O'Keefe Chapter 6: Sequences

Mike Daniels

6 November 2001

Introduction

- A good program is never written from scratch.
- A good program grows out of every previous program you've written.
- Thus, to be a good programmer, you need to have written programs to do a wide variety of tasks.
- You need to have mastered many programming schemata.

Programming Schemata

- Greedy Algorithms
- Dynamic Programming
- Standard Containers
- Sequence Operations
- Higher-Order Programming

Plan of Attack

- Learn some specific stuff about sequences: how to represent them and how to manipulate them.
- Learn how to take a wide variety of problems and express them using sequential terms in a schematic way.
- Learn how to apply general solution strategies to these schematized problems.

Sequences

- A bunch of stuff that begins in a certain spot and comes in a certain order.

- Think: the tickets you get at the entrance of a parking garage.
- Sequences can be both finite and infinite.
- See notation on page 188 of O'Keefe for sequences and metaprograms.

Representations

- Spatial: each element of the sequence occupies a cell in memory (as a fact in prolog's database or as the elements of a list).
- Temporal: Successive members of a sequence are returned by successive solutions offered by a goal (through backtracking) or by explicit function calls.

Questions to Ask

- Am I dealing with an infinite sequence?
- Can the sequence change in midstream?
- Am I making a single pass through the sequence?
- Do I need to examine multiple elements at once?

Formal Taxonomy

- Lists
- Generators
- Enumerators
- (Relations)

Lists

- Lists are the "standard" representation of sequences in Prolog.
- But even deciding on a list representation carries some flexibility.
- Normally, we distinguish between empty and non-empty lists.

- But what if we wanted to distinguish between empty, singleton, and plural lists?
- (Sidenote: why would you want to do this?)

ZOT!

- (The mnemonic is z(ero), o(ne), t(wo).)
- A zotlist of length zero is represented by the atom 'z'.
- A zotlist of length one is represented by a one-argument term (the element) with functor 'o'.
- All other zotlists are represented by a three-argument term using functor 't'. The first and second arguments are the first and second elements, and the third argument is another zotlist (the tail).
- Exercise: what does zotappend/3 look like?

---

zotappend/3

Interval

- Consider the integers between X and Y ($X < Y$) as a sequence.

- interval(Low, High, [High]) $\leftarrow$ Low = High.
- interval(Low, High, [Low|Tail]) $\leftarrow$ NewLow is Low + 1, interval(NewLow, High, Tail).
- Trivial, but useful for comparison.

Generators

- Wrap the concept of a sequence into three predicates:
- make_X(+Arguments, -State)
- X_next(?Element, +State0, -State)
- X_done(+State0)

Interval Generator

- make_interval(Low, High, i(Low, High)) $\leftarrow$ integer(Low), integer(High).
- interval_done(i(Low, High)) $\leftarrow$ Low > High.
- interval_next(Low, i(Low,High), i(Low1,High)) $\leftarrow$ plus(Low, 1, Low1), Low $\leq$ High.

Enumerators

- You enumerate a sequence by repeatedly backtracking into a predicate.
- (The voices in your head should be whispering 'failure-driven loop' right about now.)

Interval Enumerator

- ```
  enum_interval(Low, High, X) ←
        integer(Low), integer(High),
        (var(X) → Low ≤ High, enum_interval1(Low, High, X)
        ; integer(X) → Low ≤ X, X ≤ High).
  ```
- enum_interval1(L, U, L).
- enum_interval1(L, U, X) $\leftarrow$ L < U, M is L + 1, enum_interval1(M, U, X).
  ```

Enumerator Usage

- enum_interval(2,4,X): X = 2 ; X = 3 ; X = 4 ; no

Conversions

- Whenever you tell a computer scientist that X can be represented in more than way, one of the first questions you get asked is: how do I convert between these representations?
- But linguists should care as well: knowing how to convert between format X and format Y deepens your understanding of both formats.

List to Generator

- make_list(List, List).
- list_done([]).
- list_next(Head, [Head|Tail], Tail).

List to Enumerator

- member(X, List).
- It's that simple.

Generator to Enumerator

- X_member(Element, X0) ← X_next(E,X0,X1), (E=Element; X_member(Element, X1)).

Enumerator to List

- findall(X, G(X), L).
- . . . but be aware of what this encapsulates.

Enumerator to Generator

- You can't really do this directly.
- The state of such a generator would have to be Prolog's internal state when running the enumerator.
- You'd have to create a "step-wise" prolog interpreter.
- So just string together the enumerator to list and list to generator schemata.

Program Schematization

- Let's define lots of programs in terms of three types and four functions on these types.
- Type D for data items (input).
- Type W for working values.
- Type R for results (output).
- Function g: $D \rightarrow W$ to convert input to working values.
- Function f: $W \times W \rightarrow W$ is associative and combines working values.
- Function e: $\emptyset \rightarrow W$ That element in W such that f(e, X) = X (this is called the left identity of f).
- Function h: $W \rightarrow R$ converts working values to output.

Length of a list

- e = 0
- f(X, Y) = X + Y
- g(X) = 1
- h(X) = X

Sum of the elements of a list

- e = 0

- f(X, Y) = X + Y
- g(X) = X
- h(X) = X

One-level flattening

- e = nil
- f(X, Y) = X $\oplus$ Y
- g(X) = X
- h(X) = X

Generalized Union

- e = nil
- f(X, Y) = X $\cup$ Y
- g(X) = X
- h(X) = X

Predicated Filter

- e = nil
- f(X, Y) = X $\oplus$ Y
- g(X) = if σ(X) then [X] else nil
- h(X) = X

Okay. . . now what?

- We want to compute $p = h(g(X_1) f \cdots f g(X_n))$.
- $1 + \cdots + 1$
- $X_1 + \cdots + X_n$

- $X_1 \oplus \cdots \oplus X_n$
- $X_1 \cup \cdots \cup X_n$
- But this is math, and we need to turn math into Prolog.

EFGH

- e(E)
- f(Work1, Work2, WorkOut)
- g(Data, Work)
- h(Work, Result)

The Tower Method

- p(List, Result) ← p′(List, Answer), h(Answer, Result).
- p′([], Answer) ← e(Answer).
- p′([Head|Tail], Answer) ← p′(Tail, Right), g(Head, Val), f(Val, Right, Answer).

Towered List Sum

- list_sum([],0).
- list_sum([Head|Tail], Total) ← list_sum(Tail, Right), Total is Head + Right.

The Linear Method

- p(List, Result) ← e(E), p′(List, E, Answer), h(Answer, Result).
- p′([], Answer, Answer).
- p′([Head|Tail), SoFar, Answer) ← g(Head, Val), f(SoFar, Val, Next), p′(Tail, Next, Answer).

Linear List Sum

- list_sum(List, Result) ← list_sum1(List, 0, Result).
- list_sum1([], Answer, Answer).
- list_sum1([Head—Tail], SoFar, Answer) ← Next is SoFar + Head, list_sum1(Tail, Next, Answer).