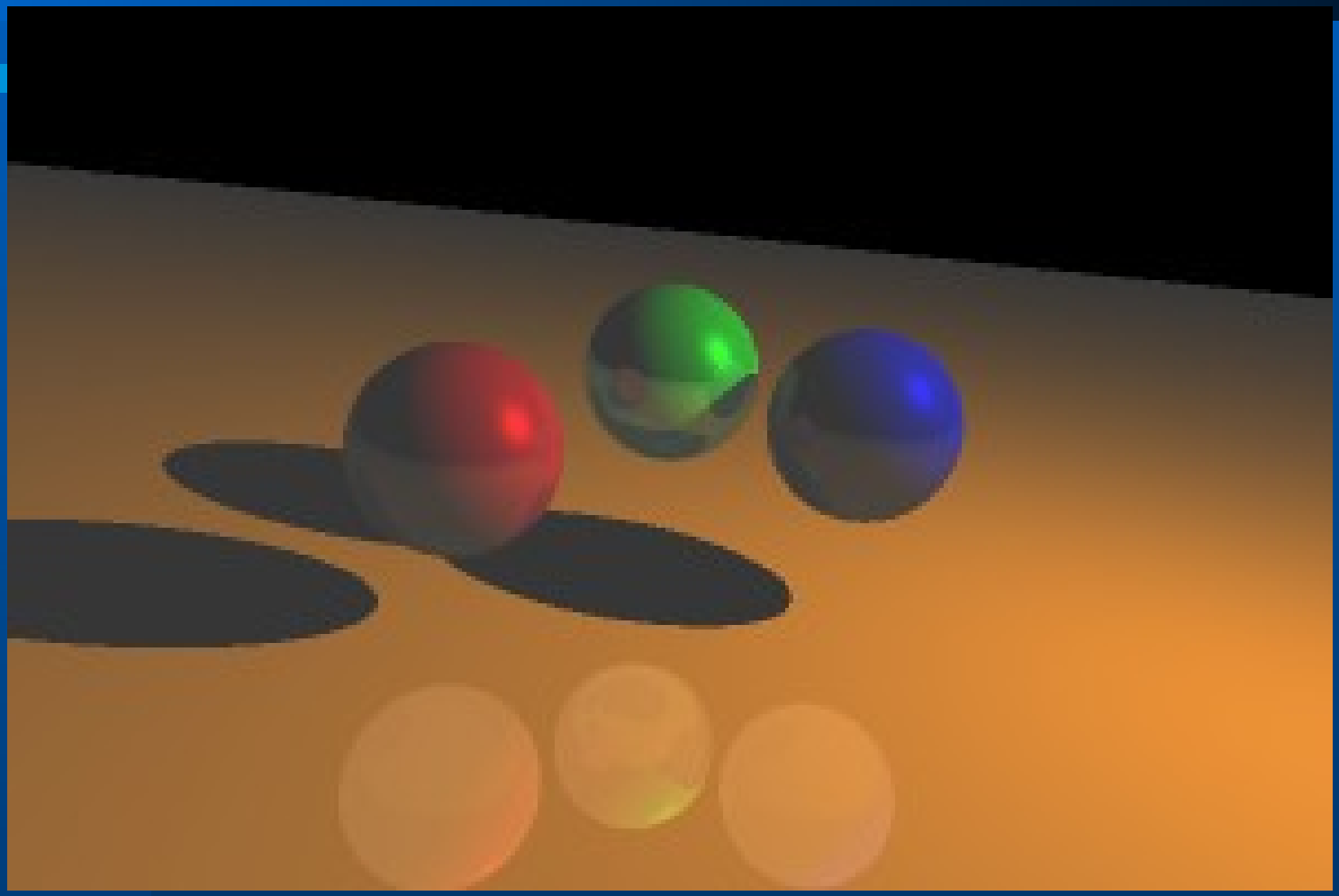


$$q(x,y,z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{23}yz + 2a_{13}xz + 2a_{12}xy + b_1x + b_2y + b_3z + c = 0$$

RT Example

```
scene {
     window 300.0 200.0  light -3.5 0.5 2.0
     cop 0.0 4.0 1.0     lookat 0.0 0.0 0.0
     viewup 0.0 0.0 1.0  ambient 0.3 0.3 0.3

     sphere 0.5 {
          translate 1.0 0.0 0.0   reflective 0.1 0.1 0.1
          specular 0.7 0.0 0.0    diffuse 0.7 0.0 0.0
     }

     sphere 0.5 {
          translate 0.0 -1.0 0.2  reflective 0.2 0.3 0.2
          specular 0.0 0.7 0.0    diffuse 0.0 0.7 0.0
     }

     sphere 0.5 {
          translate -1.0 -0.5 0.0 reflective 0.1 0.1 0.1
          specular 0.0 0.0 0.7    diffuse 0.0 0.0 0.7
     }

     plane -0.1 0.0 1.0 1.0 {
          specular 0.0 0.0 0.0    diffuse 1.0 0.5 0.0
          reflective 0.25 0.25 0.25
     }
}
```

RT Example



Adding the `ambient` Construct to an Interpreter

- **Appending the lexer definition for keyword `ambient`.**

```
%lexer
%namespace RtcLexer

%token                          DOUBLE {public double val;}
"{"                             %LBRACE
"}"                             %RBRACE
"scene"                         %SCENE
...
"viewup"                        %VIEWUP
"ambient"                       %AMBIENT
"specular"                      %SPECULAR
...
```

Adding the `ambient` Construct to an Interpreter

- **Altering the parser for the new `ambient` construct.**

```
...
%node scene_prop6 : scene_prop
{
        public double d1, d2, d3;
        public scene_prop6 (double d1,double d2,double d3)
               {this.d1=d1;this.d2=d2;this.d3=d3;}
}
...

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
             %scene_prop6 (d1.val,d2.val,d3.val);

prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp
...
```

Adding the `ambient` Construct to an Interpreter

- **Altering the functions for the new `ambient` construct.**

```
public class rtc
{
   public static void Main (string [] argv)
   {
      ...
   }

   ...

   public static void trans_scene_prop (RtcParser.scene_prop node)
   {
      if (node is RtcParser.scene_prop1)
      {
         trans_scene_prop1 ((RtcParser.scene_prop1) node);
      }
      ...
      else if (node is RtcParser.scene_prop6)
      {
         trans_scene_prop6 ((RtcParser.scene_prop6) node);
      }
   }

   ...
   public static void trans_scene_prop6 (RtcParser.scene_prop6 node)
   {
      scene.alterAmbient (node.d1, node.d2, node.d3);  // ambient
      scene.setupAxes ();
   }

   ...
   public static Scene scene;
   public static RObject robject;
   public static Bitmap bitmap;
   public static string filename;
}
```

DSL Hard-coded Virtual Machines

- Conceptual view of a DSL hard-coded VM

Source DSL

```
scene
{
   sphere 0.5
   {
      translate 3 3 0
      specular 0 0 1
   }
}
```

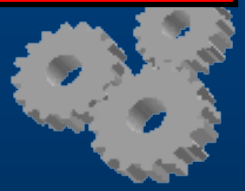
Compile

DSL Compiler

Generated Instruction Sequence

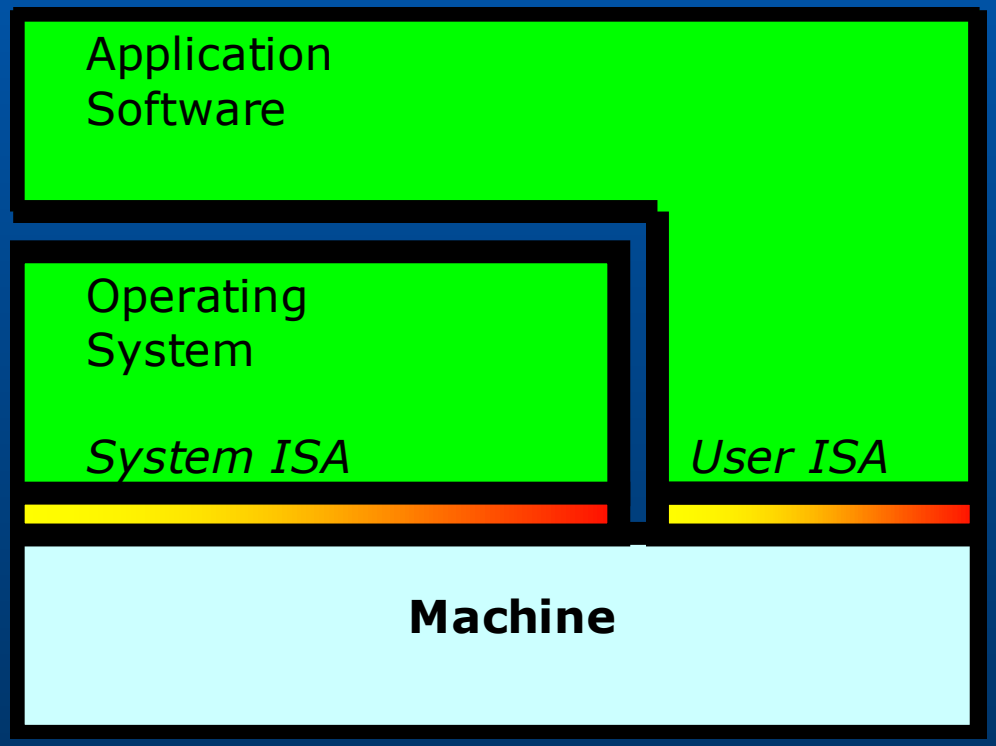
```
Quad
Push 1
Push 0.5
Push 0.5
Mul
Div
Coeff "a11"
```

Statically Interpret IS



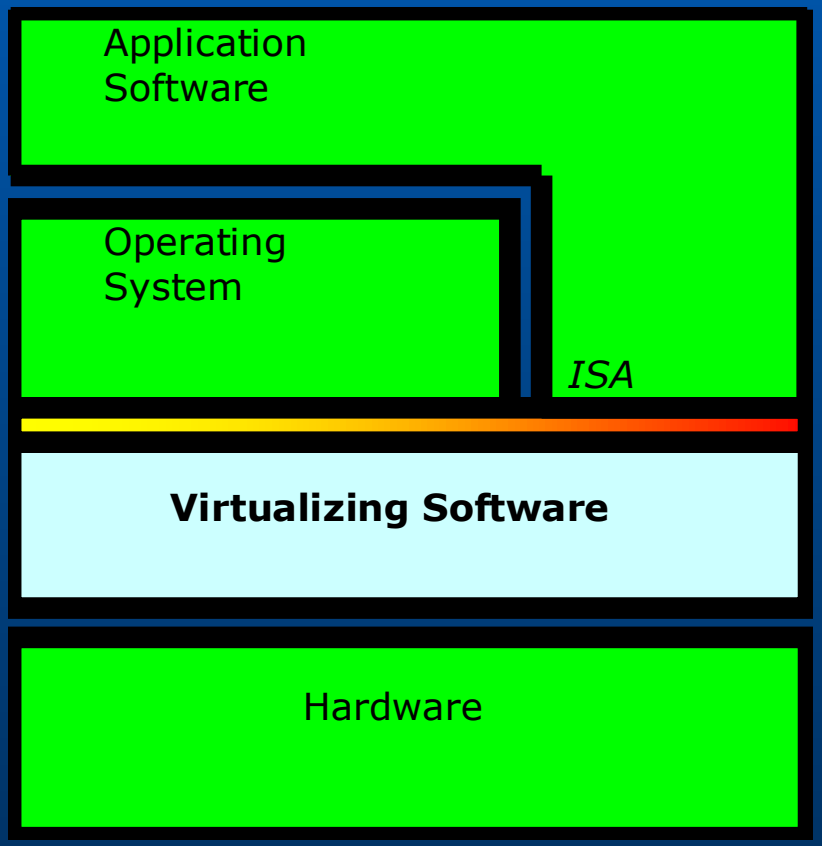
DSL Hard-coded Virtual Machines

- **Instruction Set Architectures - System ISA and User ISA**



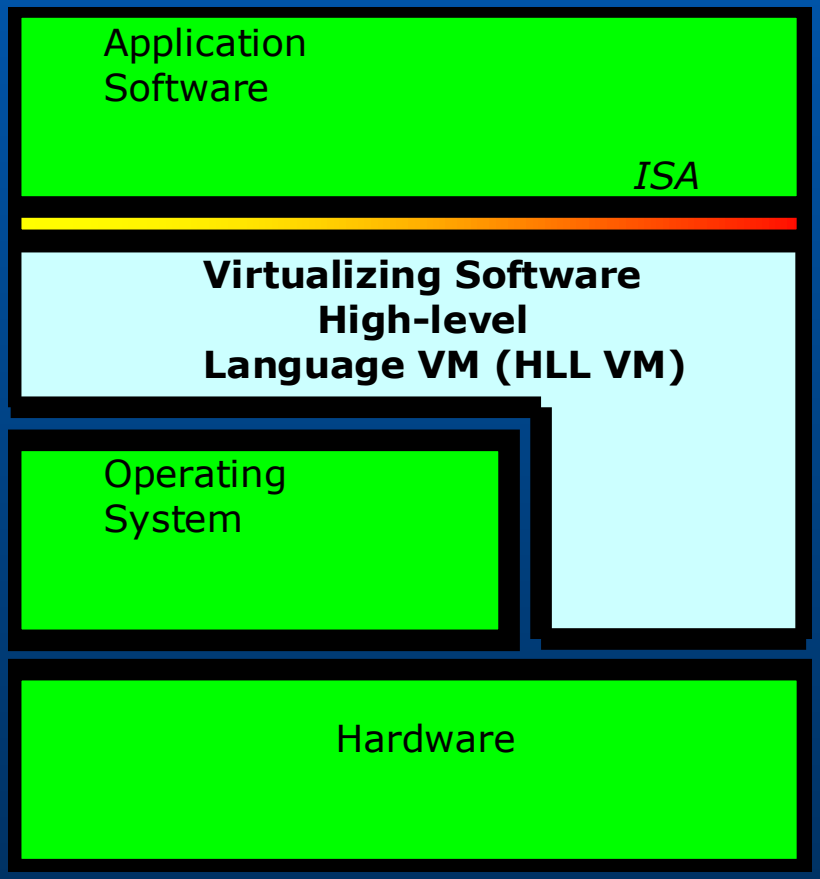
System VMs vs Process VMs

- **A System VM: Virtualizing software between hardware and OS**



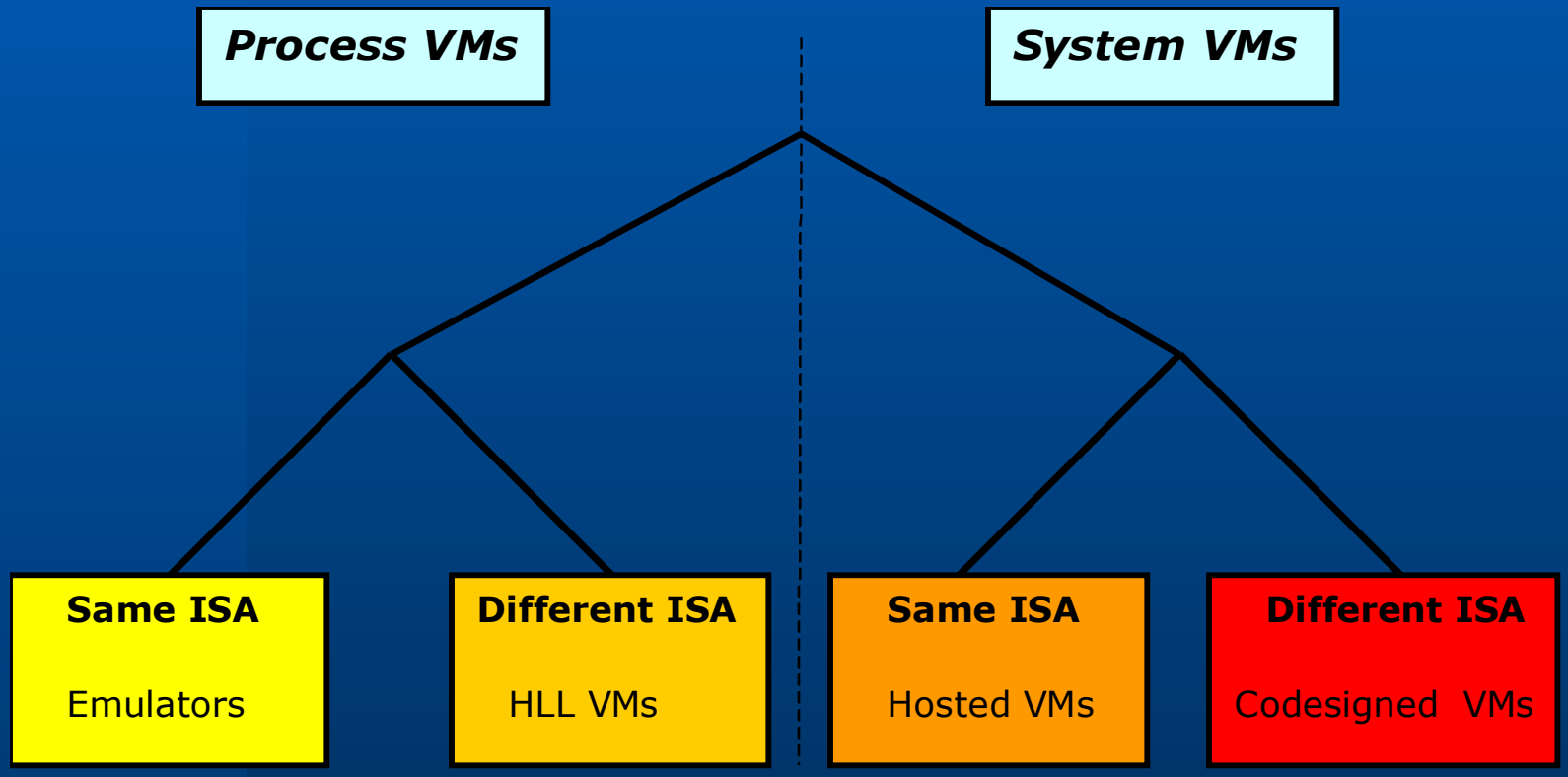
System VMs vs Process VMs

- **A Process VM: Virtualizing software supports an individual process**



System VMs vs Process VMs

- **Broad classification of VMs**

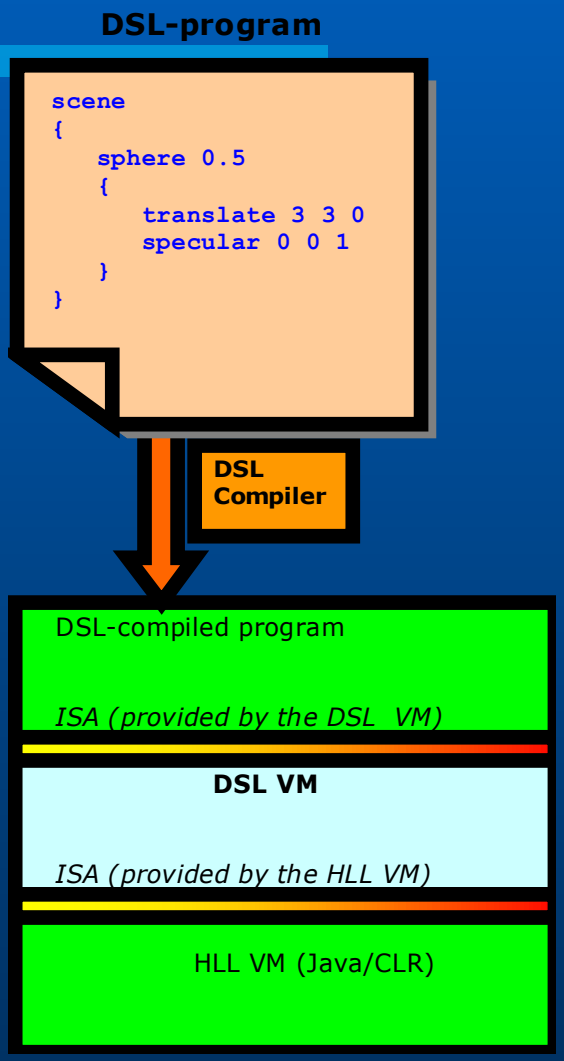


DSL VMs

- DSL VM : A concrete instance of a HLL VM, providing an ISA capable of supporting the semantics of some DSL, i.e. the semantics of a single instruction or combination of some sequence of instructions, will facilitate the semantics of the DSL, and thus will reflect the application domain for which the DSL was designed.

- DSL-compiled program : The DSL-program, once it has been compiled into a suitable ordered sequence of instructions ready to be executed on a DSL VM.

DSL VMs


DSL-program
scene
{
sphere 0.5
{
translate 3 3 0
specular 0 0 1
}
}
DSL Compiler
DSL-compiled program
ISA (provided by the DSL VM)
DSL VM
ISA (provided by the HLL VM)
HLL VM (Java/CLR)

DSL VMs

- **Lexer definition for the RT VM**

```
%lexer
%token                    DOUBLE {public double val;}
"PUSH"                    %PUSH
"SPHERE"                  %SPHERE
"PLANE"                   %PLANE
"WIN"                     %WIN
"LIGHT"                   %LIGHT
"COP"                     %COP
"LOOK"                    %LOOK
"VUP"                     %VUP
"AMB"                     %AMB
"GEOM"                    %GEOM
"SPEC"                    %SPEC
"DIFF"                    %DIFF
"REFL"                    %REFL
"REFR"                    %REFR
"AXES"                    %AXES
"REN"                     %REN
[-+]?[0-9]+(\.[0-9]+)?    %DOUBLE {val = Double.Parse (yytext);}
[ \n\r\f\t]               ;
```

DSL VMs

- **Parser definition for the RT VM**

```
asm_list :
asm_list : asm_list asm

asm : PUSH DOUBLE
asm : SPHERE
asm : PLANE
asm : WIN
asm : LIGHT
asm : COP
asm : LOOK
asm : VUP
asm : AMB
asm : GEOM
asm : SPEC
asm : DIFF
asm : REFL
asm : REFR
asm : AXES
asm : REN
```

Adding the `ambient` Construct to a Hard-coded DSL VM

- **Altering the grammar for the new `ambient` construct on the RT compiler**

```
...
%node scene_prop6 : scene_prop
{
        public double d1, d2, d3;
        public scene_prop6 (double d1,double d2,double d3)
               {this.d1=d1;this.d2=d2;this.d3=d3;}
        public override void translate (StreamWriter asmFile)
        {
                asmFile.WriteLine ("Push\t{0}", d1);
                asmFile.WriteLine ("Push\t{0}", d2);
                asmFile.WriteLine ("Push\t{0}", d3);
                asmFile.WriteLine ("Ambient");
                asmFile.WriteLine ("Axes\n");
        }
}

...

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
             %scene_prop6 (d1.val,d2.val,d3.val);

prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp
...
```

Adding the `ambient` Construct to a Hard-coded DSL VM

- **Adding the `Ambient` instruction to the VM's lexer definition**

```
%token                    DOUBLE {public double val;}
...
"Window"                  %WIN
"Light"                   %LIGHT
"COP"                     %COP
"LookAt"                  %LOOK
"VUP"                     %VUP
"Ambient"                 %AMB
...
```

Adding the `ambient` Construct to a Hard-coded DSL VM

- **Parsing the new `Ambient` instruction**

```
...
%node asm9_amb : asm
{
        public override void interpret (RTVM.RTMachine rtm)
        {
                rtm.RT_AMB ();
        }
}

...

asm_list :                      %asm_list1 (null, null);
asm_list : asm_list:al asm:a    %asm_list1 (al, a);

asm : PUSH DOUBLE:d             %asm1_push (d.val);
asm : SPHERE                    %asm2_sphere ();
asm : PLANE                     %asm3_plane ();
asm : WIN                       %asm4_win ();
asm : LIGHT                     %asm5_light ();
asm : COP                       %asm6_cop ();
asm : LOOK                      %asm7_look ();
asm : VUP                       %asm8_vup ();
asm : AMB                       %asm9_amb ();
asm : GEOM                      %asm10_geom ();
...
```

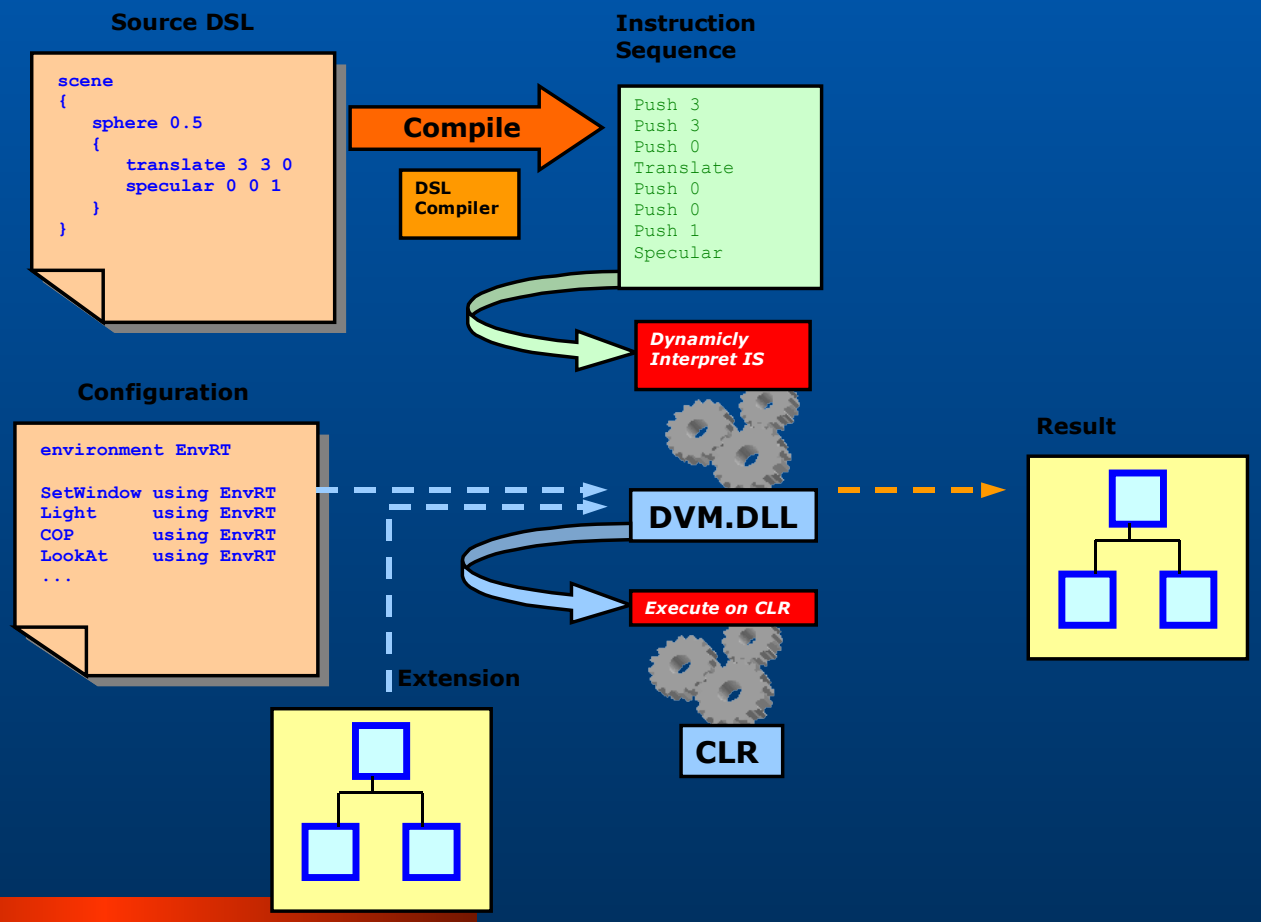
Adding the `ambient` Construct to a Hard-coded DSL VM

- **The semantics for the new `Ambient` instruction in function `RT_AMB()`**

```
public class RTMachine
{
        public RTMachine (int rd, int stackSize)
        {
                rtstack = new RTStack (stackSize);
                scene = new Scene ();
                render_depth = rd;
        }

        ...

        // Alter the current ambient colour vector.
        public void RT_AMB ()
        {
                double amb = (double) rtstack.pop ();
                double amg = (double) rtstack.pop ();
                double amr = (double) rtstack.pop ();
                scene.alterAmbient (amr, amg, amb);
        }

        ...
        private asm_list ast;
        public RTStack rtstack;
        public Scene scene;
        public int render_depth;
}
```

DSLs using the VM Framework

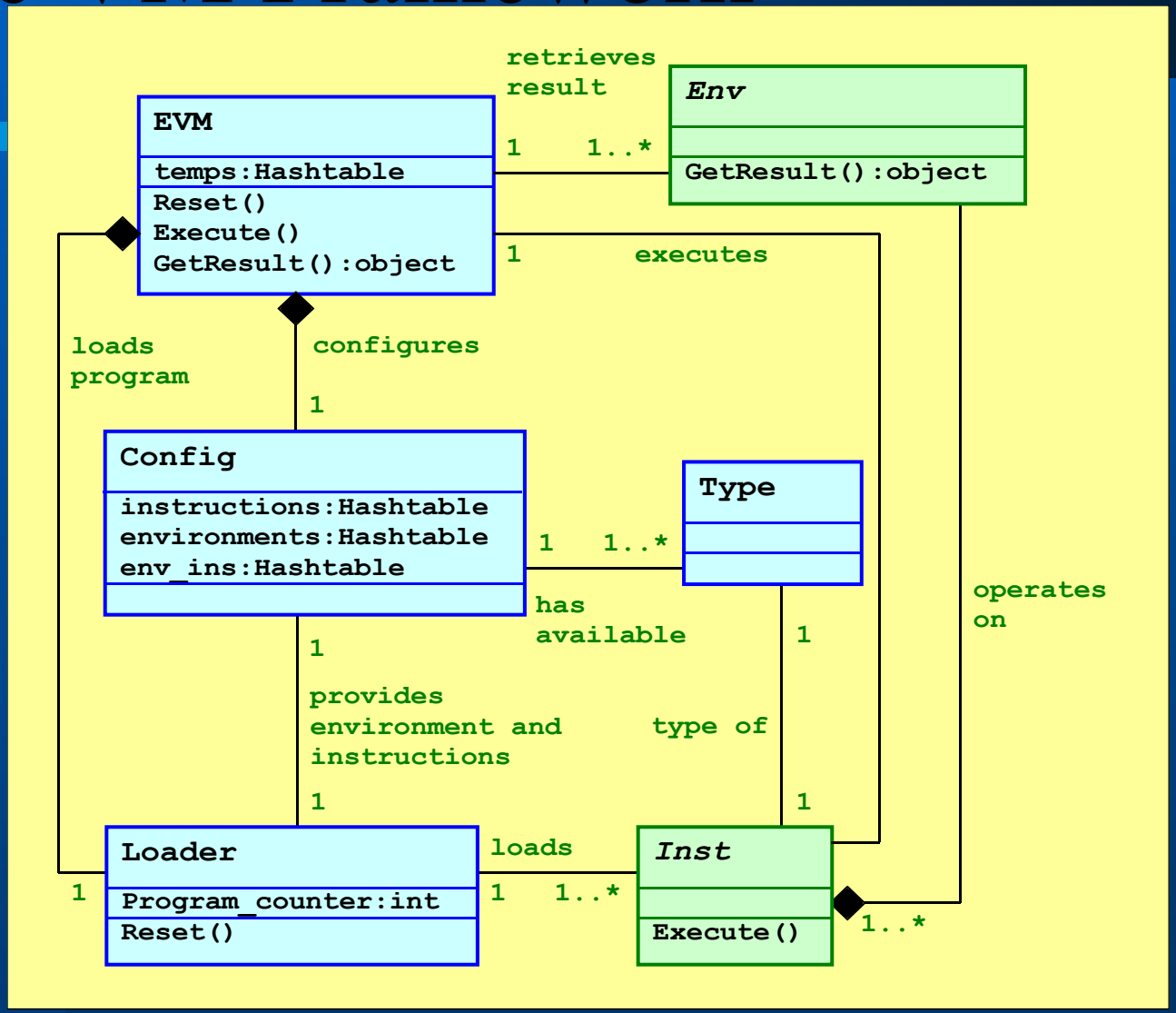
- Conceptual view of a VM using the VM Framework



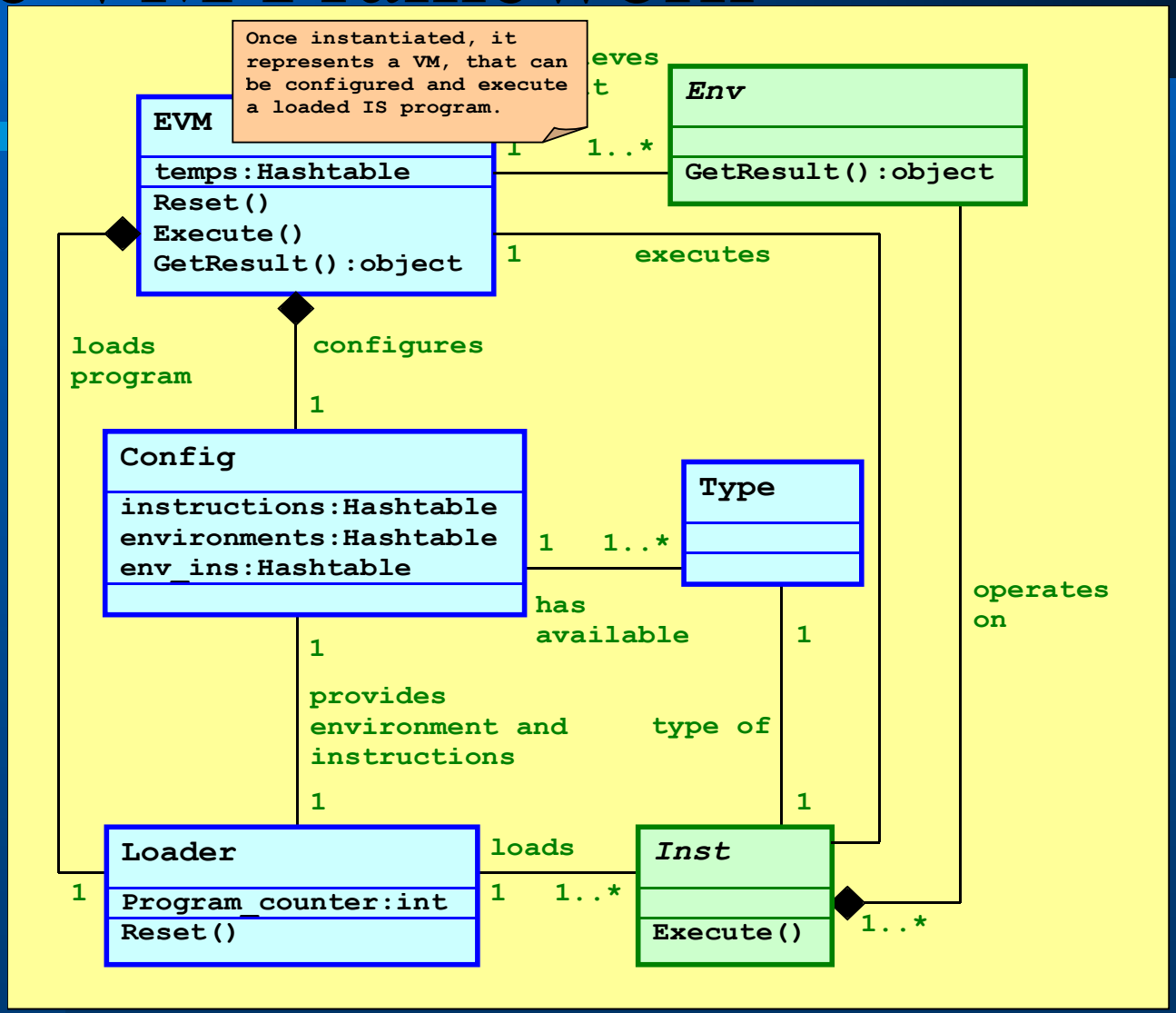
The VM Framework

- Designed on the .NET environment, written in C#
- The VM Framework provides a Virtual Machine with an empty instruction set
- The VM is extended by a set of concrete instructions and environments to facilitate the semantics of a domain

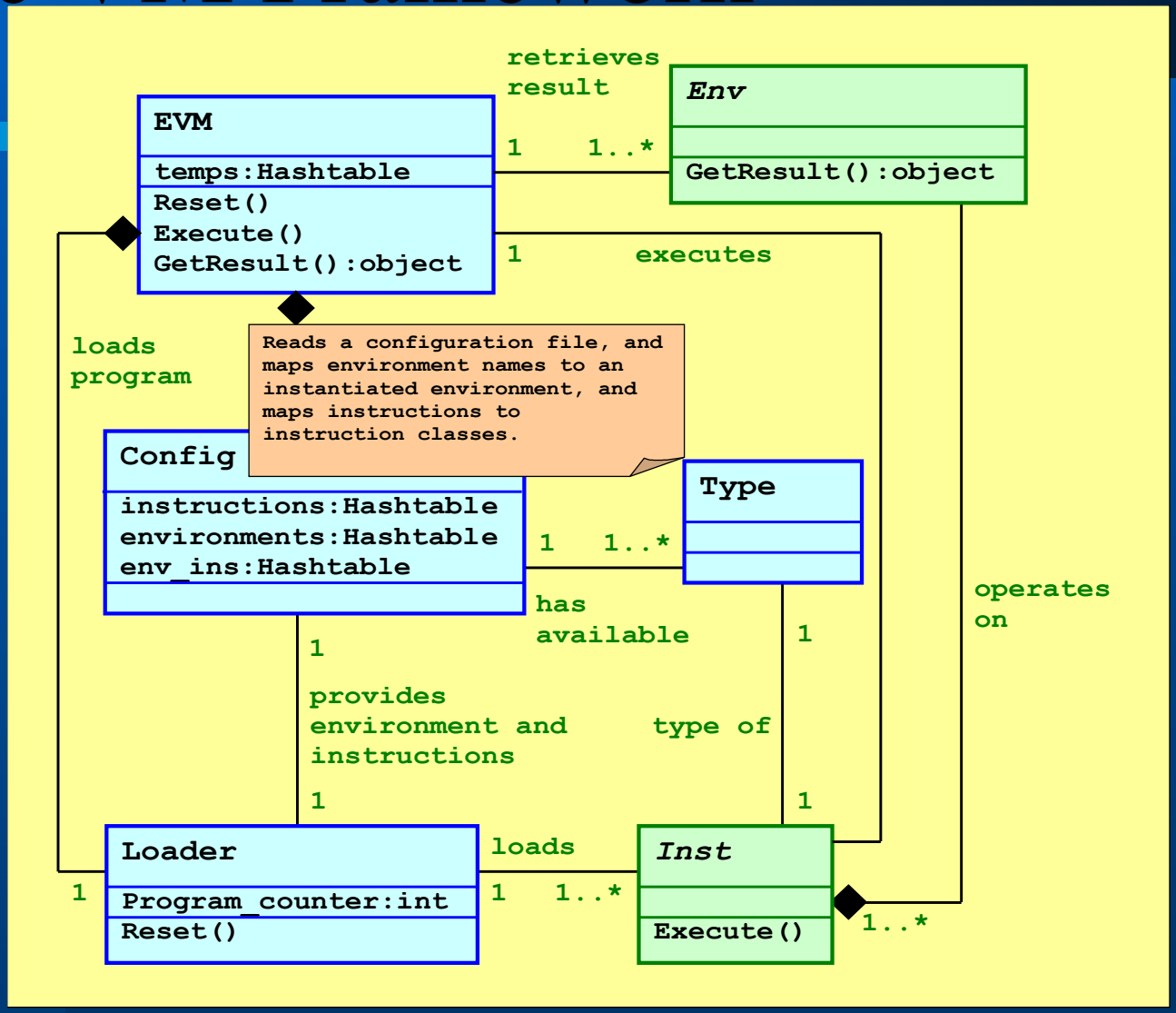
The VM Framework



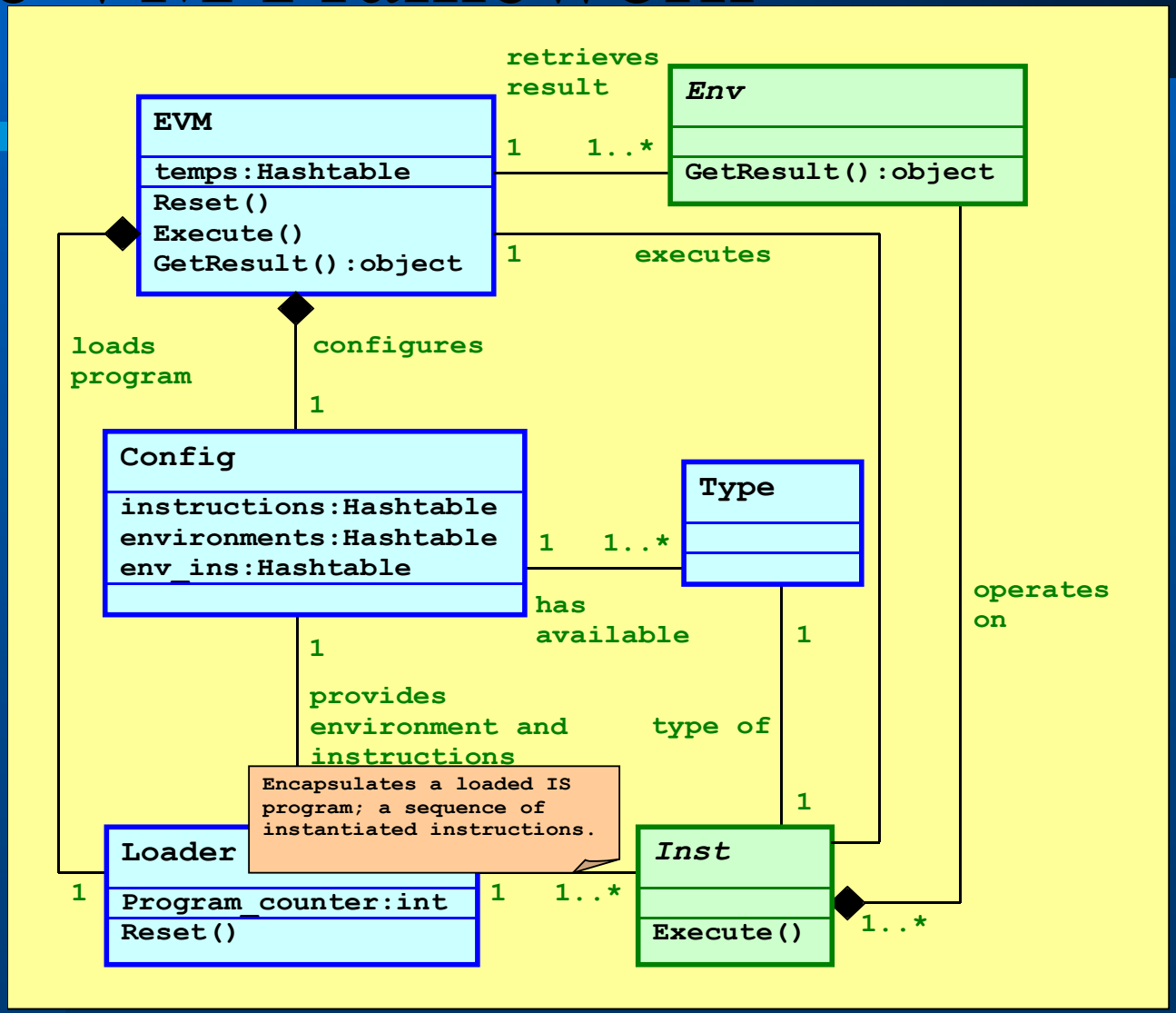
The VM Framework



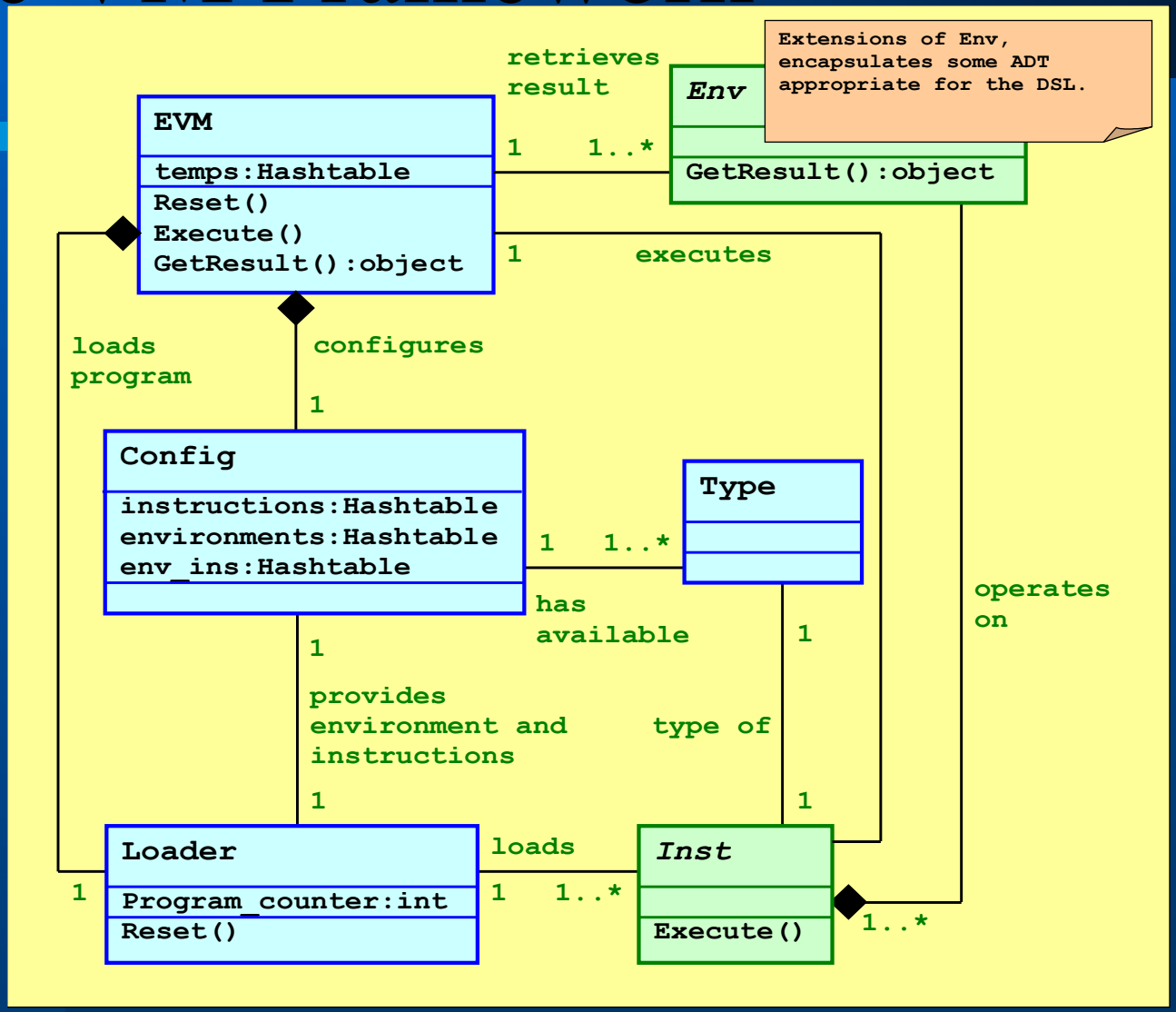
The VM Framework



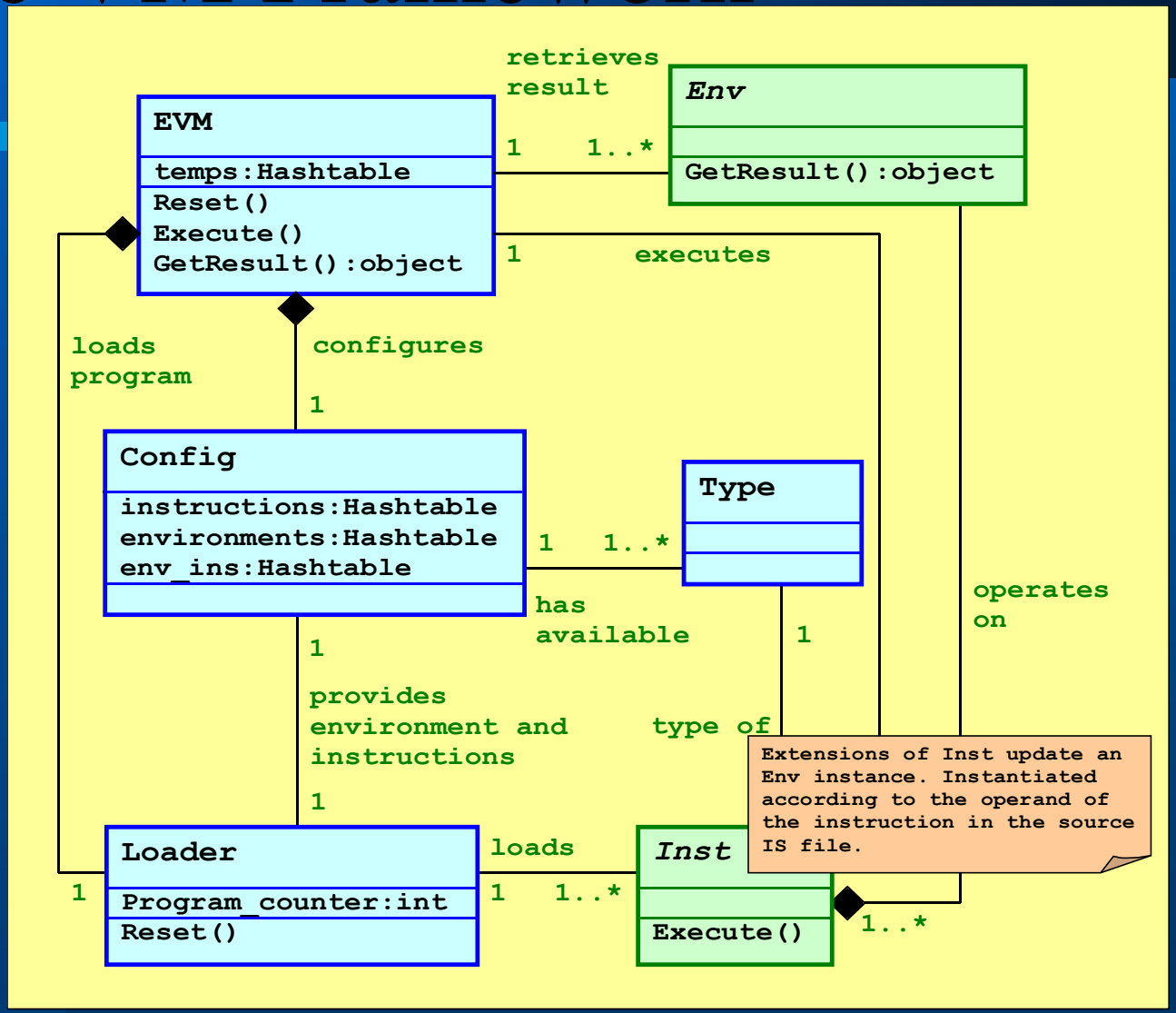
The VM Framework



The VM Framework



The VM Framework



The VM Framework

- **Environments**
  - An ADT updated during runtime.
  - Intended to aid in the processing of the semantics of the DSL in an appropriate or convenient manner. For example, in a simple real-valued expression language can use a runtime stack.
  - Classes that extend the abstract `Env` class, are obligated to override the method `GetResult()`.

The VM Framework

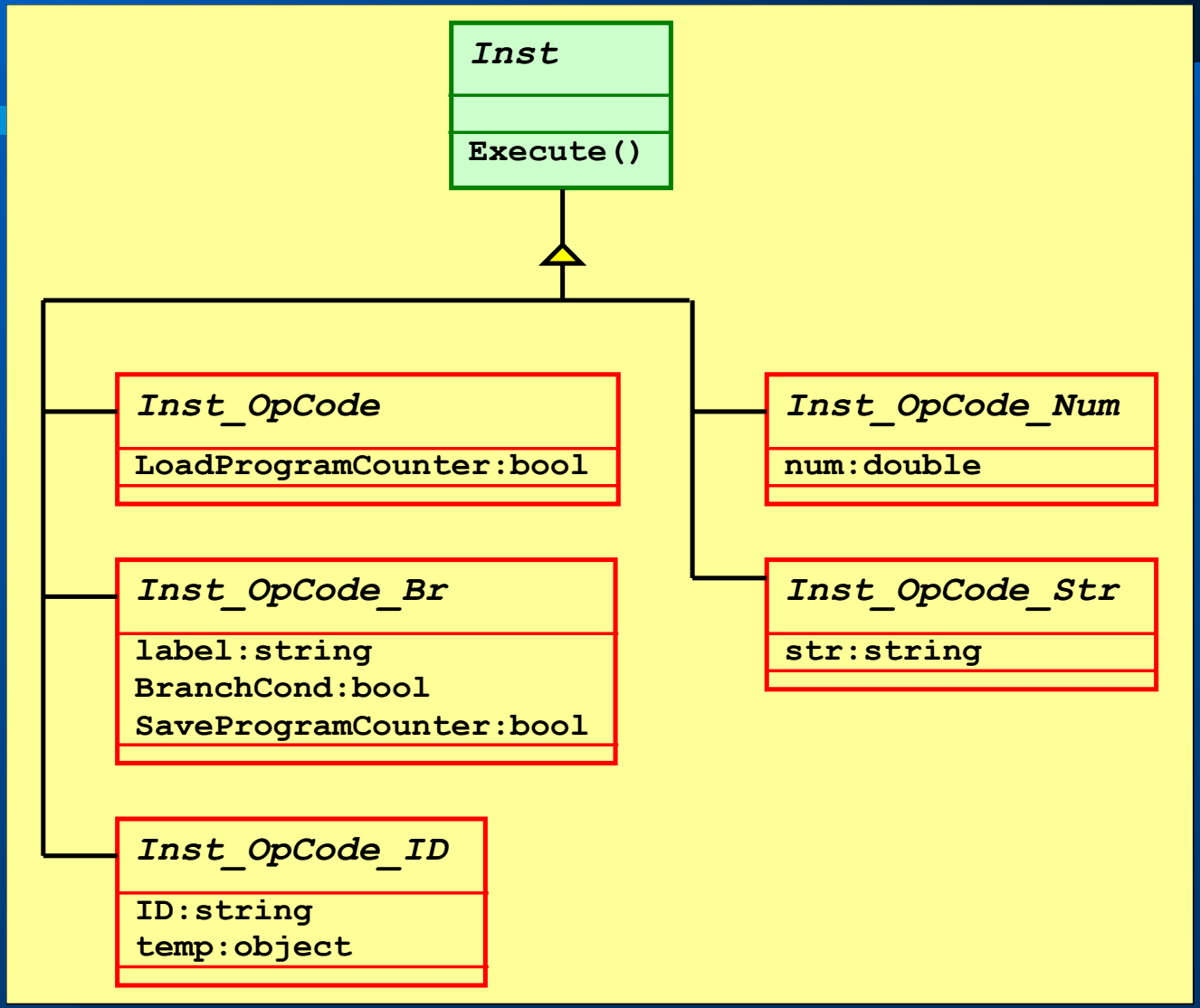
- **Example `EnvExp` class**

```
class EnvExp : Env
{
   public EnvExp () {
      stack = new Stack (100);
   }
   public override object GetResult () {
      object result;
      if (stack.isEmpty ()) {
         result = -1.0;
      }
      else {
         result = stack.peek ();
      }
      return result;
   }
   public Stack GetStack(){return stack;}
   protected Stack stack;
}
```

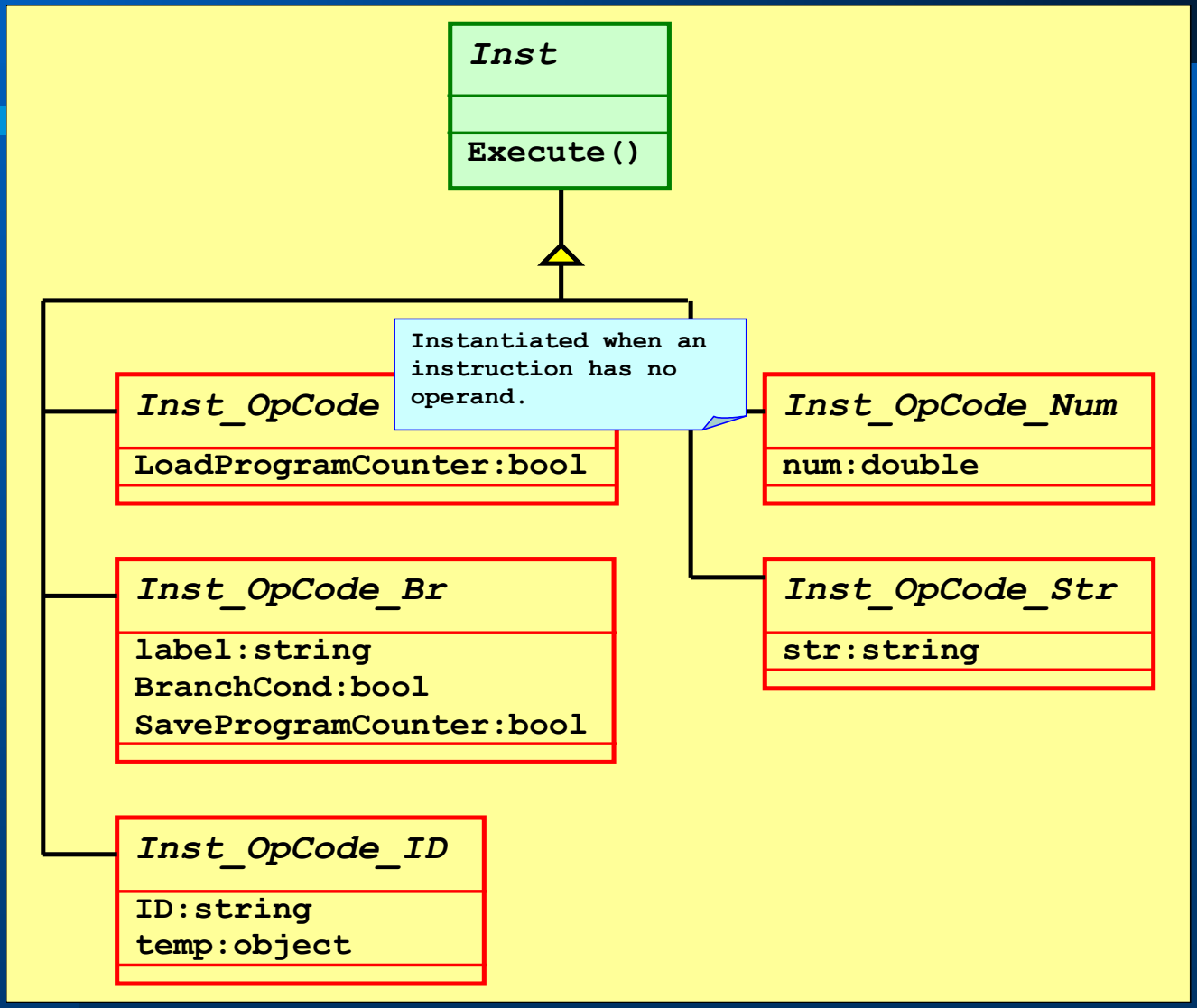
The VM Framework

- **Instructions**
  - **There are 5 classes of instructions, defined in terms of their operand types.**
  - **The instructions form the basic operational building blocks to a DSL-compiled program.**
  - **Each instruction is allowed to take at most one operand.**
  - **Classes that extend one of the 5 instruction classes, are obligated to override the method `Execute()`.**

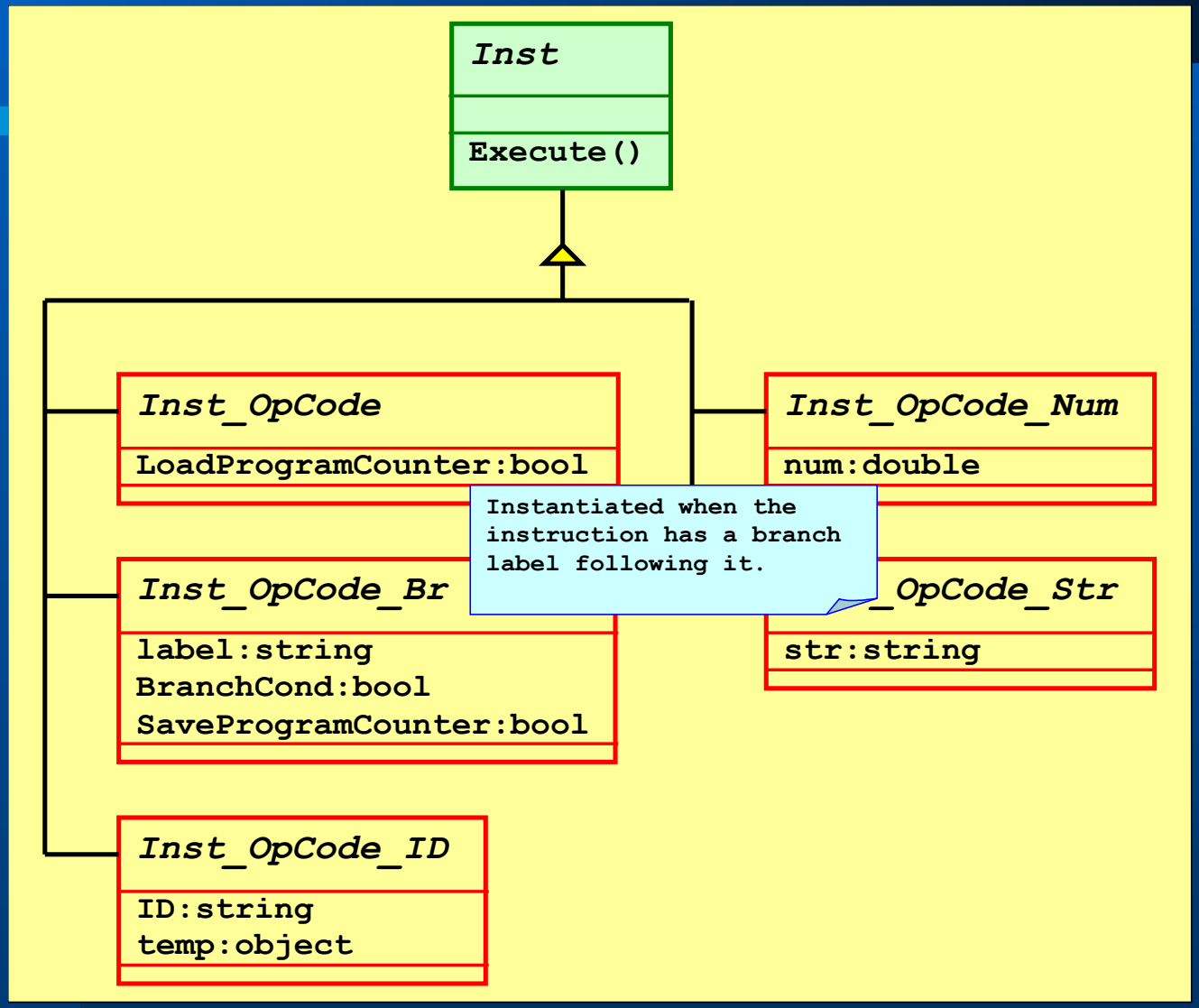
The Five Classes of Instructions



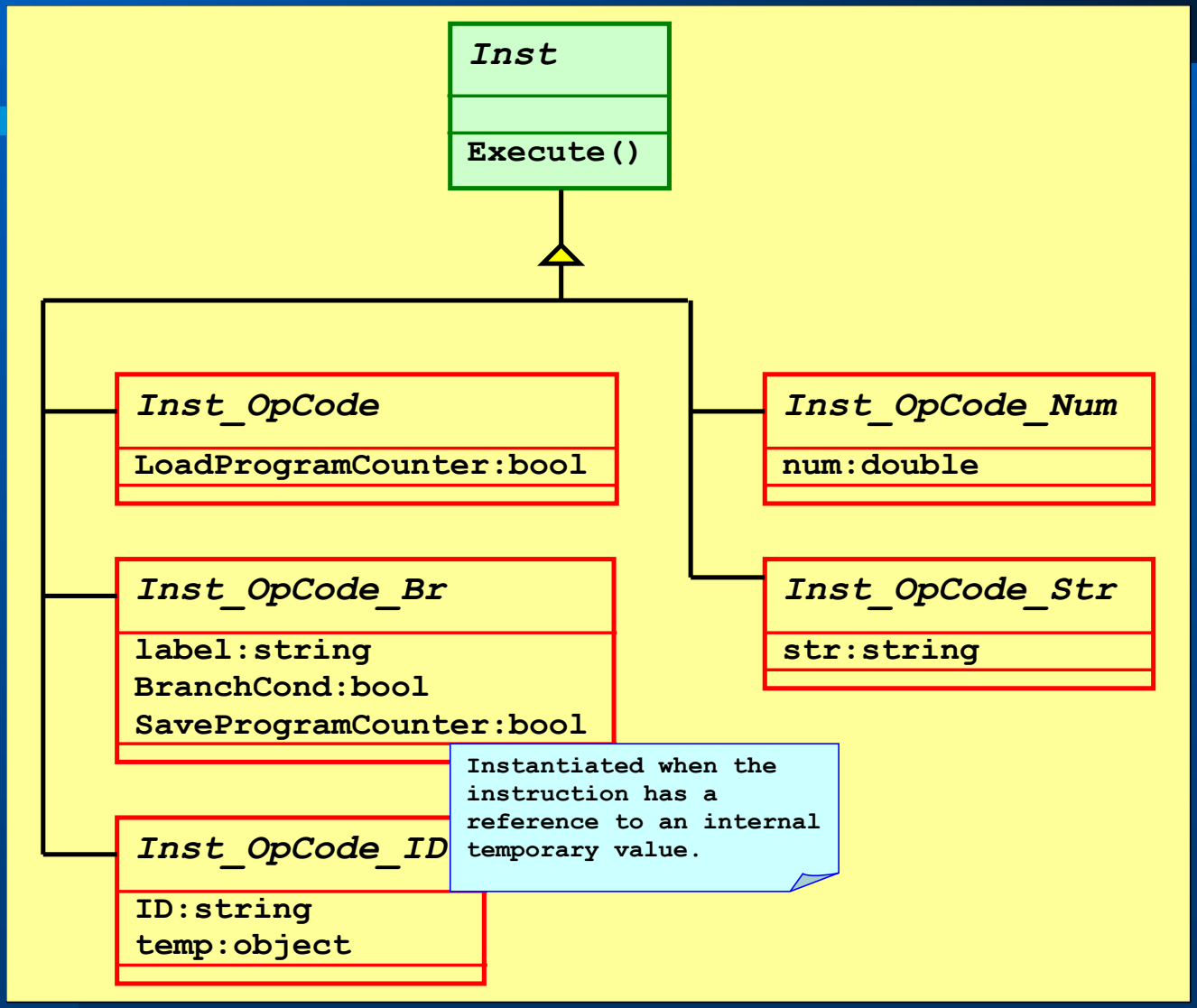
The Five Classes of Instructions



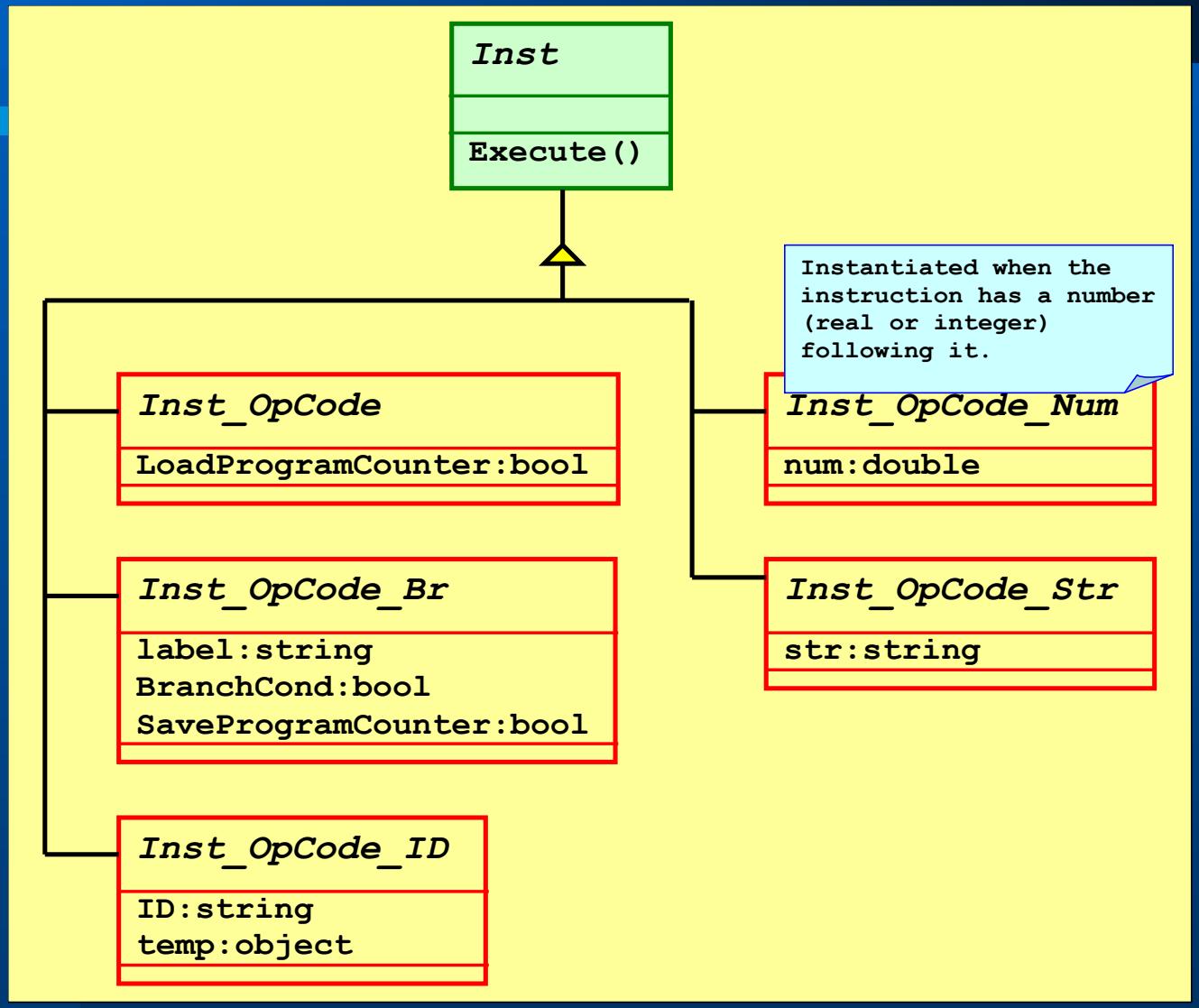
The Five Classes of Instructions



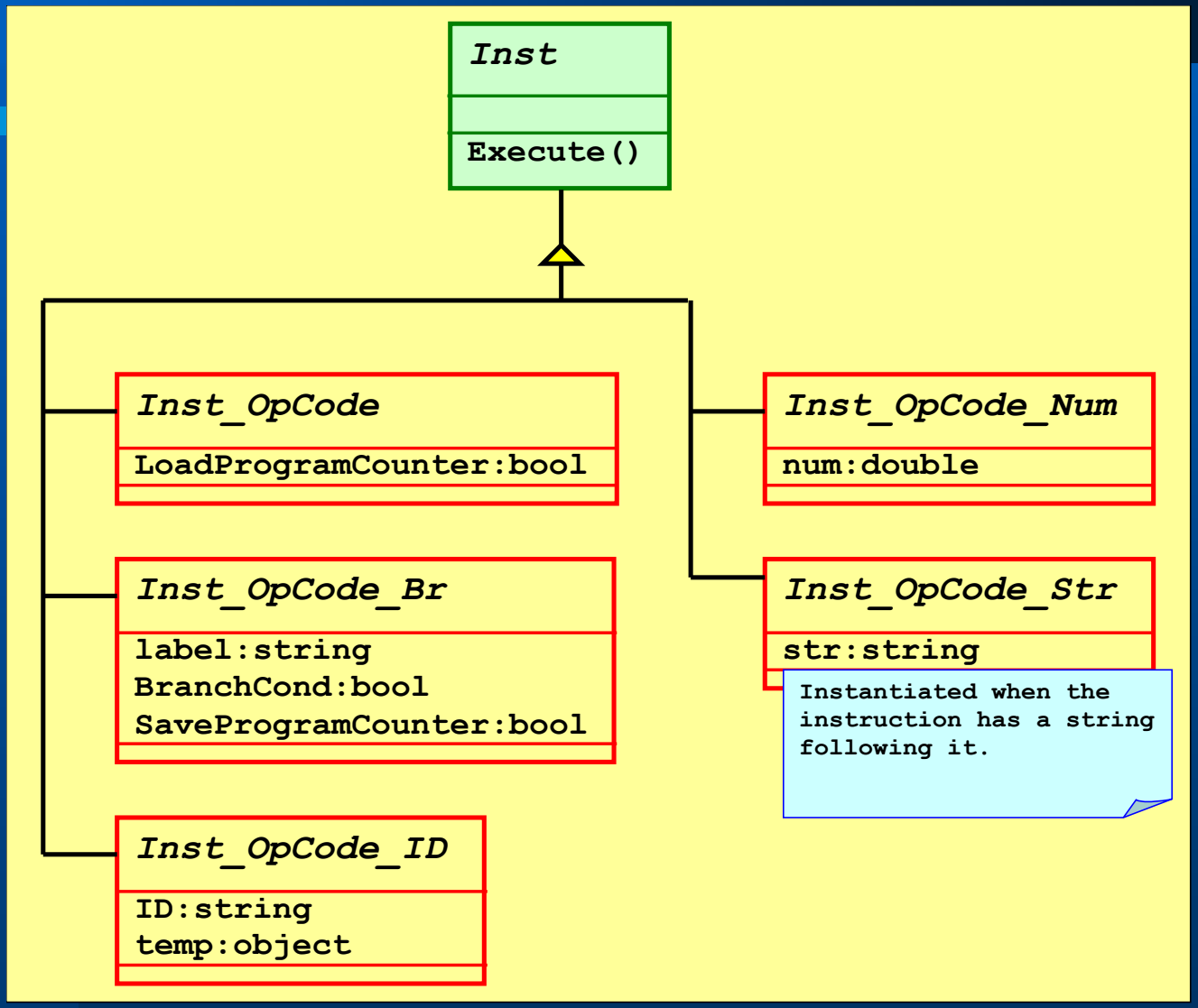
The Five Classes of Instructions



The Five Classes of Instructions



The Five Classes of Instructions



Extending the VM Framework

- **The Configuration File**
  - **The config file defines an ISA for a particular VM instance, in essence defining an instruction pool, each instruction to be later instantiated by loading a DSL-compiled program.**
  - **For each instruction specified in the config file, there needs to be a compiled DLL module on disk with the same name, which contains the compiled instruction class.**

Extending the VM Framework

- **Example configuration file for EXP**

```
(* Create an instance of the *)
(* expression environment. *)
environment EnvExp

(* Register the following expression
(* instructions with the DVM. *)
Push  using EnvExp
Store using EnvExp
Load  using EnvExp
Sub   using EnvExp
Add   using EnvExp
Mul   using EnvExp
Br    using EnvExp
Brgz  using EnvExp
Nop   using EnvExp
Div   using EnvExp

(* Some generic instructions *)
Call  using EnvExp
Ret   using EnvExp
Print using EnvExp
```

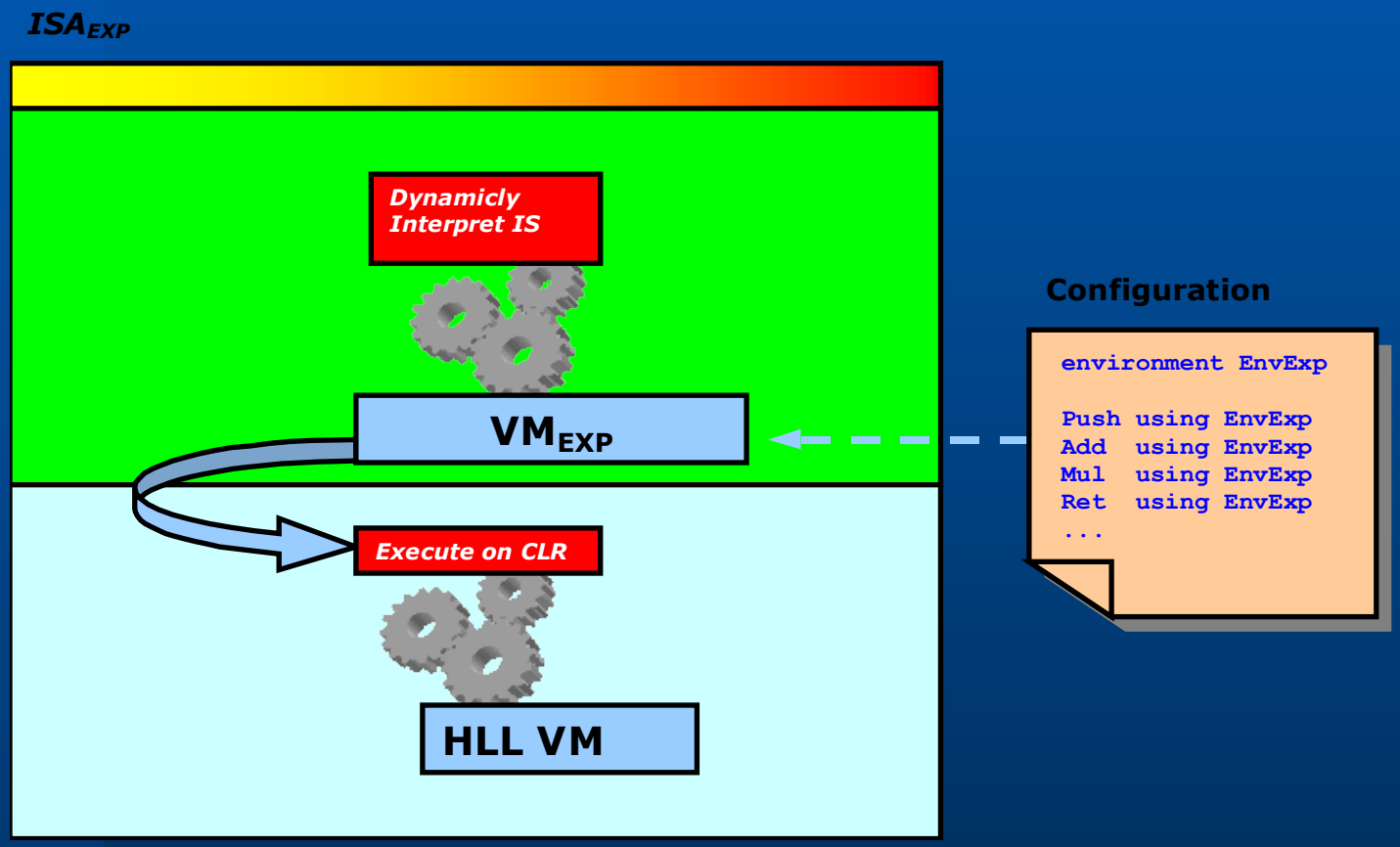
Extending the Environments

- **Environment EnvRT as a subtype of EnvExp**

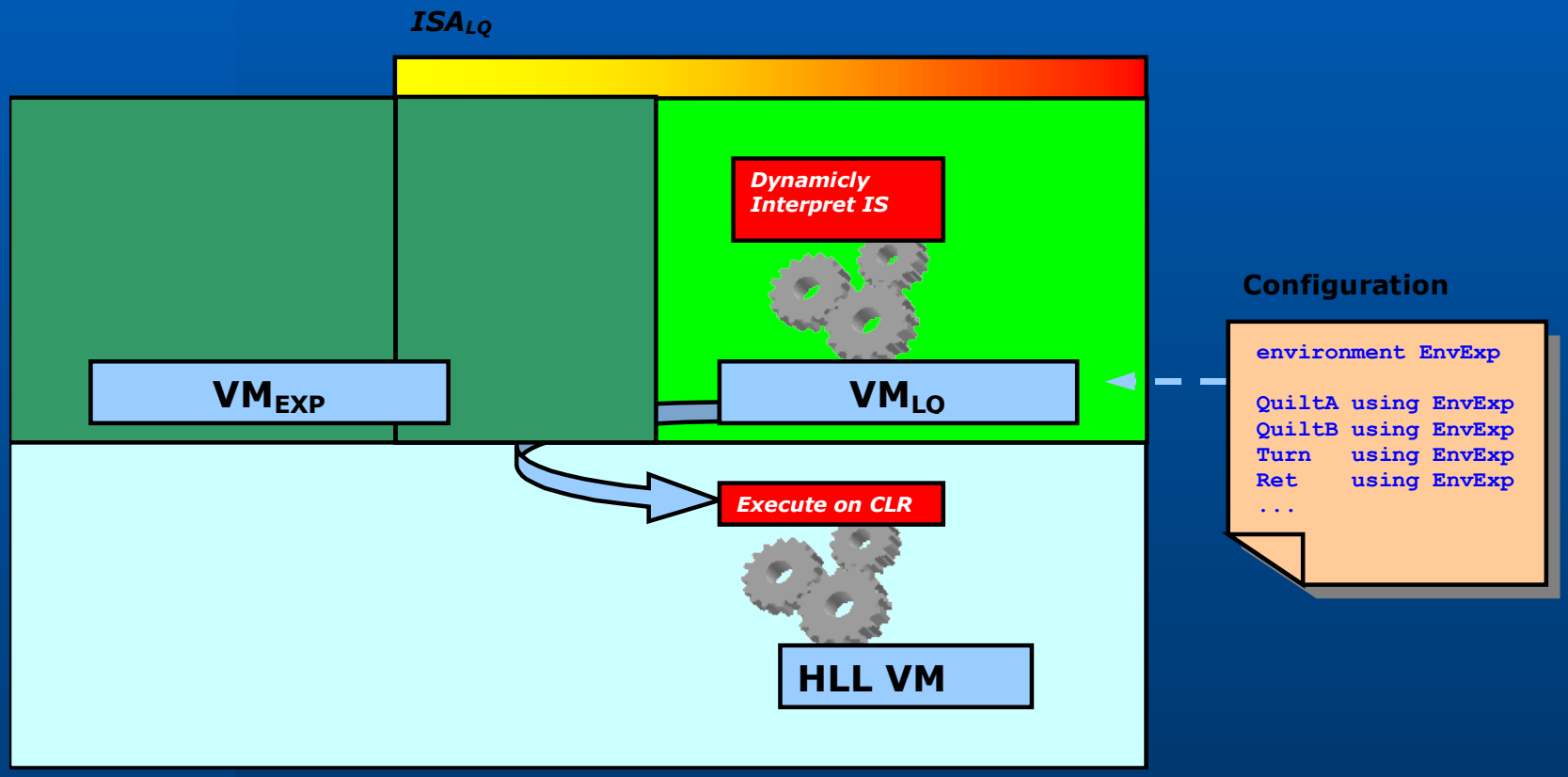
```
class EnvRT : EnvExp
{
   public EnvRT () : base ()
   {
      scene = new Scene ();
   }

   public override object GetResult ()
   {
      scene.render (4);
      return scene.GetBitmap ();
   }

   protected Scene scene;
   protected Bitmap bitmap;
}
```

Extending the Instruction Sets

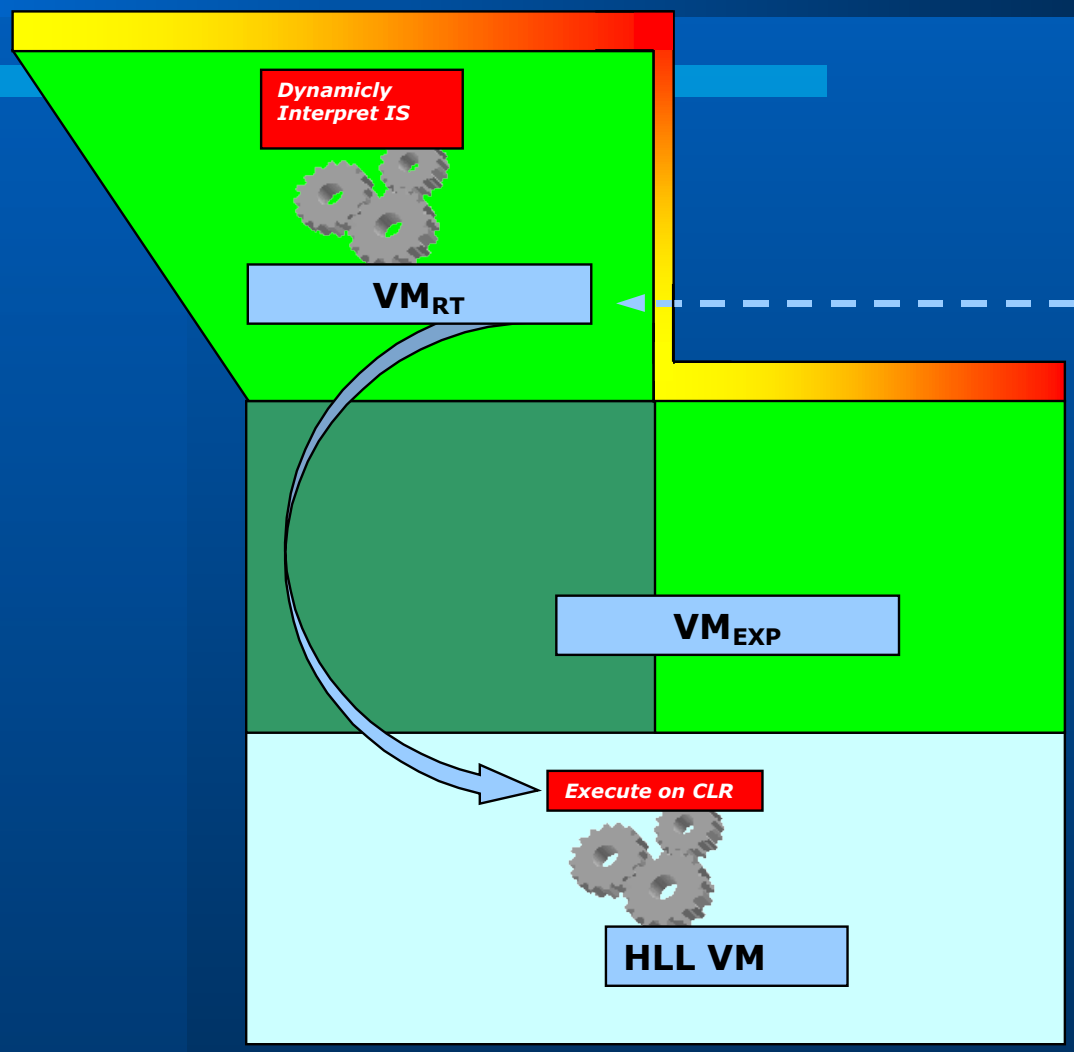
- Defining an environment and instruction set for a DSL VM



- ISA_{LQ} is a subset of ISA_{EXP}, with new instructions



- ISA_{RT} is a subset of ISA_{EXP}, with new instructions as before



Configuration

```
environment EnvRT

COP      using EnvRT
LookAt   using EnvRT
VUP      using EnvRT
Specular using EnvRT
...
Add      using EnvRT
Mul      using EnvRT
...
```

Adding the `ambient` Construct using the VM Framework

- **Altering the grammar for the new `ambient` construct on the RT compiler**

```
...
%node scene_prop6 : scene_prop
{
        public double d1, d2, d3;
        public scene_prop6 (double d1,double d2,double d3)
               {this.d1=d1;this.d2=d2;this.d3=d3;}
        public override void translate (StreamWriter asmFile)
        {
                asmFile.WriteLine ("Push\t{0}", d1);
                asmFile.WriteLine ("Push\t{0}", d2);
                asmFile.WriteLine ("Push\t{0}", d3);
                asmFile.WriteLine ("Ambient");
                asmFile.WriteLine ("Axes\n");
        }
}

...

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
             %scene_prop6 (d1.val,d2.val,d3.val);

prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp
...
```

Adding the `ambient` Construct using the VM Framework

- The `Ambient` instruction extends `Inst_OpCode` (no operands)

```
public class Ambient : Inst_OpCode
{
        public Ambient (EnvRT env)
        {
                this.env = env;
        }

        public override void Execute ()
        {
                double amz = (double) ((EnvRT) env).stack.pop ();
                double amy = (double) ((EnvRT) env).stack.pop ();
                double amx = (double) ((EnvRT) env).stack.pop ();
                ((EnvRT) env).scene.alterAmbient (amx, amy, amz);
        }
}
```

Adding the `ambient` Construct using the VM Framework

- **Adding the `Ambient` instruction to the instruction set**

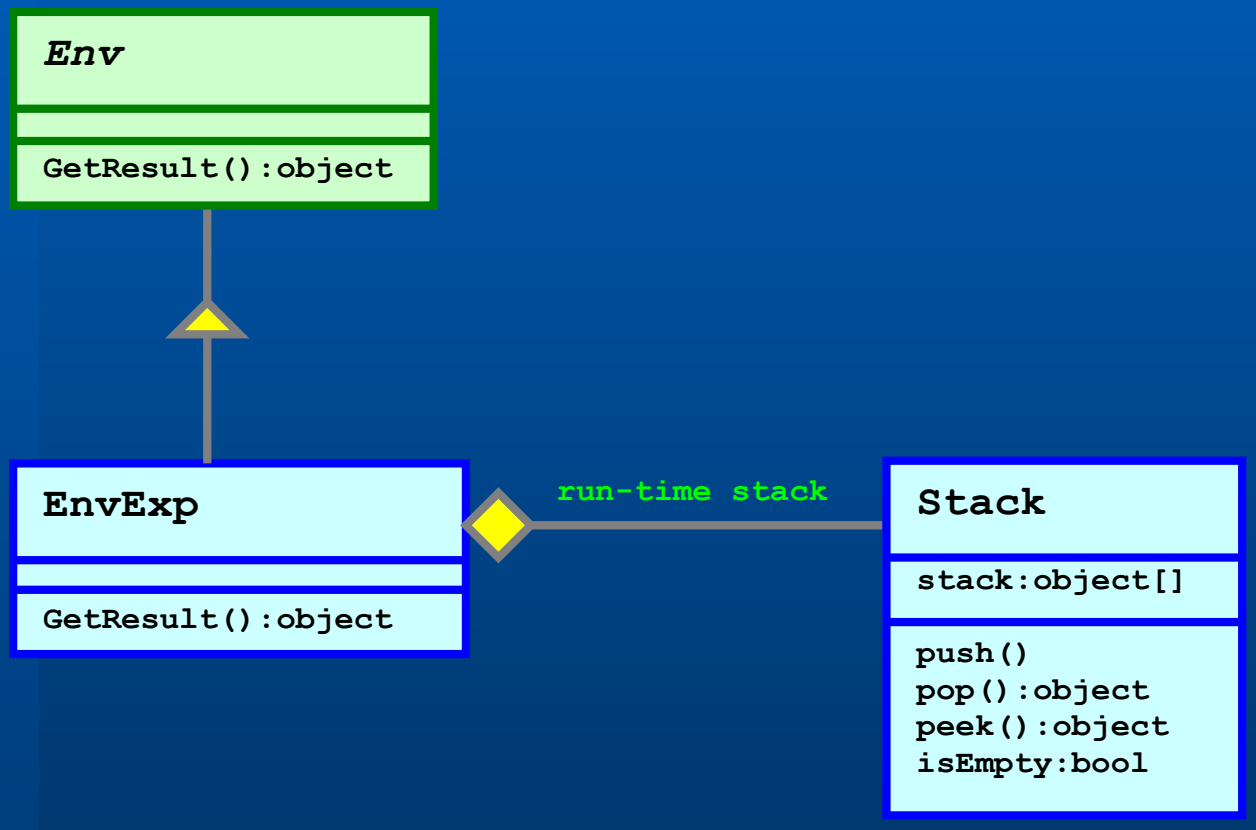
```
(* Create an instance of the ray-tracer environment. *)
environment EnvRT

...
COP        using EnvRT
LookAt     using EnvRT
VUP        using EnvRT
Ambient    using EnvRT
Geometry   using EnvRT
Specular   using EnvRT
Diffuse    using EnvRT
...
```

DSL Examples

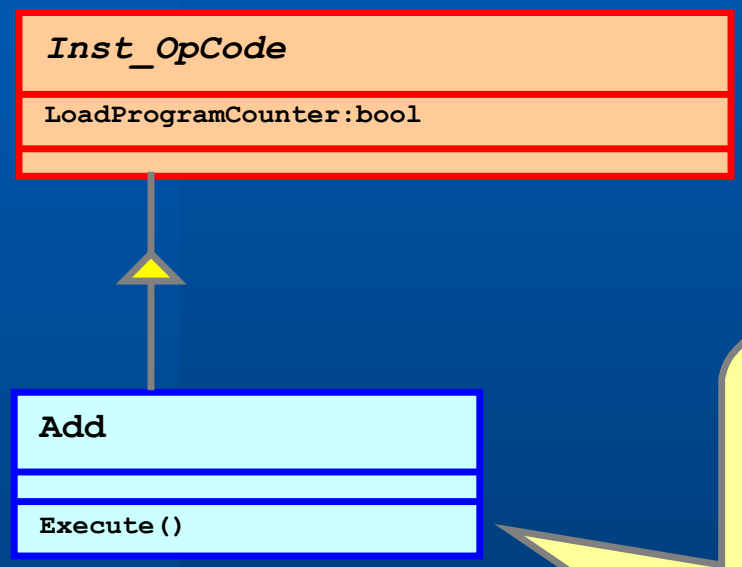
The Domain of EXP

$+ \quad - \quad * \quad \div$

The Environment of EXP



The ‘Add’ Instruction




```
public override void Execute ()
{
    double d1;
    double d2;

    d2 = (double)
        ((EnvExp) env).stack.pop ();
    d1 = (double)
        ((EnvExp) env).stack.pop ();

    ((EnvExp) env).stack.push
        (d1 + d2);
}
```

EXP Example

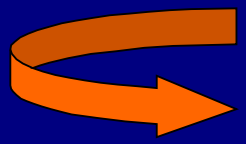
Configuration file:

```
environment EnvExp

Push  using EnvExp
Store using EnvExp
Load  using EnvExp
Sub   using EnvExp
Add   using EnvExp
Mul   using EnvExp
Br    using EnvExp
Brgz  using EnvExp
Nop   using EnvExp
Div   using EnvExp

Call  using EnvExp
Ret   using EnvExp
Print using EnvExp
```

EXP Example

```
let
    n = 5
in
    9 + sum (i) 1..n (2 * i)
end
```

Add

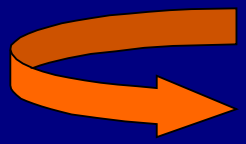


DSL Compiler

EXP Example

```
let
    n = 5
in
    9 + sum (i) 1..n (2 * i)
end
```

```
     Push 5
     Store $1
     Push 9
     Push 1
     Store $2
     Push 0
@100 Load $2
     Load $1
     Sub
     Brgz label @200
     Push 2
     Load $2
     Mul
     Add
     Load $2
     Push 1
     Add
     Store $2
     Br label @100
@200 Nop
     Add
```



DSL Compiler

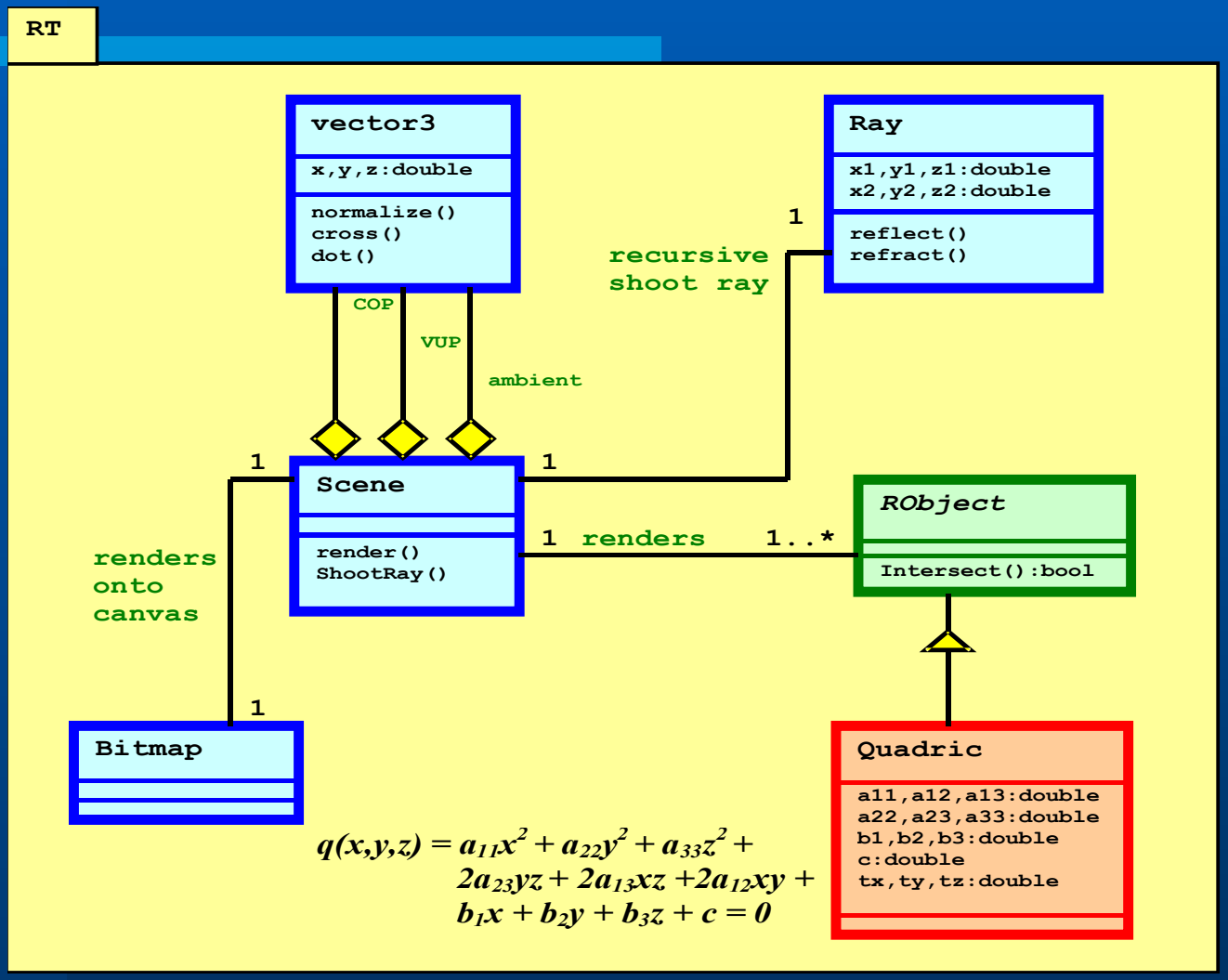
EXP Example

Output result : `System.Double`

Visual Studio .NET Command Prompt

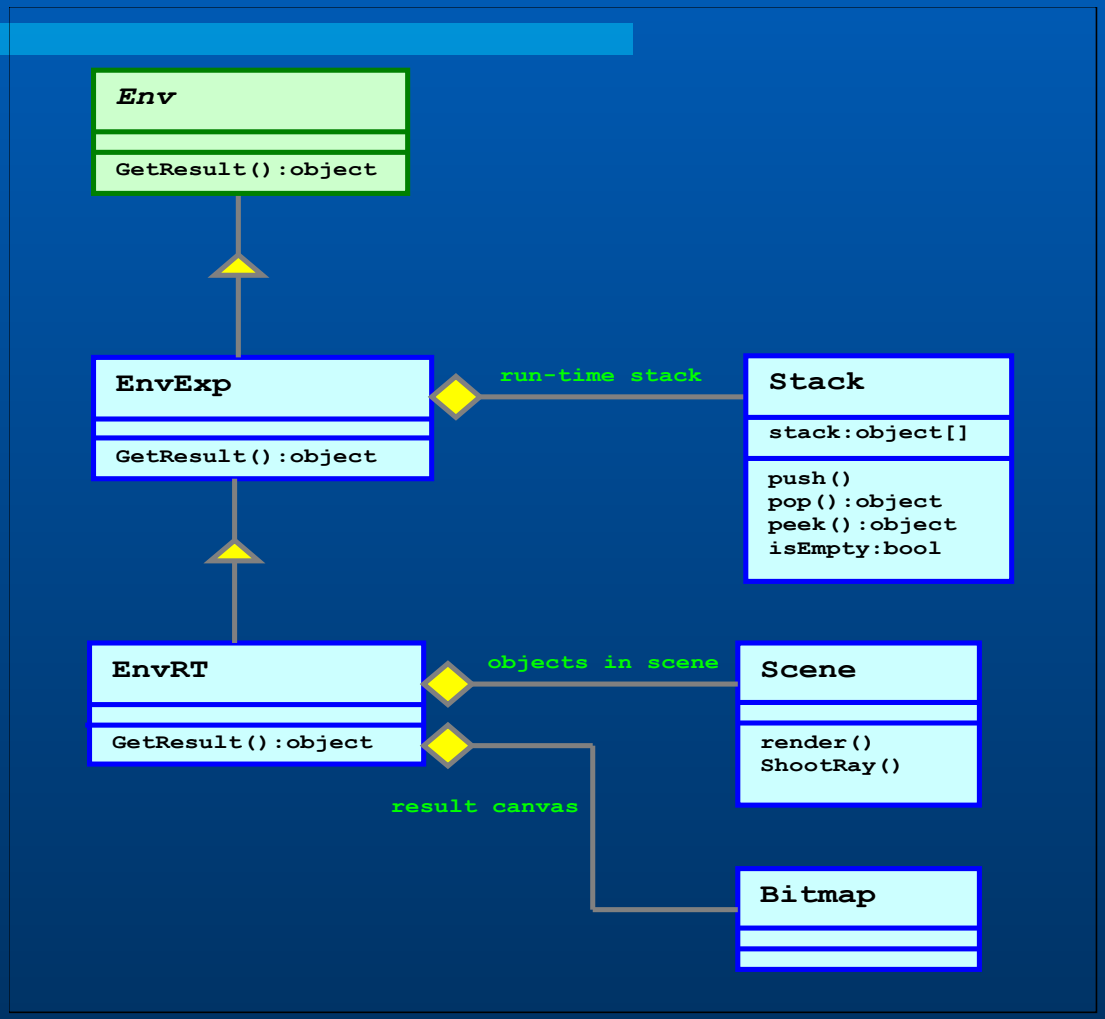
```
C:\DSL Compilers\EXP>expc test5.exp

C:\DSL Compilers\EXP>exp exp.vm test5.exp.ir
Loader: 21 instructions.
RESULT = 39

C:\DSL Compilers\EXP>
```

The Domain of RT



$$q(x,y,z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{23}yz + 2a_{13}xz + 2a_{12}xy + b_1x + b_2y + b_3z + c = 0$$

The Environment of RT



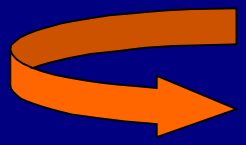
RT Example

Configuration file:

```
environment EnvRT

Push       using EnvRT
Add        using EnvRT
Sub        using EnvRT
Mul        using EnvRT
Div        using EnvRT

SetWindow using EnvRT
Light      using EnvRT
COP        using EnvRT
LookAt     using EnvRT
VUP        using EnvRT
Ambient    using EnvRT
Geometry   using EnvRT
Specular   using EnvRT
Diffuse    using EnvRT
Reflect    using EnvRT
Refract    using EnvRT
Translate using EnvRT
Axes       using EnvRT
Coeff      using EnvRT
Quad       using EnvRT
```

RT Example

```
scene {
     window 300.0 200.0  light -3.5 0.5 2.0
     cop 0.0 4.0 1.0     lookat 0.0 0.0 0.0
     viewup 0.0 0.0 1.0  ambient 0.3 0.3 0.3

     sphere 0.5 {
          translate 1.0 0.0 0.0   reflective 0.1 0.1 0.1
          specular 0.7 0.0 0.0    diffuse 0.7 0.0 0.0
     }

     sphere 0.5 {
          translate 0.0 -1.0 0.2  reflective 0.2 0.3 0.2
          specular 0.0 0.7 0.0    diffuse 0.0 0.7 0.0
     }

     sphere 0.5 {
          translate -1.0 -0.5 0.0 reflective 0.1 0.1 0.1
          specular 0.0 0.0 0.7    diffuse 0.0 0.0 0.7
     }

     plane -0.1 0.0 1.0 1.0 {
          specular 0.0 0.0 0.0    diffuse 1.0 0.5 0.0
          reflective 0.25 0.25 0.25
     }
}
```



DSL Compiler

```
Coeff "c"

Push 1
Push 0
Push 0
Translate
```

RT Example

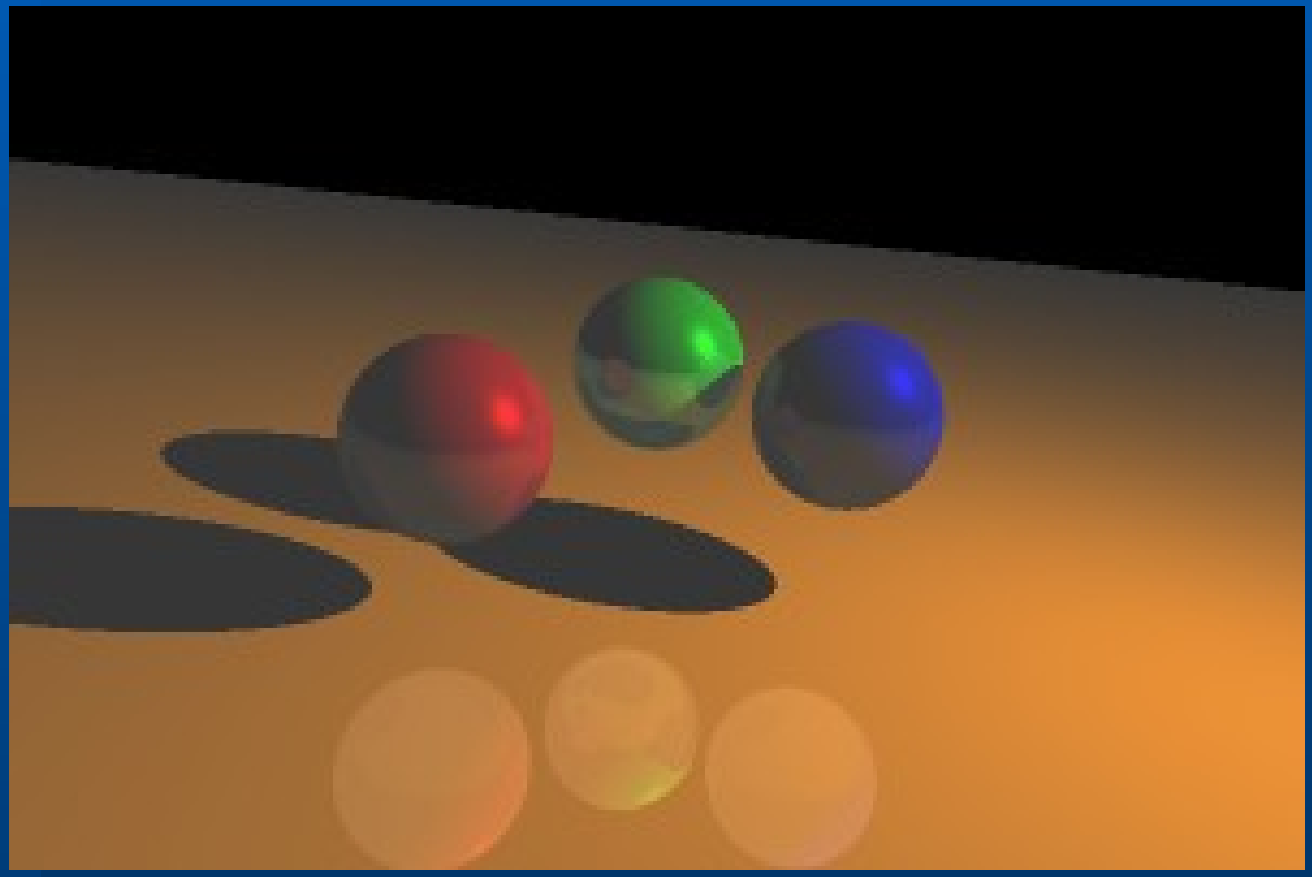
```
scene {
     window 300.0 200.0  light -3.5 0.5 2.0
     cop 0.0 4.0 1.0     lookat 0.0 0.0 0.0
     viewup 0.0 0.0 1.0  ambient 0.3 0.3 0.3

     sphere 0.5 {
          translate 1.0 0.0 0.0   reflective 0.1 0.1 0.1
          spec
     }

     sphere 0.
          tran
          spec
     }

     sphere 0.
          tran
          spec
     }

     plane -0.
          spec
          refl
     }
}
```

```
Quad
Push 1
Push 0.5
Push 0.5
Mul
Div
Coeff "a11"
Push 1
Push 0.5
Push 0.5
Mul
Div
Coeff "a22"
Push 1
Push 0.5
Push 0.5
Mul
Div
Coeff "a33"
Push -1
Coeff "c"

Push 1
Push 0
Push 0
Translate
```

DSL Compiler

RT Example

Output result : `System.Drawing.Bitmap`



The Domain of LQ

LittleQuilt

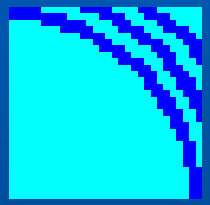
```
enum Orientation
{
    N,
    E,
    S,
    W
}
```

Quilt

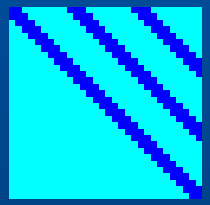
```
NumRows:uint
NumCols:uint
quilt:Orientation [,]
symbols:Symbol [,]

Right():Quilt
Sew():Quilt
```

```
enum Symbol
{
    A,
    B
}
```

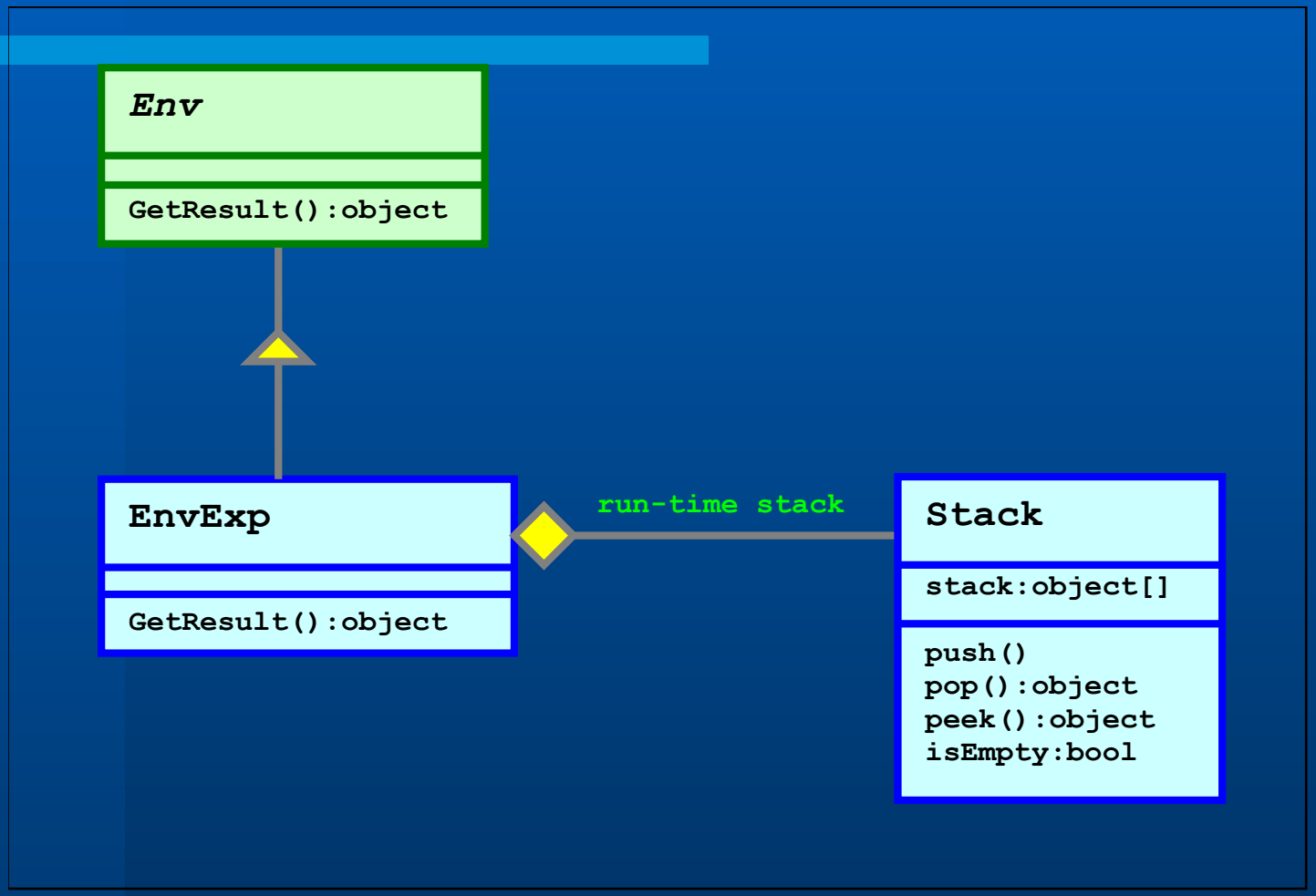
A =



B =



The Environment of LQ



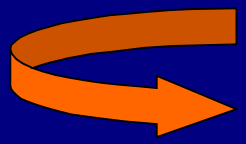
LQ Example

Configuration file:

```
environment EnvExp

QuiltA using EnvExp
QuiltB using EnvExp
Load   using EnvExp
Store  using EnvExp
Turn   using EnvExp
Sew    using EnvExp
Call   using EnvExp
Ret    using EnvExp
Br     using EnvExp
```

LQ Example

```
let
    fun unturn (x)  = turn (turn (turn (x)))
    fun pile (x, y) = unturn (sew (turn (y), turn (x)))
    val s = turn (sew (unturn (unturn (a)), b))
    val t = pile (turn (b), unturn (b))
    val u =  turn (turn (sew (s, t)))
    val m = sew (turn (a), turn (turn (a)))
    val n = sew (unturn (a), a)
    val p = pile (m, n)
    val q =  sew (sew (u, p),  turn (u))
in
    pile (q, turn (turn (q)))
end
```



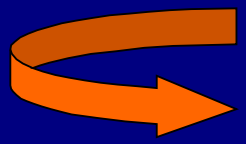
DSL Compiler

```
@300
     QuiltA
     Store $1
     Call label @100
     Store $1
     Call label @100
     QuiltB
```

LQ Example

```
let
    fun unturn (x)  = turn (turn (turn (x)))
    fun pile (x, y) = unturn (sew (turn (y), turn (x)))
    val s = turn (sew (unturn (unturn (a)), b))
    val t = pile (turn (b), unturn (b))
    val u =  turn (turn (sew (s, t)))
    val m = sew (turn (a), turn (turn (a)))
    val n = se
    val p = pi
    val q =  s
in
    pile (q, t
end
```

```
     Br label @199
@100
     Load $1
     Turn
     Turn
     Turn
     Ret
@199
     Br label @299
@200
     Load $2
     Turn
     Load $1
     Turn
     Sew
     Store $1
     Call label @100
     Ret
@299
     Br label @399
@300
     QuiltA
     Store $1
     Call label @100
     Store $1
     Call label @100
     QuiltB
```



DSL Compiler

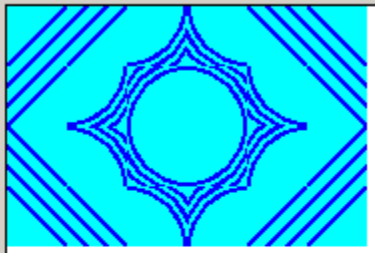
LQ Example

Output result : `LittleQuilt.Quilt`

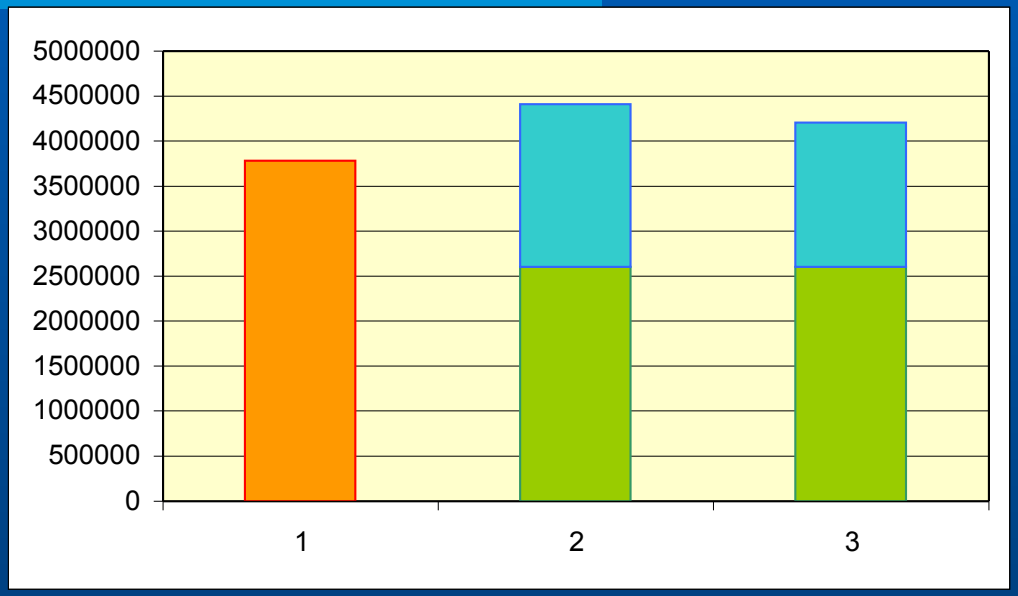
Little Quilt

File Compile

```
let
    fun unturn (x)  = turn (turn (turn (x)))
    fun pile (x, y) = unturn (sew (turn (y), turn (x)))
    val s = turn (sew (unturn (unturn (a)), b))
    val t = pile (turn (b), unturn (b))
    val u =  turn (turn (sew (s, t)))
    val m = sew (turn (a), turn (turn (a)))
    val n = sew (unturn (a), a)
    val p = pile (m, n)
    val q =  sew (sew (u, p),  turn (u))
in
    pile (q, turn (turn (q)))
end
```



Comparative Results



Legend

1 – Interpreter
2 – Hard-coded VM
3 – VM Framework

– SRC -> SEM (Interpreter)
– SRC -> IR
– IR -> SEM

Summary of Criteria

Legend:

- ☼ : Good
- ☁ : Average
- ☂ : Poor

	Scalability	Performance	Modularity
Interpreter	☂ Difficult and cumbersome to modify a language and alter semantics.	☼ Interpretation of AST done in same program, and there's no IS.	☂ Syntax and semantics issues are fairly dependant on one another.
Hard-coded VM	☁ Instructions are still required to be hard-coded as part of the core.	☁ Compiled program needs to be parsed before it can be executed.	☁ Minimal effort required adding new instructions to the VM core.
VM Framework	☼ Easily add new instructions and environments as the language grows.	☁ Comparable performance to the hard-coded VM, but slightly faster for compiled program execution.	☼ Instructions and environments form the link between the VM and the underlying namespace, and are easily modified as the namespace expands.

Thank You