comparing Lisp and Prolog

Yves Lespérance
Adapted from Peter Roosen-Runge

exam

1/3 Lisp,
2/3 Prolog;
concepts, definitions, comparisons;
small programming "fragments"
  testing mainly 'reading knowledge'
functions vs. relations

- In math, functions are a special type of relation, have functional property, i.e., at most one value for each domain element.
- Distinguish between domain and range.
- Functional expression \( f(\text{arg}) \) stands for *the value* of the function for the argument.
- Asymmetry between input and output.
- A relational expression \( r(a,b) \) is a statement that may be true or false.

composition

- Problems are solved by breaking up into sub-problems, solving them, and composing the results.
- In functional language, composition is easily specified.
- Function evaluation simplifies complex expressions by replacing a sub-expression with a value:
  evaluate arguments first, then apply the function to their values.
relation composition

- in relational languages/algebra, composition is more complex to specify
- many ways of composing, e.g.
  
  \[
  \begin{align*}
  rc1(X,Z) & \text{ :- } r1(X,Y), r2(Y,Z). \\
  rc2(X,Z) & \text{ :- } r1(Y,X), r2(Y,Z). 
  \end{align*}
  \]
  etc., worse if more than 2 arguments.
- in Prolog, must name intermediate values

reversibility

- in relational language, no fixed input and output arguments, e.g. append
- relational notation allows the same definition to support many uses, e.g. get suffix S of P in L, `append(P,S,L)`
- depends on pattern matching/unification
- but not all Prolog predicate definitions are reversible/steadfast
pattern matching

- can simplify definitions, avoid explicit selection/construction of data, e.g. Prolog list manipulation vs. Lisp FIRST, REST, and CONS
- supports manipulation of partly specified data objects
- in Lisp, only macro destructuring
- but can be added to functional languages (e.g. Miranda)

simplification through rewriting

\[ X + 0 \rightarrow X \]
\[ 0 + X \rightarrow X \]
\[ X \times 1 \rightarrow X \]
\[ 1 \times X \rightarrow X \]
\[ X - X \rightarrow 0 \]
\[ X + X \rightarrow X \times 2 \]

- what is \((a+a) - 1 \times a \times (2+0)\)?

note the evaluation of function composition, in which the result of one simplification triggers another
the core of the code

% based on Quintus Prolog Library
rewrite(Expr, Final) :-
    path_arg(Path, Expr, Lhs),
    Lhs ->> Rhs,
    change_path_arg(Path, Expr, Rhs, Modified), !,
    rewrite(Modified, Final).
rewrite(Expr, Expr).

what's the function of the 'cut' here?

higher-order programming

♦ important for capturing patterns, supports abstraction
♦ maplist is as useful in Prolog as mapcar in Lisp
♦ higher-order programming is easier to introduce into a functional approach
♦ functions can take objects/functions and return objects/functions; relations just describe properties of objects
♦ Lambda-Prolog supports higher-order functions and relations
data-driven flow

♦ data-driven flow of control
  Lisp: mapcar and its relations
  selection of cases based on structure
  [], [X], [.. | X], etc.
♦ connection between recursive data structures and recursion in evaluation
♦ importance of using a simple recursive data structure
  lists built from atoms using a constructor:
  (A . B), CONS, [ | ]

so

♦ functional advantage: composition is easier to specify (major)
♦ relational advantage: pattern matching (major, can be added to functional)
♦ relational advantage: reversibility (sometimes, minor)
♦ functional advantage: supports higher-order programming better (major, can be added to relational?)
Which is more powerful, Lisp or Prolog?

- almost all programming languages are *Turing-equivalent*
- given unbounded time and memory, they can compute any computable function
  enumerate any enumerable set
- comparison must be based on other aspects

Comparisons between languages

- useful differences in programming languages are differences in:
  *conciseness* (economy of expression)
  *quality of libraries and the development environment*
  *ease of program construction*
    - tracing, debugging, etc.
  *user interface*
    - usability
    - performance
Fifth-Generation Computing

- over-hyped effort in mid-80s (by Japanese government and research community)
- promise of a breakthrough in "knowledge-based computing" using 'parallel Prolog' special hardware
- competing with Lisp machines from MIT-linked startups

special hardware not needed

- text refers to "new architectures" for Lisp, and to "Prolog machines"
- all long extinct -- why?
- Moore's Law advances in CPU and memory destroy any temporary advantage in special architecture
- at this point, brute force speedups are better than clever optimizations
recent work

- constraint logic programming, to support lazy, backtrackable expression evaluation
- new functional languages Standard ML, Haskell, Miranda, etc.
- concurrent functional and relational languages
- agent programming languages
- rules languages and process languages for semantic web

"The Birth of Prolog"
Alain Colmerauer and Philippe Roussel

- "At the beginning of July '70, Robert Pasero and Philippe [Roussel] arrived in Montreal. They had been invited by Alain [Colmerauer] who was then Assistant Professor of Computer Science at the University of Montreal and was leading the automatic translation project, TAUM (Traduction Automatique de l'Université de Montréal). All were at turning points in their careers. Robert and Philippe were then 25 years old and had just been awarded teaching positions in Computer Science . . . Alain was 29 years old and, after a 3-year stay in Canada, was soon to return to France. At the same time, Jean Trudel, a Canadian researcher and a doctoral student of Alain's, had chosen to work on automated theorem-proving. His point of reference was Alan Robinson’s article [1965] on the resolution principle."
the French connection

♦ Canadian government's policy of official bilingualism created a need to translate large numbers of official documents from English to French

♦ early test-case for computer translation: could weather reports, written in English, be automatically translated into French?

♦ provided rationale for funding for TAUM and development of its software (Q-system)

♦ "Alain ultimately [1972] found a way of developing powerful analyzers. He associated a binary predicate \( N(x,y) \) with each non-terminal symbol \( N \) of the grammar, signifying that \( x \) and \( y \) are terminal strings for which the string \( u \) defined by \( x = uy \) exists and can be derived from \( N \). By representing \( x \) and \( y \) by lists, each grammar rule can then be encoded by a clause having exactly the same number of literals as occurrences of nonterminal symbols. It was thus possible to do without list concatenation. (This technique is now known as the difference lists technique). Alain also introduced additional parameters into each non-terminal to propagate and compute information. As in Q-systems, the analyzer not only verified that the sentence is correct but also extracted a formula representing the information that it contains. Nothing now stood in the way of the creation of a man-machine communication system entirely in 'logic'."