

Interpretação e Compilação de Linguagens de Programação

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

2017-2018

João Costa Seco (joao.seco@di.fct.unl.pt)

Disclaimer: This set of slides is the base of the lecture presentation in the Interpretation and Compilation Course at the Integrated Master Program in Computer Science and Informatics. Their source was written in Portuguese, and they are in continuous evolution towards an english version, hence, some of the older materials are still in Portuguese, and others are already in English.

Interpretação e Compilação de Linguagens de Programação

Interpretação e Compilação de Linguagens de Programação

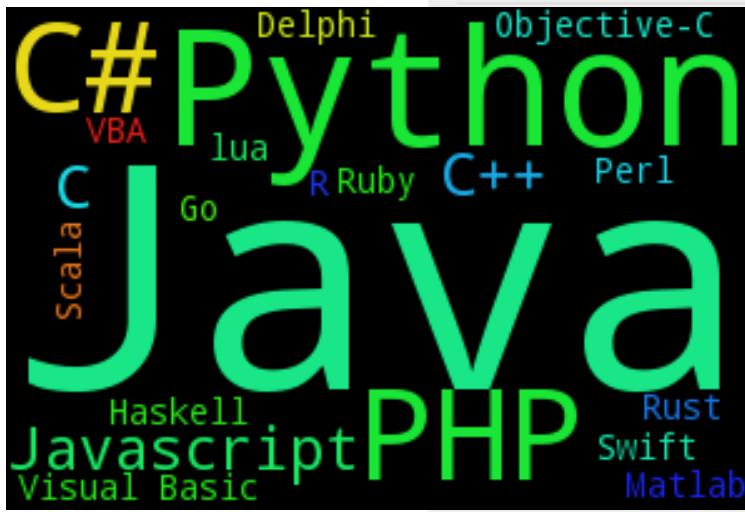
Lecture 01

Introduction and Overview

PYPL Index

PopularitY of Programming Language

- Object Oriented
- Dynamically typed
- Statically typed
- Imperative
- Functional
- Embedded Devices
- Efficient
- Dedicated Abstractions



| Rank | Change | Language | Share | Trend |
|------|--------|--------------|--------|--------|
| 1 | | Java | 23.6 % | -0.6 % |
| 2 | ↑ | Python | 13.3 % | +2.4 % |
| 3 | ↓ | PHP | 10.0 % | -0.8 % |
| 4 | | C# | 8.6 % | -0.3 % |
| 5 | ↑↑ | Javascript | 7.6 % | +0.6 % |
| 6 | ↓ | C++ | 7.0 % | -0.6 % |
| 7 | ↓ | C | 6.8 % | -0.7 % |
| 8 | | Objective-C | 4.5 % | -0.7 % |
| 9 | ↑↑ | R | 3.3 % | +0.7 % |
| 10 | | Swift | 3.1 % | +0.4 % |
| 11 | ↓↓ | Matlab | 2.8 % | +0.2 % |
| | | Ruby | 2.2 % | -0.3 % |
| | | VBA | 1.6 % | +0.0 % |
| | | Visual Basic | 1.5 % | -0.5 % |
| | | Scala | 1.2 % | +0.3 % |
| | | Perl | 1.0 % | -0.2 % |
| | | lua | 0.6 % | +0.1 % |

Java

- Class based, object-oriented
- Statically typed
- Concurrent?
- Virtual machine (Interpreter, compiler, JIT)
- Reflective
- Large Library of classes (java.lang, java.util, ...)
- Java 8 features: Lambda-Expressions, method references, default methods, type inference, repeated and type annotations.

<http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>



Python



- Class-based, object-oriented
- Dynamically typed
- Emphasises code readability
- Large Library of classes
- Open-source reference interpreter...
- How does it work? It's an encoding to the base model.

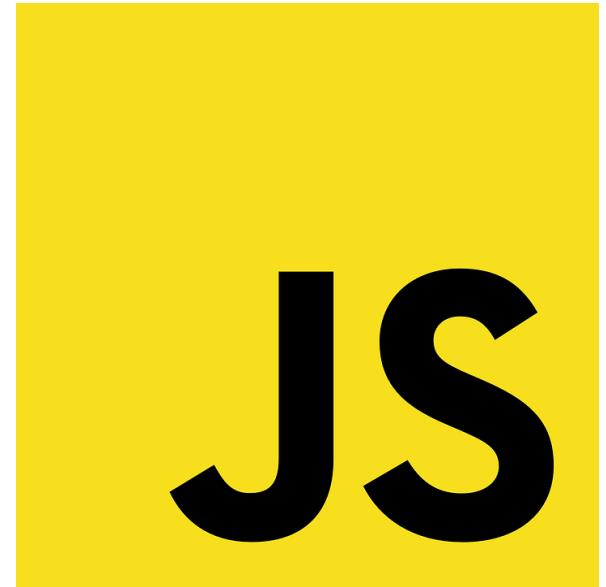
```
class ColoredSquare(Square):
    def __init__(self, side, color):
        super().__init__(side)
        self.color = color

    def setColor(self, color):
        self.color = color

    def getColor(self):
        return self.color
```

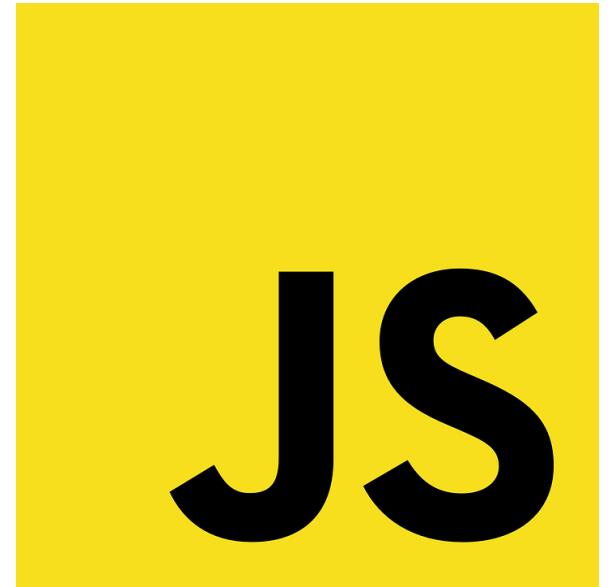
Javascript

- Functional and Object-Oriented
- From LISP and SELF (prototype based)
- Single threaded
- Dynamically typed
- The base for all web applications, languages, and frameworks
- Client and server side language



JavaScript Frameworks

- Languages, libraries, and frameworks aim at improving abstraction safety
- Language techniques, encodings, conventions, and tools.
- Node is Chrome's V8 engine (asynchronous IO) + large library
- Just-in-time compilation to native code.
- Static typing in TypeScript (MS), Dart (Google), or FlowType (FB)
- Examples: Angular.js, JQuery, React, Socket, Polymer, Node.js, Meteor, D3.js, Ember, Aurelia, Knockout, Keystone, Backbone, ...



<https://colorlib.com/wp/javascript-frameworks/>

OCaml



- Dialect of ML: “LISP with types”
- Milner, Cardelli (**DL 2016**), Huet, Cousineau, Leroy
- (pure?) Functional language =? Higher-order functions
- Object-oriented features
- Immutable values (lists), Sum types, pattern matching
- Efficient compilation of recursive calls

ocaml.org

Haskell



- Pure Functional language = No side effects
- Lazy evaluation for more efficient execution
- Side effects are isolated in a mathematical structures
- Very powerful type system
- Automatic code generation from type specification

haskell.org

- 
- # THE C PROGRAMMING LANGUAGE
- Dennis Ritchie, 1969-73
 - Low-level imperative and procedural
 - With no function nesting, local or global variables.
 - Direct translation to assembly language (almost, 1pass)
 - (almost) untyped, unsafe...

Objective-C

- NeXT, NeXTSTEP 1984
- Reflective, class-based, object-oriented
- CoCocoa Libraries, UI Guidelines
- Starting base of iOS development



Swift



- Reflective, class-based, protocol-oriented
- “multi-paradigm”
- Statically (strongly) typed
- No null values: optional values
- Apple: "Objective-C without the C"
- Current base of iOS development, open-source and available for the CLR, JVM, and Android

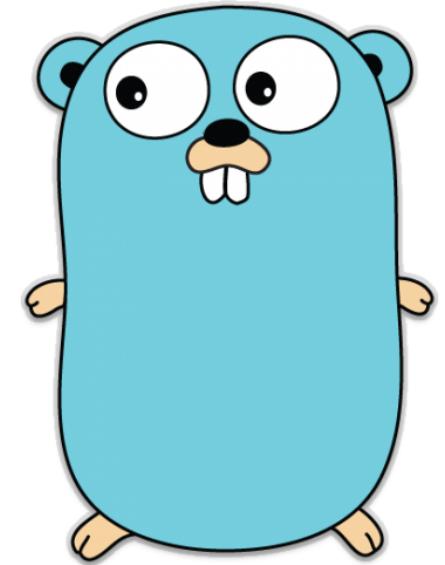
Erlang



- Concurrent, functional, immutable data, pattern matching, dynamic typing, list comprehensions.
- Its runtime system is distributed, fault-tolerant, “real-time”.
- Roots (syntax) in Prolog, but with functional aspects.
- Developed by Ericsson
www.erlang.org

```
-module(count_to_ten).  
-export([count_to_ten/0]).  
  
count_to_ten() -> do_count(0).  
  
do_count(10) -> 10;  
do_count(N) -> do_count(N + 1).
```

Go



- Google 2009
- imperative & concurrent (CSP)
- compiled & statically typed
- structural typing

```
func main() {
    t := make(chan bool)
    go timeout(t)

    ch := make(chan string)
    go readword(ch)

    select {
    case word := <-ch:
        fmt.Println("Received", word)
    case <-t:
        fmt.Println("Timeout.")
    }
}
```

golang.org

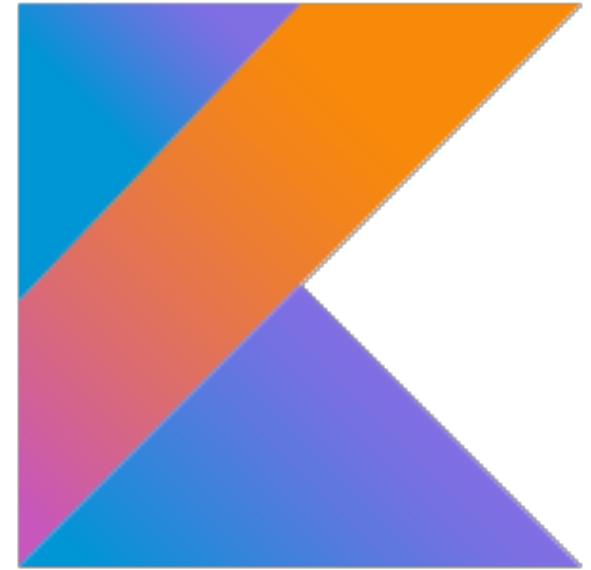
Rust



- Mozilla 2010
- “multi-paradigm”, concurrent, functional, and imperative
- Compiler in ocaml, and self-hosting compiler in Rust
- “Rust 1.0, the first stable release, on May 15, 2015”
- Guest Talk by NOVASTudentTalks, Sep 2016

www.rust-lang.org

Kotlin



- By JetBrains, 2011
- JVM based (Android), to compile as quickly as Java
- Statically typed, with type inference.
- nullable and non-nullable types

kotlinlang.org

C

```

void
qsort(void *a, size_t n, size_t es, int (*cmp)(const void *, const void *))
{
    char *pa, *pb, *pc, *pd, *pl, *pm, *pn;
    int d, swaptypes, swap_cnt;
    int r;

loop: SWAPINIT(a, es);
    swap_cnt = 0;
    if (n < 7) {
        for (pm = (char *)a + es; pm < (char *)a + n * es; pm += es)
            for (pl = pm; pl > (char *)a && cmp(pl - es, pl) > 0;
                 pl -= es)
                swap(pl, pl - es);
        return;
    }
    pm = (char *)a + (n / 2) * es;
    if (n > 7) {
        pl = a;
        pn = (char *)a + (n - 1) * es;
        if (n > 40) {
            d = (n / 8) * es;
            pl = med3(pl, pl + d, pl + 2 * d, cmp);
            pm = med3(pm - d, pm, pm + d, cmp);
            pn = med3(pn - 2 * d, pn - d, pn, cmp);
        }
        pm = med3(pl, pm, pn, cmp);
    }
    swap(a, pm);
    pa = pb = (char *)a + es;

    pc = pd = (char *)a + (n - 1) * es;
    for (;;) {
        while (pb <= pc && (r = cmp(pb, a)) <= 0) {
            if (r == 0) {
                swap_cnt = 1;
                swap(pa, pb);
                pa += es;
            }
            pb += es;
        }
        while (pb <= pc && (r = cmp(pc, a)) >= 0) {
            if (r == 0) {
                swap_cnt = 1;
                swap(pc, pd);
                pd -= es;
            }
            pc -= es;
        }
        if (pb > pc)
            break;
        swap(pb, pc);
        swap_cnt = 1;
        pb += es;
        pc -= es;
    }
    if (swap_cnt == 0) { /* Switch to insertion sort */
        for (pm = (char *)a + es; pm < (char *)a + n * es; pm += es)
            for (pl = pm; pl > (char *)a && cmp(pl - es, pl) > 0;
                 pl -= es)
                swap(pl, pl - es);
        return;
    }

    pn = (char *)a + n * es;
    r = min(pa - (char *)a, pb - pa);
    vecswap(a, pb - r, r);
    r = min((size_t)(pd - pc), pn - pd - es);
    vecswap(pb, pr - r, r);
    if ((size_t)(r = pb - pa) > es)
        qsort(a, r / es, es, cmp);
    if ((size_t)(r = pd - pc) > es) {
        /* Iterate rather than recurse to save stack space */
        a = pn - r;
        n = r / es;
        goto loop;
    }
    qsort(pn - r, r / es, es, cmp);/*
}

```

Quicksort

Haskell

```

quicksort [] = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
where
    lesser = filter (< p) xs
    greater = filter (>= p) xs

```

Different languages provide different abstraction, safety, and access to hardware characteristics.

<http://opensource.apple.com//source/xnu/xnu-1456.1.26/bsd/kern/qsort.c>

Top IDE index

- Syntax highlighting
- Code Completion
- Deployment Tools
- Compilers
- ...

| Rank | Change | IDE | Share | Trend |
|------|--------|----------------|---------|--------|
| 1 | ↑ | Visual Studio | 22.8 % | -0.5 % |
| 2 | ↓ | Eclipse | 22.45 % | -6.0 % |
| 3 | | Android Studio | 10.46 % | +2.7 % |
| 4 | | Vim | 8.2 % | +0.2 % |
| 5 | | NetBeans | 5.46 % | -0.4 % |
| 6 | | Xcode | 5.34 % | -0.5 % |
| 7 | | Sublime Text | 4.42 % | +0.1 % |
| 8 | ↑ | IntelliJ | 4.22 % | +1.2 % |
| 9 | ↓ | Komodo | 3.71 % | +0.3 % |
| 10 | | Xamarin | 3.65 % | +2.1 % |
| 11 | | Emacs | 1.85 % | +0.1 % |
| 12 | ↑ | pyCharm | 1.61 % | +0.6 % |
| 13 | ↓ | PhpStorm | 1.58 % | +0.2 % |
| 14 | ↑ | Light Table | 1.06 % | +0.1 % |
| 15 | ↓ | Cloud9 | 0.88 % | -0.2 % |
| 16 | ↑ | Qt Creator | 0.4 % | +0.0 % |
| 17 | ↓ | Aptana | 0.33 % | -0.1 % |

JVM

- Stack and Object based virtual machine
- runs Java byte code compiled from Java, Scala, Groovy, Kotlin, Clojure,...
- Provides flexible class loading and reflection
- Provides byte code verification and sandbox



<https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>

CLR



- Stack and Object based virtual machine
- runs CIL byte code compiled from C#, F#, ...
- Provides efficient base type generics
- ...

[https://msdn.microsoft.com/en-us/library/8bs2ecf4\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx)

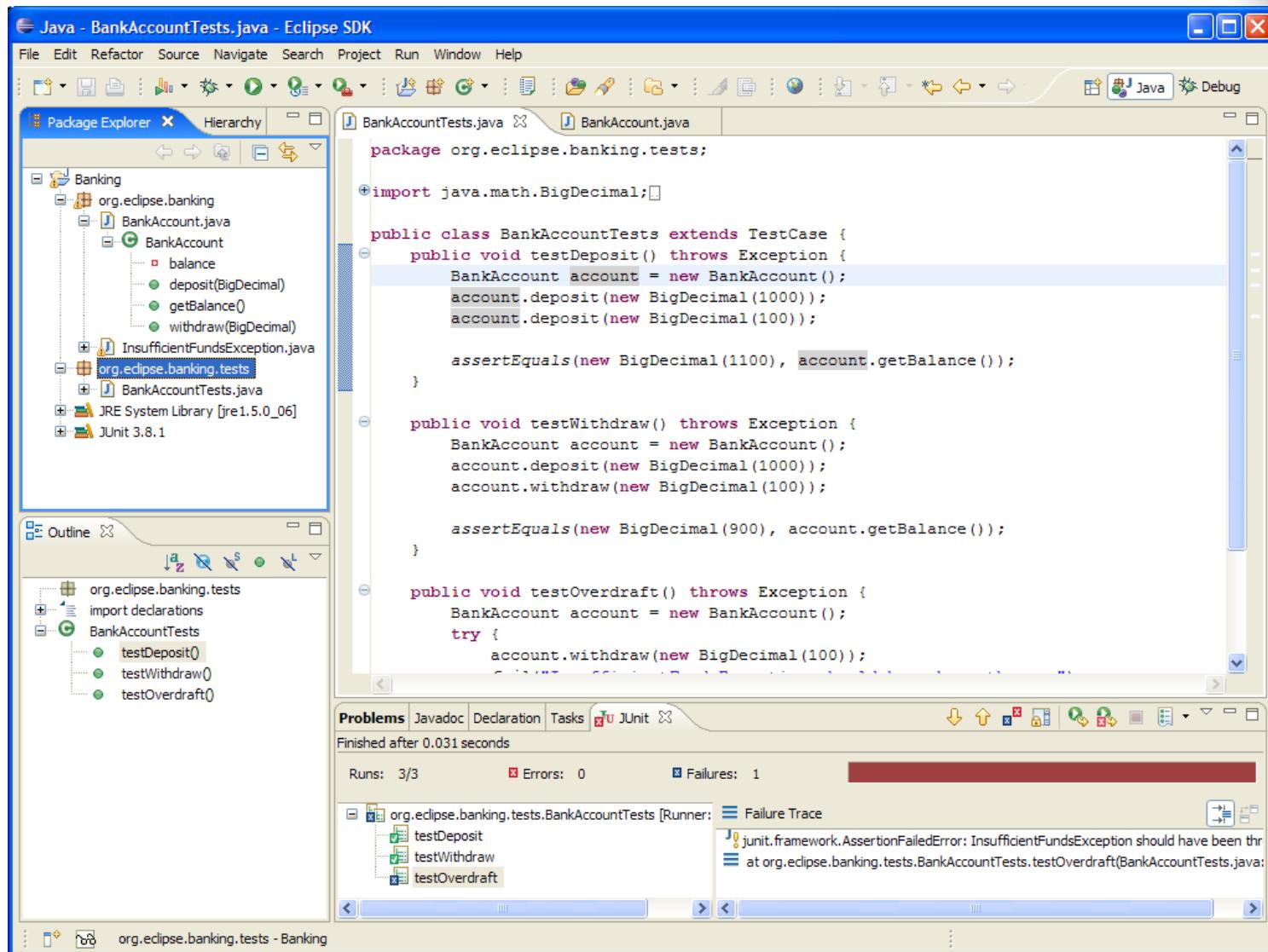
LLVM



- Low-level virtual machine, but really a compiler infrastructure for designing compiler tools
- Designed to easily define compiler optimisations.
- Provides an intermediate representation and tools.
- University of Illinois, open-source, Apple Xcode.
- clang replaces gcc entirely with LLVM IL.

<http://llvm.org>

Eclipse



<http://www.eclipse.org/>

LightTable



- Code editor on steroids
- Features real-time feedback, instant execution, easy debugging, and access to documentation.
- Targets: Clojure, Python, Javascript

<http://lighttable.com/>

Swift Playgrounds (and Xcode)



Balloons — Balloons.playground — Edited

func didMoveToView(scene : SKScene, delegate : SKPhysicsContactDelegate) {

```
// ===== Blimp Control =====
yOffsetForTime = { i in
    return 80 * sin(i / 10.0)
}

// ===== Scene Configuration =====
// Set up balloon lighting and per-pixel collisions.
balloonConfigurator = { b in
    b.physicsBody.categoryBitMask = CONTACT_CATEGORY
    b.physicsBody.fieldBitMask = WIND_FIELD_CATEGORY
    b.lightingBitMask = BALLOON_LIGHTING_CATEGORY
}

// Load images for balloon explosion.
balloonPop = (1...4).map {
    SKTexture(imageNamed: "explode_0\($0)")
}

// Install turbulent field forces.
var turbulence = SKFieldNode.noiseFieldWithSmoothness(0.7, animationSpeed:0.8)
turbulence.categoryBitMask = WIND_FIELD_CATEGORY
turbulence.strength = 0.21
scene.addChild(turbulence)

cannonStrength = 210.0

// ===== Scene Initialization =====
// Do the rest of the setup and start the scene.
setupHero(scene, delegate)
setupFan(scene, delegate)
setupCannons(scene, delegate)
}

func handleContact(bodyA : SKSpriteNode, bodyB : SKSpriteNode) {
    if (bodyA == hero) {
        bodyB.normalTexture = nil
        bodyB.runAction(removeBalloonAction)
    } else if (bodyB == hero) {
        bodyA.normalTexture = nil
        bodyA.runAction(removeBalloonAction)
    }
}
```

(Function)
(1058 times)

(Function)
(55 times)

[SKTexture, SKTexture, SKTe...
(4 times)

SKNoiseFieldNode

SKNoiseFieldNode
SKNoiseFieldNode
(GameScene (Function)) {(F...

210.0

Balloons

let y = 80 * sin(x)

<http://swiftplayground.org/post/a-quick-look-at-playground-and-xcode-6>

Swift Playgrounds (and Xcode)



The image shows a screenshot of the Swift Playgrounds app running on an iPhone. The screen is divided into two main sections: a text-based challenge description on the left and a 3D game world on the right.

Challenge: Use the AND, OR, and NOT operators to navigate Byte through the world.

Each of these operators influences the way your conditional code runs:

- The **NOT operator** (!) inverts a Boolean value, saying, "if NOT this condition, do this".
- The **AND operator** (&&) combines two conditions and runs the code only if both are true.
- The **OR operator** (||) combines two conditions and runs the code if at least one is true.

Solve the challenge by choosing the operators that will work best so that Byte collects all the gems and toggles open the switches.

```
for i in 1...6 {  
    moveForward()  
    if isOnClosedSwitch && isBlocked {  
        toggleSwitch()  
        turnLeft()  
        moveForward()  
    }  
}
```

At the bottom of the screen, there are several buttons: a back arrow, a forward arrow, a 'Run My Code' button with a play icon, a 'Hint' button with a lightbulb icon, and a menu icon.

The 3D game world on the right depicts a desert landscape with orange rock platforms, green cacti, and red gems. A character named Byte is standing on a platform. The background shows a blue sky with floating platforms and a distant planet.

<http://swiftplayground.org/post/a-quick-look-at-playground-and-xcode-6>

“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, prefácio de *Essentials of Programming Languages* de Friedman et al.)

Interpretação e Compilação de Linguagens de Programação

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

2017-2018

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 02

Syntax and Semantics

Course Objectives: To Know

- What **techniques** are used in the implementation of programming languages?
- What are the base **concepts** involved in the definition of programming languages?
- How to describe, analyse and justify the **features** of the different programming languages from the decomposition in the base concepts.
- How to design interpreters and compilers for programming languages?
- How to **predict the behaviour** of programs in a precise manner?
- How to express the properties of programs like type safety? what does type safety mean? What is a strongly typed languages?
- How to describe and implement **verification algorithms** for programming languages?
- How do runtime execution engines work (Java, .NET, LLVM, Node.JS) ?

Course Objectives: To Do

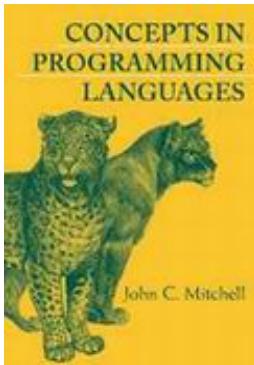
- How to **build syntactical analysers** (parsers) using generators?
- How to **represent programs as data** for other programs?
- How to **express the semantics** of a language?
- How to **build an interpreter** for a functional languages?
 - for an object-oriented language?
 - for an industrial grade target **virtual machine** (Java, .NET) ?
- How to specify a **garbage collection** algorithm?
- How to specify simple algorithms to analyse other programs?

“Roadmap”

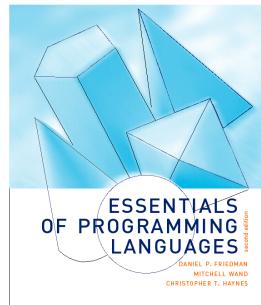
We study the fundamental concepts of programming languages; their syntactic and semantic aspects; from the perspective of type theory and pragmatic implementation perspective: We grow a language (Guy Steele) 😊

- Expressions and Values
- Binding and Scoping mechanisms
- State manipulation (memory models)
- Functional Abstraction (Functions and procedures)
- Types and Type systems
- Data abstraction
- Objects, Classes and Modules
- Compilation techniques

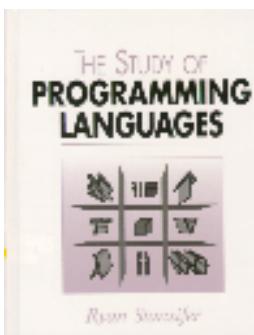
Books



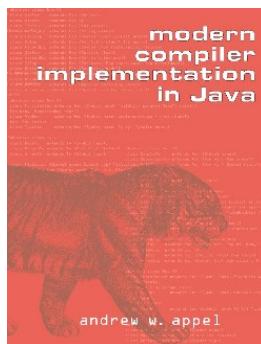
“*Concepts in Programming Languages*”,
John C. Mitchell,
Cambridge University Press.
ISBN 0 521 78098 5



“*Essentials of Programming Languages*”,
Daniel Friedman, Mitchell Wand, Christopher Haynes,
MIT Press.



“*The Study of Programming Languages*”,
Ryan Stansifer,
Prentice Hall International Edition.

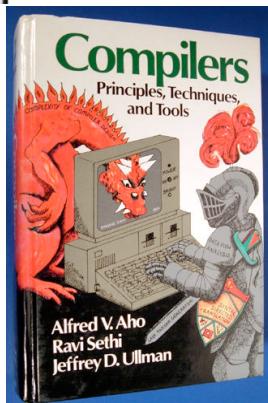


“*Modern Compiler Implementation in Java*”
Andrew W. Appel
Cambridge University Press

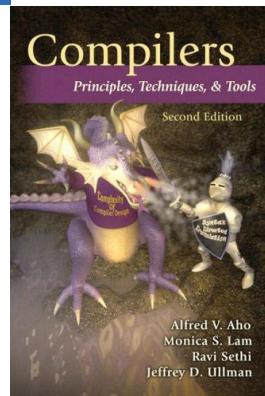
More books: “the classics”



“Principles of Compiler Design”
Alfred V. Aho, Jeffrey D. Ullman, 1977



“Compilers: Principles, Techniques, and Tools”
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 1986



“Compilers: Principles, Techniques, and Tools, Second Edition”
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, 2006

Other sources (papers & blogs)

- Great Works in Programming Languages - Collected by Benjamin C. Pierce
 - Edsger Dijkstra - 1300 EWDs
EWD215/CACM11, 1968: "Go To Statement Considered Harmful"
 - Christopher Strachey - Fundamental concepts in programming languages 1967.
 - Landin, Peter J. (1966). "The next 700 programming languages"
 - Luca Cardelli - introductory paper on Type Systems
- Blogs:
 - Existential Type (Robert Harper)
 - The Programming Languages Enthusiast
 - Lambda the Ultimate

Grading

- Theoretical component (70%):
 - 2 tests (or exam)
 - Minimum grade: 9,5 / 20
- Project component (“frequência”) (30%):
 - An interpreter and compiler program for a given example language. We build the project in a spiral, with contributions done every week from the first lab session.
 - Teams of 2 elements
 - Minimum grade: 9,5 / 20

Unit 1 - Syntax and Semantics

This unit describes general concepts about the design of programming languages and programming environments..

- Syntax and semantics of programming languages
- Abstract vs concrete syntax
- Semantic function: interpretation and compilation
- Execution environments
- Virtual machines, intermediate languages, portability

Programming Languages

- Programming languages specify (formally describe) **computational processes**.
- Languages comprise two parts that are described in a precise and non-ambiguous way:

- syntax

syntax |'sin,taks|

noun

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.

ORIGIN late 16th cent.: from French ***syntaxe***, or via late Latin from Greek ***suntaxis***, from ***sun-*** ‘*together*’ + ***tassein*** ‘*arrange*.’

- semantics

semantics |sə'mantiks|

plural noun [usu. treated as sing.]

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.

- the meaning of a word, phrase, sentence, or text : *such quibbling over semantics may seem petty stuff*

DERIVATIVES

semantician |sə'man'ti sh ən| |sə'mən'tiʃən| |'-tiʃ(ə)n| noun

semanticist |sə'mən(t)əsəst| |-tisist| noun

Linguagens de Programação

Exemplo de ambiguidade na sintaxe:

Qual o valor da expressão `f(10)` ?

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

`f(10) = ??`

Linguagens de Programação

Exemplo de ambiguidade na sintaxe:
Qual o valor da expressão `f(10)` ?

```
def f(x):
    if x > 0:
        if x < 10:
            return x
    else:
        return 10;
return 0;
```

`f(10) = ??`

Linguagens de Programação

Exemplo de ambiguidade na sintaxe:
Qual a diferença entre `@name` e `name` ???

```
class Hello
    def init(name)
        @name = name
    end

    def hello
        puts "Hello #{@name}!"
    end
end
```

Linguagens de Programação

Exemplo de ambiguidade na semântica:

Qual o valor da expressão $f(2) + g(3)$?

```
#include "stdio.h"

int a = 0;

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

int g(int y) { return y+a; }

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

int main() { printf("%d\n", sum(f(2),g(3))); }
```

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

Calling convention (cdecl)

“In the context of the C programming language, function arguments are pushed on the stack in the reverse order. In Linux, GCC sets the de facto standard for calling conventions. Since GCC version 4.5, the stack must be aligned to a 16-byte boundary when calling a function (previous versions only required a 4-byte alignment.)”

```
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

https://en.wikipedia.org/wiki/X86_calling_conventions
<https://gcc.gnu.org/onlinedocs/gccint/Stack-and-Calling.html>
<https://blogs.msdn.microsoft.com/oldnewthing/20040108-00/?p=41163>

```
caller:
; make new call frame
push    ebp
mov     ebp, esp
; push call arguments
push    3
push    2
push    1
; call subroutine 'callee'
call    callee
; remove arguments from frame
add    esp, 12
; use subroutine result
add    eax, 5
; restore old call frame
pop    ebp
; return
ret
```

Linguagens de Programação

Exemplo de ambiguidade na semântica:

Qual o valor da expressão $f(2) + g(3)$?

```
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)));  
    }  
}
```

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

Calling convention (Java?)

15.7.4. Argument Lists are Evaluated Left-to-Right

In a method or constructor invocation or class instance creation expression, argument expressions may appear within the parentheses, separated by commas. Each argument expression appears to be fully evaluated before any part of any argument expression to its right.

If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated.

Example 15.7.4-1. Evaluation Order At Method Invocation

```
class Test1 {  
    public static void main(String[] args) {  
        String s = "going, ";  
        print3(s, s, s = "gone");  
    }  
    static void print3(String a, String b, String c) {  
        System.out.println(a + b + c);  
    }  
}
```

This program produces the output:

```
going, going, gone
```

because the assignment of the string "gone" to s occurs after the first two arguments to print3 have been evaluated.

Linguagens de Programação

Exemplo de ambiguidade na semântica (será?):
Qual o valor da expressão ?

```
let rec file_read file =
  if eof file
  then []
  else
    (input_line file) :: (file_read file)

let file = open_in "a.txt" in
  let lines = file_read file in
  close_in file
```

Linguagens de Programação

Exemplo de ambiguidade na semântica:
Qual o valor da expressão `f(2)` ?

```
function g(y) {  
    a = 2*v;  
    var v;  
    return a + 1;  
}
```

```
function f(x) {  
    v = 1;  
    return g(x);  
}
```

`f(2) = ??`

Linguagens de Programação (Sintaxe)

A sintaxe caracteriza a forma como se escrevem os programas da linguagem, sem atender ao seu significado. A sintaxe é normalmente descrita por um conjunto de palavras ou “tokens”, formando um léxico, e a estrutura em que se pode organizar os tokens para formar frases bem formadas.

Linguagens de Programação (Sintaxe)

A sintaxe caracteriza a forma como se escrevem os programas da linguagem, sem atender ao seu significado. A sintaxe é normalmente descrita por um conjunto de palavras ou “tokens”, formando um léxico, e a estrutura em que se pode organizar os tokens para formar frases bem formadas.

```
Inteiro: ("0" | ["1"--"9"] ["0"--"9"]*)  
Real: ([ "0"--"9" ]) "." ([ "0"--"9" ])* ("E" ...)?  
Identificador: [a-zA-Z,_][a-zA-Z,_,0-9]*  
palavras reservadas: int, float, void, while, class,
```

The **if** statement is in the form:

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

```
switch (<expression>)  
{  
    case <label1> :  
        <statements 1>  
    case <label2> :  
        <statements 2>  
        break;  
    default :  
        <statements 3>  
}
```

Linguagens de Programação (Sintaxe)

A sintaxe caracteriza a forma como se escrevem os programas da linguagem, sem atender ao seu significado. A sintaxe é normalmente descrita por um conjunto de palavras ou “tokens”, formando um léxico, e a estrutura em que se pode organizar os tokens para formar frases bem formadas.

Exemplo de observação sobre “sintaxe”:

“Enquanto na linguagem C os blocos são delimitados por chavetas { e }, na linguagem Pascal usam-se os delimitadores begin end”

A sintaxe concreta de uma linguagem de programação pode ser descrita precisa e formalmente usando expressões regulares para os tokens e gramáticas para as frases (Teoria da Computação).

A partir gramática da linguagem constroem-se programas que reconhecem e processam os programas bem formados (Parsers)

Linguagens de Programação (Sintaxe)

```
%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]; }
```

yacc

Linguagens de Programação (Sintaxe)

antlr

```
grammar Expr;

prog:  (expr NEWLINE)* ;

expr:  expr ('*' | '/') expr
     |  expr ('+' | '-') expr
     |  INT
     |  '(' expr ')'
;

NEWLINE : [\r\n]+ ;
INT      : [0-9]+ ;
```

Linguagens de Programação (Sintaxe)

javacc

```
void Start() :  
{ }  
{  
    exp() <EOL>  
}  
  
void exp() :  
{ }  
{  
    term() [ <PLUS> exp() ]  
}  
  
void term() :  
{ }  
{  
    factor() [ <MULTIPLY> term() ]  
}  
  
void factor() :  
{ }  
{  
    <CONSTANT>  
|    <LPAR> exp() <RPAR>  
}
```

Linguagens de Programação (Sintaxe)

java parsec

```
void Start() :  
{ }  
{  
    exp() <EOL>  
}  
  
void exp() :  
{ }  
{  
    term() [ <PLUS> exp() ]  
}  
  
void term() :  
{ }  
{  
    factor() [ <MULTIPLY> term() ]  
}  
  
void factor() :  
{ }  
{  
    <CONSTANT>  
|    <LPAR> exp() <RPAR>  
}
```

Linguagens de Programação (Semântica)

Linguagens de Programação (Semântica)

A semântica descreve o significado das frases sintaticamente válidas de uma linguagem. A semântica das linguagens deve ser definida de maneira precisa.

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*

Linguagens de Programação (Semântica)

A semântica descreve o significado das frases sintaticamente válidas de uma linguagem. A semântica das linguagens deve ser definida de maneira precisa.

Exemplos de observações sobre “semântica”:

“Em C, um vector é modelado por um apontador, mas em Pascal um vector é um valor primitivo”

“A linguagem ML (OCaml) é uma linguagem imperativa, mas centrada no uso de funções”

“Na linguagem Java os argumentos são sempre passados por valor, mas em Pascal também podem ser passados por referência”

Linguagens de Programação (Semântica)

A semântica descreve o significado das frases sintaticamente válidas de uma linguagem. A semântica das linguagens deve ser definida de maneira precisa.

A semântica de uma linguagem pode ser definida precisamente por uma função **computável** I que atribui um significado a cada programa (ou fragmento de programa)

$$I : \text{Prog} \longrightarrow \text{Denot}$$

Prog = conjunto dos programas

Denot = conjunto dos significados possíveis (denotações)

A função semântica I pode ser vista como um **algoritmo** que “sabe como interpretar” todos os programas (sintacticamente correctos) de uma linguagem, determinando o seu **valor** ou **efeito**

Interpretação e compilação (software translators)

Qual a diferença entre um compilador e um interpretador?

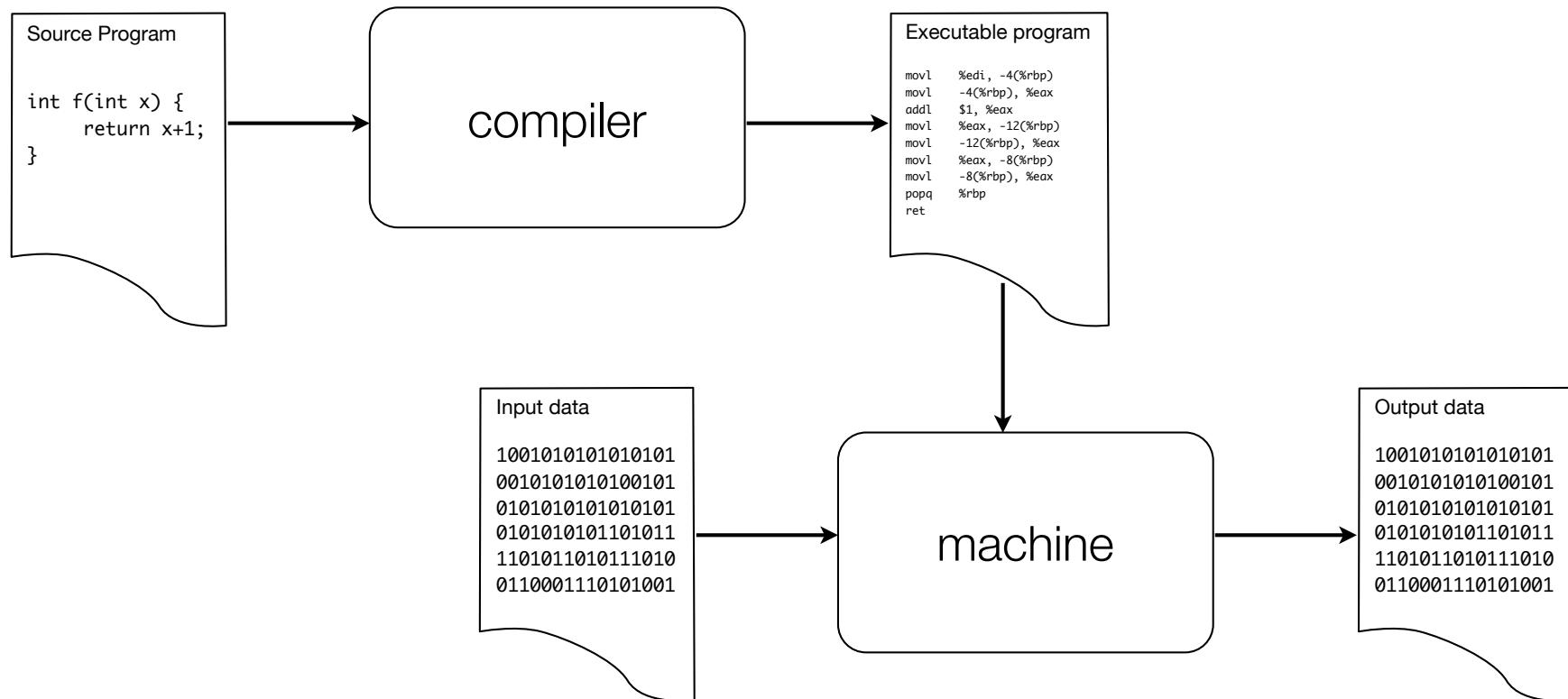
O que é uma linguagem intermédia?

O que é mais eficiente? O que é um JIT?

Qual é a arquitectura de um compilador?

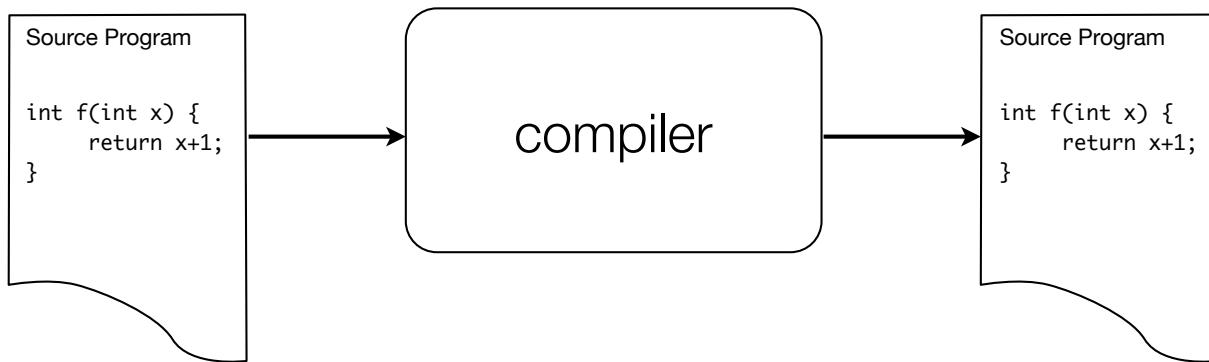
Compiladores

“Um compilador é um programa que lê um programa numa linguagem (fonte) e o traduz para um programa equivalente noutra linguagem (alvo). Um papel importante do compilador é detectar erros no programa fonte. Se a linguagem alvo for uma linguagem máquina (executável) então o programa pode ser chamado para processar dados de entrada e produzir dados de saída.” (in Aho et al)



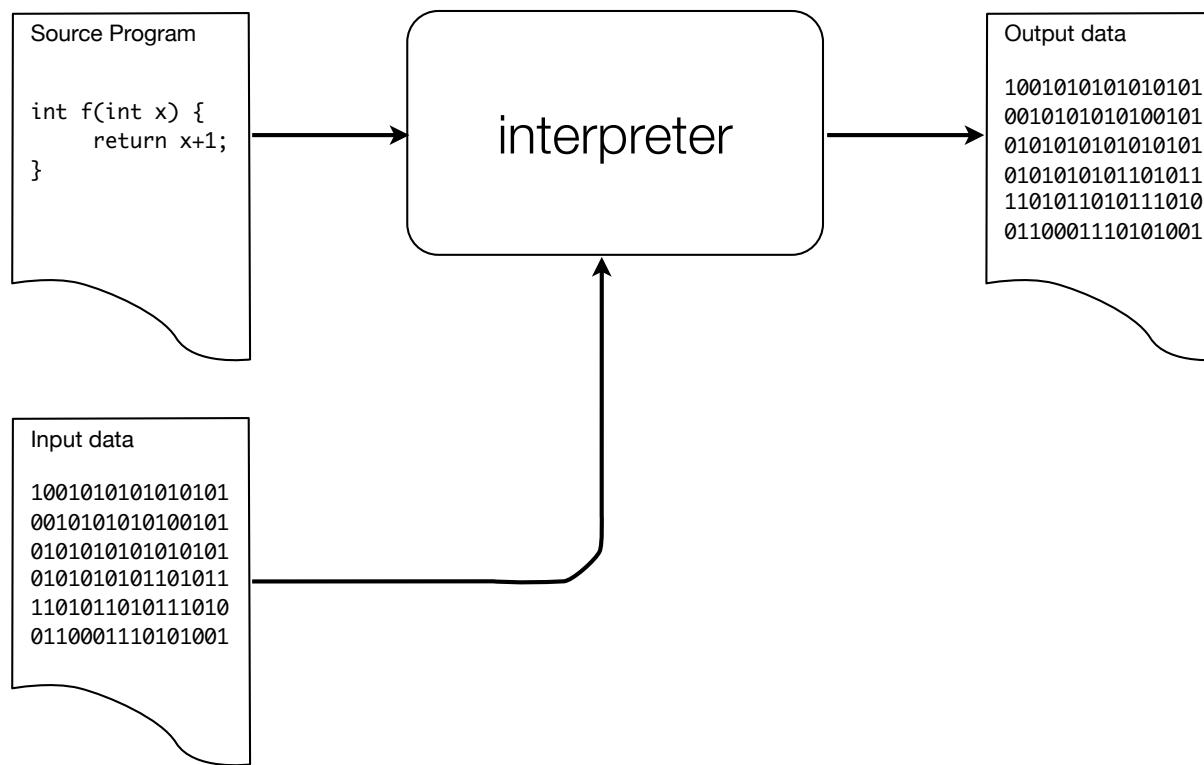
Compiladores Source2Source

“Um compilador é um programa que lê um programa numa linguagem (fonte) e o traduz para um programa equivalente noutra linguagem (alvo). Um papel importante do compilador é detectar erros no programa fonte. Se a linguagem alvo for uma linguagem máquina (executável) então o programa pode ser chamado para processar dados de entrada e produzir dados de saída.” (in Aho et al)



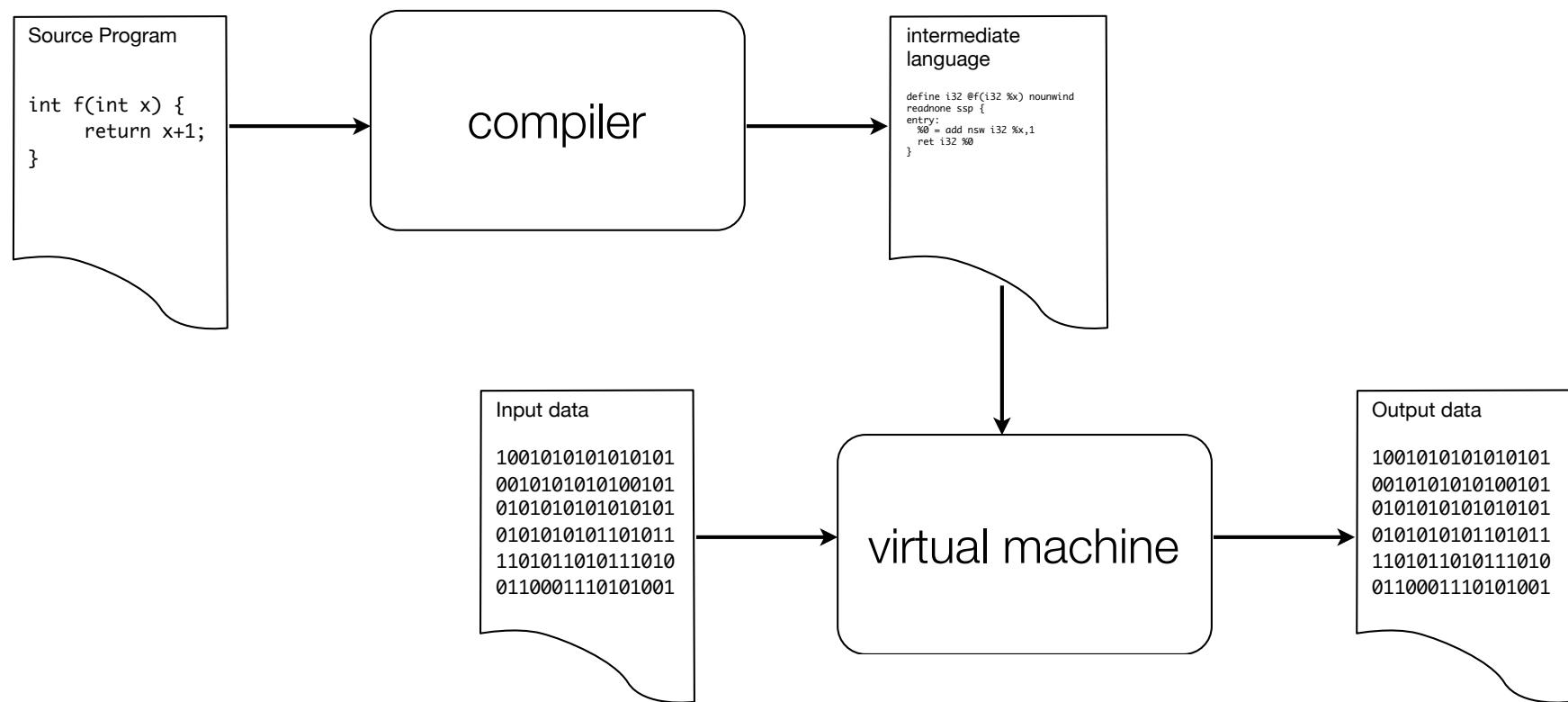
Interpretadores

Um interpretador é um programa que lê um programa numa linguagem (fonte) e produz um valor ou um efeito no seu próprio estado. Um interpretador é normalmente mais lento na produção dos dados de saída.

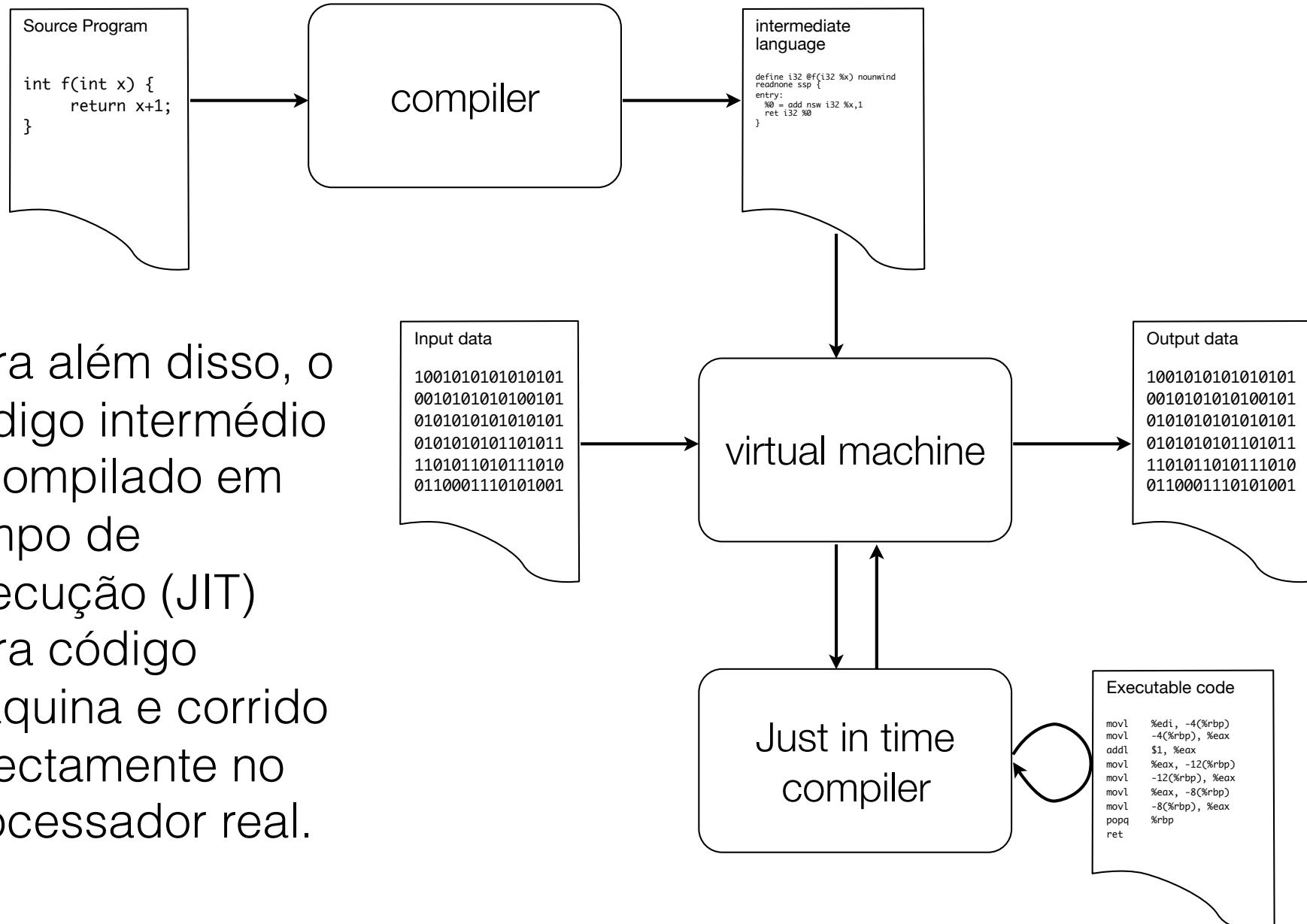


Interpretadores

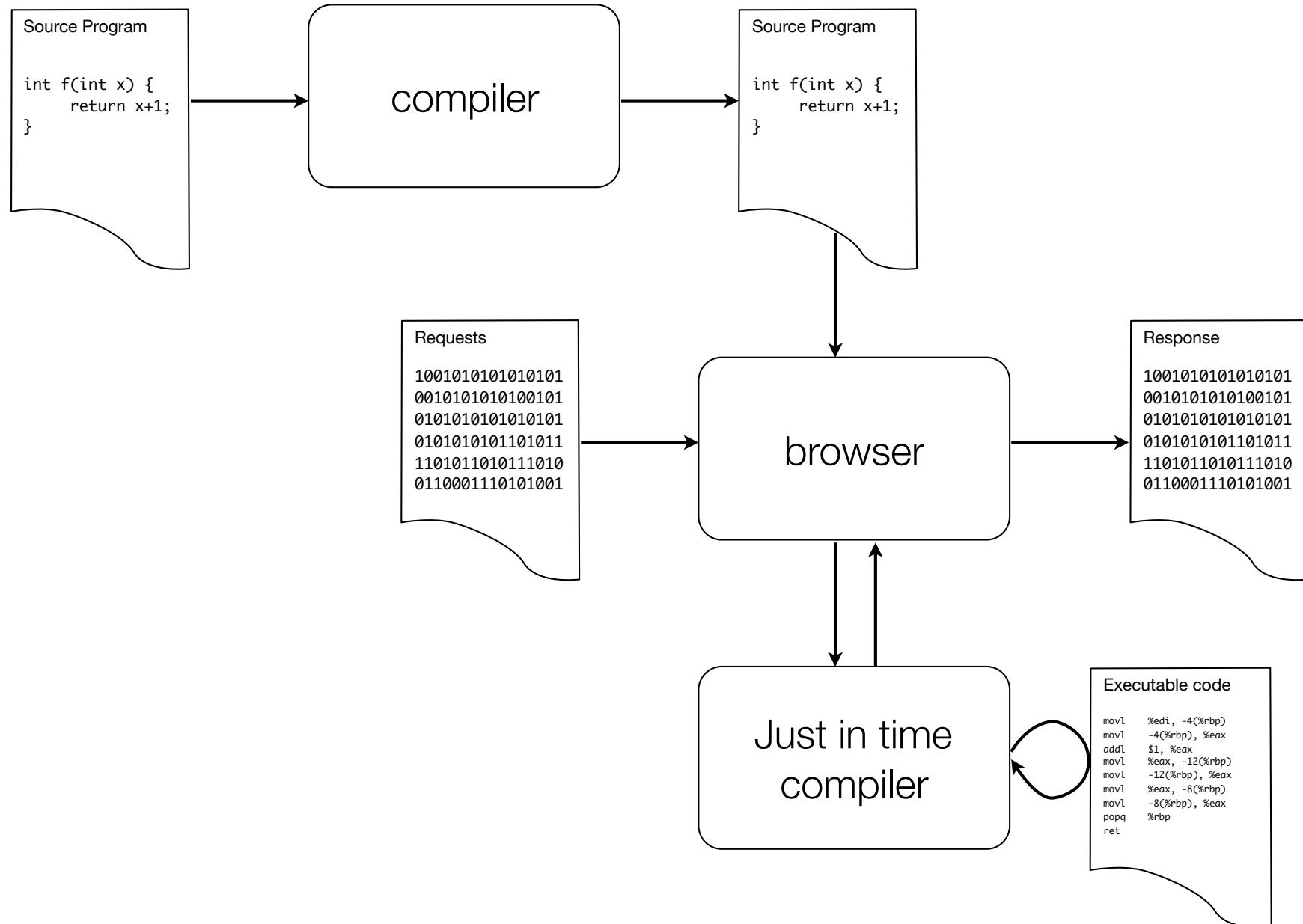
No caso da linguagem Java é utilizada uma linguagem intermédia que é interpretada por uma máquina virtual (a JVM). Para além disso, o código intermédio é compilado em tempo de execução (JIT) para código máquina e corrido directamente no processador real.



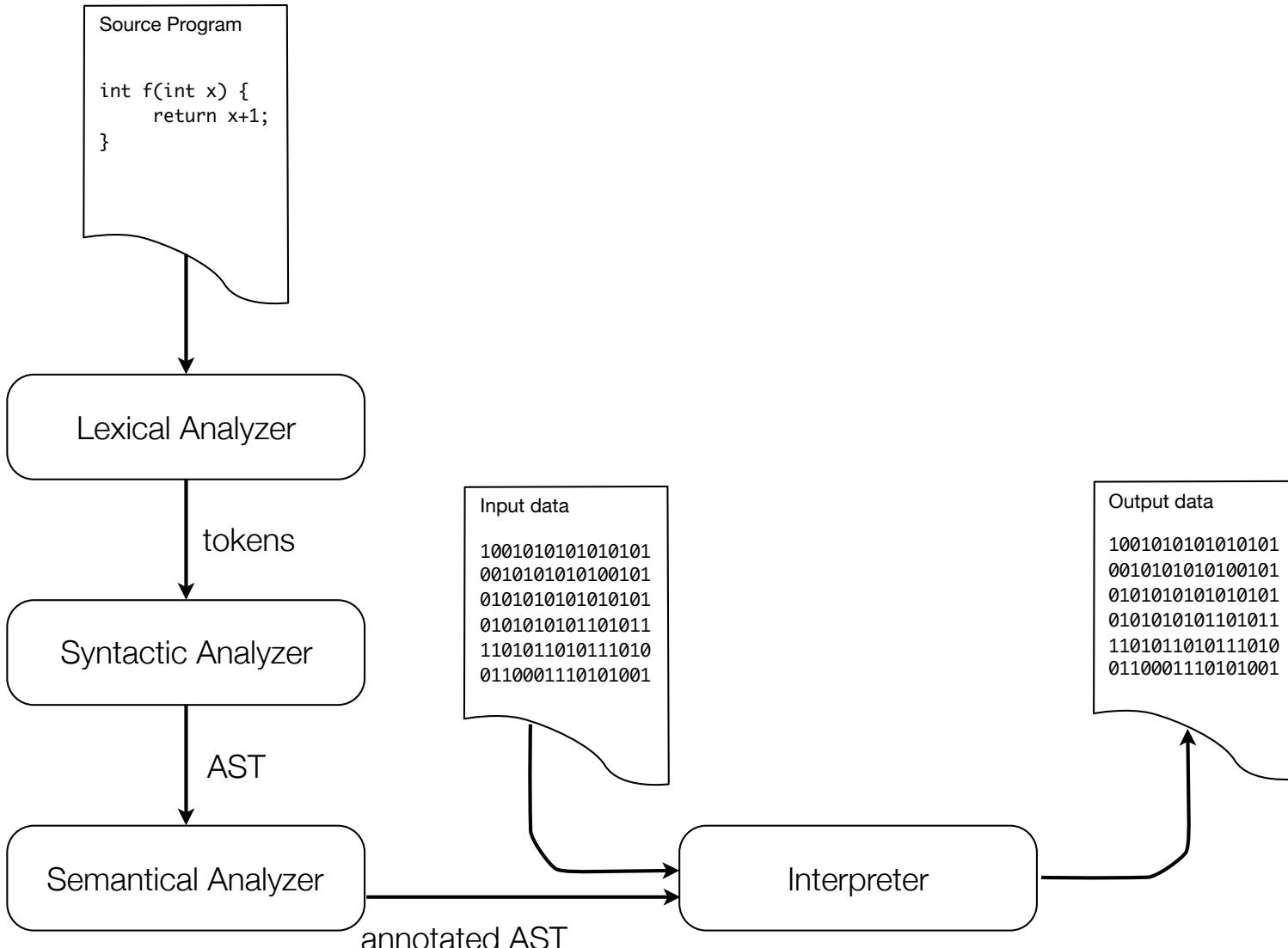
Interpretadores



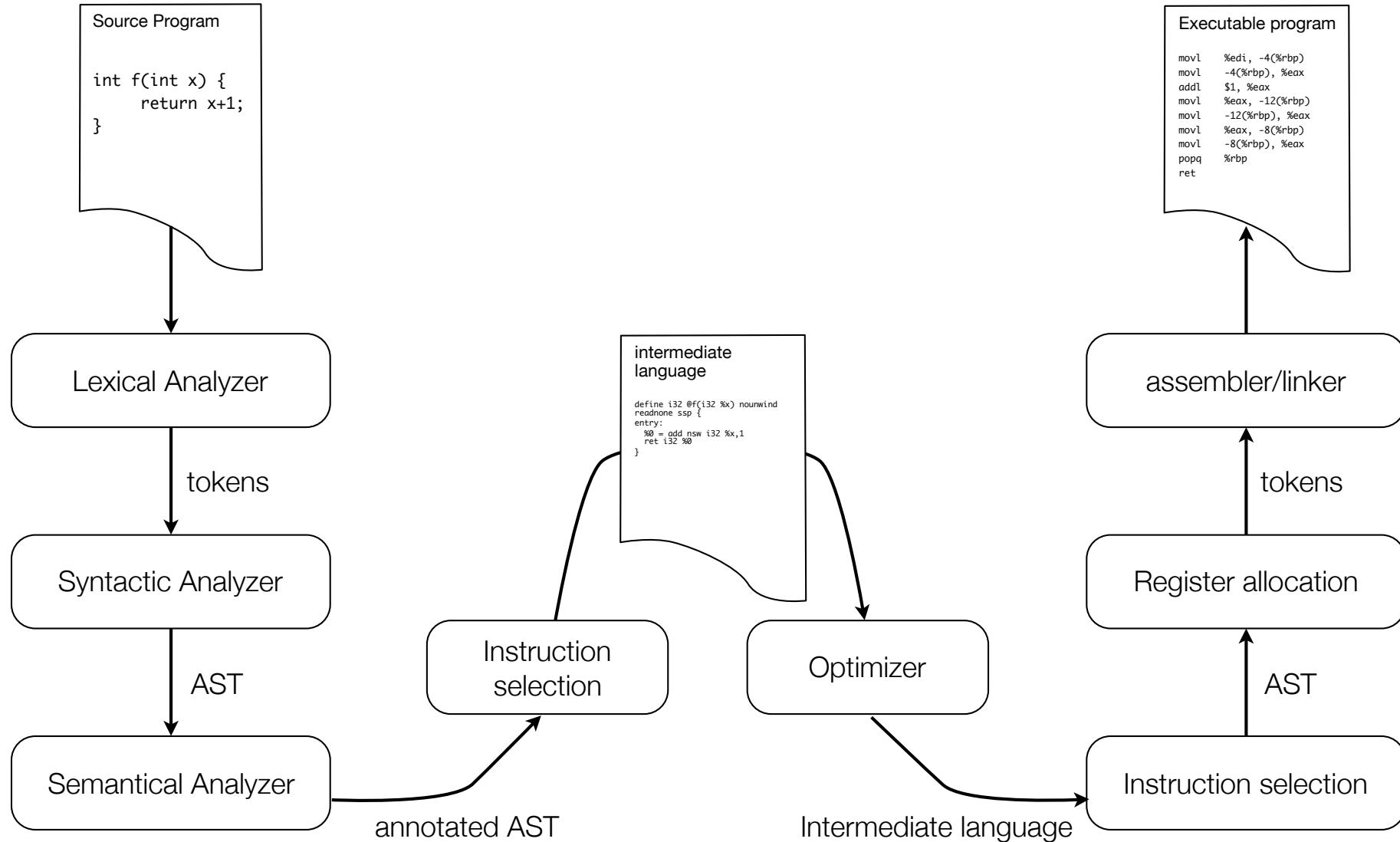
Compiladores Source2Source (Javascript)



Arquitectura de um interpretador

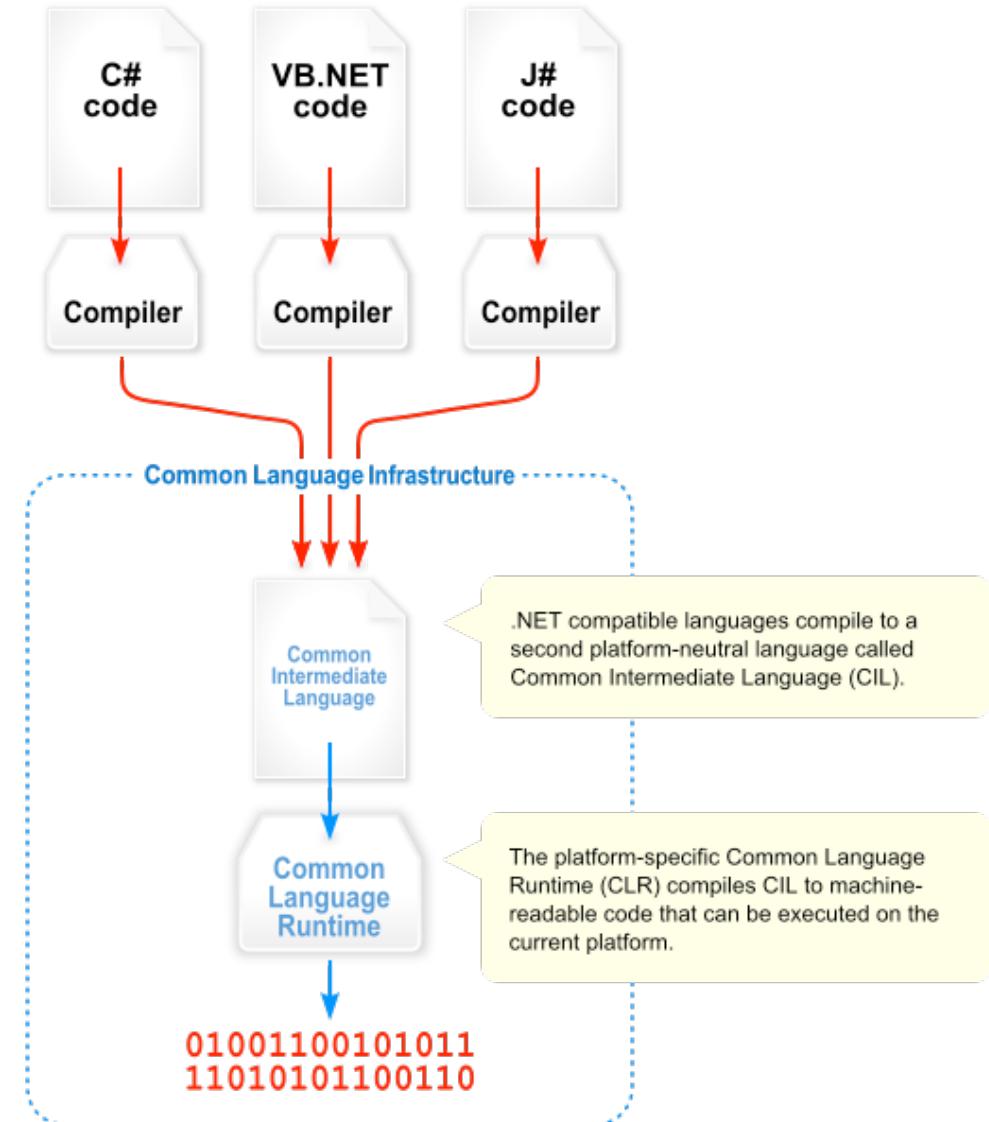


Arquitectura de um compilador

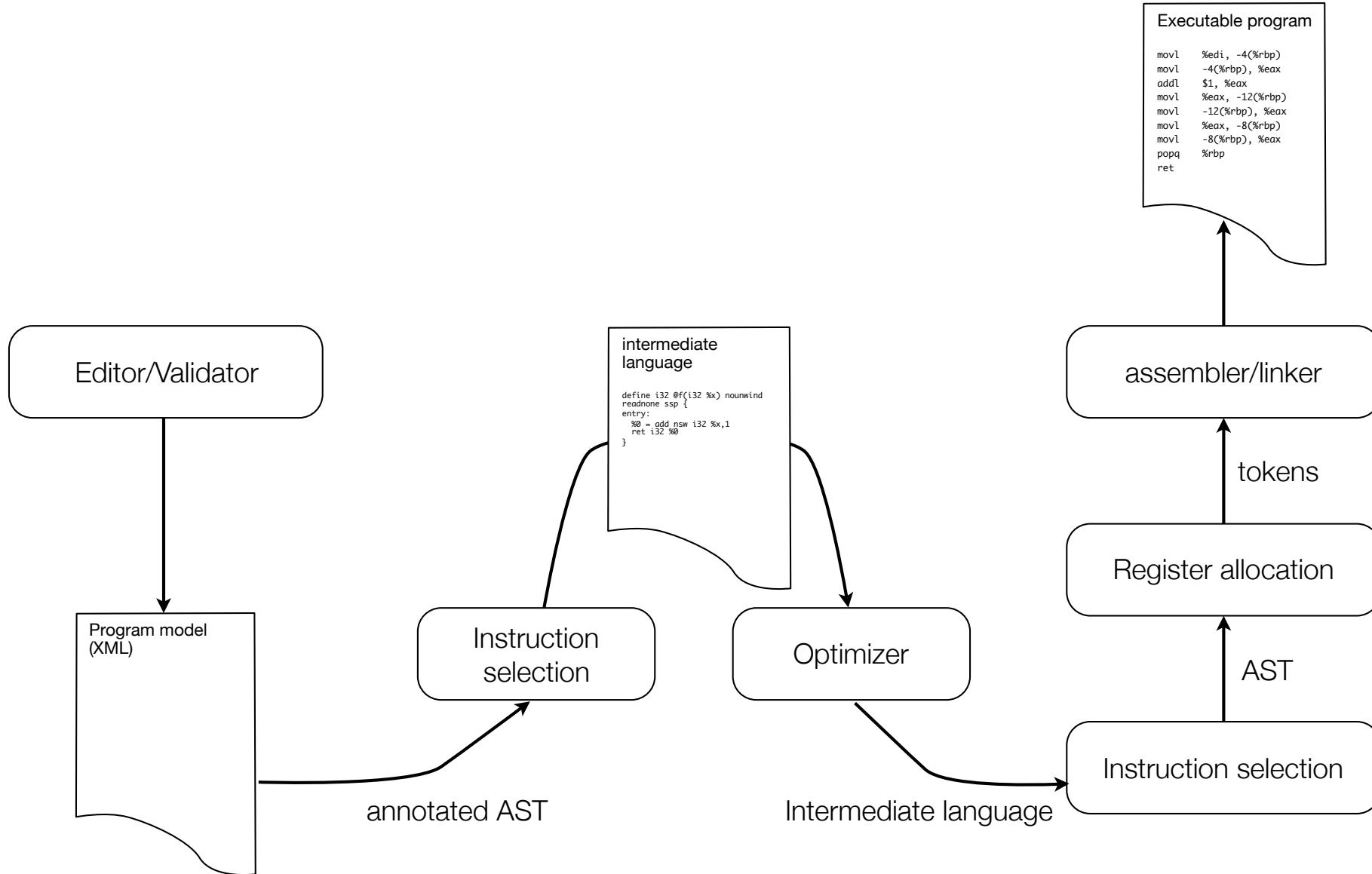


Arquitectura de um compilador

- A utilização de uma linguagem intermédia permite ainda compilar várias linguagens fonte para várias plataformas



Linguagens Visuais

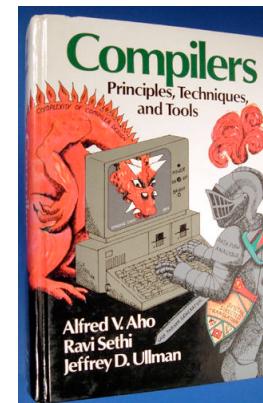
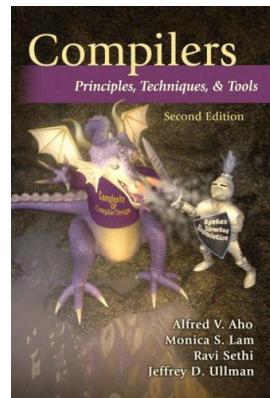
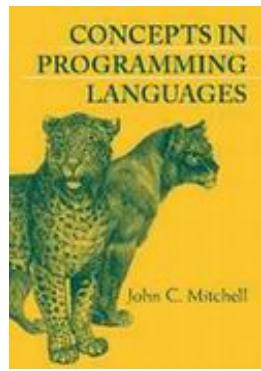


Leituras: Ideias a procurar!!

Secção 4.1: “Concepts in Programming Languages”

Capítulo 1: “Compilers: Principles, Techniques, and Tools”

<http://en.wikipedia.org/wiki/Compiler>



Sintaxe Abstracta

Os interpretadores e compiladores são algoritmos que aceitam programas sintaticamente válidos e produzem denotações (valores, efeitos, ou outros programas). Os programas são os dados de entrada para esta classe de algoritmos.

Como podemos representar os programas para que sejam processados?

Como podemos definir o processamento de programas.

- Definição inductiva de tipos de dados
- Algoritmos recursivos sobre tipos de dados inductivos
- Representação da sintaxe abstracta como tipo inductivo
- Sintaxe concreta vs sintaxe abstracta (parsing)
- Um programa como dados de input para outro programa

Sintaxe e Semântica

- **Conceito chave:** definição da semântica de uma linguagem por indução na estrutura sintática da linguagem
- **Conceito chave:** definição da semântica de uma linguagem por meio de um algoritmo interpretador
- Como definir a sintaxe de uma linguagem de programação como um tipo de dados ?
- Como definir a semântica de uma linguagem de programação como um algoritmo interpretador ?

Tipos de Dados

- Um tipo de dados pode ser representado como um conjunto de valores decidível.
- Tal conjunto de valores pode ser infinito, mas cada valor é “finito”.
- Exemplos:
 - Booleanos (**true / false**)
 - Números inteiros (..., -2, -1, 0, 1, 2, ...)
 - Listas
 - Árvores binárias
 - etc ...

Definição de tipos de dados

- Quando o conjunto de valores possíveis é finito, podemos definir os valores do tipo por enumeração :
 - Boolean $\triangleq \{ \text{true}, \text{false} \}$
 - BasicColor $\triangleq \{ \text{red}, \text{green}, \text{blue} \}$
- Como definir o conjunto dos valores de um tipo quando o número de valores possível é infinito ?
 - NaturalNumber $\triangleq ?$
 - List $\triangleq ?$
 - Tree $\triangleq ?$

Definição indutiva de dados

- O mecanismo de definição “universalmente” usado em informática é a definição **indutiva**.

O tipo *NaturalNumber* é definido pelas seguintes regras :

1. **0** é um número natural (caso base)
2. se ***n*** é um número natural, então ***succ(n)*** é também um número natural
3. não existem mais números naturais, excepto os construídos de acordo com as regras 1 e 2 acima.

Definição indutiva de dados

- O mecanismo de definição “universalmente” usado em informática é a definição **indutiva**.

O tipo *List* (de elementos de tipo *T*) é definido por:

1. *nil* é uma lista (a lista vazia)
2. se *x* é um valor de tipo *T* e *L* é uma lista, então **cons(x, L)** é também uma lista
3. não existem mais listas, excepto as construídas de acordo com as regras 1 e 2 acima.

```
type list = Nil | Cons of int * list
```

Definição indutiva de dados

- Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:

O tipo *LabeledTree* (de valores de tipo T) é definido por:

1. **empty** é uma árvore (a árvore vazia)
2. se x é um valor de tipo T, e T1 e T2 são árvores, então
node(T1, x, T2) também é uma árvore, com raiz etiquetada por x, cuja subárvore esquerda é **T1** e cuja subárvore direita é **T2**.
3. Não existem mais ... 1. e 2.

```
type tree = Empty | Node of tree * int * tree
```

Definição indutiva de dados

- Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:

O tipo *Stack* (de valores de tipo T)

1. **empty** é uma pilha (a pilha vazia)
2. se **x** é um valor de tipo T, e **S** é uma pilha, então **push(x, S)** também é uma pilha, cujo topo contém o valor x, por trás do qual está a pilha S.
3. Não existem mais ... 1. e 2.

```
type stack = Empty | Push of int * stack
```

Definição indutiva de dados

- Os ingredientes da definição indutiva de um tipo de dados são:
 1. o **nome** do tipo T
 2. um conjunto de **construtores**
- Um construtor é uma função que permite construir novos valores do tipo T a partir de valores já construídos do mesmo tipo ou de outros tipos.
- Cada construtor tem uma assinatura, que indica os tipos dos seus parâmetros.
- O tipo resultado de cada construtor do tipo T é o tipo T

Definição indutiva de dados

- Elementos da definição indutiva do tipo *lista* de valores de tipo **integer**
- o nome do tipo: **ListInt**
- um conjunto de construtores: **nil**, **cons**

nil: () → ListInt

cons: Integer × ListInt → ListInt

Definição indutiva de dados

- Elementos da definição indutiva do tipo *lista* de valores de tipo **integer**
- o nome do tipo: **ListInt**
- um conjunto de construtores: **nil**, **cons**

nil: () → ListInt

cons: Integer × ListInt → ListInt

```
/* ListInt.h */  
  
typedef struct ListInt ListInt;  
  
ListInt* nil();  
  
ListInt* cons(int elem, ListInt *tail);
```

Definição indutiva de dados

- Elementos da definição indutiva do tipo *lista* de valores de tipo **integer**
- o nome do tipo: **ListInt**
- um conjunto de construtores: **nil**, **cons**

nil: () → ListInt

cons: Integer × ListInt → ListInt

```
class ListInt {  
    static ListInt nil();  
    static ListInt cons(int elem, ListInt tail);  
};
```

Definição indutiva de dados

- Elementos da definição indutiva do tipo *árvore etiquetada* de valores de tipo **integer**
- o nome do tipo: **Treelnt**
- um conjunto de construtores: **empty**, **node**

empty: () → Treelnt

node: Treelnt × Integer × Treelnt → Treelnt

Definição indutiva de dados

- Elementos da definição indutiva do tipo *árvore etiquetada* de valores de tipo **integer**
- o nome do tipo: **Treeint**
- um conjunto de construtores: **empty**, **node**

empty: () → TreeInt

node: TreeInt × Integer × TreeInt → TreeInt

```
class TreeInt {  
    TreeInt();  
    TreeInt(TreeInt l, int elem, TreeInt r);  
};
```

Definição indutiva de dados

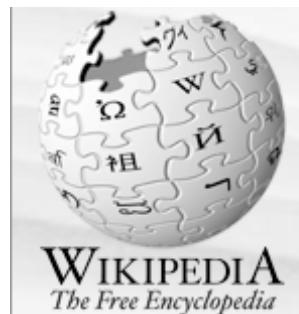
- Elementos da definição indutiva do tipo *árvore etiquetada* de valores de tipo **integer**
- o nome do tipo: **Treeint**
- um conjunto de construtores: **empty**, **node**

empty: () → TreeInt

node: TreeInt × Integer × TreeInt → TreeInt

```
/* TreeInt.h */  
  
typedef struct TreeInt TreeInt;  
  
TreeInt* empty();  
  
TreeInt* node(TreeInt *l, int elem, TreeInt *r);
```

Indução Matemática



navigation

- Main page
- Contents
- Featured content
- Current events
- Random article

search

interaction

Your [continued donations](#) keep Wikipedia running!

Try B

[article](#)

[discussion](#)

[edit this page](#)

[history](#)



Help build the future of Wikipedia and its sister projects!
[Read a letter](#) from Jimmy Wales and Michael Snow.

Mathematical induction

From Wikipedia, the free encyclopedia

Mathematical induction is a method of [mathematical proof](#) typically used to establish that a given statement is true of all [natural numbers](#). It is done by proving that the **first** statement in the infinite sequence of statements is true, and then proving that if **any one** statement in the infinite sequence of statements is true, then so is the **next** one.

The method can be extended to prove statements about more general [well-founded](#) structures, such as [trees](#); this generalization, known as [structural induction](#), is used in [mathematical logic](#) and [computer science](#). Mathematical induction in this extended sense is closely related to [recursion](#).

Mathematical induction should not be misconstrued as a form of [inductive reasoning](#), which is considered [non-rigorous](#) in mathematics (see [Problem of induction](#) for more information). In fact, mathematical induction is a form of [deductive reasoning](#) and is rigorous.



Definição indutiva de algoritmos

The Smaller-Subproblem Principle

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

- A definição de algoritmos que operam sobre tipos de dados de tipo indutivo, funcionam por **análise de casos** no construtores e decomposição do problema em problemas mais pequenos (valores mais pequenos).
- Dado um valor v qualquer de um tipo T , o algoritmo
 - **verifica-se** qual é o último construtor aplicado na construção de v (por exemplo, v é $\text{cons}(3, \text{cons}(2, \text{nil}))$);
 - **extraem-se** os componentes aos quais o construtor foi aplicado (neste caso, o inteiro 3 e a lista $\text{cons}(2, \text{nil})$);
 - **aplica-se** recursivamente aos componentes de tipo T . Assim, obtêm-se os resultados intermédios que permitem produzir o resultado final.

Definição indutiva de algoritmos

- Tipo **ListInt**
- Construtores: **nil**, **cons**
- Algoritmo $\text{length}(L)$ para calcular o **comprimento** de uma lista qualquer L :

se L é da forma **nil**

$$\text{length}(L) \triangleq 0$$

se L é da forma **cons**(x, L')

$$\text{length}(L) \triangleq 1 + \text{length}(L')$$

```
type listInt = Nil  
           | cons of int * listInt  
  
let rec length l = match l with  
  Nil -> 0  
  | Cons(x,l) -> 1+(length l)
```

Definição indutiva de algoritmos

- Tipo **ListInt**
- Construtores: **nil**, **cons**
- Algoritmo sum(L) para calcular a **soma** dos valores numa lista qualquer L:

se L é da forma **nil**

$$\text{sum}(L) \triangleq 0$$

se L é da forma **cons(x,L')**

$$\text{sum}(L) \triangleq x + \text{sum}(L')$$

```
type listInt = Nil  
           | cons of int * listInt  
  
let rec sum l = match l with  
  Nil -> 0  
  | Cons (x,l) -> x + (sum l)
```

Definição indutiva de algoritmos

- Tipo **TreeInt**
- Construtores: **empty**, **node**
- Algoritmo numnodes(T) para calcular o **número de nós** numa árvore qualquer T :

se T é da forma **empty**

$$\text{numnodes}(L) \triangleq 0$$

se T é da forma **node**(L' , x , L'')

$$\text{numnodes}(L) \triangleq 1 + \text{numnodes}(L') + \text{numnodes}(L'')$$

```
type treeInt = Empty
```

```
| Node of treeInt * int * treeInt
```

```
let rec numnodes t = match t with
```

```
    Empty -> 0
```

```
| Node(l, x, r) -> 1 + (numnodes l) + (numnodes r)
```

Definição indutiva de algoritmos

- Tipo **TreeInt**
- Construtores: **empty**, **node**
- Algoritmo $\text{depth}(T)$ para calcular a altura de uma árvore qualquer T :

se T é da forma **empty**

$$\text{depth}(T) \triangleq 0$$

se T é da forma **node(L' , x,L'')**

$$\text{depth}(T) \triangleq 1 + \max(\text{depth}(L'), \text{depth}(L''))$$

```
type treeInt = Empty
| Node of treeInt * int * treeInt

let rec depth t = match t with
  Empty -> 0
  | Node(l,x,r) -> 1 + (max (depth l) (depth r))
```

Análise inductiva de algoritmos

- Propriedades sobre os algoritmos definidos inductivamente na estrutura dos dados podem ser analisadas usando indução matemática.
- Ex: Será que uma árvore binária construída pela função insert, é uma árvore binária de pesquisa?

```
type treeInt = Empty
| Node of treeInt * int * treeInt

let rec depth t = match t with
    Empty -> 0
  | Node(l,x,r) -> 1 + (max (depth l) (depth r))

let rec insert t y = match t with
    Empty -> Node(Empty,y,Empty)
  | Node(l,x,r) -> if x <= y then Node(insert l y,x,r)
                     else Node(l,x,insert r y)
```

Example -
BlackBoard!

Definição indutiva de programas

- O conjunto de todos os **programas** de uma linguagem de programação pode ser visto como um **tipo de dados**
- É fácil **definir indutivamente o conjunto de todos os programas** de uma linguagem de programação
- A definição indutiva de linguagens melhorou o desenho das mesmas (Fortran/spaghetti code vs. Pascal/estrutura em blocos)
- Discussão:
definição indutiva de algoritmos versus definição indutiva de programas
(a que se referem as duas ocorrências do adjetivo “indutivo”?)

Definição indutiva de programas

Copyright Notice

The following manuscript

EWD 215: A Case against the GO TO Statement

was published as a letter entitled

Go-to statement considered harmful

in *Commun. ACM* 11 (1968), 3:
with permission.

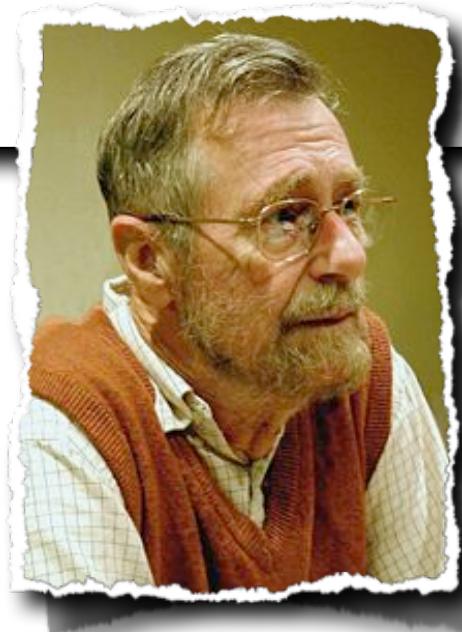
A Case against the GO TO Statement.

by Edsger W. Dijkstra

Technological University

Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the degree of use of the go to statement in the programs they produce. Later I discovered that the go to statement has such disastrous effects and did I conclude that the go to statement should be abolished from all "high level" programming languages (i.e. everything except - perhaps - p



Estruturação hierárquica de linguagens

```
10 for I=1 to 10 do
20 if I<10 goto 40
25 for J=I to 20 do
30 next I
40 next J
```



```
for i=1 to 10 do
begin
if i<10 then
begin
end
end;
```

Exemplo: A Linguagem CALC

- CALC é uma linguagem simples de expressões aritméticas.
- Cada programa da linguagem CALC é uma expressão algébrica construída com base em numerais inteiros, nos quatro operadores aritméticos básicos (+, -, ×, ÷) e nos parêntesis.
- Exemplos:

(21+32) * 42

2 / (7-2)

- A semântica pretendida para a linguagem CALC é a esperada para uma linguagem de expressões aritméticas:
a denotação de cada expressão da linguagem CALC é o seu valor.

Semântica da linguagem CALC

Em geral, a semântica de uma linguagem pode ser caracterizada por uma função **I** que atribui um significado (ou denotação) a cada programa (ou fragmento de programa) sintaticamente correcto.

- No caso da linguagem CALC:

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

CALC = conjunto dos programas válidos

Integer = conjunto dos significados (denotações)

Como representar a linguagem CALC como um tipo de dados?

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: **num**, **add**, **mul**, **div**, **sub**

num: Integer \rightarrow CALC

add: CALC \times CALC \rightarrow CALC

mul: CALC \times CALC \rightarrow CALC

div: CALC \times CALC \rightarrow CALC

sub: CALC \times CALC \rightarrow CALC

- Cada valor do tipo indutivo CALC representa uma expressão na linguagem CALC. Diz-se que o tipo indutivo CALC define a sintaxe abstracta da linguagem CALC
- A **sintaxe concreta** de uma linguagem:
 - Caracteriza a forma como as expressões e programas são efectivamente escritos em termos de sequências de caracteres, formatação, etc ...
- A **sintaxe abstracta** de uma linguagem:
 - Caracteriza a estrutura das suas expressões e programas, apresentando essa estrutura em termos de construtores abstractos.

Sintaxe abstracta vs Sintaxe concreta

- Constantes inteiras
 - em decimal: 12
 - em hexadecimal: 0x0C
 - sintaxe abstracta: num(12)
- Variáveis
 - em Pascal: A
 - em bash: %A
 - sintaxe abstracta: var("A")
- Afectação
 - em C: x = 2
 - em Pascal: x := 2
 - sintaxe abstracta: assign(var("x"),num(2))

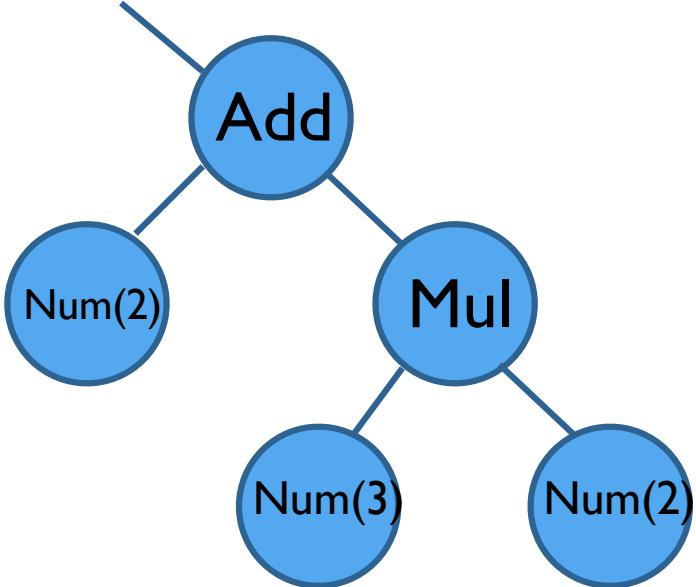
Sintaxe abstracta vs Sintaxe concreta

- Expressões algébricas
 - em C: $2 * 3 + 2$
 - em RPN: $2 \ 3 \ * \ 2 \ +$
 - em Lisp: $(+ (* 2 3) 2)$
 - sintaxe abstracta: `add(mul(num(2),num(3)),num(2))`
- Blocos
 - em C: $\{S_1 \ S_2 \ \dots \ S_n\}$
 - em Pascal: `begin S1; S2; ...; Sn end`
 - sintaxe abstracta: `block(S1,S2,...,Sn)`
- Ciclo while:
 - em C: `while (C) S`
 - em Pascal: `while C do S`
 - sintaxe abstracta: `while(C,S)`

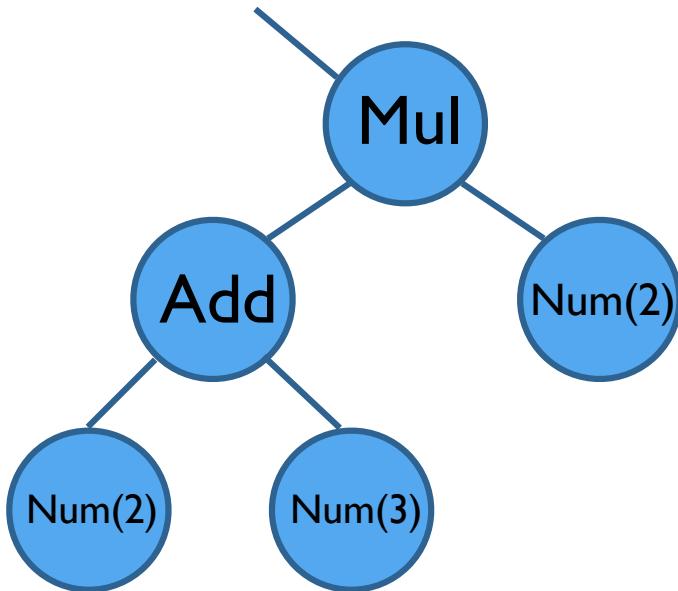
Árvore Sintáctica Abstracta

- Representa a estrutura de uma frase da linguagem em termos dos seus constructores abstractos.
- Abstract Syntax Tree (AST)

$2 + 3 * 2$



$(2 + 3) * 2$



Árvore Sintáctica Abstracta

- Representa a estrutura de uma frase da linguagem em termos dos seus constructores abstractos.
- Abstract Syntax Tree (AST)

2 + 3 * 2

(2 + 3) * 2

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

let e   = Add(Num(2),Mul(Num(3),Num(2)))
let e' = Mul(Add(Num(2),Num(3)),Num(2))
```

Semântica Operacional

Os interpretadores e compiladores são algoritmos que aceitam programas sintaticamente válidos e produzem denotações (valores, efeitos, ou outros programas). Os interpretadores têm como resultado um valor ou efeito, os compiladores têm como resultado um programa numa linguagem de uma máquina.

Como podemos definir o processamento de programas?

- Definição composicional de Semântica operacional.
- Algoritmo interpretador de CALC: implementação usando uma linguagem orientada por objectos (Java)
- Algoritmo compilador de CALC: introdução à máquina virtual CLR

Semântica de CALC

- A função semântica I pode ser definida por um algoritmo que “sabe como interpretar” todos os programas sintacticamente correctos de uma linguagem, determinando o seu valor ou efeito

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

CALC = conjunto dos programas válidos

Integer = conjunto dos significados (denotações)

Interpretadores

- Um interpretador para uma linguagem de programação é um algoritmo que atribui um valor ou efeito a cada programa legítimo da linguagem
- Os programas fornecidos como dados de entrada a um interpretador são representados por valores da sintaxe abstracta da linguagem a que pertencem
- Um algoritmo interpretador pode ser definido indutivamente na sintaxe abstracta da linguagem

Interpretadores

- Um interpretador para uma linguagem de programação é um algoritmo que atribui um valor ou efeito a cada programa legítimo da linguagem
- Os programas fornecidos como dados de entrada a um interpretador são representados por valores da sintaxe abstracta da linguagem a que pertencem
- Um algoritmo interpretador pode ser definido indutivamente na sintaxe abstracta da linguagem

Resumindo:

- Programas exprimem algoritmos que processam dados
- Programas também podem ser vistos como dados
- Um interpretador é um programa que processa programas

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: num, add, mul, div, sub

num: Integer → CALC

add: CALC × CALC → CALC

mul: CALC × CALC → CALC

div: CALC × CALC → CALC

sub: CALC × CALC → CALC

A Linguagem CALC (como tipo indutivo)

- Tipo de dados CALC com os construtores: num, add, mul, div, sub

num: Integer → CALC

add: CALC × CALC → CALC

mul: CALC × CALC → CALC

div: CALC × CALC → CALC

sub: CALC × CALC → CALC

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

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

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

| | |
|--|--|
| se E é da forma num (n): | $\text{eval}(E) = n$ |
| se E é da forma add (E' , E''): | $v1 = \text{eval}(E');$ $v2 = \text{eval}(E''');$ $\text{eval}(E) \triangleq v1 + v2$ |
| se E é da forma mul (E' , E''): | $v1 = \text{eval}(E');$ $v2 = \text{eval}(E''');$ $\text{eval}(E) \triangleq v1 * v2$ |
| se E é da forma sub (E' , E''): | $v1 = \text{eval}(E');$ $v2 = \text{eval}(E''');$ $\text{eval}(E) \triangleq v1 - v2$ |
| se E é da forma div (E' , E''): | $v1 = \text{eval}(E');$ $v2 = \text{eval}(E''');$ $\text{eval}(E) \triangleq v1 / v2$ |

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

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

| | |
|------------------------------------|---|
| $\text{eval}(\text{num}(n))$ | $\triangleq n$ |
| $\text{eval}(\text{add}(E', E''))$ | $\triangleq \text{eval}(E') + \text{eval}(E'')$ |
| $\text{eval}(\text{mul}(E', E''))$ | $\triangleq \text{eval}(E') * \text{eval}(E'')$ |
| $\text{eval}(\text{sub}(E', E''))$ | $\triangleq \text{eval}(E') - \text{eval}(E'')$ |
| $\text{eval}(\text{div}(E', E''))$ | $\triangleq \text{eval}(E') / \text{eval}(E'')$ |

[notação mais legível, usando pattern matching]

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

eval : CALC → Integer

eval(num(n)) \triangleq n

```
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)
```

Interpretador de CALC

- Algoritmo eval(E) para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

eval : CALC → Integer

- Note bem: a função de interpretação eval(-) é definida recursivamente na estrutura do seu argumento!
- Semântica composicional: o significado do todo só depende do significado das suas partes
- O mesmo não acontece nas linguagens “naturais”:
 - time **flies** like an arrow
 - fruit **flies** like a banana

Semântica Operacional Estrutural

- Sintaxe (abstracta) definida por um tipo indutivo, apresentada por um conjunto de construtores;
- Semântica definida por um algoritmo interpretador, que atribui um significado (valor) a cada expressão da linguagem, calculando-o composicionalmente a partir do significado das suas subexpressões;
- A esta técnica de definição da semântica de uma linguagem de programação chama-se:

Semântica Operacional Estrutural

Implementação em Java

- Usando uma linguagem baseada em objectos, um tipo de dados indutivo pode ser representado por uma interface (que representa o tipo indutivo), e por um conjunto de classes (em que cada classe representa um construtor do tipo indutivo)
- A interface pode declarar uma ou mais operações sobre o tipo indutivo, por exemplo:

```
public interface CALC {  
    int eval();  
}
```

Ou seja, $\text{eval}: \text{CALC} \rightarrow \text{Integer}$

Implementação em Java

- Cada classe representa um (e um só) dos construtores do tipo indutivo
- Cada classe fornece a implementação relativa ao construtor que representa, para cada operação definida sobre o tipo indutivo

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

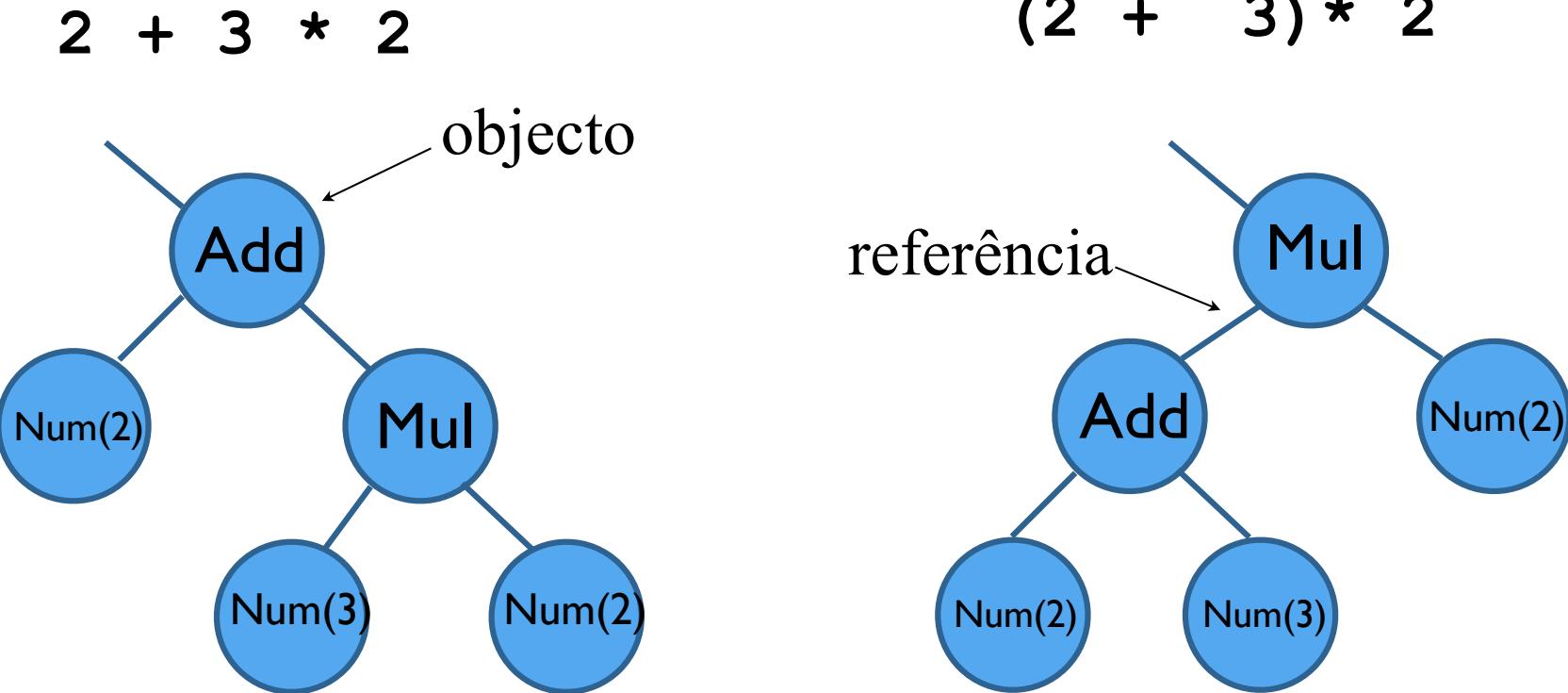
Implementação em Java

- Cada expressão é representada por uma árvore (n-ária) de objectos (uma AST);
- Cada construtor do tipo indutivo é aplicado chamando o construtor da classe respectiva:

```
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() ;
```

Árvore Sintáctica Abstracta

- Representa a estrutura de uma frase da linguagem em termos dos seus constructores abstractos.
- Abstract Syntax Tree (AST)



Implementação em Java

- No nosso exemplo, temos as classes **Num**, **Add**, **Sub**, **Mul** e **Div**, implementando os construtores da linguagem **num**, **add**, **sub**, **mul** e **div**.
- A definição das operações fica “dispersa” pelas várias classes (é mais cômodo acrescentar novos construtores a uma linguagem do que novas operações)

```
public class Add implements CALC {  
    private CALC lhs;  
    private CALC rhs;  
    Add(CALC l, CALC r) { lhs = l; rhs = r; }  
    int eval() { return lhs.eval()+rhs.eval(); }  
}
```

Interpretação e Compilação de Linguagens de Programação

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

2017-2018

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 03

Static Type Checking

Type Systems

Type systems are language based tools that perform static analysis for programs and ensure good properties. Static analysis is a form of (abstract) interpretation. The main feature of all static analyses is that they always terminate, even for programs that do not terminate when executed.

A common, light-weight form of static analysis is implemented in type checkers of interpreters, compilers and virtual machines. Type Systems ensure the absence of a class of runtime errors.

- Execution errors
- Detection of runtime errors
- Abstract interpretation
- Type Systems
- Soundness of a type system

Runtime errors

- What can go wrong in this program?

```
function gcd( x, y ) {
    while (y != 0) {
        z = x
        y = x % y
        x = z
    }
}
```

Runtime errors

- What can go wrong in this program?

```
function f(x,y) {
    return 2 + x > 0 && y;
}

console.log(f(true, 0));
```

Runtime errors

- What can go wrong in this program?

```
int *f(x,y) {
    int *a = malloc(sizeof(int));
    *a = 0;

    int *b = malloc(sizeof(int));
    *b = 2;

    bool c = malloc(sizeof(bool));
    c = a > b;

    if ( ! c ) {
        a = *a + 1;
        c = 1 < *a;
    }
    return *c;
}
```

Runtime errors

- What can go wrong in this program?

```
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 ]
```

It is impossible to determine the result beforehand for all values and avoid execution errors. The domain of values are infinite sets...

We may interpret programs by aggregating infinite sets of values in a single representative, a “type”

Base operations in different domains

- 1, 2, 3, 4, ... are represented by **int**
- true, false are represented by **bool**
- in the domain of values
 - $1 + 2 = 3$:)
 - $\text{true} \&\& \text{false} = \text{false}$
- in the domain of types
 - $\text{int} * \text{int} = \text{int}$
 - $\text{int} == \text{int} = \text{bool}$
 - $\text{bool} \&\& \text{int} == \text{int} = \text{bool}$

Semantic Functions

- **eval** (partially) computes the value of any expression:

eval : BCALC → INTEGER ∪ {true, false}

- **typecheck** computes the type of any expression:

typecheck : BCALC → {int, bool, none}

- **compile** (partially) computes the bytecode image of any expression:

compile : BCALC → Instructions

Type checking

- **typecheck** computes the type of any expression:

typecheck : BCALC \rightarrow {int, bool, none}

- The **typecheck** function can be seen as an interpreter that evaluates a program a different, more abstract, semantics. Its values are “types”.
- The semantic of operators changes (in a synchronised way)
 - int + int = int
 - bool && bool = bool
 - ...

Type checking

- **typecheck** computes the type of any expression:

typecheck : BCALC → {int, bool, none}

If **E** is of the form **add(E1, E2)** and

if **E1** is of type **int**, and

if **E2** is of type **int**

then **E** is of type **int**

what if one of these conditions fails?

The execution of the expression (eval) would not be defined

else, the type of **E** is **none**

Type checking

- **typecheck** computes the type of any expression:

typecheck : BCALC → {int, bool, none}

```
typecheck( add(E1, E2) ) ≡      [ t1 = typecheck ( E1 )
                                         t2 = typecheck ( E2 )
                                         if (t1 == int ) and (t2 == int )
                                         then int
                                         else none ]
```

Type checking

- **typecheck** computes the type of any expression:

typecheck : BCALC \rightarrow {int, bool, none}

All expressions **num(n)** denote integer values...
the representative of all integer values is **int**

```
typecheck( num(n) ,) ≡ int ;  
  
typecheck( true )           ≡ bool ;  
  
typecheck( false )          ≡ bool ;
```

Type checking

- **typecheck** computes the type of any expression:

typecheck : BCALC → {int, bool, none}

```
typecheck( and(E1, E2) ) ≡      [ t1 = typecheck ( E1 )  
                                         t2 = typecheck ( E2 )  
                                         if (t1 == bool) and (t2 == bool)  
                                         then bool  
                                         else none ]
```

Soundness of typing

- We can check that, for all program P we know that

typecheck(P)

is well defined and always terminates [Why?]

- We can prove the following theorem that relates the typing with the evaluation of programs. [How?]

Teorema: For all program P e tipo \mathcal{T} ,

If **typecheck(P) = \mathcal{T}** and **eval(P) = v** then $v \in \mathcal{T}$.

Correcção da Tipificação

Theorem: For all program P e tipo \mathcal{T} ,

If $\text{typecheck}(P) = \mathcal{T}$ and $\text{eval}(P) = v$ then $v \in \mathcal{T}$.

In particular,

if $\text{typecheck}(P) \neq \text{none}$ then $\text{eval}(P) \neq \text{error}$.

Which means that either:

P does not terminate, or

P terminates and has no runtime errors.

“Well-typed programs don’t go wrong” (Robin Milner)

Correcção da Tipificação

Teorema: For all program P e tipo \mathcal{T} ,

If $\text{typecheck}(P) = \mathcal{T}$ and $\text{eval}(P) = v$ then $v \in \mathcal{T}$.

This property is known as the “soundness of the type system”. Ensures that

“Well-typed programs don’t go wrong” (Robin Milner)

There are many programs P such that $\text{eval}(P) = v$ e $v \in \text{int}$ but $\text{typecheck}(P) = \text{none}!$ [Example?]

Robin Milner (1934-2010)

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.



Interpretação e Compilação de Linguagens de Programação

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

2017-2018

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 03 (b)

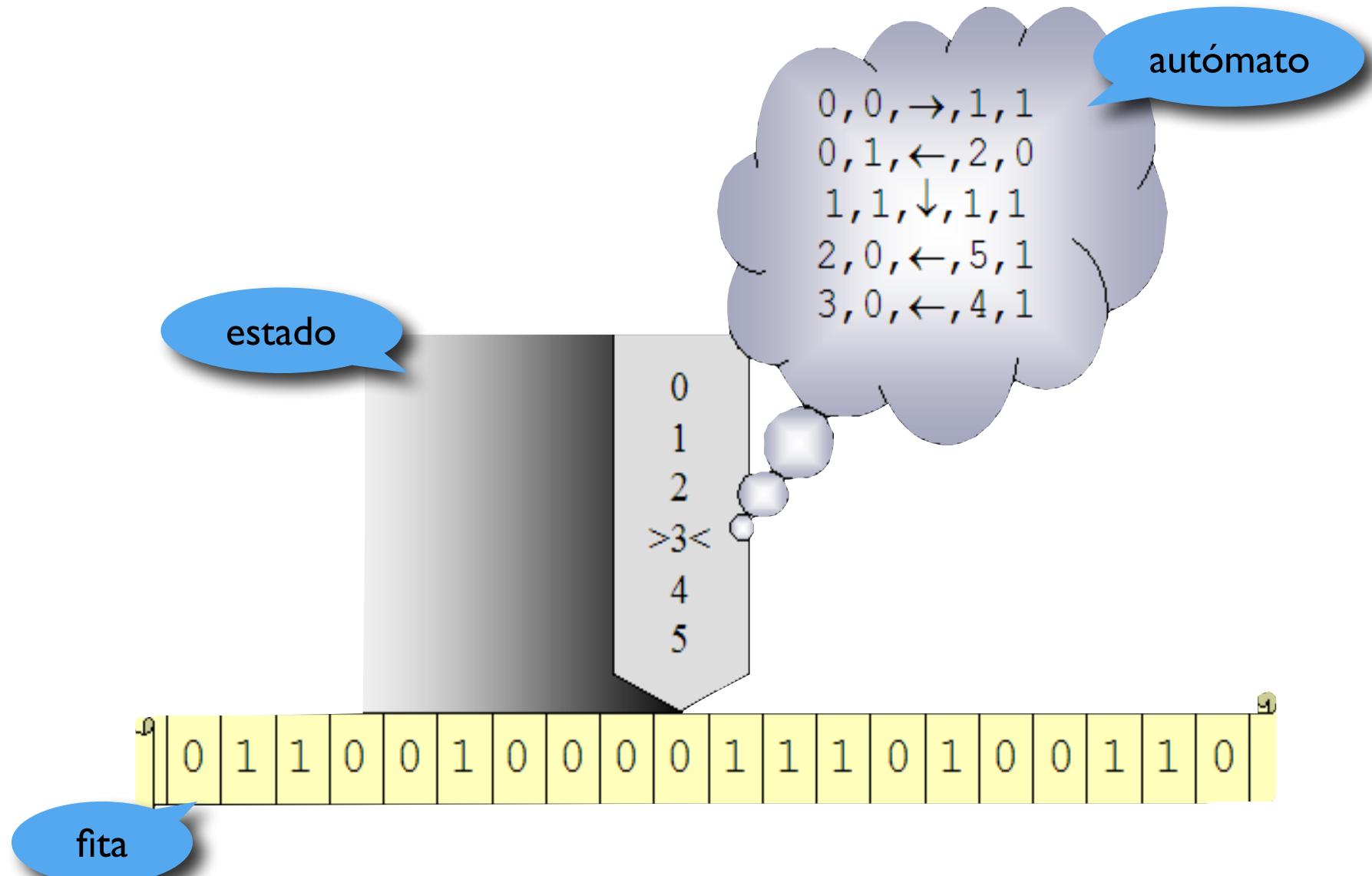
Stack based machines

Execution Environments

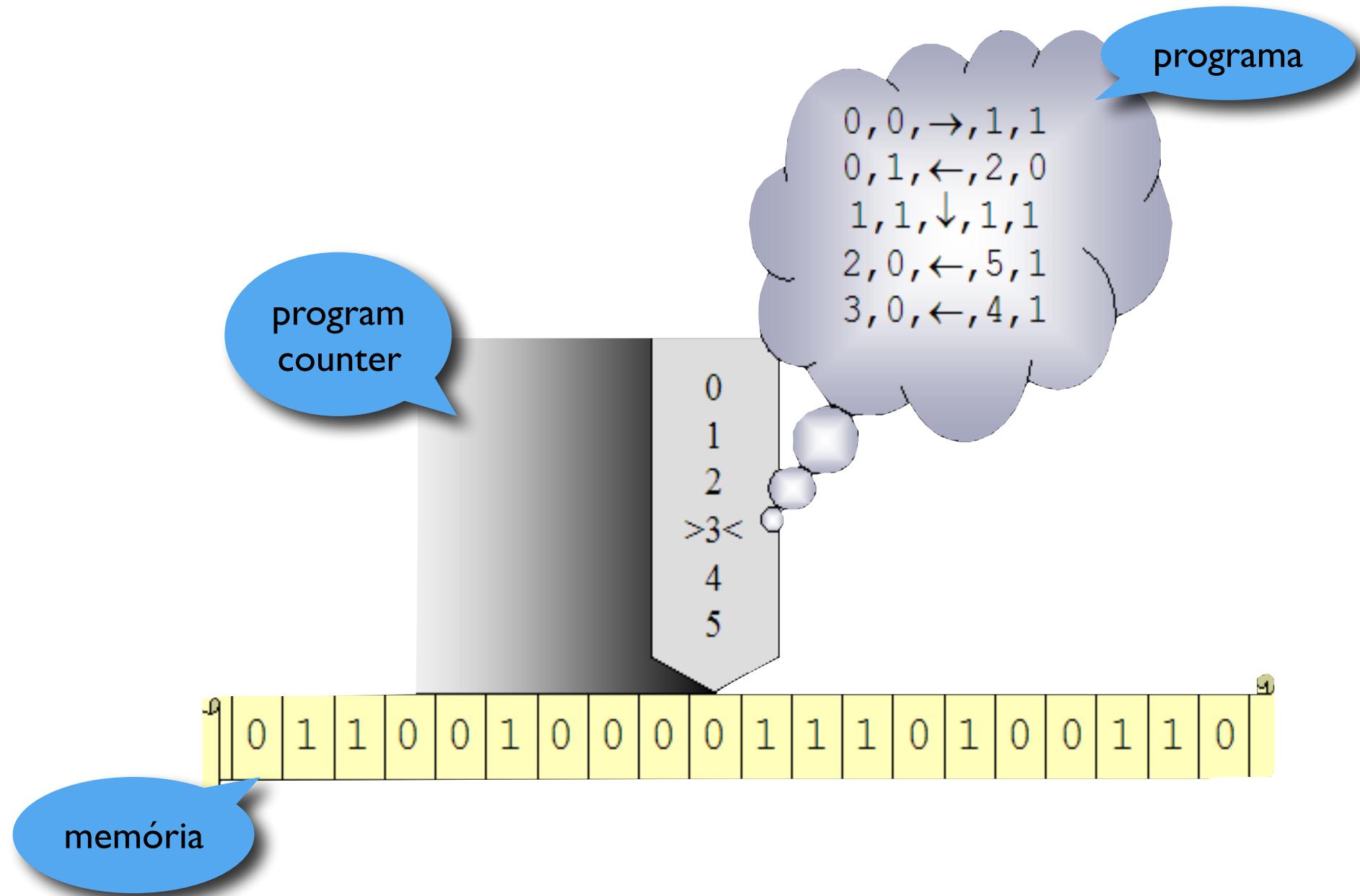
Virtual Machines (software processors)

- Máquina de Turing (Turing, 1931)
 - The first...
- SECD (Landin, 1962)
 - Stack, Environment, Code, Dump
- P-code machine (Wirth 1972)
 - Stack machine, Pascal p-code compiler
- JVM (Sun, 1995)
 - Multi-threaded, typed, targeting the Java language
- CLR (Microsoft, 2000)
 - Multi-threaded, typed, multi language
- LLVM (Vikram Adve and Chris Lattner, 2000)
 - Intermediate language, compiler infrastructure

Máquina de Turing (Turing, 1931)



Máquina de Turing (Turing, 1931)



SECD - machine (Landin, 1962)

- The first machine designed to implement the lambda calculus
- It has four registers that point to lists in memory
 - S : stack
 - E : Environment
 - C : Code
 - D : Dump
- Each memory cell may contain a datum (12) or a list (a pair with two pointers, the head and the tail of the list)
- The list (1 2 3) is represented by:



- Sample Instructions : nil, ldc, ld, sel, join, ap, ret, dum, rap

P-code machine (Wirth 1972)

- It's a stack machine. Each instruction takes its operands from the stack and returns the results to it.
- Its implementation is straight forward. It contains only a stack that is shared by code and data.

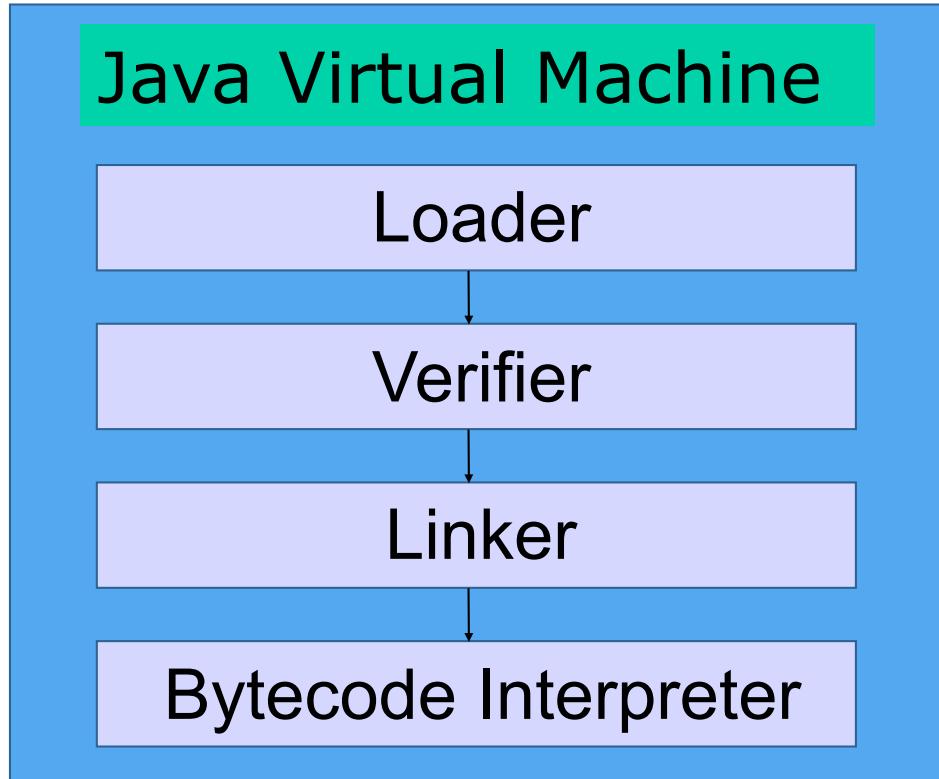
```
procedure interpret;
  const stacksize = 500;

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

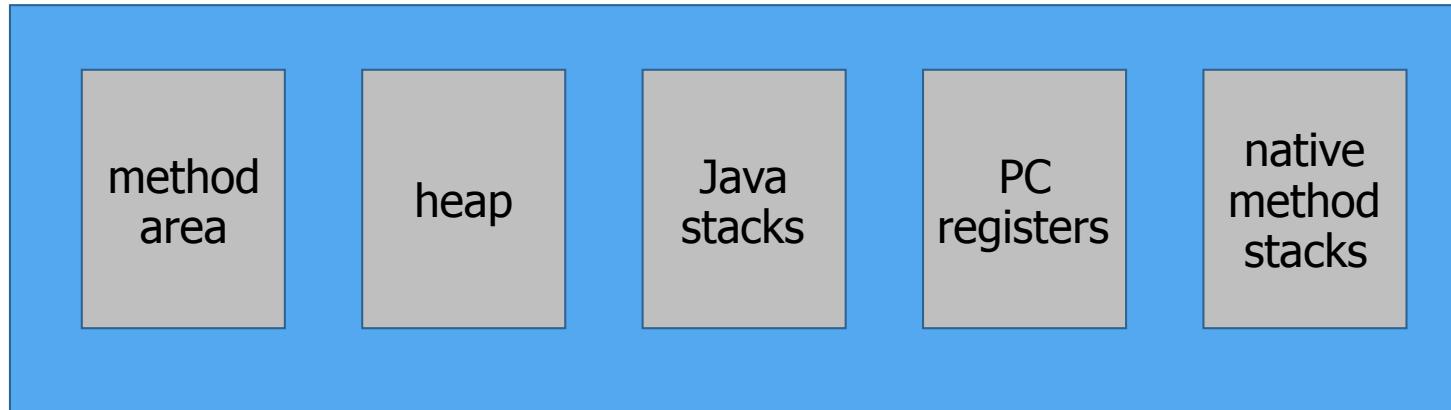
    function base(l: integer): integer;
      var b1: integer;
    begin
      b1 := b; {find base l levels down}
      while l > 0 do
        begin b1 := s[b1]; l := l - 1
        end;
      base := b1
    end {base};

  begin
    writeln(' start p1/0');
    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;
        end;
    until i = 0;
  end
```

Java Virtual Machine (Sun, 1995)

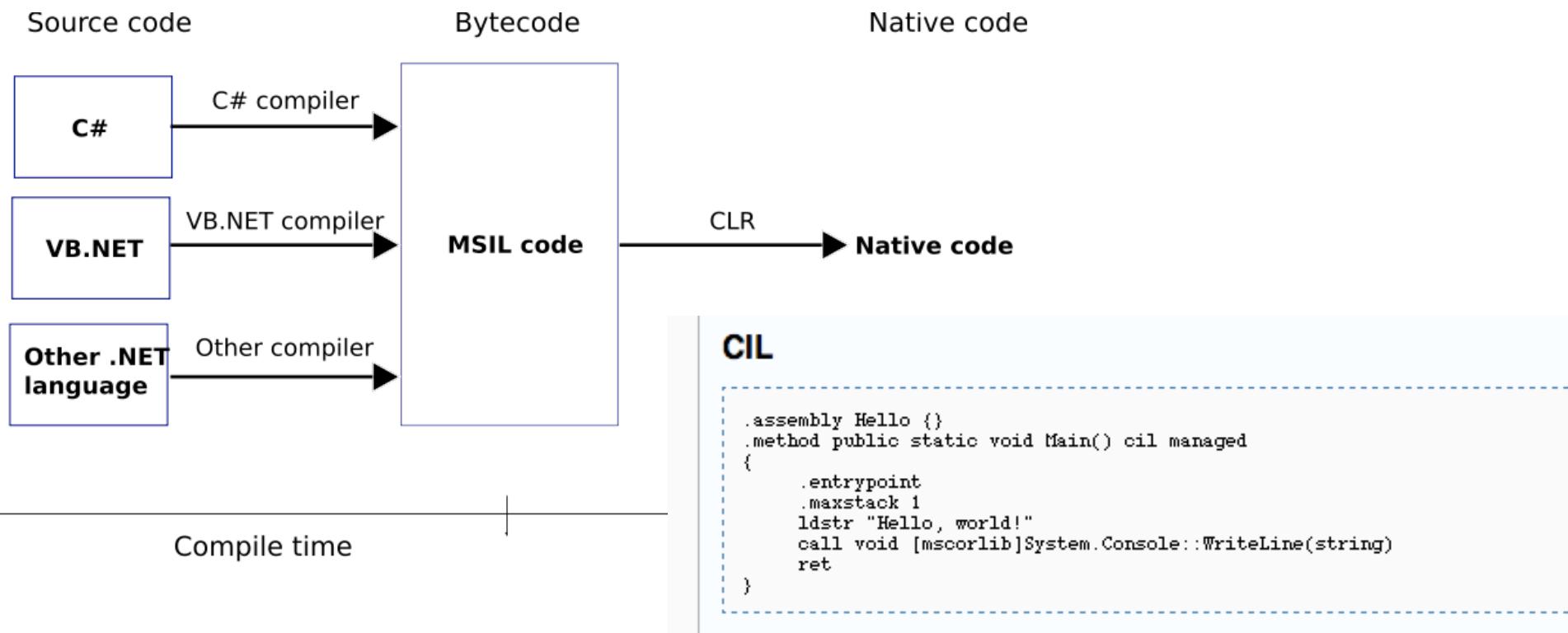


```
// Bytecode stream: 03 3b 84  
// 00 01 1a 05 68 3b a7 ff f9  
// Disassembly:  
iconst_0 // 03  
istore_0 // 3b  
iinc 0, 1 // 84 00 01  
iload_0 // 1a  
iconst_2 // 05  
imul // 68  
istore_0 // 3b  
goto -7 // a7 ff f9
```



Common Language Runtime (Microsoft, 2000)

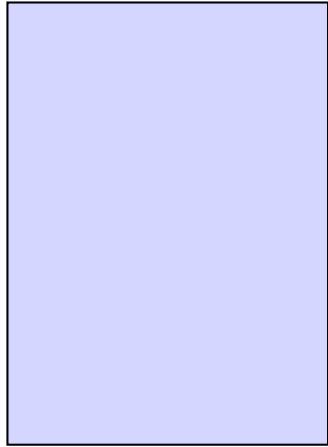
- Stack machine, the base of Microsoft .NET platform
- Designed to support several languages.
- CIL - Common Intermediate Language



Common Language Runtime

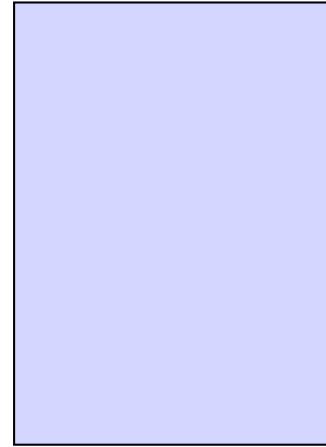
- Estruturas principais da máquina virtual CLR
 - ES: guarda resultados temporários durante a avaliação de expressões;
 - CS: guarda variáveis locais e endereços de retorno;
 - Heap: guarda objectos e outras entidades dinâmicas

Pilha de Avaliação
(Evaluation Stack)



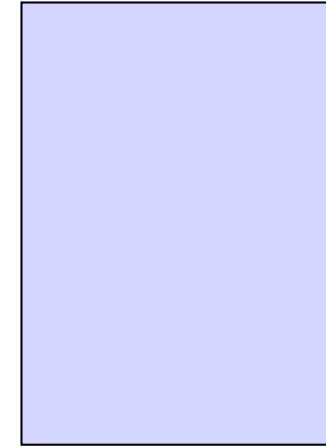
add, ...

Pilha de Chamada
(Call Stack)



call, ...

Memória Global
(Heap)



new, ...

Further reading...

Compiler Implementation, Appel

http://en.wikipedia.org/wiki/Turing_machine

http://en.wikipedia.org/wiki/SECD_machine

http://en.wikipedia.org/wiki/P-code_machine

http://en.wikipedia.org/wiki/Common_Language_Runtime

<http://en.wikipedia.org/wiki/JVM>

LLVM



- Low-level virtual machine, but really a compiler infrastructure for designing compiler tools
- Designed to easily define compiler optimisations.
- Provides an intermediate representation and tools.
- University of Illinois, open-source, Apple Xcode.
- clang replaces gcc entirely with LLVM IL.

<http://llvm.org>

LLVM - Compiler Infrastructure

- Register based intermediate language i1
- Typed registers and instructions i32
- Single static assignment i1942652
- Defines Basic block graphs (Phi operation) half
float

```
%03 = add i32 %01 %02
%04 = icmp slt i32 %03 0
br i1 %04, label %L0, label %L1
L0:
%05 = add i32 %03 2
br label %end
L1:
%06 = add i32 %03 3
br label %end
end:
%07 = phi i32 [ 0, %L0 ], [%03,%L1]
```

double
fp128
x86_fp80
ppc_fp128
[40 x i32]
[12 x [10 x float]]
[4 x i32]*
i32 (i32*)*
{i32, i32, i32}
{float, i32 (i32)*}

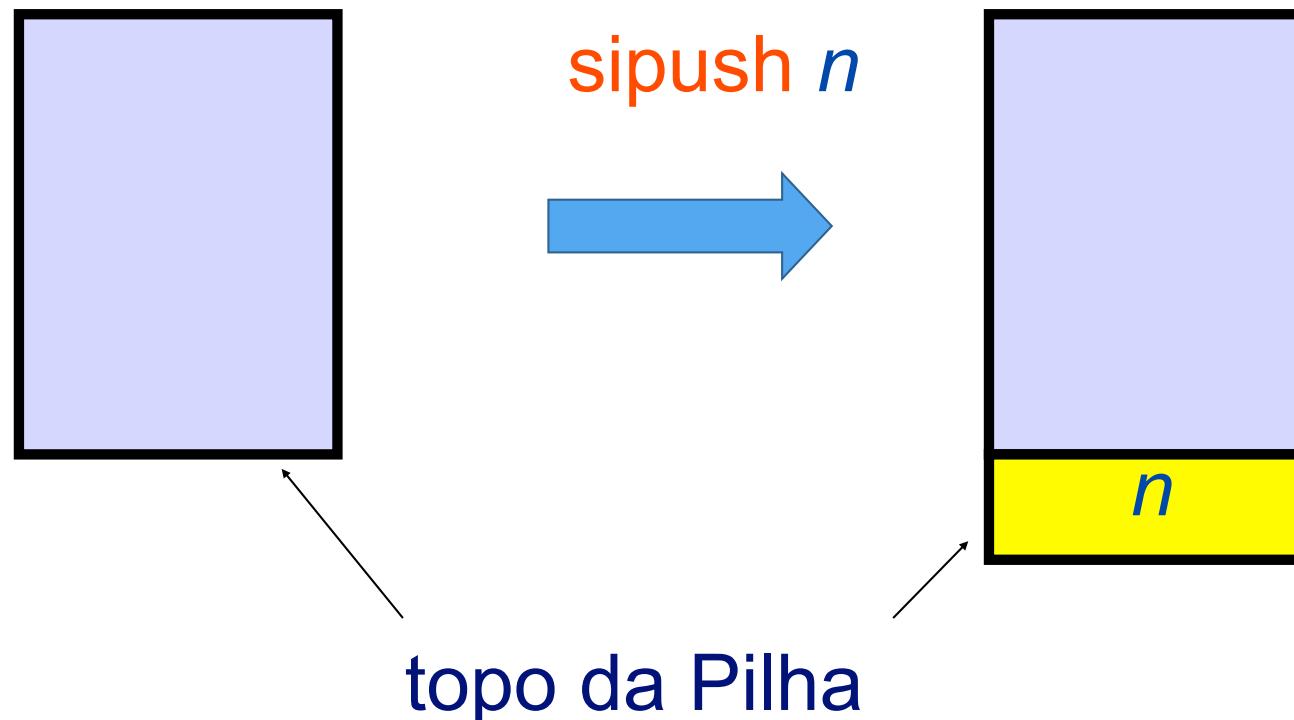
<http://llvm.org/docs/LangRef.html>

Instruções da JVM

- Máquina de pilha:
 - todas as instruções consomem argumentos do topo da Pilha, e deixam um resultado no topo da Pilha.
- “Primeiras” (5) instruções da JVM:
 - **sipush n** : Carrega o valor n (short integer) no topo da pilha
 - **iadd**: Retira dois valores inteiros do topo da pilha e coloca na pilha o resultado da soma
 - **imul** : idem para a multiplicação
 - **idiv** : idem para a divisão
 - **isub** : idem para a subtracção

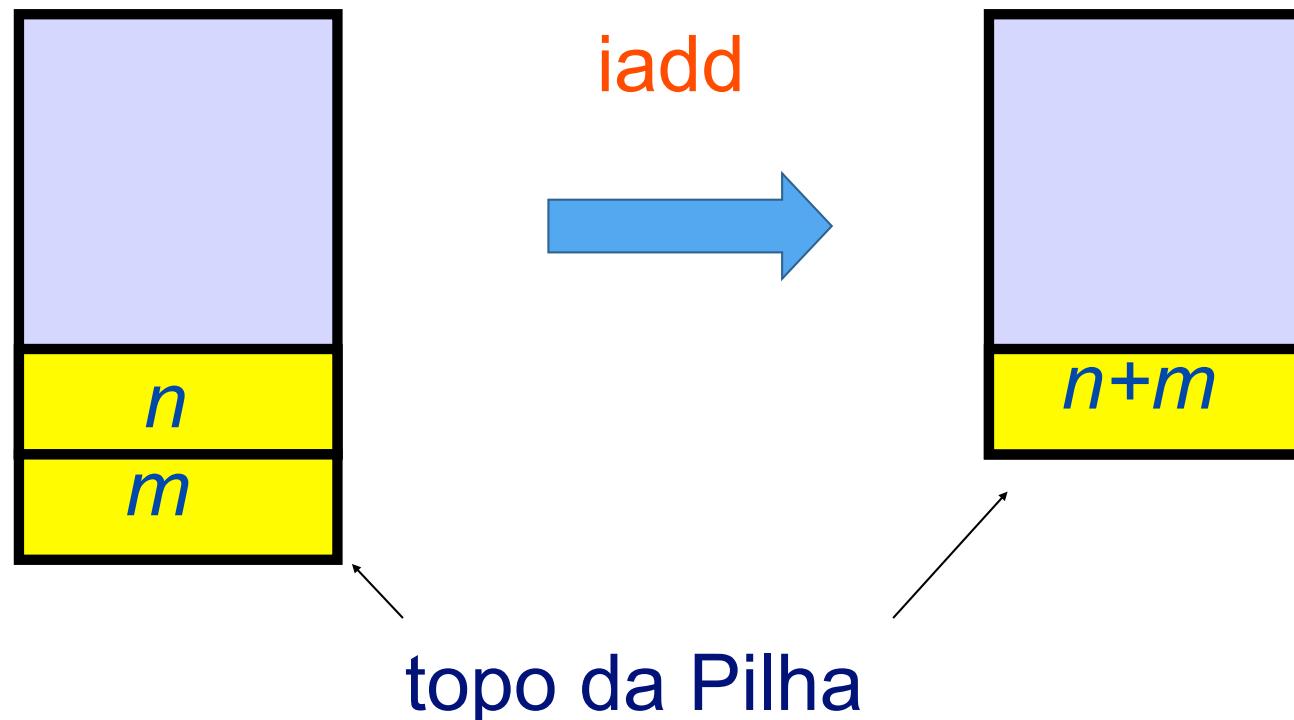
Common Language Runtime

- “Primeiras” (5) instruções: `sipush n`, `iadd`, `imul`, `idiv`, `isub`.
- Load Constant (`sipush n`)



Common Language Runtime

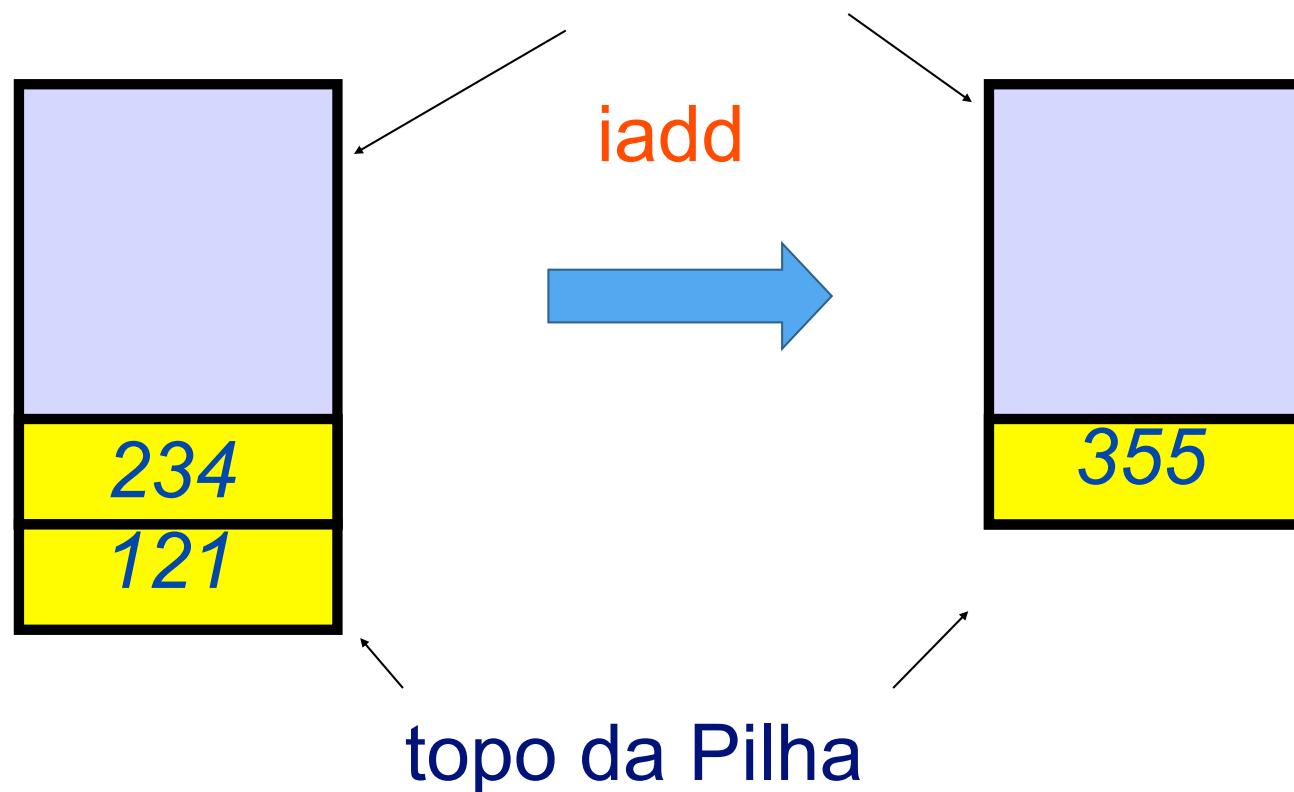
- “Primeiras” (5) instruções: `sipush n`, `iadd`, `imul`, `idiv`, `isub`.
- Add (`iadd`)



Common Language Runtime

- “Primeiras” (5) instruções: `sipush n`, `iadd`, `imul`, `idiv`, `isub`.
- Add (`iadd`)

O fundo da pilha é inalterado!



Common Language Runtime

- “Primeiras” (5) instruções: `sipush n`, `iadd`, `imul`, `idiv`, `isub`.
- $\text{Comp}(“2+2*(7-2)”) = \text{Comp}(\text{add}(\text{num}(2), \text{mul}(\text{num}(2), \text{sub}(\text{num}(7), \text{num}(2)))))$

Common Language Runtime

- “Primeiras” (5) instruções: `sipush n`, `iadd`, `imul`, `idiv`, `isub`.
- $\text{Comp}(“2+2*(7-2)”) = \text{Comp}(\text{add}(\text{num}(2), \text{mul}(\text{num}(2), \text{sub}(\text{num}(7), \text{num}(2)))))$

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

Compilador de CALC

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

$\text{comp} : \text{CALC} \rightarrow \text{CodeSeq}$

| | |
|--|---|
| se E é da forma num (n): | $\text{comp}(E) \triangleq < \text{sipush } n >$ |
| se E é da forma add (E' , E''): | $s1 = \text{comp}(E');$ $s2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{iadd} >$ |
| se E é da forma mul (E' , E''): | $v1 = \text{comp}(E');$ $v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{imul} >$ |
| se E é da forma sub (E' , E''): | $v1 = \text{comp}(E');$ $v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{isub} >$ |
| se E é da forma div (E' , E''): | $v1 = \text{comp}(E');$ $v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{idiv} >$ |

Compilador de CALC

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

$\text{comp} : \text{CALC} \rightarrow \text{CodeSeq}$

$\text{comp}(\text{num}(n)) \triangleq < \text{sipush } n >$

$\text{comp}(\text{add}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{iadd} >$

$\text{comp}(\text{mul}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{imul} >$

$\text{comp}(\text{sub}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{isub} >$

$\text{comp}(\text{div}(E', E'')) \triangleq \text{comp}(E') @ \text{comp}(E'') @ < \text{idiv} >$

Correcção do Compilador

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer da linguagem CALC numa sequência de instruções da linguagem CIL (CLR)

$$\text{comp} : \text{CALC} \rightarrow \text{CodeSeq}$$

- **Propriedade de Correcção:** Quando a sequência de instruções $\text{comp}(E)$ é executada num estado da máquina virtual em que a pilha está no estado p , quando termina deixa sempre a máquina no estado $\text{push}(v, p)$, em que v é o valor da expressão E .

Compilador de CALCI

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer de CALCI numa sequência de instruções CLR

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

No algoritmo interpretador para expressões abertas é possível saber o valor de um identificador usando o nome.

O processo de compilação deve depender o menos possível de nomes do domínio dos programas. Onde for possível, deve ser estabelecida uma ligação entre os identificadores e um endereço de memória.

Ideia geral: Usar um vector de inteiros na pilha de execução da JVM para guardar os valores dos identificadores declarados por uma expressão.

Interpretação e Compilação de Linguagens de Programação

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

2017-2018

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 04

Binding and Scope

Binding and Scope

The use of identifiers is the first basic tool to create abstraction in programming languages. An identifier used in an expression (or program) is a sub-expression whose meaning is determined separately. It follows the substitution principle that states that the meaning is determined by directly replacing identifiers by their definitions in the expressions

- Literals and identifiers
- Declaration expression
- The scope of a declaration
- Occurrences of an identifier (free, bound and binding)
- Closed and open expressions
- The fundamental construct **decl id=E in E end.**
- A programming language with identifiers: CALCI.
- Substitution based operational semantics
- Environment based operational semantics
- Environment based compiler semantics

Literals and Identifiers

- Literals (or constants)
 - denote well determined entities or values in all contexts
 - In natural languages they correspond to “nouns”
 - ML: **true**, **false**, []
 - C: 1, 1.0, 0xFF, “hello”, int
- Identifiers (or names)
 - denote entities or values that depend on the context where they are referenced
 - In natural languages they correspond to pronouns.
 - Java: x, Count, System.out
 - C: printf

Binding and Scope

- Os **Literals** and **identifiers** always denote a well-determined and immutable entity.
- The entity denoted by a literal (or the value of a literal) is determined by the literal itself (**23**, “**hi !**”, etc.).
- The association between an identifier and the denoted entity or value is called *binding*.
- In general, the binding between an identifier and the denoted entity is established in a given syntactical and is introduced by a *declaration*.
- The syntactic context where a binding is valid is called the *scope* of a binding.

Ligação e Âmbito

- O identificador **x** denota uma variável de estado (célula de memória)

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

Ligação e Âmbito

- O identificador **x** denota uma variável de estado (célula de memória)

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

Âmbito da ligação

Ligação e Âmbito

- O identificador **j** denota uma variável de estado (célula de memória)

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

Ligação e Âmbito

- O identificador **j** denota uma variável de estado (célula de memória)

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

Âmbito da ligação

Elementos de um âmbito

- A ligação entre um identificador e a respectiva entidade por este denotada (valor, posição de memória, etc) envolve os seguintes ingredientes:
 - Uma (única!) ocorrência ligante:
em geral, corresponde à declaração do identificador.
 - O âmbito da ligação
A “parte/região/zona/fragmento” do programa onde a ligação em causa tem efeito
 - Várias (zero ou mais) ocorrências ligadas
todas as ocorrências do identificador, distintas da ocorrência ligante, que existem dentro do âmbito

Ocorrências ligantes e ligadas

- Ocorrências do identificador **x**

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

The diagram illustrates the binding occurrences of the identifier **x** in the provided C code. Two instances of **x** are highlighted with blue circles: one at the function parameter declaration and another at the local variable declaration **int x=j+y;**. A curved arrow originates from each of these highlighted nodes and points to the label **Ocorrências ligantes**, which is positioned to the right of the code block.

Ocorrências ligantes

Ocorrências ligantes e ligadas

- Ocorrências do identificador **x**

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

Ocorrências ligadas

Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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

Ocorrências ligadas

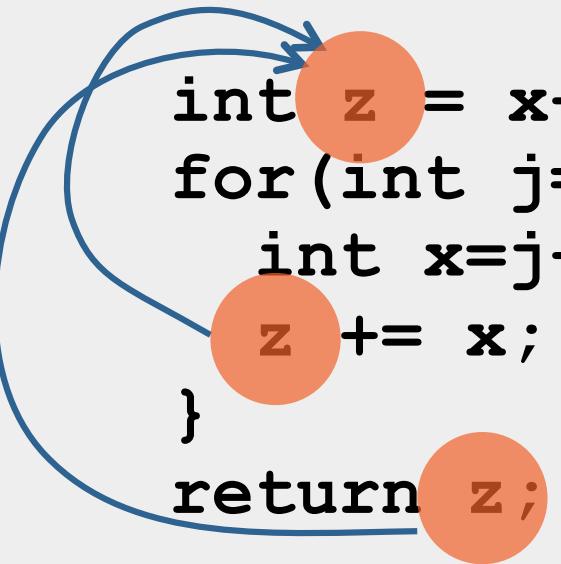
- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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

Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

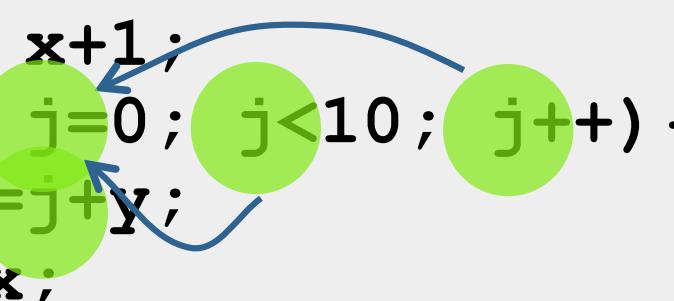
```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++) {
        int x=j+y;
        z += x;
    }
    return z;
}
```



Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

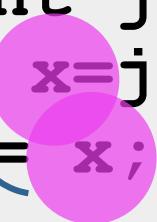
```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++) {
        int x=j+y;
        z += x;
    }
    return z;
}
```



Ocorrências ligadas

- Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

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



Ocorrências livres

- Uma ocorrência de identificador que não é ligada nem ligante diz-se **livre**

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

Expressões Abertas e Fechadas

- Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- Exemplos de expressões abertas:

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

C

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

OCaml

Expressões Abertas e Fechadas

- Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- Exemplos de expressões abertas:

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

ocorrência livre

C

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

ocorrência livre

OCaml

Semântica de expressões abertas

- A denotação de uma subexpressão de programa só pode ser calculada se se conhecer a denotação de cada identificador que nela ocorra livre.
- A definição de uma semântica composicional para linguagens com declarações de identificadores tem necessariamente que considerar expressões abertas.

Por exemplo, a expressão OCaml

```
let x = 2 in (x+x)
```

é fechada mas contém uma subexpressão aberta.

- Um programa (fragmento fechado) pode conter no seu interior expressões abertas.

[Dê exemplos de linguagens de programação onde seja possível compilar um programa aberto]

Ambiente

- Um programa fechado fornece necessariamente ligações para todas as ocorrências livres de identificadores que ocorram nas suas subexpressões (através de declarações).

Para cada subexpressão E de um programa P , ao conjunto de todas as ligações no âmbito das quais E ocorre chama-se o **ambiente** de E em P .

Ambiente (Quiz)

- Qual o ambiente da subexpressão “**x+1**”?

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

Ambiente (Quiz)

- Qual o ambiente da subexpressão “**z+=x**”?

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

Ambiente (Quiz)

- Qual o ambiente da subexpressão “**return z**”?

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

Exemplo: A Linguagem CALCI

- A linguagem CALCI estende a linguagem CALC com a possibilidade de se poderem introduzir e usar identificadores usando a construção **declare**:

```
decl Id = Expressão1 in Expressão2 end
```

Numa expressão **decl**, a primeira ocorrência de *Id* é **ligante**, no âmbito definido pela *Expressão2*

- Definimos os programas CALCI como sendo as **expressões fechadas** da linguagem CALCI.

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

A Linguagem CALCI (como tipo indutivo)

- Tipo de dados CALCI com os construtores: num, add, mul, div, sub, id, decl

num: Integer → CALCI

id: String → CALCI

add: CALCI × CALCI → CALCI

mul: CALCI × CALCI → CALCI

div: CALCI × CALCI → CALCI

sub: CALCI × CALCI → CALCI

decl: String × CALCI × CALCI → CALCI

A Linguagem CALC1 (como tipo indutivo)

- Tipo de dados CALC1 com os construtores: num, add, mul, div, sub, id, decl

num: Integer → CALC1

id: String → CALC1

add: CALC1 × CALC1 → CALC1

mul: CALC1 × CALC1 →

div: CALC1 × CALC1 →

sub: CALC1 × CALC1 →

decl: String × CALC1 × C

```
type calc1 =
| Number of int
| Add of calc1 * calc1
| Sub of calc1 * calc1
| Mul of calc1 * calc1
| Div of calc1 * calc1
| Id of string
| Decl of string * calc1 * calc1
```

```
| Dec of string * calc1 * calc1
| Id of string
```

Semântica de CALCI (1)

- A função semântica I de CALCI pode ser definida por um **algoritmo** que “sabe como interpretar” **todas** as expressões de CALCI, determinando o seu **valor ou efeito**.

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

CALCI = conjunto das expressões fechadas

Integer = conjunto dos significados (denotações)

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

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

eval(num(n))

$\triangleq n$

eval(add(E_1, E_2))

$\triangleq \text{eval}(E_1) + \text{eval}(E_2)$

...

eval(decl(s, E_1, E_2))

$\triangleq ???$

Intuitivamente: o significado da expressão contendo identificadores deve ser o mesmo da expressão em que os identificadores são substituídos pelas subexpressões que eles representam.

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

eval : CALCI → Integer

eval(num(n))

$\triangleq n$

eval(add(E_1, E_2))

$\triangleq \mathbf{eval}(E_1) + \mathbf{eval}(E_2)$

...

eval(decl(s, E_1, E_2))

$\triangleq [G = \mathbf{subst}(E_1, s, E_2); \mathbf{eval}(G);]$

Intuitivamente: o significado da expressão contendo identificadores deve ser o mesmo da expressão em que os identificadores são substituídos pelas subexpressões que eles representam.

A Função Subst

`subst(F, s, E)`

Calcula a expressão que resulta de substituir todas as ocorrências livres do identificador `s` pela expressão `F` na expressão `E`.

$$\text{subst}(y+z, \textcolor{red}{s}, \textcolor{red}{s+s+2}) = (y+z)+(y+z)+2$$

$$\text{subst}(u, \textcolor{red}{y}, \text{decl } x=\textcolor{red}{y} \text{ in decl } y=2 \text{ in } x+y) = \text{decl } x=u \text{ in decl } y=2 \text{ in } x+y$$

Definição da Função Subst

subst(F, s, num(n)) \triangleq num(n);

subst(F, s, id(s)) \triangleq F;

subst(F, s, add(E1, E2)) \triangleq add(**subst(F, s, E1)**, **subst(F, s, E2)**);

...

subst(F, s, decl(s, E1, E2)) \triangleq [/* caso s = s' */ * ???]

subst(F, s, decl(s', E1, E2)) \triangleq [/* caso s \neq s' */ ???]

Definição da Função Subst

```
subst(F, s, num(n) )       $\triangleq$  num(n);  
subst(F, s, id(s) )        $\triangleq$  F;  
subst(F, s, add(E1, E2) )  $\triangleq$  add( subst(F, s, E1), subst(F, s, E2));  
...  
subst(F, s, decl(s, E1, E2))  $\triangleq$  [/* caso s = s' */ G = subst(F, s, E1);  
           decl(s, G, E2); ]  
subst(F, s, decl(s', E1, E2))  $\triangleq$  [/* caso s ≠ s' */ G = subst(F, s, E1);  
           decl(s', G, subst(F, s, E2)); ]
```

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

eval : CALCI → integer

eval(num(n))

$\triangleq n$

eval(add(E_1, E_2))

$\triangleq \mathbf{eval}(E_1) + \mathbf{eval}(E_2)$

...

eval(decl(s, E_1, E_2))

$\triangleq \mathbf{eval}(\mathbf{subst}(E_1, s, E_2));]$

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

eval : CALCI → integer

eval(num(n))

$\triangleq n$

eval(add(

```
let rec eval a =
  match a with
  | Number n -> n
  | Add (l,r) -> (eval l) + (eval r)
  | Sub (l,r) -> (eval l) - (eval r)
  | Mul (l,r) -> (eval l) * (eval r)
  | Div (l,r) -> (eval l) / (eval r)
  | Decl (s,l,r) -> eval (subst l s r)
  | Id s -> ???
```

eval(dec(

| If s -> $\lambda x. f(x)$
| Decl (s, t, u) -> eval (subst t s u)
| Def A -> eval A

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

`eval : CALCI → integer`

```
eval( num(n) )           ≡ n
eval( add(l,r) )          let rec eval a =
eval( dec(s,l,r) )        match a with
eval( id(s) )              | Number n -> n
                           | Add (l,r) -> (eval l) + (eval r)
                           | Sub (l,r) -> (eval l) - (eval r)
                           | Div (l,r) -> (eval l) / (eval r)
                           | Decl (s,l,r) -> eval (subst l s r)
                           | Id s -> ???
```

é preciso programar o caso do identificador??

```
| Id s -> ???
```

```
| If s -> ...
```

```
| Decl (s,l,r) -> eval (subst l s r)
```

ICLP 2017-2018

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

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

eval(num(n))

$\triangleq n$

eval(add(

```
let rec eval a =
  match a with
  | Number n -> n
  | Add (l,r) -> (eval l) + (eval r)
  | Sub (l,r) -> (eval l) - (eval r)
```

eval(dec(

Porquê? o que fazer?

```
| Div (l,r) -> (eval l) / (eval r)
| Decl (s,l,r) -> eval (subst l s r)
| Id s -> ???
```

```
| If s -> sss
| Dec (s,l,r) -> eval (subst l s r)
| Decl (s,l,r) -> eval (subst l s r)
```

Interpretador de CALCI

- Algoritmo eval(E) para calcular o valor de uma expressão fechada qualquer E de CALCI:

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

eval(num(n))

$\triangleq n$

eval(add(

```
let rec eval a =
  match a with
    | Number n -> n
    | Add (l,r) -> (eval l) + (eval r)
    | Sub (l,r) -> (eval l) - (eval r)
```

eval(dec(

Estamos a trabalhar com
expressões fechadas. A existência
de identificadores denota a falta
de uma declaração, um erro!

Semântica de CALCI (2)

- A semântica da linguagem CALCI baseada em substituições é muito conveniente do ponto de vista da especificação pois é muito simples.

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

- Já do ponto de vista operacional, é conveniente definir uma semântica mais concreta, recorrendo à manipulação de ambientes.
- A manipulação de ambientes também é mais conveniente como técnica de implementação de interpretadores (como estruturas de dados auxiliares).

Semântica de CALCI (2)

A função semântica I de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

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

CALCI = programas abertos

ENV = ambientes

Integer = significados (denotações)

Semântica de CALCI (2)

A função semântica I de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

$$I : \text{CALCI} \times \text{ENV} \rightarrow \text{integer}$$

| | |
|-----------|--|
| CALCI = | <pre>let rec eval e env = match e with Number n -> n Id s -> find s env Add (l,r) -> (eval l) + (eval r) Sub (l,r) -> (eval l) - (eval r) Mul (l,r) -> (eval l) * (eval r) Div (l,r) -> (eval l) / (eval r) Decl (s,l,r) -> ...</pre> |
| ENV = | |
| Integer = | |

Semântica de CALCI (2)

A função semântica I de CALCI pode ser definida por um **algoritmo** interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

CALCI

ENV

Integer

$I : \text{CALCI} \times \text{ENV} \rightarrow \text{integer}$

```
let rec eval e env =
  match e with
    | Number n -> n
    | Id s -> find s env
    | Add (l,r) -> (eval l env) + (eval r env)
    | ...
    | Decl (s,l,r) ->
        let v = eval l env in
        let new_env = assoc s v env in
        eval r new_env
```


Ambiente “mutável”

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável ao estilo Object-Oriented.
- Numa linguagem estruturada, com âmbitos encaixados hierarquicamente, a adição e remoção de ligações entre identificadores e valores segue uma disciplina LIFO.
- Um ambiente guarda as associações correspondentes a um determinado âmbito (e todos os âmbitos envolventes). A partir de um ambiente pode criar-se um novo nível, correspondendo a um âmbito encaixado.
- Os ambientes têm duas operações fundamentais:

Environ BeginScope()

- que cria um novo nível local vazio, onde serão colocadas as novas ligações.
- Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível (Porquê?)

Environ EndScope()

- que coloca o ambiente no estado anterior à última operação BeginScope().

Ambiente “mutável”

- Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável ao estilo Object-Oriented.

- Outras operações fundamentais:

void Assoc(String id, Value val)

- Adiciona uma nova ligação que associa ao identificador **id** o valor **val** indicado.
- A ligação é adicionada ao último nível (mais recente) do ambiente.

- Devolve o valor associado ao identificador **id** no ambiente.

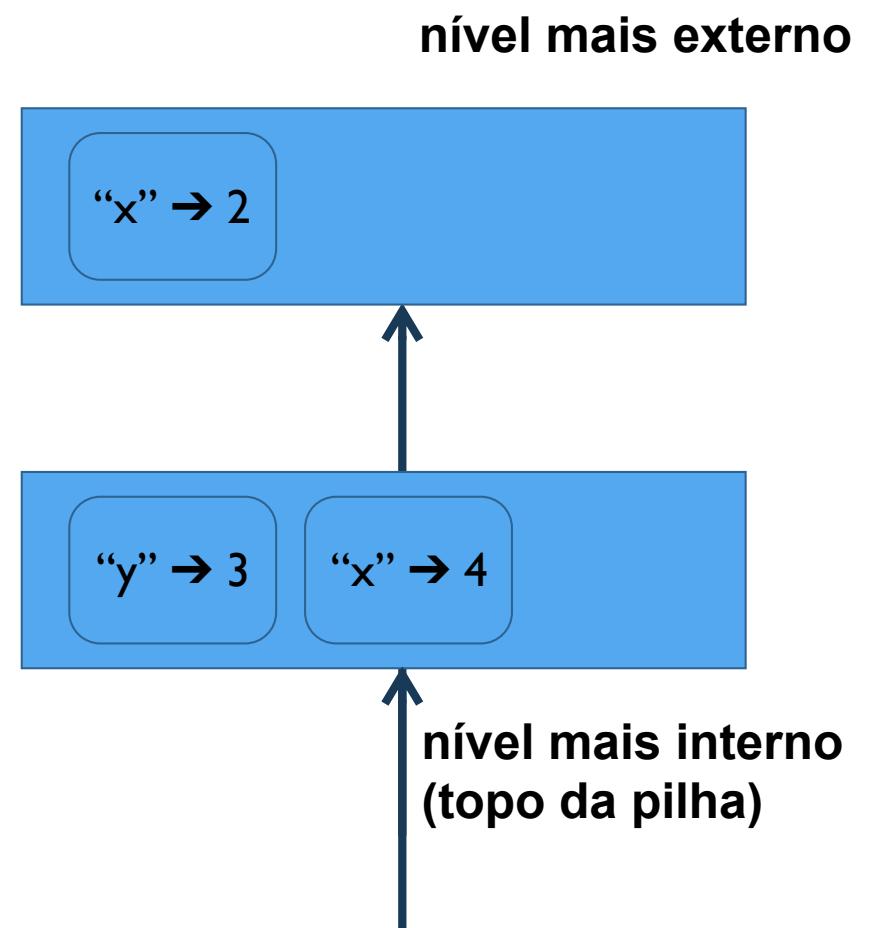
Value Find(String id)

- A pesquisa é efectuada do nível mais “recente” para o mais “antigo”, de modo a respeitar o encaixe dos âmbitos das declarações.

A “interface” Ambiente

- Simule mentalmente:

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x");          // devolve 2
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y");          // devolve 3
val = env.Find("x");          // devolve 4
env=env.EndScope()
val = env.Find("x")           // devolve 2
```



A “interface” Ambiente

- Implementado como pilha de dicionários ...

```
env = new Environment();
env.Assoc("x", 2);

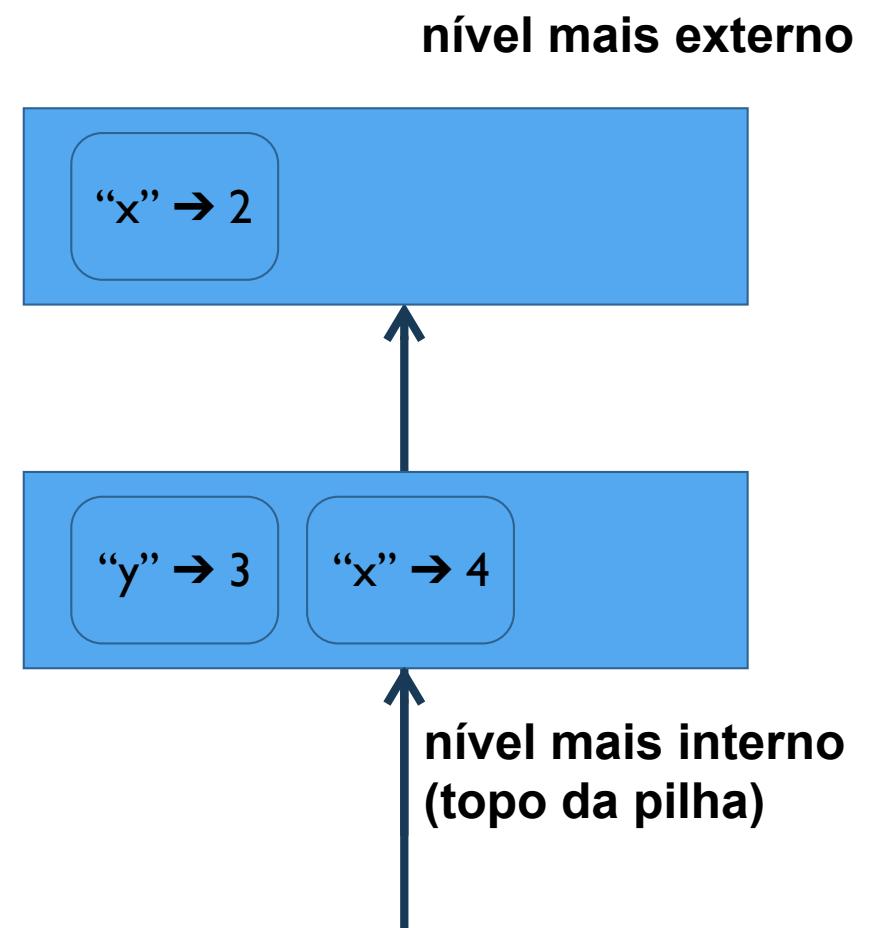
val = env.Find("x");          // devolve 2

env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);

val = env.Find("y");          // devolve 3
val = env.Find("x");          // devolve 4

env=env.EndScope()

val = env.Find("x")           // devolve 2
```



Interpretador de CALCI

- Algoritmo $\text{eval}(E, \text{env})$ para calcular o valor de uma expressão E da linguagem CALCI:

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

eval(num(n) , env)

$\triangleq n$

eval(id(s) , env)

$\triangleq \text{env}.\text{Find}(s)$

eval(add(E_1, E_2) , env)

$\triangleq \text{eval}(E_1, \text{env}) + \text{eval}(E_2, \text{env})$

...

eval(decl(s, E_1, E_2) , env) \triangleq [

$v_1 = \text{eval}(E_1, \text{env});$
 $\text{env} = \text{env}.\text{BeginScope}();$

$\text{env}.\text{Assoc}(s, v_1);$

$\text{val} = \text{eval}(E_2, \text{env});$

$\text{env} = \text{env}.\text{EndScope}();$

$\text{val}]$

Erros de execução

- O que é que acontece se não for encontrado o identificador no ambiente?
A que corresponde essa falha?
A um erro de execução do tipo “identificador não declarado”.
- A função não está definida para os programas em que há ocorrências de identificadores sem declaração.
- Que outros tipos de erros de execução podem ocorrer?
- São o mesmo tipo de erros? É possível evitar uns e outros não?

Interpretação e Compilação de Linguagens de Programação

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

2017-2018

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 05

Typing and Compiling Declarations

Runtime errors

- What can go wrong in this semantics?

eval(num(n) , env)

$\triangleq n$

eval(id(s) , env)

$\triangleq env.\text{Find}(s)$

eval(add(E_1, E_2) , env)

$\triangleq \mathbf{eval}(E_1, env) + \mathbf{eval}(E_2, env)$

...

eval(decl(s, E_1, E_2) , env) \triangleq [$v_1 = \mathbf{eval}(E_1, env);$
 $env = env.\text{BeginScope}();$
 $env.\text{Assoc}(s, v_1);$
 $val = \mathbf{eval}(E_2, env);$
 $env = env.\text{EndScope}();$
val]

Type checking

- **typecheck** computes the type of any expression:

typecheck : CALC1 → {int, bool, none}

If **E** if of the form **decl(x,E1, E2)** and

if E1 is of type **T1**, and

if E2 is of type **T2**, when x is of type **T1**

then **E** is of type **T2**

else, the type of **E** is **none**

what if one of these conditions fails?

The execution of the expression (eval) would not be defined

else, the type of **E** is **none**

Type checking

- **typecheck** computes the type of any expression:

typecheck : CALC1 → {int, bool, none}

typecheck(num(*n*) , *env*) \triangleq int

typecheck(id(*s*) , *env*) \triangleq *env*.Find(*s*)

typecheck(add(*E*1,*E*2) , *env*) \triangleq ...

...

typecheck(decl(*s*, *E*1, *E*2), *env*) \triangleq [*t*1 = **typecheck**(*E*1, *env*);
env = *env*.BeginScope();
env.Assoc(*s*, *t*1);
t = **typecheck**(*E*2, *env*);
env = *env*.EndScope();
t]

Recall: Soundness of typing

- We can check that, for all program P we know that

typecheck(P)

is well defined and always terminates [Why?]

- We can prove the following theorem that relates the typing with the evaluation of programs. [How?]

Teorema: For all program P e tipo \mathcal{T} ,

If **typecheck(P) = \mathcal{T}** and **eval(P) = v** then $v \in \mathcal{T}$.

Compiling declarations

- Evaluation stack
 - Supports the (assembly) language operations.
 - Not for local variables (that's in the call stack)
- Call stack
 - Stores local variables (FIFO), stack frames lifespan is limited to the block life time.
 - Heap allocated stack frames (support languages with closures, discussed later in the course)
 - Pointer from call stack to heap allocated frames (SP).

Recall: CALC1 compiler

- Algoritmo $\text{comp}(E)$ para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

$\text{comp} : \text{CALC1} \rightarrow \text{CodeSeq}$

| | |
|--|---|
| se E é da forma num (n): | $\text{comp}(E) \triangleq < \text{sipush } n >$ |
| se E é da forma add (E' , E''): | $s1 = \text{comp}(E');$ $s2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{iadd} >$ |
| se E é da forma mul (E' , E''): | $v1 = \text{comp}(E');$ $v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{imul} >$ |
| se E é da forma sub (E' , E''): | $v1 = \text{comp}(E');$ $v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{isub} >$ |
| se E é da forma div (E' , E''): | $v1 = \text{comp}(E');$ $v2 = \text{comp}(E'');$ $\text{comp}(E) \triangleq s1 @ s2 @ < \text{idiv} >$ |

CALCI compiler using heap allocated records

- Each stack frame is allocated (in the heap) and dimensioned to store all values of declared identifiers (this can be highly optimized).
- Nested declarations produce a chain of stack frames accessible from the SP pointer (the pointer from one frame to another is the static link SL).
- The head of the list of stackframes (from local to global declarations) is kept updated at all times (SP).
- The structure of frames is typed and abstracts source code identifiers.
- So, each stack frame is different.

decl

x1 = E1

x2 = E2

...

xn = En

in

E

```
.class frame_id
.super java/lang/Object
.field public SL Lancester_frame_id;
.field public loc_00 type
.field public loc_01 type
...
.field public loc_n type

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

CALCI compiler

decl

x1 = E1

x2 = E2

...

xn = En

in

E

new *frame_id*
dup
invokespecial *frame_id/*<init>()
dup
aload 0 ; SP
putfield *frame_id/*SL *Lframe_up*;
dup
[[E1]]
putfield *frame_id/*loc_00 *type*;
...
astore 0; SP
[[E]]
aload 0; SP
checkcast *frame_id*
getfield *frame_id/*SL *Lframe_up*;
astore 0 ; SP

CALCI compiler

```
decl
  x1 = E1
in decl
  x2 = E2
in
  x1 + x2
```

```
aload 0 ;SP
checkcast frame_2
getfield frame_2/SL Lframe_1;
getfield frame_1/loc_00 I
```

```
aload 0 ;SP
checkcast frame_2
getfield frame_2/loc_00 I
```

iadd

CALCI compiler

- An identifier is localized by a pair: the number of SL hops needed (jumps), and the localisation in the frame (offset).
- The environment stores the offset and computes the number of jumps.

aload 0 ;SP
checkcast *frame_id*
getfield *frame_n*/SL L*frame_n1*;
getfield *frame_n2*/SL L*frame_n2*;
getfield *frame_n2*/SL L*frame_n3*;
...
getfield *frame_1*/loc_X I

decl

x1 = E1

in decl

x2 = E2

in

x1 + x2

Compiling to a typed assembly language

- Modern intermediate languages (LLVM, CLR and JVM bytecode) are typed.
- Low-level typing allows:
 - Load time verification.
 - Runtime validation of operations (soft crash of the virtual machine)
 - Optimizations: code specialization

aload
iload
aaload
caload
dload

decl x = 1 **in**
decl y = 2 **in**
 x + y
end
end



```
new frame_1
dup
invokespecial frame_1/<init>()V
dup
sipush 1
putfield frame_1/loc_00 I
astore 0
new frame_2
dup
invokespecial frame_2/<init>()V
dup
aload 0
putfield frame_2/SL Lframe_1;
dup
sipush 2
putfield frame_2/loc_00 I
astore 0
aload 0
checkcast frame_2
getfield frame_2/SL Lframe_1;
getfield frame_1/loc_00 I
aload 0
checkcast frame_2
getfield frame_2/loc_00 I
iadd
aload 0
checkcast frame_2
getfield frame_2/SL Lframe_1;
astore 0
aconst_null
astore 0
```

Type Directed Compilation

- Frames are typed:
 - No fetching errors in the stack
 - Typed jumps in the SL chain.
- Operations are specialised:
 - No unnecessary type checks (to select operations)
 - More efficient code
- Code generation is based on type annotations in the AST

Typed AST

- To tag the AST with types during the semantic analysis phase allows the compilation phase to produce specialised code.

```
public interface ASTNode {  
    IValue eval(IEnvironment<IValue> env);  
    IValue typecheck(IEnvironment<IType> env);  
    IType getType();  
}  
  
public class ASTNum implements ASTNode {  
    ...  
    IValue typecheck(IEnvironment<IType> env) {  
        return IntType.singleton;  
    }  
    IType getType() {  
        return IntType.singleton;  
    }  
}
```

Typed AST

- To tag the AST with types during the semantic analysis phase allows the compilation phase to produce specialised code.

```
public class ASTAdd implements ASTNode {  
    ...  
    IValue typecheck(IEnvironment<IType> env) {  
        IType l = left.typecheck(env);  
        IType r = right.typecheck(env);  
        ...  
        type = IntType.singleton;  
        ...  
        return type;  
    }  
    IType getType() {  
        return type;  
    }  
}
```

Interpretação e Compilação de Linguagens de Programação

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

2017-2018

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 06

Imperative languages

Imperative languages

The expressions of the languages considered so far denote **pure values**. Identifiers are always evaluated to the same value in a program (they are constants). However, the dominant programming paradigm is the imperative one characterized by state mutation (C, Java).

The fundamental concepts in imperative languages are:

- Memory model (cell memory, set and get)
- Environment vs Memory
- Aliasing
- L-value and R-value
- Lifetime of memory cell vs scope of identifier
- Pointer manipulation, references, etc.
- The structure of imperative languages. Algol Family vs ML family
- statements based syntax vs expression based syntax
- Memory areas (stack/heap)
- Internal representation of values and objects

Modelo de Memória

- Memória:
 - é um conjunto (potencialmente infinito) de **células** cujo conteúdo é mutável.
- Cada célula de memória tem um designador **único** (a **referência** da célula) e pode conter **qualquer valor** da linguagem.
- As referências **são valores** de um tipo de dados especial **ref** que só podem ser usados no contexto da memória a que dizem respeito.
- Operações primitivas sobre uma memória \mathcal{M}

| | |
|--------------|---|
| new: | $\mathcal{M} \times \mathbf{void} \rightarrow \mathbf{ref}$ |
| set: | $\mathcal{M} \times \mathbf{ref} \times \mathbf{Value} \rightarrow \mathbf{void}$ |
| get: | $\mathcal{M} \times \mathbf{ref} \rightarrow \mathbf{Value}$ |
| free: | $\mathcal{M} \times \mathbf{ref} \rightarrow \mathbf{void}$ |

Modelo de Memória

- Operações sobre uma memória \mathcal{M}

new: $\mathcal{M} \times \mathbf{void} \rightarrow \mathbf{ref}$

Devolve uma referência para uma **nova** célula livre, e define-a como estando “em uso”.

set: $\mathcal{M} \times \mathbf{ref} \times \mathbf{Value} \rightarrow \mathbf{void}$

Altera o conteúdo da célula referida para o valor indicado. O valor “antigo” perde-se irremediavelmente.

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

Devolve o valor contido na célula referida.

free: $\mathcal{M} \times \mathbf{ref} \rightarrow \mathbf{void}$

Define a célula referida como estando **livre**, devolvendo-a ao gestor de memória, para ser reciclada.

Ambiente versus Memória

- Um ambiente indica a denotação de cada identificador declarado num programa e reflecte a estrutura estática do programa.
- A associação estabelecida no ambiente entre um identificador e o seu valor denotado é fixa e imutável dentro do âmbito respectivo.
- A memória agrega o conteúdo das variáveis de estado mutáveis, indicando o valor contido em cada localização (ou referência).
- Uma variável de estado é visível nos programas através de identificadores.
- A associação entre o identificador de uma variável de estado e a sua localização de memória é imutável e é mantida pelo ambiente.

Ambiente versus Memória

Ambiente

| Identificador | Valor |
|---------------|------------------|
| PI | 3,14 |
| x | loc ₀ |
| k | loc ₁ |
| j | loc ₁ |
| TEN | 10 |

Memória

| Localização | Valor |
|------------------|------------------|
| loc ₀ | 25 |
| loc ₁ | 12 |
| loc ₂ | loc ₁ |
| ... | ... |
| loc | 0 |

Ambiente versus Memória

Ambiente

| Identificador | Valor |
|---------------|--------|
| PI | 3,14 |
| x | 0x00FF |
| k | 0x0100 |
| j | 0x0100 |
| TEN | 10 |

Memória

| Endereço | Valor |
|----------|--------|
| 0x00FF | 25 |
| 0x0100 | 12 |
| 0x0102 | 0x0100 |
| ... | ... |
| 0xFFFF | 0 |

Propriedades do modelo de memória

Ambiente

| Identificador | Valor |
|---------------|------------------|
| PI | 3,14 |
| x | loc ₀ |
| k | loc ₁ |
| j | loc ₁ |
| TEN | 10 |

Memória

| Localização | Valor |
|------------------|------------------|
| loc ₀ | 25 |
| loc ₁ | 12 |
| loc ₂ | loc ₁ |
| ... | ... |
| loc | 0 |

Uma mesma célula de memória pode ser referida por vários identificadores distintos (**aliasing**).

Aliasing

- Dois identificadores diferentes que referem a mesma localização de memória.

```
class A {  
    int x;  
    boolean equals(A a) { return x == a.x}  
}  
  
A a = new A(); a.equals(a);  
  
  
  
  
  
int x = 0;  
void f(int* y) { *y = x+1; }  
...  
f(&x);  
// x = ?
```

Propriedades do modelo de memória

Ambiente

| Identificador | Valor |
|---------------|------------------|
| PI | 3,14 |
| x | loc ₀ |
| k | loc ₁ |
| j | loc ₁ |
| TEN | 10 |

Memória

| Localização | Valor |
|------------------|------------------|
| loc ₀ | 25 |
| loc ₁ | 12 |
| loc ₂ | loc ₁ |
| ... | ... |
| loc | 0 |

Uma célula pode conter uma referência para outra célula, permitindo a construção de estruturas de dados dinâmicas.

Operações Imperativas nas linguagens

- Reserva e inicialização de uma célula nova dada uma expressão E qualquer

var(E)

- Pode ser encontrada de diversas formas:

```
{                                em Java tem um significado  
    int a;                      em C++ tem outro,  
    MyClass m;                  qual a diferença?  
    ...  
}  
  
new int[10];  
  
malloc(sizeof(int));  
  
new MyClass();
```

Operações Imperativas nas linguagens

- Afectação de um valor a uma variável dadas as expressões E e F

E := F

A expressão E denota uma referência para uma célula, F é uma expressão qualquer

- Pode ser encontrada de diversas formas:

a = 1

i := 2

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

*(p+2) = y

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

Readln(MyLine);

Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão E que denota uma referência para uma célula.

!E

- Pode ser encontrada de diversas formas:

i := !i + 1

(linguagem ML)

*p

(linguagem C)

i = i + 1

(linguagem C)

i++

(linguagem C)

Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão E que denota uma referência para uma célula.

!E

- Pode ser encontrada de diversas formas:

i := !i + 1

(linguagem ML)

*p

(linguagem C)

i = i + 1

(linguagem C)

i++

referências

(linguagem C)

Operações Imperativas nas linguagens

- Desreferenciação de uma célula de memória dada uma expressão E que denota uma referência para uma célula.

!E

- Pode ser encontrada de diversas formas:

i := !i + 1

(linguagem ML)

*p

(linguagem C)

i =  i + 1

(linguagem C)

i++

valor

(linguagem C)

L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma **dependente do contexto**

E := 2

- (**Left-Value**) À “esquerda” do símbolo de afectação, denota o seu valor efectivo (que é uma referência)

E := E + 1

- (**Right-Value**) À “direita” do símbolo de afectação, denota o **conteúdo** da célula referida, evitando-se escrever a desreferenciação explícita

E := !E + 1

L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma **dependente do contexto**.

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

- A terminologia “L-Value” e “R-Value” não é muito feliz. Por exemplo, na expressão acima as duas subexpressões da forma A[2], uma à esquerda e outra à direita, são ambas desreferenciadas implicitamente.

Desreferenciação

- A operação de desreferenciação !E torna a interpretação dos programas mais precisa e evita qualquer ambiguidade.

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

- Por outro lado, pode argumentar-se que torna os programas mais difíceis de ler.

A desreferenciação implícita pode ser vista como uma operação de coerção (conversão ou cast).

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E)

instantiação

free(E)

libertaçāo

E := E

afectação

! E

desreferenciaçāo

```
{  
/* linguagem C */  
  
const int k = 2;  
int a = k;  
int b = a + 2;  
...  
b = a * b  
...  
}
```

```
decl  
  k = 2  
  a = var(k)  
  b = var(!a+2)  
in  
  ...  
  b := !a * !b  
  ...  
  free(a);  
  free(b)  
end
```

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E)

instanciação

free(E)

libertaçāo

E := E

afectaçāo

! E

desreferenciaçāo

```
{  
/* linguagem C */  
  
const int k = 2;  
int a = k;  
int b = a + 2;  
...  
b = a * b  
...  
}
```

```
decl  
  k = 2  
  a = var(k)  
  b = var(!a+2)  
in  
...  
b := !a * !b  
...  
free(a);  
free(b)  
end
```

libertaçāo implícita (das células atribuídas aos ids a e b)

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

| | |
|------------------|------------------|
| var(E) | instanciação |
| free(E) | libertaçāo |
| E := E | afectaçāo |
| ! E | desreferenciaçāo |

```
{  
/* linguagem C */  
  
    int k = 2;  
    int *a = &k;  
    ... *a = k+*a ...  
  
}
```

```
decl  
    k = var(2)  
    a = var(k)  
in  
    ... !a := !k+!a ...  
    free(k);  
    free(a)  
end
```

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E)

instanciação

free(E)

libertação

E := E

affectação

! E

desreferenciação

```
{  
/* linguagem C */  
  
    int k = 2;  
    const int *a = &k;  
    int b = *a;  
    ... *a = k+b ...  
  
}
```

```
decl  
    k = var(2)  
    a = k  
    b = var(!a)  
in  
    ... a := !k+!b ...  
    free(k);  
    free(b)  
end
```

Operações Imperativas

- Todas as declarações e usos de variáveis mutáveis podem exprimir-se usando as primitivas

var(E)

instantiação

free(E)

libertaçāo

E := E

afectação

! E

desreferenciaçāo

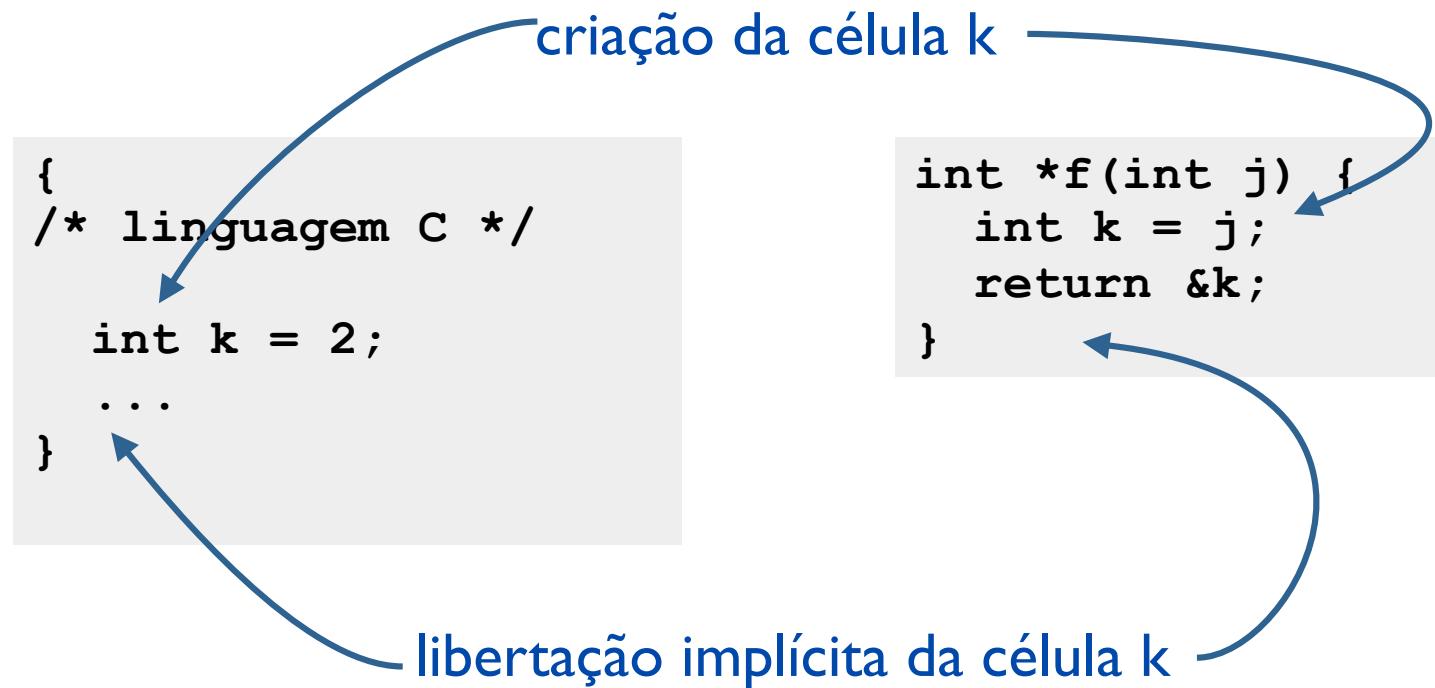
```
{  
/* linguagem C */  
  
int k = 2;  
int *a = &k;  
... k = k+*a ...  
  
}
```

```
decl  
  k = var(2)  
  a = var(k)  
in  
  ... k := !k+!a ...  
  free(k);  
  free(a);  
end
```

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando `var(_)` e a sua libertação usando `free(_)`.

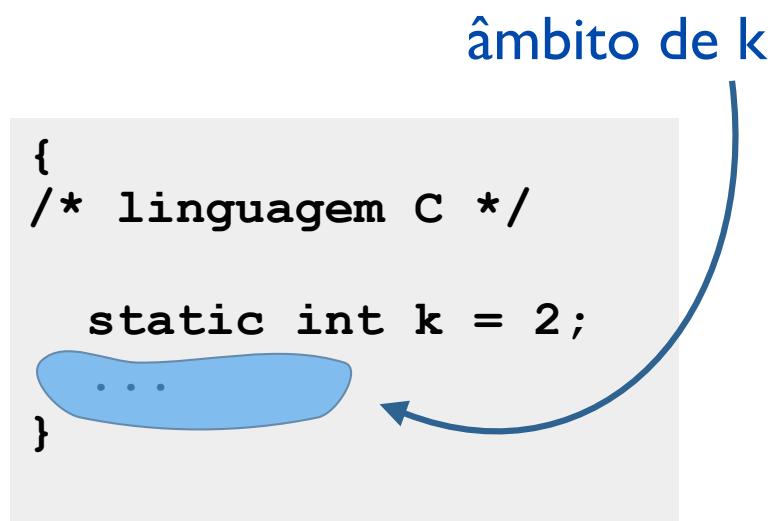
- Em muitas situações, o tempo de vida da célula **coincide** com o âmbito do(s) seu(s) identificador.



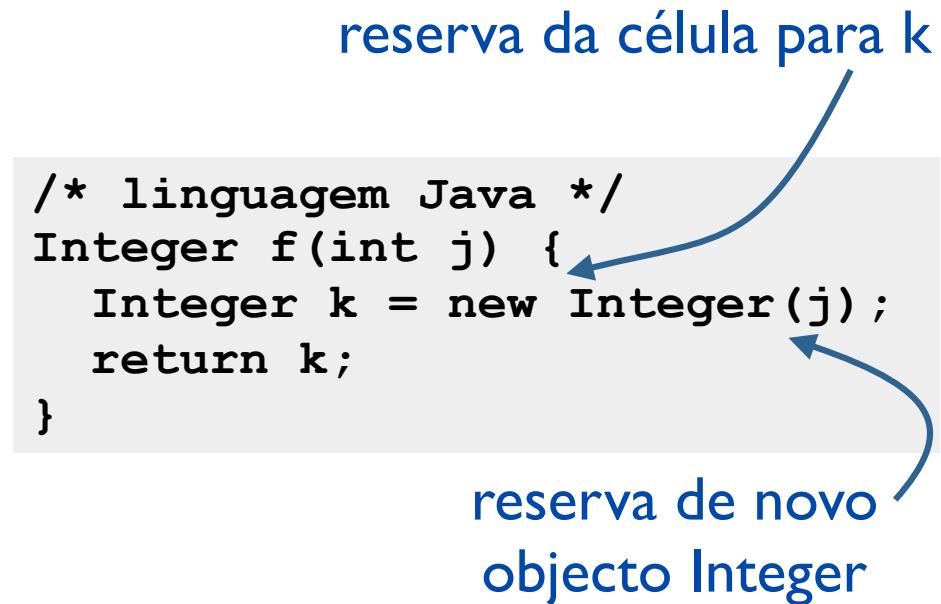
Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula **extravasa** o âmbito do(s) seu(s) identificador.



O tempo de vida da célula associada a k é o tempo do programa



há libertação implícita da célula de k mas o objecto sobrevive ao bloco!

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **var(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula **extravasa** o âmbito do(s) seu(s) identificador.

âmbito de k

```
{  
/* linguagem C */  
  
static int k = 2;  
...}
```

O tempo de vida da
célula associada a k é o
tempo do programa

reserva da célula para k

```
/* linguagem C */  
int* f(int j) {  
    int *k = malloc(sizeof(int));  
    *k = 2;  
    return k;  
}
```

reserva de novo
bloco de memória

há libertação implícita da célula de k
mas a memória reservada predura.

Linguagens Imperativas

Linguagens da família do ALGOL (Pascal, C, ...)

Assumem como princípio de desenho uma separação muito clara, logo ao nível sintático, entre expressões e comandos

- Expressões:
 - Denotam valores puros (inteiros, booleanos, funções)
 - A avaliação de expressões não deve ter efeitos (laterais)
- Comandos:
 - Denotam efeitos (na memória)
 - Um comando é executado pelo efeito que produz na memória: representa uma **acção**.

Uma linguagem de tipo ALGOL

- Definida com base em duas categorias sintácticas: Expressões (EXP) e comandos (COM):

| | |
|----------------|--|
| num: | $\text{Integer} \rightarrow \text{EXP}$ |
| bool: | $\text{Boolean} \rightarrow \text{EXP}$ |
| id: | $\text{String} \rightarrow \text{EXP}$ |
| add: | $\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$ |
| and: | $\text{EXP} \times \text{EXP} \rightarrow \text{EXP}$ |
| if: | $\text{EXP} \times \text{COM} \times \text{COM} \rightarrow \text{COM}$ |
| while: | $\text{EXP} \times \text{COM} \rightarrow \text{COM}$ |
| assign: | $\text{EXP} \times \text{EXP} \rightarrow \text{COM}$ |
| seq: | $\text{COM} \times \text{COM} \rightarrow \text{COM}$ |
| var: | $\text{String} \times \text{EXP} \times \text{COM} \rightarrow \text{COM}$ |
| const: | $\text{String} \times \text{EXP} \times \text{COM} \rightarrow \text{COM}$ |

Linguagens Imperativas

Linguagens da família do ML

Todas as construções pertencem a uma única categoria sintática, de expressões.

- Nestas linguagens, qualquer expressão pode potencialmente produzir um efeito lateral...
- Por exemplo, em OCAML a afectação `x := E` é uma expressão (de tipo **unit** (a.k.a. **void**)).
- N.B. Existem linguagens que combinam conceitos! Por exemplo, a linguagem C, contém expressões e comandos: a afectacão (`x = y`) é uma expressão, e expressões podem produzir efeitos (`i++`).

Uma linguagem tipo ML (microML)

- Consideramos uma só categoria sintáctica para expressões (EXP):

num: Integer → EXP

bool: Boolean → EXP

id: String → EXP

add: EXP × EXP → EXP

var: EXP → EXP

deref: EXP → EXP (!x)

if: EXP × EXP × EXP → EXP

while: EXP × EXP → EXP

assign: EXP × EXP → EXP (x := y + z)

seq: EXP × EXP → EXP (S1 ; S2)

decl: String × EXP × EXP → EXP

Semântica de microML

A semântica de uma linguagem imperativa pode ser caracterizada por uma função I que dá uma denotação a todos os programas abertos dado um ambiente e uma memória.

$$I : P \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

P = Fragmentos de programa (abertos)

ENV = Ambientes (funções $\text{ID} \rightarrow \text{VAL}$)

MEM = Memórias

VAL = Valores (Denotações)

O conjunto das denotações possíveis:

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

Esta função traduz a intuição que, em geral, um fragmento de programa P produz um **valor** e gera um **efeito** (na memória).

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

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

```
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) ]
```

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

eval : microML × ENV × MEM → VAL × MEM

```
eval( var(E) , env , m0) ≡ [ (v1 , m1) = eval( E, env, m0 );
                                (ref, m2) = m1.new(v1);
                                (ref, 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) ;
                                         (v2 , m3) ]
```

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

eval : microML × ENV × MEM → VAL × MEM

eval(seq(E1, E2) , env , m0) ≡ [(v1 , m1) = **eval(E1, env, m0);**

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

(v2 , m2)]

eval(if(E1, E2, E3) , env , m0) ≡

[(v1 , m1) = eval(E1, env, m0);

if (v1 = T) then (v2 , m2) = eval(E2, env, m1);

else (v2 , m2) = eval(E3, env, m1);

(v2 , m2)]

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

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

```
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) ]
```

iteração interpretada em
termos de recursão.

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

eval : microML × ENV × MEM → VAL × MEM

```
eval( decl(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) ]
```

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

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

```
decl a = var(2) in
decl b = var(!a) in
decl c = a in
(
    a := !b + 2;
    c := !c + 2
)
```

a e c são aliases (sinónimos), ou seja,
referem a mesma célula de memória.

Interpreting microML in Java

- Isolate the management of memory in an abstract data type
- Allows for garbage collection algorithms (if needed)

```
interface RefValue
    extends IValue { }

class MemoryCell
    implements RefValue {
    IValue value;
    MemoryCell(IValue value) {
        this.value = value;
    }
}

interface MemoryManagement {
    RefValue new(IValue value);
    IValue get(RefValue);
    IValue set(RefValue ref, IValue value);
    void free(RefValue);
}
```

Interpreting microML in Java

- Isolate the management of memory in an abstract data type
- Allows for garbage collection algorithms (if needed)
- Simple: do nothing, sophisticated: accelerate, check, etc.

```
class MemoryImpl implements MemoryManagement {  
    RefValue new(IValue value) {  
        return new MemoryCell(value);  
    }  
    IValue get(RefValue ref) {  
        return ((MemoryCell) ref).value;  
    }  
    IValue set(RefValue ref, IValue value) {  
        return ((MemoryCell) ref).value = value;  
    }  
    void free(RefValue) {}  
}
```

Interpretação e Compilação de Linguagens de Programação

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

2016-2017

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 06(b)

(Recap) Type Systems

Unidade 6: Sistemas de tipos

Os sistemas de tipos são ferramentas de análise estática que garantem boas propriedades de programas concretos. A análise estática é uma forma de interpretação “abstracta” de programas. A principal característica da análise estática é que termina sempre, mesmo para programas que não terminam.

A forma mais comum de análise estática é a verificação de tipos (**type checking**). Os sistemas de tipos garantem a ausência de certos tipos de erros durante a execução.

- Erros de execução
- Interpretação abstracta
- Sistemas de tipos
- Consistência de um sistema de tipos
- Detecção de erros de execução

Runtime errors...

- What can go wrong here?

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

Runtime errors...

- What can go wrong here?

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

Runtime errors...

- What can go wrong here?

eval(add(E1, E2) , env , m0) \triangleq [(v1 , m1) = eval(E1, env, m0);

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

(v1 + v2 , m2)]

eval(deref(E) , env , m0) \triangleq [(ref , m1) = eval(E, env, m0);

(m1.get(ref) , m1)]

eval(assign(E1, E2) , env , m0) \triangleq [(v1 , m1) = eval(E1, env, m0);

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

m3 = m2.set(v1, v2) ;

(v1 , m3)]

Impossible to compute in advance all kinds of values that may result from the evaluation of an expression.

Let's interpret the values of miniML using their type representatives.

(Recall) Semantic functions

- Function **eval** computes the value of an expression:

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

- Function **comp** computes a target program in a low-level language, that is behaviour equivalent to a given expression.

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

- Function **typecheck** computes the type of an expression:

$$\text{typecheck} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$$
$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$
$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$

Types for microML

- Function **typecheck** computes the type of an expression:

$\text{typecheck} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$

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

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

- Function **typecheck** is an interpreter that evaluates expressions according to a similar but more abstract semantics (where the denotations are “types”).
- Types of operations are defined based on the types of their components.

Types for microML

- Function **typecheck** computes the type of an expression:

typecheck : microML × ENV → TYPE

ENV : ID → TYPE

TYPE = { int, bool, ref{TYPE}, none }

int: the type of integer values

bool: the type of boolean values

ref{T}: the type of references to memory cells containing values of type $\textcolor{red}{T}$.

Example: **ref{ref{int}}** is the type of the references to memory cells that can only contain references to cells that contain integer values.

none: is the type for expressions (programs) whose denotations in function **eval** and **comp** may be undefined.

(Recall) Type checking

- **typecheck** computes the type of any expression:

typecheck : BCALC → {int, bool, none}

```
typecheck( add(E1, E2) ) ≡      [ t1 = typecheck ( E1 )
                                         t2 = typecheck ( E2 )
                                         if (t1 == int ) and (t2 == int )
                                         then int
                                         else none ]
```

(Recall) Type checking

- Function **typecheck** computes the type of any expression:

typecheck : microML × ENV → TYPE

```
typecheck( num(n) , env) ≡ int ;  
typecheck( id(s) , env)      ≡ env.Find(s) ;  
typecheck( true , env)        ≡ bool ;  
typecheck( false , env)       ≡ bool ;
```

Type checking

- Function **typecheck** computes the type of any expression:

typecheck : microML \times ENV \rightarrow TYPE

typecheck(var(E) , env) \triangleq [*t* = **typecheck(E, env)**; **ref{t}**]

Type checking

- Function **typecheck** computes the type of any expression:

typecheck : microML × ENV → TYPE

typecheck(deref(E) , *env*) \triangleq [ref{*t*} = **typecheck(E, *env*); *t***]

Type checking

- Function **typecheck** computes the type of any expression:

typecheck : microML × ENV → TYPE

```
typecheck( assign(E1, E2) , env ) ≡  
    [ t1 = typecheck( E1, env );  
      t2 = typecheck( E2, env );  
      if (t1 == ref{t2})  
        then t2;  
      else none ; ]
```

Type checking

- Function **typecheck** computes the type of any expression:

typecheck : microML \times ENV \rightarrow TYPE

```
typecheck( if(E1, E2, E3) , env ) ≡  
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool ) then none  
    else [ t2 = typecheck( E2, env );  
           t3 = typecheck( E3, env );  
           if (t2 == none ) or (t3 == none ) or (t2 != t3)  
             then none  
           else t2 ] ]
```

Type checking

- Compare to the **eval** case for **if**

Branches E2 and E3 are **both** analysed and there is the constraint that $t2 == t3$. [Why?]

```
typecheck( if(E1, E2, E3) , env ) ≡  
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool ) then none  
    else [ t2 = typecheck( E2, env );  
           t3 = typecheck( E3, env );  
           if (t2 == none ) or (t3 == none ) or (t2 != t3)  
             then none  
           else t2 ] ]
```

Type checking

- Function **typecheck** computes the type of any expression:

typecheck : microML × ENV → TYPE

```
typecheck( while(E1, E2) , env ) ≡  
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool) then none  
    else [ t2 = typecheck( E2, env );  
          if (t2 == none ) then none  
          else bool ] ]
```

Type checking

- Compare with the **eval** case for **while**.

The subexpression E2 is analysed exactly **onde**. [Why?
The final type is of type **bool**. [Why?]

```
typecheck( while(E1, E2) , env ) ≡  
  [ t1 = typecheck( E1, env );  
    if ( t1 != bool) then none  
    else [ t2 = typecheck( E2, env );  
      if (t2 == none ) then none  
      else bool ] ]
```

Quiz

1. Is this program well typed? if not, present a correct version.
2. What's the typing environment for `!y` ?
3. What's the result/effect of the program?

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

Interpretação e Compilação de Linguagens de Programação

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

2016-2017

João Costa Seco (joao.seco@di.fct.unl.pt)

Lecture 07

Compilação dirigida por informação de tipos

Type-based compilation

Static typing is a deterministic and terminating process which is integrated in the compilation phase. It allows to understand the real nature of the values manipulated by a program.

It allows the generation of specialised, and more efficient, target code. Also, by preserving type information in low-level languages, it allows for extra safety on dynamic loading systems. (e.g. LLVM, JVM and CLR bytecode).

- Type annotations in the AST
- Specialized generated code
- Low-Level verification of code

Compiling to a typed assembly language

- Modern intermediate languages (LLVM, CLR and JVM bytecode) are typed.
- Low-level typing allows:
 - Load time verification.
 - Runtime validation of operations (soft crash of the virtual machine)
 - Optimizations: code specialization

aload
iload
aaload
caload
dload

decl x = 1 **in**
decl y = 2 **in**
 x + y
end
end



```
new frame_1
dup
invokespecial frame_1/<init>()V
dup
sipush 1
putfield frame_1/loc_00 I
astore 0
new frame_2
dup
invokespecial frame_2/<init>()V
dup
aload 0
putfield frame_2/SL Lframe_1;
dup
sipush 2
putfield frame_2/loc_00 I
astore 0
aload 0
checkcast frame_2
getfield frame_2/SL Lframe_1;
getfield frame_1/loc_00 I
aload 0
checkcast frame_2
getfield frame_2/loc_00 I
iadd
aload 0
checkcast frame_2
getfield frame_2/SL Lframe_1;
astore 0
aconst_null
astore 0
```

Compiling heap variables to a typed assembly language (arrays, objects)

- To implement memory cells with a lifetime larger than their creation scope we need to allocate memory in the heap.
- Each cell is correctly typed to store values of a single type.
- The type of each node must be pre-determined.

Compiling heap variables to a typed assembly language (arrays, objects)

decl x = **var(1)** in **!x**

```
new frame_1
dup
invokespecial frame_1/<init>()V
dup
new ref_int
dup
invokespecial ref_int/<init>()V
dup
sipush 1
putfield ref_int/value I
putfield frame_1/loc_00 Lref_int;
astore 0
aload 0
checkcast frame_1
getfield frame_1/loc_00 Lref_int;
getfield ref_int/value I
aconst_null
astore 0
```

Compiling heap variables to a typed assembly language (arrays, objects)

decl x = **var(1)** in **!x**

```
.source frame_1.j
.class frame_1
.super java/lang/Object
.implements frame

.field public loc_00 Lref_int; ; x

.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
```

Compiling heap variables to a typed assembly language (arrays, objects)

decl x = **var(1)** in **!x**

```
.source ref_int.j
.class ref_int
.super java/lang/Object

.field public value I

.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
```

Compiling heap variables to a typed assembly language (arrays, objects)

decl x = **var(var(1))** in **!!x**

```
.source ref_ref_int.j
.class ref_ref_int
.super java/lang/Object

.field public value Lref_int;

.method public <init>()V
aload_0
invokespecial java/lang/Object/<init>()V
return
.end method
```

Compiling heap variables to a typed assembly language (arrays, objects)

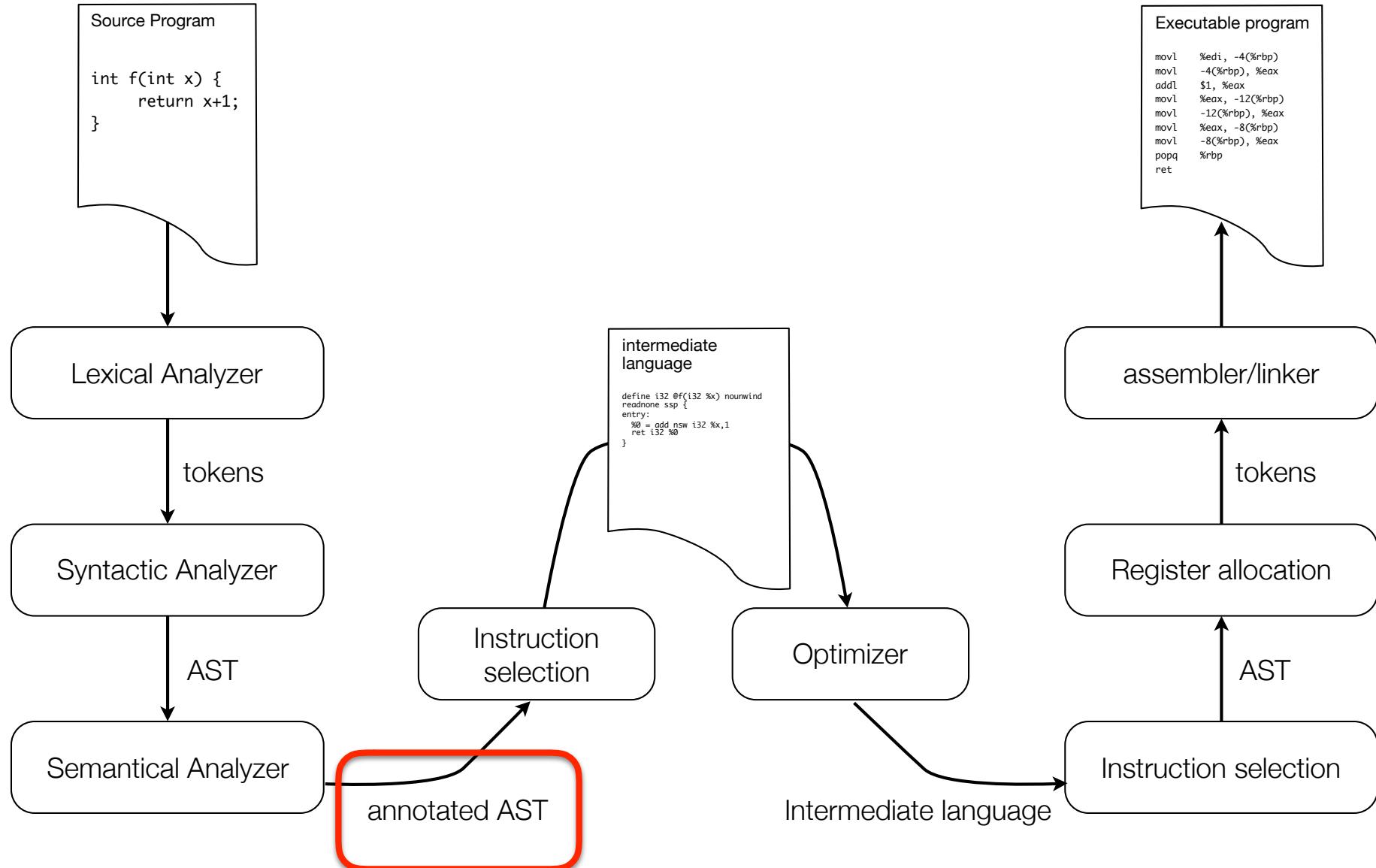
decl x = **var(var(1))** in **!!x**

```
new frame_1
dup
invokespecial frame_1/<init>()V
dup
new ref_ref_int
dup
invokespecial ref_ref_int/<init>()V
new ref_int
dup
invokespecial ref_int/<init>()V
dup
sipush 1
putfield ref_int/value I
putfield ref_ref_int/value Lref_int;
putfield frame_1/loc_00 Lref_ref_int;
astore 0
aload 0
checkcast frame_1
getfield frame_1/loc_00 Lref_ref_int;
getfield ref_ref_int/value Lref_int;
getfield ref_int/value I
acconst_null
astore ICE 2011-2018 0
```

Annotated AST

- The type system is independent from the code generation process. Its results can be seen as an input to the compilation process.
- The AST can be imperatively annotated with all intermediate results of the type system, which can then be used by the code generation function.

(Recall) Architecture of a compiler



Annotated AST

- The type system is independent from the code generation process. Its results can be seen as an input to the compilation process.
- The AST can be imperatively annotated with all intermediate results of the type system, which can then be used by the code generation function.

```
public class ASTNum implements ASTNode {  
  
    private int value ;  
  
    Num(int v) { value = v; }  
  
    IValue eval(Env<Value> env) { return value; }  
  
    IType typecheck(Env<IType> env) { return IntType.singleton; }  
  
    IType getType() { return IntType.singleton; }  
  
}
```

Types in Java

- The use of patterns (like the singleton) allows for more efficient implementations of the type system (the singleton pattern allows the use of `==` instead of reflection and `equals`).

```
public interface IType {}  
  
public class IntType implements IType {  
  
    private IntType();  
  
    public singleton = new IntType();  
  
}  
  
public class RefType implements IType {  
  
    private IType type;  
  
    public RefType(IType type) { this.type = type; }  
  
    public Type getType() { return type; }  
  
}
```

Annotated AST

- The type system is independent from the code generation process. Its results can be seen as an input to the compilation process.
- The AST can be imperatively annotated with all intermediate results of the type system, which can then be used by the code generation function.

```
public class ASTAdd implements ASTNode {  
  
    private int value ;  
  
    private IType type;  
  
    Num(int v) { value = v; }  
  
    IValue eval(Env<Value> env) { return value; }  
  
    IType typecheck(Env<IType> env) { ... type = IntType.singleton }  
  
    IType getType() { return type; }  
}
```

Challenge

- How to compile an imperative language with the following variante for expressions:

`var a = 1 in E end` `a := E` `a`

- The lifetime of variable `a` is expression `E`, its uses are implicitly dereferenced.
- Hint: To use methods' local variables to store local variables (JVM).