

Section Notes 9—Hitting the Left Paren

CS51—Spring, 2008

Week of April 20, 2008

1 Scheme51

From here on out, you'll be implementing SCHEME51, a subset of R5RS, in C++.

This week, you will implement the parser and enough of the object model to test your parsing strategy. In a later assignment, you will implement the evaluation rules that are necessary for turning a parsed expression into an interpreted program.

We have already provided some parts of the interpreter for you, in particular, the lexer and an object model. Take some time to review these sections to understand how they work. You may be surprised by what you do (or don't) need to do.

2 Design Overview

An interpreter requires several individual components working together, each of which handling a facet of the read-eval-print loop. The interpreter interacts with the user through a driver loop, which typically hands off most input to the parsing subsystem to produce logical structure from a stream of characters. The structured data that results from parsing can then be evaluated using “evaluation rules”.

3 The Object Model

Before you begin to think about the process of taking user input and turning it into data that can be evaluated, you must have some idea of what that data should look like. In particular, the basic unit processed by the interpreter is the “s-expression” (“symbolic” expression). An s-expression can be either a single element, such as a symbol or number, or a list of sub-expressions.

Each **type** of thing that makes up an s-expression needs to be represented by SCHEME51. In true object-oriented style, all possible components of an s-expression are objects of the type **value**, which is an abstract class that implements an interface for, at the moment, getting information about a particular value or printing the value. (For instance, in assignment 8, you must implement the **insert** method for each sub-type of **value** in order to print to a given output stream.)

3.1 Values

Every **value** type implements **insert** and **type**, which returns a **tag** indicating the subclass of the item. This will come in handy when different types need to be handled differently—when will this show up on assignment 8?

S-expressions will be composed entirely of pointers to values; that is, they will be **value ***s and a given s-expression will itself be a **value ***.

3.1.1 Nil

As you've seen in the past, `nil`, `()`, has special meaning in Scheme, as it marks the end of every proper list. There is also no reason for there to be more than one instance of `nil`. To provide this uniqueness property, the `nil` class implements an interface that allows access to one, and only one, instance of the class. This "singleton" design pattern means that only one copy of `nil` will exist. You may remember that we mentioned Design Patterns two weeks ago in section. The Singleton is one example of a Design Pattern. Refer back to Section Notes 7 for pointers to some resources that discuss Design Patterns.¹ You can access this copy via the static method `nil::instance` of class `nil`.

3.1.2 Booleans

Like the empty list, there are a restricted number of booleans in the world. In particular, `#t` and `#f`. You can access these two objects via the static methods `t` and `f` of the class `boolean`.

3.1.3 Numbers

In SCHEME51, there are only `longs`; you can't use floating point numbers. The `number` class is used to represent numbers in `s`-expressions. For a given numerical value, it is possible that there are two different `value *` objects that store the same number. How might you get around this?

3.1.4 Symbols

Symbols are actually quite interesting; a given symbol is distinguished by name, so it will eventually be important that for each name, we always get back the same symbol. For the current assignment, you don't need to worry about this part, but you will need to respect an interface that allows us to later change how things work. Therefore, we have provided yet another static method, `intern`, that takes a symbol name and returns the associated `symbol *`.

In assignment 9, having used the `intern` interface already will allow us to modify how symbols are created without having to change our parsing strategy.

3.1.5 Pairs

Finally, `pairs` are the heart of structured data in Scheme. Pairs are what make programming in Scheme possible, since they are the only means of combination provided by the language. Without them, we would have nothing more than an inert sequence of tokens.

A `pair` is composed of a `car` and a `cdr`, each of which are `value *`s. Your parser will probably need to create these and most parsed `s`-expressions will be returned as `pair *`s.

The printing routine for a `pair` is probably the most interesting, in particular because you must be able to handle both proper and improper lists (those that do not end with `nil`).

3.1.6 Procedures

You won't be working with procedures yet in assignment 8. Later, we'll see that procedures get subclassed into two different types, one for primitive (built-in) procedures and another for user-defined procedures (those created with `lambda`).

¹For a more detailed analysis of the Singleton Pattern and the challenges of implementing it in a Multithreaded program, advanced students may consider reading Meyers and Alexandrescu's paper titled C++ and the Perils of Double-Checked Locking that appeared in Dr. Dobbs Journal in the summer of 2004 – Google for it if you are interested.

3.2 Type checking

All code in the interpreter will be given `value *`s to work on. However, in most cases you expect that this generic pointer actually refers to a subclass of a particular type. For example, the `car` function needs a `pair` as its argument.

For this reason, a mechanism for checking the type of a `value *` is an important piece of infrastructure. This component has the following requirements:

- In as little code as possible, be able to specify that a certain `value *` needs to be of a certain subtype.
- Raise an appropriate exception if the type does not match, causing a useful error message to print.
- Result in a pointer of the correct type, e.g. `pair *`.

To meet the third requirement, you will have to perform what is known as a *dynamic down-cast*. Down-casting means converting a base class type to a child class type (as you know, the opposite conversion is always valid). The C++ syntax is

```
value *someFunction(value *arg)
{
    pair *p;

    ...
    p = dynamic_cast<pair *>(arg);
    ...
}
```

The `dynamic_cast` operator is useful because it will return `NULL` if the argument is not, in fact, of the correct type. However, needing to do a dynamic cast, `NULL` check, and raising an exception for every type check violates our first requirement that the code be concise. Abstraction is warranted. You should come up with a way to wrap this functionality more conveniently. Hint: a template might be useful.

4 Syntax—Lexing and Parsing

The “lexer” and “parser” work together to take input from a user and turn it into a `value *` of the proper form.

4.1 Lexing

A “lexical analyzer” or “lexer” tokenizes the input for use by the parser. Tokens are logical subdivisions of the input; usually, tokens are distinguished by whitespace or other pieces of syntax. For instance, the (and) characters will always be treated as individual tokens in our system, but multiple characters may be joined together to form a longer token—for instance, the symbol for boolean truth, `#t`. The lexer will use a struct, `token`, with which to transmit information to the parser about input tokens. You should read the lexer carefully to understand exactly what functionality it will provide and what your parser will need to take care of.

4.2 Parsing

The “parser”, built on top of the lexer, adds *structure* to the stream of tokens by creating an *s-expression*, which is then handed back to the driver loop which can either print the expression immediately (using the overloaded `<<` operator) or evaluate it. For assignment 8, we just print it, but this does not mean that your program will simply echo back the characters that are sent to it. Parsers allow us to represent languages in a structured format that permits interpretation and transformation as we will discuss by example in this section.

5 Style

Your interpreter code had better be pretty and, ideally, elegant and easy to read. This is especially true because you will be working with your code for two assignments. Design mistakes or poor style will be more difficult to recover from because you'll be stuck with them (and, potentially, bugs caused by them).

- Value consistency.
- Use descriptive variable names. This helps debugging a lot and makes our lives better. Vowels are friendly and like to nuzzle your hand. Keep them; they're housebroken.
- Capitalize `ClassNames`, and put method names in camelCase.
- Put variable declarations all in one place—i.e., at the top of the code block in which you use them. This makes it easier to find them or change how you initialize them.
- Comment well. If you don't, you will forget what was going on. And we won't be able to understand your code well enough to give partial credit. Good comments explain complex code or give rationale for a design decision.
- Getting it to “work” is not enough. It should work well, and the solution should be understandable and clean. This will make your life easier later.
- Magic doesn't exist. Neither should magic numbers. Use `const` for constants that would have been `#defined` in C.
- Decompose the problem into multiple functions. You don't want to debug a one-function parser. We don't want to grade it.