Theme 2. Foundations of Programming Languages

Programming Languages, Technologies and Paradigms (LTP)

DSIC, ETSInf

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

etsinf
Escuela Técnica Superior de Ingeniería Informática

Formal description of a PL

- **Syntax**: which character sequences form a “legal” program:
  - Syntactic elements of the language
- **Semantics**: what is the (computational) meaning of a given legal program. Relevance:
  1. helpful to reason about the program
  2. we need it to appropriately implement the language (execution models)
  3. essential to develop techniques and tools for:
     - Analysis and Optimization
     - Debugging
     - Verification
     - Transformation

Syntax

Use of BNF grammars

BNF Notation:

- With <w> we give name to a group of expressions which is defined by means of some rules
- symbol | means “or”

```
<letter>  ::=  a | b | c | d | A | B | C | D
 <digit>  ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    <id>  ::=  <letter> | <id><letter> | <id><digit>
```

- square brackets [and] enclose optional items
- curly brackets {} (or a star *) denote a sequence of 0 or more items
- symbol + denotes a sequence of 1 or more items

```
<IdNum>  ::=  <letter>+ {<digit>}
```

Syntax

Use of BNF grammars

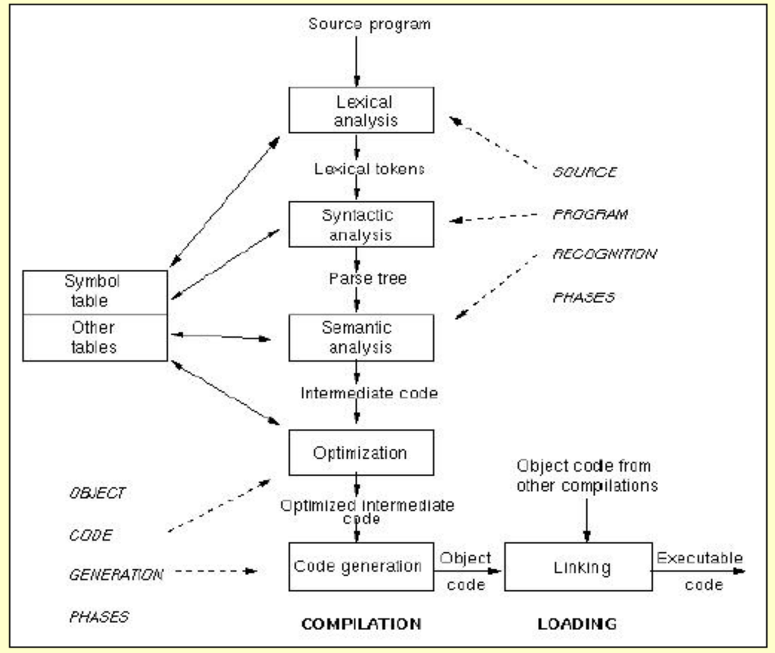
Example: syntax of while loops

Java

<while_statement> ::= `while ( `<expression>` ) `<statement>

Modula-2

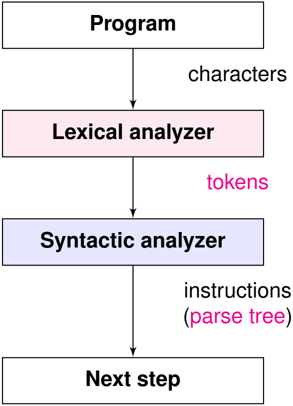
<while_statement> ::= `WHILE` <expression> `DO`
<statement> { `;` <statement> }
`END`



Processing a source program

Scanning and parsing

- The lexical analyzer (scanner) decomposes a sequence of characters (the program) into a sequence of primitive syntactic components, the so-called tokens (identifiers, numbers, PL keywords, etc.)
- The syntactic analizer (parser) recognizes a sequence of tokens and builds a sequence of instructions (which is actually presented as a syntactic tree: the parse tree)



Example

1. Sequence of characters (a string)

```
f,u,n,{,F,a,c,t,' ',N,},\n,' ',i,f,' ',N, =,=,0, ' ', t,h,e,n,
' ',1,\n,' ',e,l,s,e,' ',N,*,{,F,a, c,t,' ',N, -,1,},'
',e,n,d,i,f,\n,e,n,d
```

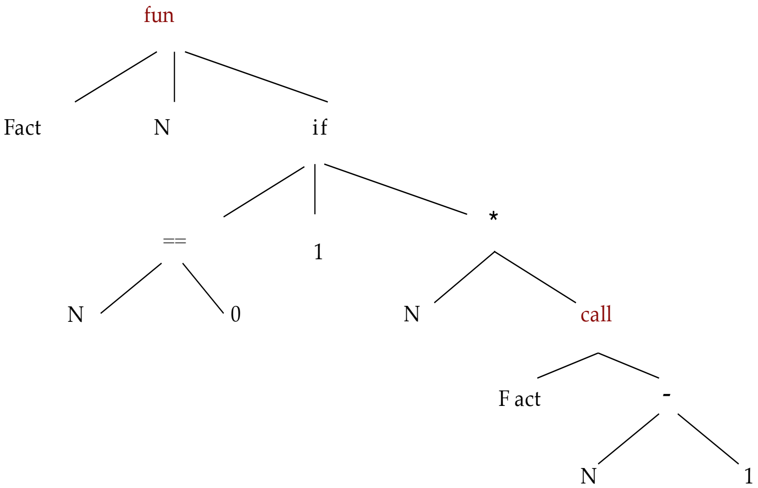
2. Sequence of tokens

```
fun,{,Fact,N,},if,N,==,0,then,1,else,N,*,{,Fact,N,
-,1,},endif,end
```

3. Instruction

```
fun {Fact N}
   if N == 0 then 1 else N*{Fact N-1}
   endif
end
```

Example



Parse tree

Semantic analysis

Semantic description: what is needed for?

Static semantics: syntax restrictions that cannot be expressed by using BNF but can be checked in compilation time

Example

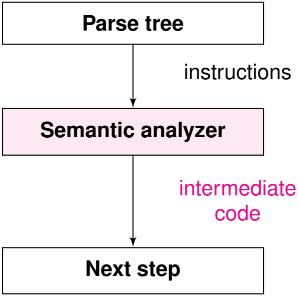
A := B + C could be illegal if A, B or C were not previously declared

Dynamic semantics: Restrictions that can only be checked during the execution of the program (runtime) (e.g. indexing an array within its limits)

Semantic analysis

Static semantics

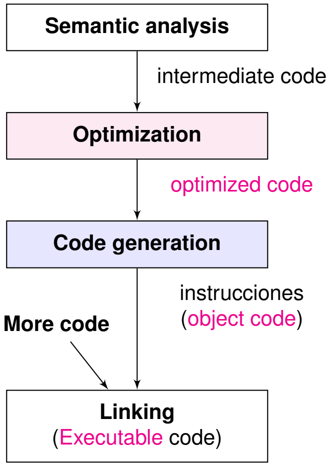
- Semantic analyzer checkings:
  1. Variables are declared before used
  2. Type compatibility and conversion (coercion)
  3. Function profile: actual and formal parameters coincide in number and type
  4. ...
- An intermediate code is produced as a basis for compilation and code generation



Compilation and Linking

Code generation

- First, the input intermediate code is optimized
- The code generation step produces the program object code
- The object code is linked to other code from other programs or libraries to obtain the executable code.



Example

Evolution of the internal representation of the following program[1] throughout the compilation process.

```
position = initial + speed * 60
```

where the type of `initial`, `position` and `speed` is real.

[1] Pages 12 and 13 of *Aho, Sethi and Ullman. Compiladores: Principios, técnicas y herramientas. Addison-Wesley Iberoamericana, 1990*.

Dynamic semantics

Why semantics is not always "static"'?

The compilador is unable to detect all possible errors:

1. Some errors only arise during the execution

   - Z=X/Y raises an error when executed with $Y = 0$
   - Z=V[Y] raises an error if the value of Y is out of range for vector V

2. Many (and most) interesting program properties are undecidable:

   - termination (but it is 'semidecidable': just execute the program to "semi-decide")
   - are two programs computing the same function (semantic equivalence)?
   - are two BNF descriptions generating the same language (syntactic equivalence)?

Dynamic semantics

Semantic definition styles

- **Operational**
- **Axiomatic**
- Declarative:
    - Denotational
    - Algebraic
    - Model theory
    - Fixpoint

Operational Semantics

Define an (abstract) machine M and give meaning to the program instructions in terms of the actions performed by the machine to execute each of such instructions.

Operational Semantics

- The state of the (abstract) machine that executes the program is represented as a mapping $s : \mathcal{X} \to D$ assigning values in a domain D to program variables $x, y, \ldots \in \mathcal{X}$.

Notation

Since programs use a finite set $\mathcal{X} = \{x_1, \ldots, x_n\}$ of variables, the state can be represented as a finite set of pairs variable-value:

$$s = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}.$$

- A machine configuration is a pair

$$\langle i, s \rangle$$

consisting of the current state (s) and the program instruction (i) to be evaluated, either simple or complex (a program is considered as a compound instruction).

Operational Semantics

- We formalize the program execution in the machine by means of a transition relation '$\rightarrow$' on configurations.
- The relation is defined by means of transition rules:

$$\frac{\mathit{premise}}{\langle i, s\rangle \rightarrow \langle i', s'\rangle} \tag{1}$$

  describing the configuration $\langle i', s'\rangle$ which is obtained from a given configuration $\langle i, s\rangle$ when the premise or condition on the configuration $\langle i, s\rangle$ is satisfied.
- We also use other relations to describe:
  - the evaluation of arithmetic expressions ($\langle exp, s\rangle \Rightarrow n$).
  - the direct computation of a final state ($\langle i, s\rangle \Downarrow s'$).

  and define them by using rules like (1).

The language SIMP

Syntax

A BNF-like grammar of the Small IMPerative language SIMP we will be using in the sequel is given as follows:

- Arithmetic expressions:

$$a ::= C \mid V \mid a_1{+}a_2 \mid a_1{-}a_2 \mid a_1{*}a_2$$

  where C and V denote the numeric constants $(0, 1, 2, \ldots)$ and variables $(x, y, \ldots)$ respectively

- Boolean expressions:

$$b ::= \texttt{true} \mid \texttt{false} \mid a_1{=}a_2 \mid a_1{\leq}a_2 \mid \neg b \mid b_1{\vee}b_2$$

- Instructions:

$$i ::= \mathit{skip} \mid V{:=}a \mid i_1;i_2 \mid \texttt{if}\ b\ \texttt{then}\ i_1\ \texttt{else}\ i_2 \mid \texttt{while}\ b\ \texttt{do}\ i$$

  where *skip* is the empty instruction.

The language SIMP

Evaluation of expressions

- By writing $\langle exp, s\rangle \Rightarrow n$ we mean that expression *exp* is evaluated to *n* in the state *s*.
- We use such kind of evaluation relation to evaluate both arithmetic and boolean expressions.

The language SIMP

Evaluation of arithmetic expressions

- Numeric constants:

$$\langle \mathrm{n}, s \rangle \Rightarrow n$$

- Variables:

$$\langle x, s \rangle \Rightarrow s(x)$$

Remind: a state s is a mapping from variables into values. $s(x)$ is just the value of variable x in the machine state s.

- Addition:

$$\frac{\langle a_1, s \rangle \Rightarrow n_1 \qquad \langle a_2, s \rangle \Rightarrow n_2}{\langle a_1 + a_2, s \rangle \Rightarrow n}$$

if n is the addition of n_1 and n_2.

- Subtraction and product: similar.

The language SIMP

Evaluation of boolean expressions

- Boolean values:

$$\langle \texttt{false}, s\rangle \Rightarrow \mathit{false} \qquad \langle \texttt{true}, s\rangle \Rightarrow \mathit{true}$$

- Equality:

$$\frac{\langle a_1, s\rangle \Rightarrow n_1 \qquad \langle a_2, s\rangle \Rightarrow n_2}{\langle a_1 = a_2, s\rangle \Rightarrow \mathit{true}} \quad \text{if } n_1 \text{ and } n_2 \text{ coincide}$$

$$\frac{\langle a_1, s\rangle \Rightarrow n_1 \qquad \langle a_2, s\rangle \Rightarrow n_2}{\langle a_1 = a_2, s\rangle \Rightarrow \mathit{false}} \quad \text{if} n_1 \text{ and } n_2 \text{ differ}$$

- Less than or equal to:

$$\frac{\langle a_1, s\rangle \Rightarrow n_1 \qquad \langle a_2, s\rangle \Rightarrow n_2}{\langle a_1 \leq a_2, s\rangle \Rightarrow \mathit{true}} \quad \text{if } n_1 \text{ is lesser than or equal to } n_2$$

$$\frac{\langle a_1, s\rangle \Rightarrow n_1 \qquad \langle a_2, s\rangle \Rightarrow n_2}{\langle a_1 \leq a_2, s\rangle \Rightarrow \mathit{false}} \quad \text{if } n_1 \text{ is bigger than } n_2$$

- Negation: $\dfrac{\langle b, s\rangle \Rightarrow \mathit{true}}{\langle \neg b, s\rangle \Rightarrow \mathit{false}} \qquad \dfrac{\langle b, s\rangle \Rightarrow \mathit{false}}{\langle \neg b, s\rangle \Rightarrow \mathit{true}}$

- Disjunction: EXERCISE

Operational semantics

Small-step

- In small-step operational semantics description, program execution can be followed instruccion-by-instruction.
- When executing a program P from the initial state s_I (where no variable is bound to any value, i.e.: $s_I = \{\}$), we obtain a sequence of configurations (a trace):

$$\langle P, s_I \rangle = \langle P_1, s_1 \rangle \to \langle P_2, s_2 \rangle \to \cdots \to \langle P_n, s_n \rangle$$

We distinguish two situations:

1. P_n is the empty instruction (*skip*) for some $n \geq 1$. Then, the program execution terminates with final state $s_F = s_n$.
2. P_n is not the empty instruction for any n: the program execution does not terminate.

The language SIMP

Small-step semantics (I)

- Sequence:

$$\frac{}{\langle skip; i, s\rangle \rightarrow \langle i, s\rangle} \qquad \frac{\langle i_1, s\rangle \rightarrow \langle i_1', s'\rangle}{\langle i_1; i_2, s\rangle \rightarrow \langle i_1'; i_2, s'\rangle}$$

- Assignment:

$$\frac{\langle a, s\rangle \Rightarrow n}{\langle x := a, s\rangle \rightarrow \langle skip, s[x \mapsto n]\rangle}$$

where the new state $s[x \mapsto n]$ is given by removing from s any possible binding for x and then adding the new binding $x \mapsto n$:

$$s[x \mapsto n](y) = \begin{cases} s(y) & \text{if } y \neq x \\ n & \text{if } y = x \end{cases}$$

The language SIMP

Small-step semantics (II)

- Conditional:

$$\frac{\langle b,s\rangle \Rightarrow \mathit{true}}{\langle \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2, s\rangle \rightarrow \langle i_1, s\rangle} \qquad \frac{\langle b,s\rangle \Rightarrow \mathit{false}}{\langle \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2, s\rangle \rightarrow \langle i_2, s\rangle}$$

- *While* loop:

$$\frac{\langle b,s\rangle \Rightarrow \mathit{false}}{\langle \texttt{while } b \texttt{ do } i, s\rangle \rightarrow \langle \mathit{skip}, s\rangle} \qquad \frac{\langle b,s\rangle \Rightarrow \mathit{true}}{\langle \texttt{while } b \texttt{ do } i, s\rangle \rightarrow \langle i; \texttt{while } b \texttt{ do } i, s\rangle}$$

Exercise

Provide a small-step semantic description of the *while* loop by using a single rule and the conditional instruction.

Operational semantics

Big-step

- The big-step operational semantics description describes the execution of a program P as a direct transition from the initial configuration $\langle P, s_I \rangle$ to the final state s_F.
- In contrast to the small-step semantics, the big-step transition relation $\Downarrow$ relates configurations and states: $\langle P, s \rangle \Downarrow s'$

The language SIMP

Big-step semantics

- Empty instruction:

$$\frac{}{\langle skip, s\rangle \Downarrow s}$$

- Sequence:

$$\frac{\langle i_1, s\rangle \Downarrow s_1 \qquad \langle i_2, s_1\rangle \Downarrow s'}{\langle i_1; i_2, s\rangle \Downarrow s'}$$

- Assignment:

$$\frac{\langle a, s\rangle \Rightarrow n}{\langle x{:=}a, s\rangle \Downarrow s[x \mapsto n]}$$

- Conditional:

$$\frac{\langle b,s\rangle \Rightarrow true \qquad \langle i_1,s\rangle \Downarrow s'}{\langle \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2, s\rangle \Downarrow s'} \qquad \frac{\langle b,s\rangle \Rightarrow false \qquad \langle i_2,s\rangle \Downarrow s'}{\langle \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2, s\rangle \Downarrow s'}$$

- *While* loop:

$$\frac{\langle b,s\rangle \Rightarrow false}{\langle \texttt{while } b \texttt{ do } i, s\rangle \Downarrow s} \qquad \frac{\langle b,s\rangle \Rightarrow true \qquad \langle i,s\rangle \Downarrow s' \qquad \langle \texttt{while } b \texttt{ do } i, s'\rangle \Downarrow s''}{\langle \texttt{while } b \texttt{ do } i, s\rangle \Downarrow s''}$$

The language SIMP

Big-step semantics

Exercise

Define the big-step semantics of the *while* loop by using a single rule and the conditional instruction.

Program semantics

We define the semantics $\mathcal{S}(P)$ of a (terminating) program P by using the *small-step* and *big-step* operational descriptions:

- $\mathcal{S}^{small}(P)$ is the (unique) finite trace

  $$\langle P, s_I \rangle = \langle P_1, s_1 \rangle \to \langle P_2, s_2 \rangle \to \cdots \to \langle P_n, s_n \rangle = \langle skip, s_F \rangle$$

  which is obtained from the small-step transition system.
- $\mathcal{S}^{big}(P)$ is the final state s_F which is obtained from the big-step transition system to compute $\langle P, s_I \rangle \Downarrow s_F$.

Both are related (same s_F). However, $\mathcal{S}^{big}$ has a bigger abstraction level than $\mathcal{S}^{small}$ ($\mathcal{S}^{big}$ keeps no information about the computation of s_F)

Exercise

Compute the semantics of: P= (x:=4; while x>3 do x:=x-1)

Axiomatic semantics

A Hoare triple $\{P\}\ S\ \{Q\}$ represents the correctness of program programa S with respect to

- a precondition P (that restricts the input states to S) y
- a postcondition Q (that represents the desired output states)

Program correctness

Whenever a state s satisfies P, the final state s' which is obtained after the execution of S also satisfies Q

Examples

$$\begin{array}{ccc} \{y = 4\} & x{:=}y & \{x = 4\} \\ \{y \leq x\} & z{:=}x;\ z{:=}z{+}1 & \{y < z\} \end{array}$$

Axiomatic semantics

Dijkstra invented a predicate transformer that for a given instruction i and postcondition Q yields a weakest precondition $wp(i, Q)$

Such a *weakest precondition* must hold for any program state previous to the execution of i so that Q holds after the execution of i

In this setting, correctness of a program S with respect to P and Q, i.e., $\{P\}\ S\ \{Q\}$ is checked in two steps as follows:

1. Compute $P' = wp(S, Q)$.
2. Prove $P \Rightarrow P'$.

Axiomatic semantics

The predicate transformer *wp*

- Asignment:

$$wp(x{:=}a, Q) = Q[x \mapsto a]$$

where $x \mapsto a$ is a substitution that replaces a variable x in a expresión by another expression a. In this way, $Q[x \mapsto a]$ is the application of such a substitution to the logical expression Q.

- Conditional:

$$wp(\texttt{if}\ b\ \texttt{then}\ i_1\ \texttt{else}\ i_2, Q) \ = \ (b \wedge wp(i_1, Q)) \ \vee \ (\neg b \wedge wp(i_2, Q))$$

- Sequence:

$$wp(i_1;i_2, Q) = wp(i_1, wp(i_2, Q))$$

Axiomatic semantics

Example of *wp* calculation

$$
\begin{array}{rcl}
\{P\} & = & \{x = 0 \wedge y = 1 \wedge z = 2\} \\
 & & \{P_1\} \\
 & & \qquad t := x \\
 & & \{P_2\} \\
 & & \qquad x := y \\
 & & \{P_3\} \\
 & & \qquad y := t \\
\{Q\} & = & \{x = 1 \wedge y = 0\}
\end{array}
$$

1. Bottom-up calculation of P' (here it is equal to P_1):
   - $P_3 = wp(y{:=}t, Q) = Q[y \mapsto t] = (x = 1 \wedge t = 0)$.
   - $P_2 = wp(x{:=}y, P_3) = P_3[x \mapsto y] = (y = 1 \wedge t = 0)$.
   - $P_1 = wp(t{:=}x, P_2) = P_2[t \mapsto x] = (y = 1 \wedge x = 0)$.
2. Since $P_1 = wp(S, Q)$, we check that $P \Rightarrow P_1$, i.e.,

$$(x = 0 \wedge y = 1 \wedge z = 2) \Rightarrow (y = 1 \wedge x = 0)$$

which clearly holds.

Axiomatic semantics

Example

Given the following Hoare triple:

$$
\begin{aligned}
\{P\} &= \{x = 1\} \\
&\quad x := x - 1 \\
\{Q\} &= \{x \geq 0\}
\end{aligned}
$$

is the program correct?

Solution: Since

$$wp(x := x - 1, x \geq 0) = (x - 1 \geq 0) \Leftrightarrow x \geq 1$$

and

$$x = 1 \Rightarrow x \geq 1,$$

we conclude the correctness of the program w.r.t. P and Q.

Semantic properties

Program equivalence

By using the semantics of a PL we can reason about program equivalence

Program equivalence

Two programs P and P' are equivalent with respect to a semantic description $\mathcal{S}$ (e.g., $\mathcal{S}^{big}$ or $\mathcal{S}^{small}$) if and only if

$$\mathcal{S}(P) = \mathcal{S}(P')$$

We then write $P \equiv_{\mathcal{S}} P'$.

For instance, for programs

$$P: \quad x{:=}1; \qquad\qquad P': \quad x{:=}2;$$
$$\quad\;\; x{:=}2;$$

we have $P \equiv_{\mathcal{S}^{big}} P'$, but $P \not\equiv_{\mathcal{S}^{small}} P'$ (Why?).

Semantic properties

Example

Assume the syntax and semantics of language SIMP enriched with product and quotient operators (exercise), and consider the following programs:

$P: \quad sum := (n * (n+1))/2;$

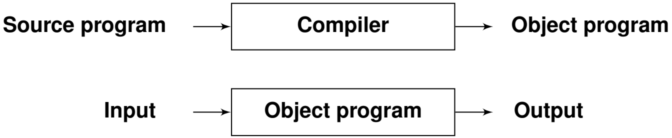
$$
\begin{array}{ll}
P': & sum := 0; \\
 & i := 1; \\
 & \texttt{while } i \leq n \texttt{ do} \\
 & \quad sum := sum + 1; \\
 & \quad i := i + 1;
\end{array}
$$

that compute $1 + 2 + \cdots + n$ for a given positive number n.

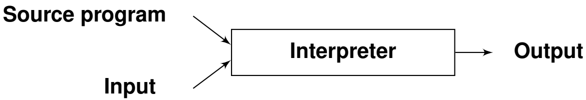
- From the point of view of the computation steps, which is the most efficient one?
- Are $\mathcal{S}^{small}$ or $\mathcal{S}^{big}$ able to capture this?
- Are P and P' equivalent with respect to $\mathcal{S}^{small}$ or $\mathcal{S}^{big}$ (or both)? Why?

Implementation of programming languages

- Compiled languages



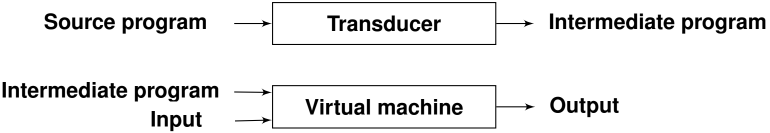
- Interpreted languages



Good environments include both interpreters (useful during software development) and compilers (useful in the final steps).

Translation vs interpretatión (I)

- Pure translation and interpretation are rare.
- In practice, pure translation is never used, except for languages of very similar level (e.g., assemblers)
- Pure interpretation is not frequent either, except for scripting or interactive languages
- A mixed implementation is usual: the source program is first translated into a 'more executable' format, which is then really executed by using an interpreter.



Translation vs interpretatión (II)

- Typically compiled languages:

C, C++, Fortran, Ada

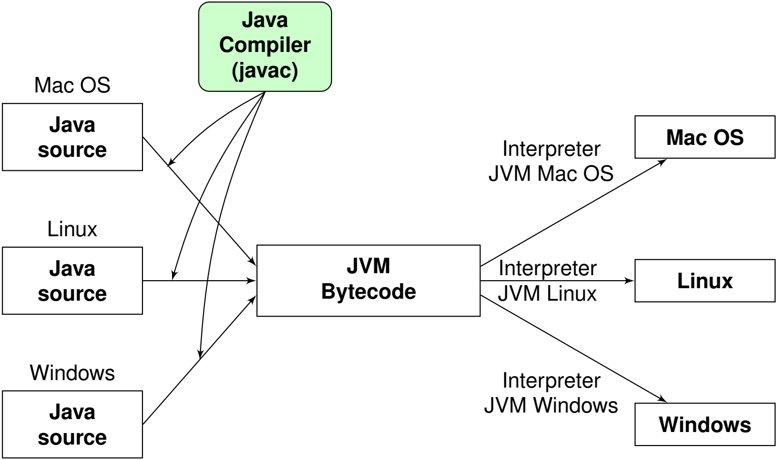
- Typically interpreted languages:

LISP, ML, Smalltalk, Perl, Postscript

- Languages with mixed (usually more portable):

- Pascal (P-code),
- Prolog (WAM-code),
- Java (byte-code, which is the JVM code, i.e., the standard format for the distribution of Java code)

Java Virtual Machine (JVM)



References

Basic:

- Winskel, G. The formal Semantics of Programming Languages. An introduction. MIT Press, 1993.
- Pratt,T.W. and Zelkowitz, M.V. Programming Languages. Design and Implementation, Prentice-Hall, 1996.
- Scott, M.L. Programming Language Pragmatics, Morgan Kaufmann Publishers, 2003.

Complementary

- Stuart, T. Understanding Computation (Chapter 2). Ed. O'Reilly, 2013.
- Kenneth Slonneger, Barry L. Kurtz. Formal Syntax and Semantics of Programming Languages. A Laboratory Based Approach (Chapters 1 and 11). Addison-Wesley, 1995.