Today's Lecture Notes for cs173

sk and dbtucker

September 8, 2000

Purpose: To make you forget about syntax.

Quote of the Day: "Syntatic sugar causes cancer of the semicolon."—Alan Perlis

Using datatypes

We know how to make lists using `cons`, `first`, `rest`, and `empty`. Now we show how to create lists of numbers using the `define-datatype` mechanism:

```
(define-datatype LoN LoN?
  [mt]
  [kons (fst number?)
        (rst LoN?)])
```

We can create an empty `LoN`:

```
(mt)
```

Or a `LoN` with one number:

```
(kons 3 (mt))
```

How do we write a function that sums the numbers in a list? Use the `cases` construct.

```
(define (sum L)
  (cases LoN L
    [mt () 0]
    [kons (fst rst)
      (+ fst
         (sum rst))]))
```

Parsing

We are given *information* in some format, which we want to turn into *data*. Here are some examples:

Information	**Data**
3	`(num 3)`
$1 + 2$	`(add (num 1) (num 2))`
$5 \times (1 + 2)$	`(mult (num 5) (add (num 1) (num 2)))`

We will write programs that operate on the data, so they will work regardless of the information, provided that you can translate the information into data. Such translators are called *parsers*.

We need to define a datatype for the data representation given above:

```
(define-datatype AExp AExp?
  [num (n number?)]
  [add (lhs AExp?) (rhs AExp?)]
  [mult (lhs AExp?) (rhs AExp?)])
```

This says that an arithmetic expression (`AExp`) is either:

- a number,
- an addition of two arithmetic expressions, or
- a multiplication of two arithmetic expressions

In order to write a parser, we also need an external representation of the information. Here's the grammar:

$$
\begin{array}{rcl}
L & ::= & \langle \text{number} \rangle \\
 & | & \texttt{(}\langle L \rangle\ \texttt{+}\ \langle L \rangle\texttt{)} \\
 & | & \texttt{(}\langle L \rangle\ \texttt{*}\ \langle L \rangle\texttt{)}
\end{array}
$$

For example, `(5 * (1 + 2))` is an L program. In Scheme, this is just a list of symbols.

How do we turn an L program into an *AExp*? We write a parser:

```
;; parse : L program -> AExp
(define (parse L)
  (cond
    [(number? L) (num L)]
    [(symbol=? (second L) '+) (add (parse (first L))
                                   (parse (third L)))]
    [(symbol=? (second L) '*) (mult (parse (first L))
                                    (parse (third L)))]))
```

That's it. The first line specifies the *contract*—the type of input the `parse` expects, and the type of output it produces.

Our first interpreter

Now that we have an `AExp`, how do we compute a value from it?

```
;; calc : AExp -> number
(define (calc a)
  (cases AExp a
    [num (n) n]
    [add (le re) (+ (calc le) (calc re))]
    [mult (le re) (* (calc le) (calc re))]))
```

Voilà. Note that we use the Scheme definitions of `+` and `*` to specify how we reduce `add` and `mult`, respectively.

There are two types of terms in our algebra, values and computations:

- values: `num`
- computations: `add`, `mult`

An *interpreter* is a program that consumes computations and values, and returns values.