Inductive Sets of data
(Essentials of Programming Languages

Chapter 1

Outline

This chapter introduces the basic programming tools we will need to write interpreters, checkers and similar programs that form the heart of a programming language processor.

- ◊ Recursively Specified Data
- ◊ Deriving Recursive Programs
- ◊ Auxiliary Procedures and Context Arguments

1.1 Recursively Specified Data

Two formal techniques for specifying sets of values

1) Inductive specification
2) Defining sets using grammars

Different styles of inductive specifications

1) Top-down definition
2) Bottom-up definition
3) Rules of inference

A grammar is a set of productions. Each production has the form $Lhs ::= Rhs$ where Lhs is a nonterminal and Rhs consists of terminals and nonterminals.

Inductive Specification

[Example] A top-down style inductive specification:

A natural number n is in S if and only if

1. $n = 0$ or
2. $n - 3 \in S$

Inductive Specification (cont.)

[Example] A bottom-up definition:

The set S to be the smallest set contained in $N = \{0, 1, 2, \cdots\}$ and satisfying the following two properties:

1. $0 \in S$, and
2. if $n \in S$ then $n + 3 \in S$.

[Example] Rules of inference:

The same set S as above

$$\frac{}{0 \in S} \qquad \frac{n \in S}{(n+3) \in S}$$

Defining Sets Using Grammars

Grammars

- Nonterminal symbols: the names of the sets being defined
- Terminal symbols: the characters in the external representation
- Production (rule)

Nonterminals = syntactic categories.

[Example]

$$
\begin{aligned}
List\text{–}of\text{–}int &::= () \\
&::= (Int\ .\ List\text{–}of\text{–}int)
\end{aligned}
$$

cf. Variations of production rules

[Example] $S\text{–}list$, $S\text{–}exp$, $Bintree$, $LcExp$

Defining Sets Using Grammars (cont.)

The grammars are said to be *context-free* because a rule defining a given syntactic category may be applied in any context that makes reference to that syntactic category.

Sometimes we have to look at the context in which the production is applied. Such constraints are called *context-sensitive constraints* .
- E.g., in many languages every variable must be declared before it is used.

In practice, the usual approach is first to specify a context-free grammar. Context-sensitive constraints are then added using other methods.

Induction

We use the inductive definitions in two ways:

- to prove theorems about members of the set and
- to write programs that manipulate them.

1.2 Deriving Recursive Programs

We have seen that we can analyze an element of an inductively defined set to see how it is built from smaller elements of the set.

We use this idea to define/write a general procedure that compute on inductively defined sets.

When defining a procedure that operates on inductively defined data, the structure of the program should be patterned after the structure of the data.

1.2 Deriving Recursive Programs (cont.)

list-length determines the number of elements in a list:

```
> (list-length '(a b c))
3
> (list-length '((x) ()))
2
```

nth-element picks the n-th element of a list:

```
> (nth-element '(a b c d e) 3)
d
```

1.2 Deriving Recursive Programs (cont.)

remove-first:

```
> (remove-first ‘a ‘(a b c))
(b c)
> (remove-first ‘b ‘(e f g))
(e f g)
> (remove-first ‘a4 ‘(c1 a4 c1 a4))
(c1 c1 a4)
> (remove-first ‘x ‘())
()
```

1.2 Deriving Recursive Programs (cont.)

The procedure occurs-free? takes a variable *var*, and a lambda-calculus expression *exp*, and it determines whether or not *var* occurs free in *exp*.

```
> (occurs-free? 'x 'x)
  #t
  > (occurs-free? 'x 'y)
  #f
  > (occurs-free? 'x '(lambda (x) (x y)))
  #f
  > (occurs-free? 'x '((lambda (x) x) (x y)))
  #t
  > (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
  #t
```

1.2 Deriving Recursive Programs (cont.)

The procedure *subst* takes three arguments: two symbols, *new* and *old*, and an s-list, *slist*. All elements of *slist* are examined, and a new list is returned that is similar to *slist* but with all occurrences of *old* replaced by instances of *new*.

```
> (subst 'a 'b '((b c) (b () d)))
  ((a c) (a () d)
```

1.3 Auxiliary Procedure and Context Arguments

Auxiliary procedures

- number-elements-from
- number-elements
- list-sum
- partial-vector-sum
- vector-sum