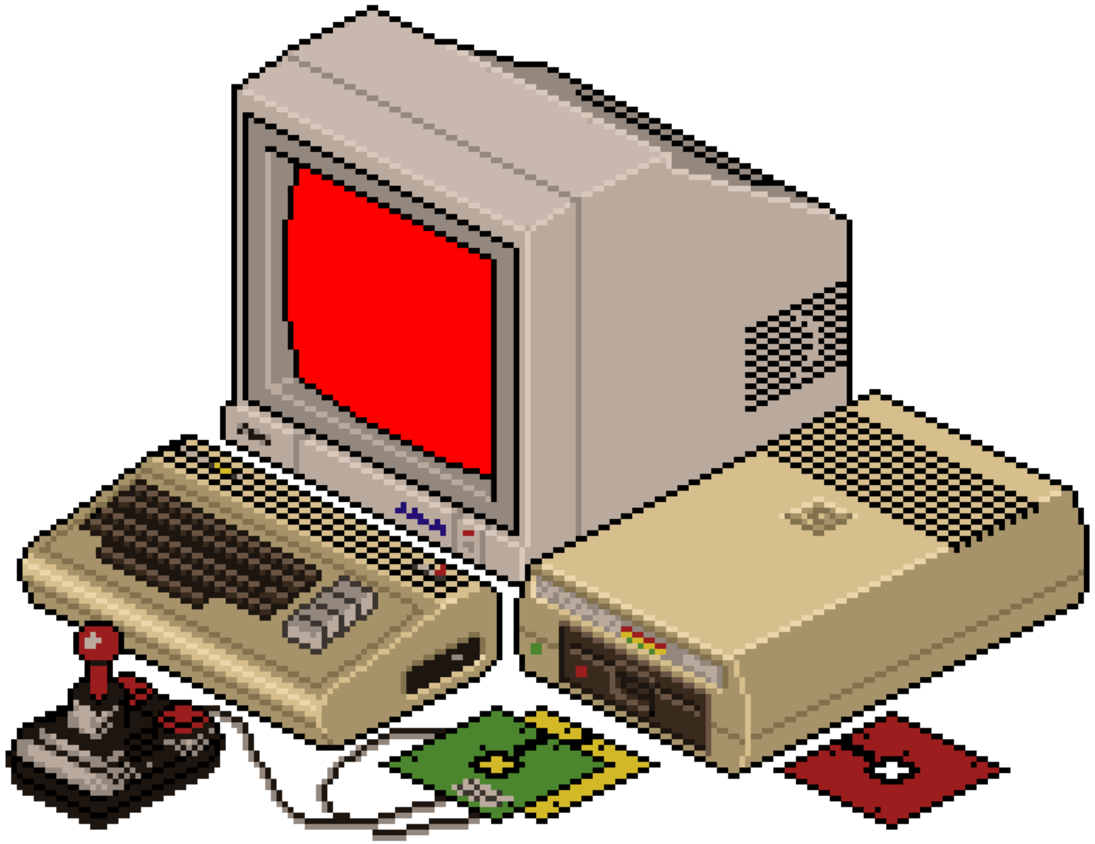
PROLOG

Programmation en Logique

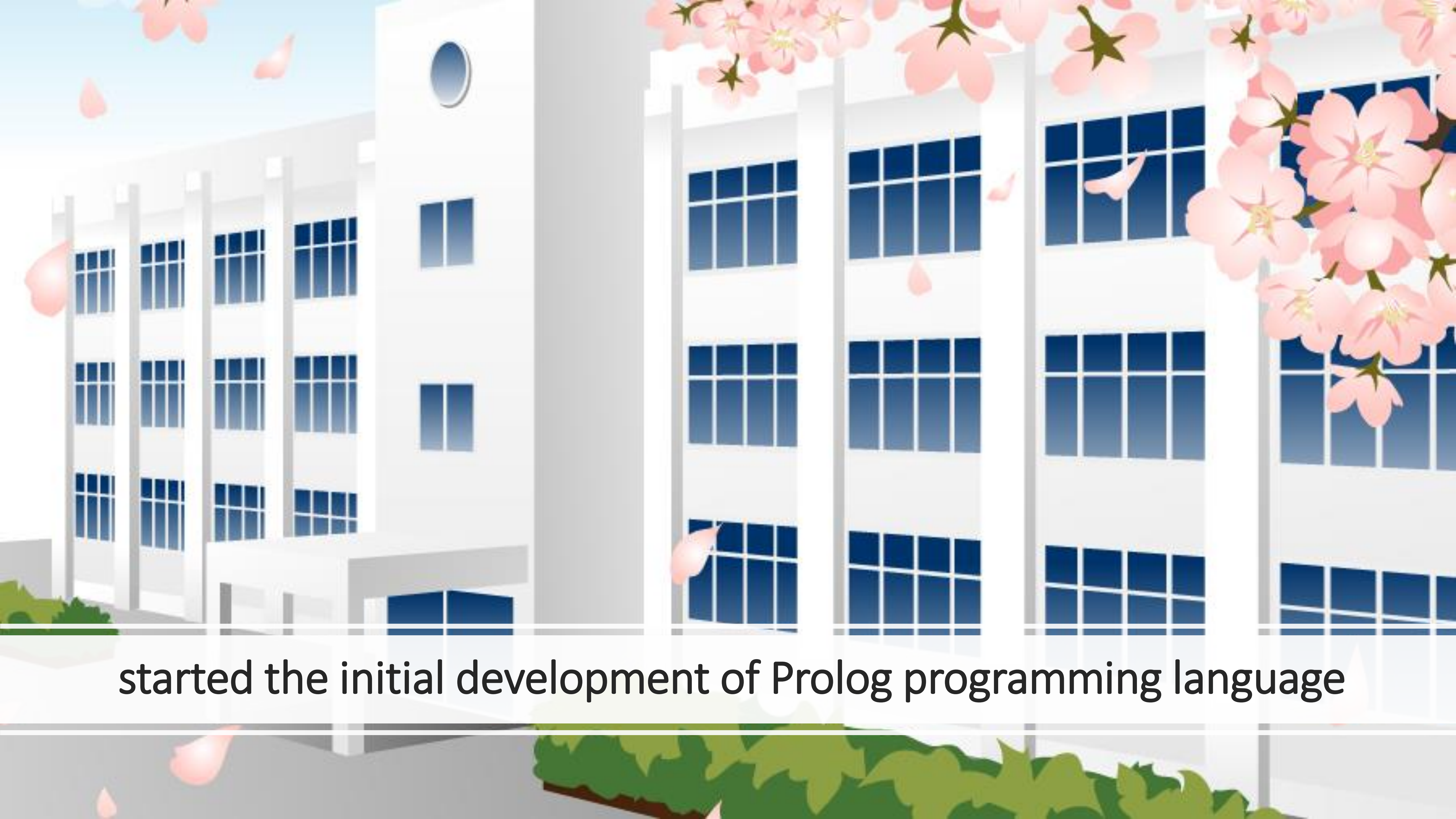
ARZADON - AWIT – BARITUA - BATOMALAQUE

Logical & Declarative programming language



HISTORY

1970s at University of Aix-Marseille, France

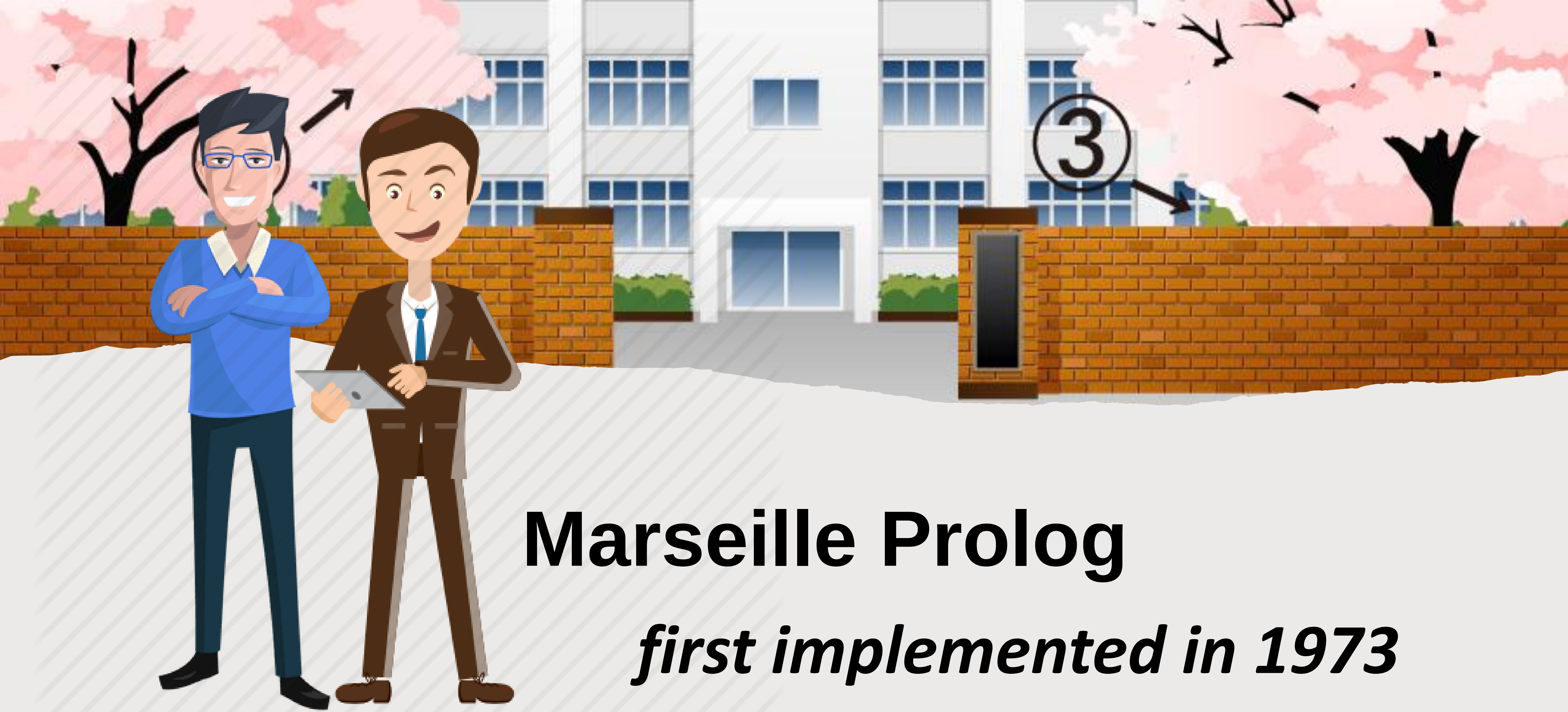


started the initial development of Prolog programming language

DEVELOPED BY:

Alain Colmerauer

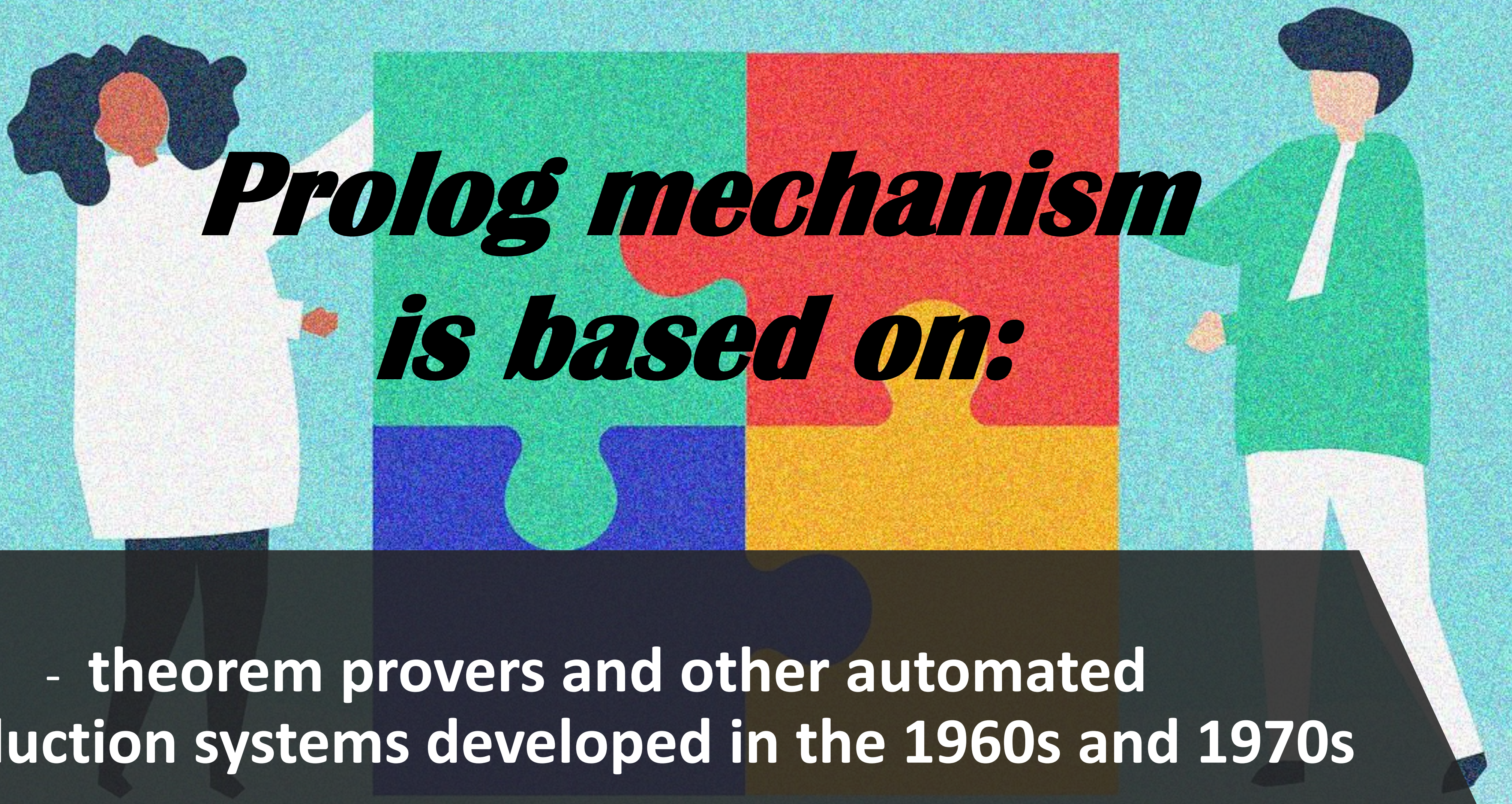
Philippe Roussel



Marseille Prolog

first implemented in 1973

Prolog mechanism is based on:



- theorem provers and other automated deduction systems developed in the 1960s and 1970s

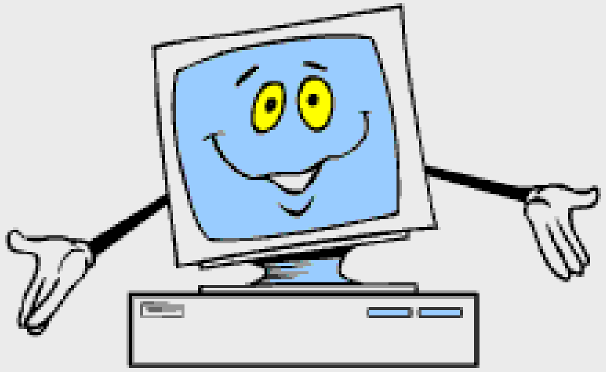
Resolution Theorem Proving

- **1963 at the U.S. Atomic Energy Commission's Argonne National Laboratory**
- **Alan Robinson (British Logician)**


prolog was further developed by
Robert Kowalski
Logician
at University of Edinburgh

Prolog

can determine whether or not a given statement follows logically from other given statements



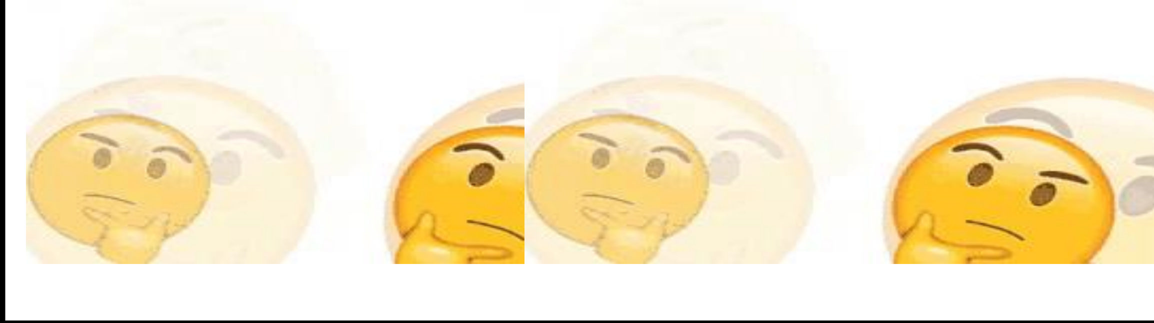
"All logicians are rational."

"Robinson is a logician."

"Robinson is rational?"

Widely used for AI work, especially in Europe and Japan

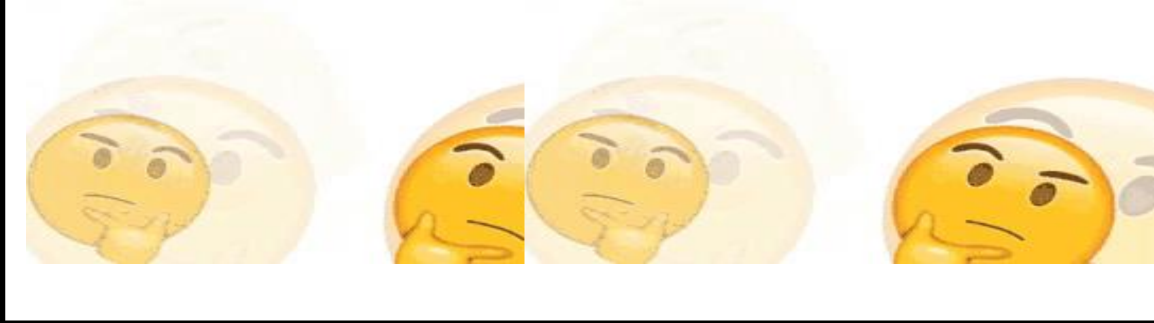
BASIC USE



- Proving theorems
- Natural language processing
- Expert systems

- Creation of graphic based user interfaces
- Networked & administrative applications

BASIC USE



- Database searches
- Template fillings
- Voice control systems

PROLOG: DOMAIN AND PARADIGM

PARADIGM

- Paradigm is Logic programming.
- Express facts and rules about problems within a system of formal logic.
- Rules are written as logical clauses with a head and a body.
- Facts are expressed similar to rules but without a body.
- Visual Prolog is a multi paradigm programming language.
- Prolog is highly used in artificial intelligence.
- Prolog is also used for pattern matching over natural language parse trees.

DOMAIN SECTIONS

- Defines a set of domains in the current scope.

```
DomainsSection:
    domains DomainDefinition-dot-term-list-opt
```

DOMAIN DEFINITIONS

- Defines a named domain in the current scope.

DomainDefinition:
    DomainName FormalTypeParameterList-**opt** = *DomainExpression*

DOMAIN EXPRESSIONS

- Denotes a type in a domain definition.

DomainExpression: **one of**
  TypeName
  CompoundDomain
  ListDomain
  PredicateDomain
  IntegralDomain
  RealDomain
  TypeVariable
  ScopeTypeVariable
  TypeApplication

TYPE EXPRESSIONS

- A subset of Domain Expressions that are used in other context since Domain Expressions can only be used in Domain Definitions.

TypeExpression: **one of**
  TypeName
  ListDomain
  TypeVariable
  ScopeTypeVariable
  TypeApplication

TYPE NAMES

- Either an interface name or the name of a value domain.

TypeName: **one of**
  InterfaceName
  DomainName
  ClassQualifiedDomainName

InterfaceName:
  LowercaseIdentifier

DomainName:
  LowercaseIdentifier

ClassQualifiedDomainName:
  ClassName::*DomainName*

ClassName:
  LowercaseIdentifier

COMPOUND DOMAINS

- Also known as algebraic data types.
- Used to represent lists, trees, and other tree structured values.

```
CompoundDomain:
   Alignment-opt FunctorAlternative-semicolon-sep-list
```

```
Alignment:
    align IntegralConstantExpression
```

```
FunctorAlternative:
   FunctorName FunctorName ( FormalArgument-comma-sep-list-opt )
```

```
FormalArgument:
   TypeExpression ArgumentName-opt
```

LIST DOMAINS

- Represents a sequence of values of a certain domain

```
ListDomain:
    TypeExpression *
```

```
ListExpression:
    [ Term-comma-sep-list-opt ]
    [ Term-comma-sep-list | Tail ]
```

```
Tail:
    Term
```

PREDICATE DOMAINS

- Values of a predicate domain are predicates with the same signature.

```
PredicateDomain:
   ( FormalArgument-comma-sep-list-opt ) ReturnArgument-opt
   PredicateModeAndFlow-list-opt CallingConvention-opt

FormalArgument:
   TypeExpression VariableName-opt
   Ellipsis

ReturnArgument:
    -> FormalArgument

VariableName:
   UpperCaseIdentifier
```

```
PredicateModeAndFlow:
     PredicateMode-opt
     FlowPattern-list-opt
```

PREDICATE MODE

PredicateMode: **one of**

- erroneous
- failure
- procedure
- determ
- multi
- nondeterm

erroneous = {}
failure = {Fail}
procedure = {Succeed}
determ = {Fail, Succeed}
multi = {Succeed, BacktrackPoint}
nondeterm = {Fail, Succeed, BacktrackPoint}

FLOW PATTERN

- Defines the input/output direction of the argument

```
FlowPattern:
    ( Flow-comma-sep-list-opt ) AnyFlow

Flow: one of
    i
    o
    FunctorFlow
    ListFlow
    Ellipsis
```

```
Ellipsis:
    ...
```

```
FunctorFlow:
    FunctorName ( Flow-comma-sep-list-opt )
```

```
ListFlow:
    [ Flow-comma-sep-list-opt ListFlowTail-opt]

ListFlowTail:
    | Flow
```

CALLING CONVENTION

- Determines how arguments are passed to the predicate.

CallingConvention:
    language *CallingConventionKind*

CallingConventionKind: **one of**
    c thiscall stdcall apicall prolog

Feature	Implementation
Argument-passing order	Right to left.
Argument-passing convention	By value, unless a compound domain term is passed. So it cannot be used to predicates with variable number of arguments.
Stack-maintenance responsibility	Called predicate pops its own arguments from the stack.
Name-decoration convention	An underscore (_) is prefixed to the predicate name.
Case-translation convention	No case translation of the predicate name is performed.

Keyword	Stack cleanup	Predicate name case-translation	Link predicate name decoration convention
c	**Calling** predicate pops the arguments from the stack.	None.	An underscore (_) is prefixed to the predicate name.
thiscall	**Calling** predicate pops the arguments from the stack except the implicit **This** argument which is passed in the register.	None.	**c** link name strategy is used.
stdcall	**Called** predicate pops its own arguments from the stack.	None.	An underscore (_) is prefixed to the predicate name.
apicall	**Called** predicate pops its own arguments from the stack.	The first letter of the predicate name is changed to the capital letter.	An underscore (_) is prefixed to the name. The first letter is changed to the upper case. The '**A**', the '**W**' or nothing is suffixed. The sign @ is suffixed. The (decimal) number of bytes in the argument list is suffixed.

FORMAT STRINGS

```
predicates
    writef : (string Format [formatstring], ...).
```

f	Format real's in fixed-decimal notation (such as 123.4 or 0.004321). This is the default for real's.
e	Format real's in exponential notation (such as 1.234e+002 or 4.321e-003).
g	Format real's in the shortest of f and e format, but always in e format if exponent of the value is less than -4 or greater than or equal to the precision. Trailing zeros are truncated.
d or D	Format as a signed decimal number.
u or U	Format as an unsigned integer.
x or X	Format as a hexadecimal number.
o or O	Format as an octal number.
c	Format as a char.
B	Format as the Visual Prolog binary type.
R	Format as a database reference number.
p	Format as the presented value.
P	Format as a procedure parameter.
s	Format as a string.

INTEGRAL DOMAINS

- Used for representing Integral Numbers.

```
IntegralDomain:
    DomainName-opt IntegralDomainProperties
```

```
IntegralDomainProperties:
    IntegralSizeDescription IntegralRangeDescription-opt
    IntegralRangeDescription IntegralSizeDescription-opt

IntegralSizeDescription:
    bitsize DomainSize

DomainSize:
    IntegralConstantExpression
```

```
IntegralRangeDescription:
    [ MinimalBoundary-opt .. MaximalBoundary-opt ]

MinimalBoundary:
    IntegralConstantExpression

MaximalBoundary:
    IntegralConstantExpression
```

```
MinimalBoundary <= MaximalBoundary
```

REAL DOMAINS

- Used to represent fractional parts/floating point numbers.

```
RealDomainProperties: one of
    RealPrecisionDescription RealRangeDescription-opt
    RealRangeDescription RealPrecisionDescription

RealPrecisionDescription:
    digits IntegralConstantExpression
```

```
RealRangeDescription:
   [ MinimalRealBoundary-opt .. MaximalRealBoundary-opt ]

MinimalRealBoundary:
   RealConstantExpression

MaximalRealBoundary:
   RealConstantExpression
```

```
MinimalBoundary <= MaximalBoundary
```

GENERIC DOMAIN

FormalTypeParameterList:
  TypeVariable-**comma-sep-list-opt**

TypeVariable:
  UpperCaseIdentifier

TypeApplication:
  TypeName { *TypeExpression*-**comma-sep-list-opt** }

SOURCE

- https://wiki.visual-prolog.com/index.php?title=Language_Reference/Domains
- https://www.computerhope.com/jargon/l/logic-programming.htm

PROLOG: Features

DATA TYPES “Terms”

Numbers

Atoms

Variables

NUMBERS

- Can be written as any number sequence from 0 to 9

- Optionally preceded by a + or – sign

ATOMS

- Any sequence of one or more letters, numerals, and underscores
- Starts with a lowercase letter

- rocky, today_is_Thursday, a32_BCD

ATOMS

- Any sequence of characters, including spaces, and uppercase letters, included in single quotes
- ‘Today is Friday’, ’32bcd’

ATOMS

- Any sequence of one or more special characters in a list containing:
  +-*/><=&#@

- +++, >=, >, +-

VARIABLES

- Any sequence one or more letters, numerals, and underscores
- Starting with an uppercase letter/underscore

- X, Author, Person_A

OPERATORS

Logical

Arithmetic

Relational

LOGICAL

Prolog	Read as	Logical operation
:-	IF	Implication
,	AND	Conjuction
;	OR	Disjunction
not	NOT	Negation

ARITHMETIC

Symbol	Operation
+	addition
-	subtraction
*	multiplication
/	real division
//	integer division
mod	modulus
**	power

RELATIONAL

Operator	Meaning
X = Y	equal to
X \= Y	not equal to
X < Y	less than
X > Y	more then
X =< Y	less then or equal to
X >= Y	more then or equal to

DATA STRUCTURES

Lists

Pairs

Association Lists

LISTS

- A special case of terms
- Represents a collection of elements

PAIRS

- Terms with principal functor
- Also called *Key-Value*

ASSOCIATION LISTS

- Available to allow faster than linear access to a collection of elements
- Typically based on balanced trees like AVL Trees

CONTROL STRUCTURES

Disjunction

Conjunction

Cut

Control Abstraction

DISJUNCTION

- Logical OR
- Parenthesis are always present for clarity

```
likes(alice, music).
likes(bob, hiking).

// Either alice likes music, or bob likes hiking will succeed.
```

CONJUNCTION

- Logical AND
- Represented by the Comma (,)

```
triangleSides(X,Y,Z) :-
    X + Y > Z, X + Z > Y, Y + Z > X.
```

CUT

- Stops Prolog from backtracking into alternative solutions

- Written as !

```
% (percent signs mean comments)
% a is the parent of b, c, and d.
parent(a,b).
parent(a,c).
parent(a,d).
```

```
?- parent(a,X), !.
```

CONTROL ABSTRACTION

- For loop

```
loop(0).
loop(N):-N>0,write('The value is:'),write(N),nl,
M is N-1,loop(M).
```