Interpretação e Compilação de Linguagens (de Programação)

22/23

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Programming Languages (PL)

- The purpose of a PL is to allow computational processes to be described specified by linguistic means.
- In early times, programs were mostly written in (physical) machine code.
- The definition of a PL involves to aspects, that must be characterised by a precise, **non-ambiguous way**:
  - Syntax

> syntax |ˈsinˌtaks|
>
> the arrangement of words and phrases to create well-formed sentences in a language *: the syntax of English.*
> - a set of rules for or an analysis of this *: generative syntax.*
> - the branch of linguistics that deals with this.

  - Semantics

> semantics |səˈmantiks|
>
> the branch of linguistics and logic concerned with meaning. There are a number of branches and subbranches of semantics, including **formal semantics**, which studies the logical aspects of meaning, such as sense, reference, implication, and logical form, **lexical semantics**, which studies word meanings and word relations, and **conceptual semantics**, which studies the cognitive structure of meaning.

Programming Languages

Example of syntactic ambiguity:

What is the value of `f(10)` ?

```
int f(int x) {
  if ( x > 0 )
    if ( x < 10 ) return x;
  else return 10;
  return 0;
}
```

```
f(10) = ??
```

Programming Languages

Example of semantic ambiguity:
What is the value of `f(2)+g(3)` ?

```
f(2) + g(3) = ??
```

```
public class A {
   static int a = 0;

   static int f(int x) {
      a = a + 1;
      return x;
   }
   static int g(int y) { return y + a; }

   static int sum(int x, int y) { return x + y; }

   public static void main(String[] args) {
      System.out.println(sum(f(2), g(3)));

   }
}
```

Programming Languages (Syntax)

Syntax specifies the **form** of programs in the language, how they should be written, without attending to their meaning. Syntax is just about the structure of program phrases.

A PL syntax is conveniently defined by a **lexicon** (set of words or tokens) and a **grammar** (set of formation rules).

```
integer literal:  ("0" | ["1"-"9"]["0"-"9"]*)
real literal : (["0"-"9"]) "." (["0"-"9"])* ("E" ...)?
identifier: [a-z,A-Z,_][a-z,A-Z,_,0-9]*
reserved symbol: int, float, void, while, class, ...
```

The `if` statement is in the form:

```
if (<expression>)
    <statement1>
else
    <statement2>
```

Programming Languages (Syntax)

The concrete syntax of a PL may be formally specified using regular languages for tokens and context free grammars for program phrases (see course "Theory of Computation").

Given a description of tokens (eg. using regular expressions) and given a description of a grammar (eg., using grammar rules) one may construct lexical analysers and parsers, which are programs that check the syntax programs for syntactical correctness and construct **abstract syntax trees (AST).**

ASTs are data structures that represent syntactically correct programs in a structured form (not just as text - sequence of characters, in the source files).

Good news: there are **tools** that will do that for you automatically.

You will still need to know how to specify regular languages and (non-ambiguous) context free grammars.

Grammar for arithmetic expressions (yacc)

```
%token  NAME
%token  NUMBER
%token  EQ
%token PLUS MINUS TIMES DIV
%left MINUS PLUS
%left TIMES DIV
%nonassoc UMINUS

%%

statement_list
        : statement
        | statement statement_list

statement
        : NAME EQ expression ';' {vbltable[$1] = $3; }

expression
        : expression PLUS expression {$$ = $1 + $3;}
        | expression MINUS expression {$$ = $1 - $3;}
        | expression TIMES expression {$$ = $1 * $3;}
        | expression DIV   expression {$$ = $1 / $3;}
        | MINUS expression %prec UMINUS {$$ = - $2;}
        | '(' expression ')' { $$ = $2; }
        | NUMBER
        | NAME   { $$ = vbltable[$1]; }
```

Grammar for arithmetic expressions (javacc)

```
void Start() :
{ }
{
    exp() <EOL>
}

void exp() :
{ }
{
    term() [ <PLUS> exp() ]
}

void term() :
{ }
{
    factor() [ <MULTIPLY> term() ]
}

void factor() :
{  }
{
    <CONSTANT>
|   <LPAR> exp() <RPAR>
}
```

Grammar for arithmetic expressions (javacc)

JavaCC

Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications.

A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar.

In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions and debugging.

All you need to run a JavaCC parser, once generated, is a Java Runtime Environment (JRE).

Programming Languages (Semantics)

A PL semantics describes in a precise way the meaning of program elements and phrases. Typically you find verbose descriptions in reference manuals. There is no place for ambiguity or arbitrariness in meaning!

10.4 Array Access

A component of an array is accessed by an array access expression (§15.13) that consists of an expression whose value is an array reference followed by an indexing expression enclosed by `[` and `]`, as in `A[i]`. All arrays are `0`-origin. An array with length *n* can be indexed by the integers `0` to *n*-1.

Arrays must be indexed by `int` values; `short`, `byte`, or `char` values may also be used as index values because they are subjected to unary numeric promotion (§) and become `int` values. An attempt to access an array component with a `long` index value results in a compile-time error.

All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown.

in *The Java Language Specification*

Programming Languages (Semantics)

The semantics of a PL may be defined by giving a computable function I which assigns a definite meaning to each program (fragment)

$$I : PROG \rightarrow DENOT$$

PROG = set of all programs (as syntactical structures)

DENOT = set of all meanings (denotations)

The semantic mapping I may be mathematically defined, typically using set theoretic constructions.

This is referred to as an **denotational semantics**.

More pragmatically the semantic mapping may also be given by an interpreter algorithm.

Such **interpreter algorithm** takes as input any program (as a syntactical data structure / AST) and yields its **value** and/or **effect**.

This is referred to as an **operational semantics** (approach in this course).

Code as Data

An interpreter for a language L is a program I that accepts as input the representation of a program in the source language (as an data structure) and realizes its execution according to the semantics of L.

The interpreter may be written in some language X, in general the language that is accepts is a different one. Eg., we may write a Python interpreter in C.

This process is called "bootstrapping" and is used in general to define interpreters for arbitrary languages. Languages that allows one to write interpreter for any language are called Turing complete, and this is the main content of the so-called Turing universality.

An expressive programming language should allow one to write an interpreter / compiler for itself (f"ull bootstrapping").

Code as Data

A compiler for a language L is a program I that accepts as input the representation of a program in the source language (as an data structure) and outputs an equivalent program (with the same meaning) in another language.

A compiler is a **translator**, unlike an interpreter it does not execute the source program but translates the source program in a program in a simpler language, or in a language for which an interpreter or compiler already exists, or even a physical (an hardware processor) or an abstract machine (e.g., JVM) able to execute the target code directly.

The translation process must preserve the meaning of programs, that is, the code generated by the compiler (target code) must have the same meaning as the source code.

Goals of the Course: Knowledge

- What techniques are used in the design and implementation of programming languages ?
- What are the building blocks of a programming language ?
- How to describe, analize and justify the features and characteristics of a programming language using the concepts studied?
- How to design interpreters and compilers ?
- How can we define and predict the effect of programming constructs in a precise manner ?
- How can we express and ensure properties of programming languages such as error absence and type safety ?
- How can we define and express validation algorithms for programming languages, that will work for any program written on it ?
- How do modern runtime support environments for programming languages (JVM, .NET, LLVM) work ?

Goals of the Course: Hands-on

- How do we construct a syntactic analyser (parser) ?
- What tools are there to assist on that ?
- How do we represent programs as data ?
- How do we specify the semantics of a programming language ?
- How to do it for abstract high level concepts such as higher order functions, objects, classes, etc ?
- How do we use it to implement an interpreter or compiler ?
- How does a compiler generate code for a real or abstract machine?
- How do we define and implement basic static analysis algorithms (e.g., type-checking) ?

"Road Map"

We will progress in a "onion" layer incremental way, covering key construct present in all programming languages.

The course tightly couples principles design and implementation, in the hands-on part you will develop an interpreter and compiler for a realistic programming language.

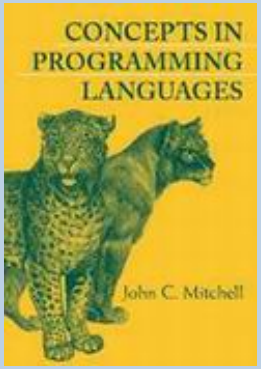
This is an "integrative" course, you combine lots of stuff: AED, TC, AC, SE,

- Values, Basic Operations and Expressions
- Naming and Binding
- State (mutable memory)
- Functional abstraction
- Types and Type Systems
- Data Abstraction
- Objects, Classes and Modules
- We will see how the various primitives may be interpreted and compiled to a target machine (JVM / LLVM)

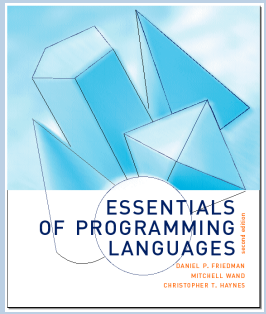
"If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master."

(Hal Abelson, Essentials of Programming Languages de Friedman et al.)

Bibliography

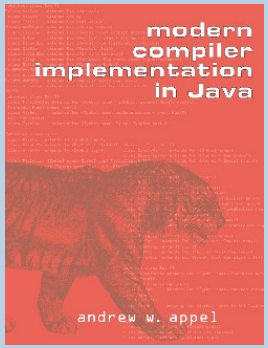


"*Concepts in Programming Languages*",
John C. Mitchell,
Cambridge University Press.
ISBN 0 521 78098 5



"Essentials of Programming Languages",
Daniel Friedman, Mitchell Wand, Christopher Haynes,
MIT Press.

"The Study of Programming Languages",
Ryan Stansifer,
Prentice Hall International Edition.



"Modern Compiler Implementation in Java"
Andrew W. Appel
Cambridge University Press

Course Grading

Continuous Evaluation:

Midterm test (8)
Final test (8)

Handout (2 phases) 4

Must be realised in groups of 2

Exam (for those failing in the CE)

Admittance to exam requires submission of the handout

Interpretação e Compilação de Linguagens (de Programação)

Part 2A
Interpreters and Compiler Basics: Simple Expressions

Mestrado em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Operational Semantics

Interpreters and compilers are language processors that accept syntactically correct programs in a source language and produce denotations (values, effects, or other larger programs).

An **interpreter** produces a value or effect (executes source program directly)

A **compiler** produces a program in a (lower level) **target** language (typically, a machine language). The target program then implements the source program.

How do we define the semantics of a language?

Let's look at a simple example (the CALC language)

- We construct a **compositional** definition
  - Interpreter for CALC: implementation using an OO language (Java) the evaluation function is defined "in pieces", one case for each constructor of the AST
  - Compiler for CALC: implementation using an OO language (Java) the compilation function is defined "in pieces", one case for each constructor of the AST, we target the JVM (Java virtual machine)

The CALC Language (arithmetic expressions)

- The abstract syntax of CALC is given by the following constructors

$$\begin{aligned}
&\textbf{num}: \text{Integer} \rightarrow \text{CALC}\\
&\textbf{add}: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}\\
&\textbf{mul}: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}\\
&\textbf{div}: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}\\
&\textbf{sub}: \text{CALC} \times \text{CALC} \rightarrow \text{CALC}
\end{aligned}$$

- Each constructor represents an operator for building a new expression of CALC given already defined expressions.
- The concrete syntax of a language:
  - defines the form how expressions and programs are effectively written in terms off formatting, sequences of (ascii / unicode) characters, etc…
- The abstract syntax of a language:
  - defines the deep structure of expressions and programs in terms of a composition of abstract constructors (like the functions above).

Abstract Syntax vs. Concrete Syntax (examples)

- Numeral literals
  - decimal notation : 12
  - hexadecimal notation : 0x0C
  - abstract syntax: num(12)
- Identifiers
  - C: xpto
  - bash: %A
  - abstract syntax: id("A")
- Assignment
  - C: x = 2
  - OCAML: x := 2
  - abstract syntax: assign(id("x"),num(2))

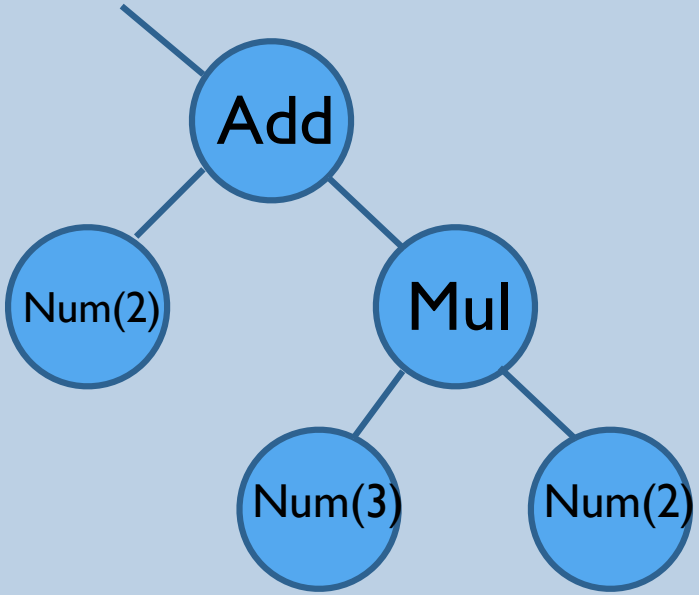
Abstract Syntax vs. Concrete Syntax (examples)

- Algebraic Expressions
  - C: 2*3+2
  - RPN: 2 3 * 2 +
  - Lisp: (+ (* 2 3) 2)
  - abstract syntax: add(mul(num(2),num(3)),num(2))
- Block
  - C: {S1 S2 … Sn }
  - Python: \tab S1 \tab S2 … \tab Sn
  - abstract syntax: block(S1,S2,…,Sn) or seq(S1, seq(S2, seq (…))
- while loop:
  - C: while (C) S
  - Pascal: while C do S
  - abstract syntax: while(C,S)

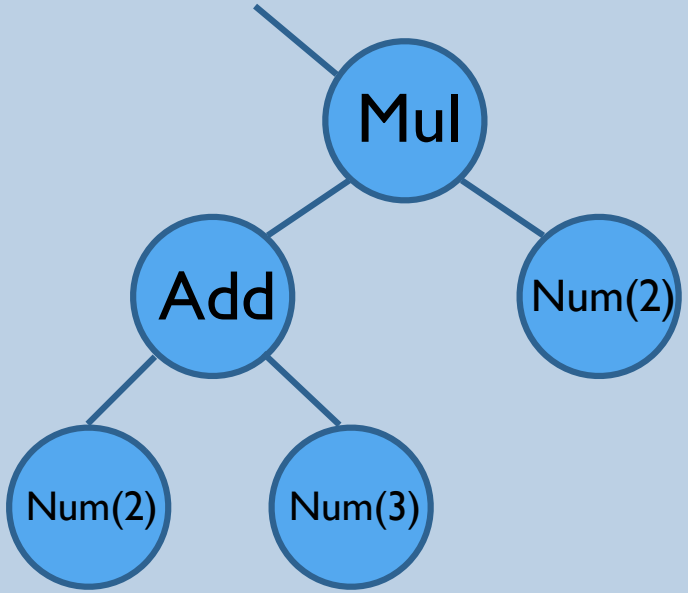
Abstract Syntax Tree

- Represents the structure of expressions and programs in terms of a composition of abstract constructors, depicted as a tree-like structure.
- Abstract Syntax Tree (AST)

`2 + 3 * 2`



`(2 + 3) * 2`



Semantics of CALC

The semantics of a PL may be defined by giving a computable function I which assigns a definite meaning to each program (fragment)

$$I : CALC \rightarrow DENOT$$

CALC = set of all programs (as syntactical structures)

DENOT = set of all meanings (denotations)

The denotation (value) of a CALC program is an integer value, so we may set

$$I : CALC \rightarrow Integer$$

CALC = set of all programs (as syntactical structures)

DENOT = Integer = set of all meanings (denotations)

CALC Interpreter (evaluation map)

- Algorithm eval(E) that computes the denotation (integer value) of any CALC expression:

$$eval : CALC \rightarrow Integer$$

if E has the form **num**(n):	eval(E) = n
if E has the form **add**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1+v2
if E has the form **mul**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1*v2
if E has the form **sub**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1-v2
if E has the form **div**(E',E''):	v1 = eval(E'); v2 = eval(E'');
	eval(E) ≜ v1/v2

CALC Interpreter (evaluation map)

- Algorithm eval(E) that computes the denotation (integer value) of any CALC expression:

$$eval : CALC \rightarrow Integer$$

eval(num(n))	≜ n
eval(add(E',E''))	≜ eval(E') + eval(E'')
eval(mul(E',E''))	≜ eval(E') * eval(E'')
eval(sub(E',E''))	≜ eval(E') - eval(E'')
eval(div(E',E''))	≜ eval(E') / eval(E'')

CALC AST as an (OCAML) inductive data type

```
type calc =  Num of int
           | Add of calc * calc
           | Mul of calc * calc
           | Div of calc * calc
           | Sub of calc * calc
```

CALC eval map as an (OCAML) recursive function

```
let rec eval e =
    match e with
       Num(n) -> n
     | Add(e1,e2) -> (eval e1)+(eval e2)
     | Mul(e1,e2) -> (eval e1)*(eval e2)
     | Sub(e1,e2) -> (eval e1)-(eval e2)
     | Div(e1,e2) -> (eval e1)/(eval e2)
```

Structural Operational Semantics (Plotkin81)

- Algorithm eval(E) that computes the denotation (integer value) of any CALC expression:

$$eval : CALC \rightarrow Integer$$

- The semantic map eval(-) is recursively defined in the structure of the source abstract syntax.
- For each constructor, the semantics of the compound expression only depends on the meaning of each component.

$$eval(\textbf{mul(E1,E2)}) \triangleq eval(\textbf{E1}) * eval(\textbf{E2})$$

- **Compositional Semantics**: the meaning of the whole results from the meaning of parts (in particular, it does not depend on the context).
- The same does not depend on "natural" languages
  - **time flies like an arrow**
  - **fruit flies like a banana**
- **Compositionality** is a very important property of a principled formal semantics, it allows us to define the semantics modular and manageable!

Java Implementation

- In a OO language, an inductive data type may be represented by an interface type (opaque) and a collection of classes, where each class represents a particular constructor
- The interface declares operations over the inductive data type, for instance::

```
public interface CALC {

     int eval();

}
```

This represents the map eval: CALC → Integer

Java Implementation

- Each class represents a particular constructor
- For each operation / map $f: T \rightarrow V$ defined over the inductive data type T, each class provides the implementation for f on the respective constructor, e.g.;

eval(num(n)) ≜ n

```
public class ASTNum implements CALC {
     private int value ;
     ASTNum(int v) { value = v; }
     int eval() { return value; }
}
```

Java Implementation

- Each class represents a particular constructor
- For each operation / map $f: T \rightarrow V$ defined over the inductive data type T, each class provides the implementation for f on the respective constructor, e.g.;

eval(add(e1,e2)) ≜ eval(e1)+eval(e2)

```
public class ASTAdd implements CALC {
CALC lhs, rhs;
ASTAdd(CALC e1,e2) { lhs=e1, rhs=e2; }
int eval() {
return lhs.eval()+rhs.eval(); }
}
```

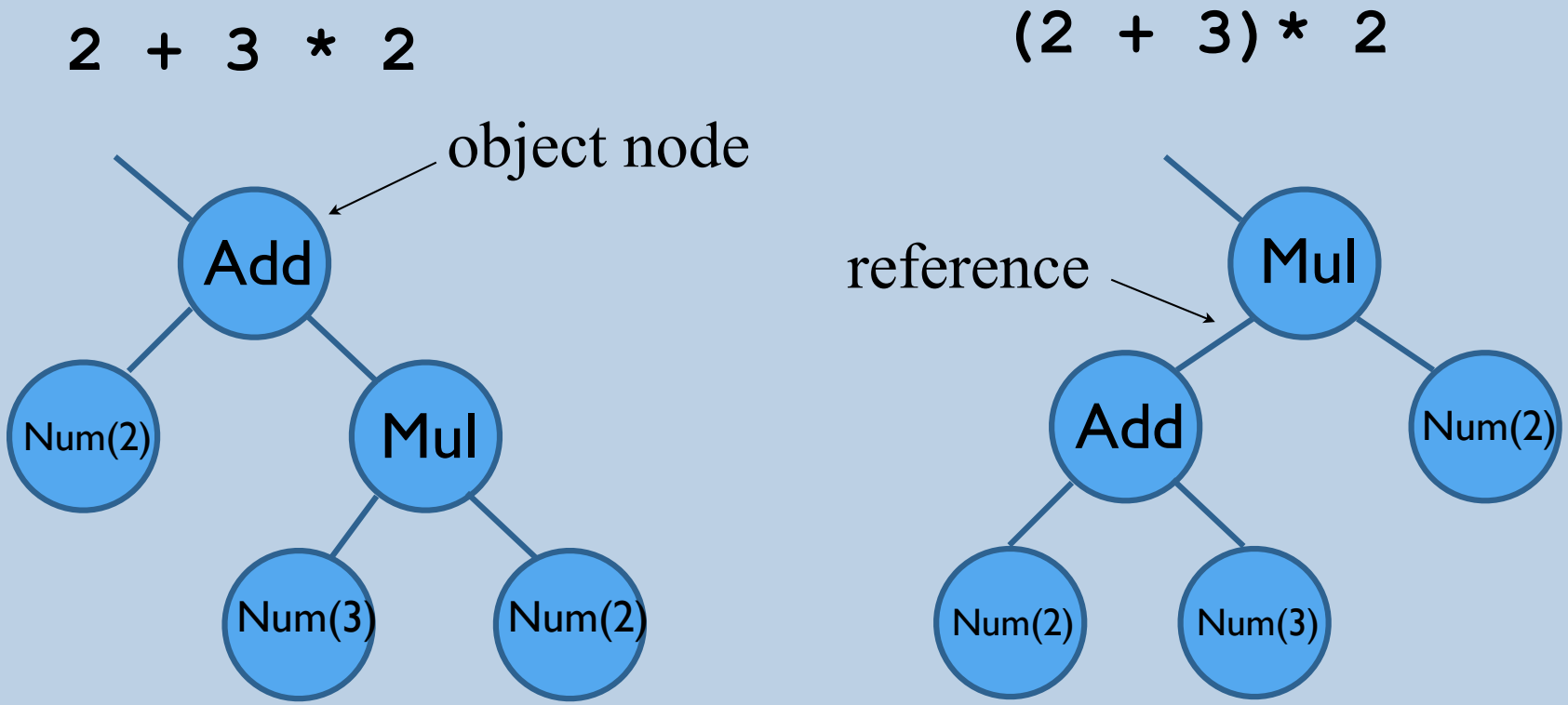
Java Implementation

- Each expression of the language is thus represented by an (n-ary) tree of objects (the AST)
- Constructors of the AST are the constructors of the inductive data type, and implemented by the AST classes's constructors

```
CALC expr1 = new Add(new Num(2),new Num(3));
int result1 = expr1.eval();
CALC expr2 =
     new Add(new Sub(new Num(2),new Num(3)),new Num(3));
int result2 = expr2.eval();
```

Abstract Syntax Tree

- Represents the structure of a program expression in terms of the application of the abstract constructors, implemented as a data structure
- Abstract Syntax Tree (AST)



Java Implementation

- In our example, classes `ASTNum`, `ASTAdd`, `ASTSub`, `ASTMul` e `ASTDiv`, implement the language construtors num, add, sub, mul and div.
- The definition of the evaluation map eval: CALC→ V becomes dispersed by the several classes of the abstract syntax.
- This makes it easier to add new constructs to the language, we just need to add a new AST class type and the associated implementation of the evaluation maps.
-

```
public class ASTNeg implements CALC {
private CALC exp;
ASTNeg(CALC e) { exp = e;}
int eval() { return -exp.eval(); }
}
```

Java Implementation

- In our example, classes `ASTNum`, `ASTAdd`, `ASTSub`, `ASTMul` e `ASTDiv`, implement the language construtors num, add, sub, mul and div.
- The definition of the evaluation map eval: CALC→ V becomes dispersed by the several classes of the abstract syntax.
- On the other hand, to add new functions f: CALC→ V to a language, one needs to modify all classes for AST nodes of CALC. Later we'll discuss ways of factoring the code to mitigate this "problem" (visitor pattern).

```
public class ASTNeg implements CALC {

private CALC exp;

ASTNeg(CALC e) { exp = e;}

int eval() { return -exp.eval(); }

}
```

Compilers

- A compiler produces a program in a (lower level) target language (typically, a machine language).
- The target program then implements the source program.
- The target of the compiler may be code in another language (example javacc compiles grammars into Java, gcc compiles C into LLVM).
- The compiler of a source language S into a target language T preserves the meaning, that is the denotation of the source and target program should be the same. that is,

$$evalt(comp(P)) = evals(P)$$

$$evalt: T \rightarrow V$$

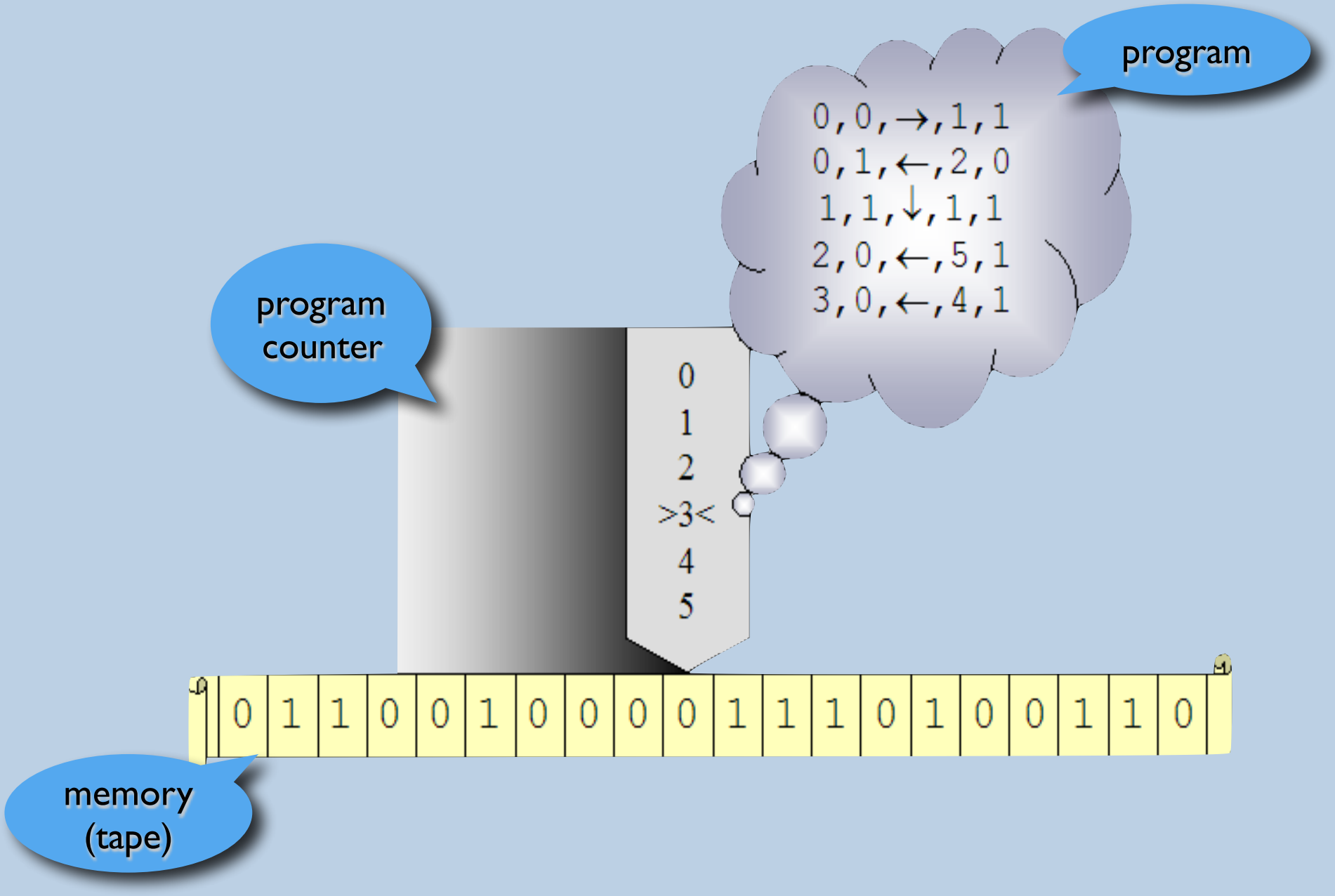
$$evals: S \rightarrow V$$

- Like an interpreter, a compiler may be conveniently defined by recursion on the abstract syntax (AST) of the language
- The target machine of a compiler may be physical (eg., Intel A7) or virtual (eg., JVM, CLR). We will use virtual machines as targets in our course.

Virtual Machines (software processors)

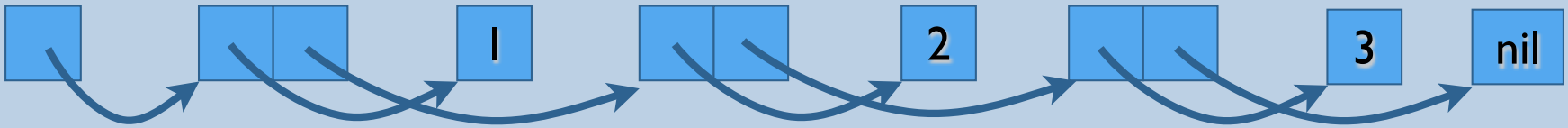
- Turing Machine (Turing, 1931)
  - The first one …
- SECD (Landin, 1962)
  - Stack, Environment, Code, Dump
- P-code machine (Wirth 1972)
  - Stack machine, Pascal p-code compiler
- JVM (Sun, 1995)
  - Type safe, Multi-threaded, focusing on the Java Language
- CLR (Microsoft, 2000)
  - Type safe, Multi-threaded, multi-language (C#, Eiffel, Ada, Python, F#, … etc)

Turing Machine (Turing, 1931)



SECD - machine (Landin, 1962)

- The first designed to implement the lambda calculus (basis of all functional languages)
- 4 registers holding linked lists
  - S : stack
  - E : Environment
  - C : Code
  - D : Dump



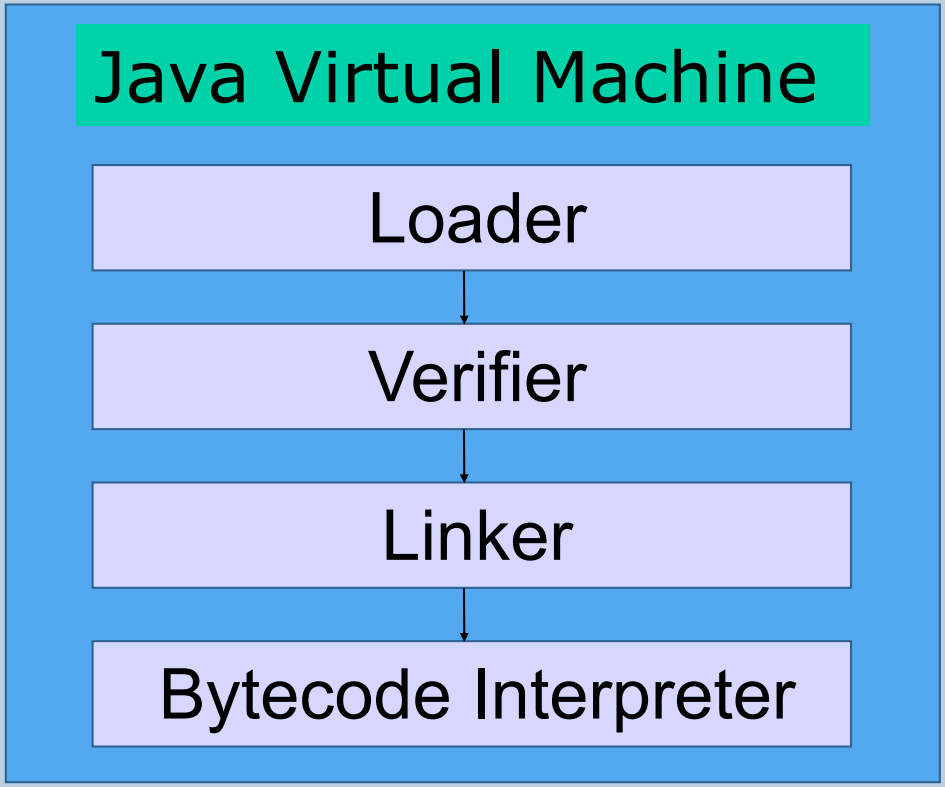
- 9 instructions: nil, ldc, ld, sel, join, ap, ret, dum, rap
- This can implement any computable function (like the Turing machine), but is much higher level

P-code machine (Wirth 1972)

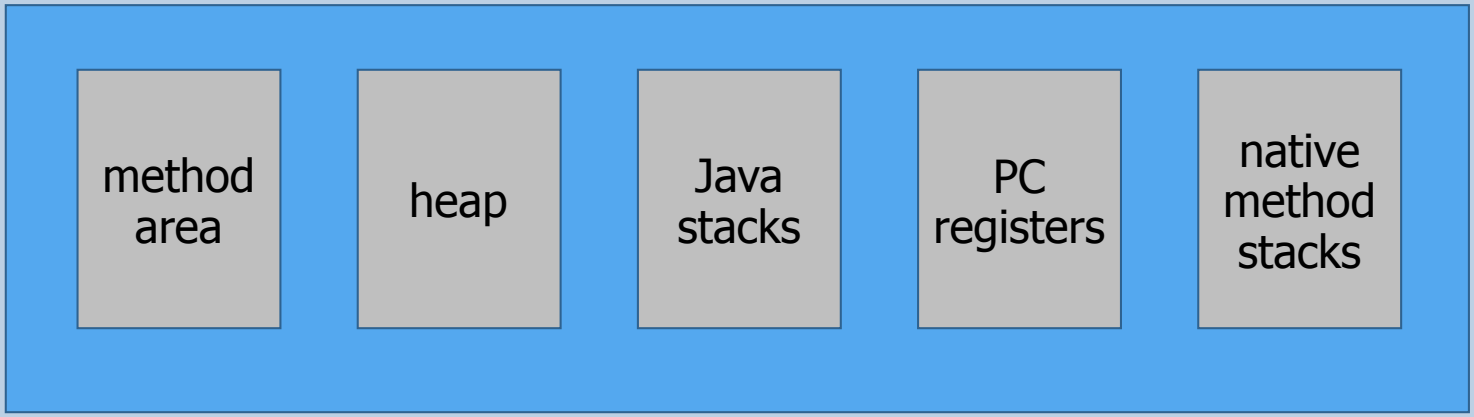
- Stack machine devoted to run code for a block structured imperative language (algol like)
- Designed specifically for the Pascal Language, a language very popular in the 70s 80s.
- Pascal was used widely for teaching introductory programming world wide (UCSD Pascal
- The first Apple Mac software (1984) was entirely written in Pascal

```
procedure interpret;
    const stacksize = 500;

    var
        p,b,t: integer; {program-, base-, topstack-registers}
        i: instruction; {instruction register}
        s: array [1..stacksize] of integer; {datastore}

    function base(l: integer): integer;
        var b1: integer;
    begin
        b1 := b; {find base l levels down}
        while l > 0 do
            begin b1 := s[b1]; l := l - 1
            end;
        base := b1
    end {base};

begin
    writeln(' start pl/0');
    t := 0; b := 1; p := 0;
    s[1] := 0; s[2] := 0; s[3] := 0;
    repeat
        i := code[p]; p := p + 1;
        with i do
        case f of
        lit: begin t := t + 1; s[t] := a end;
        opr: case a of {operator}
                0: begin {return}
                        t := b - 1; p := s[t + 3]; b := s[t + 2];
                    end;
                1: s[t] := -s[t];
                2: begin t := t - 1; s[t] := s[t] + s[t + 1] end;
```

Java Virtual Machine (Sun, 1995)

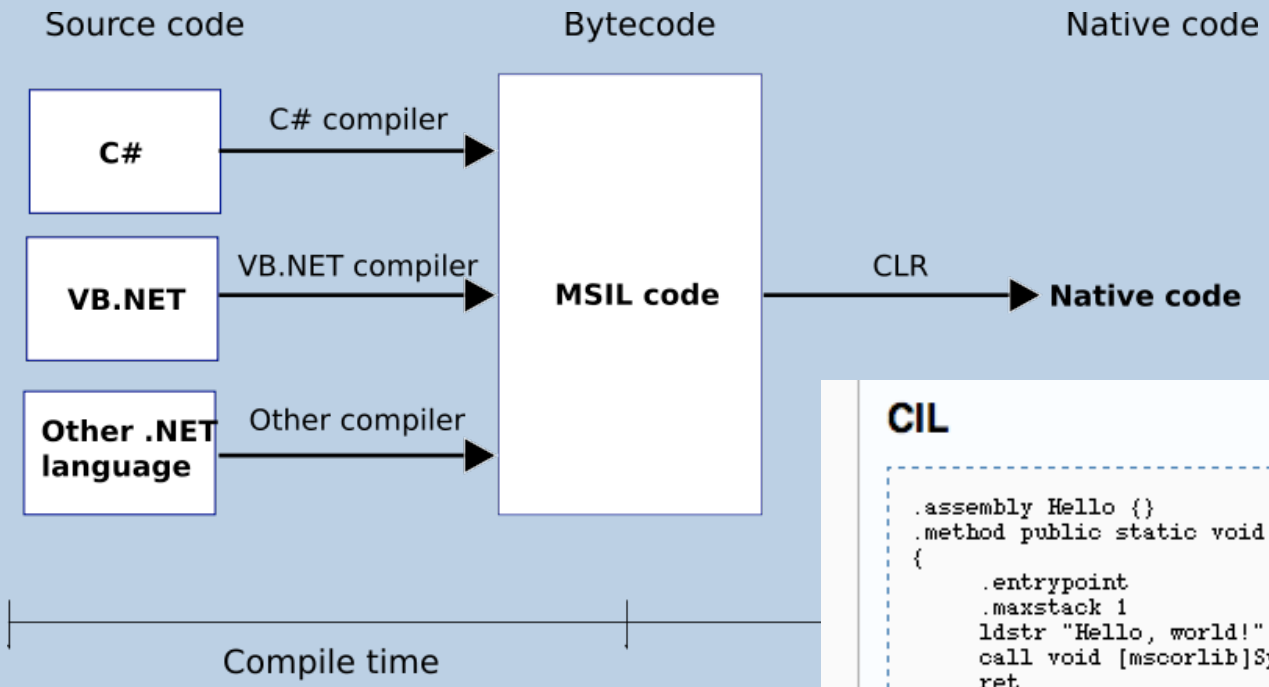



```
// Bytecode stream: 03 3b 84
// 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
iconst_0 // 03
istore_0 // 3b
iinc 0, 1 // 84 00 01
iload_0 // 1a
iconst_2 // 05
imul // 68
istore_0 // 3b
goto -7 // a7 ff f9
```



Common Language Runtime (Microsoft, 2000)

- Stack Machine designed for Microsoft .NET
- Quite independent of the source language, unlike the JVM.
- CIL - Common Intermediate Language



CIL

```
.assembly Hello {}
.method public static void Main() cil managed
{
      .entrypoint
      .maxstack 1
      ldstr "Hello, world!"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
}
```

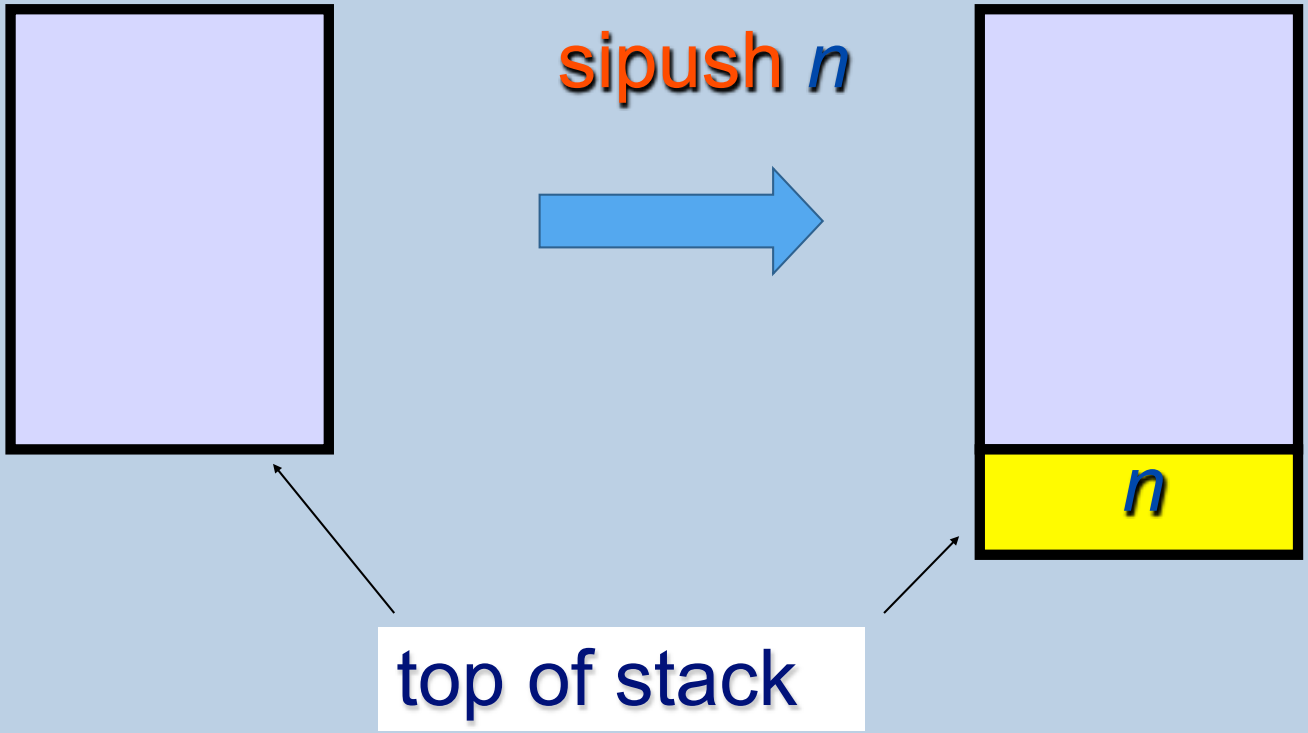
Java Virtual Machine

- Stack Machine: all instructions consume their arguments from the top of the stack and leave a result in the top of the stack

- "first" (5) machine instructions of the JVM:
  - sipush *n* : pushes the integer n on the top of the stack (tos)
  - iadd : Pops two integer values from the tos and pushes their sum
  - imul : likewise for their multiplication
  - idiv : likewise for their division
  - isub : likewise for their subtraction

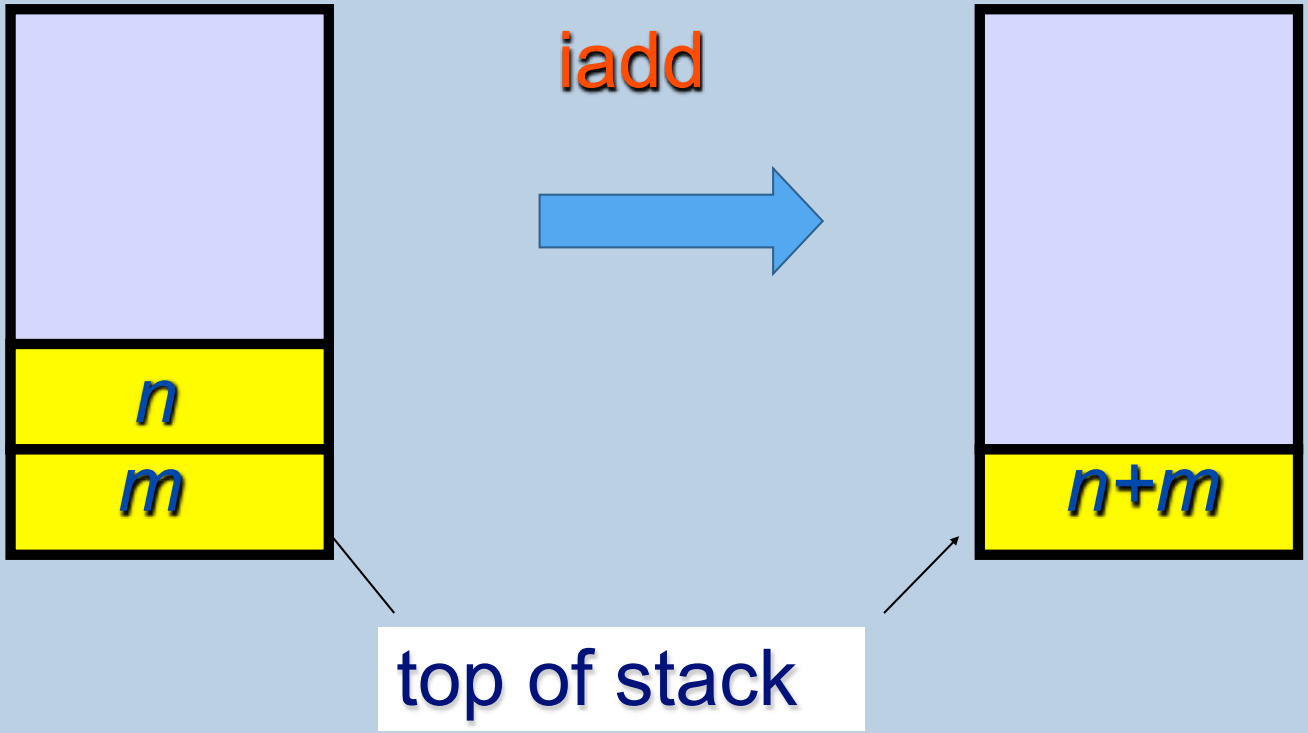
JVM instructions

- first" (5) machine instructions of the JVM:
- sipush *n*, iadd, imul, idiv, isub.
- (short integer) push (sipush *n*)



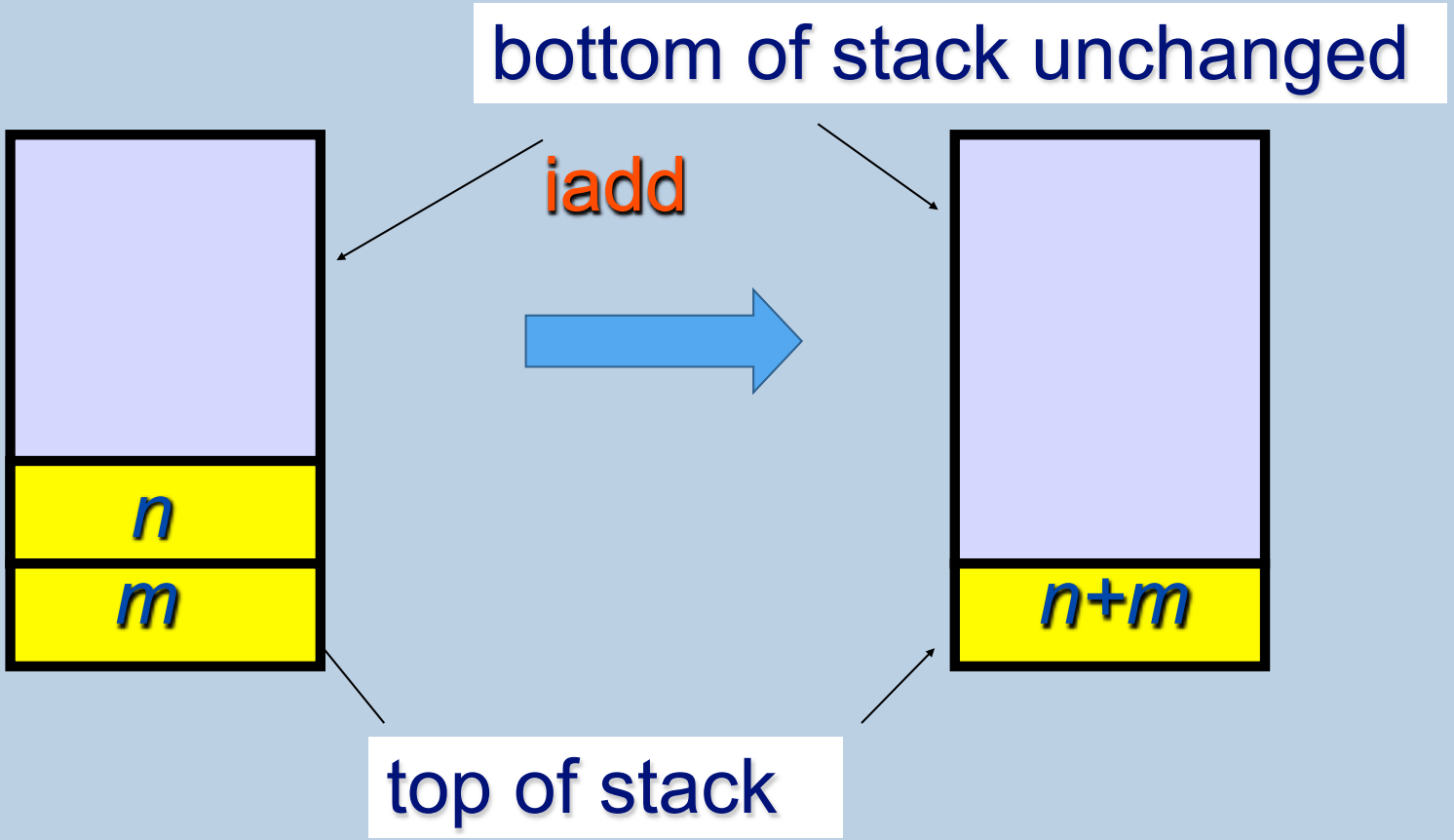
JVM instructions

- first" (5) machine instructions of the JVM:
- sipush *n*, iadd, imul, idiv, isub.
- iadd



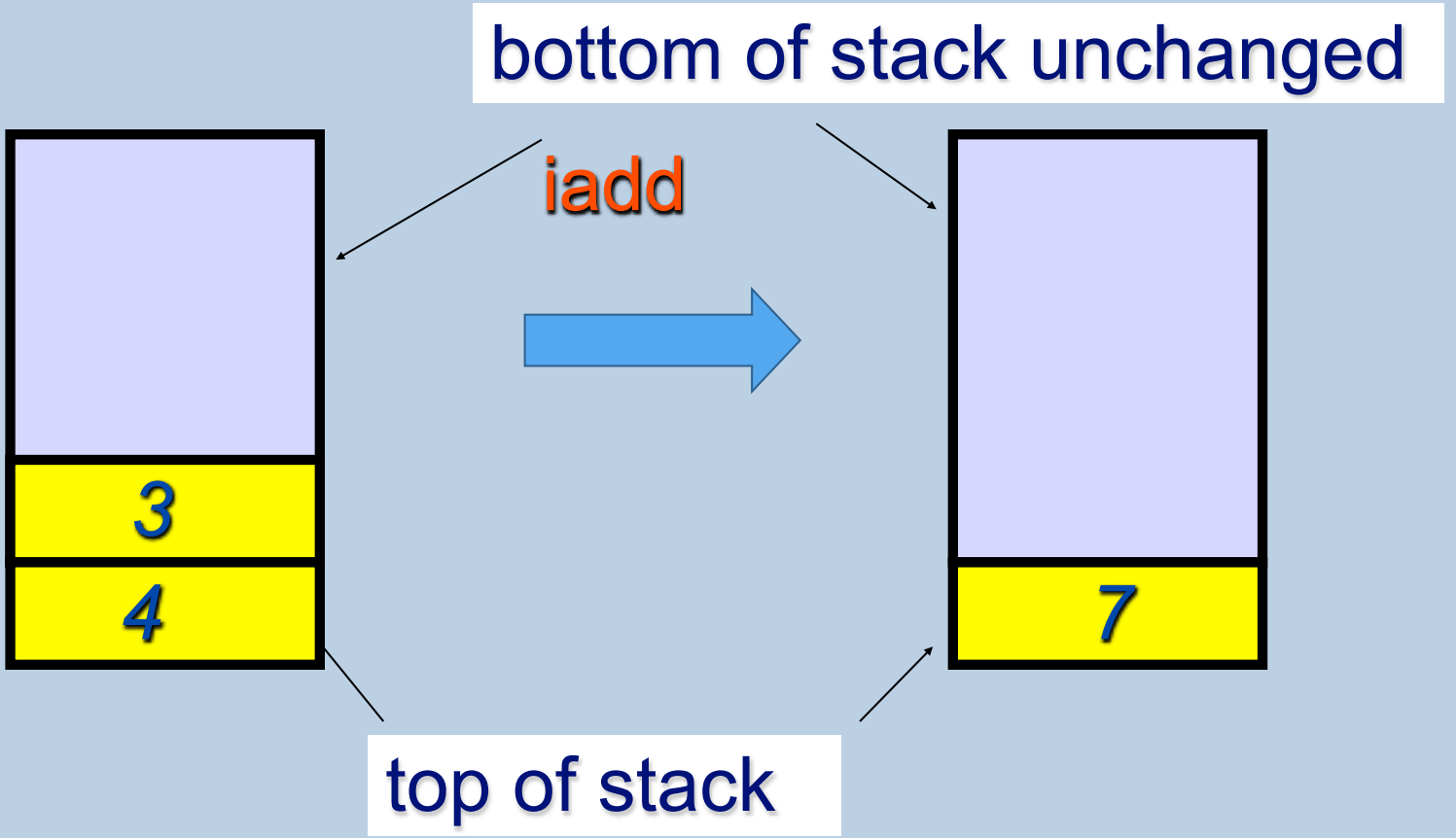
JVM instructions

- first" (5) machine instructions of the JVM:
- sipush *n*, iadd, imul, idiv, isub.
- iadd



JVM instructions

- first" (5) machine instructions of the JVM:
- sipush *n*, iadd, imul, idiv, isub.
- iadd



CALC compiler (compilation map)

- algorithm comp(E) that translates CALC expression E into a sequence of JVM instructions

$$comp : CALC \rightarrow CodeSeq$$

if E is of the form **num**(n):	comp(E) ≜ < **`sipush n`** >
if E is of the form **add**(E',E''):	s1 = comp(E'); s2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < **`iadd`** >
if E is of the form **mul**(E',E''):	v1 = comp(E'); v2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < **`imul`** >
if E is of the form **sub**(E',E''):	v1 = comp(E'); v2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < **`isub`** >
if E is of the form **div**(E',E''):	v1 = comp(E'); v2 = comp(E'');
	comp(E) ≜ s1 @ s2 @ < id**`iv`** >

CALC compiler (compilation map)

- algorithm comp(E) that translates CALC expression E into a sequence of JVM instructions

$$comp : CALC \rightarrow CodeSeq$$

comp(**num**(n)) ≜ < **ldc.i4 n** >

comp(**add**(E',E'')) ≜ comp(E') @ comp(E'') @ < **add** >

comp(**mul**(E',E'')) ≜ comp(E') @ comp(E'') @ < **mul** >

comp(**sub**(E',E'')) ≜ comp(E') @ comp(E'') @ < **sub** >

comp(**div**(E',E'')) ≜ comp(E') @ comp(E'') @ < **div** >

CALC compiler (compilation map)

- algorithm comp(E) that translates CALC expression E into a sequence of JVM instructions

$$comp : CALC \rightarrow CodeSeq$$

```
let rec comp e = match e with
      Num(n) -> ["sipush "^(string_of_int n)]
    | Add(e1,e2) -> (comp e1)@(comp e2)@["iadd"]
    | Mul(e1,e2) -> (comp e1)@(comp e2)@["imul"]
    | Sub(e1,e2) -> (comp e1)@(comp e2)@["isub"]
    | Div(e1,e2) -> (comp e1)@(comp e2)@["idiv"]
```

OCAML code

Compiler correction property

- algorithm comp(E) that translates CALC expression E into a sequence of JVM instructions

$$comp : CALC \rightarrow CodeSeq$$

- Correction invariant for the compiler map:
- execution of comp(E) code starting in a JVM state with stack *p* always terminates in a JVM state push(*v*,*p*), where *v* = eval(E).

CALC compiler (compilation map)

- first" (5) machine instructions of the JVM:
- sipush *n*, iadd, imul, idiv, isub.
- Comp("2+2*(7-2)") =
- Comp(add(num(2),mul(num(2),sub(num(7),num(2)))))

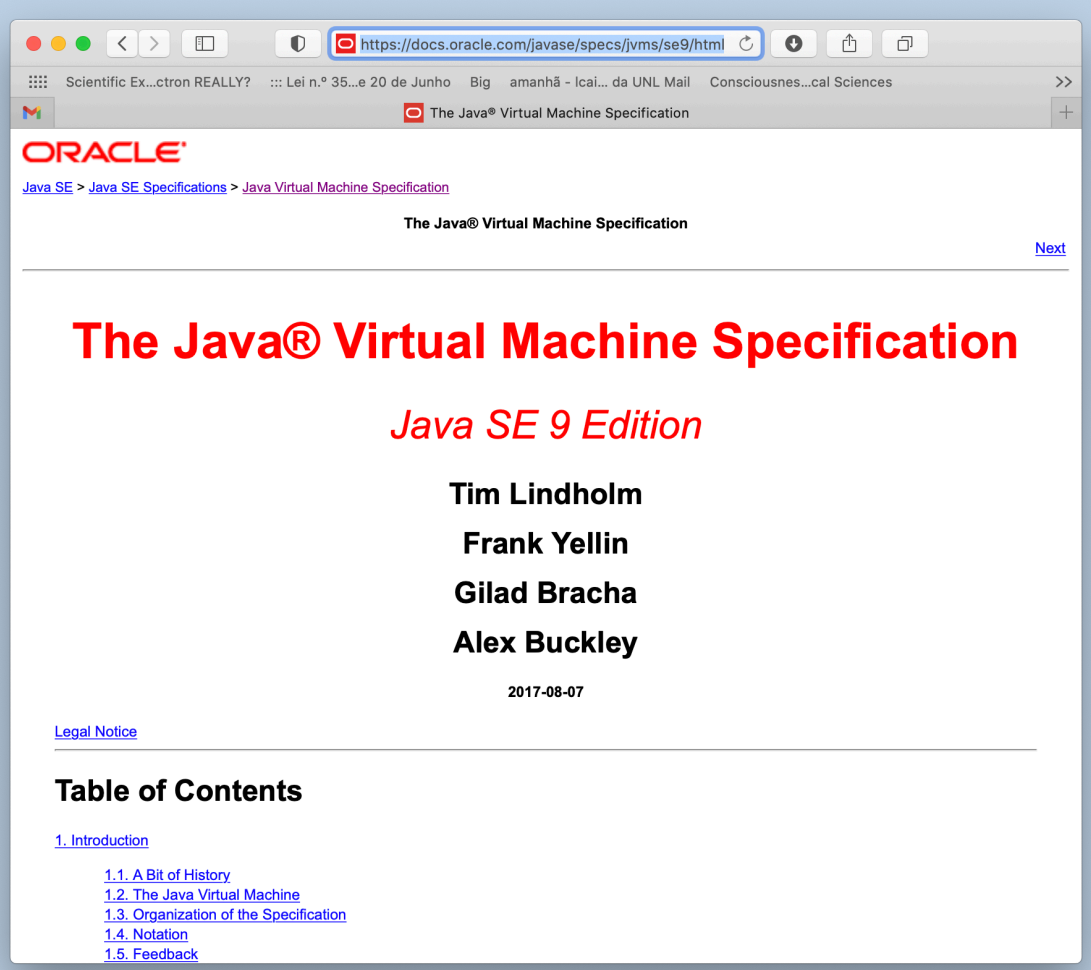
```
sipush 2
sipush 2
sipush 7
sipush 2
isub
imul
iadd
```

CALC compiler (compilation map)

- Comp(add(num(2),mul(num(2),sub(num(7),num(2))))) =

Java Virtual Machine (Sun, 1995)

- https://docs.oracle.com/javase/specs/jvms/se9/html/index.html



Interpretação e Compilação de Linguagens (de Programação)

Part 2A
Interpreters and Compiler Basics: Simple Expressions

Mestrado em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Naming

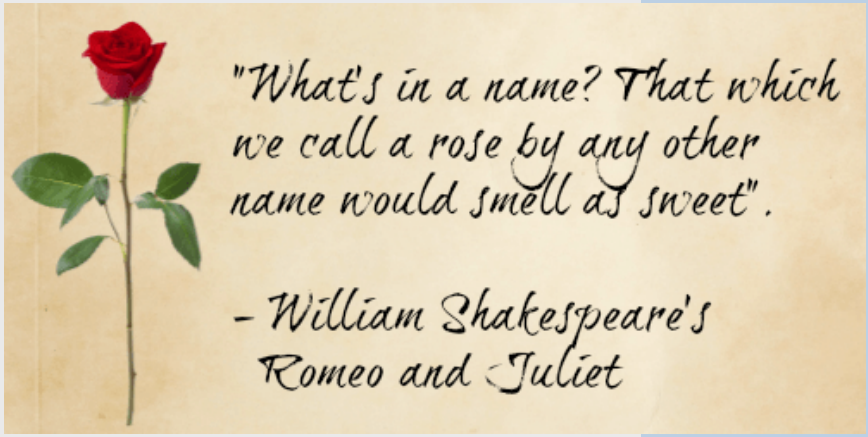
Names are the first tool one uses to introduce abstraction in a programming language (and any language in fact!).

Names allows us to refer to complex things in a concise way!

A name / identifier used in some expression or program always denotes a value previously defined.

Fundamentally, the meaning of a program fragment with names is obtained by replacing each name with the value assigned to it in its definition.

- Literals versus names
- Binding (declaration) of names
- Scope of a definition
- Occurrences of names (free, bound, binding)
- Open and closed code fragments
- Fundamental construct **def** id=E **in** E **end**.
- Language with definitions: CALCI.
- Interpreter using substitution
- Interpreter using environments



Naming Syntax

- Literals
  - Denote fixed values in every context of occurrence
  - Java: `true, false, "foo", float`
  - OCAML: `true, false, []`
  - C: `1, 1.0, 0xFF, "hello", int`
- Identifiers
  - Denote values that depend of the context of occurrence
  - In programming languages, identifiers are names for defined constants, variables, functions, methods, classes, modules, types, etc…
  - Java: `x2, y, Count, System.out`
  - C: `printf`

Binding and Scope

- The association between an identifier and the value it denotes is called a *binding*.
- A *binding* between an identifier to the value associated is always established in a well-defined syntactical context (some zone of the program text) and is created by a program construct called a declaration
- The syntactical context (zone of the program text) in which the binding is established is called the *scope* of the binding / declaration.

Binding and Scope

- The identifier **x** denotes (the address of) a memory cell

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j+y;
         z += x;
      }
      return z;
}
```

Binding and Scope

- The identifier **x** denotes (the address of) a memory cell

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
      int x=j+y;
      z += x;
    }
    return z;
}
```

scope of the binding

Binding and Scope

- The identifier `j` denotes (the address of) a memory cell

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j+y;
         z += x;
      }
      return z;
}
```

Binding and Scope

- The identifier `j` denotes (the address of) a memory cell

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
      }
      return z;
}
```

scope of the binding

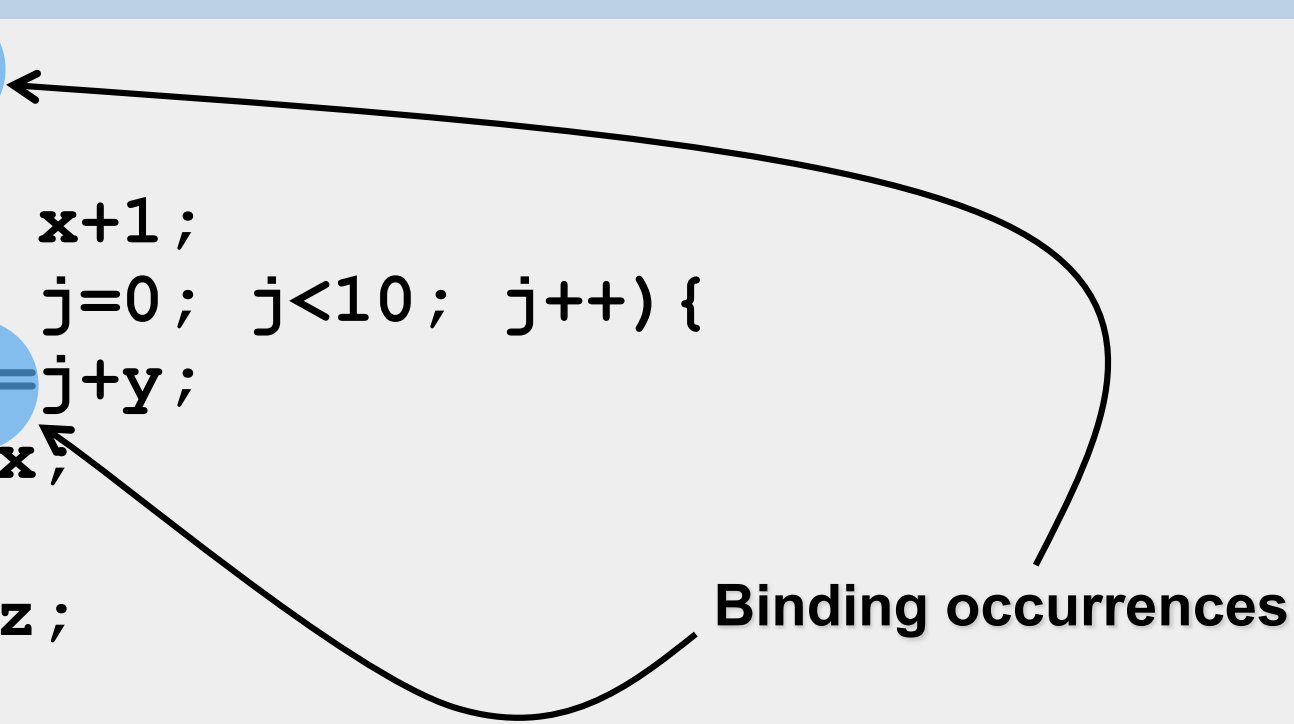
Parts of a Scope

- The binding of an identifier X to its denotation (value, memory address, etc) always involve the following ingredients:
  - A (single!) binding occurrence of the identifier X
    in general, it corresponds to the part of the program text that initialises the binding, where the binding becomes active
  - The scope of the binding
    This is the part (zone of the program text) in which the binding introduced by the binding occurrence is active
  - Several bound occurrences
    All occurrences of X, distinct from the binding occurrence, that lie inside the scope

Binding and Bound Occurrences

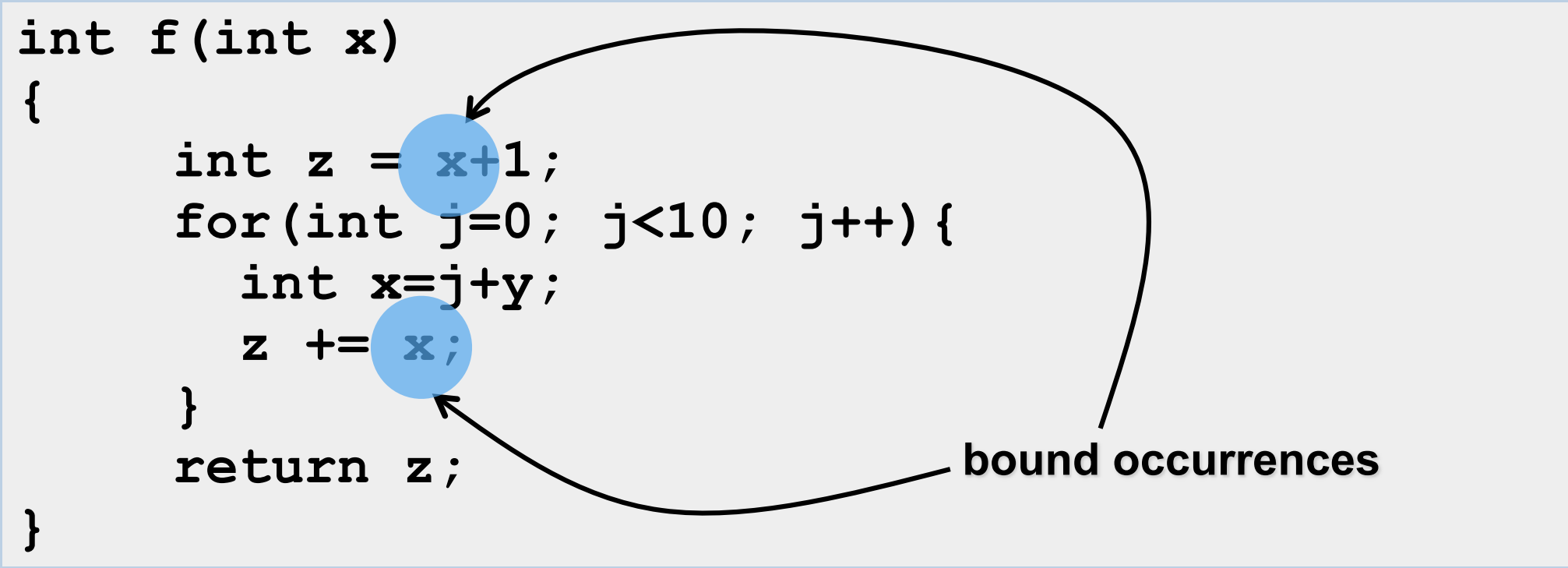
- Occurrences of name **x**

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j+y;
         z += x;
      }
      return z;
}
```



Binding and Bound Occurrences

- Occurrences of name **x**



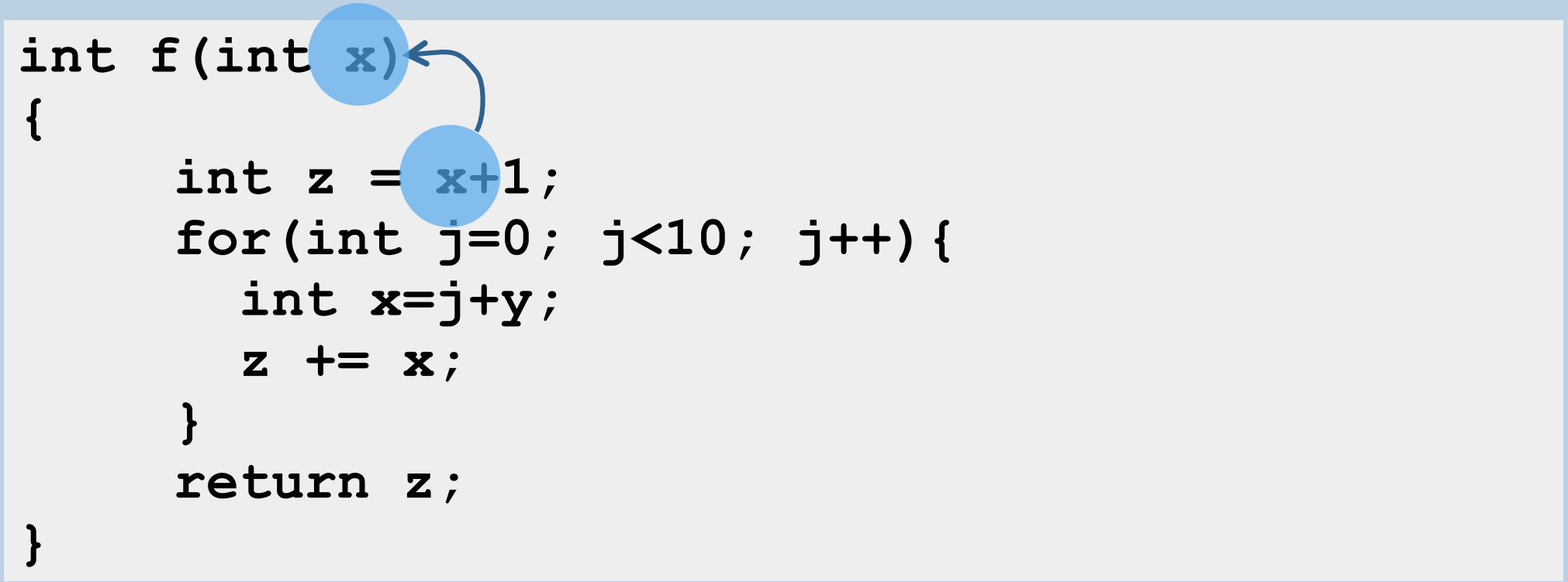
Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

```
int f(int x)
{
      int z = x+K;
      for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
      }
      return z;
}
```

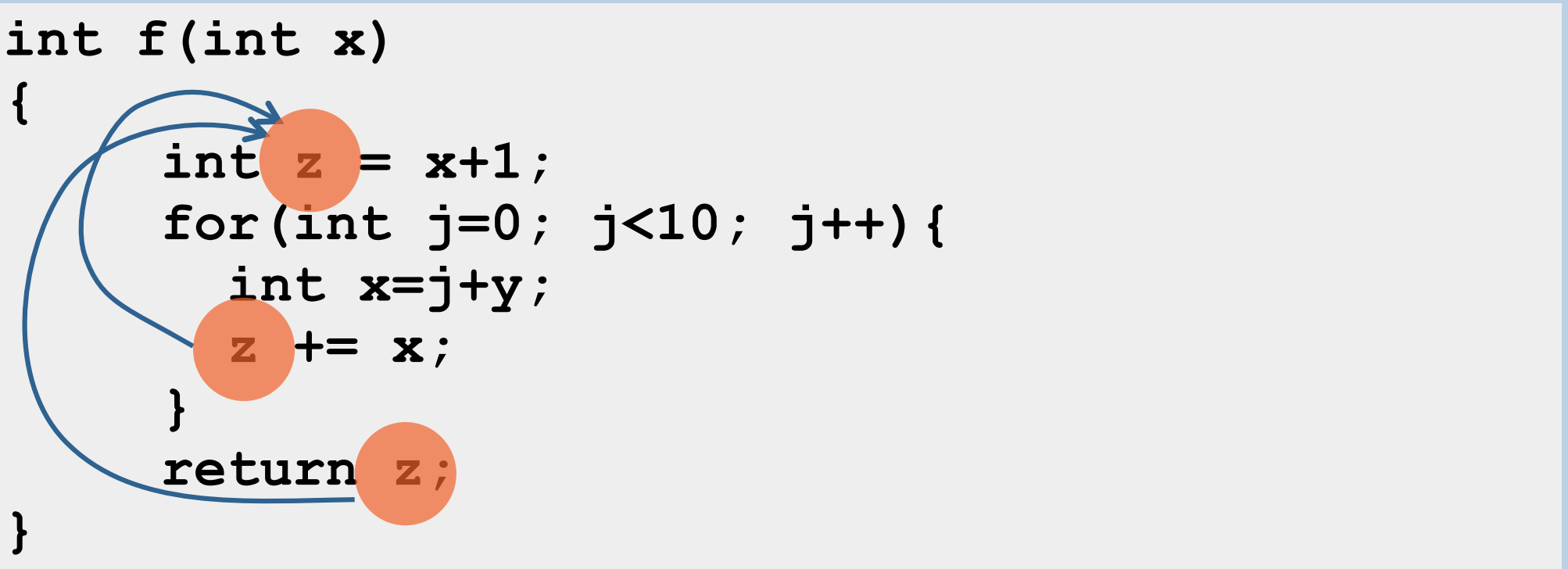
Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)



Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)



Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j+y;
         z += x;
      }
      return z;
}
```

Bound Occurrences

- For each bound occurrence there is one and only one binding occurrence (de one occurring in the declaration)

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j+y;
         z += x;
      }
      return z;
}
```

Free occurrences

- Any occurrence of an identifier that is not binding nor bound is said **free**

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j+y;
         z += x;
      }
      return z;
}
```

Open and Closed fragments

- A program fragment is said to be **open** if it contains free occurrences of identifiers
- Otherwise, a program fragment is said to be **closed** that is, if it does not contain free occurrences of identifiers
- Open fragments (examples):

```
void f(int x)                                          C
{     int i;
      for(int i=0;i<TEN;i++) x+=i;
      printf("%d\n",x);
}
```

```
let x=1 in (f x)                                       OCaml
```

Open and Closed fragments

- A program fragment is said to be **open** if it contains free occurrences of identifiers
- Otherwise, a program fragment is said to be **closed** that is, if it does mot contain free occurrences of identifiere
- Open fragments (examples):

```
void f(int x)                                    C
{     int i;
      for(int i=0;i<TEN;i++) x+=i;
      printf("%d\n",x);
}
```

free occurrence

```
let x=1 in (f x)                                 OCaml
```

free occurrence

Semantics of open fragments

- The meaning of a program fragment can only be computed if the value of evert free identifier is known.
- The definition of a compositional semantics for languages with declared identifiers has to consider open fragments.
  For instance, the C block

```
{ int x = 2 ; x = x+2 }
```

  is closed but contains open fragments (e.g., x+2).

- In general a complete program (closed fragment) contains open fragments (inside declarations).

Environment

- A closed program necessarily provides bindings all free ocorrentes that inside it, (they must appear in the scope of declarations!).

Given any fragment $\mathcal{E}$ inside a program $\mathcal{P}$, we call **environment of $\mathcal{E}$ in $\mathcal{P}$** to the set of all bindings in which scope $\mathcal{E}$ occurs.

Environment (Quiz)

- What is the environment of subexpression "x+1"?

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
        int x=j;
        z+=x;
      }
      return z;
}
```

Environment (Quiz)

- What is the environment of subexpression "x+1"?

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
        int x=j;
        z+=x;
      }
      return z;
}
```

x -> par(0)

Environment (Quiz)

- What is the environment of subexpression "z+=x"?

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j;
         z+=x;
      }
      return z;
}
```

Environment (Quiz)

- What is the environment of subexpression "z+=x"?

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
        int x=j;
        z+=x;
      }
      return z;
}
```

x -> par(0)
z -> loc(0)

Environment (Quiz)

- What is the environment of subexpression "`return z`"?

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
         int x=j;
         z+=x;
      }
      return z;
}
```

Environment (Quiz)

- What is the environment of subexpression "`return z`"?

```
int f(int x)
{
      int z = x+1;
      for(int j=0; j<10; j++){
        int x=j;
        z+=x;
      }
      return z;
}
```

z -> loc(0)

The language CALCI

- CALCI extends our basic expression language CALC with general declarations **def**:

```
def Id = Exp1 in Exp2 end
```

In a **def** expression the first occurrence of *Id* **is** binding, with scope *Exp2*

- A CALCI program is a closed expression of CALCI.

Example:

```
def x=2 in def y=x+2 in (x+y) end end
```

The language CALCI (abstract syntax)

- CALCI AST constructors: num, add, mul, div, sub, id, def

num:	Integer → CALCI
id:	String → CALCI
add:	CALCI × CALCI → CALCI
mul:	CALCI × CALCI → CALCI
div:	CALCI × CALCI → CALCI
sub:	CALCI × CALCI → CALCI
def:	String × CALCI × CALCI → CALCI

The language CALCI (concrete syntax)

- CALCI AST constructors: num, add, mul, div, sub, id, def

```
def x = 2 in
  (def x = x+2
     in
       x + x
   end) + x
end
```

The language CALCI (concrete syntax)

- AST CALCI com os construtores: num, add, mul, div, sub, id, def

```
def x = 2
      z = 2 * x
  in
     def y = def z = x+2 in z+z end
     in
        y + def y = 2+x in y end
    end
end
```

Semantics of CALCI (first definition)

The semantics of CALCI may be defined by giving a computable function I which assigns a definite meaning to each program (fragment)

$$I : CALC \rightarrow Integer$$

CALC = set of all programs (closed)

DENOT = set of all meanings (denotations)

CALC Interpreter (evaluation map)

- Algorithm eval(E) that computes the denotation (integer value) of any CALCI expression:

$$\text{eval} : \text{CALC} \rightarrow \text{Integer}$$

eval(**num**(*n*))	≜ *n*
eval(**add**(E1,E2))	≜ **eval**(E1) + **eval**(E2)
eval(**mul**(E1,E2))	≜ **eval**(E1) * **eval**(E2)
…	
eval(**def**(s, E1, E2))	≜ { V = eval(E1); G = **substv**(s, E2,V); **eval**(G); }

Fundamentally, the meaning of a program with names is always obtained by replacing each name with the value assigned to it in its definition.

The substitution function

substv(s, E, V)

Computes the expression (AST) that results from replacing in program (AST) E all free occurrences of identifier s by value V.

Examples (what does substv do?)

subst(s, s+s+2, 4) = 4+4+2

subst(y, def x=y in def y=2 in x+y, 2) =

def x=2 in def y=2 in x+y

Definition of Substv function (on ASTs)

substv(s, **num**(*n*), V) ≜ num(*n*);

substv(s, **id**(*s*), V) ≜ V;

substv(s, **add**(E1, E2),F) ≜ add(**substv**(s, E1, V), **substv**(s, E2 , V));

…

```
substv(s, def(s', E1, E2), V) ≜ if s = s'
                                   { G = substv(s, E1, V);
                                     def(s, G, E2);  }
                                 else
                                 { G = substv(s, E1, V);
                                   def(s', G, substv(s, E2, V)); }
```

CALC Interpreter (evaluation map)

- Algorithm eval(E) that computes the denotation (integer value) of any CALCI expression:

$$\text{eval} : \text{CALC} \rightarrow \text{Integer}$$

eval(**num**(*n*))	≜ *n*
eval(**add**(E1,E2))	≜ **eval**(E1) + **eval**(E2)
eval(**mul**(E1,E2))	≜ **eval**(E1) * **eval**(E2)
…	
eval(**def**(s, E1, E2))	≜ { V = eval(E1); G = **substv**(s, E2,V); **eval**(G); }

- Note: we don't need to define the case **eval**(**id**(*s*)). Why ?

Semantics of CALCI (better definition)

- The substitution-based semantics of CALCI is very simple and intuitive from the perspective of specification because it is very simple, and conforms to the essential meaning of names.

$$\text{eval} : \text{CALCI} \rightarrow \text{Integer}$$

- However, it is not efficient, requires runtime manipulation of ASTs and does not scale well for compilation.
- Using a notion of runtime environment (or spaggetti stack) the effect of explicit syntactical substitution can be performed in a lazy way.

Semantics of CALCI (better definition)

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCI expression:

$$\text{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$$

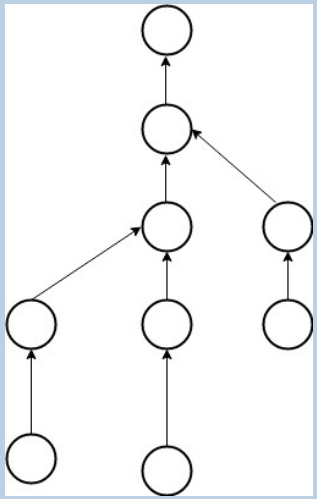
CALCI = open programs

ENV = environments

Integer = meanings (denotations)

The Environment as an ADT

- In practice, it is convenient to implement environments using a mutable stack-like data structure called a "**spaghetti stack**".
- **NOTE**: In block structures languages (eg., in all "decent" modern languages) the addition and remotion of biddings between identifiers and values follows a strict stack LIFO discipline.
- An environment stores all bindings relative to the current scope and all involving scopes in **frames**.
- From any environment state one may create a new "child" frame, corresponding to a new nested scope.
- Each frame links to the ancestor frame using a reference.



The Environment as an ADT

- Environment operations:

```
Environ BeginScope()
```

  - Pushes into the environment a new frame, where new bindings will be stored.
  - A given identifier can only be bound once in a given frame, but may be bound in different frames (to possibly different values).

```
Environ EndScope()
```

  - returns the father environment (pops off top frame).

```
void assoc(String id, Value val)
```

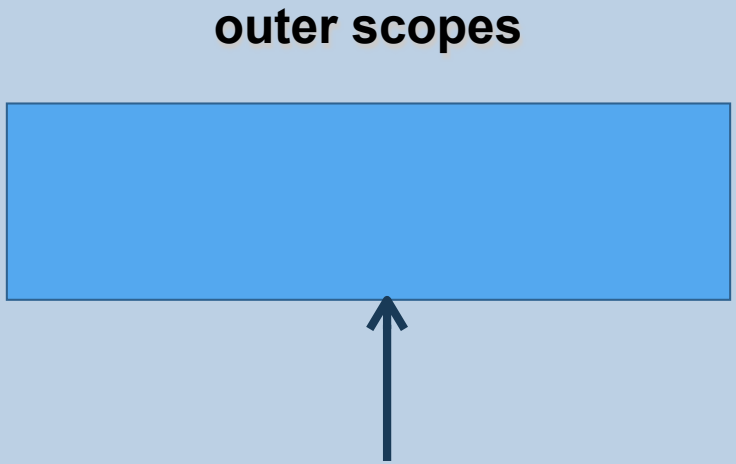
  - Adds a new binding for identifier id to the value val in the top frame of the environment (if id is not bound there yet).

```
Value Find(String id)
```

  - Returns the value associated to id in the environment, as defined by the innermost binding (the binding in the topmost frame that binds id).
  - In practice, **Find** searches for id from top to bottom following the stack frame chain, from "most recent" up, so that the appropriate scoping is respected.

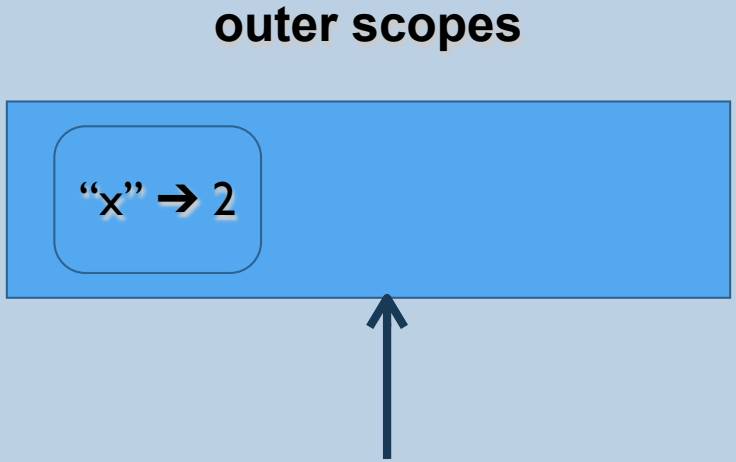
Environment in action

```
env = new Environment();
```



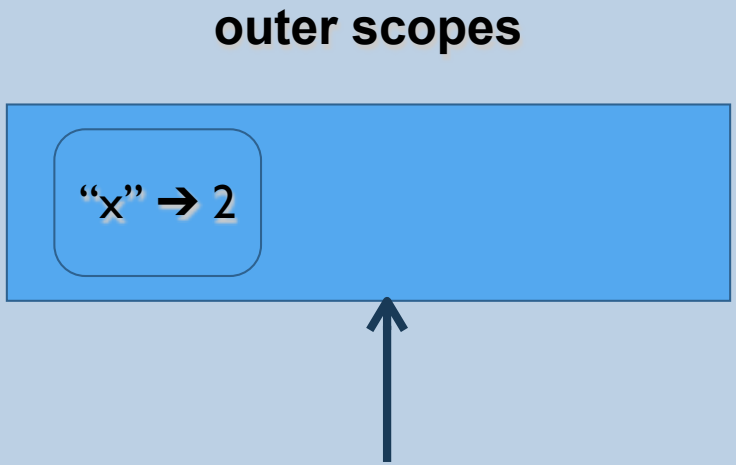
Environment in action

```
env = new Environment();

env.Assoc("x", 2);
```



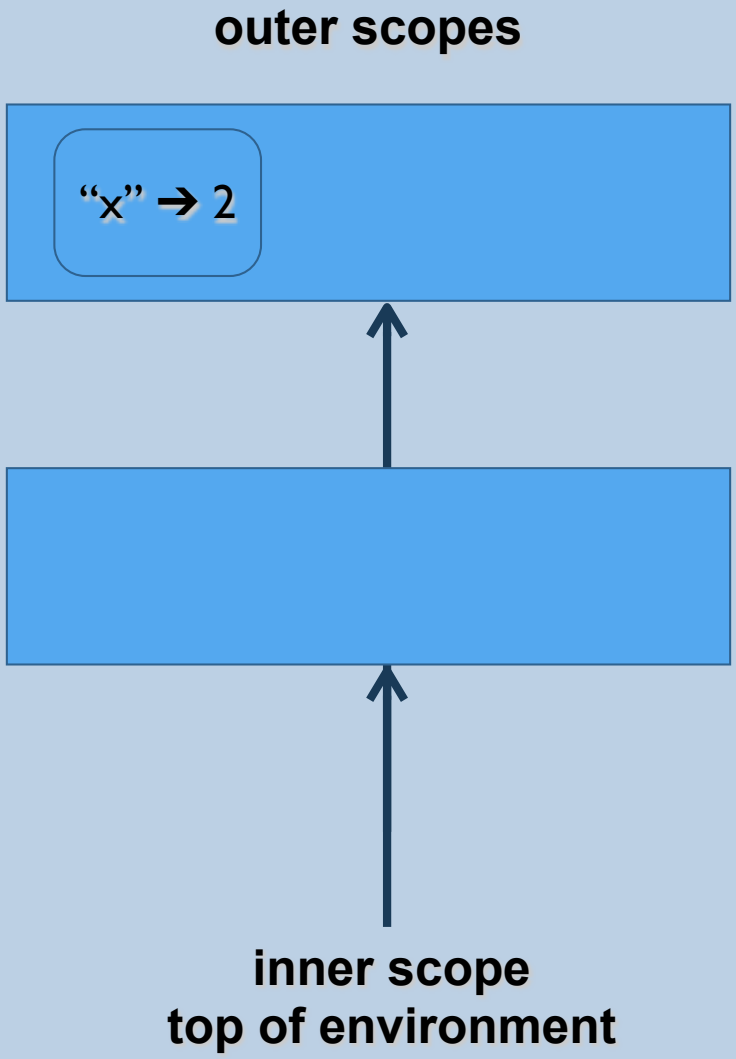
Environment in action

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // returns 2
val = env.Find("y");          // raises "Not declared"
```



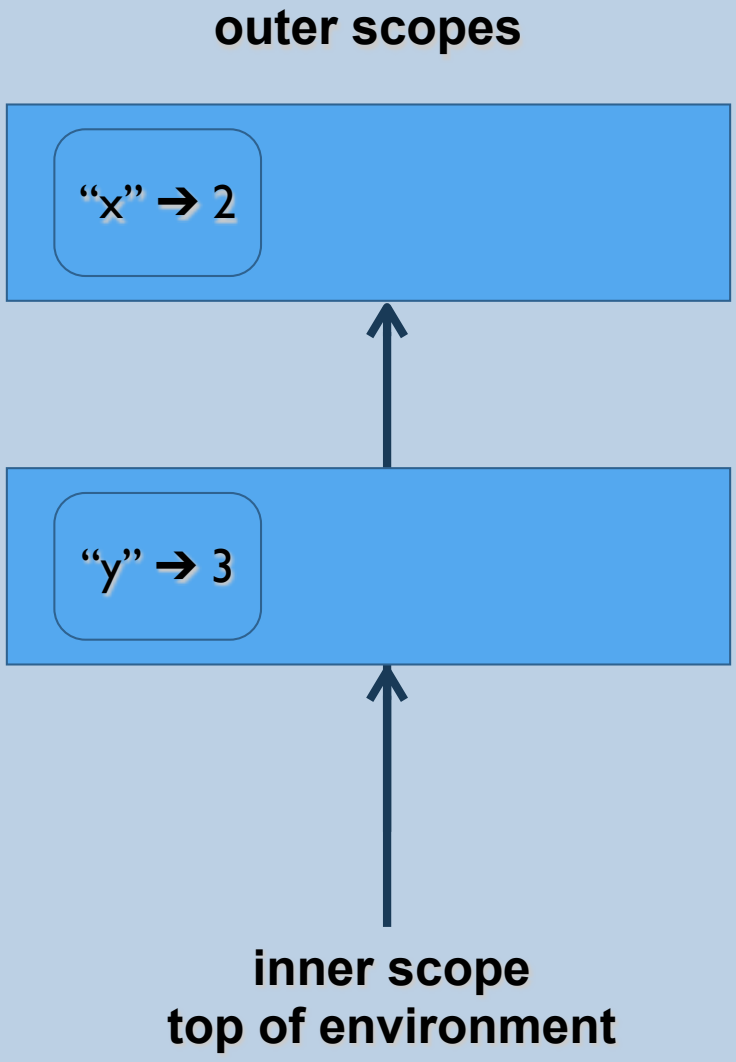
Environment in action

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // returns 2
val = env.Find("y");          // raises "Not declared"
env = env.BeginScope();
```



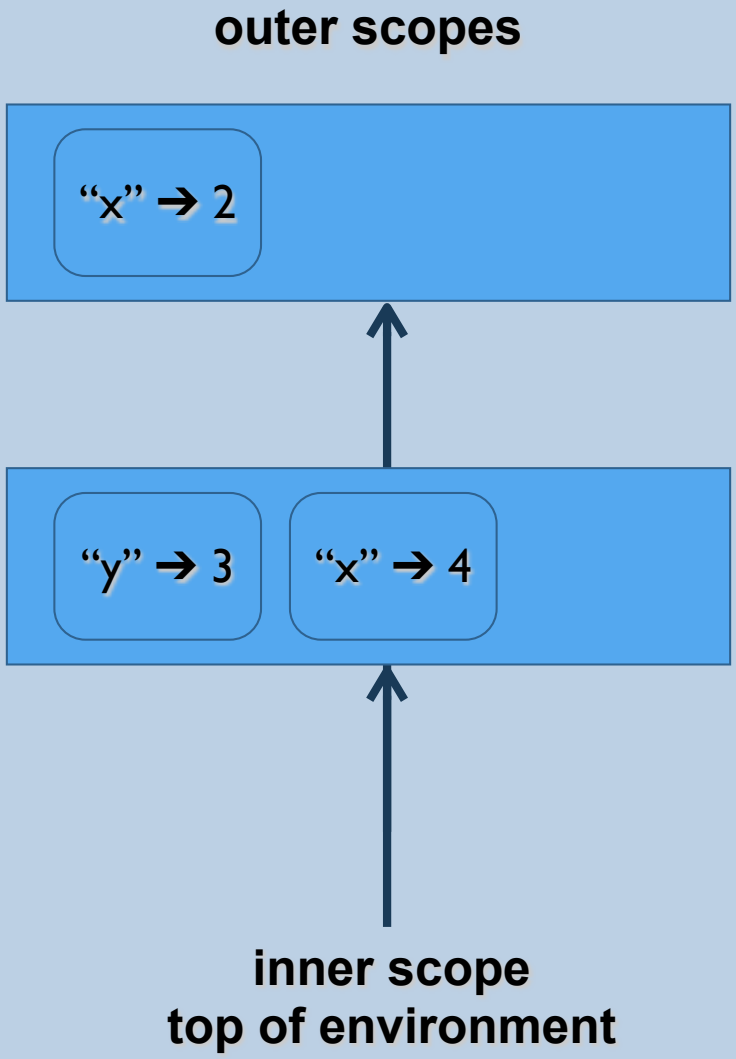
Environment in action

```
env = new Environment();
env.Assoc(“x”, 2);
val = env.Find(“x”);          // returns 2
val = env.Find(“y”);          // raises “Not declared”
env = env.BeginScope();
env.Assoc(“y”, 3);
```



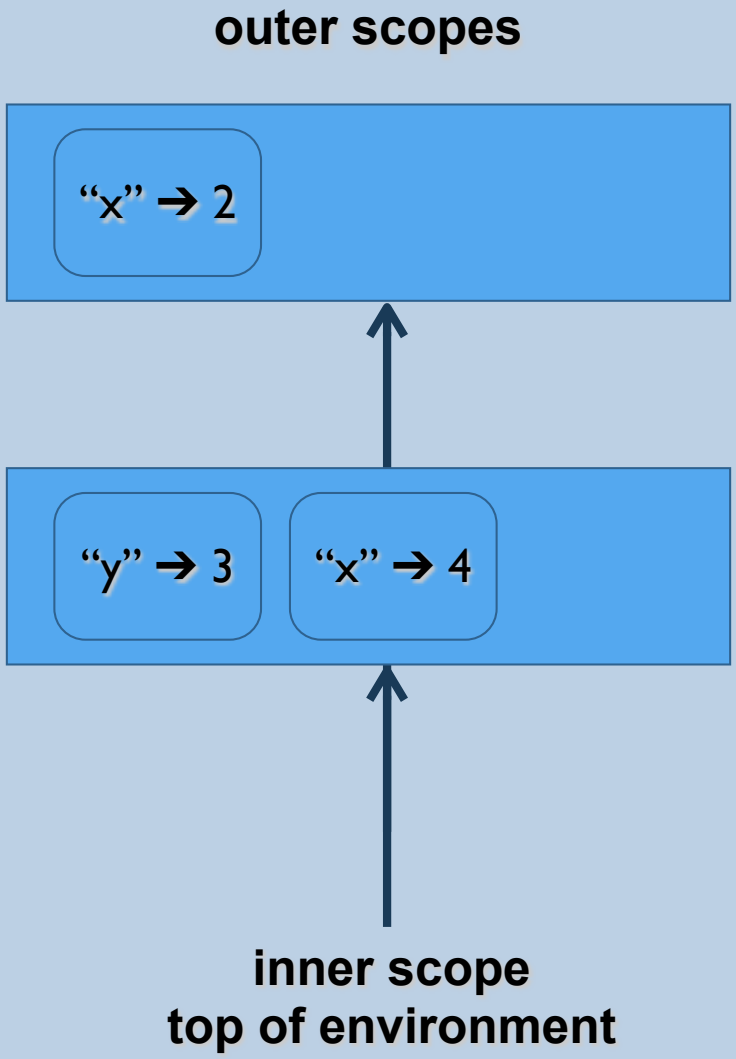
Environment in action

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // returns 2
val = env.Find("y");          // raises "Not declared"
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
```



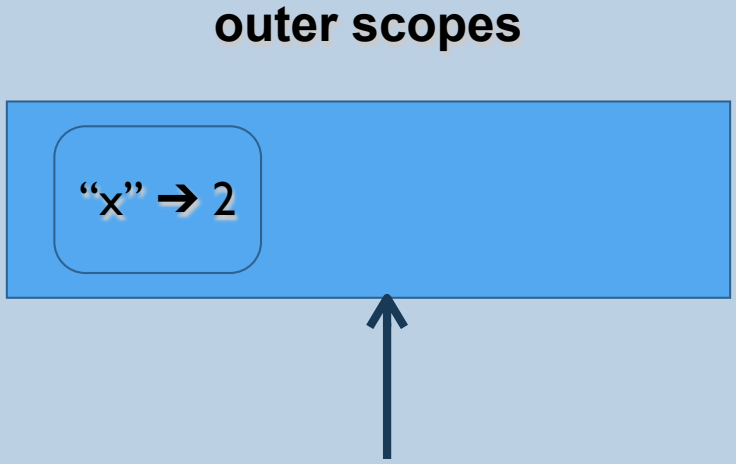
Environment in action

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // returns 2
val = env.Find("y");          // raises "Not declared"
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y");          // returns 3
val = env.Find("x");          // returns 4
```



Environment in action

```
env = new Environment();
env.Assoc(“x”, 2);
val = env.Find(“x”);          // returns 2
val = env.Find(“y”);          // raises “Not declared”
env = env.BeginScope();
env.Assoc(“y”, 3);
env.Assoc(“x”, 4);
env.Assoc(“y”, 0);        // raises “Declared twice
val = env.Find(“y”);          // returne 3
val = env.Find(“x”);          // returns 4
env=env.EndScope()
val = env.Find(“x”)          // returns 2
```



CALC Interpreter (environment based)

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCI expression:

$$\text{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$$

eval(**num**(*n*) , *env*) ≜ *n*

eval(**id**(s) , *env*) ≜ *env*.Find(s)

eval(**add**(E1,E2) , *env*) ≜ **eval**(E1, *env*) + **eval**(E2, *env*)

…

```
eval( def(s, E1, E2), env) ≜ [ v1 = eval(E1, env);
                               env = env.BeginScope();
                               env =  env.Assoc(s, v1);
                               val = eval(E2, env);
                               env = env.EndScope();
                               return val ]
```

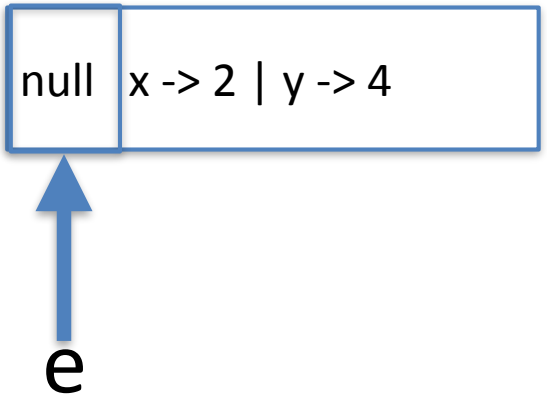
- Note: Case of id(s) implemented by lookup of the value of s in the current environment

Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```

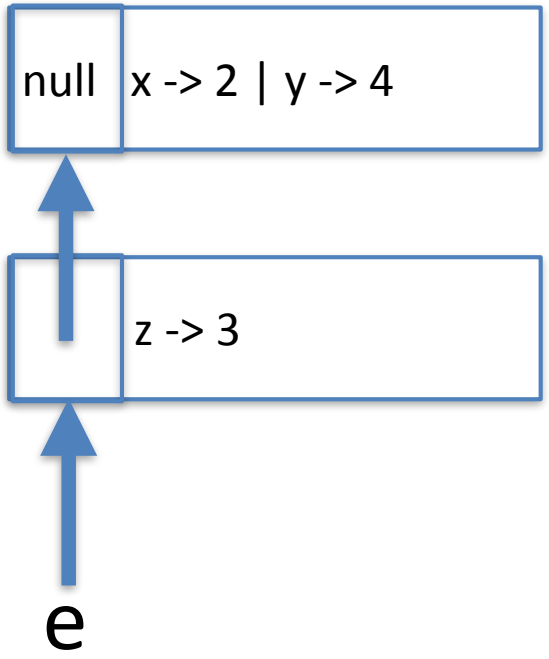
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



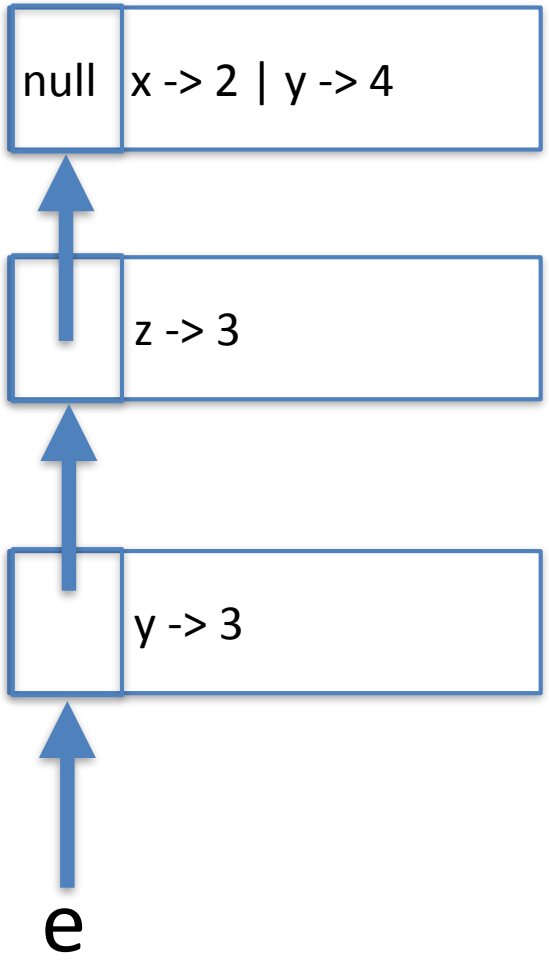
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



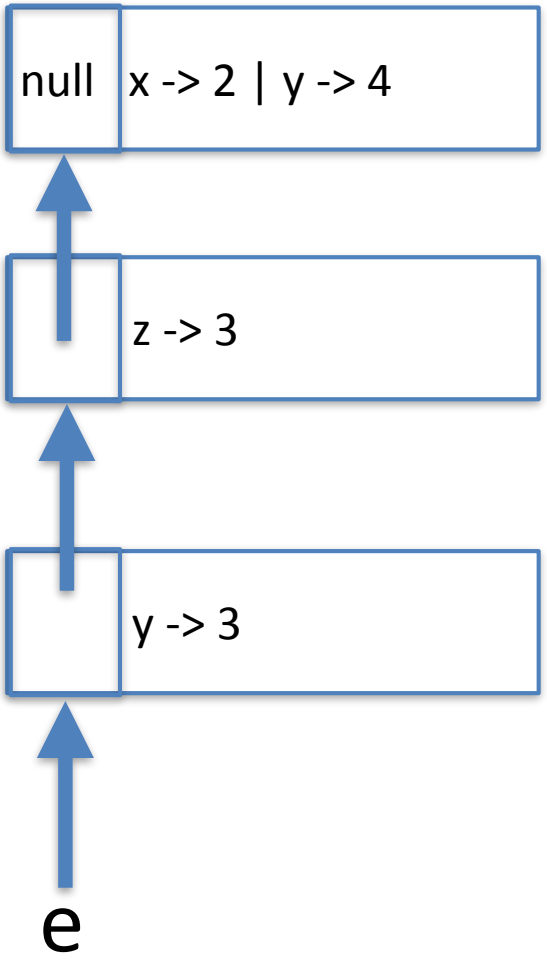
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



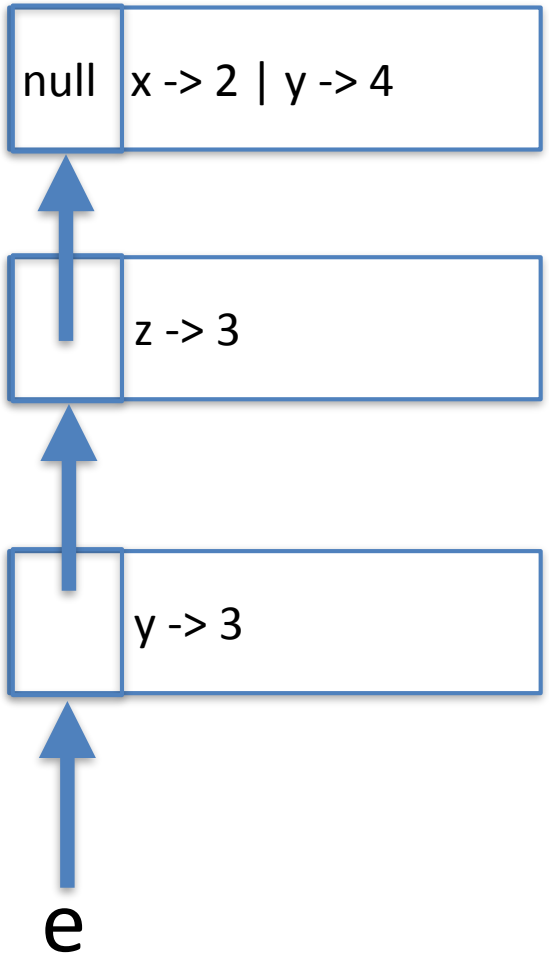
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



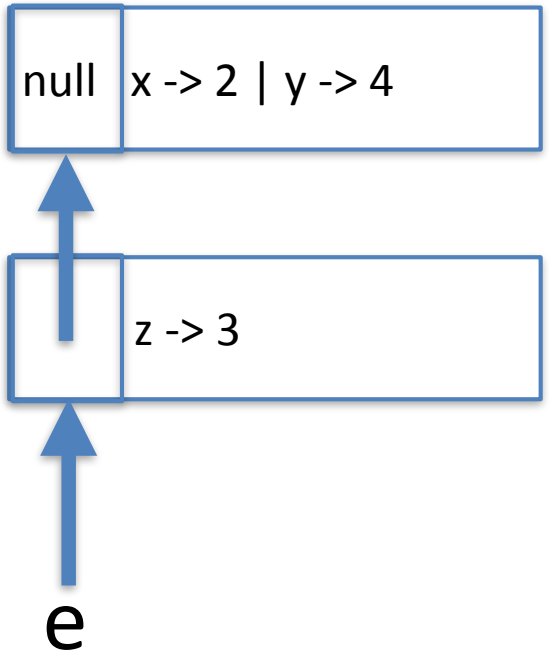
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



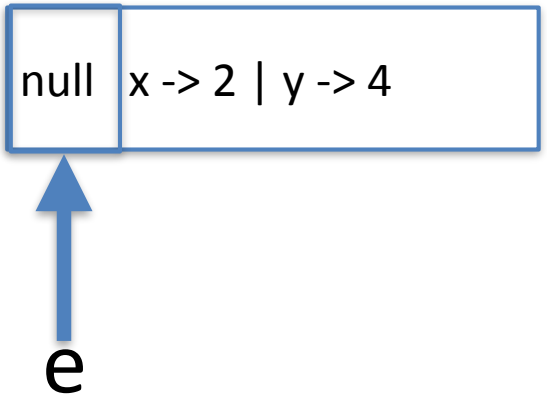
Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```



Sample execution (environment actions)

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

The language CALCI (abstract syntax)

- CALCI AST constructors: num, add, mul, div, sub, id, def

num:	Integer → CALCI
id:	String → CALCI
add:	CALCI × CALCI → CALCI
mul:	CALCI × CALCI → CALCI
div:	CALCI × CALCI → CALCI
sub:	CALCI × CALCI → CALCI
def:	List(String × CALCI) × CALCI → CALCI

CALC Interpreter (environment based)

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCI expression:

$$\text{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$$

eval(**num**(*n*) , *env*) ≜ *n*

eval(**id**(s) , *env*) ≜ *env*.Find(s)

eval(**add**(E1,E2) , *env*) ≜ **eval**(E1, *env*) + **eval**(E2, *env*)

…

```
eval( def(s, E1, E2), env) ≜ [ v1 = eval(E1, env);
                               env = env.BeginScope();
                               env =  env.Assoc(s, v1);
                               val = eval(E2, env);
                               env = env.EndScope();
                               return val ]
```

- Note: Case of id(s) implemented by lookup of the value of s in the current environment

CALC Compiler (environment based)

- Algorithm compile( ) that generates machine code for any **open** CALCI expression:

$$\text{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{CodeSeq}$$

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$$[\![E]\!]D = \ldots \text{ code sequence } \ldots.$$

- E : program fragment
- D: environment (associates identifiers to "coordinates")

$[\![E_1+E_2]\!]D =$

$[\![E_1]\!]D$

$[\![E_2]\!]D$

iadd

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$[[E_1 + E_2]]D =$
$[[E_1]]D$
$[[E_2]]D$
iadd

$[[E_1 * E_2]]D =$
$[[E_1]]D$
$[[E_2]]D$
imil

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$[\![E_1 - E_2]\!]D =$
$[\![E_1]\!]D$
$[\![E_2]\!]D$
isub

$[\![E_1 / E_2]\!]D =$
$[\![E_1]\!]D$
$[\![E_2]\!]D$
idiv

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

[[n]]D =
    sipush n

[[- E]]D =
    sipush 0
    [[E]]D
    isub

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

[[n]]D =
    sipush n

[[- E]]D =
    [[E]]D
    ineg

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

[[id]]D =

???

[[**def** x = E **in** E **end**]]D =

???

Compilation Environment (D)

The compilation environment D maps each free name of the program to be compiled to its **coordinates**:

D(x) = (d, s) where

d: the depth of the topmost stack frame (from bottom of stack) where the identifier is declared.

s: the slot in the frame where the associated value is stored

At runtime, the value of an identifier s where D(x) = (d, s) is to be found by climbing the runtime environment N-d frames, and getting the value in the s slot, where N is the depth of the current environment D.

Compilation of declarations

The JVM code generated for an expression

def x1 = E1 ... xn = En **in** E **end**

1. creates and pushes a new stack frame (heap allocated) to store the value of the n identifiers x1 ... xn

2. initialises the slot for each xi with the value of Ei

3. pops off of the frame

At all times, a reference to the top of the runtime stack environment is stored in a JVM local variable SL

Compilation of declarations

The JVM code generated for an expression

def x1 = E1 ... xn = En **in** E **end**

1. creates and pushes a new stack frame (heap allocated) to store the value of the n identifiers x1 ... xn

2. initialises the slot for each xi with the value of Ei

3. pops off of the frame

The runtime stack is a runtime JVM representation of the interpreter environment.

Compilation of declarations

The JVM code generated for an expression

def x1 = E1 ... xn = En **in** E **end**

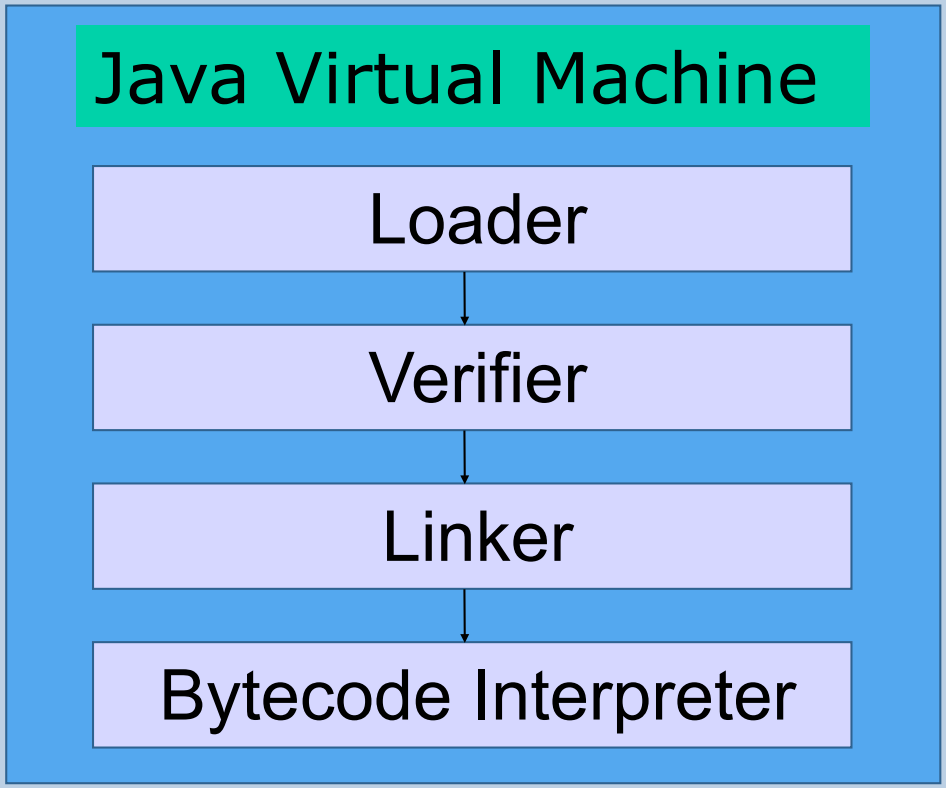
1. creates and pushes a new stack frame (heap allocated) to store the value of the n identifiers x1 ... xn

2. initialises the slot for each xi with the value of Ei

3. pops off of the frame

The generated code for an id does not need to dynamically search up in the stack, it knows its coordinates (from D(id))

JVM Architecture



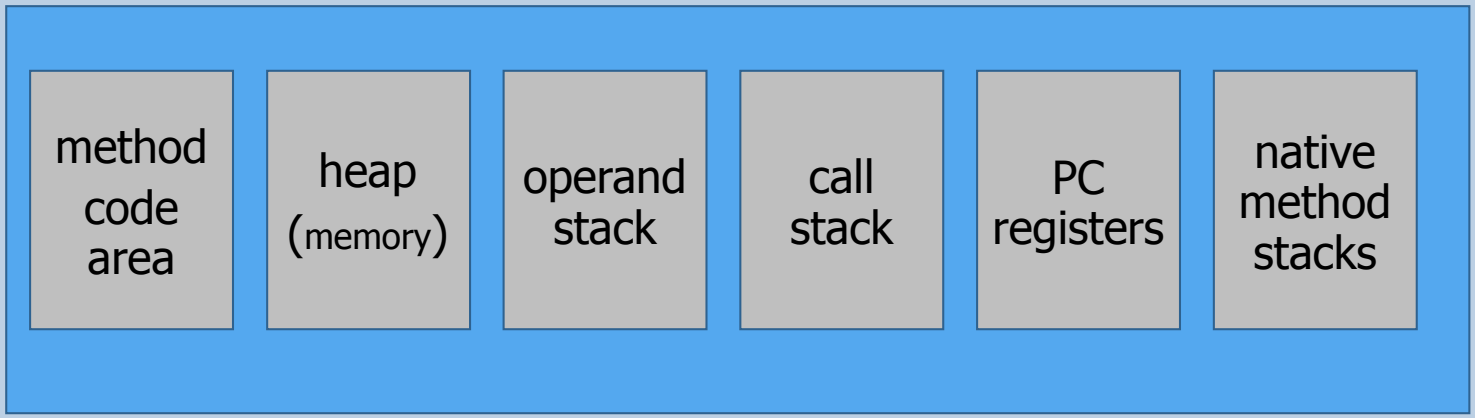
Operand Stack:

stores arguments and results of instructions

Heap:

stores dynamically allocated objects, arrays, strings, classes

….



Structure of Stack Frames (jasmin syntax)

```
class frame_id
.super java/lang/Object
.field public sl Lancestor_frame_id;
.field public x_0 type
.field public x_1 I;
..
.field public x_n I;
.end method
```

frame_id : unique type name generated by compiler

sl : holds reference to ancestor frame

ancestor_frame_id : name of the ancestor frame type

xi : slot #i (stores value of id #i in this level)

Note: a stack frame is represented a JVM class with no methods (just public fields). This is akin to a C struct.

Structure of Stack Frames (jasmin syntax)

```
class frame_id
.super java/lang/Object
.field public sl Lancestor_frame_id;
.field public x_0 type
.field public x_1 I;
..
.field public x_n I;
.end method
```

frame_id : unique type name generated by compiler

sl : also called the "static link"

ancestor_frame_id : name of the ancestor frame type

xi : slot #i (stores value of id #i in this level)

Note: a stack frame is represented a JVM class with no methods (just public fields). This is akin to a C struct.

Compilation of declarations

[[**def** x1 = E1 … xn = En **in** E **end**]]D =

frame creation and linkage into environment stack (push)

```
new frame_id
dup
invokespecial frame_id/<init>()V
dup
aload SL
putfield frame_id/sl Lcurrframetype
astore SL
```

JVM instructions

new

Operation

Create new object

Format

```
new
indexbyte1
indexbyte2
```

Forms

new = 187 (0xbb)

Operand Stack

... →

..., *objectref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in a class type. Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (§2.3, §2.4). The *objectref*, a `reference` to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

JVM instructions

dup

Operation

Duplicate the top operand stack value

Format

dup

Forms

dup = 89 (0x59)

Operand Stack

..., *value* →

..., *value*, *value*

Description

Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

The *dup* instruction must not be used unless *value* is a value of a category 1 computational type (§2.11.1).

JVM instructions

aload

Operation

Load `reference` from local variable

Format

```
aload
index
```

Forms

aload = 25 (0x19)

Operand Stack

... →

..., *objectref*

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes

The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction (§*astore*) is intentional.

The *aload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

JVM instructions

astore

Operation

Store `reference` into local variable

Format

```
astore
index
```

Forms

astore = 58 (0x3a)

Operand Stack

..., *objectref* →

...

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

Notes

The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clause of the Java programming language (§3.13).

The *aload* instruction (§*aload*) cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *astore* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

JVM instructions

putfield

Operation

Set field in object

Format

```
putfield
indexbyte1
indexbyte2
```

Forms

putfield = 181 (0xb5)

Operand Stack

..., *objectref*, *value* →

...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The class of *objectref* must not be an array. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

The referenced field is resolved (§5.4.3.2). The type of a *value* stored by a *putfield* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class, and the instruction must occur in an instance initialization method (`<init>`) of the current class (§2.9).

The *value* and *objectref* are popped from the operand stack. The *objectref* must be of type `reference`. The *value* undergoes value set conversion (§2.8.3), resulting in *value*', and the referenced field in *objectref* is set to *value*'.

Compilation of declarations

[[**def** x1 = E1 ... xn = En **in** E **end**]]D =

initialization of identifier slots in frame

```
aload SL
[[E1]]D+{x1|->t1, xn|->tn}
putfield frame_id/x1 Lt1
aload SL
[[E2]]D+{x1|->t1, xn|->tn}
putfield frame_id/x2 Lt2
....
aload SL
[[En]]D+{x1|->t1, xn|->tn}
putfield frame_id/xn Ltn
```

Compilation of declarations

[[**def** x1 = E1 ... xn = En **in** E **end**]]D =

code for definition body

[[E]]D+{x1|->t1, xn|->tn}

Compilation of declarations

[[**def** x1 = E1 … xn = En **in** E **end**]]D =

frame pop off and update of local SL

```
aload SL
getfield frame_id/sl Lcurrframetype
astore SL
```

JVM instructions

getfield

Operation

Fetch field from object

Format

```
getfield
indexbyte1
indexbyte2
```

Forms

getfield = 180 (0xb4)

Operand Stack

..., *objectref* →

..., *value*

Description

The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2). The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.

The type of *objectref* must not be an array type. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Compilation of identifier uses

[[x]]D =

Assume D(x) = (d, s)

```
aload SL
getfield frame_id/sl Lancestor_frame_id
…
getfield frame_id/sl Lancestor_frame_id
getfield frame_id/s Lxtype
```

} N-d stack up dereferences

the number k of dereference (using getfield sl) is N-d

the coordinates stored in the compilation environment are used to generate code that fetches the identifier value from the appropriate frame at runtime.

Compilation of identifier uses

[[x]]D =

Assume D(x) = (d, s)

```
aload SL
getfield frame_id/sl Lancestor_frame_id
...
getfield frame_id/sl Lancestor_frame_id
getfield frame_id/s Lxtype
```

D(x) = (d, s) where

d: the depth of the topmost stack frame (from bottom of stack) where the identifier is declared.

s: the slot in the frame where the associated value is stored

Compilation of CALCI (example code)

```
.class public frame_0
.super java/lang/Object
.field public sl Ljava/lang/Object;
.field public v0 I
.field public v1 I

.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return


.end method
```

default constructor (JVM requires it)

```
.class public frame_1
.super java/lang/Object
.field public sl Lframe_0;
.field public v0 I

.end method
```

```
def
  x  = 2
  y  = 3
in
 def
     k = x + y
 in
     x + y + k
 end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Obje
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

The language CALCI (abstract syntax)

- CALCI AST constructors: num, add, mul, div, sub, id, def

num:	Integer → CALCI
id:	String → CALCI
add:	CALCI × CALCI → CALCI
mul:	CALCI × CALCI → CALCI
div:	CALCI × CALCI → CALCI
sub:	CALCI × CALCI → CALCI
def:	List(String × CALCI) × CALCI → CALCI

The language CALCI (concrete syntax)

- Sample CALCI program

```
def x = 2
      z = 2 * x
  in
     def y = def z = x+2 in z+z end
     in
         y + def y = 2+z in y end
    end
end
```

CALC Interpreter (environment based)

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCI expression:

$$\text{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$$

eval(**num**(*n*) , *env*) ≜ *n*

eval(**id**(s) , *env*) ≜ *env*.Find(s)

eval(**add**(E1,E2) , *env*) ≜ **eval**(E1, *env*) + **eval**(E2, *env*)

…

```
eval( def(s, E1, E2), env) ≜ [ v1 = eval(E1, env);
                               env = env.BeginScope();
                               env =  env.Assoc(s, v1);
                               val = eval(E2, env);
                               env = env.EndScope();
                               return val ]
```

- Note: Case of id(s) implemented by lookup of the value of s in the current environment

CALC Compiler (environment based)

- Algorithm compile( ) that generates machine code for any **open** CALCI expression:

$$\text{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{CodeSeq}$$

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$$[\![E]\!]D = \ldots \text{ code sequence } \ldots.$$

- E : program fragment
- D: environment (associates identifiers to "coordinates")

$[\![E_1+E_2]\!]D =$

$[\![E_1]\!]D$

$[\![E_2]\!]D$

iadd

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$[[E_1 + E_2]]D =$
$[[E_1]]D$
$[[E_2]]D$
iadd

$[[E_1 * E_2]]D =$
$[[E_1]]D$
$[[E_2]]D$
imul

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$[\![E_1 - E_2]\!]D =$
$[\![E_1]\!]D$
$[\![E_2]\!]D$
isub

$[\![E_1 / E_2]\!]D =$
$[\![E_1]\!]D$
$[\![E_2]\!]D$
idiv

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

[[n]]D =
    sipush n

[[- E]]D =
    sipush 0
    [[E]]D
    isub

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

[[n]]D =
    sipush n

[[- E]]D =
    [[E]]D
    ineg

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

[[x]]D =

???

[[**def** x = E **in** E **end**]]D =

???

Compilation Environment (D)

The compilation environment D maps each free name of the program to be compiled to its **coordinates**:

D(x) = (d, s) where

d: the depth of the topmost stack frame (from bottom of stack) where the identifier is declared.

s: the slot in the frame where the associated value is stored

At runtime, the value of an identifier s where D(x) = (d, s) is to be found by climbing the runtime environment N-d frames, and getting the value in the s slot, where N is the depth of the current environment D.

Compilation of declarations

The JVM code generated for an expression

def x1 = E1 ... xn = En **in** E **end**

1. creates and pushes a new stack frame (heap allocated) to store the value of the n identifiers x1 ... xn
2. initialises the slot for each xi with the value of Ei
3. generates code for E
3. pops off of the frame

At all times, a reference to the top of the runtime stack environment is stored in a JVM local variable SL

Compilation of declarations

The JVM code generated for an expression

def x1 = E1 ... xn = En **in** E **end**

1. creates and pushes a new stack frame (heap allocated) to store the value of the n identifiers x1 ... xn
2. initialises the slot for each xi with the value of Ei
3. generates code for E
3. pops off of the frame

The runtime stack is a runtime JVM representation of the interpreter environment.

Compilation of declarations

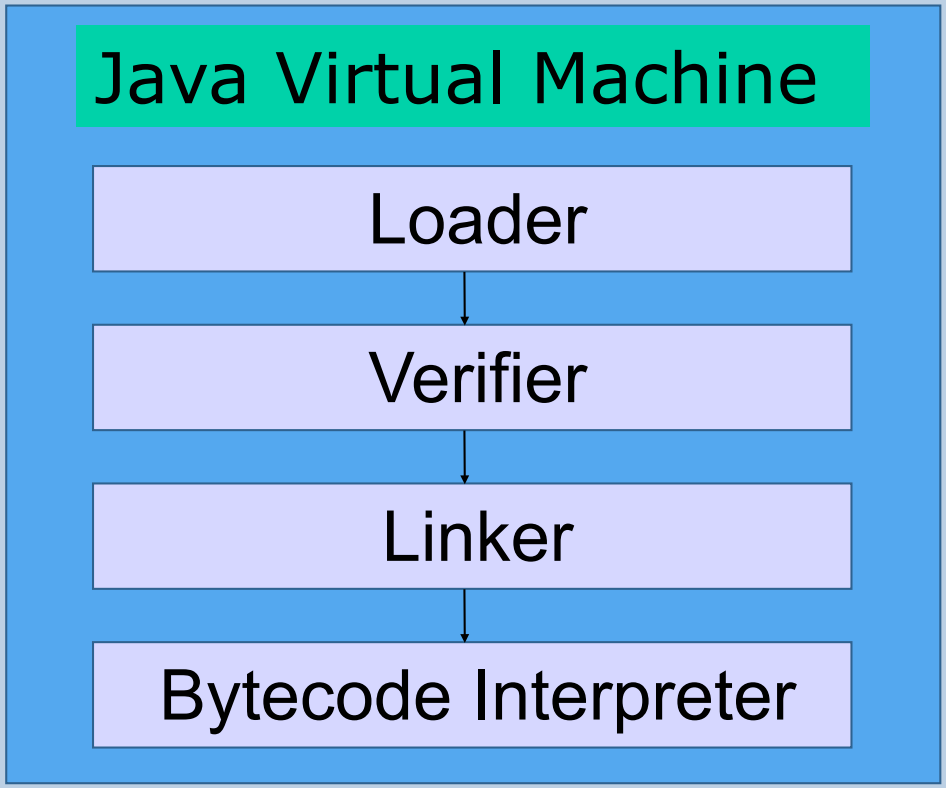
The JVM code generated for an expression

def x1 = E1 ... xn = En **in** E **end**

1. creates and pushes a new stack frame (heap allocated) to store the value of the n identifiers x1 ... xn
2. initialises the slot for each xi with the value of Ei
3. generates code for E
3. pops off of the frame

The generated code for an id does not need to dynamically search up in the stack, it knows its coordinates (from D(id))

JVM Architecture



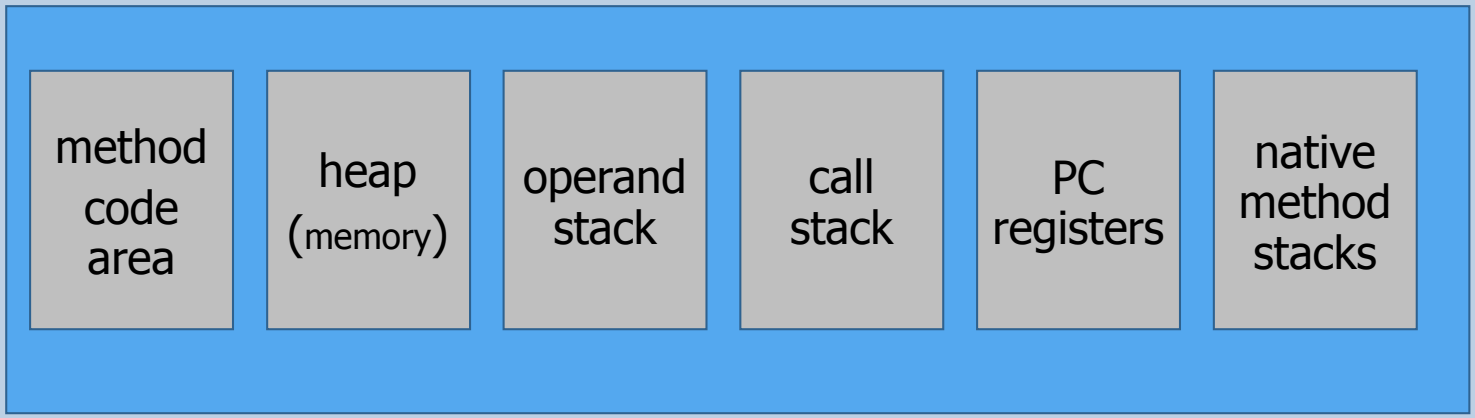
Operand Stack:

stores arguments and results of instructions

Heap:

stores dynamically allocated objects, arrays, strings, classes

….



Structure of Stack Frames (jasmin syntax)

```
class frame_id
.super java/lang/Object
.field public sl Lancestor_frame_id;
.field public s_1 I
.field public s_2 I
..
.field public s_n I
.end method
```

frame_id : unique type name generated by compiler

sl : holds reference to ancestor frame

ancestor_frame_id : name of the ancestor frame type

s_i : slot #i (stores value of id #i in this level)

Note: a stack frame is represented a JVM class with no methods (just public fields). This is akin to a C struct.

Structure of Stack Frames (jasmin syntax)

```
class frame_id
.super java/lang/Object
.field public sl Lancestor_frame_id;
.field public s_1 I
.field public s_2 I
..
.field public s_n I
.end method
```

frame_id : unique type name generated by compiler

sl : (reference also called the "static link")

ancestor_frame_id : name of the ancestor frame type

s_i: slot #i (stores value of id #i in this level)

Note: a stack frame is represented a JVM class with no methods (just public fields). This is akin to a C struct.

Compilation of declarations

[[**def** x1 = E1 … xn = En **in** E **end**]]D =

frame creation and linkage into environment stack (push)

```
new frame_id
dup
invokespecial frame_id/<init>()V
dup
aload SL
putfield frame_id/sl Lcurrframetype
astore SL
```

JVM instructions

new

Operation

Create new object

Format

```
new
indexbyte1
indexbyte2
```

Forms

new = 187 (0xbb)

Operand Stack

... →

..., *objectref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in a class type. Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (§2.3, §2.4). The *objectref*, a `reference` to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

JVM instructions

dup

Operation

Duplicate the top operand stack value

Format

dup

Forms

dup = 89 (0x59)

Operand Stack

..., *value* →

..., *value*, *value*

Description

Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

The *dup* instruction must not be used unless *value* is a value of a category 1 computational type (§2.11.1).

JVM instructions

aload

Operation

Load `reference` from local variable

Format

```
aload
index
```

Forms

aload = 25 (0x19)

Operand Stack

... →

..., *objectref*

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes

The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction (§*astore*) is intentional.

The *aload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

JVM instructions

astore

Operation

Store `reference` into local variable

Format

```
astore
index
```

Forms

astore = 58 (0x3a)

Operand Stack

..., *objectref* →

...

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

Notes

The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clause of the Java programming language (§3.13).

The *aload* instruction (§*aload*) cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *astore* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

JVM instructions

putfield

Operation

Set field in object

Format

```
putfield
indexbyte1
indexbyte2
```

Forms

putfield = 181 (0xb5)

Operand Stack

..., *objectref*, *value* →

...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The class of *objectref* must not be an array. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

The referenced field is resolved (§5.4.3.2). The type of a *value* stored by a *putfield* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class, and the instruction must occur in an instance initialization method (`<init>`) of the current class (§2.9).

The *value* and *objectref* are popped from the operand stack. The *objectref* must be of type `reference`. The *value* undergoes value set conversion (§2.8.3), resulting in *value*', and the referenced field in *objectref* is set to *value*'.

Compilation of declarations

[[**def** x1 = E1 ... xn = En **in** E **end**]]D =

initialization of identifier slots in frame

```
aload SL
[[E1]]D+{ x1|->(s_1,s'), xn|->(s_n,s') }      // s' = |D|+1
putfield frame_id/s_1 I
aload SL
[[E2]]D+{ x1|->(s_1,s'), xn|->(s_n,s') }
putfield frame_id/s_2 I
....
aload SL
[[En]]D+{ x1|->(s_1,s'), xn|->(s_n,s') }
putfield frame_id/s_n I
```

Compilation of declarations

[[**def** x1 = E1 ... xn = En **in** E **end**]]D =

code for definition body

[[E]]D+{ x1|->(s_1,s'), xn|->(s_n,s') }

Compilation of declarations

[[**def** x1 = E1 … xn = En **in** E **end**]]D =

frame pop off and update of local SL

```
aload SL
getfield frame_id/sl Lcurrframetype
astore SL
```

JVM instructions

getfield

Operation

Fetch field from object

Format

```
getfield
indexbyte1
indexbyte2
```

Forms

getfield = 180 (0xb4)

Operand Stack

..., *objectref* →

..., *value*

Description

The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2). The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.

The type of *objectref* must not be an array type. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Compilation of identifiers (bound uses)

[[x]]D =

Assume D(x) = (d, s)

```
aload SL
getfield frame_id/sl Lancestor_frame_id
...
getfield frame_id/sl Lancestor_frame_id
getfield frame_id/s I
```

} N-d stack up dereferences

the number k of dereference (using getfield sl) is N-d

the coordinates stored in the compilation environment are used to generate code that fetches the identifier value from the appropriate frame at runtime.

Compilation of identifiers (bound uses)

[[x]]D =

Assume D(x) = (**d**, **s**)

```
aload SL
getfield frame_id/sl Lancestor_frame_id
…
getfield frame_id/sl Lancestor_frame_id
getfield frame_id/s I
```

} N-d stack up dereferences

D(x) = (**d**, **s**) where

d: the depth of the topmost stack frame (from bottom of stack) where the identifier is declared.

s: the slot in the frame where the associated value is stored

Compilation of CALCI (example code)

```
.class public frame_0
.super java/lang/Object
.field public sl Ljava/lang/Object;
.field public v0 I
.field public v1 I

.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return


.end method
```

default constructor (JVM requires it)

```
.class public frame_1
.super java/lang/Object
.field public sl Lframe_0;
.field public v0 I

.end method
```

```
def
  x  = 2
  y  = 3
in
 def
     k = x + y
 in
     x + y + k
 end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
 def
   k = x + y
 in
      x + y + k
 end
end;;
```

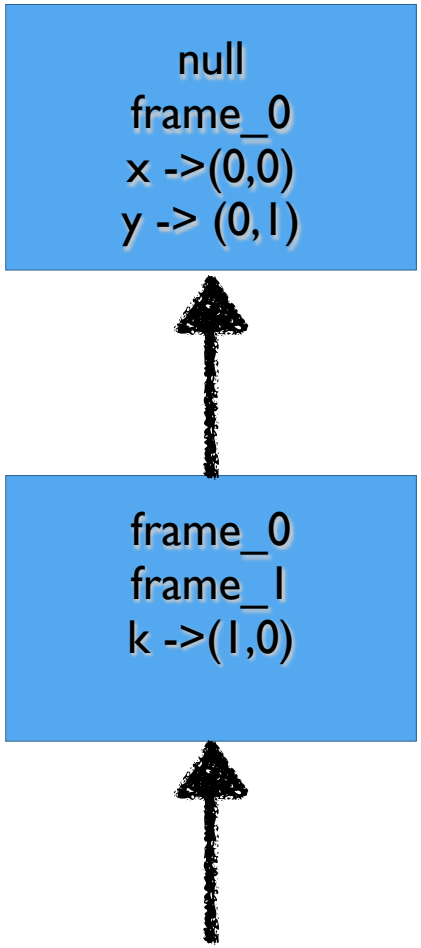
Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation Environment



D
(compilation
environment)

```
def
  x  (0,0) = 2
  y  (0,1) = 3
in
 def
   k (1,0)  = x (1,0) + y (1,1)
 in
      x (1,0) + y (1,1) + k (0,0)
 end
end;;
```

Compilation of identifiers (bound uses)

[[x]]D =

Assume D(x) = (d, s)

```
aload SL
getfield frame_id/sl Lancestor_frame_id
…
getfield frame_id/sl Lancestor_frame_id
getfield frame_id/s I
```

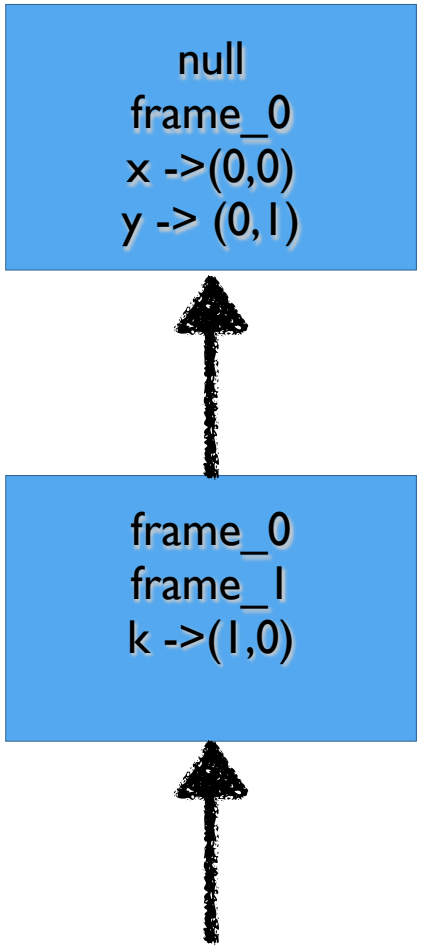
} N-d stack up dereferences

D(x) = (d, s) where

d: the depth of the topmost stack frame (from bottom of stack) where the identifier is declared.

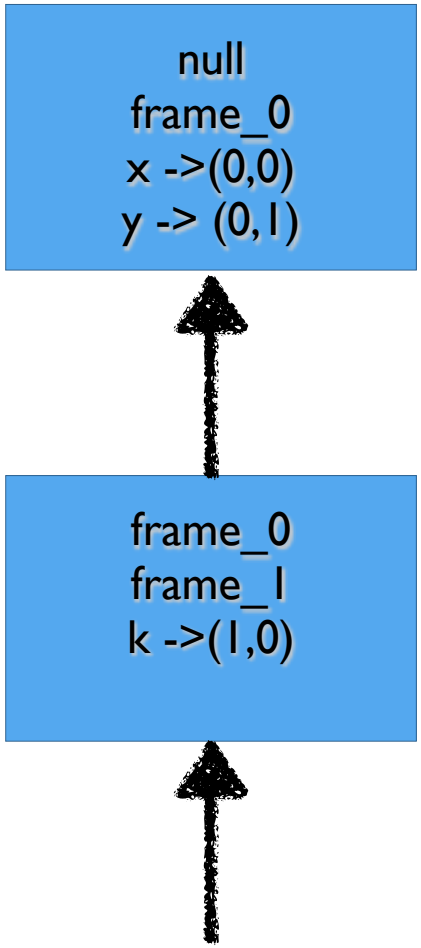
s: the slot in the frame where the associated value is stored

Compilation Environment



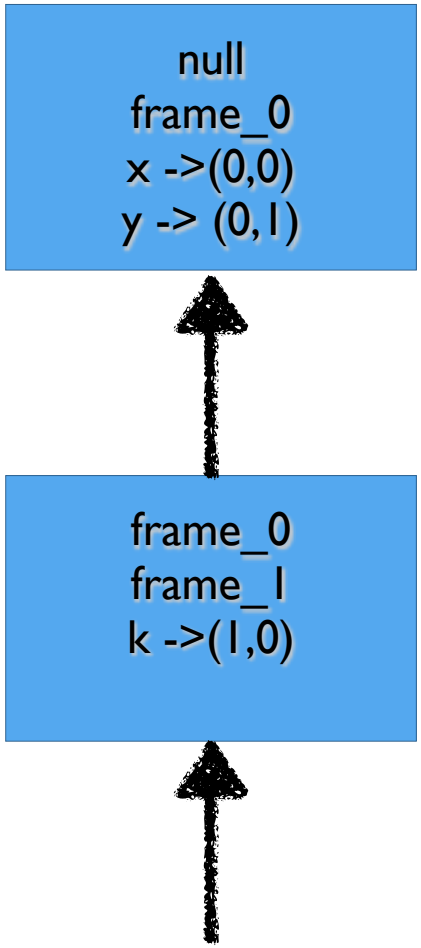

```
def (N=0)
  x  (0,0) = 2
  y  (0,1) = 3
in
 def  (N=1)
   k (1,0)  = x (N-d,0) + y (N-d,1)
 in
      x (N-d,0) + y (N-d,1) + k (N-d,0)
 end
end;;
```

Compilation Environment




```
def (N=0)
  x  (0,0) = 2
  y  (0,1) = 3
in
 def (N=1)
   k (1,0)  = x (1-0,0) + y (1-0,1)
 in
      x (1-0,0) + y (1-0,1) + k (1-1,0)
 end
end;;
```

Compilation Environment




```
def (N=0)
  x  (0,0) = 2
  y  (0,1) = 3
in
 def (N=1)
   k (1,0)  = x (1,0) + y (1,1)
 in
      x (1,0) + y (1,1) + k (0,0)
 end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Compilation of CALCI (example code)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
```

```
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def
    k = x + y
  in
      x + y + k
  end
end;;
```

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Imperative Languages

Till now, expressions of our language have denoted pure values. and every expression always denotes an immutable fixed value in a given scope.

Imperative languages (C, Java, …) introduce mutable values (memory cells) and fundamental imperative operations:

- Allocation of memory (var x:Integer; int x;)
- Operations to read / write memory (e.g., x := 2, y = y + 2)

- Memory Model (new, set, get, free)
- Environment versus memory
- Aliasing
- L-value e R-value
- lifetime versus scope
- References, pointers, etc
- Intepreter for imperative language

Basic Memory Model

- Memory: dynamic store of memory cells, each with a mutable content.
- Each memory cell has a unique designator (the cell reference or address)
- We assume general cells, that can store any value of the language.
- The memory contais a pool of unused cells (the free pool), the other cells are considered in use by the executing program.
- A cell reference is also a value of a special (opaque) data type ref
- Interface for memory $\mathcal{M}$

new:	$\mathcal{M}$ × Value → **ref**
set:	$\mathcal{M}$ × **ref** × Value → **void**
get:	$\mathcal{M}$ × **ref** → Value
free:	$\mathcal{M}$ × **ref** → **void**

Basic Memory Model

- Operations for a memory $\mathcal{M}$, functionally defined

new: $\mathcal{M} \times \text{Value} \rightarrow \mathcal{M} \times \textbf{ref}$

Guess back a reference for a newly allocated from the free pool.

set: $\mathcal{M} \times \textbf{ref} \times \text{Value} \rightarrow \mathcal{M}$

Mutates (changes) the value stored in the cell ref . The previous value is lost.

get: $\mathcal{M} \times \textbf{ref} \rightarrow \mathcal{M} \times \text{Value}$

Returns the value stores in the cell ref.

free: $\mathcal{M} \times \textbf{ref} \rightarrow \mathcal{M}$

releases the cell ref to the free pool.

Environment versus Memory

- The environment gives the value associated to every identifier declared in the program and reflects the static structure of such program (nesting of scopes).
- In the environment the binding between an identifier and its value is fixed and immutable. The value is bound just once using the assoc() operation.

- The memory contains a set of mutable cells, each cell is named by a reference value and holds a value.
- The value stored in a reference may be changed during execution, using assignment operations (e,.g., X := E),
- We may refer to a reference using an identifier (usually called a "state variable"), The binding between the name of a "state variable" and its associated memory location is defined by the ambiente. This binding is immutable in its scope.

Environment versus Memory

Environment

Identifier	Value
PI	3,14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memory

Reference	Stored Value
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Environment versus Memory

Environment

Identifier	Value
PI	3,14
x	0x00FF
k	0x0100
j	0x0100
TEN	10

Memory

Reference	Stored Value
0x00FF	25
0x0100	12
0x0102	0x0100
...	...
0xFFFF	0

Memory Model (properties)

Identifier	Value
PI	3,14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Reference	Stored Value
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

The same memory cell may be bound to different names (aliasing).

Aliasing

- Different names / expressions may refer to the same memory cell.

```
class A {
  int x;
  boolean equals(A b) { return x == b.x}
}
A a = new A(); a.equals(a);
```

```
int x = 0;
void f(int* y) { *y = x+1; }
...
f(&x);
// x = ?
```

Memory Model (properties)

Identifier	Value
PI	3,14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Reference	Stored Value
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

References are values (first-class references)

A cell may store a reference to other cell, allowing the manipulation of dynar
data structures, and even cyclic data structures.

Language level imperative primitives

- Allocates a new cell, initialises it with value of expression E and returns the reference

ref(E)

- We may this kind of operation more or less explicit in all imperative programming languages:

```
{
  int a = 2;
  MyClass m;
  …
}

new int[10];

malloc(sizeof(int));

new MyClass();
```

Language level imperative primitives

- Assignment of a value to a reference cell

E := F

Expression E denotes a reference cell, expression F denotes some value

- Assignments are present in programming languages in many forms:

```
a = a + 1

i := 2

b[x+2][b[x-2]] = 2

*(p+2) = y

myTable(i,j) = myTable(j,i)

Readln(MyLine);
```

Language level imperative primitives

- Dereference of the memory cell denoted by expression E.

!E

- We may find the presence of dereference in many forms:

```
i := !i + 1                    (OCAML)

*p                             (C)

i = i + 1                      (Java)

i++                            (Java)
```

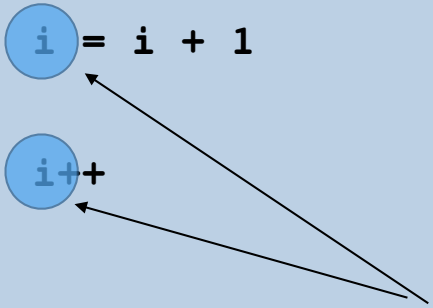
Language level imperative primitives

- Dereference of the memory cell denoted by expression E.

!E

- We may find the presence of dereference in many forms:

```
i := !i + 1          (OCAML)

*p                   (C)

i = i + 1            (Java)

i++                  (Java)
```



references

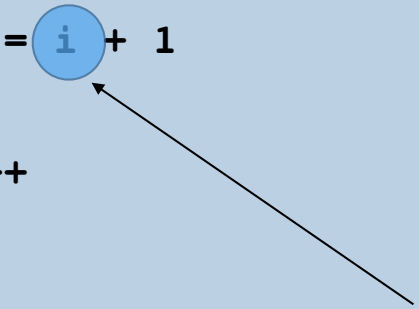
Language level imperative primitives

- Dereference of the memory cell denoted by expression E.

!E

- We may find the presence of dereference in many forms:

```
i := !i + 1          (OCAML)

*p                   (C)

i = i + 1            (Java)

i++                  (Java)
```



denotes contents of i

(implicit dereference) L-Value e R-Value

- If expression E denotes a reference, most programming languages interprets E in a context dependent way

```
E := 2
```

- (Left-Value) To the left of the assignment symbol, denotes its true value (a reference)

```
E := E + 1
```

- (Right-Value) To the left of the assignment symbol, implicitly denotes the contents of the reference cell, without explicit dereference

```
E := !E + 1
```

(desreferenciação implícita) L-Value e R-Value

- If expression E denotes a reference, most programming languages interprets E in a context dependent way,
- NB. The terminology "L-Value" e "R-Value" although standard (you should know it) is not very sound.
- For example, consider, e.g.,

```
A[A[2]] := A[2] + 1
```

- In this statement, both subexpressions **A[2]**, one to the left and other to the rught are dereferenced implicitly, However A[A[2]] to the left is not dereferenced (so that it denotes a reference)

```
A[!A[2]] := !A[2] + 1
```

Explicit deference

- The explicit dereference operation **!E** allows the semantics of programs to be more precise, context free, and escapes any ambiguity.

```
A[!A[2]] := !A[2] + 1
```

- On the other hand, the use of dereference **!-** make programs more verbose. Most programming languages adopt implicit deference, for historical reasons.
- NB. The implicit dereference may be better understood as a coercion(cast) operation in which the interpreter compiler inserts the missing **!**
- To do this, types of expression must be known at evaluation / compilation time.
- In our language we will adopt uniform explicit dereference.

Language level imperative primitives

- All patterns of use mutable state in programming languages can be expressed using the basic imperative primitives

new (E)	allocation
free(E)	release
E := E	assignement
! E	dereference

```
{
/* C language */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b + k
  …
}
```

```
def
  k = 2
  a = new(k)
  b = new(!a+2)
in
  …
  b := !a * !b + k
  …

end
```

Language level imperative primitives

- All patterns of use mutable state in programming languages can be expressed using the basic imperative primitives

new(E)	allocation
free(E)	release
E := E	assignement
! E	dereference

```
{
/* C language */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b
  …
}
```

```
def
  k = 2
  a = new(k)
  b = new(!a+2)
in
  …
  b := !a * !b
  …
end
```

implicit release (of cells denoted by a e b)

Language level imperative primitives

- All patterns of use mutable state in programming languages can be expressed using the basic imperative primitives

new (E)	allocation
free(E)	release
E := E	assignement
! E	dereference

```
{
/* C++ */

  int k = 2;
  const int *a = &k;
  int b = *a;
  … *a = k+b …

}
```

```
def
  k = var(2)
  a = k
  b = var(!a)
in
  … a := !k+!b …


end
```

Language level imperative primitives

- All patterns of use mutable state in programming languages can be expressed using the basic imperative primitives

new (E)	allocation
free(E)	release
E := E	assignement
! E	dereference

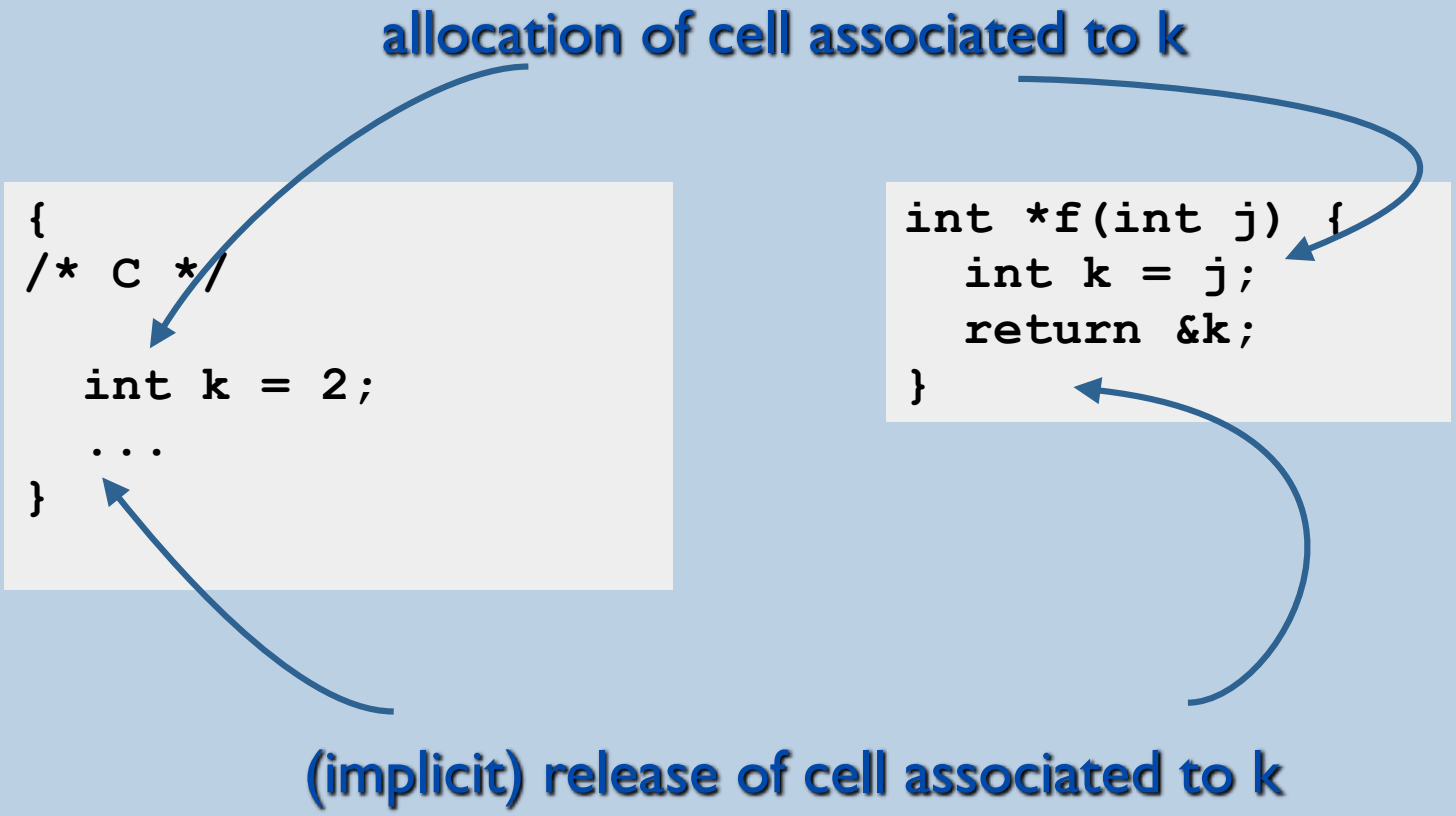
```
{
/* C */

  int k = 2;
  int *a = &k;
  … k = k+*a …

}
```

```
def
  k = var(2)
  a = var(k)
in
  … k := !k+!!a …


end
```

Lifetime vs Scope

The lifetime of a memory cell is the time (during program execution) that intermediates bewteen its allocation new(_) and its release free(_).

- Sometimes, a memory cell lifetime concides with the execution of the scope of declaration of its associated identifier.



Lifetime vs Scope

The lifetime of a memory cell is the time (during program execution) that intermediates bewteen its allocation new(_) and its release free(_).

- Most often, a cell lifetime leaks out of the scope in which it is created.

scope of k

```
{
/* linguagem C */

  static int k;
  ...
}
```

In this example, the lifetime of k is the whole program execution

allocation of cell associated to k

```
/* linguagem Java */
Integer f(int j) {
  Integer k = new Integer(j);
  g(k);
  return k;
}
```

allocation of Integer object

Here the cell associated to k is implicitly released at the end of scope but the Integer object may survive

The language CALCS (abstract syntax)

num: Integer → CALCS

bool: Integer → CALCS

id: String → CALCS

add: CALCS × CALCS → CALCS

gt: CALCS × CALCS → CALCS

def: (String × CALCS)+ × CALCS → CALCS

seq: CALCS x CALCS → CALCS

if: CALCS × CALCS × CALCS → CALCS

while: CALCS × CALCS → CALCS

new: CALCS → CALCS

deref: CALCS → CALCS

assign: CALCS × CALCS → CALCS

println: CALCS → CALCS

The language CALCS (concrete syntax)

```
def
    N = new(676)
in
    while (!N ~= 1) do
        if (2*(!N/2) = !N) then
            N := !N/2
        else
            N := 3*!N + 1
        end;
        println !N
    end;
    println “HELLO”
end
```

The language CALCS (concrete syntax)

```
def T = 10 in
def a = new(0) in
  while (!a < T) do
     a := !a + 1;
  end
end
```

```
def a = new(2) in
  def b = new(!a) in
    def c = a in
      a := !b + 2;
      c := !c + 2
    end
  end
end
```

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: \text{AST} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

AST = open programs

ENV = Environments (funções ID → VAL)

MEM = Memories

VAL = Values

$$\text{Val} = \text{Boolean} \cup \text{Integer} \cup \text{Ref}$$

The evaluation map expresses that in general a CALCS program P produces a resulting value and performs a (side) effect (in memory).

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: \text{AST} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

eval(**add**(E1, E2) , *env* , *m0*) ≜**[** (*v1* , *m1*) = **eval**(E1, *env*, *m0*);
(*v2* , *m2*) = **eval**(E2, *env*, *m1*);
(*v1* + *v2* , *m2*) **]**

eval(**and**(E1, E2) , *env* , *m0*) ≜**[** (*v1* , *m1*) = **eval**(E1, *env*, *m0*);
(*v2* , *m2*) = **eval**(E2, *env*, *m1*);
(*v1* && *v2* , *m2*) **]**

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: \text{AST} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

```
eval( new(E) , env , m0) ≜ [ (v1 , m1) = eval( E, env, m0 );
                             (m1.new(v1), m1);
                           ]
eval( !(E) , env , m0) ≜ [ (ref , m1) = eval( E, env, m0);
                             (m1.get(ref) , m1) ]
eval(E1 := E2 , env , m0) ≜ [(v1 , m1) = eval( E1, env, m0);
                               (v2 , m2) = eval( E2, env, m1);
                               m3 = m2.set(v1, v2) ;
                               (v2 , m3) ]
```

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: AST \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( seq(E1, E2) , env , m0) ≜ [(v1 , m1) = eval( E1, env, m0);
                                  eval( E2, env, m1)
                                ]
eval( if(E1, E2, E3) , env , m0) ≜
                    [  (v1 , m1) = eval( E1, env, m0);
                       if (v1 = true) then eval( E2, env, m1);
                                      else eval( E3, env, m1);
                    ]
```

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: AST \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( while(E1, E2) , env , m0) ≜
                                  [(v1 , m1) = eval( E1, env, m0 );
                                   if (v1 = T) then  [ (v2 , m2) = eval( E2, env, m1);
                                                       (v,m1) =eval( while(E1, E2), m2) ]
                                                     else (F , m1) ]
```

NB. Here we interpret iteration (while) in terms of recursion.

Sematics of CALCS (schematic)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: \text{AST} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

```
eval( def(s, EI, EB) , env , m0) ≜
                    [(v1 , m1) = eval( EI, env, m0 );
                     env = env.BeginScope();
                     env.Assoc(s, v1);
                     (v2 , m2) = eval(EB, env, m1);
                     env = env.EndScope();
                     (v2 , m2) ]
```

NB. The "functional" semantics explicitly specifies an order of evaluation, by threading memory use.

Interpreter for CALCS (Java implementation)

NB. In the object oriented implementation, we use global "reference" objects.

```
class VCell implements IValue
{  IValue v;
   VCell(IValue v0) { v = v0; }
   IValue get() { return v;}
   void set(IValue v0) { v = v0;}
}
```

Interpreter for CALCS (Java implementation)

```
class ASTNew implements ASTNode {

Value eval(Env<IValue> e)

   {  ASTNode exp;
      Value v1 = exp(e);

      return new VCell(v1);

   }
}
```

Interpreter for CALCS (Java implementation)

```
class ASTAssign implements ASTNode {
Value eval(Env<IValue> e)
   {  ASTNode lhs;
      ASTNode rhs;
      Value v1 = lhs.eval(e);
      if (v1 instanceof VCell) {
            Value v2= rhs.eval(e);
            ((VCell)v1).set(v2);
           return v2;
       }
     throw new RuntimeError("Assign: reference expected");
   }
}
```

Compilation Schemes (CALCS)

Compilation Schemes

We will use compilation schemes to define our compiler

A compilation scheme defines the sequence of instructions generated for a programming language construct

$$[\![E]\!]D = \ldots \text{ code sequence } \ldots.$$

- E : program fragment
- D: environment (associates identifiers to "coordinates")

$[\![E_1+E_2]\!]D =$
$[\![E_1]\!]D$
$[\![E_2]\!]D$
iadd

Compilation Schemes (rel ops)

in the JVM, we represent booleans by integers
(0 - false, 1 - true)

```
[[ E1 > E2 ]]D =
[[E1]]D
[[E2]]D
isub
ifgt L1
sipush 0
goto L2
L1: sipush 1
L2:
```

Compilation Schemes (bool ops)

[[E1 && E2]]D =

[[E1]]D

[[E2]]D

iand

[[E1 || E2]]D =

[[E1]]D

[[E2]]D

ior

Compilation Schemes (conditionals)

[[**if** E1 **then** E2 **else** E3]]D =

[[E1]]D

ifeq L1

[[E2]]D

goto L2

L1: [[E3]]D

L2:

Compilation Schemes (while loop)

[[**while** E1 **do** E2 **end**]]D =

L1: [[E1]]D

ifeq L2

[[E2]]D

pop

goto L1

L2:

Short Circuit Evaluation of Conditionals

Here we explain an alternative, more efficient, compilation scheme for conditionals, in which the value of boolean expressions is not explicitly computed, but directly results in a control flow choice between two jump labels

Short Circuit Evaluation of Conditionals

The code [[BE]] generated by a boolean typed expression BE will leave a boolean value on top of the stack

If BE appears inside a conditional expression, such as an if-then-else or while expression, such boolean value will be immediately consumed by a ifxx branch instruction

Short circuit evaluation of conditionals "skips over" the intermediate boolean value, and compiles a boolean typed expression BE relative to two given labels

- TL (true label)
- FL (false label)

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

Let's see the definition of [[BE (TL, FL)]] for the various boolean valued expressions BE

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

[[true (**TL, FL**)]]D =

goto TL

[[false (**TL, FL**)]]D =

goto FL

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

[[~ E2, **TL**, **FL**]]D =

[[E1, FL, TL]]D

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ E1 && E2, TL, FL]]D =
[[E1, AuxLabel, FL]]D
AuxLabel:
[[E2, TL, FL]]D
```

```
[[ E1 || E2, TL, FL]]D =
[[E1, TL, AuxLabel]]D
AuxLabel:
[[E2, TL, FL]]D
```

Short Circuit Evaluation of Conditionals

Notice:

- in E1 && E2, code E2 is executed only if E1 yields true
- in E1 || E2, code E2 is executed only if E1 yields false

```
[[ E1 && E2, TL, FL]]D =
[[E1, AuxLabel, FL]]D
AuxLabel:
[[E2, TL, FL]]D
```

```
[[ E1 || E2, TL, FL]]D =
[[E1, TL, AuxLabel]]D
AuxLabel:
[[E2, TL, FL]]D
```

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ E1 > E2, TL, FL]]D =
[[E1]]D
[[E2]]D
isub
ifgt TL
goto FL
```

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ while E1 do E2 end]]D =
LStart:
[[ E1 (TL, FL) ]]D
TL:
[[ E2 ]]
pop
goto LStart
FL:
```

Short Circuit Evaluation of Conditionals

Code [[BE (TL, FL)]] does not change the stack, but will:

- jump to TL if the value of BE is true
- jump to FL if the value of BE is false

```
[[ if E1 then E2 else E3 end]]D =
[[E1 ( TL, FL) ]]D
TL: [[E2]]
goto LExit
FL: [[E3]]
LExit:
```

Compilation Schemes (reference cells)

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ is the JVM type corresponding to the cell content type, e.g.

int maps to typeJ = I

bool maps to typeJ = Z

ref int maps to typeJ = Lref_of_int;

ref ref bool maps to typeJ = Lref_of_ref_int;

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_int
.super java/lang/Object
.field public v I
.end method
```

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_bool
.super java/lang/Object
.field public v Z
.end method
```

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_ref_of_int
.super java/lang/Object
.field public v Lref_of_int;
.end method
```

Compilation Schemes (reference cells)

At runtime, we will represent a reference cell as a JVM class:

```
.class ref_of_type
.super java/lang/Object
.field public v typeJ
.end method
```

typeJ will be the JVM type corresponding to the cell content type, e.g.

```
.class ref_of_ref_of_ref_of_bool
.super java/lang/Object
.field public v Lref_of_ref_of_bool;
.end method
```

Compilation Schemes (new)

typeJ is the JVM type corresponding to the cell content type, e.g.

```
E[[ new E ]]D =
new ref_of_type
dup
invokespecial ref_of_type/<init>()V
dup
[[E]]D
putfield ref_of_type/v typeJ
```

Compilation Schemes (dereference)

typeJ is the JVM type corresponding to the cell content type, e.g.

```
E[[ ! E ]]D =
[[E]]D
getfield ref_type/v typeJ
```

Compilation Schemes (assign)

typeJ is the JVM type corresponding to the cell content type, e.g.

```
E[[ E1 := E2 ]]D =
[[E1]]D
[[E2]]D
putfield ref_type/v typeJ
```

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Interpretação e Compilação de Linguagens (de Programação)

22/23

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Type Systems

A type system is a kind of "program logic" that ensures "good behaviour" of programs.

A type systems is a form of so-called "static analysis" or "abstract interpretation" of programs. For any syntactically correct program a static analysis always terminates, even for programs that do not terminate at runtime.

A very common static analysys technique is type checking.

A type system statically ensures (e.g. at compile time) the absence of certain kinds of runtime errors and also provides useful information for compilation.

- Runtime Errors
- Abstract Interpretation
- Type Systems and Type Checking
- Soundness of a Type Systems
- Trapping of Runtime Errors

Runtime errors …

- What may go wrong during a program execution ?

```
def
  a = new 0
  b = new 2
  c = new (!a > !b)
in
  if !c then
     a := a + 1
  else c := !b < !c
end
```

Runtime errors …

- What may go wrong during a program execution ?

```
def
  a = new 0
  b = new 2
  c = new (!a > !b)
in
  if c then
     a := !a + 1;
     c := !b < !a
end
```

Runtime errors …

- What may go wrong during a program execution ?

eval(**add**(E1, E2) , *env* , *m0*) ≜ **[** (*v1* , *m1*) = **eval**(E1, *env*, *m0*);

(*v2* , *m2*) = **eval**(E2, *env*, *m1*);

(*v1* + *v2* , *m2*) **]**

eval(**deref**(E) , env , m0) ≜ **[** (ref , m1) = **eval**(E, env, m0);

(m1.**get**(ref) , m1) **]**

eval(**assign**(E1, E2) , env , m0) ≜ **[**(v1 , m1) = **eval**(E1, env, m0);

(v2 , m2) = **eval**(E2, env, m1);

m3 = m2.**set**(v1, v2) ;

(v1 , m3) **]**

It is not feasible during compilation to compute concrete values, and check if all operations will not go wrong…
The set of possible values is infinite

However, we may approximate such sets of values by classifying them into "types".

Semantic maps (we have seen before)

- Map **eval** to compute a value + effect for any CALCS programs:

$$\text{eval} : \text{CALCS} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **comp** that translates CALCS programs in JVM bytecode

$$\text{comp} : \text{CALCS} \times \text{ENV} \rightarrow \text{CodeSeq}$$

Typechecker (as an abstract interpreter)

- Map **eval** to compute a value + effect for any CALCS programs:

$$\text{eval} : \text{CALCS} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **comp** that translates CALCS programs in JVM bytecode

$$\text{comp} : \text{CALCS} \times \text{ENV} \rightarrow \text{CodeSeq}$$

- Map **typchk** that computes a type for a CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\} \}$$

Typechecker (as an abstract interpreter)

- Map **eval** to compute a value + effect for any CALCS programs:

$$\text{eval} : \text{CALCS} \times \text{ENV<IValue>} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **comp** that translates CALCS programs in JVM bytecode

$$\text{comp} : \text{CALCS} \times \text{ENV<Coord>} \rightarrow \text{CodeSeq}$$

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV<TYPE>} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$

Types for CALCS

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV<TYPE>} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\} \}$$

- Map **typchk** may be understood as an "abstract" interpreter that evaluates the program according to a special semantics, more abstract, in which values are approximated by types.
- In the abstract sematics, the values are the types and operations compute types from types.

Types for CALCS

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV<TYPE>} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\} \}$$

int: type of integer values.

bool: type of boolean values.

ref{$\mathcal{T}$}: type of references that may only hold values of type $\mathcal{T}$.

Example: ref{ref{int}} is the type of references that may only hold references that may only hold integers.

Types for CALCS (defined by a grammar)

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV<TYPE>} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\} \}$$

int: type of integer values.

bool: type of boolean values.

ref{$\mathcal{T}$}: type of references that may only hold values of type $\mathcal{T}$.

Type → int | bool | ref [Type]

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

typchk(**add**(E1, E2) , *env*) ≜ [*t1* = **typchk** (E1, *env*)

t2 = **typchk** (E2, *env*)

if (*t1* == **int**) **and** (*t2* == **int**)

then int

else TYPEERROR]

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

typchk(**num**(*n*) , *env*)	≜ **int** ;
typchk(**id**(s) , *env*)	≜ *env*.Find(s) ;
typchk(**true** , *env*)	≜ **bool** ;
typchk(**false** , *env*)	≜ **bool** ;

  - All integers num(n) are "evaluated " into int.
  - All booleans are "evaluated " into bool.

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

$$typechk : CALCS \times ENV\langle TYPE\rangle \rightarrow TYPE \cup \{ TYPEERROR \}$$

$$ENV : ID \rightarrow TYPE$$

```
typchk( def(s, E1, E2), env) ≙ [ ts = typchk(E1, env);
                                 env = env.BeginScope();
                                 env =  env.Assoc(s, ts);
                                 t = typchk(E2, env);
                                 env = env.EndScope();
                                 return t ]
```

– Associate each names **s** to a type **ts** of initialiser **E2**

– return the type of the body **E2**.

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

typchk(**and**(E1, E2) , *env*) ≜ [*t1* = **typchk** (E1, *env*)

t2 = **typchk** (E2, *env*)

if (*t1* == **bool**) **and** (*t2* == **bool**)

then bool

else TYPEERROR]

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( if(E0, E1, E2) , env ) ≜ [ t0 =  typchk ( E0, env)
        if  (t0 != bool ) then TYPEERROR
       else [  t1 =  typchk ( E1, env)
               t2 =  typchk ( E2, env)
               if  (t1 == t2 ) then t2  else TYPEERROR
      ] ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

typchk(**new** (E) , *env*) ≜

[*t* = **typchk**(E, *env*);

if (*t* == **TYPEERROR**)

then t

else **ref (*t*)**]

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

ENV : ID → TYPE

```
typchk( deref(E) , env ) ≜
                                    [ t = typchk( E, env );
                                      if (t == ref (tr )
                                              then  tr
                                              else TYPEERROR  ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

ENV : ID → TYPE

```
typchk( assign(E1, E2) , env ) ≜
                                  [ t1 = typchk( E1, env );
                                    t2 = typchk( E2, env );
                                    if (t1 == ref ( t2 ))
                                            then t2
                                            else TYPEERROR ; ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV<TYPE>} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

```
typchk( seq(E1, E2) , env ) ≜
                    [ t1 = typchk( E1, env );
                      if ( t1 == TYPEERROR ) then TYPEERROR
                      else [ t2 = typchk( E2, env );
                             return t2 ] ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

$$typechk : CALCS \times ENV\langle TYPE\rangle \rightarrow TYPE \cup \{ TYPEERROR \}$$

$$ENV : ID \rightarrow TYPE$$

```
typchk( if(E1, E2, E3) , env ) ≜
                    [ t1 = typchk( E1, env );
                      if ( t1 != bool ) then TYPEERROR
                      else [ t2 = typchk( E2, env );
                             t3 = typchk( E3, env );
                             if (t2 != t3)
                                or (t2 == TYPEERROR) or (t3 == TYPEERROR)
                             then TYPEERROR
                             else t2 ] ]
```

Sematics of CALCS (recall)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: AST \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( seq(E1, E2) , env , m0) ≜ [(v1 , m1) = eval( E1, env, m0);
                                  eval( E2, env, m1)
                                ]
eval( if(E1, E2, E3) , env , m0) ≜
                    [  (v1 , m1) = eval( E1, env, m0);
                       if (v1 = true) then eval( E2, env, m1);
                                      else eval( E3, env, m1);
                    ]
```

CALCS Typing Rules

- Let's compare this with the case for **if** in Map **eval**.

  Here, **both** branches E2 e E3 are analysed;
  and we impose (as a conditions) *t2* == *t3* . [Why?]

```
typchk( if(E1, E2, E3) , env ) ≜
                    [ t1 = typchk( E1, env );
                      if ( t1 != bool ) then TYPEERROR
                      else [ t2 = typchk( E2, env );
                             t3 = typchk( E3, env );
                             if (t2 != t3)
                                or (t2 == TYPEERROR) or (t3 == TYPEERROR)
                             then TYPEERROR
                             else t2 ] ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

$$typechk : CALCS \times ENV\langle TYPE\rangle \rightarrow TYPE \cup \{ TYPEERROR \}$$

$$ENV : ID \rightarrow TYPE$$

```
typchk( while(E1, E2) , env ) ≜
                    [ t1 = typchk( E1, env );
                      if ( t1 != bool) then TYPEERROR
                      else [ t2 = typchk( E2, env );
                             if ( t2 == TYPEERROR) TYPEERROR
                             else bool
                           ]
                     ]
```

Sematics of CALCS (recall)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$eval: AST \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( while(E1, E2) , env , m0) ≜
                                  [(v1 , m1) = eval( E1, env, m0 );
                                   if (v1 = T) then  [ (v2 , m2) = eval( E2, env, m1);
                                                       (v,m1) =eval( while(E1, E2), m2) ]
                                                     else (F , m1) ]
```

CALCS Typing Rules

- Let's compare this with the case for **while** in Map **eval**.

  Here, **body** E2 is analysed exactly once (no iteration); and the returned type is bool . [Why?]

```
typchk( while(E1, E2) , env ) ≜
                    [ t1 = typchk( E1, env );
                      if ( t1 != bool) then TYPEERROR
                      else [ t2 = typchk( E2, env );
                             if ( t2 == TYPEERROR) TYPEERROR
                             else bool
                           ]
                    ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

ENV : ID → TYPE

```
typchk( def(s, E1, E2) , env ) ≜
                                [ t1 = typchk( E1, env);
                                  if ( t1 == TYPEERROR) then TYPEERROR
                                  else envlocal = env.BeginScope();
                                  envlocal.assoc(s, t1);
                                  t2 = typchk(E2, envlocal);
                                  env = envlocal.EndScope();
                                  t2 end ]
```

Decidability and Soundness of Typing

- Notice that for any program P and environment *env* which covers all free identifiers of P the typechecking operation

typchk(P, *env*)

is well defined and always terninates [Why?]

- We may also prove the following result, which related well typing with program evaluation

Theorem: For any CALCS program P and type $\mathcal{T}$,
If **typchk**(P,∅) = $\mathcal{T}$ and **eval**(P, ∅, ∅) = v then v : $\mathcal{T}$.

Decidability and Soundness of Typing

Theorem: For any CALCS program P and type $\mathcal{T}$,
If **typchk**(P,$\emptyset$) = $\mathcal{T}$ and **eval**(P, $\emptyset$, $\emptyset$) = v then v : $\mathcal{T}$.

That is, if **typchk**(P,$\emptyset$) = $\mathcal{T}$ then:

either P does not terminate (loops for ever),

or P terminates and does not get into any runtime errors due to illegal operations.

"*Well-typed programs don't go wrong*" (Robin Milner)

Decidability and Soundness of Typing

Theorem: For any CALCS program P and type $\mathcal{T}$,
If **typchk**(P,∅) = $\mathcal{T}$ and **eval**(P, ∅, ∅) = v then v : $\mathcal{T}$.

The property stated in the Theorem above is known as "Type Safety". It implies that

"*Well-typed programs don´t go wrong*" (Robin Milner)

NOTE (incompleteness of type checking)
There are many programs P such that eval(P, ∅, ∅) = v but typchk(P, ∅) = **TYPEERROR** ! [Can you give an example?]

Robin Milner

ACM Turing Award (1991)

For three distinct and complete achievements:

1) LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2) ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3) CCS, a general theory of concurrency. In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

Quiz

1. Is the program below well-typed? if not, what is a corrected version.
2. what is the typing environment at `w := ...` ?
3. what is the output of the (possibly corrected) program ?

```
def
        x = 10
        y = new(0)
in
  def
        z = new(y)
        w = new(false)
  in
        while !w do
           w := ((!z := !!z + !y + 1) < x)
        end;
        println !y
  end
end
```

Quiz

1. Is the program below well-typed? if not, what is a corrected version.
2. what is the typing environment at `w := ...` ?
3. what is the output of the (possibly corrected) program ?

```
def
        x = 10              x -> INT
        y = new(0)          y -> REF[INT]
in
  def
        z = new(y)
        w = new(false)
  in
        while !w do
           w := ((!z := !!z + !y + 1) < x)
        end;
       println !y
  end
end
```

Quiz

1. Is the program below well-typed? if not, what is a corrected version.
2. what is the typing environment at `w := ...` ?
3. what is the output of the (possibly corrected) program ?

```
def
        x = 10           x -> INT
        y = new(0)       y -> REF[INT]
in
  def
        z = new(y)       z -> REF[REF[INT]]
        w = new(false)   w -> REF[BOOL]
  in
        while !w do      % !w : BOOL
          w := ((!z := !!z + !y + 1) < x)
        end;
       println !y
  end
end
```

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Functional Abstraction

Every programming language needs to incorporate abstraction mechanisms.

Abstraction constructs allow parametrised expressions to be defined and used in the appropriate contexts.

We may find simple functional or procedural abstraction, used in function and procedure definitions, type abstraction, used in generics, and so on…

In this module we will study how to incorporate functional abstraction in our core programming language by introducing functional values, also called first class functions.

- summarise

First class functions

A first class function is a special value defined by a parametrised expression

fun $x_1, \ldots, x_n \rightarrow$ E **end**

Such expression is called an **abstraction** (or λ-abstraction)

The identifiers $x_1, \ldots, x_n$ are the abstraction **parameters**.

The parameters are binding occurrences, with scope the abstraction **body** E, where E is **any expression of the language**.

First class functions

An abstraction is a syntactical expression that denotes a function.

A function is a special value F that supports an application operation F.apply, we may apply the function F to a value v, to obtain a value as result F.apply(v)

A programming language may support functions but not abstractions (C, C++)

Most modern languages support abstraction:

Python: `lambda x : x*2`

Rust: `|x| { x*2 }`

JavaScript: `(x) => x*2`

OCaml: `fun x -> x *2`

CALCF: `fun x -> x*2 end`

The origin of functional abstraction is Church's lambda calculus.

The λ-Calculus

- The lambda-calculus is a minimal programming language created by Alonzo Church in 1936, it is the basis for all abstraction mechanisms used in modern programming
- It uses functions as "first class" entities, and albeit extremely simple, can represent all constructs of Turing complete programming languages.
- Syntax of the lambda-calculus:

$$\mathbf{E ::= } \quad x \quad | \quad \lambda x.\mathbf{E} \quad | \quad \mathbf{E1\ E2}$$

- Abstract syntax:

var:	string → LAMBDA	(x)
abs:	string × LAMBDA → LAMBDA	$(\lambda x.\mathbf{E})$
app:	LAMBDA × LAMBDA → LAMBDA	**(E1 E2)**

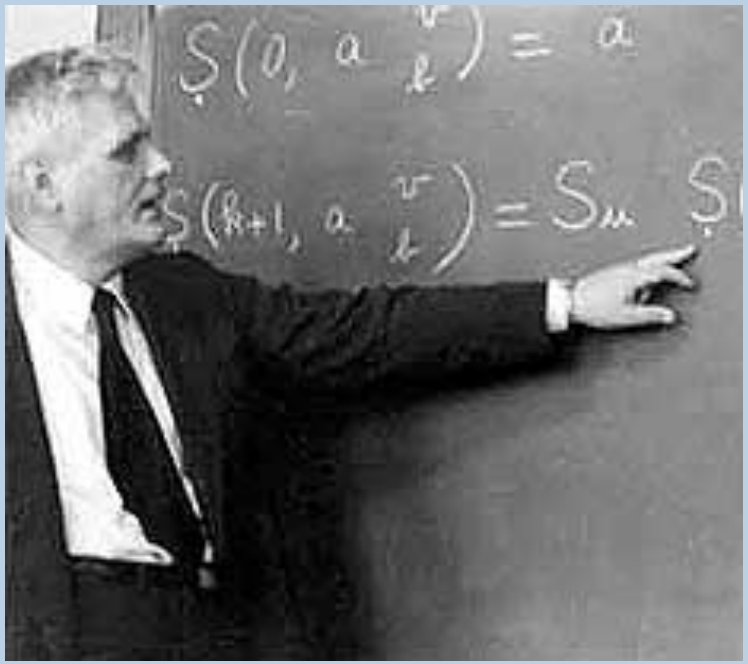
- A multi-parameter function and application is defined using "currying"

$$\lambda x_1 \lambda x_2 \ldots \lambda x_n.\mathbf{B} \qquad\qquad \mathbf{F\ E_1\ E_2 \ldots E_n}$$

Alonzo Church (1903-1995)

Church Thesis:

"The set of all computable functions is the set of functions defined in the lambda-calculus."



Church was Alan Turing PhD advisor.

Together, they have shown that the lambda calculus has the same computational power as the Turing machine.

The language CALCF (abstract syntax)

num:	Integer → CALCF
id:	String → CALCF
add:	CALCF × CALCF → CALCF
mul:	CALCF × CALCF → CALCF
div:	CALCF × CALCF → CALCF
sub:	CALCF × CALCF → CALCF
….	
def:	List(String × CALCF) × CALCF → CALCF
fun:	String × CALCF → CALCF
app:	CALCF × CALCF → CALCF

The language CALCF (example)

```
fun x -> x*2 end (4);;
```

The language CALCF (example)

```
def f = fun x -> x+1 end
  in
  def g = fun y -> f(y)+2 end
    in
    def x = g(2)
      in
       x+x
      end
  end
end;;
```

Semantics of CALCF

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCF expression:

$$\text{eval} : \text{CALCF} \times \text{ENV}\langle\text{Value}\rangle \rightarrow \text{Value}$$

CALCF = open programs

ENV = environments

Value = Bool U Integer U Closure U {ERROR}

Semantics of CALCF

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCF expression:

$$\text{eval} : \text{CALCF} \times \text{ENV<Value>} \rightarrow \text{Value}$$

CALCF = open programs

ENV = environments

Value = Bool U Integer U Closure U {ERROR}

Closures are special values used to represent functions.

A closure is a triple of the form [id, E, *env*]

id is an identifier (the closure parameter)

E is an expression (the closure body)

env is an environment (must declare all free names in body except id)

- The closure body E may be evaluated by providing some value for the parameter id (this will correspond to function call)

CALCF Interpreter

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCF expression:

$$eval : CALCI \times ENV\langle Value\rangle \rightarrow Value$$

```
eval( num(n) , env)          ≜ n
eval( id(s) , env)           ≜ env.Find(s)
eval( add(E1,E2) , env)      ≜ eval(E1, env) + eval(E2, env)
eval( abs(s, E), env) ≜  {   return [ s, E, env ] ;  }
eval( app(E1, E2), env) ≜ { v1 = eval(E1,env); v2 = eval(E2,env);
                             if v1 is [ param, B, env0 ] then
                               e = env0. beginScope();
                               e.assoc(param,v2);
                               result = B.eval(e);
                               e = e. endScope();
                               return result;
                            else ERROR
                           }
```

Example Evaluation

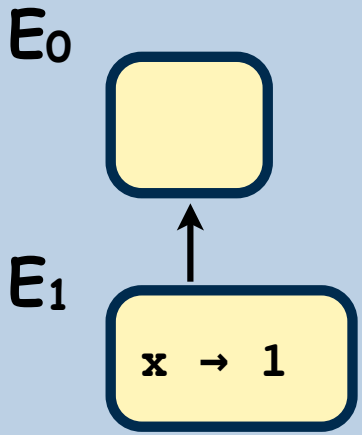
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
    end
  end
end
```

Example Evaluation

E_0

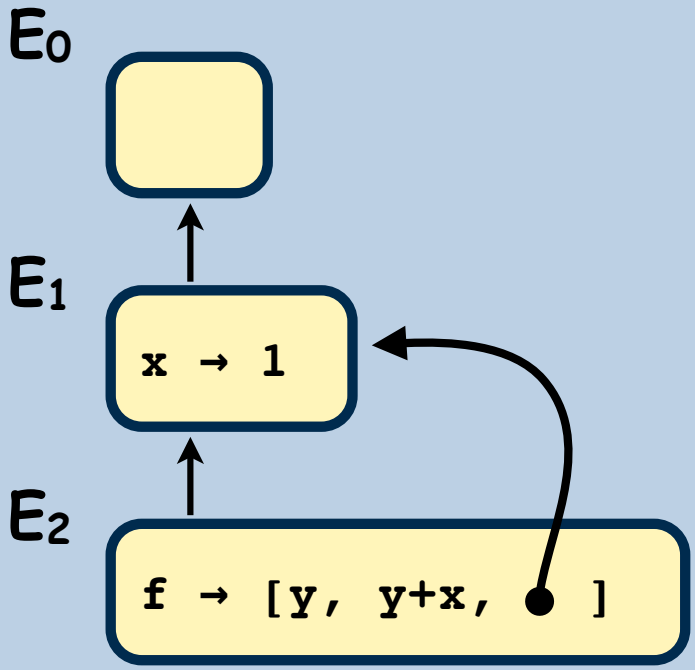
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2) end end end;;
```

Example Evaluation



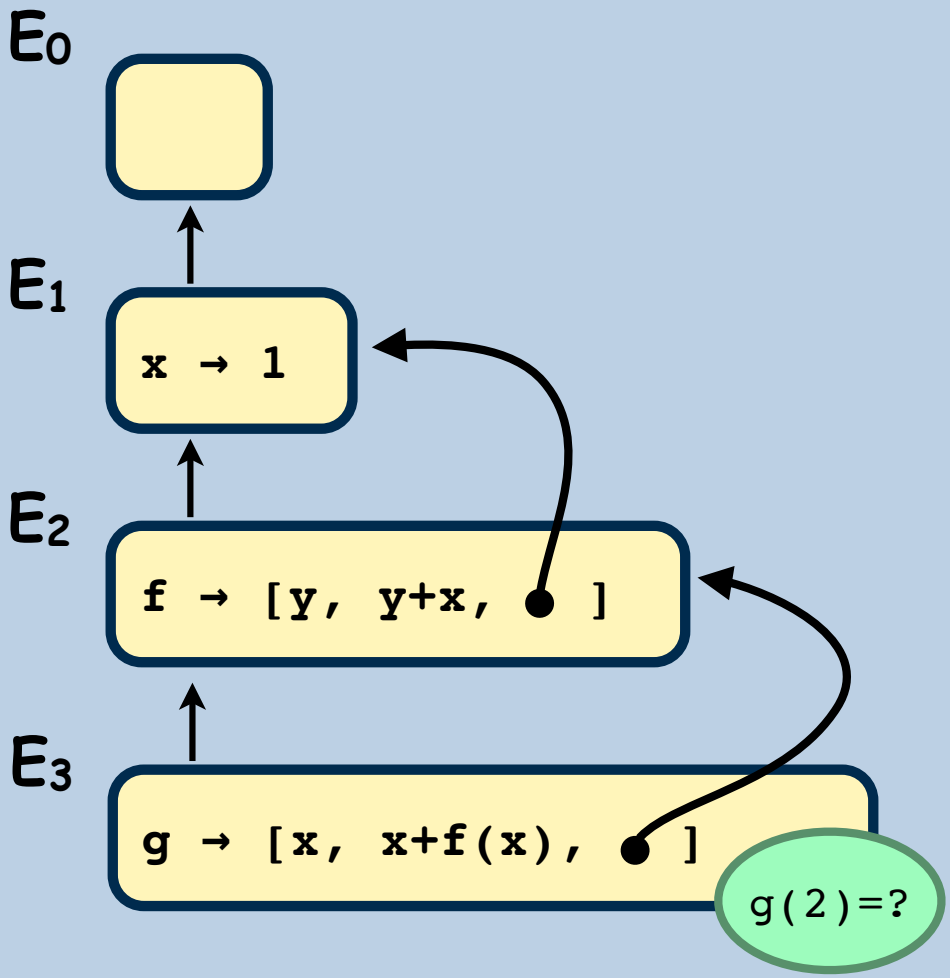

```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation



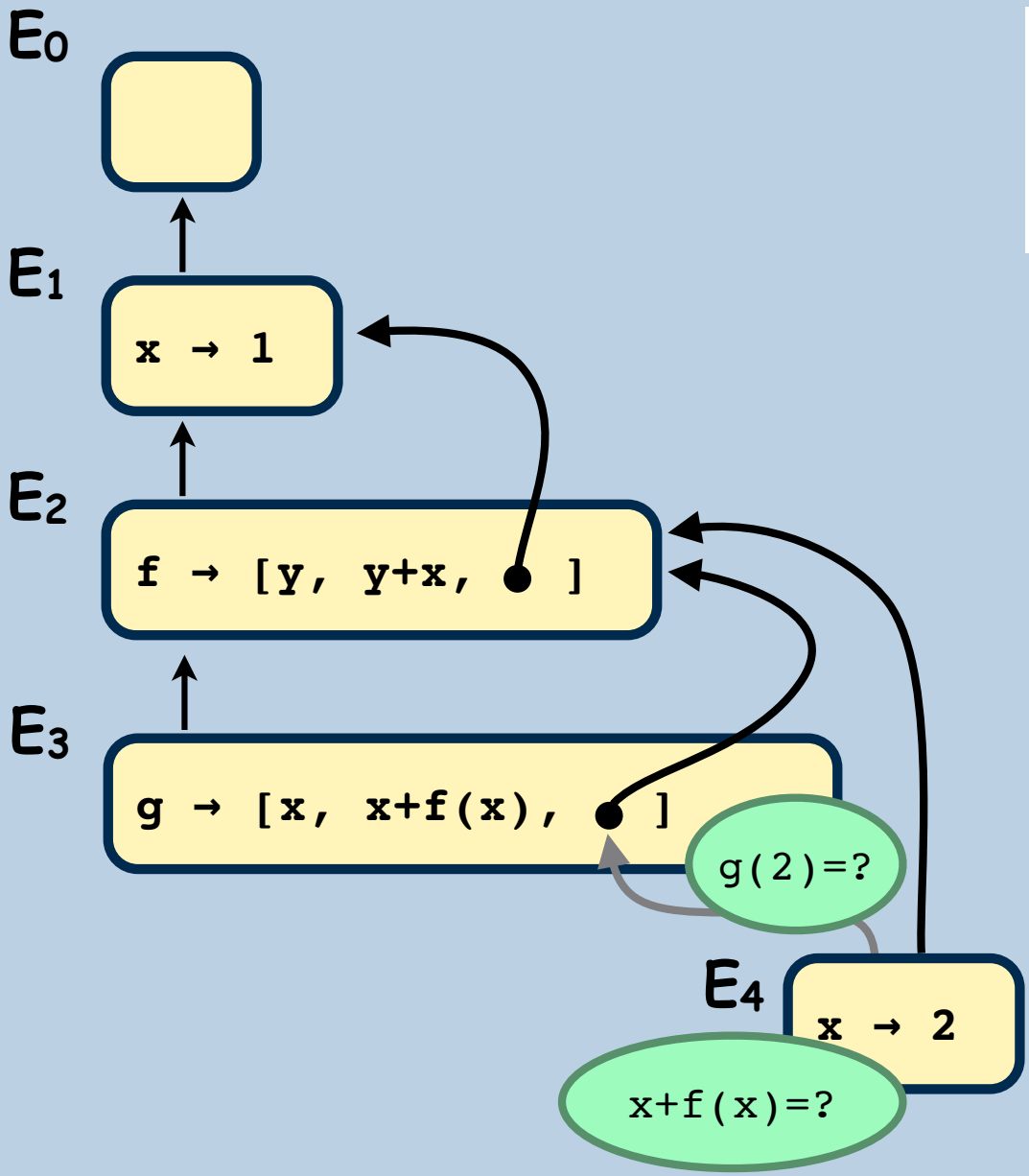

```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation




```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

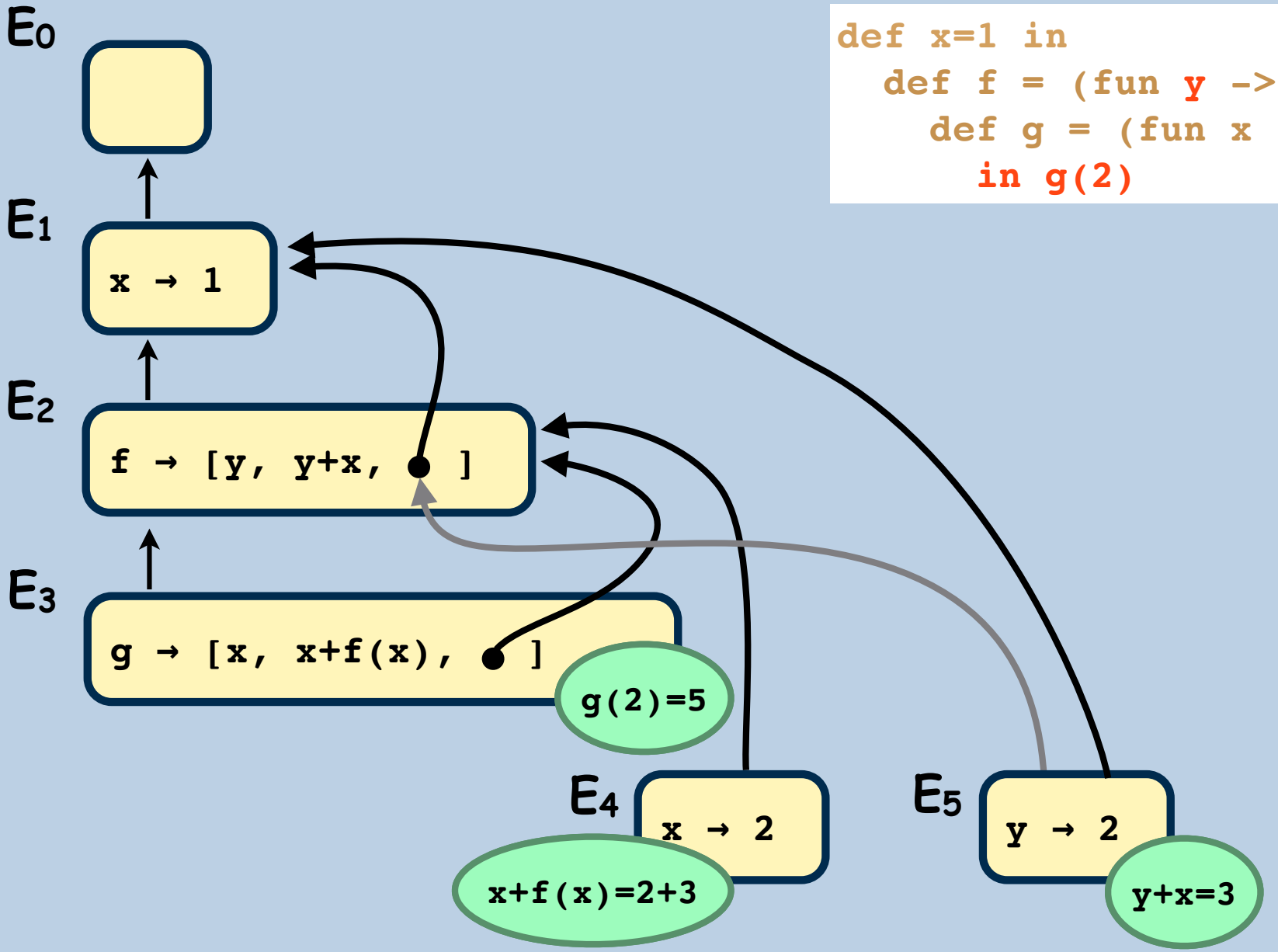
Example Evaluation




```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation

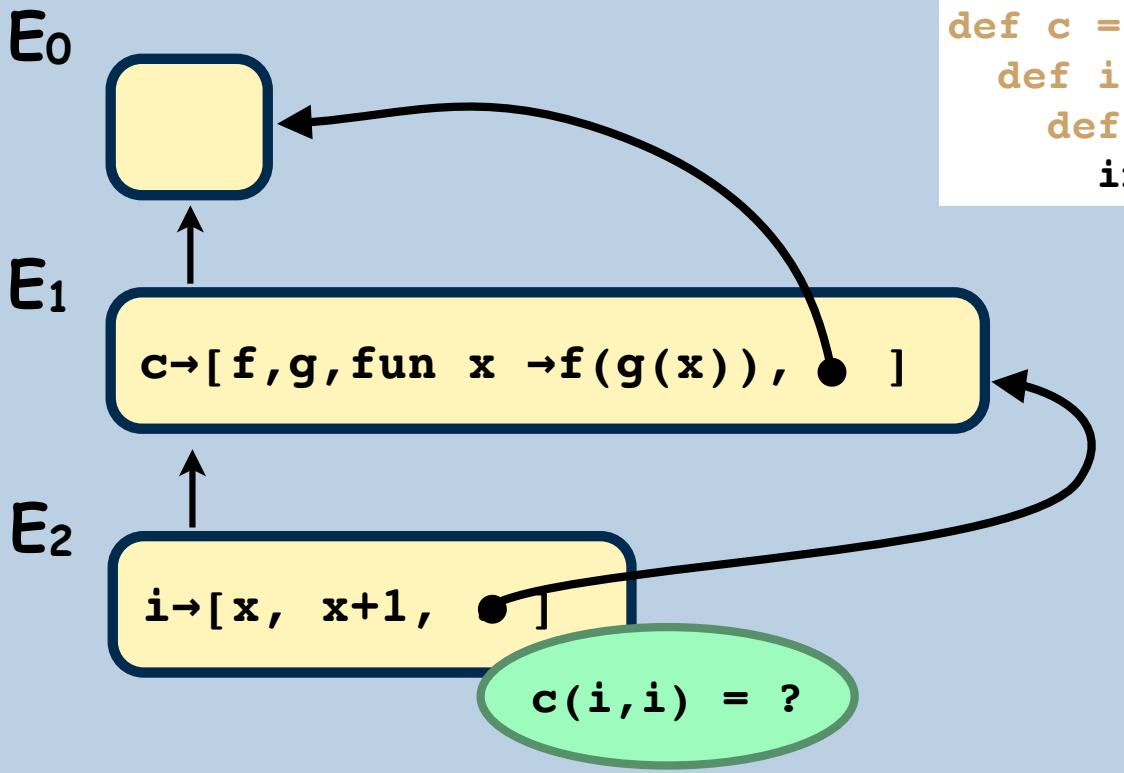
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```



Example Evaluation

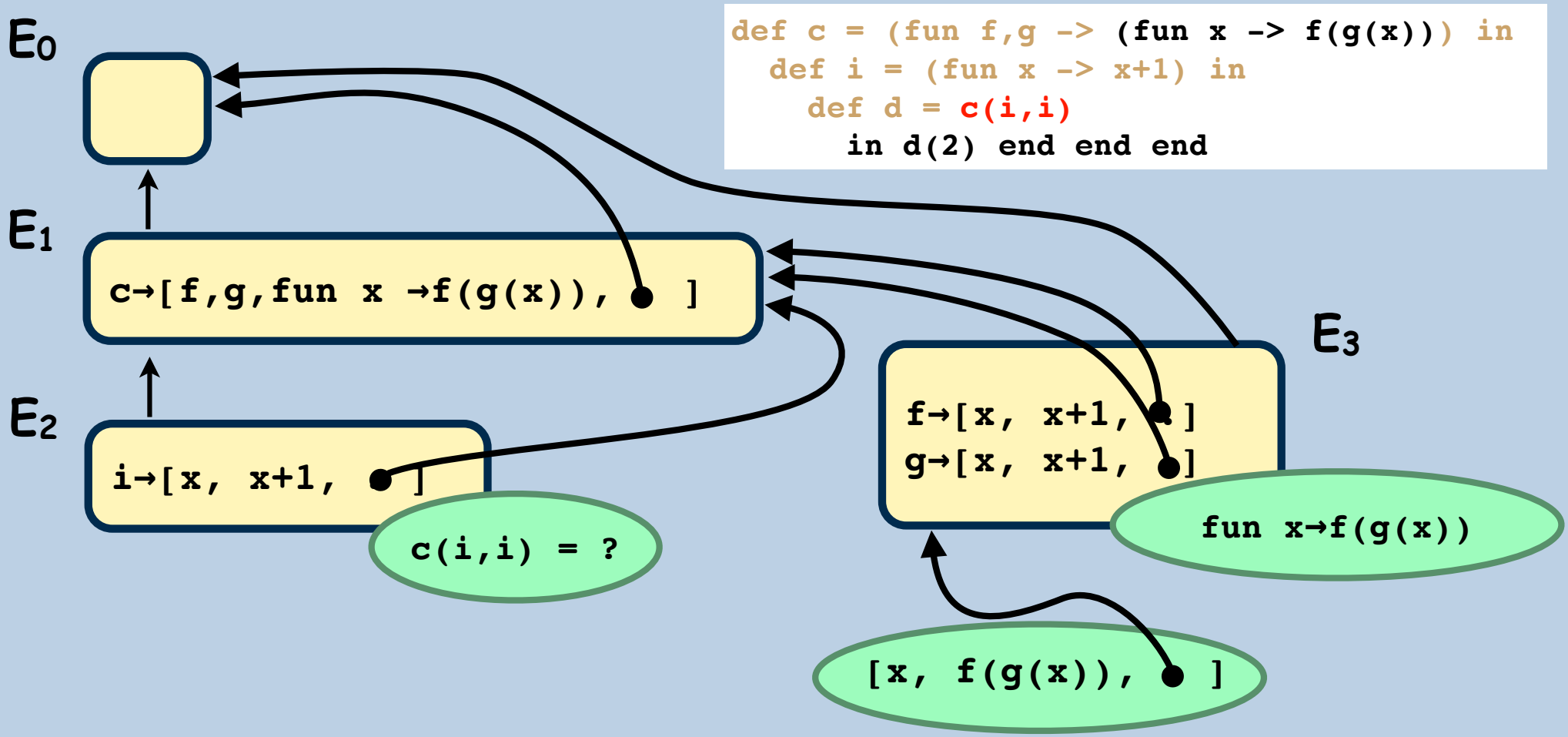
```
def comp = (fun f,g -> (fun x -> f(g(x))))
  in
  def inc = (fun x -> x+1)
    in
    def dup = comp(inc,inc)
      in dup(2)
    end
  end
end
```

Example Evaluation



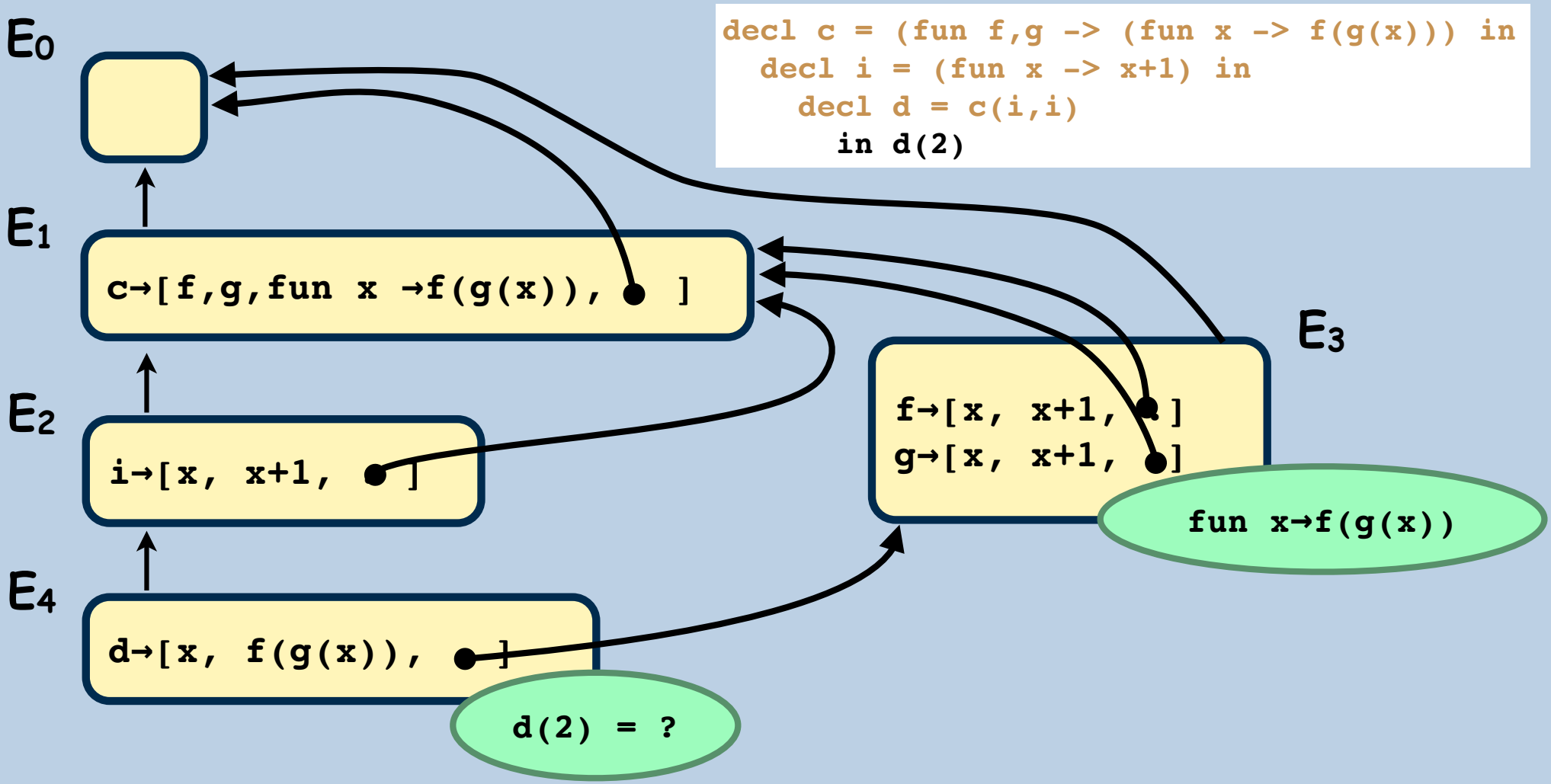

```
def c = (fun f,g -> (fun x -> f(g(x))) in
  def i = (fun x -> x+1) in
    def d = c(i,i)
      in d(2) end end end
```

Example Evaluation

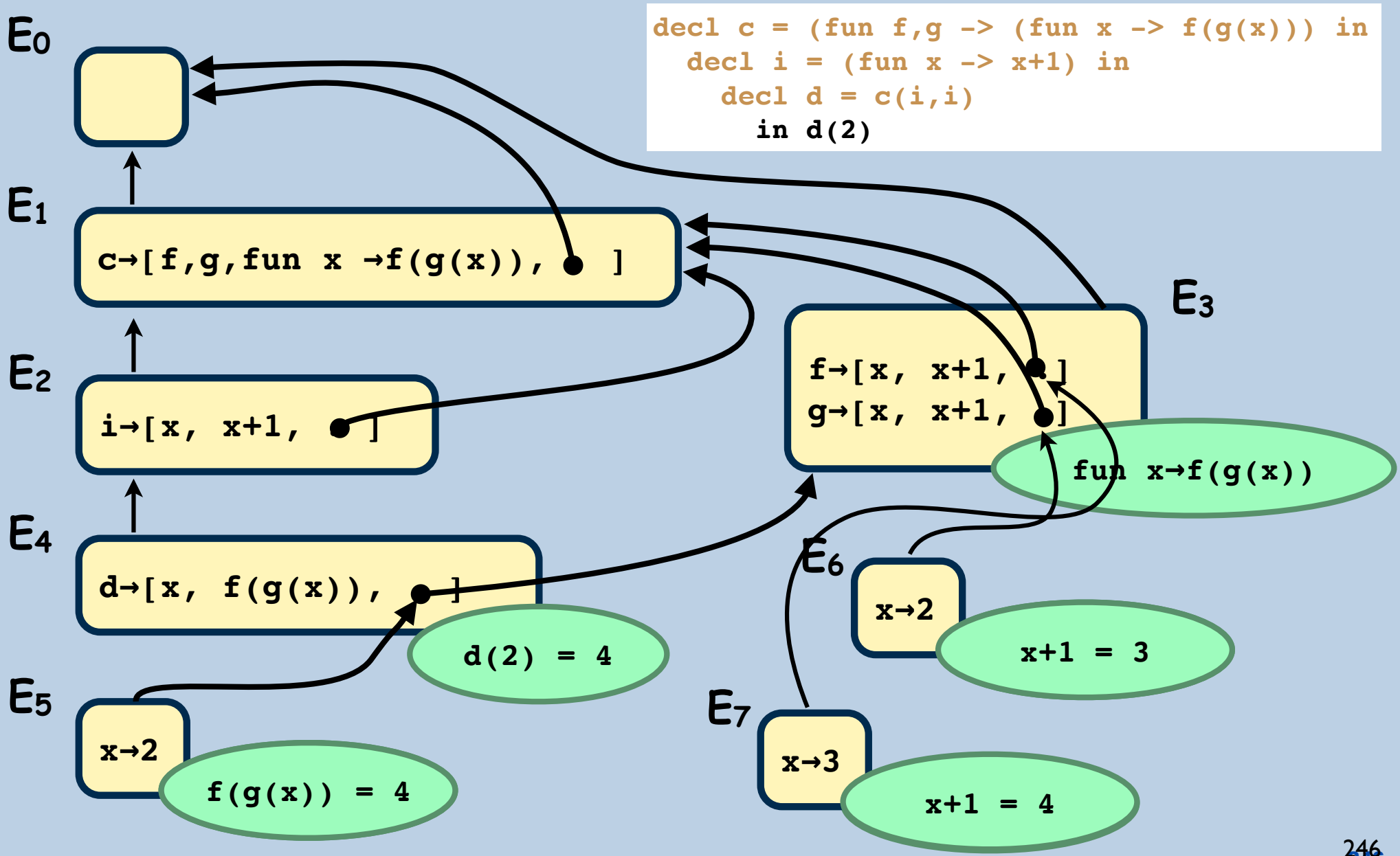



```
def c = (fun f,g -> (fun x -> f(g(x)))) in
  def i = (fun x -> x+1) in
    def d = c(i,i)
      in d(2) end end end
```

Example Evaluation



Example Evaluation



CALCF Types and Typechecking

- Map **eval** to compute a value + effect for any CALCS programs:

$$\text{eval} : \text{CALCS} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **typchk** that computes a type for a CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \ \text{bool}, \ \text{ref}\{\text{TYPE}\}, \ \text{fun}[\text{TYPE},\text{TYPE}] \ \}$$

CALCF Types and Typechecking

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCF} \times \text{ENV<TYPE>} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{fun}\{\text{TYPE}, \text{TYPE}\} \}$$

int: type of integer values.

bool: type of boolean values.

ref{$\mathcal{T}$}: type of references that may only hold values of type $\mathcal{T}$.

fun{$\mathcal{A}$,$\mathcal{B}$}: type of functions that take arg of type $\mathcal{A}$ and return a value of type $\mathcal{B}$

Example: fun{int,bool} is the type functions that take an integer as argument and return a boolean

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( fun(s:T0, E) , env ) ≜ {
                                    env0 = env.beginScope();
                                    env 0= env0.assoc(s,T0);
                                    t1 =  typchk ( E, env0);
                                    env.endScope();
                                    if (t1 == TYPEERROR) then return t1
                                    else return fun(T0,t1);
                                    }
```

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

typchk(**fun**(s:**T0**, E) , *env*) ≜ {

env0 = *env*.beginScope();

env 0= *env0.*assoc(s,T0);

t1 = **typchk** (E, *env0*);

*env.*endScope();

if (t1 == **TYPEERROR**) **then** return t1

else return fun(T0,t1);

}

need to declare argument type!

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( app(E1, E2) , env ) ≜ { t1 =  typchk ( E1, env);
                                if (t1 is fun(T0,T1)) then
                                    t2 =  typchk ( E2, env);
                                    if (t2 != T0) then return TYPEERROR;
                                    return T1;
                                else return TYPEERROR
                                }
```

Sample Typed Program

```
def  f  = fun n:int, b:int ->
             def
               x  = new n
               s = new b
             in
                while !x>0 do
                   s := !s + !x ;
                   x := !x – 1
                end;
               !s
            end
         end
in
   f(10,0)+f(100,20)
end;;
```

Sample Typed Program

```
def  g = new 0
in
def f  = fun n:int -> g := !g + n end
in
    f(2);
    f(3);
    f(4);
    println !glo
end;;
```

CALCF Compilation

Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Functional Abstraction

Every programming language needs to incorporate abstraction mechanisms.

Abstraction constructs allow parametrised expressions to be defined and used in the appropriate contexts.

We may find simple functional or procedural abstraction, used in function and procedure definitions, type abstraction, used in generics, and so on…

In this module we will study how to incorporate functional abstraction in our core programming language by introducing functional values, also called first class functions.

- summarise

First class functions

A first class function is a special value defined by a parametrised expression

fun $x_1, \ldots, x_n \rightarrow$ E **end**

Such expression is called an **abstraction** (or λ-abstraction)

The identifiers $x_1, \ldots, x_n$ are the abstraction **parameters**.

The parameters are binding occurrences, with scope the abstraction **body** E, where E is **any expression of the language**.

First class functions

An abstraction is a syntactical expression that denotes a function.

A function is a special value F that supports an application operation F.apply, we may apply the function F to a value v, to obtain a value as result F.apply(v)

A programming language may support functions but not abstractions (C, C++)

Most modern languages support abstraction:

Python: `lambda x : x*2`

Rust: `|x| { x*2 }`

JavaScript: `(x) => x*2`

OCaml: `fun x -> x *2`

CALCF: `fun x -> x*2 end`

The origin of functional abstraction is Church's lambda calculus.

The λ-Calculus

- The lambda-calculus is a minimal programming language created by Alonzo Church in 1936, it is the basis for all abstraction mechanisms used in modern programming
- It uses functions as "first class" entities, and albeit extremely simple, can represent all constructs of Turing complete programming languages.
- Syntax of the lambda-calculus:

$$\mathbf{E ::= } \quad x \quad | \quad \lambda x.\mathbf{E} \quad | \quad \mathbf{E1\ E2}$$

- Abstract syntax:

var:	string → LAMBDA	(x)
abs:	string × LAMBDA → LAMBDA	$(\lambda x.\mathbf{E})$
app:	LAMBDA × LAMBDA → LAMBDA	**(E1 E2)**

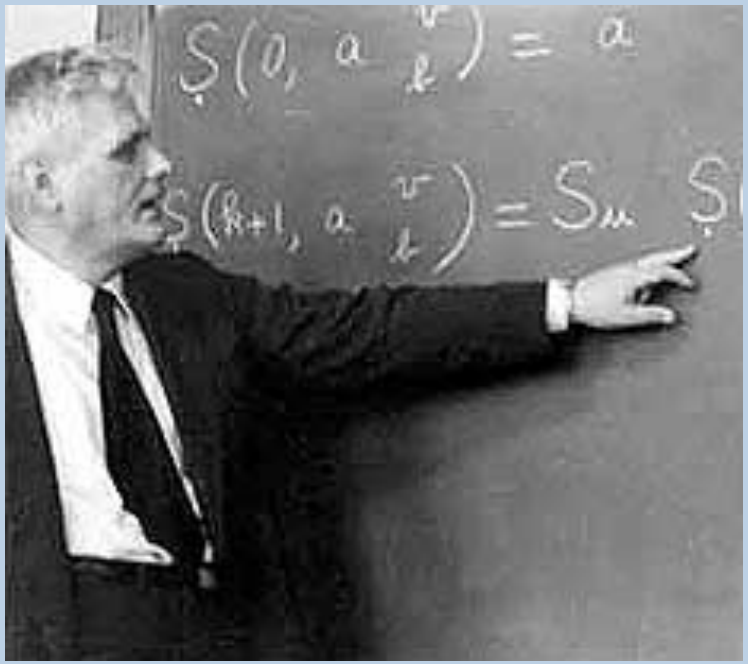
- A multi-parameter function and application is defined using "currying"

$$\lambda x_1 \lambda x_2 \ldots \lambda x_n.\mathbf{B} \qquad\qquad \mathbf{F\ E_1\ E_2 \ldots E_n}$$

Alonzo Church (1903-1995)

Church Thesis:

"The set of all computable functions is the set of functions defined in the lambda-calculus."



Church was Alan Turing PhD advisor.

Together, they have shown that the lambda calculus has the same computational power as the Turing machine.

The language CALCF (abstract syntax)

num:	Integer → CALCF
id:	String → CALCF
add:	CALCF × CALCF → CALCF
mul:	CALCF × CALCF → CALCF
div:	CALCF × CALCF → CALCF
sub:	CALCF × CALCF → CALCF
….	
def:	List(String × CALCF) × CALCF → CALCF
fun:	String × CALCF → CALCF
app:	CALCF × CALCF → CALCF

The language CALCF (example)

```
fun x -> x*2 end (4);;
```

The language CALCF (example)

```
def f = fun x -> x+1 end
  in
  def g = fun y -> f(y)+2 end
    in
    def x = g(2)
      in
       x+x
      end
  end
end;;
```

Semantics of CALCF

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCF expression:

$$\text{eval} : \text{CALCF} \times \text{ENV}\langle\text{Value}\rangle \rightarrow \text{Value}$$

CALCF = open programs

ENV = environments

Value = Bool U Integer U Closure U {ERROR}

Semantics of CALCF

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCF expression:

$$\text{eval} : \text{CALCF} \times \text{ENV<Value>} \rightarrow \text{Value}$$

CALCF = open programs

ENV = environments

Value = Bool U Integer U Closure U {ERROR}

Closures are special values used to represent functions.

A closure is a triple of the form [id, E, *env*]

id is an identifier (the closure parameter)

E is an expression (the closure body)

env is an environment (must declare all free names in body except id)

- The closure body E may be evaluated by providing some value for the parameter id (this will correspond to function call)

CALCF Interpreter

- Algorithm eval( ) that computes the denotation (integer value) of any **open** CALCF expression:

$$eval : CALCI \times ENV\langle Value\rangle \rightarrow Value$$

```
eval( num(n) , env)        ≜ n
eval( id(s) , env)         ≜ env.Find(s)
eval( add(E1,E2) , env)    ≜ eval(E1, env) + eval(E2, env)
eval( abs(s, E), env) ≜  {  return [ s, E, env ] ;  }
eval( app(E1, E2), env) ≜ { v1 = eval(E1,env); v2 = eval(E2,env);
                            if v1 is [ param, B, env0 ] then
                              e = env0. beginScope();
                              e.assoc(param,v2);
                              result = B.eval(e);
                              e = e. endScope();
                              return result;
                            else ERROR
                           }
```

Example Evaluation

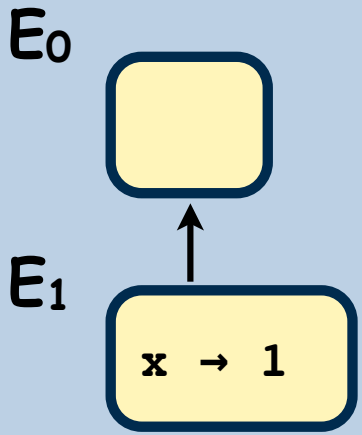
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
    end
  end
end
```

Example Evaluation

E_0

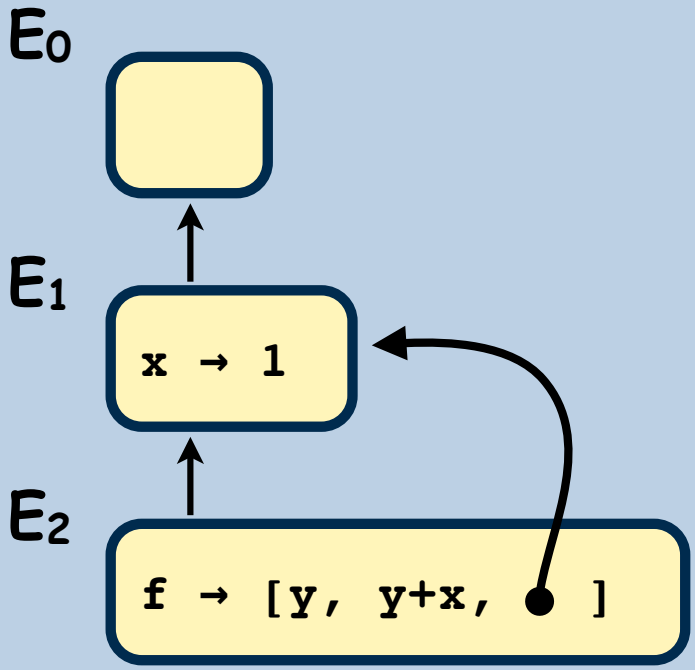
```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2) end end end;;
```

Example Evaluation



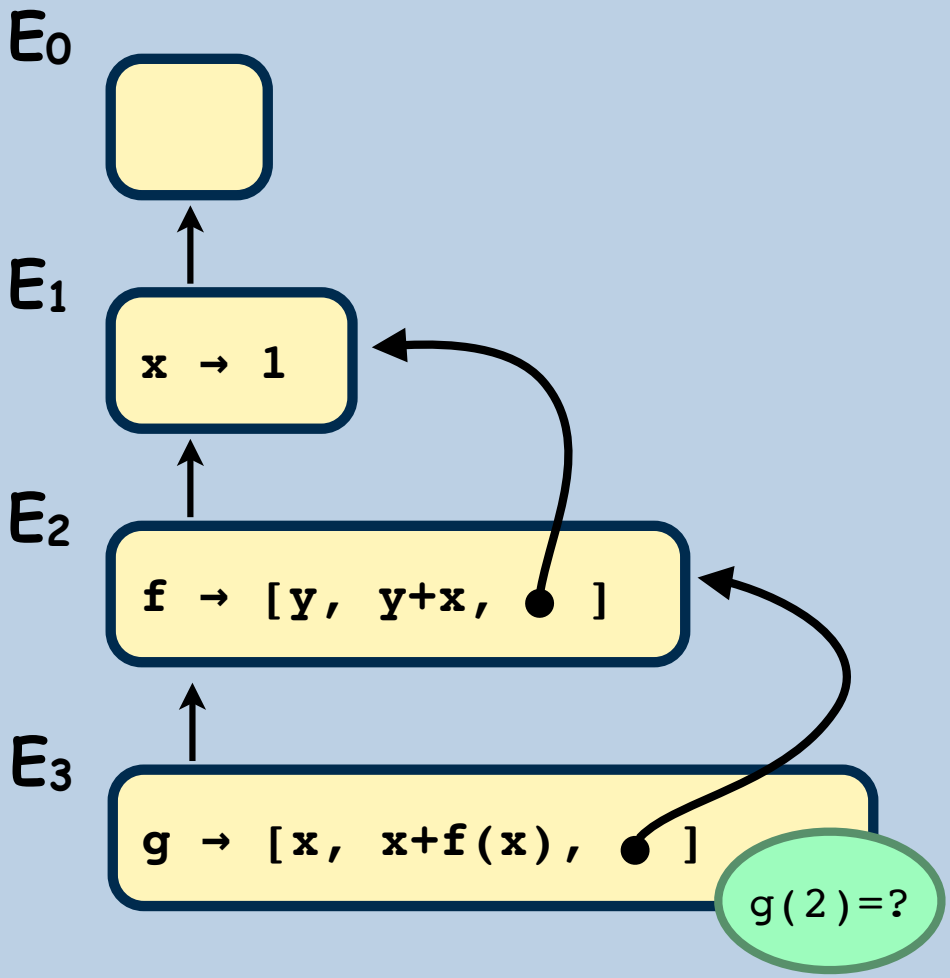

```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation



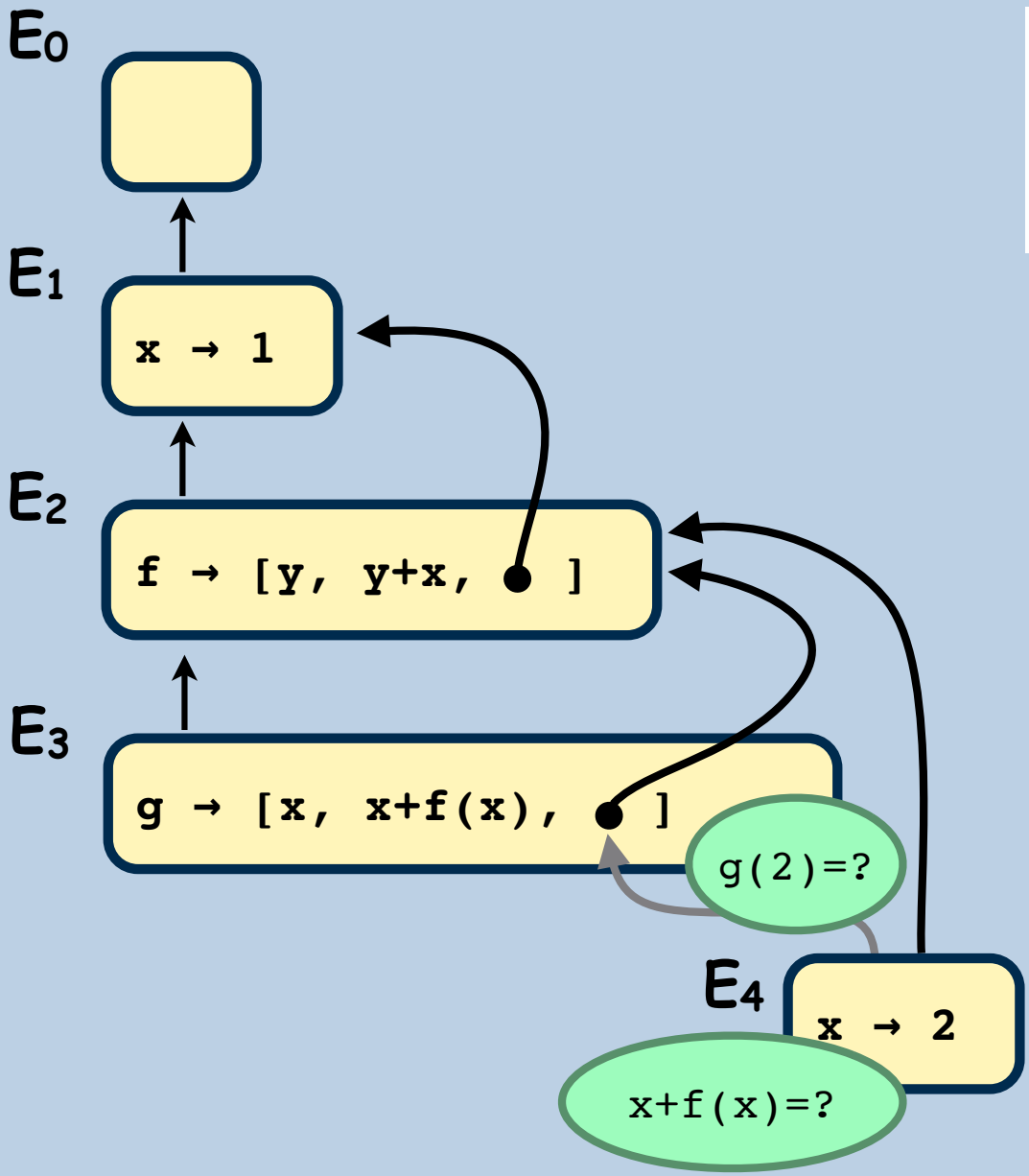

```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation



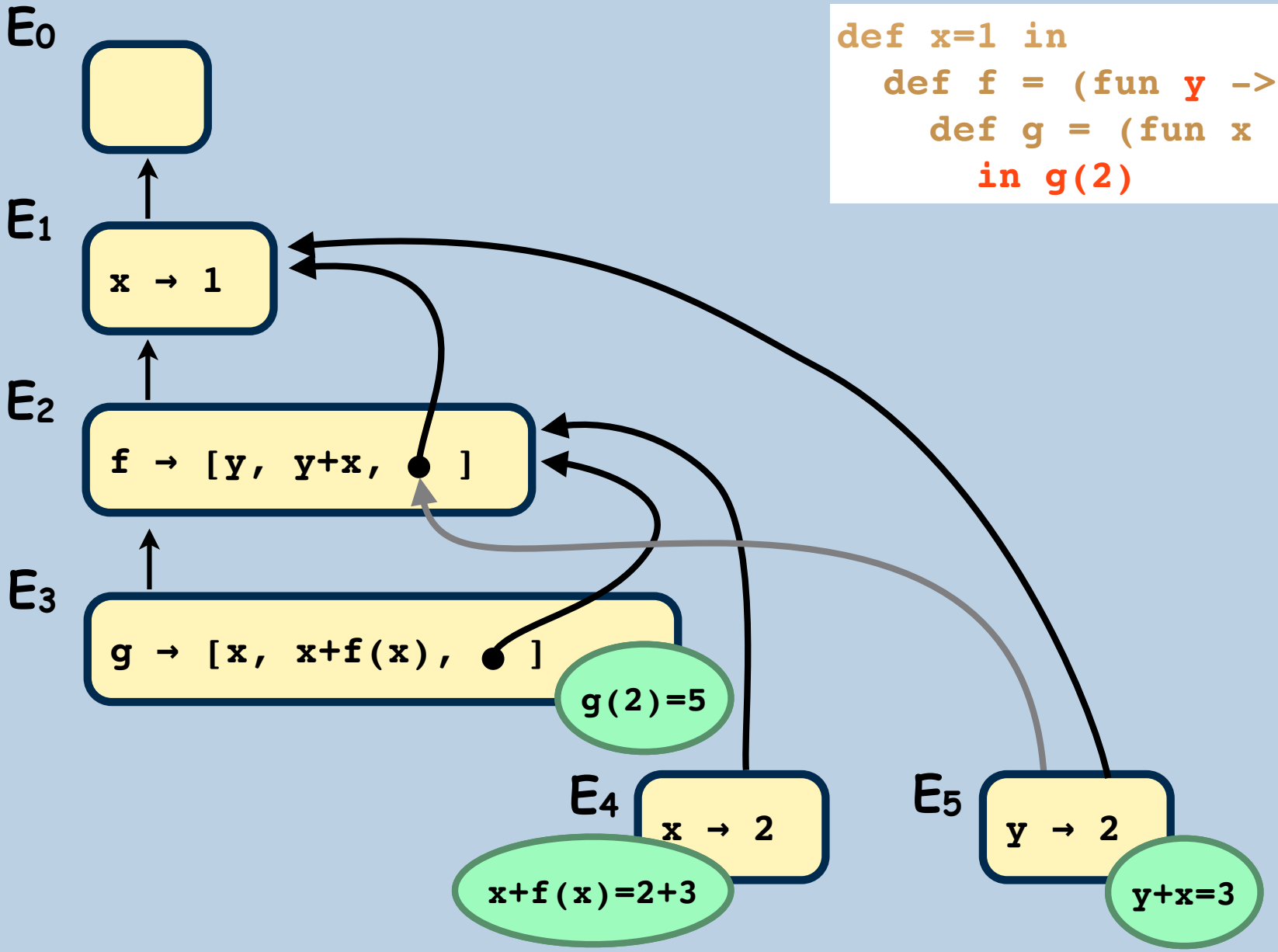

```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation




```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation

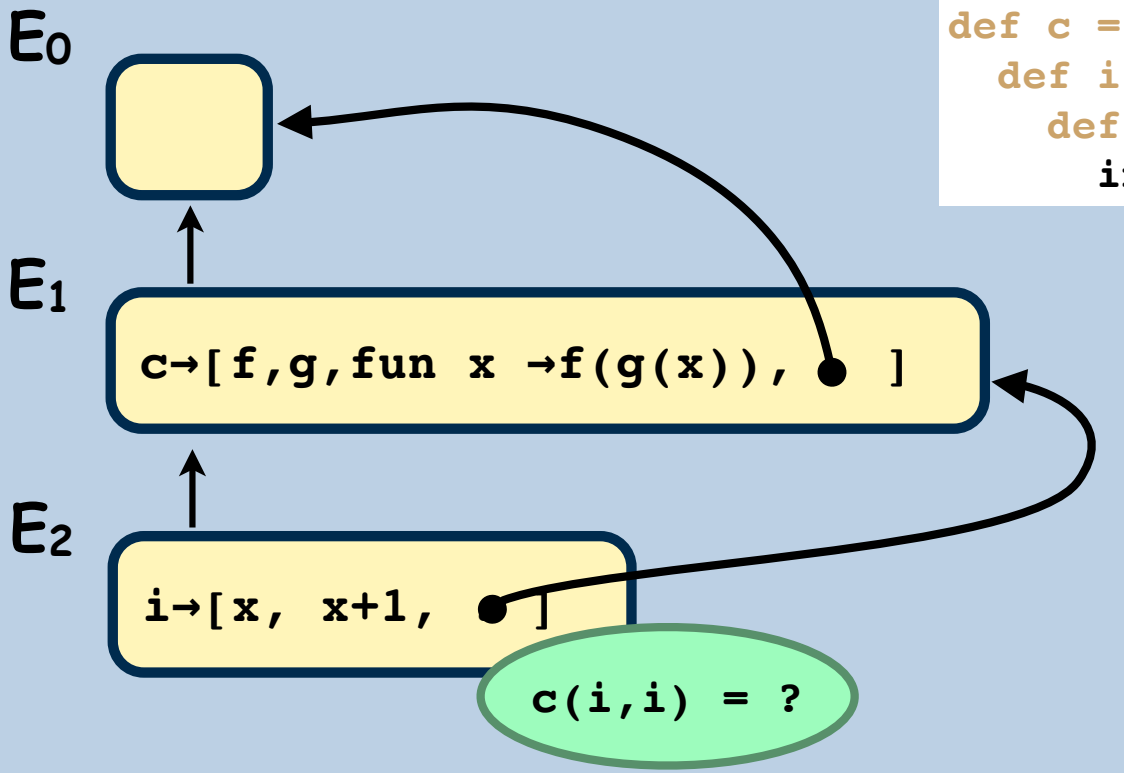



```
def x=1 in
  def f = (fun y -> y+x) in
    def g = (fun x -> x+f(x))
      in g(2)
```

Example Evaluation

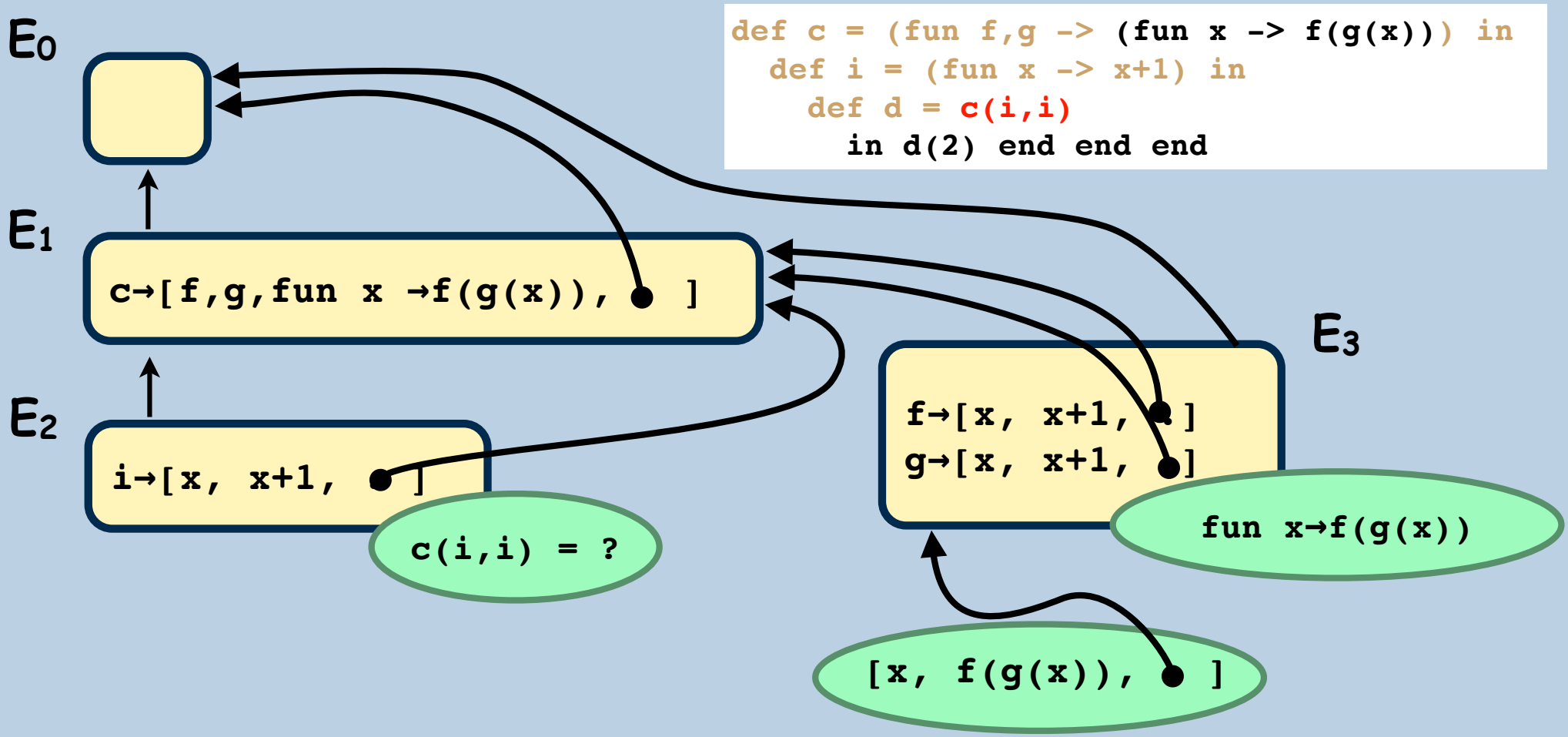
```
def comp = (fun f,g -> (fun x -> f(g(x))))
  in
  def inc = (fun x -> x+1)
    in
    def dup = comp(inc,inc)
      in dup(2)
    end
  end
end
```

Example Evaluation

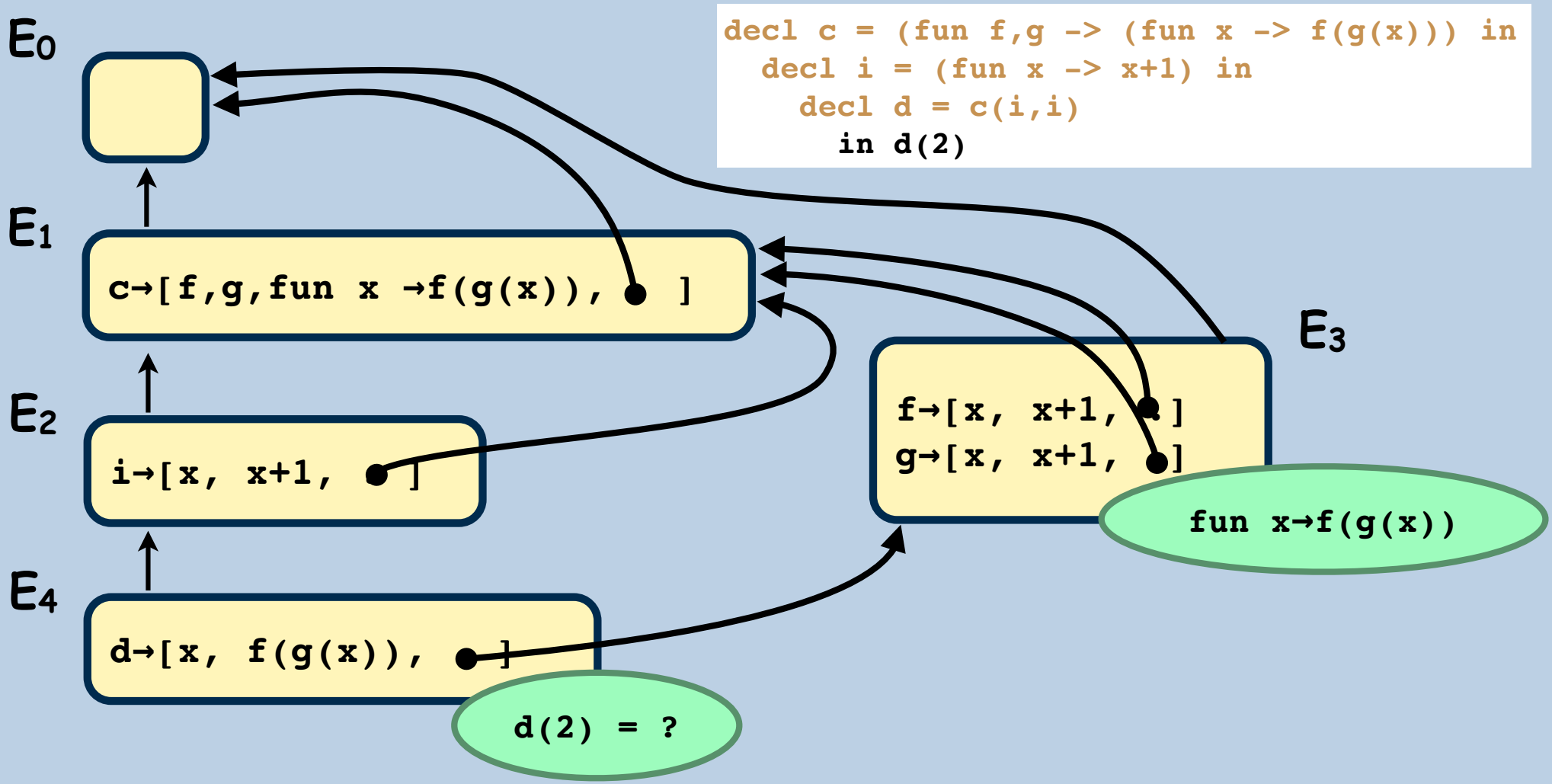



```
def c = (fun f,g -> (fun x -> f(g(x))) in
  def i = (fun x -> x+1) in
    def d = c(i,i)
      in d(2) end end end
```

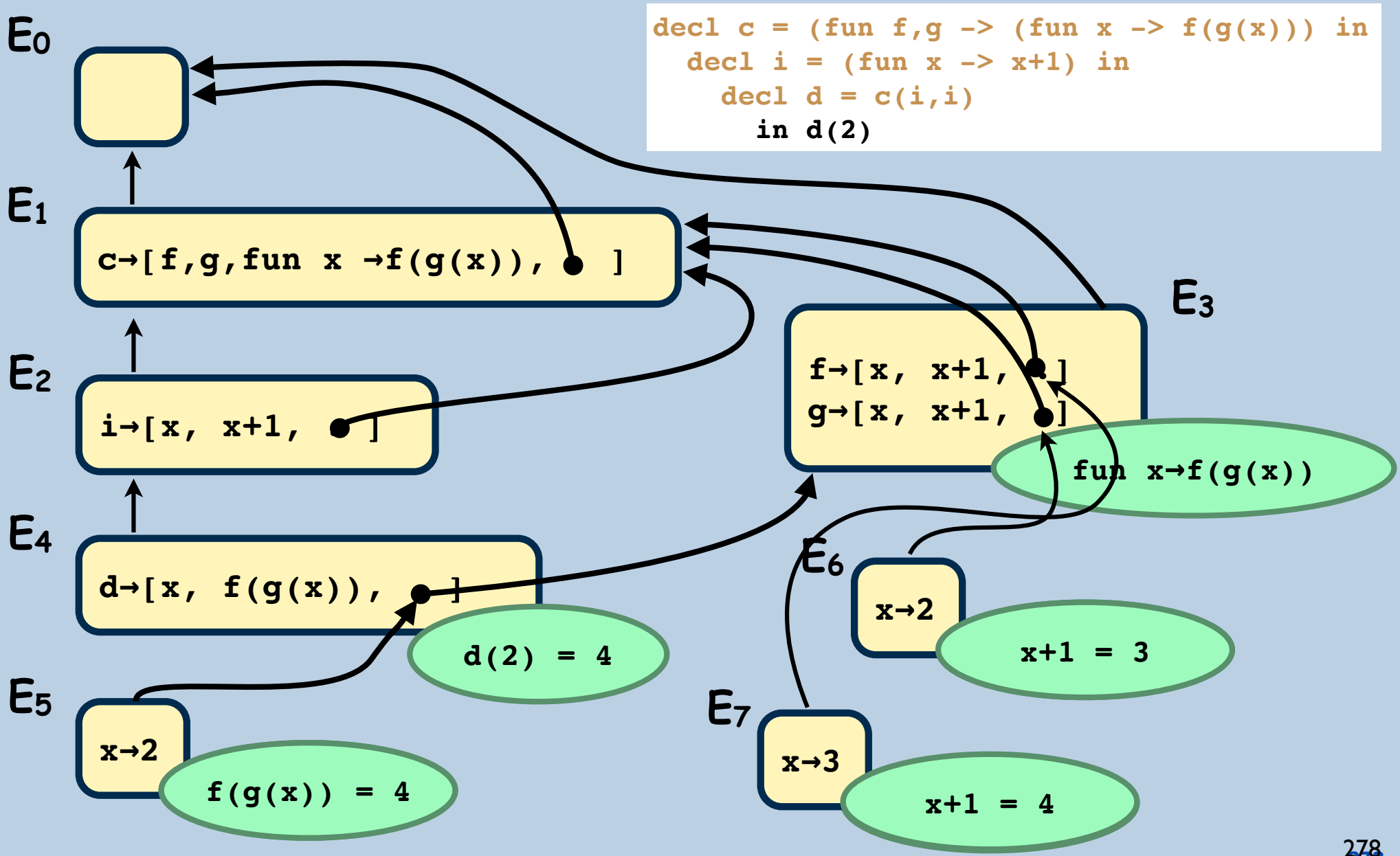
Example Evaluation



Example Evaluation



Example Evaluation



278

CALCF Types and Typechecking

- Map **eval** to compute a value + effect for any CALCS programs:

$$\text{eval} : \text{CALCS} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **typchk** that computes a type for a CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \ \text{bool}, \ \text{ref}\{\text{TYPE}\}, \ \text{fun}[\text{TYPE},\text{TYPE}] \ \}$$

CALCF Types and Typechecking

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCF} \times \text{ENV<TYPE>} \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{fun}\{\text{TYPE}, \text{TYPE}\} \}$$

int: type of integer values.

bool: type of boolean values.

ref{$\mathcal{T}$}: type of references that may only hold values of type $\mathcal{T}$.

fun{$\mathcal{A}$,$\mathcal{B}$}: type of functions that take arg of type $\mathcal{A}$ and return a value of type $\mathcal{B}$

Example: fun{int,bool} is the type functions that take an integer as argument and return a boolean

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( fun(s:T0, E) , env ) ≜ {
                                    env0 = env.beginScope();
                                    env 0= env0.assoc(s,T0);
                                    t1 =  typchk ( E, env0);
                                    env.endScope();
                                    if (t1 == TYPEERROR) then return t1
                                    else return fun(T0,t1);
                                    }
```

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

typchk(**fun**(s:**T0**, E) , *env*) ≜ {

env0 = *env*.beginScope();

env 0= *env0.*assoc(s,T0);

t1 = **typchk** (E, *env0*);

*env.*endScope();

if (t1 == **TYPEERROR**) **then** return t1

else return fun(T0,t1);

}

need to declare argument type!

CALCF Typing Rules

- Map **typchk** that computes a type for any CALCF program:

  typechk : CALCF × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( app(E1, E2) , env ) ≜ { t1 =  typchk ( E1, env);
                                if (t1 is fun(T0,T1)) then
                                    t2 =  typchk ( E2, env);
                                    if (t2 != T0) then return TYPEERROR;
                                    return T1;
                                else return TYPEERROR
                                }
```

Sample Typed Program

```
def  f  = fun n:int, b:int ->
             def
               x  = new n
               s = new b
             in
                while !x>0 do
                   s := !s + !x ;
                   x := !x – 1
                end;
               !s
            end
         end
in
   f(10,0)+f(100,20)
end;;
```

Sample Typed Program

```
def  g = new 0
in
def f  = fun n:int -> g := !g + n end
in
    f(2);
    f(3);
    f(4);
    println !glo
end;;
```

CALCF Compilation

Interpretação e Compilação de Linguagens de Programação

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Luís Caires

Abstract Syntax

num: **integer** → EXP

id: **string** → EXP

add: EXP × EXP → EXP

and: EXP × EXP → EXP

eq: EXP × EXP → EXP

var: EXP → EXP

assign: EXP × EXP → EXP

if: EXP × EXP × EXP → EXP

while: EXP × EXP → EXP

deref: EXP → EXP

do: COM × EXP → EXP

let: **string** × EXP × EXP → EXP

fun: **string** × EXP → EXP

call: EXP × EXP → EXP

Example

```
let  f = fun n, b->
               let
                 x = new n
                 s = new b
               in
                   while !x>0 do
                      s := !s + !x ; x := !x – 1
                   end;
                   !s
               end
          end
in
  f(10,0) + f(100,20)
end;;
```

Type System

- Types ($\mathcal{T}$)

  int

  bool

  Ref($\mathcal{T}$) : Type of references to values of type $\mathcal{T}$.

  $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n)\mathcal{T}$: Type of functions that take *n* arguments of types respectively $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$ and return a value of type $\mathcal{T}$.

- Examples of valid typing assertions:

  x: Ref(int) $\vdash$ **while** (!x<0) **do** x:=!x+1 **end** ok

  y: int $\vdash$ (**fun** x: int -> $y + x$ **) :** (int)int

Typed Abstract Syntax

- To define a simple type checking algorithm ensuring type unicity we extend our language with type declarations in declarations and function parameters

let: **string** × TYPE × EXP × EXP → EXP

fun: **string** × TYPE × EXP → EXP

Example:s

fun x:int → $y + x$ **end**

let f: (int)int = **fun** x:int → $f(x\text{-}1)$ **in** ... **end**

Typed Abstract Syntax

- To define a simple type checking algorithm ensuring type unicity we extend our language with type declarations in declarations and function parameters

Ty -> **int**	ASTIntType()
| **bool**	ASTBoolType()
| **ref** Ty	ASTRefType(IType)
| (Ty,...,Ty)Ty	ASTFunType(List< IType >, IType)

Example:s

fun x:int $\rightarrow$ $y + x$ **end**

let f: (int)int = **fun** x:int $\rightarrow$ $f(x\text{-}1)$ **in ... end**

Example

```
let  f :( int,int)int = fun n:int, b:int->
                let
                  x : ref int = new n
                  s : ref int = new b
                in
                    while !x>0 do
                       s := !s + !x ; x := !x – 1
                    end;
                    !s
                end
            end
in
  f(10,0)+f(100,20)
end;;
```

Type Rules

- Typing of expressions

R1 (id)

$$Env,\ x{:}\mathcal{T},\ Env' \vdash \mathbf{id}(x) : \mathcal{T}$$

R2 (function)

$$\frac{Env,\ x{:}\ \mathcal{T} \vdash \mathrm{E} : \mathcal{U}}{Env \vdash \mathbf{fun}(x, \mathcal{T}, \mathrm{E}) : (\mathcal{T})\ \mathcal{U}}$$

R3 (call)

$$\frac{Env \vdash \mathrm{M} : (\mathcal{T})\ \mathcal{U} \qquad Env \vdash \mathrm{N} : \mathcal{T}}{Env \vdash \mathbf{call}(\mathrm{M},\mathrm{N}) : \mathcal{U}}$$

Type Rules

- Typing of expressions

R4 (let)

$$\frac{Env,\ x\colon \mathcal{T} \vdash \mathrm{M} : \mathcal{T} \qquad Env,\ x\colon \mathcal{T} \vdash \mathrm{N} : \mathcal{U}}{Env \vdash \textbf{let}\ x\colon \mathcal{T} = \mathrm{M}\ \textbf{in}\ \mathrm{N}\ \textbf{end} : \mathcal{U}}$$

- Note that to support, recursive declarations we require type annotation of let-bound identifiers.
- If M does not type to a functional value, the type annotation could be omitted (why?)

```
let
  f : (int)int = fun x:int → f (x-1) end
  y: int = x+3
in ... end
```

Type Rules

- Typing of expressions

R5 (int)

$$Env \vdash \mathbf{num}(\mathrm{x}) : \mathbf{int}$$

R6 (+)

$$\frac{Env \vdash \mathrm{N} : \mathbf{int} \qquad Env \vdash \mathrm{M} : \mathbf{int}}{Env \vdash \mathbf{add}(\mathrm{M}, \mathrm{N}) : \mathbf{int}}$$

R7 (=)

$$\frac{Env \vdash \mathrm{N} : \mathbf{int} \qquad Env \vdash \mathrm{M} : \mathbf{int}}{Env \vdash \mathbf{eq}(\mathrm{M}, \mathrm{N}) : \mathbf{bool}}$$

Type Rules

- Typing of expressions

R6 (true) $Env \vdash \mathbf{true} : \mathbf{bool}$ $Env \vdash \mathbf{false} : \mathbf{bool}$ R7 (false)

R8 (and)
$$\frac{Env \vdash \mathrm{N} : \mathbf{bool} \qquad Env \vdash \mathrm{M} : \mathbf{bool}}{Env \vdash \mathbf{and}(\mathrm{M}, \mathrm{N}) : \mathbf{bool}}$$

R9 (equal)
$$\frac{Env \vdash \mathrm{N} : \mathbf{bool} \qquad Env \vdash \mathrm{M} : \mathbf{bool}}{Env \vdash \mathbf{eq}(\mathrm{M}, \mathrm{N}) : \mathbf{bool}}$$

Type Rules

- Typing of expressions

R10 (var)

$$\frac{Env \vdash \mathrm{E} : \mathcal{T}}{Env \vdash \textbf{new}\ \mathrm{E} : \textbf{Ref}(\mathcal{T})}$$

R11 (deref)

$$\frac{Env \vdash \mathrm{E} : \textbf{Ref}(\mathcal{T})}{Env \vdash \textbf{!}\,\mathrm{E} : \mathcal{T}}$$

Type Rules

- Typing of expressions

$$\frac{Env \vdash \mathrm{N}:\mathbf{Ref}(\mathcal{T}) \quad Env \vdash \mathrm{E}:\mathcal{T}}{Env \vdash \mathrm{N} := \mathrm{E} : \mathcal{T}} \quad \textbf{R13 (assign)}$$

$$\frac{Env \vdash \mathrm{E}:\mathbf{bool} \quad Env \vdash \mathrm{C} : \mathcal{T} \quad Env \vdash \mathrm{C}' : \mathcal{T}}{Env \vdash \textbf{if } \mathrm{E} \textbf{ then } \mathrm{C} \textbf{ else } \mathrm{C}' \textbf{ end} : \mathcal{T}} \quad \textbf{R14 (if)}$$

$$\frac{Env \vdash \mathrm{E}:\mathbf{bool} \quad Env \vdash \mathrm{C} : \mathcal{T}}{Env \vdash \textbf{while } \mathrm{E} \textbf{ do } \mathrm{C} \textbf{ end} : \mathbf{bool}} \quad \textbf{R15 (while)}$$

Typing Algorithm

```
interface ASTNode {
…
IType typecheck(Env<IType> et)
{

}
}

// et assigns Types to Identifiers
```

Typing Algorithm

```
class ASTAdd {
ASTNode lhs, rhs;
IType typecheck(Env<IType> et)
{
  IType lhst = lhs.typecheck(et);
  IType rhst = rhs.typecheck(et);
  if (lhst.equals(TIntSingleton)
      rhst.equals(TIntSingleton)
     return TIntSingleton;
  else throw new TypingError("Illegal type for +");
}
}
```

- **Additional Reading**

https://en.wikipedia.org/wiki/Type_system

https://en.wikipedia.org/wiki/Type_rule

Wikipedia reasonable reading list

https://dl.acm.org/citation.cfm?doid=234313.234418

Type systems, Luca Cardelli
Systems Research Center, Digital Equipment Corporation