Abstract Syntax for Arithmetic Expressions
6.001 Section Notes

Albert R. Meyer

29 March 2000

In these notes we'll consider a very simple case of arithmentic expressions. There will be just four kinds of expressions: numbers, variables, sums and products. We can express this with a data-type equation:

$$\text{Expr} \cong \text{Number-expr} \mid \text{Variable} \mid \text{Sum} \mid \text{Product}$$

where the special symbol "$\cong$" should be read as "has the shape of". The "$|$" operation on types indicates that the four kinds of Expr's are separate cases—every expression is of exactly one of the four kinds above, and there are procedures which, given an expression, test for each case. That is, there are procedures:

$$\texttt{number-expr?},\ \texttt{variable?},\ \texttt{sum?},\ \texttt{product?} : \text{Expr} \to \text{Bool},$$

and these satisfy the contract that for any given arithmetic expression, exactly one of them returns the value `#t`.

We'll consider sums first. Abstractly, all that matters about a sum is that it can be identified as a sum—the procedure `sum?` does that—and that it has two summands. So a data type can serve as Sum if abstractly it "has the shape of" pairs of expressions:

$$\text{Sum} \cong \text{Pair}(\text{Expr}, \text{Expr}).$$

"Having the shape of" pairs means that there are constructors and selectors for Sum of types

$$\begin{aligned} \texttt{make-sum} &: \text{Expr}, \text{Expr} \to \text{Expr} & & \\ \texttt{addend} &: \text{Sum} \to \text{Expr} & & \text{(aka "first-summand")} \\ \texttt{augend} &: \text{Sum} \to \text{Expr} & & \text{(aka "second-summand")} \end{aligned}$$

which satisfy the *pairing contract*:

$$\texttt{(addend (make-sum <e\_1> <e\_2>))} \longrightarrow \texttt{<e\_1>},$$
$$\texttt{(augend (make-sum <e\_1> <e\_2>))} \longrightarrow \texttt{<e\_2>}.$$

Products work exactly the same way:

$$\text{Product} \cong \text{Pair}(\text{Expr}, \text{Expr})$$

$$
\begin{aligned}
\texttt{make-product} &: \text{Expr}, \text{Expr} \to \text{Expr} & \\
\texttt{multiplier} &: \text{Product} \to \text{Expr} & \text{(aka ``first-multiplicand'')} \\
\texttt{multiplicand} &: \text{Product} \to \text{Expr} & \text{(aka ``second-multiplicand'')}
\end{aligned}
$$

For simplicity, we won't introduce a Number-expr abstraction; we'll just use Scheme numbers:

$$\text{Number-expr} = \text{Scheme-numbers},$$

(Notice this time it is "=" not "≅".)

```
(define number-expr? number)
```

Finally, we consider variables. Certainly one important feature of a variable is its name. Names are intended for use in printing and as identification tags. Scheme Symbol's have exactly the properties we want for names, so we'll use them without further abstraction. To get the symbolic name of a variable, we require a selector:

$$\texttt{variable-name} : \text{Variable} \to \text{Symbol}.$$

Another important feature of a variable is its numerical value. This value is determined by an abstract structure called a *variable-values environment* whose type is designated Vvenv.[1]. (The point of variables is that their values may vary, which is why values are not "built into" variables.) A given expression may be evaluated using different environments which may assign different values to the same variable. There is a selector for looking up the value an environment assigns to a variable:

$$\texttt{lookup} : (\text{Variable}, \text{Vvenv}) \to \text{Number}$$

We'll examine the Variable and Vvenv abstractions in more detail later.

Evaluating expressions

At this point, the procedure for finding the numerical value of an expression should be easy to understand: the `eval-arith` procedure takes *two* arguments, the expression whose numerical value is to be calculated and a variable-value environment to assign the values of variables; it returns the numerical value of the expression.

$$\texttt{eval-arith} : (\text{Expression}, \text{Vvenv}) \to \text{Number}$$

[1] The simple variable-value environments we use here are sometimes called *dictionaries* or (abstract) *association lists*.

```
(define (eval-arith expr env)
  (cond ((number-expr? expr) expr)
        ((sum? expr)
         (let ((rand1
                (eval-arith (addend expr) env))
               (rand2
                (eval-arith (augend expr) env)))
           (+ rand1 rand2)))
        ((product? expr)
         (let ((rand1
                (eval-arith (multiplier expr) env))
               (rand2
                (eval-arith (multiplicand expr) env)))
           (* rand1 rand2)))
        ((variable? expr)
         (lookup expr env))
        (else (error "Bad expression" expr))))
```

Derivatives

A procedure for computing the derivative of a abstract expression was described in lecture. Here's what it looks like with abstract variables:

```
(define (deriv expr var)
  (cond ((number-expr? expr) 0)
        ((sum? expr)
         (make-sum
          (deriv (addend expr) var)
          (deriv (augend expr) var)))
        ((product? expr)
         (let ((plier (multiplier expr))
               (plicand (multiplicand expr)))
           (make-sum
            (make-product (deriv plier var) plicand)
            (make-product (deriv plicand var) plier))))
        ((variable? expr)
         (if (eq? (variable-name expr) (variable-name var)) 1 0))
        (else (error "Bad expression" expr))))
```

Exercise 1. Simplification is another useful operation on abstract expressions. Complete the definition of the `simplify` procedure below by defining an appropriate `simplify-prod` procedure.

```
(define (simplify expr)
  (cond ((sum? expr) (simplify-sum expr))
        ((product? expr) (simplify-prod expr))
        (else expr)))
```

```
(define (simplify-sum sum)
  (let ((ad (simplify (addend sum)))
        (au (simplify (augend sum))))
    (cond ((and (number? ad) (number? au))
           (+ ad au))
          ((and (number? ad) (zero? ad)) au)
          ((and (number? au) (zero? au)) ad)
          (else (make-sum ad au)))))

(define (simplify-prod prod)...)
```

Printing and Inputting Expressions

So far we have considered *computations* with abstract expressions, but another important activity is *printing* them. The standard Scheme/Lisp approach to printing expressions is an interesting contrast to printing in other languages: we don't try to write our own printing procedure. Instead, we take advantage of Scheme's built-in printer and the Scheme conventions for printing list structures. Namely, in Scheme, lists are conventionally displayed as parenthesized expressions which resemble completely parenthesized mathematical expressions as well as Scheme expressions. This, of course, is no coincidence.

So our printing strategy for abstract expressions is to *transform* an abstract expression *into a list structure*—more precisely a *tree* structure—whose printed representation looks like an expression in the format we desire.

For example, suppose we want to exhibit expressions in "infix" format—where the operator appears in between the operands as in

$$((x + 3) * 7)$$

The following procedure transforms an expression into a tree structure which prints in infix format:

$$\texttt{expr->infix-tree} : \text{Expr} \to \text{Tree(Sch-Num} \mid \text{Symbol)}$$

```
(define (expr->infix-tree expr)
  (cond ((number-expr? expr) expr)
        ((variable? expr) (variable-name expr))
        ((sum? expr)
         (let ((addend-tree (expr->infix-tree (addend expr)))
               (augend-tree (expr->infix-tree (augend expr))))
           (list addend-tree '+ augend-tree)))
        ((product? expr)
         (let ((multiplier-tree (expr->infix-tree (multiplier expr)))
               (multiplicand-tree (expr->infix-tree (multiplicand expr))))
           (list multiplier-tree '* multiplicand-tree)))
        (else (error "Bad expression" expr))))
```

To test this procedure, assume `varx`, `vary`, and `varz`, have been defined to be Variables with names `x`, `y`, `z`:

```
(define test-expr
  (make-product
   (make-sum 3 (make-product varz 9))
   (make-sum vary varz)))

>>  (expr->infix-tree test-expr)
;Value: ((3 + (x * 9)) * (y + z))
```

To get prefix format, we merely reorder the arguments in the two `list` expressions within the `expr->infix-tree` procedure:

```
(define (expr->prefix-tree expr)
  (cond ((number-expr? expr) expr)
        ((variable? expr) (variable-name expr))
        ((sum? expr)
         (let ((addend-tree (expr->prefix-tree (addend expr)))
               (augend-tree (expr->prefix-tree (augend expr))))
           (list '+ addend-tree augend-tree))) ; REORDERED
        ((product? expr)
         (let ((multiplier-tree (expr->prefix-tree (multiplier expr)))
               (multiplicand-tree (expr->prefix-tree (multiplicand expr))))
           (list '* multiplier-tree multiplicand-tree))) ; REORDERED
        (else (error "Bad expression" expr))))

>> (expr->prefix-tree test-expr)
;Value 7: (* (+ 3 (* x 9)) (+ y z))
```

In case you were wondering:

$$\text{Tree}(\text{Anytype}) \cong \text{Anytype} \mid \text{List}(\text{Tree}(\text{Anytype}))$$

where

$$\text{List}(\text{Anytype}) \cong \text{Niltype} \mid \text{Pair}(\text{Anytype}, \text{List}(\text{Anytype}))$$

and Niltype is some datatype with exactly one element.

Instead of constructing a tree, we could construct a string; this time we have to generate our own parentheses:

```
(define (expr->infix-string expr)
  (cond ((number-expr? expr) (number->string expr))
        ((variable? expr) (symbol->string (variable-name expr)))
        ((sum? expr)
         (let ((addend-string (expr->infix-string (addend expr)))
               (augend-string (expr->infix-string (augend expr))))
           (string-append
            "(" addend-string " + " augend-string ")")))
        ((product? expr)
         (let ((multiplier-string (expr->infix-string (multiplier expr)))
               (multiplicand-string (expr->infix-string (multiplicand expr))))
```

```
          (string-append "(" multiplier-string " * " multiplicand-string ")")))
        (else (error "Bad expression" expr))))

>> (expr->infix-string test-expr)
;Value: "((3 + (x * 9)) * (y + z))"
```

Note that all these printing procedures preserve the expression abstraction. The procedures print in the same desired format even if the implementation of expressions changes. Conversely, we can change the printing format by selecting the desired printing procedure—not by changing the expression implementation.

Exercise 2. Write an `expr->prefix-tree` procedure.

Exercise 3. Modify `expr->infix-string` so the outermost matching parentheses are not included in the result. Is this possible to do with infix lists? ... explain.

Exercise 4. So far, the only way to construct abstract expressions is by explicitly using the expression constructors as in the `test-expr` example above. It would be more convenient for users of an abstract expression system if they could use familiar input notation. For example, we would like an input procedure `infix-tree->expr` to turn an infix notation expression into an abstract one. Then, for example, instead of

```
(define test-expr
  (make-product
   (make-sum 3 (make-product varz 9))
   (make-sum vary varz)))
```

we could allow a user to enter

```
(define test-expr (infix-tree->expr '((3 + (z * 9)) * (y + z))))
```

Write such an `infix-tree->expr` procedure.

With the solution to the previous exercise we have a simple way to translate from infix to prefix notation:

```
(expr->prefix-tree (infix-tree->expr '((3 + (z * 9)) * (y + z))))
>>(* (+ 3 (* z 9)) (+ y z))
```

Variables and variable-values environments

Contracts

For our purposes, all that matters about a variable is its name and how to find its value. We know there are selectors `variable-name` and `lookup` for finding names and values.

To contruct Vvenv, we specify that there be an “initial” environment, and that there is a way to extend a given environment so it assigns a specified value for a new variable:

$$\texttt{initial-env} : \text{Vvenv}$$
$$\texttt{extended-env} : (\text{Variable}, \text{Number-expr}, \text{Vvenv}) \to \text{Vvenv}$$

The contract for these procedures is that:

1. (lookup <var> (extended-env <var> <val> <env>)) $\longrightarrow$ <val>, and
2. if <var_1> is not equal to <var_2>, then the two expressions

   (lookup <var_1> (extended-env <var_2> <val> <env>))

   and

   (lookup <var_1> <env>)

   return the same value.

From these constructors, we can see that Vvenv acts like a list of variable-value pairs:

$$\text{Vvenv} \cong \text{List}(\text{Pair}(\text{Variable}, \text{Number-expr}))$$

Implementing variables and Variable-values Environments

In many cases, the *only* thing that matters about a variable is its name, in which case a simple implementation of variables as Scheme symbols works well:

$$\text{Variable} = \text{Symbol}.$$

This allows a straightforward implemention of a variable-values environment as an association list of variables and numbers:

```
(define (make-variable symb) symb)
(define (variable-name var) var)

(define initial-env '())

(define (extended-env name val env)
  (cons (list name val) env))

(define (lookup var env) (assq var env))
```

For example, the environment in which variables x, y, z have respective values 6, 20, 20 would be a length three association list:

```
(define varx 'x)
(define vary 'y)
(define varz 'z)

(define env-1
  (extended-env
    varx 6
    (extended-env
     vary 20
     (extended-env
```

```
        varz 20
        initial-env))))

>>env-1
;Value: ((x 6) (y 20) (z 20))
```

and we could use this environment to evaluate our test expression above:

```
>> (eval-arith test-expr env-1)
;Value: 2280
```

Exercise 5. Complete an implementation of these abstract arithmetic expressions.

Exercise 6. Implement variable-values environments as trees whose leaves are numbers, and a variable consists of a name and a path to its value in the tree. You may assume that variables with the same path also have the same name, but if you do assume this, illustrate how your implementation would fail the `lookup` contract if the assumption was violated.

Expressions as Records

In this section we will examine an implementation of expressions as *records*. Abstractly, a record has a tag and a set of distinctly tagged values called the *fields* of the record.

$$\text{Record} \cong \text{Pair}(\text{Tag}, \text{Set}(\text{Field}))$$
$$\text{Field} \cong \text{Pair}(\text{Tag}, \text{Anytype})$$

For example, an arithmetic sum will be represented as a record with tag `sum`. The record will have a field tagged `addend` containing the record representing the augend of the sum; there will be a similar `augend` field. We'll assume there is a procedure

$$\texttt{field-content} : \text{Tag}, \text{Record} \to \text{Anytype}$$

for looking up the contents of a field within a record. Namely,

```
(field-content key record)
```

looks up and returns the value of the field in `record` whose tag is `key`.

The benefit of this implementation is that it is easy to attach useful data to expressions, and this can make some computations a lot easier.

For example, suppose a typical expression is going to be repeatedly printed in infix notation. We could apply our `expr->infix-string` procedure above each time the expression was to be printed, but, knowing that the resulting string will be needed repeatedly, it's sensible to compute it once and simply look it up when it's needed instead of constructing it again. The question is where to save the string and how to look it up? With expressions implemented as records, there is a simple answer to this question: store the string *in the expression.* More precisely, store the string in an `infix-string` field of the record representing the expression. Now the `expr->infix-string` procedure need only get the contents of this field:

```
(define (expr->infix-string expr)
  (field-content 'infix-string expr)).
```

The benefit here is that looking up the string in the record will typically be much faster than constructing it. For example, with records and fields implemented as tagged lists, looking up the infix string takes a *constant* number of operations for any expression, while the number of operations to construct an infix string is *linear* in the size of an expression.

To make this work, we have to modify the constructors of expression records so they include an `infix-string` field into the records, but that's easy:

```
(define (make-sum expr1 expr2)
 (attach-tag
   'sum
   (attach-tag
     'addend expr1)
   (attach-tag
     'augend expr2)
   (attach-tag
     'infix-string
     (string-append
       "(" (expr->infix-string expr1) "+" (expr->infix-tree expr2) ")"))))

(define (make-variable symb)
  (attach-tag
   'variable
   (attach-tag 'variable-name symb)
   (attach-tag 'infix-string symb)))

(define (make-number-expr num)
  (attach-tag
   'number-expr
   (attach-tag 'number num)
   (attach-tag 'infix-string (number->string num))))
```

Notice that we have paid a price to have this efficient infix-string by look up: the expression constructors now have to do the extra work of constructing and storing an infix string. This is a good tradeoff if an expression will indeed be printed as an infix string more than once, but is a complete waste in the case of an expression whose infix string is *never* needed.

This last problem can be overcome by initially omitting construction of an `infix-string` field, and then adding the field the first time it is needed. With this idea, the constructors become simple again:

```
(define (make-sum expr1 expr2)
 (attach-tag
   'sum
   (attach-tag
     'addend expr1)
   (attach-tag
     'augend expr2)))
```

```
(define (make-variable symb)
  (attach-tag
   'variable
   (attach-tag 'variable-name symb)))


(define (make-number-expr num)
  (attach-tag
   'number-expr
   (attach-tag 'number num)))
```

but the `expr->infix-string` procedure has to be modified to check for an `infix-string` field and to create one if it's missing:

```
(define (expr->infix-string expr)
  (begin
   (or (has-field? 'infix-string expr)
       (add-field!
         expr
         'infix-string
         (cond
           ((variable? expr)
            (symbol->string
             (field-content 'variable-name expr)))
           ((number-expr? expr)
            (number->string
             (field-content 'number expr)))
           ((sum? expr)
            (string-append
             "(" (expr->infix-string expr1)
             "+" (expr->infix-string expr2) ")" ))
           ((product? expr)
            (string-append
            "(" (expr->infix-string expr1)
            "*" (expr->infix-string expr2) ")" ))
           (else (error "Bad expression" expr)))))
    (field-content 'infix-string expr)))
```

Partial Evaluation and An Analyzing Evaluator

The techniques in the previous section for computing, saving, and looking up infix strings apply more generally to any repeatedly reused results of computation. A typical situation is when an expression is going to be evaluated repeatedly. For example, within a numerical computation package, such an expression might be a polynomial used for approximating the cosine function. The twist in this case is that while the polynomial expression is likely to be repeatedly evaluated—it is *not* likely to be evaluated many times with the *same values* for its variables, that is, in the same environment. But we can still find and save reusable results in an expression evaluation by separating the evaluation process into a part that depends only on the expression and not on the environment. This idea is known as *partial evaluation*.

To partially evaluate an expression, we define a higher-order procedure analyze which preprocesses an expression before any environment is available. The preprocessing yields a single procedure that will map any environment into the value of the expression in that environment. That is, analyze satisfies the contract:

((analyze <expr>) <env>) = (eval-arith <expr> <env>)

For example, if <expr1> was a sum of expressions <add1> and <add2>, we could define (analyze <expr1>) to be the procedure expr1-proc:

```
(define expr1-proc
 (lambda (env)
   (+ (eval-arith <add1> env)
      (eval-arith <add2> env))))
```

Now, whenever we want to evaluate <expr1> in some environment env, we could simply evaluate (expr1-proc env). This wins a little bit because it eliminates the need to parse <expr1> into <add1> and <add2> at each evaluation. The top-level parsing gets done only once—when we do the analysis of <expr1>. It will not be repeated when <expr> is evaluated in different environments by applying expr1-proc.

On the other hand, the recursive calls to eval-arith within the body of expr1-proc will wind up parsing add1 and add2 every time expr1-proc is applied, so the *only* parse that is not repeated is the top-level one. The problem is that we haven't saved the results of analyzing the subexpressions. We should have analyzed the subexpressions <add1> and <add2> recursively and then saved the resulting analyze-procedures for reuse. A simple place to save these analyze-procedures is *within* the expr1-proc procedure environment. We do this by including the following let bindings within the definition:

```
(define expr1-proc1
  (let ((add1-proc (analyze <add1>))
        (add2-proc (analyze <add2>)))
    (lambda (env)
      (+ (add1-proc env)
         (add2-proc env)))))
```

Now the parsing of <add1> and <add2> will not be repeated when expr1-proc is applied.

This approach leads to the following recursive definition of analyze:

```
(define (analyze expr)
  (cond
   ((number-expr? expr)
    (lambda (env) (field-content 'number expr))

    ((variable? expr)
     (lookup (field-content 'variable-name expr) env))
    ((sum? expr)
     (let ((addend-proc (analyze (addend expr)))
           (augend-proc (analyze (augend expr))))
       (lambda (env)
         (+ (augend-proc env)
```

```
            (addend-proc env)))))
    ((product? expr)
     (let ((multiplier-proc (analyze (multiplier expr)))
           (multiplicand-proc (analyze (multiplicand expr))))
       (lambda (env)
         (+ (multiplier-proc env)
            (multiplicand-proc env)))))
    (else (error "Bad expression" expr)))))

(define analyze1 (analyze <expr1>))
```

Computing the value of `<expr1>` in an environment by applying `analyze1` to the environment will be significantly more efficient than applying `eval-arith`, because *none* of the parsing of `<expr1>` gets repeated when `analyze1` is applied. So `analyze1` is a useful procedure to save for reuse when `<expr1>` needs to be evaluated in different environments—and the more often `<expr1>` needs to be evaluated, the bigger the win. Our record representation of expressions allows us to save analyze-procedures within expression records in the same way we did for infix strings.

Notice that now we could legitimately redefine the expression evaluation procedure in terms of `analyze`:

```
(define (eval-arith exp env)
  ((analyze exp) env))
```

However, with this definition there is no gain in efficiency—there is even a slight loss—when evaluating arithmetic expressions. That's because the value of `(analyze exp)` is recomputed whenever `eval-arith` is applied. Since the construction of the analyze-procedure requires a complete parsing of the `exp` argument, this version of `eval-arith` is doing essentially the same work as the original one, just in a different order.

But it's worth looking ahead a bit to observe that defining `eval` in terms of `analyze` in this way will really pay off for *recursively defined procedures*. The point is that the body of a recursive procedure is an expression that will be evaluated in many environments—a different one at each recursive call—during the course of a single recursive evaluation. In this case, partially evaluating the procedure body at the start of evaluation eliminates having to reparse the procedure body at each recursive call; this can speed up evaluation by a large constant factor (as much as ten or twenty for the 6.001 meta-circular Scheme evaluator) compared to an evaluator styled after `eval-arith`.

Exercise 7. Implement expressions as records which *may* have an `analyzed` field containing an analyzed-expression procedure. Then define `eval-arith` so it checks an expression record for an `analyzed` field, and creates such a field if it's missing. (Pattern your definition after the definition of `expr->infix-string` in the previous section.)