Mini Project 1 - Implementing our First Interpreter

EECS 662 - Programming Languages

Spring 2013

The objective of this miniproject is to develop your first rudimentary interpreter and extend that interpreter to include additional language features. You will start with the WAE language to get started with Scheme and interpreters. You will then extend WAE to allow multiple binding instances per `with` expression and to handle binary operations in a more general fashion.

Exercise 1 *Write a parser and interpreter for the WAE language discussed in class and presented in our text. (Yes, I know much of the code is in the book, so don't snicker.)*

1. *Write an type for representing the abstract syntax of the WAE language using* `define-type`*.*
2. *Write a function,* `parse-wae`*, that accepts the concrete syntax of WAE and generates a WAE data structure representing it. If your parser fails, it should throw an error using the* `error` *function discussed in class.*
3. *Write a function,* `interp-wae`*, that takes a WAE data structure and interprets it and returns a value. If your interpreter fails, it should throw an error using the* `error` *function discussed in class.*
4. *Write a function,* `eval-wae` *that combines your parser and interpreter into a single operation that parses and then interprets a WAE expression. Note that your interpreter should not be called if the parser fails. The* `error` *function should help you out with this.*

WAE ::=	*number* |
	id |
	{ + WAE WAE } |
	{ - WAE WAE } |
	{ with { *id* WAE} WAE }

Table 1: Grammar for WAE where *id* is an identifier and *number* is an integer.

IT IS USEFUL FOR Exercise 1 to realize that DrRacket interprets the `{` and `}` symbols the same way it interprets `(` and `)`. Thus, the expression `'{with {x 1} {+ x x}` will be processed by the Scheme interpreter in the same way as `(with (x 1) (+ x x))` would. This should make writing your parser much simpler.

The `error` function mentioned above is used in code examples in the text. It takes two arguments, a symbol and a string. The symbol identifies where the error occurs and the string is the error message. Make your errors meaningful.

Exercise 2 *Modify the abstract syntax, parser and interpreter for WAE to define a new language, WAEE (WAE Extended) to include the following new features in addition to your original WAE features:*

WAEE ::=	*number* |
	id |
	{ + *WAEE WAEE* } |
	{ - *WAEE WAEE* } |
	{ ∗ *WAEE WAEE* } |
	{ / *WAEE WAEE* } |
	{ with { *binding** } *WAEE* }
binding ::=	{ *id WAEE* }

Table 2: Grammar for WAEE where *id* is an identifier and *number* is an integer.

MULTI-ARMED `with` EXPRESSIONS – *As of now, each* `with` *expression has exactly one binding instance. Specifically, there is only one identifier*

bound in each `with` *expression. If one needs to define two (or more) identifier,* `with` *expressions must be nested:*

```
{with {x 2}
  {with {y 3} {+ x y}}}
```

Another option is to allow more than one binding instance in a `with` *expression:*

```
{with {{x 2}
       {y 3}}
      {+ x y}}
```

HANDLING BINARY OPERATIONS – *The original WAE language defines two binary operations,* `+` *and* `-`*, each of which have their own entry in the WAE data type. As noted in class, they are virtually identical – we frequently skipped details for* `-` *after going through* `+`*. Let's take advantage of this in WAEE by creating a single WAEE value for all binary expressions. Let's also create a table that maps binary operations to the function they perform. When parsing, you will generate the new WAEE binary operation value for all binary operations. When interpreting, you will look up the function associated with the binary operation and use that to calculate its value. To demonstrate this new feature, WAEE will include multiplication (*`*`*) and division (*`/`*) in addition to addition and subtraction.*

1. *Write a type for representing the abstract syntax of the WAEE language using* `define-type`*. Use your WAE abstract syntax definition as a starting point.*
2. *Write a function,* `parse-waee`*, that accepts the concrete syntax of WAEE and generates a WAEE data structure representing it. If your parser fails, it should throw an error using the* `error` *function discussed in class.*
3. *Write a function,* `interp-waee`*, that takes a WAEE data structure and interprets it and returns a value. If your interpreter fails, it should throw an error using the* `error` *function discussed in class.*
4. *Write a function,* `eval-waee` *that combines your parser and interpreter into a single operation that parses and then interprets a WAE expression. Note that your interpreter should not be called if the parser fails. The* `error` *function should help you out with this.*

EXTENDING THE WAE DATA TYPE to handle WAEE expressions is reasonably simple. You will need to replace the entries for `plus` and `minus` with a single entry for `binop`. Like `plus` and `minus`, `binop` will

have fields for left- and right-hand expressions. It will also have a third field that will contain the name of the operation it represents. Where the original data structure would represent `{+ 1 2}` as:

```
(plus (num 1) (num 2))
```

the new data structure would use the form:

```
(binop (op 'plus) (num 1) (num 2))
```

Of course you can choose any name for the constructor `op`.

Parsing will still need to account for individual binary operations separately, but all such operations will generate `binop` with the appropriate field values.

THERE ARE TWO NEW ERRORS that you need to handle in WAEE that are not present in WAE. First, if a `with` expression tries to define the same identifier twice, an error should be thrown. Second, if a bound expression in a `with` expression uses an identifier defined earlier or later in the same expression, an error should be thrown. For example:

```
{with {{x 2}
       {x 3}}
  {+ x y}}
```

is illegal because it attempts to define `x` twice. Similarly:

```
{with {{x 2}
       {y {+ x 5}}}
  {+ x y}}
```

should throw an error because the bound expression for `y` tries to reference the earlier definition for `x`. In contrast:

```
{with {x 3}
  {with {{x 2}
         {y {+ x 5}}}
    {+ x y}}}
```

is legal because the `x` in the bound expression for `y` can refer to the *first* instance of `x`

THERE ARE NUMEROUS WAYS to implement the lookup table for binary operations. The simplest is to define a list of symbol procedure pairs and search the list for the function you are looking for. Here's an interesting code snippet that you might find useful:

```
(define-type Binop
  (binop (name symbol?) (op procedure?)))

(define lookup
  (lambda (op-name op-table)
    (cond ((empty? op-table) (error 'lookup "Operator not found"))
          (else (if (symbol=? (binop-name (car op-table)) op-name)
                    (binop-op (car op-table))
                    (lookup op-name (cdr op-table)))))))
```

The type defines a pair containing a name and an operation. The function searches a list of these pairs, returning the operation associated with a particular name. An alternative to `lookup` would be using the built-in `assoc` function and an association list.

The table is defined once and stored somewhere – either as a global value using `define`, a local value using a `let` expression, or a local environment using `local`. For this project, the approach you choose is immaterial.

Notes

Most if not all the code for the WAE interpreter can be found in our text. I would encourage you to try to write as much of it as possible without referring to the textbook examples. Much of the code for WAEE can be adapted from the WAE interpreter. Note that there are places where `map` and `foldr` or `foldl` are particularly useful when dealing with lists of binding instances. If you're not familiar with these functions, ask in class or go online.

To give you a rough idea of the effort required for this miniproject, my code is about 100 lines long and took me rougly an hour to write and debug. I view this as a moderately difficult project at this point in the semester. Do not put it off.

This miniproject is adapted from Shriram Krishnamurthy's homework assignments associated with *Programming Languages: Application and Interpretation*. Any errors in the writeup are exclusively mine.