PS2: S-Expressions

Lior Zur-Lotan, Shahaf Shperberg and Avi Hayoun

November 2, 2017

Contents

1 S-Expressions (Symbolic-Expressions)

S-Expressions are a notation for nested lists, consisting of pairs with values as leaves. The grammar of the Scheme language is based on s-expressions. A result of this is that a snippet of Scheme code and the AST of the same snippet are extremely similar - both are nested lists, i.e. trees.
1.1 The Grammar of S-Expressions

\[ \langle S-Expr \rangle ::= \langle Atom \rangle \]
\[ \ | ( \langle S-Expr \rangle^+ . \langle S-Expr \rangle ) \]
\[ \ | ( \langle S-Expr \rangle^* ) \]

\[ \langle Atom \rangle ::= \langle String \rangle \]
\[ \ | \langle Integer \rangle \]
\[ \ | ... \]

\[ \langle String \rangle ::= " \langle Char \rangle^* " \]

\[ \langle Integer \rangle ::= 0 \ | \langle Digits1-9 \rangle \langle Digits0-9 \rangle^* \]

The empty list \( \) is called \textbf{nil}. S-expressions that end in nil are called \textbf{proper-lists}, while those which end in any non-nil value are called \textbf{improper-lists}.

Another thing to note is that, just like in Scheme, s-expression lists are actually nested pairs. This means that the s-expressions \((1 . ())\) and \((1)\) are semantically equivalent.

1.2 Types of Lists and How to Create Them

As we’ve seen, the main part of the S-Expression grammar, and thus the Scheme grammar, is lists - both proper and improper. This means that lists are \textit{very} important, so let’s see how to create them!

There are various ways to create proper- and improper-lists in Scheme: the \textsf{quote} keyword, the \textsf{list} function, the \textsf{cons} function, the dot notation for pairs and the \textbf{Quasi-Quote}.

1.3 Quasi-Quote

Quasi-Quote (QQ) is one of the list constructors in Scheme. It’s more flexible than \textsf{quote (‘)}, \textsf{list} or \textsf{cons} as it allows for mixing symbols with expression evaluations in the context of one QQ.

Just like \textsf{quote}, QQ has two syntactic forms: the expanded form \texttt{(quasiquote ...)}, and its syntactic sugar form ‘ (a "back quote", aka "that button above the tab key, to the left of the number 1").

The QQ has two helper mechanisms, each of which also have expanded and sugar forms:

1. \texttt{(unquote ...)} and its syntactic sugar form , (a comma);

2. \texttt{(unquote-splicing ...)} and its syntactic sugar form ,@ (a comma followed by an at sign).

\textbf{Remark.} The expansion of all three of these forms follows the same rules as the expansion of ‘ (quote): the single s-expression immediately following the sugared form is wrapped in the expanded form. For example: ‘\((a,b,@c d)\) is expanded into \texttt{(quasiquote (a (unquote b) (unquote-splicing c) d))}.

When QQ is used, a \textsf{quote} context is initiated. \textsf{unquote} allows us to momentarily escape the \textsf{quote} context and evaluate the unquoted s-expression. \texttt{unquote-splicing} both escapes an entire list from the \textsf{quote} context as well as “unwraps” the list. Note that this “unwarpping” is only one level deep; a single \texttt{unquote-splicing} cannot be used to flatten a deeply nested list.
1.3.1 The Differences between quote and quasiquote

Given the following definition:

\[
\text{(define b 5)}
\]
\[
\text{(define s (list 1 '(2)))}
\]

Note the differences in the values of the following expressions:

<table>
<thead>
<tr>
<th>Input</th>
<th>Quote</th>
<th>Quasi-Quote</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(a b c)</td>
<td>'(a b c)</td>
<td></td>
</tr>
<tr>
<td>(a b c)</td>
<td>(a b c)</td>
<td></td>
</tr>
<tr>
<td>'(a ,b c)</td>
<td>'(a ,b c)</td>
<td></td>
</tr>
<tr>
<td>(a ,b c)</td>
<td>(a ,b c)</td>
<td></td>
</tr>
<tr>
<td>'(a ,s c)</td>
<td>'(a ,s c)</td>
<td></td>
</tr>
<tr>
<td>(a ,s c)</td>
<td>(a ,s c)</td>
<td></td>
</tr>
<tr>
<td>'(a ,@s c)</td>
<td>'(a ,@s c)</td>
<td></td>
</tr>
<tr>
<td>(a ,@s c)</td>
<td>(a ,@s c)</td>
<td></td>
</tr>
</tbody>
</table>

1.4 Everything is an S-Expression

As noted at the beginning, code in Scheme is a collection of s-expressions. Additionally, as you may have already noticed, data in Scheme are also made up of s-expressions. What this means is that code and data are interchangeable in Scheme.

Given that our primary focus in this course is code which generates code, this is an important/interesting detail to note. If the data that is output from a function is an s-expression, then it is also code.

As a result of this fact, there are a couple of important points to understand:

- The distinction between values and structures of expressions.
- The correct expansion and nesting of expressions.

1.4.1 Understanding the Structures of S-Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expanded Form</th>
<th>Evaluates in Scheme to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(or)</td>
<td>(or . () )</td>
<td>#f</td>
</tr>
<tr>
<td>'a</td>
<td>(quote . (a .()))</td>
<td>a</td>
</tr>
<tr>
<td>(+ . (1 . (2 .())))</td>
<td>(+ . (1 . (2 .())))</td>
<td>3</td>
</tr>
<tr>
<td>(- (- 1 2) 3)</td>
<td>(- . ((- . (1 . (2 .()))) . (3 . ())))</td>
<td>-4</td>
</tr>
<tr>
<td>'','e</td>
<td>(quasiquote . ((quote . ((unquote . (e . ()))) . ()))) . ())</td>
<td>' [value of e]</td>
</tr>
</tbody>
</table>
1.5 Generating Code from Data

One type of code generation which we can try is that of code from data. We would like to be able to write a function which takes any valid s-expression and outputs valid Scheme code which creates that expression. For example, given ’(1 . (2 . ())) our function would return (list 1 2).

We will call our function describe. It will take an s-expression and return a Scheme expression that generates that expression.

```scheme
1 (define describe
2   (lambda (e)
3     (cond
4       ((list? e) ‘(list ,@(map describe e)))
5       ((pair? e) ‘(cons ,(describe (car e))
6         ,(describe (cdr e))))
7       ((vector? e) ‘(vector ,@(map describe (vector->list e))))
8       ((or (null? e) (symbol? e)) ‘,e)
9       (else e))))
```