Interpretação e Compilação de Linguagens (de Programação) Part 2: Basic Expressions 21/22

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

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Operational Semantics

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

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

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

How do we define the semantics of a language?

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

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

The CALC Language (arithmetic expressions)

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

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

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

Abstract Syntax vs. Concrete Syntax (examples)

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

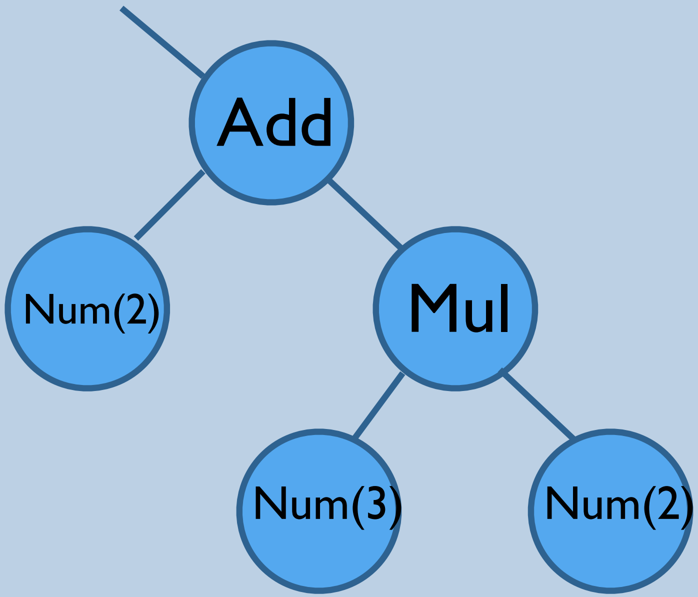
Abstract Syntax vs. Concrete Syntax (examples)

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

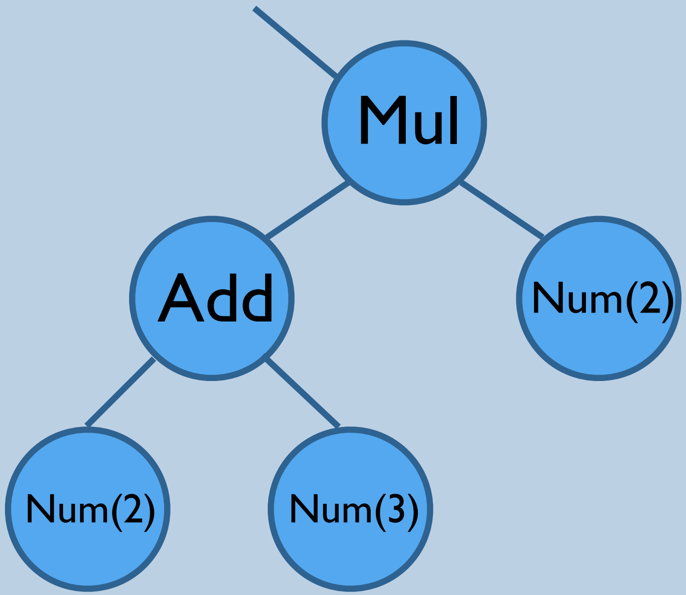
Abstract Syntax Tree

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

`2 + 3 * 2`



`(2 + 3)* 2`



Semantics of CALC

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

$$\mathrm{I} : \mathrm{CALC} \rightarrow \mathrm{DENOT}$$

CALC = set of all programs (as syntactical structures)

DENOT = set of all meanings (denotations)

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

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

CALC = set of all programs (as syntactical structures)

DENOT = Integer = set of all meanings (denotations)

CALC Interpreter (evaluation map)

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

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

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

CALC Interpreter (evaluation map)

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

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

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

CALC AST as an (OCAML) inductive data type

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

CALC eval map as an (OCAML) recursive function

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

Structural Operational Semantics (Plotkin81)

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

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

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

eval(**mul(E1,E2)**) ≜ eval(**E1**) * eval(**E2**)

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

Java Implementation

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

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

This represents the map eval: CALC → Integer

Java Implementation

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

$$\mathrm{eval}(\mathrm{num}(n)) \triangleq n$$

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

Java Implementation

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

$$eval(add(e1,e2)) \triangleq eval(e1)+eval(e2)$$

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

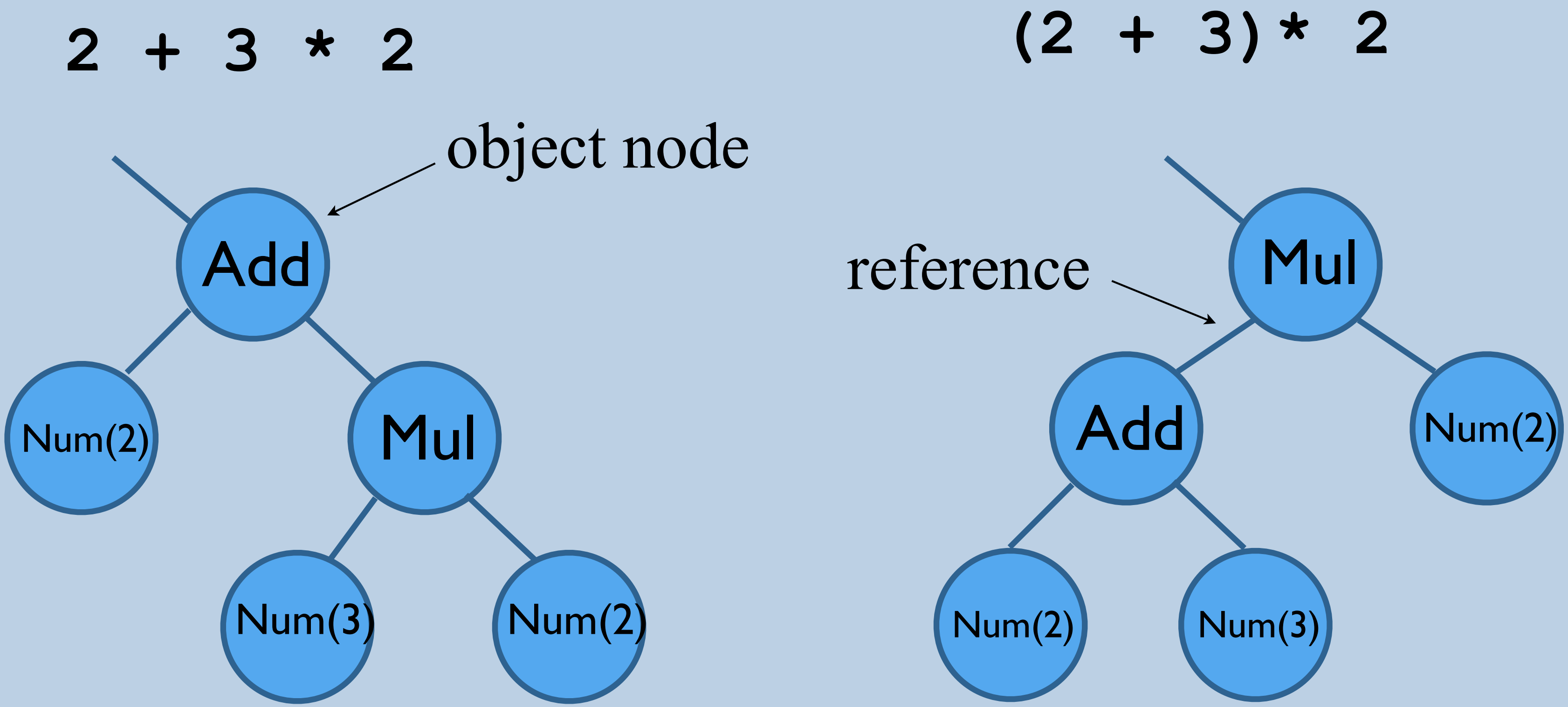
Java Implementation

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

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

Abstract Syntax Tree

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



Java Implementation

- In our example, classes **`ASTNum`**, **`ASTAdd`**, **`ASTSub`**, **`ASTMul`** e **`ASTDiv`**, implement the language construtors **num**, **add**, **sub**, **mul** and **div**.
- The definition of any function f: CALC→ V becomes dispersed by the several classes of the abstract syntax.
- This makes it easier to add new constructs to the language, we just need to add a new AST class type and the associated implementation of the evaluation maps.
- ```
  public class ASTNeg implements CALC {
  private CALC exp;
  ASTNeg(CALC e) { exp = e;}
  int eval() { return -exp.eval(); }
  }
  ```
  ```

Compilers

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

$$\mathrm{evalt}(\mathrm{comp}(P)) = \mathrm{evals}(P)$$

$$\mathrm{evalt}: T \rightarrow V$$

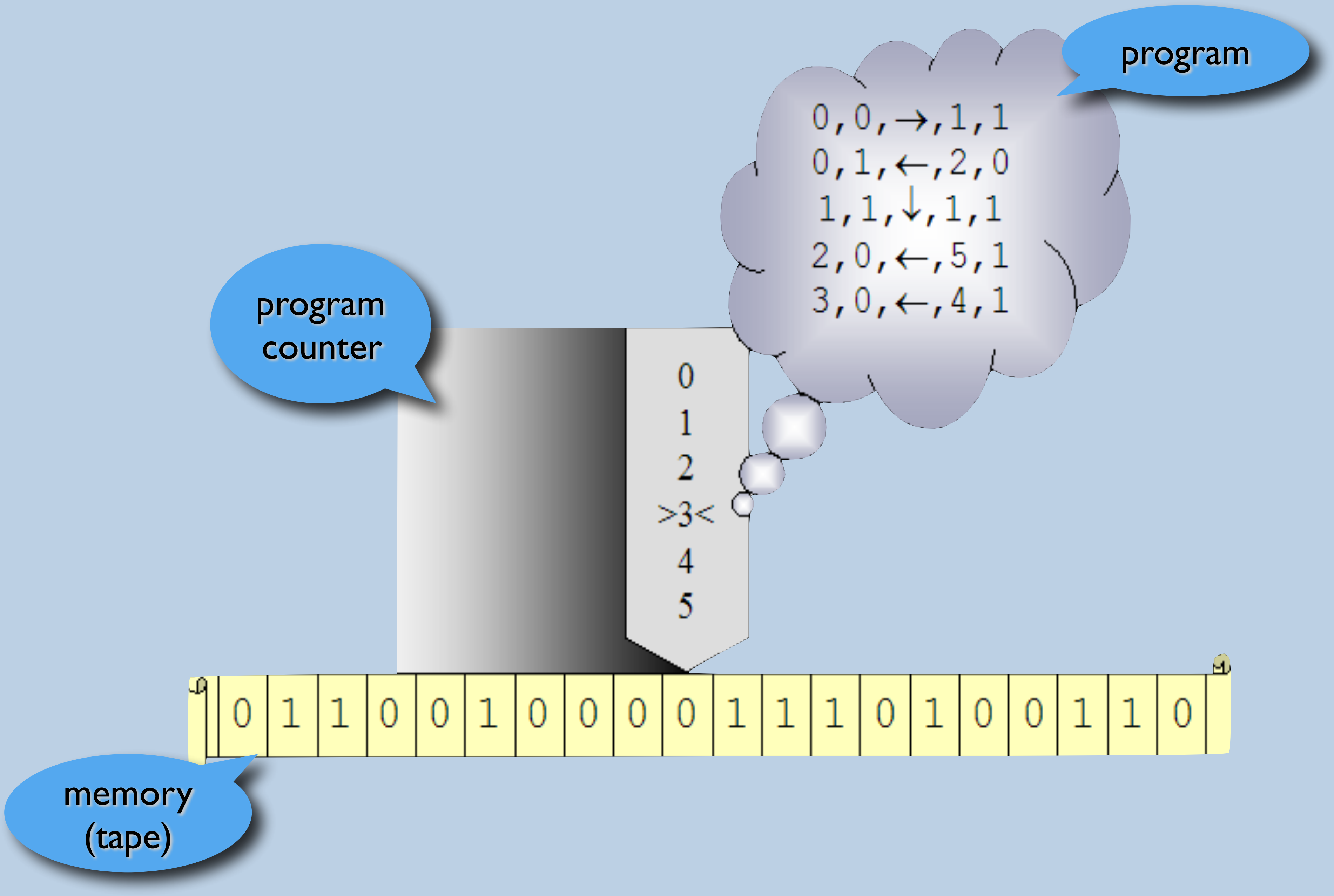
$$\mathrm{evals}: S \rightarrow V$$

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

Virtual Machines (software processors)

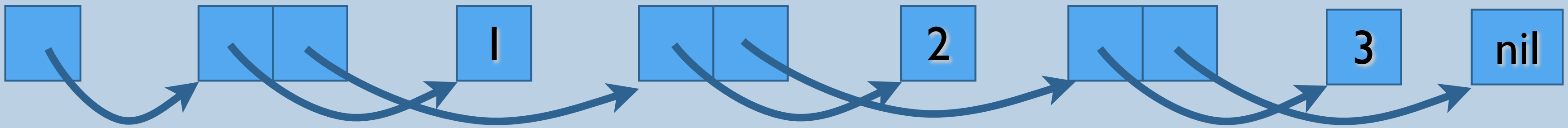
- Turing Machine (Turing, 1931)
  - The first one …
- SECD (Landin, 1962)
  - Stack, Environment, Code, Dump
- P-code machine (Wirth 1972)
  - Stack machine, Pascal p-code compiler
- JVM (Sun, 1995)
  - Type safe, Multi-threaded, focusing on the Java Language
- CLR (Microsoft, 2000)
  - Type safe, Multi-threaded,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

Turing Machine (Turing, 1931)



SECD - machine (Landin, 1962)

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



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

P-code machine (Wirth 1972)

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

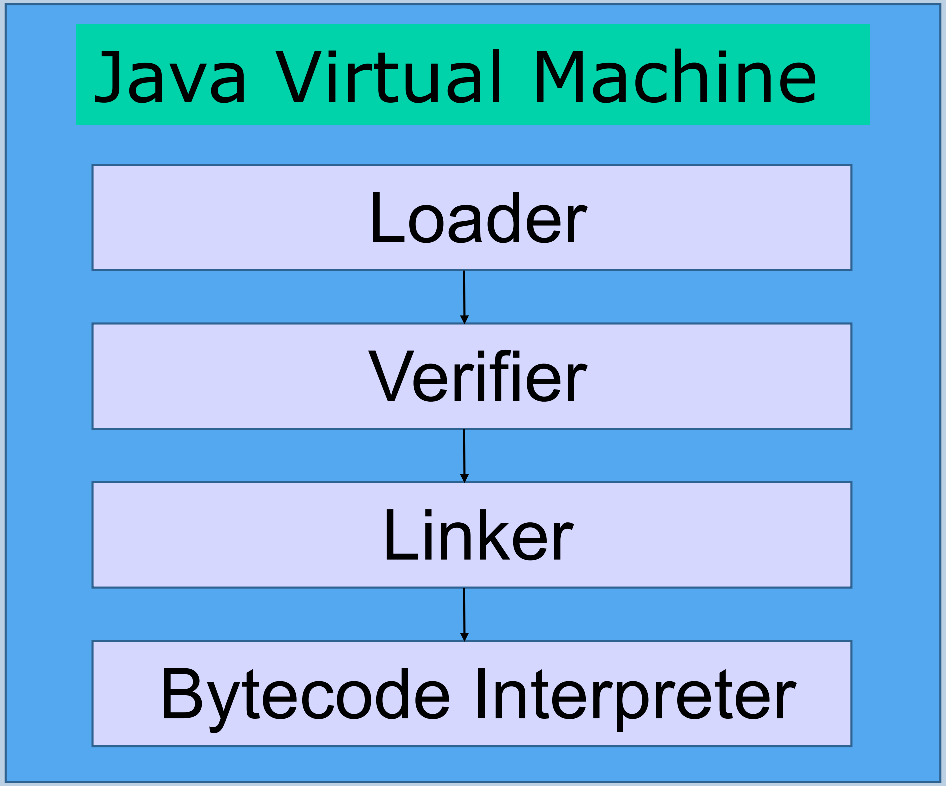
```
procedure interpret;
   const stacksize = 500;

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

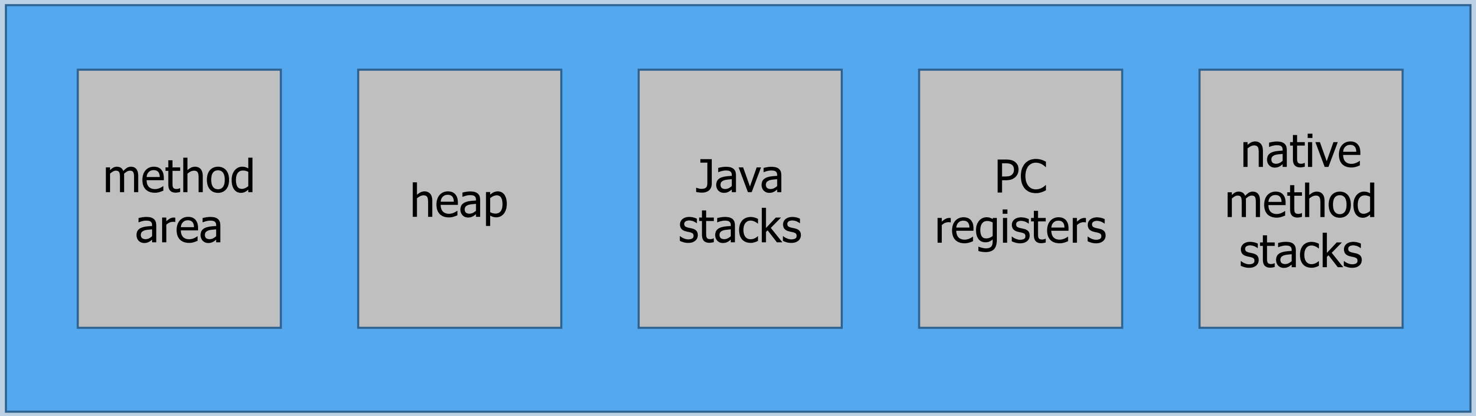
   function base(l: integer): integer;
      var b1: integer;
   begin
      b1 := b; {find base l levels down}
      while l > 0 do
         begin b1 := s[b1]; l := l - 1
         end;
      base := b1
   end {base};

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

Java Virtual Machine (Sun, 1995)

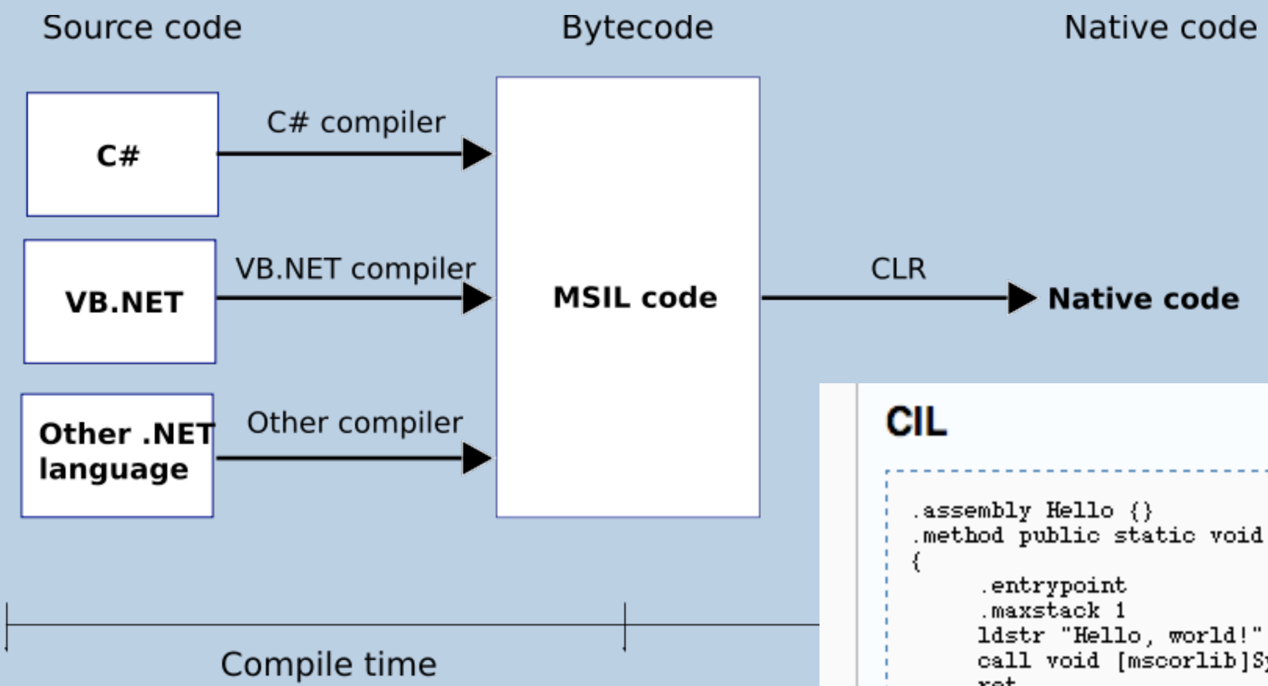



```
// Bytecode stream: 03 3b 84
// 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
iconst_0 // 03
istore_0 // 3b
iinc 0, 1 // 84 00 01
iload_0 // 1a
iconst_2 // 05
imul // 68
istore_0 // 3b
goto -7 // a7 ff f9
```



Common Language Runtime (Microsoft, 2000)

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



CIL

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

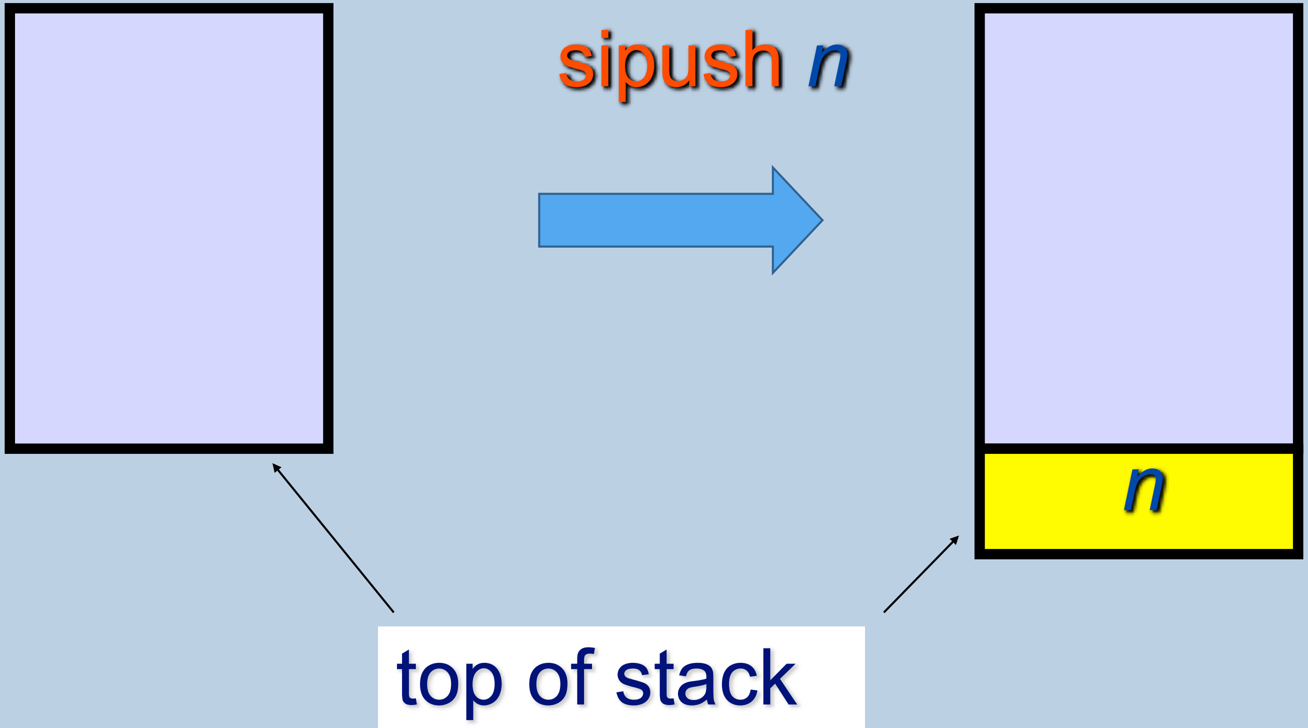
Java Virtual Machine

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

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

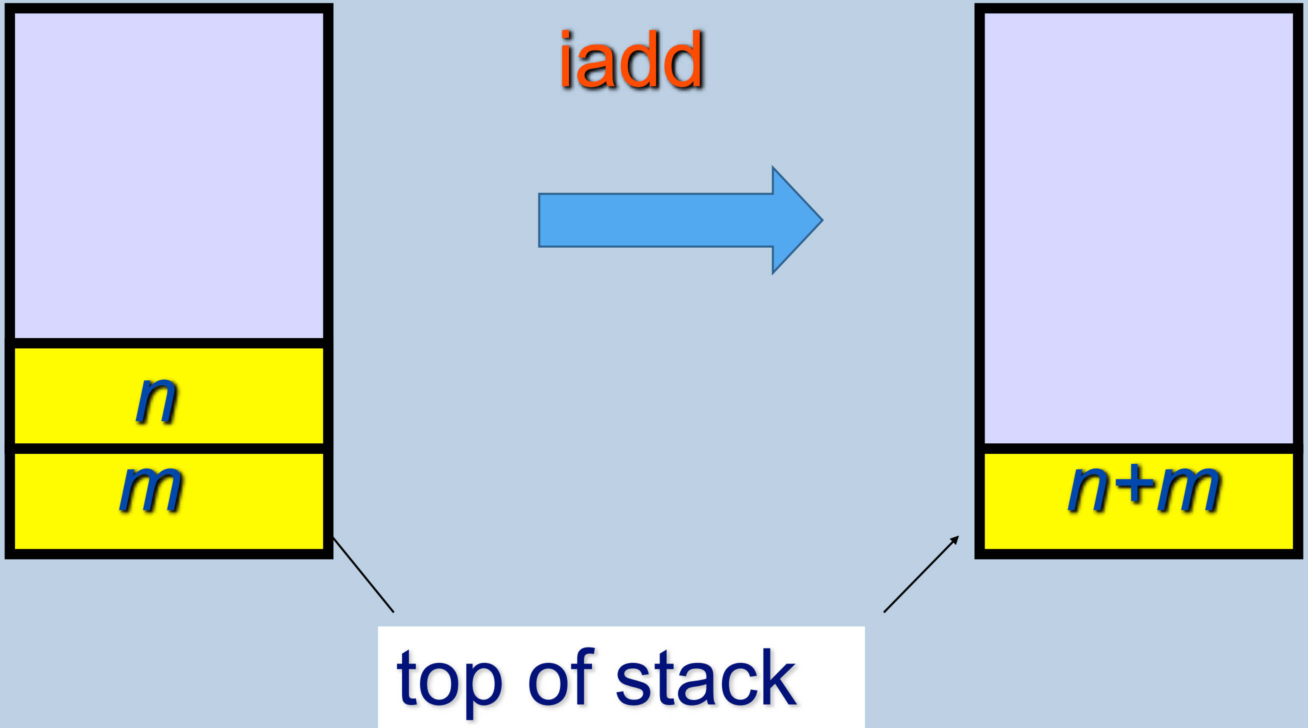
JVM instructions

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



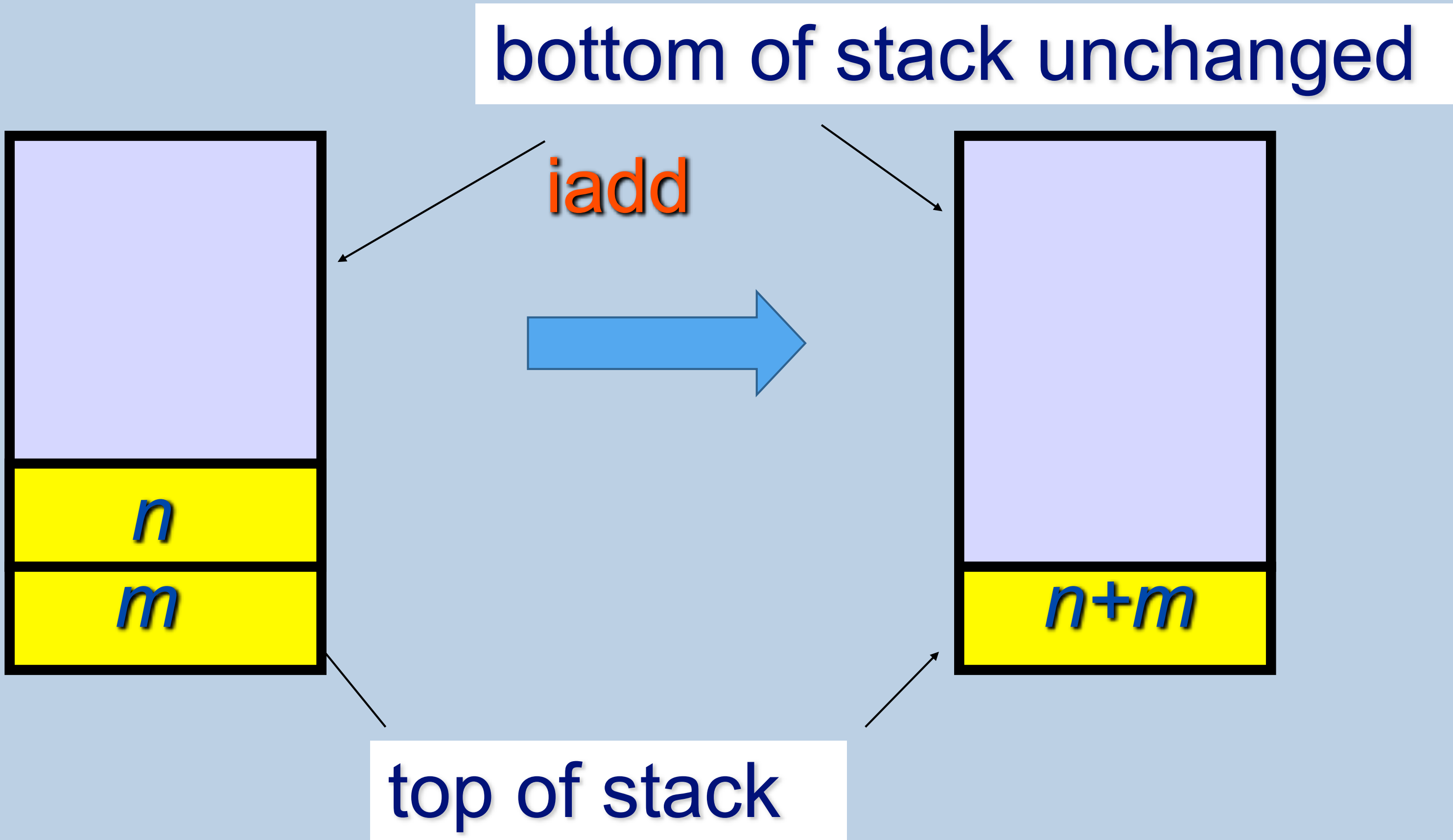
JVM instructions

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



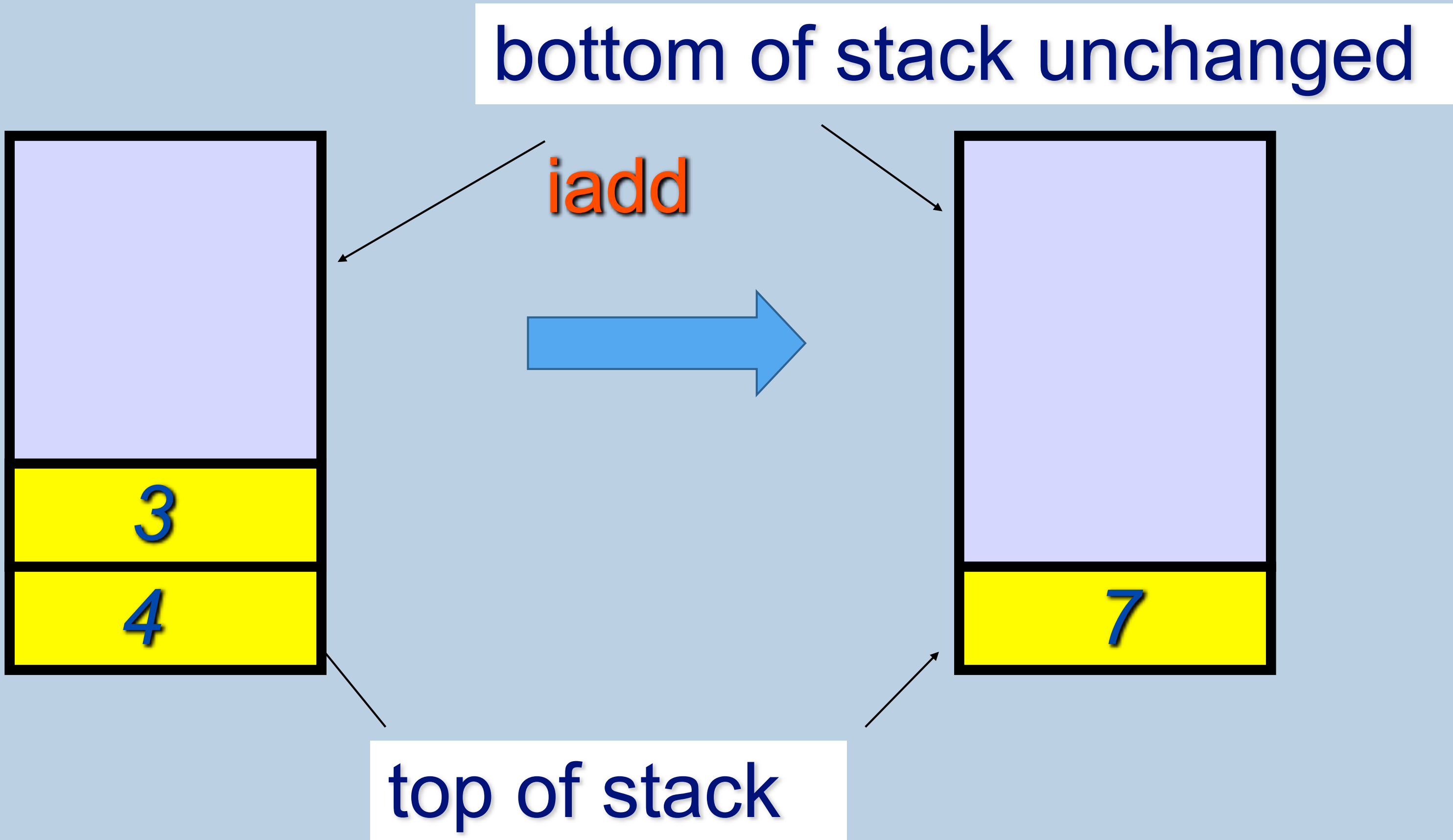
JVM instructions

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



JVM instructions

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



CALC compiler (compilation map)

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

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

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

CALC compiler (compilation map)

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

comp : CALC → CodeSeq

comp(**num**(n)) ≜ < **ldc.i4 n** >

comp(**add**(E’,E”)) ≜ comp(E’) @ comp(E”) @ < **add** >

comp(**mul**(E’,E”)) ≜ comp(E’) @ comp(E”) @ < **mul** >

comp(**sub**(E’,E”)) ≜ comp(E’) @ comp(E”) @ < **sub** >

comp(**div**(E’,E”)) ≜ comp(E’) @ comp(E”) @ < **div** >

CALC compiler (compilation map)

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

comp : CALC → CodeSeq

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

OCAML code

Compiler correction property

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

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

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

CALC compiler (compilation map)

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

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

CALC compiler (compilation map)

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

Java Virtual Machine (Sun, 1995)

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