On the Unimportance of Syntax
Lecture Notes for cs173, Fall 2001

sk and rob
September 7, 2001
updated September 25, 2001

Syntactic sugar causes cancer of the semicolon.
—Alan Perlis

Syntax is the Vietnam of programming languages
—?

1 Knowledge, Information, and Data

1.1 A Simple Example

Picture a thermometer. Looking at it, we can see the mercury rising or lowering as time passes. We can see the numbers and hash marks on the glass of the thermometer. These types of quantities/observations fall under the category of knowledge.

A physicist, however, can tell us relevant information about the thermometer, such as the temperature at different times.

But if we’d like to process lots of temperature and time values, we’ll need a computer. Computers, though, can’t understand information—they only understand data.

1.2 A More Related Example

Picture the DrScheme environment. Consider the GUI, the welcome message, the prompt, the text entered by the user, and the text returned by the computer. These entities are in the realm of knowledge.

The information available in this environment is, for example, that the user typed in (+ 1 2), and DrScheme returned 3.
In CS 173, we want to work at the *data* level. Because we have to get our data from somewhere, though, we start by converting information to data.

## 2 Information to Data

Let’s write a program that processes other programs. This is a profound leap, and is one of the great things that you can accomplish with computer science. Specifically, we are going to write an interpreter in Scheme for the invented language Rip.

Let’s try to describe all Rip programs.

### 2.1 First Attempt to Write Down all Rip Programs

An RP (Rip Program) is either

- a number
- `{ + number number }
- `{ * number number }

This is close, but consider the program `{ + 2 `{ + 1 3 }`. This program does not exist in our specification. It should be clear that our specification needs to be *recursive*.

### 2.2 Second Attempt to Write Down All Rip Programs

An RP is either

- a number
- `{ + RP RP }
- `{ * RP RP }

Now we’ve captured the *information* of being a Rip program in a formal way, but we still don’t have *data*. We will need a data definition.

## 3 Data Definitions

### 3.1 Data Representation

Expressions contains two types of information. Information *about* the expression, and information *in* the expression.
The *about* of a number expression is that it's a number. The *in* information is the number itself.

Similarly with expressions of the form \{ + RP RP \}: The information about it is that the expression is an add expression. The information in it is the left and right pieces of the add expression.

Note that the *about* information acts as a tag on data. Once we read the tag, we know for certain what kinds of things to expect inside the expression, and we act accordingly. As an aside, this is why program templates (from *How to Design Programs*) are so effective.

### 3.1.1 Lists or Structs for Representation?

Lists are appropriate when you have a collection of an arbitrary number of things of the same kind. Structs are appropriate when you want to group together a fixed number of things of different kinds. You should eventually have a deep understanding of this, but for now, take our word for it.

Number expressions, add expressions, and multiplication expressions should all be represented as structs.

### 3.2 Finally, We Get Data

We now have what we need to write our data definition. Data definitions are extremely important. For one, they give us the code template for writing functions that operate on this data.

An RP is either

- `(make-numE number)`
- `(make-addE RP RP)`
- `(make-multE RP RP)`

The data definition implies the following struct definitions:

`(define-struct numE (num))`

`(define-struct addE (left right))`

`(define-struct multE (left right))`
4 Writing a Simple Interpreter

Let’s write an interpreter that works on Rip expressions.

Following the program design rules of How to Design Programs, write down examples of the data:

\[
(\text{numE } 1) \\
(\text{make-addE } (\text{numE } 1) (\text{numE } 4)) \\
(\text{make-multE } (\text{make-addE } (\text{numE } 1) (\text{numE } 2)) (\text{numE } 3))
\]

Let’s write down a contract for the interpreter.

;; interp : RP → numE
;; interp reduces the RP to a numE.

Now we write out the program template.

\[
(\text{define } (interp \ an-rp))
\]

\[
(\text{cond})
\]

\[
[(\text{numE? } an-rp) \ldots (\text{numE-num an-rp}) \ldots]
\]

\[
[(\text{addE? } an-rp) \ldots (\text{interp } (\text{addE-left an-rp})) \ldots (\text{interp } (\text{addE-right an-rp})) \ldots]
\]

\[
[(\text{multE? } an-rp) \ldots (\text{interp } (\text{multE-left an-rp})) \ldots (\text{interp } (\text{multE-right an-rp})) \ldots]
\]

Principle 1 (the goodness of templates) The templates write a heck of a lot of code for you. Use them.

So what does our interpreter actually look like? Well, it only requires a few small changes to the template.

\[
(\text{define } (interp \ an-rp))
\]

\[
(\text{cond})
\]

\[
[(\text{numE? } an-rp) \ an-rp]
\]

\[
[(\text{addE? } an-rp) (\text{numE+ } (\text{interp } (\text{addE-left an-rp}))
\ (\text{interp } (\text{addE-right an-rp})))]
\]

\[
[(\text{multE? } an-rp) (\text{numE* } (\text{interp } (\text{multE-left an-rp}))
\ (\text{interp } (\text{multE-right an-sp})))]
\]

The functions \( \text{numE+} \) and \( \text{numE*} \) simply add and multiply \( \text{numE} \) expressions. You need to write them yourself, but they are very simple.

5 Changing Syntax

Uh oh. Your inbox just received this:
From: Your Boss
Subject: Syntax is Changing!

The syntax of our language is changing.
Attached is the newest version.
Please update your data definition and your interpreter immediately!

Let’s assume the syntax change is that angle brackets are now used for addition.
As an example, the new syntax for \( \{+ \ 2 \ 3\} \) would be \( <2,3> \).
The most important part of today’s lecture: The data definition and the interpreter do not change a lick. You can immediately respond to your boss with the message:

It’s all good.

**Principle 2 (the non-importance of syntax)** Making purely syntactic changes to a programming language should not affect your data definition nor any interpreters you write for that language.

### 6 A Parser (not discussed in class)

Syntax isn’t all bad, of course. In fact, the whole point of syntax is that instead of writing the program

\[
(make\text{-}mult\ (make\text{-}add\ (make\text{-}mult\ (make\text{-}num\ 2\ (make\text{-}num\ 3))\ (make\text{-}num\ 4))\ (make\text{-}add\ (make\text{-}num\ 1\ (make\text{-}num\ 1))})
\]

the programmer can just write

\[
\{\star\ \{\star\ 2\ 3\}\ 4\ \{\star\ 1\ 1\}\}
\]

Syntax is information—convenient (hopefully) for a human, but useless to a computer. We need to translate this information to data. This is what a **parser** does.

### 6.1 Reading Characters

We need some way of getting characters the programmer types into some convenient form which we can then parse. As far as CS 173 is concerned, reading characters is not an interesting problem. Luckily, Scheme makes it very easy for us by providing the function **read**. In its simplest use, **read** takes no arguments. When invoked, it prompts the user to enter text. With some restrictions, the entered text is converted to a Scheme expression. Lucky for us, **read** converts Rip expressions to appropriate Scheme list representations, as shown in the following example.

---

5
6.1.1 Examples of Read

Welcome to DrScheme, version 103p1.
Language: Beginning Student.

> (read)

4

4

> (read)

(+ 1 2)

(cons ’+ (cons 1 (cons 2 empty)))

> (read)

(+ {* 2 3} {* 4 5})

(cons ’+ (cons ’* (cons 2 (cons 3 empty)))
 (cons (cons ’+ (cons 4 (cons 5 empty))) empty)))

6.2 Using the Design Recipe to Make a Parser

6.2.1 Our Definition of a List-Represented Program

An LRP (list-represented program) is either
- a number
- (cons ’+ (cons LRP (cons LRP empty)))
- (cons ’* (cons LRP (cons LRP empty)))

Since we don’t have our program as data yet, we can’t use structs. And since
we aren’t using structs, we don’t get any predicates for free. So we write our
own:
(def (list-rep-num? LRP)
  (number? LRP))

(def (list-rep-add? LRP)
  (and
   (list? LRP)
   (not (empty? LRP)))

6
\[
(eq? \ (first \ LRP) \ '+))
\]

\[
(define \ (list-rep-mult? \ LRP)
(\ and
(\ list? \ LRP)
(\ not \ (empty? \ LRP))
(eq? \ (first \ LRP) \ '*))\)
\]

We similarly write selectors \(list-rep-add-left\), \(list-rep-add-right\), \(list-rep-mult-left\), and \(list-rep-mult-right\).

### 6.2.2 Contract and Header for the Parser

\[
;; \ parse : LRP \rightarrow RP
;; \ parse \ consumes \ an \ LRP \ and \ returns \ a \ parsed \ RP \ program \ (define \ (parse \ UP) \ldots )
\]

### 6.2.3 Examples

Here are some examples of how \(parse\) should work:

\[
(parse \ 4) = (make-numE \ 4)
\]

\[
(parse \ (cons \ '*' \ (cons \ (cons \ '+' \ (cons \ 1 \ (cons \ 2 \ empty))))) \ (cons \ 3 \ empty))) =
(make-mult \ (make-add \ (make-num \ 1) \ (make-num \ 2)) \ (make-num \ 3)))
\]

### 6.2.4 The Template

\[
(define \ (parse \ an-LRP)
(\ cond
 \[(\ list-rep-num? \ an-LRP) \ldots \ an-LRP \ldots ]
 \[(\ list-rep-add? \ an-LRP) \ldots \ (parse \ (list-rep-add-left \ an-LRP))
\ldots \ (parse \ (list-rep-add-right \ an-LRP)) \ldots ]
 \[(\ list-rep-mult? \ an-LRP) \ldots \ (parse \ (list-rep-mult-left \ an-LRP))
\ldots \ (parse \ (list-rep-mult-right \ an-LRP)) \ldots ]))
\]

### 6.2.5 The Body

\[
(define \ (parse \ an-LRP)
(\ cond
 \[(\ list-rep-numE? \ an-LRP) \ (make-numE \ an-LRP)]
 \[(\ list-rep-addE? \ an-LRP) \ (make-addE \ (parse \ (list-rep-add-left \ an-LRP))
\ (parse \ (list-rep-add-right \ an-LRP)))]
 \[(\ list-rep-multE? \ an-LRP) \ (make-multE \ (parse \ (list-rep-mult-left \ an-LRP))
\ (parse \ (list-rep-mult-right \ an-LRP)))]))
\]

### 6.2.6 Testing

We’ll leave that to you.