SCAN : A Static Code Analyser for JavaScheme

by

Tammy Yap

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

at the

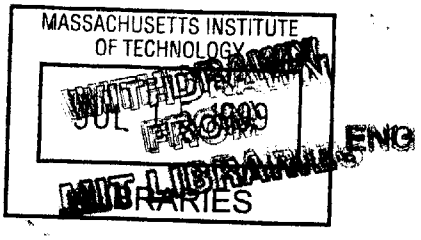
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Friday, May 21, 1999
[June 1999]

Author .....
Department of Electrical Engineering and Computer Science
Friday, May 21, 1999

Certified by .....
Saman P. Amarasinghe
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses



SCAN : A Static Code Analyzer for JavaScheme

by

Tammy Yap

Submitted to the Department of Electrical Engineering and Computer Science

on

May 21, 1999

In partial fulfillment of the requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The static code analyzer performs analysis on code fragments without any explicit execution in order to determine potential syntactical or run-time errors. The analyzer provides beginning Scheme programmers with a debugging tool that complements the standard Scheme debugger. The analyzer indicates the exact locations at which errors occur to facilitate quick corrections. It checks for correct parentheses and Scheme keyword syntactical structure. Using heuristic techniques, it determines potential typographical errors and calls attention to them. Undefined variables used in code are pointed out. Set-based analysis techniques are developed and implemented in the analyzer, and used to locate possible run-time errors caused by argument type violations.

Thesis Supervisor: Saman P. Amarasinghe
Title: Assistant Professor, Electrical Engineering and Computer Science

Table of Content

List of Figures

List of Tables

Chapter 1

Introduction

Computers are essential in today's society, and computer programming is an essential skill. For many, however, learning the art of programming is a difficult task. At the heart of programming, is the ability to describe a process or algorithm in a precise fashion that can be understood by a computer. Countless hours are spent by novice programmers trying to master this technique, which is so different from how humans normally interact. Not only do they have to learn high-level programming concepts, they also have to learn rigid syntax and semantic rules of the particular programming language they are using.

Scheme[1], a dialect of LISP, is a language that imposes minimal semantic requirements on the programmer, thus easing the burden placed on the beginning programmer. Using Scheme allows emphasis to be placed on learning programming concepts, instead of the rules of the language.

Scheme is very flexible and expressive, but as a result, does not perform many semantic checks on user programs. While this expressiveness is valued in experienced programmers, it allows many types of programming errors to arise, most often in programs written by beginning programmers.

Introduction to SCAN

In this thesis, problems encountered by novice programmers have been identified, and static program analysis techniques have been developed to provide warnings of possible programming errors. A significant portion of this project is devoted to the development and implementation of a set-based analysis system.

SCAN, which stands for *Static Code ANalyzer*, was developed to perform static analysis of Scheme code. It is meant to be a debugging tool for the beginner Scheme programmer that complements the standard Scheme debugger.

The Scheme interpreter does not evaluate code until the value of an expression is required and as a result, some programming errors may go undetected. SCAN however, checks *all* the code given to it. It looks for correct syntax of parentheses and special forms, undefined variables, possible typographical errors and violations of argument type restrictions.

Previous work : DrScheme and MrSpidey

A significant project that has dealt specifically with problems encountered in learning Scheme is DrScheme[2]. It is a comprehensive programming environment for Scheme that is able to catch typical syntactic mistakes of beginning programmers.

The developers had noted that experienced Scheme programmers develop code in a batch-oriented fashion, re-evaluating the entire text buffer upon each execution. Thus, DrScheme was built in a way that intrinsically ensures that the beginning developer uses this programming style -- forcing a complete re-evaluation each time. This avoids the common problem of the students altering the text of a previous definition, and forgetting to evaluate it and update the definition in the environment, resulting in inexplicable errors.

The programming environment in which SCAN is implemented, JavaScheme, allows the programmer to evaluate code one line at a time. Thus, SCAN is unable to impose this batch-oriented programming style on the student.

Additionally, the DrScheme system includes an integrated static debugger called MrSpidey[3] that allows the programmer to browse program invariants and their derivations. MrSpidey implements set-based analysis which is improved upon by SCAN to

obtain information to help the student programmer locate potential code errors quickly and easily by pinpointing the exact location in the student's code at which an error occurs.

Chapter 2

Common Scheme Errors

Scheme is used in Structure and Interpretation of Computer Programs (SICP), an introductory programming course required of all students majoring in computer science and electrical engineering at MIT.

The SICP laboratory is large enough for forty students, but is usually only staffed by one or two laboratory assistants at any one time. These LAs are kept busy by a constant queue of students requesting help with their programming assignments.

The high demand for LA attention results from a number of factors. Although there is a standard debugger available for Scheme, it is usually not very helpful to a programmer who is still trying to get used to Scheme. Scheme's error messages are not helpful in pinpointing mistakes because they give no indication of the location of error within the code. This is a problem because a run-time error will often occur at a point in the code that is located far from the actual coding mistake.

Laboratory assistants have recognized a number of common problems encountered by students :

Some of these include:

1. Dynamic typing - data type errors.
2. REPL pitfalls - changing definitions of procedures without re-evaluating them.
3. Typographical errors - mistyping procedure or variable names.
4. Parentheses - misplaced or mismatched parentheses.

Dynamic typing

Scheme is a dynamically typed language. A statically typed language such as C requires that each variable is declared to be of a certain type upon creation. In Scheme, however, a

program variable is not constrained to be any specific type, and over the course of execution, it may even take on values of different types.

A common programming error made by students is to apply procedures to variables of the incorrect type. Due to the nature of dynamic typing, such an error cannot be detected by the system until the code is actually executed, unlike in static typing where type restriction violations are immediately apparent at compile-time.

Suppose a student writes a program that contains an illegal procedure application. If the branch containing the erroneous code is never executed during testing, the code appears to work correctly. The student goes on to integrate the code into a larger program, at which point, the bug surfaces. Assuming the previous code to be correct, the student may spend a lot of time looking for a non-existent error in the new code.

```
(define (treefind tree key)
  (let ((left (car tree)) (right (cdr tree)))
    (if  (pair? left)
         (let ((leftfind (treefind left key)))
           (if  (null? leftfind)
                (treefind right key)
                leftfind))
         (if  (= left key)
              right
              (if  (pair? right)
                   (treefind right key)
                   (= right key)))))))
```

`Treefind` is supposed to return `#f` if `key` is not in the binary tree `tree`. If `key` is a left child of its parent, the right child is returned. If `key` is a right child, `#t` is returned. The procedure appears to work correctly on the following test case.

```
(define a '((5 . (6 . 7)) . (2 . ((8 . 9) . 4))))
(treefind a 2)
;Value: ((8 . 9) . 4)
```

However, if `key` is not in the left subtree of `tree`, and the right subtree is a single node, as in the case of `key` being `4` and `tree` being `((8 . 9) . 4)`, the program calls `(treefind 4 4)`. A run-time error occurs when `(car 4)` is evaluated.

If `treefind` was integrated into a large program that searched many trees, the student may end up searching for a bug in that program, without realizing `treefind` was buggy.

In another instance, the student may use some buggy code in an iterative procedure and run it for a few thousand cycles. At the last cycle, the program enters a branch where an error is located for the first time. Execution exits with an error message at this point, wasting time and effort spent during computation of the earlier iterations.

SCAN addresses these possible problems by ensuring that all code is analysed, not just code that will be evaluated.

REPL pitfalls

Scheme code is developed using the Read-Eval-Print-Loop. The user enters text in the buffer and instructs Scheme to enter the REPL sequence. Scheme reads the buffer, evaluates the code, prints a result and loops back to the beginning. The user may also command Scheme to only read the expression just preceding the cursor, to save having to evaluate the entire buffer again.

Occasionally, the student may edit an earlier definition of a procedure in the buffer, but forget to call on Scheme to evaluate the new text. Scheme retains the current environment in memory, which contains the most recent values assigned to each variable. In this case, Scheme still maintains the previous definition of the newly-updated procedure. When the student uses this procedure in another piece of code, he gets results different from what he

expects. This problem is not obvious, because on screen, the code appears to be completely correct, but seems to be executing incorrectly.

SCAN enables the user to look out for this problem by allowing the user to examine the current definitions held in the environment.

Typographical errors

Typographical errors are hard to catch because it does not always register in a person's consciousness that what appears on screen differs from what he thought he typed. As a result, a student may repeatedly read over a piece of code on screen and still not notice a typing mistake. On execution, variables will not contain the values they are expected to, and seemingly inexplicable errors arise.

Confusion often arises when very similar variable names are intentionally used in a program, such as `studentrecord`, `studentrecords` and `student-record`. It is considered good programming practice to use variable names that are easily distinguishable from each other, but novice programmers are not always made aware of this.

In the example below, in the fourth line of the code `cpd_rate` is used where `cpd-rate` was intended. The final result is not what the user intended.

```
(define cpd_rate 1.05)
(define (interest princ cpd-rate years)
   (if   (= years 0)
         (* princ cpd_rate)
         (interest princ (* 1.05 cpd-rate) (- years 1))))

(interest 1000 1 4)
;Value: 1050.
```

SCAN points out pairs of variable names that are very similar, and that may possibly be mistaken for each other.

Parentheses

With parentheses occurring throughout the code, a beginner can easily lose track of and misplace parentheses.

Execution of the following code :

```
(define (fact n)
   (if  (= n 0)
        1
        (* n ((fact (- n 1))))))
```

returns the error message :

```
; the object 1 is not applicable
```

This reply is cryptic to the beginning programmer and does not explicitly point out the extra parentheses on the fourth line, that are the cause of the error. SCAN will allow high-lighting of the exact error, which will make the mistake immediately obvious.

Chapter 3

Description of SCAN

SCAN is a static debugging tool. It examines Scheme code without performing any execution or evaluation. It checks for correct Scheme syntax, undefined variables, possible typographical errors and violations of argument type restrictions.

SCAN is implemented as a special form, and is called with a Scheme expression as its only argument. When the student calls SCAN on a piece of code that he has been having problems with, SCAN checks it for errors and lists the locations in the code where the errors occurred.

SCAN differs from the standard Scheme debugger in a number of ways. Most significantly is that the standard debugger is invoked by Scheme if an error is found while performing a dynamic check on the code during execution. The debugger then allows the user to trace through the series of procedure calls that resulted in the illegal call. SCAN, on the other hand, is invoked by the user on a piece of code, and it locates errors by statically analyzing the code, not executing it. SCAN is also able to detect multiple errors, unlike the debugger which finds one error at a time.

Using static analysis to detect errors can also find potential errors that will not arise in execution.

For example :

```
(define proc
  (lambda (a)
    (case a
      ((0) (car a))
      ((1) (cons 1 nil))
      (else (cons a (proc (- a 2)))))))
```

```
(proc 7)
```

has an obvious error in the fourth line, in trying to take the `car` of the number `0`. The error does not surface unless `proc` is called with a positive even integer. In the example above, execution proceeds smoothly, and the debugger is never invoked. Running SCAN on the code however, will detect the potential error.

One other important difference is that SCAN is able to indicate the actual location of the error within the code. Popular imperative languages like C and Java come with Integrated Development Environments, in which the user compiles code. Upon discovering an error, the compiler halts and highlights the particular line at which the error occurs. SCAN provides useful information much in this fashion, unlike the debugger that opens a separate window that only displays the procedure call that generated the error, with no obvious reference point to the source code.

What SCAN does

Scan performs a number of checks and operations on the code it receives. In order of execution, these are :

1. Parenthesis and Scheme syntax checking
2. Code transformation
3. Typographical checking
4. Undefined variables
5. Set-based analysis

At the end of each stage, SCAN may proceed on to the next stage with information obtained, or return an informative error output to the user.

Parenthesis and Scheme syntax checking

SCAN firstly ensures that every opening parenthesis has a matching close parenthesis. If this is not the case, SCAN informs the user of this, and exits.

For example, calling SCAN on the following code :

```
(begin (display "Hello world! :)")
```

returns the message "ScanError: Incorrectly parenthesized code."

Next, SCAN checks that all the essential Scheme procedures as defined in the Revised(4) Report on the Algorithmic Language Scheme[1], receive the correct number of arguments (e.g. `car`, which requires one argument), and that any instances of Scheme keywords have the proper structure (e.g. `let` which must have the syntax "`(let (<bindings>) <body>)`").

No type checking is performed at this point. SCAN only examines the structure of the code and the syntax of keywords. Any errors at this stage causes SCAN to exit with an appropriate error message.

For example :

```
(define myproc (lambda (car a b)))
```

returns the error "Poorly formatted lambda expression : (lambda (car a b))."

Code transformation

To simplify its subsequent operation, SCAN makes cosmetic changes to the code. It removes comments, converts all non-literal characters to lowercase (Scheme is case-insensitive) and replaces non-literal sequences of whitespace by single spaces. Next, syntactic sugar and macro occurrences in the code are expanded. None of these changes alter the meaning of the code, but provides a consistent input format for SCAN to analyse. *Table 3.1* lists the cases in which SCAN transforms the code.

Old code	Transformed code
(define (a v1 ...) <*body*>)	(define a (lambda (v1 ...) <*body*>))

Table 3.1: Code transformation rules

Old code	Transformed code
(quote <*val*>)	'<*val*>
(list ...) and '(...)	(cons ...)
'(1 . 2)	(cons 1 2)
(caar a) and other such macros	(car (car a)) and so on
(let* ((a 1) (b 2) ...) <*body*>)	(let ((a 1)) (let ((b 2)) (...))
'()	nil
#\	#\space

Table 3.1: Code transformation rules

Typographical checking

SCAN looks for common typographical errors such as transposed or omitted letters in variable names, including undefined ones. It also checks for commonly substituted characters such as "_" and "-" or "O" and "0". The algorithm used at this stage is :

Close(a, b)

- If a or b are of length 1, return false.
- If, ignoring case, a and b are identical, return false.
- If a and b are of the same length :
  - If a and b differ by 2 consecutive transposed characters, return true.
  - If a and b differ by 1 character which is easily mistaken for another, return true. (these are "_" and "-", "O" and "0" and "l" and "1")
- If the length of a and b differ by one, and the shorter string is identical to the longer string missing the ith character (where i is not the length of the longer string), return true.

Examples of pairs of variable names that would cause SCAN to generate a warning :

```
new_rate          new-rate
numerator         nuemrator
hairs             chairs
integerlist       integerlst
student1          studentl
```

Pairs of variable names that will not generate any warning :

`r`	`er`
`nameset`	`nameset2`
`MyFunction`	`myfunction`
`accumulation`	`acmuculation`

SCAN prints a list of pairs of variable names it has determined to be suspiciously similar, and continues on to the next stage.

Undefined variables

When SCAN encounters variables that are undefined within the provided code, it requests the current definition of them from the top-level environment, and prepends the definition of these variables to the code. At this point, any variables used in the code that have never been defined will cause SCAN to return an appropriate error message to the user.

Set-based analysis

Set-based analysis involves the creation of labels for every symbol or expression in the code, and assigning sets of possible values to each label in order to determine if type restrictions of procedure arguments are violated. This will be discussed in further detail in *Chapter 4* .

Chapter 4

Set-Based Analysis

The type-checking functionality of SCAN is implemented using set-based analysis and is used to identify possible errors in the types of arguments passed to procedures.

A Scheme expression may be a constant, such as `15` or `"hello"`, a variable, such as `datestring`, or involve subexpressions, as in `(car lst)`. Non-constant expressions may take on different values during execution. Set-based analysis involves deriving the set of all these possible values for each expression.

The location of each expression in the code is assigned a label and the set of values that may be assigned to that location is referred to as S($label). SCAN generates appropriate constraints describing members of each of these sets. This process will be illustrated with an example.

The process of set-based analysis is composed of the following steps :

1. Assigning labels and label restrictions
2. Generating constraints
3. Propagating constraints
4. Compiling value sets and set-type checking

Assigning labels and label restrictions

SCAN generates a code tree and labels every node, each of which is an expression in the original code. An in-order numbering system is used; i.e. the current expression is assigned a label before its sub-expressions are labelled. The labels are assigned in numerically ascending order : $0, $1, and so on. An example is shown in *Figure 4.1*.

Code example :

```
(define incr (lambda (a) (+ 1 a))) (car (incr 3))
```

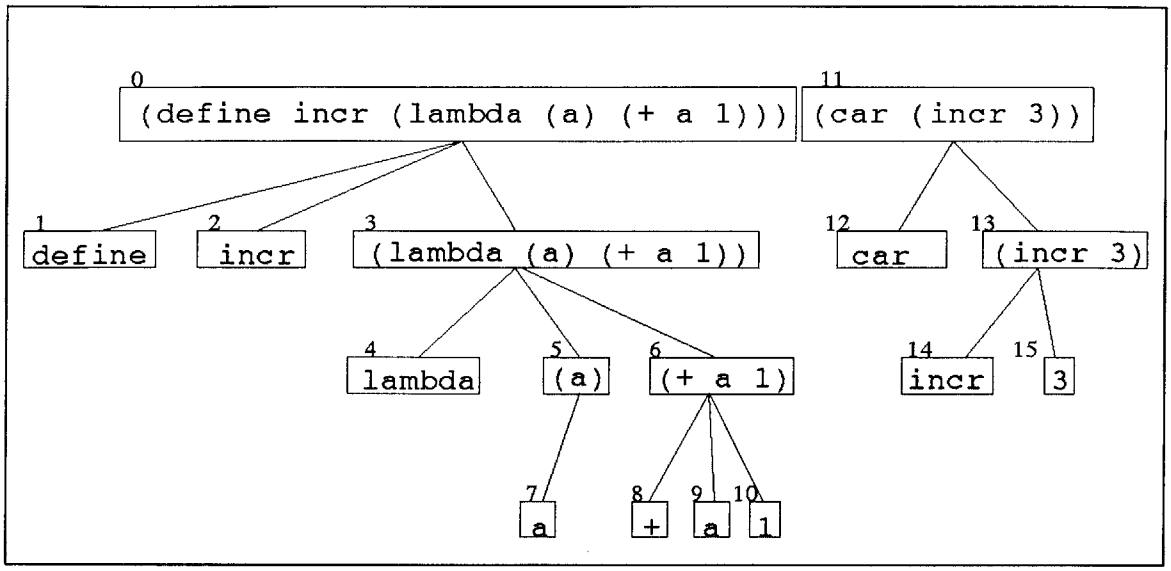


Figure 4.1: Labelled code tree example

SCAN handles these labels by means of label restrictions. Along with information on the location of the label in the code, each label restriction also specifies the position of the label in the current expression, with the first position being 0. For example, label $3 in the example has a corresponding label restriction of position 2. Because these label restrictions are used later to indicate procedure applications and their arguments, any label restrictions on expressions that are not subexpressions of a procedure application are set to position -1. The labels on the topmost level, $0 and $11 have restrictions of position -1, indicating that they are not procedure applications.

The complete list of label restrictions that are generated at this stage are listed in *Table 4.1.*

No.	Expression	Position
$0	(define incr (lambda (a) (+ a 1)))	-1
$1	define	0
$2	incr	1

Table 4.1: Generated label restrictions

No.	Expression	Position
$3	(lambda (a) (+ a 1))	2
$4	lambda	0
$5	(a)	1
$6	(+ a 1)	2
$7	a	0
$8	+	0
$9	a	1
$10	1	2
$11	(car (incr 3))	-1
$12	car	0
$13	(incr 3)	1
$14	incr	0
$15	3	1

Table 4.1: Generated label restrictions

Generating constraints

After the code has been labelled, SCAN generates constraints from each element in the code tree. Constraints are generated using the rules in *Table 4.2*.

Code	Constraint
$c^{\$1}$, where c is a constant	$c \in S(\$1)$
(define $a^{\$1}$ $b^{\$2}$) for each instance of a^{lx} in b or in following code	$S(\$2) \subset S(\$1)$ $S(\$1) \subset S(lx)$

Table 4.2: Constraint generation rules

Code	Constraint
(let $((v1^{\$1}\ a^{\$2})\ldots)$ <expr 1>…<expr n>$^{\$3}$)$^{\$4}$	$S(\$3) \subset S(\$4)$
for each $(vx^{lx}\ ay^{ly})$	$S(ly) \subset S(lx)$
for each vx^{lz} in <expr m>	$S(lx) \subset S(lz)$
(letrec $((v1^{\$1}\ a^{\$2})\ldots)$ <expr 1>…<expr n>$^{\$3}$)$^{\$4}$	$S(\$3) \subset S(\$4)$
for each $(vx^{lx}\ ay^{ly})$	$S(ly) \subset S(lx)$
for each vx^{lz} in <expr m> or <a>	$S(lx) \subset S(lz)$
(lambda $(v1^{\$1}\ldots)$ <expr 1>…<expr n>$^{\$2}$)$^{\$3}$	(lambda ($1…) $2) $\in S(\$3)$
for each vx^{ly} in <expr m>	$S(ly) \subset S(lx)$
(and <expr 1>$^{\$1}$… <expr n>$^{\$2}$)$^{\$3}$	$S(\$2) \subset S(\$3)$
for each <expr x>^{lx}	$\dfrac{\text{nil or \#f} \in S(lx)}{S(lx) \subset S(\$3)}$
(and)$^{\$1}$	\#t $\in S(\$1)$
(begin <expr 1>…<expr n>$^{\$1}$)$^{\$2}$	$S(\$1) \subset S(\$2)$
(case $k^{\$1}$ $((d1\ldots)\ldots)\ldots$(else…<expr n>$^{\$2}$))$^{\$3}$	$S(\$2) \subset S(\$3)$
for each dx^{lx}	$dx \in S(lx)$
for each dx in $((dx\ldots)\ldots$<expr y>^{ly})	$\dfrac{dx \in S(\$1)}{S(ly) \subset S(\$3)}$
for each $((dx\ldots))$	$\dfrac{dx \in S(\$1)}{\text{\#t} \in S(\$3)}$
for each dx^{lx} in $(dx\ldots)$ and each k^{lk} in each <expr y>, in each $((dx\ldots)\ldots$<expr z>)	$S(lx) \subset S(lk)$

Table 4.2: Constraint generation rules

Code	Constraint
(cond (<test 1>$^{\$1}$ ...)...(else...<expr n>$^{\$2}$))$^{\$3}$ for each (<test x>^{lx}...<expr y>^{ly})	$S(\$2) \subset S(\$3)$ $\frac{\text{any non nil value} \in S(lx)}{S(ly) \subset S(\$3)}$
(or <expr 1>$^{\$1}$...<expr m>^{lm})$^{\$2}$; for each <expr x>^{lx}	$\frac{\text{any non nil value} \in S(lx)}{S(lx) \subset S(\$2)}$
(cons $a^{\$1}$ $b^{\$2}$)$^{\$3}$	(cons \$1 \$2) $\in S(\$3)$
(car $a^{\$1}$)$^{\$2}$	$\frac{(\text{cons } \#lx\ \#ly) \in S(\$1)}{S(\#lx) \subset S(\$2)}$
(cdr $a^{\$1}$)$^{\$2}$	$\frac{(\text{cons } \#lx\ \#ly) \in S(\$1)}{S(\#ly) \subset S(\$2)}$
(proc_a$^{\$1}$ vx^{lx}...)$^{\$2}$ where proc_a is not a Scheme defined procedure for each vx^{lx} or (cons vx^{lx} vy^{ly}) that corresponds to argument az^{lz} in the lambda definition of proc_a	$\frac{(\text{lambda } \#\text{vars } \#\text{val}) \in S(\$1)}{S(\#\text{val}) \subset S(\$2)}$ $\frac{\text{a lambda value} \in S(\$1)}{(S(lx) \text{ or } (\text{cons } lx\ ly)) \subset S(lz)}$

Table 4.2: Constraint generation rules

Applying these rules to the example code, the constraints generated are :

$$S(\$3) \subset S(\$2) \tag{4.1}$$

$$(\text{lambda } (\$7)\ \$6) \in S(\$3) \tag{4.2}$$

$$\frac{(\text{lambda } \#\text{vars } \#\text{val}) \in S(\$8)}{S(\#\text{val}) \subset S(\$6)} \tag{4.3}$$

$$\frac{\text{a lambda value} \in S(\$8)}{S(\$9) \subset S(\text{corresponding argument 1})} \tag{4.4}$$

$$\frac{\text{a lambda value} \in S(\$8)}{S(\$10) \subset S(\text{corresponding argument 2})} \tag{4.5}$$

$$+ \in S(\$8) \tag{4.6}$$

$$1 \in S(\$9) \tag{4.7}$$

$$S(\$7) \subset S(\$10) \tag{4.8}$$

$$\frac{(\text{cons \#lx \#ly}) \in S(\$13)}{S(\#\text{lx}) \subset S(\$11)} \tag{4.9}$$

$$\text{car} \in S(\$12) \tag{4.10}$$

$$\frac{(\text{lambda \#vars \#val}) \in S(\$14)}{S(\#\text{val}) \subset S(\$13)} \tag{4.11}$$

$$\frac{\text{a lambda value} \in S(\$14)}{S(\$15) \subset S(\text{corresponding argument 1})} \tag{4.12}$$

$$S(\$2) \subset S(\$14) \tag{4.13}$$

$$3 \in S(\$15) \tag{4.14}$$

As previously mentioned, label restrictions with non-negative positions indicate procedure applications. Thus, during this stage, label restrictions of locations corresponding to special form usage (such as `define` and `lambda`) are set to position -1. After this is done, only applied procedures will have label restrictions of position 0, with their corresponding arguments having position restrictions of their argument number. All other label restrictions will have -1 assigned. For the example, the restrictions will be updated to those in *Table 4.3*. Note the changes in the positions for the restrictions marked with a *.

No.	Expression	Position
$0	(define incr (lambda (a) (+ a 1)))	-1
$1 *	define	-1
$2 *	incr	-1
$3 *	(lambda (a) (+ a 1))	-1
$4 *	lambda	-1
$5 *	(a)	-1
$6 *	(+ a 1)	-1
$7 *	a	-1
$8	+	0

Table 4.3: Updated label restrictions

No.	Expression	Position
$9	a	1
$10	1	2
$11	(car (incr 3))	-1
$12	car	0
$13	(incr 3)	1
$14	incr	0
$15	3	1

Table 4.3: Updated label restrictions

Constraint propagation

After the initial constraint set has been generated, it is used to create the final set values for each variable through constraint propagation. Every pair of constraints is compared to determine if a new constraint can be added to the set. New constraints are added and compared with the old constraints to generate further constraints as well. The rules listed in *Table 4.4* are used to propagate constraints.

Constraint 1	Constraint 2	New constraint
$a \in S(b)$	$S(b) \subset S(c)$	$a \in S(c)$
$a \in S(b)$	$S(b) \cap$ type x $\subset S(c)$	If a is of type x, $a \in S(c)$
$a \in S(b)$	$S(b) \cap \neg$type x $\subset S(c)$	If a is not of type x, $a \in S(c)$
$a \in S(b)$	$\frac{\text{pattern} \in S(b)}{\text{subconstraint}}$	If pattern matches a, subconstraint
$a \in S(b)$	$\frac{\text{any non nil value} \in S(b)}{\text{subconstraint}}$	If a is not null, subconstraint
$S(a) \subset S(b)$	$S(b) \subset S(c)$	$S(a) \subset S(c)$
$S(a) \subset S(b)$	$S(c) \cap$ type x $\subset S(a)$	$S(c) \cap$ type x $\subset S(b)$
$S(a) \subset S(b)$	$S(c) \cap \neg$type x $\subset S(a)$	$S(c) \cap \neg$type x $\subset S(b)$

Table 4.4: Constraint propagation rules

Constraint 1	Constraint 2	New constraint
$S(a) \subset S(b)$	value of type x $\in S(a)$	value of type x $\in S(b)$
$S(a) \cap$ type x $\subset S(b)$	value of type x $\in S(a)$	value of type x $\in S(b)$
$S(a) \cap \neg$type x $\subset S(b)$	value of type y $\in S(a)$	value of type y $\in S(b)$
$\frac{\text{pattern} \in S(a)}{\text{subconstraint}}$	value of type x $\in S(a)$	If pattern matches value, subconstraint
value of type x $\in S(a)$	$\frac{\text{any non nil value} \in S(b)}{\text{subconstraint}}$	If type x is not null, subconstraint
(lambda $(b\ldots)\ldots) \in S(a)$	$\frac{\text{a lambda value} \in S(a)}{\text{args } (d\ldots) \text{ to proc at } a}$	$S(d) \subset S(b)$
$proc \in S(a)$ where proc is a Scheme-defined procedure that returns a known type	$\frac{\text{a lambda value} \in S(a)}{\text{args } (d\ldots) \text{ to proc at } a}$	$(\lambda\ ()\ (proc\ d\ldots)) \in S(a)$
$(\lambda\ ()\ (proc\ldots)) \in S(a)$ where proc is a Scheme-defined procedure that returns known type x	$\frac{\text{pattern } (\lambda\ \#a\ \#v) \in S(a)}{\text{subconstraint } S(\#v) \subset S(b)}$	$(proc\ldots)$ of type x $\in S(b)$

Table 4.4: Constraint propagation rules

New propagated constraints from the example are :

$$4.1\ \&\ 4.2 : (\text{lambda } (\$7)\ \$6) \in S(\$2) \tag{4.15}$$

$$4.1\ \&\ 4.13 : S(\$3) \subset S(\$14) \tag{4.16}$$

$$4.4\ \&\ 4.6 : (\lambda\ ()\ (+\ \$9\ \$10)) \in S(\$8) \tag{4.17}$$

$$4.2\ \&\ 4.16 : (\text{lambda } (\$7)\ \$6) \in S(\$14) \tag{4.18}$$

$$4.3\ \&\ 4.17 : (+\ \$9\ \$10) \text{ of type number} \in S(\$6) \tag{4.19}$$

$$4.11\ \&\ 4.18 : S(\$6) \subset S(\$13) \tag{4.20}$$

$$4.12\ \&\ 4.18 : S(\$15) \subset S(\$7) \tag{4.21}$$

$$4.8\ \&\ 4.21 : S(\$15) \subset S(\$10) \tag{4.22}$$

$$4.14\ \&\ 4.21 : 3 \in S(\$7) \tag{4.23}$$

$$4.19\ \&\ 4.20 : (+\ \$9\ \$10) \text{ of type number} \in S(\$13) \tag{4.24}$$

$$4.14 \,\&\, 4.22 : 3 \in S(\$10) \tag{4.25}$$

Set type checking

After no new constraints can be generated, SCAN proceeds to examine the set assigned to each label for invalid values. This is done by examining both the full set of label restrictions and the set of values for that label.

Position 0

If the label restriction is of position 0, this indicates a procedure application. All the elements of this set have to be either Scheme procedures or lambda values. Any other elements, such as string or number constants, generate a warning.

Labels \$8, \$12 and \$14 are all of position 0 in the example.

$S(\$8) = \{+\}$
$S(\$12) = \{\text{car}\}$
$S(\$14) = \{(\text{lambda } (\$7)\ \$6)\}$

The three corresponding value sets only contain valid values.

All other positions

Procedure arguments are labelled with a label restriction position greater than 0. For each argument, it has a set of procedures to which it may be applied. This set of procedures is the value set of the position 0 just preceding the current position. This would effectively be the current position subtracted from the current label number. This set of procedures should only contain Scheme procedures or lambda values as previously discussed. If the current expression is an argument to a Scheme procedure, SCAN checks that all constants and expressions in the current set are of the correct type. If it is an argument to a lambda value, SCAN checks that the lambda value is not being applied to too many arguments.

In the example, labels \$9, \$10, \$13 and \$15 have positions greater than 0.

$S(\$9) = \{1\}$

$S(\$10) = \{3\}$
$S(\$13) = \{(+\ \$9\ \$10)$ of type number$\}$
$S(\$15) = \{3\}$

$9 has a position 1 restriction, which means that it is an argument of position 1 to values in the set of S($(9 - 1)) = S($8). Since S($8) only contains +, SCAN does not report anything. However, $13 is an argument of position 1 corresponding to $12, which means that `car` is applied to a number. An argument to `car` can only be a pair (i.e. an argument of type "`(cons la lb)`"), so SCAN reports a possible error : "Label $13, argument 1 to `car` may receive an illegal input of type number : (+ $9 $10)".

Chapter 5

Implementation

Introduction to JavaScheme

Currently, multiple versions of Scheme are used on a number of different computer platforms. Students may run Scheme on HP workstations in the SICP laboratory, on UNIX workstations around the MIT campus, or on their personal computers. As a result, many Scheme implementations must be maintained for the various platforms, which makes updating Scheme a complicated and tedious affair.

This flexibility also creates a problem, when students who develop code outside the laboratory need to seek help from the teaching staff, who work only in the laboratory. Because the Scheme environment is altered by the code that has been executed during the current session, it is difficult to reproduce the exact same environment, and usually the same error, on another computer. Thus, code often cannot be debugged without physically being at the computer on which it is running. Even when the students go the laboratory, they experience difficulties in accessing the code stored on a remote computer, or in reproducing the original environment in which the error occurred.

JavaScheme was developed to overcome these particular difficulties. First, it is implemented in Java, which resolves the multi-platform issue since it can be run in any Java-enabled web browser, regardless of the type of machine. Second, it uses a central repository for students code, which eliminates the problem of transferring code between systems.

Most importantly, JavaScheme is a networked environment, which permits communication between teaching staff and remote students. It will eventually allow for interactive instruction in the form of virtual "over-the-shoulder" assistance, where the student's work-

space may be viewed by a physically remote laboratory assistant (LA). The LA may view both the code and the current working environment, and even make changes and run them.

Interface to SCAN

SCAN is written in Java and integrated into JavaScheme. It is made available to students as a special form and is called with a single Scheme expression as its only argument.

```
(scan (begin (define (square x) (* x x)) (square '(4))))
```

Output of SCAN

SCAN returns its results in a window like the one shown in Figure 5.1. It displays its find-

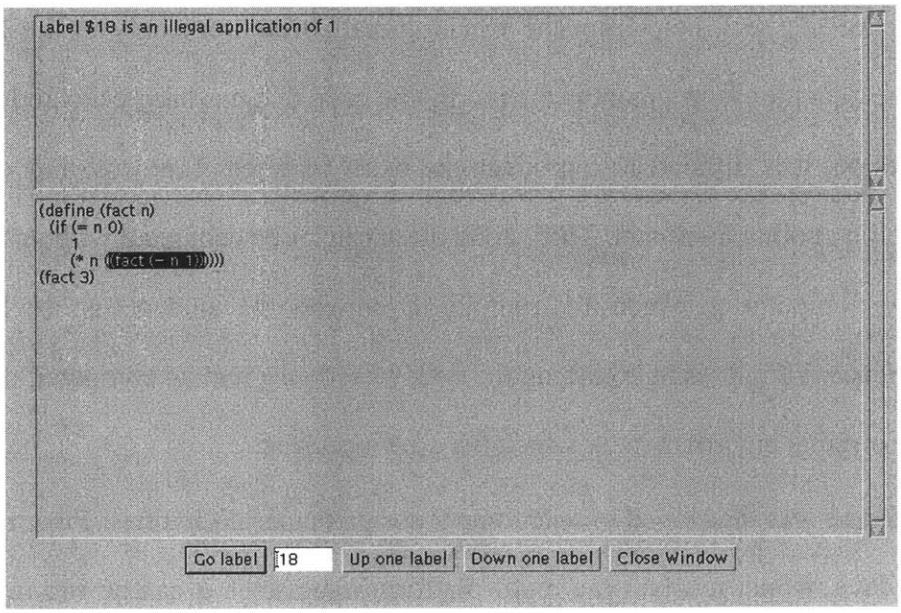


Figure 5.1: Sample output of SCAN

ings, and allows the user to examine the current definitions of code, as well as the label assignments used by SCAN, in order to pinpoint errors.

Limitations of implementation

In order to keep the scope of this project at a manageable level, the current implementation of SCAN does not handle every possible legal Scheme expression. The excluded range is restricted to a minimum and is mostly Scheme notation that is unlikely to be used by beginning Scheme programmers.

Character set

SCAN assumes the valid character set to be the keystrokes generated on a standard 101-key keyboard. More specifically, it is the characters : **a** through **z**, **A** through **Z**, **0** through **9**, those in *Table 5.1* and the 3 whitespace characters : ' ' (space), \t (tab), \n (carriage return).

!	@	#	$	%
^	&	*	(	)
-	_	+	=	\
~	;	:	"	'
<	>	.	?	/

Table 5.1: Valid characters

The characters **{**, **}**, **[**, **]**, and **|** are currently reserved for future extensions to Scheme. The characters **`,`** and **`\`** are used in `quasiquote` notation, which SCAN does not handle. These characters should not be used, but if they are, SCAN treats them the same way alphabets are treated, and considers them as part of identifiers. This may cause SCAN to return inaccurate results.

Redefinition of essential procedures

SCAN relies on knowing the required input types for essential Scheme procedures as defined in the Revised(4) Report on the Algorithmic Language Scheme [1]. Thus, while Scheme allows redefinition of those procedures, doing so will cause SCAN to return inac-

curate results. For example, if + is redefined as `(lambda lst (car (car lst)))`, SCAN will consider numerical arguments for + correct, whereas in actual fact, the proper argument will be a pair.

Scheme notation not handled

The following notations are not handled by SCAN. Some of these involve setting specific values, and it is difficult to adapt set-based analysis to accurately handle them. Some others are considered fairly rare usages in the target user group of students and thus were not included.

`=>`	`apply`	`assoc`
`assq`	`assv`	`delay`
`do`	`for-each`	`force`
`list->string`	`make-string`	`make-vector`
`map`	`quasiquote` (‘,’ and ‘,’)	`set!`
`set-car!`	`set-cdr!`	`string-fill!`
`string->number`	`unquote`	`unquote-splicing`
`vector` (and ‘#’)	`vector-fill!`	`vector-set!`

If an unhandled keyword is used, SCAN will report an error and exit. If an unhandled procedure is used, SCAN reports a warning, and proceeds, but it may be unable to correctly detect all the errors in the code.

Chapter 6

Summary and Further Work

The main objective of this project was to develop a tool that will aid beginning Scheme programmers in debugging. The new addition to JavaScheme is meant to complement, not replace the standard Scheme debugger, and to provide debugging assistance in areas that are not well served by that tool.

SCAN has achieved this objective. SCAN provides effective debugging information in many instances of errors, including those described in *Chapter 2*. It supplements the debugger in the various areas mentioned in *Chapter 3*.

Published literature on set-based analysis for Scheme [3] [4] only describes the process for a small subset of the language, concentrating on the main constructs of the language, such as `car`, `cons` and `if`. SCAN implements the analysis system and extends it to include other Scheme keywords and procedures.

According to the scope that was defined for this project, SCAN's coverage of Scheme syntax is comprehensive enough to prove useful to the target group of students.

Strengths

SCAN takes a conservative approach to creating value sets. All possible values of an expression are taken into consideration. Thus if there is any chance at all of an error occurring, SCAN will inform the user of it. This results in more comprehensive testing than manually possible, as even extensive execution of test code may not cover all possible cases.

SCAN is able to pinpoint the location of errors in the code, which aids students in quickly finding and changing their mistakes, increasing their productivity.

Checking for typographical errors helps students as it also discourages them from naming variables with extremely similar names, and potentially developing an undesirable programming habit.

Limitations

Static analysis has its limitations. The value of some expressions simply cannot be determined without evaluation, such as `(= b 0)`. Mutation also causes difficulties because of the temporal nature of mutation. The execution order of statements affect the result of mutations. Because of these, SCAN is limited in the accuracy of its error reports.

However, the analysis system implemented in SCAN is robust and extensible. It has the potential to be upgraded to increase SCAN's capabilities, making it a useful tool for even more advanced programmers.

Further Extensions to SCAN

A number of extensions can be made within the framework of SCAN to enhance its functionality. Besides enlarging the set of Scheme syntax that SCAN accurately analyses, some other functions that SCAN can conceivably provide are :

1. Procedure header generation
2. Recursion checking
3. Structuring

Procedure header generation

By using back propagation, some input type restrictions on user-defined procedures can be determined. For example :

```
(define (length l)
  (cond   ((null? l) 0)
          (else (+ 1 (length (cdr l))))))
```

With appropriate constraints, it can be determined that the `else` clause is only activated when `l` is not null, and in that case, `l` has to be a pair. Furthermore, `(cdr l)` has to have the same characteristic, and it can be concluded that the argument `l` to `length` needs to be a list.

Recursion checking

Infinite recursion is another problem commonly encountered by students. Recursive procedures have two main features : one or more calls made to itself, and a parameter to itself that moves towards a fixed point.

Static analysis can detect a recursive procedure by checking for the first feature. A simple static check for infinite recursion may be to determine if the parameters to the recursive call are reducing towards zero, or at least monotonically changing.

Structuring

Occasionally, a properly parenthesized piece of Scheme code may not be performing its intended function. For example, the following code segment illustrates a legal but unintended use of `cond`.

```
(define (length l)
   (cond   ((null? l) 0)
           (else 1 + (length (cdr l)))))
```

This procedure returns 0 when called on any list, because, for any non-null input, it returns `(length (cdr l))`.

Errors such as these are not always visually obvious. Static analysis may be able to detect unusual, albeit legal, usage of procedures, such as in this case of the `else` clause containing expressions that serve no obvious purpose.

Acknowlegments

The author wishes to credit her thesis supervisor, Saman P. Amarasinghe, for providing the guidance and vision for this project, and for his invaluable advice and feedback throughout the course of development and writing. She also wishes to thank Mark A. Herschberg for his assistance in drafting and proofreading this report, and for his moral and emotional support while she was completing this thesis.

References

[1] Clinger and Rees. Revised(4) Report on the Algorithmic Language Scheme. *http://swissnet.ai.mit.edu/~u6001/manuals/r4rs/r4rstoc.html*.

[2] Findler, Flanagan, Flatt, Krishnamurthi and Felleisen. DrScheme: A Pedagogic Programming Environment for Scheme. *http://www.cs.rice.edu/CS/PLT/Publications/*

[3] Flanagan, Flatt, Krishnamurthi, Weirich and Felleisen. Catching Bugs in the Web of Program Invariants. *In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 23-32, May 1996*

[4] Flanagan and Felleisen. Set-based analysis for full Scheme and its use in soft-typing. *Rice University Computer Science TR95-253*

Appendix A

Pattern matching

The constraint propagation system relies on a pattern matcher for its operation. SCAN has its own pattern matching system built in. It matches pattern strings with proper Scheme expressions. The BNF grammar for patterns is :

<pattern> = <pattern string> | #<pattern variable name> | (<pattern>*)

<pattern string> = <character>+

<character> = <normal character> | ' ' | \# | \\

<normal character> = <A>..<Z> | <a>..<z> | <0>..<9> | ! | @ | $ | % | ^ | & | *

| - | _ | + | = | ~ | : | < | > | . | ? | /

<pattern variable name> = <normal character>+

Working from left to right, pattern string and input string are matched character for character. Whenever a pattern variable name is encountered, the next Scheme value is matched to the pattern variable, following which, character for character matching resumes.

If the end of the pattern is reached, and the end of the input string is encountered at the same time, the string is matched successfully.

Appendix B

Code

B.1 SCAN.java

```
package projects.javascheme.tools.scan;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

import projects.javascheme.tools.RGScheme.eval.Environment;

public class SCAN extends Frame implements ActionListener{

  String finalcode;
  int corr[];

  private Button golabel;
  private TextArea typelabel;
  private Button uplabel;
  private Button downlabel;
  private TextArea resulttxt;
  private TextArea resultcode;
  private int currlab;
  private Button closescan;

  public SCAN(String original, Environment env) throws ScanError{

    String resultstr;
    int i, j;
    int token1[], token2[], token3[], token4[], token5[], token6[];

    String code = original;
    finalcode = original;

    ////////////////////////////////////////////////////////////////////
    // Simple syntax checking
    ////////////////////////////////////////////////////////////////////
    if (!syntax.checkParen(code)) {
      throw new ScanError("Incorrectly parenthesized code.");
    }

    ////////////////////////////////////////////////////////////////////
    // transform code
    ////////////////////////////////////////////////////////////////////
    token1 = Parser.token(code);
    code = Parser.condition(Parser.uncomment(code));
    token2 = Parser.token(code);

    ////////////////////////////////////////////////////////////////////
    // More syntax checking
    ////////////////////////////////////////////////////////////////////
    resultstr = syntax.checkKeys(Parser.brsplit(code));

    Vector result = Parser.transform(code, 0);
    code = (String)result.elementAt(0);
    token3 = (int[])result.elementAt(1);


    ////////////////////////////////////////////////////////////////////
    // typo checking
    ////////////////////////////////////////////////////////////////////
    Vector tree = generate.gentree("(" + code + ")", 0);
    tree = mu.subVector(tree, 0, tree.size() / 2);
    Vector symbols = new Vector();
```

```
for (i=0; i<tree.size(); i++) {
  String s = (String)tree.elementAt(i);
  if (Parser.atom(s) && (symbols.indexOf(s) == -1)) {
    symbols.addElement(s);
  }
}
Vector res = dbg.closevars(symbols);
if (res.size() > 0) {
  resultstr = resultstr +
    "The following pairs of variables appear close to each other.\n";
  resultstr = resultstr + "Make sure no typing errors were made.\n";
  resultstr = resultstr + res.toString();
}

////////////////////////////////////////////////////////////////////////
// look for undefined values
////////////////////////////////////////////////////////////////////////
Vector newcode = Parser.undef(code, env);
code = (String)newcode.elementAt(0);
int offset = ((Integer)newcode.elementAt(1)).intValue();

token4 = token3;
token5 = null;
token6 = token3;
if (offset > 0) {
  result = Parser.transform(Parser.condition
                        (Parser.uncomment
                         (code.substring(0, offset))), 0);
  String defs = (String)result.elementAt(0);
  token5 = Parser.token(defs);
  code = defs + " " + code.substring(offset);
  token6 = Parser.token(code);
  offset = defs.length() + 1;
  finalcode = defs + "\n" + original;

  int k = Parser.token(code.substring(0, offset)).length;
  token4 = new int[token3.length + k];
  for (i=0; i<k; i++) {
    token4[i] = -1;
  }
  for (i=0; i<token3.length; i++) {
    token4[i + k] = token3[i];
  }
}

////////////////////////////////////////////////////////////////////////
// syntax analysis
////////////////////////////////////////////////////////////////////////
Vector candr = generate.gencon(code);
Vector labels = (Vector)candr.elementAt(1);
resultstr = resultstr + "\n" +
  labelrestriction.setcheck(constraint.propagate
                        ((Vector)candr.elementAt(0)),
                        labels, token6);

////////////////////////////////////////////////////////////////////////
// correspondence with old code
////////////////////////////////////////////////////////////////////////

int lr[] = new int[code.length()];

for (i=0; i<lr.length; i++) { lr[i] = -1; }
for (i=0; i<labels.size(); i++) {
  lr[((labelrestriction)labels.elementAt(i)).start] = i;
}
corr = new int[labels.size()];
int k = 0;
for (i=0; i<lr.length; i++) { // for the ith character
  if (lr[i] > -1) { // if it is token lr[i]
    if (i < offset) {
      corr[lr[i]] = token5[lr[i]];
    } else {
```

```
        if (token4[k] > -1) { // if it is a token in the original code
          corr[lr[i]] = token1[index(token2, token4[k])] + offset;
        } else {
          corr[lr[i]] = -1;
        }
      }
      k++;
    }
  }

  resulttxt = new TextArea(resultstr, 10, 80,
                           TextArea.SCROLLBARS_VERTICAL_ONLY);
  resulttxt.setEditable(false);
  add(resulttxt);
  resultcode = new TextArea(finalcode, 20, 80,
                            TextArea.SCROLLBARS_VERTICAL_ONLY);
  resultcode.setEditable(false);
  add(resultcode);
  golabel = new Button();
  golabel.setLabel("Go label");
  golabel.setActionCommand("GO");
  golabel.addActionListener(this);
  add(golabel);
  typelabel = new TextArea("", 1, 5,
                           TextArea.SCROLLBARS_NONE);
  typelabel.setEditable(true);
  add(typelabel);
  uplabel = new Button();
  uplabel.setLabel("Up one label");
  uplabel.setActionCommand("UP");
  uplabel.addActionListener(this);
  add(uplabel);
  downlabel = new Button();
  downlabel.setLabel("Down one label");
  downlabel.setActionCommand("DOWN");
  downlabel.addActionListener(this);
  add(downlabel);
  currlab = -1;

  closescan = new Button();
  closescan.setLabel("Close Window");
  closescan.setActionCommand("CLOSE");
  closescan.addActionListener(this);
  add(closescan);

  setLayout(new FlowLayout());
  setSize(720, 500);
  setTitle("SCAN results");

  show();
}

public int[] label(int labelno) {
  int indices[] = null;
  if ((labelno > -1) && (labelno < corr.length)) {
    int x = corr[labelno];
    if (x > -1) {
      indices = new int[2];
      indices[0] = x;
      indices[1] = x + Parser.nextval(finalcode.substring(x)).length();
    }
  }
  return indices;
}

private void setlabel() {
  String x = typelabel.getText();
  try {
    Integer i = new Integer(x);
    hilitelabel(i.intValue());
  } catch (NumberFormatException e) {
  }
```

```
  }

  private void hilitelabel(int labelno) {
    int ix[] = label(labelno);
    if (ix != null) {
      currlab = labelno;
      resultcode.setSelectionStart(ix[0]);
      resultcode.setSelectionEnd(ix[1]);
      typelabel.setText("" + currlab);
    }
  }

  private void labup() {
    int newlab = currlab + 1;
    while ((newlab < corr.length) && (label(newlab) == null)) {
      newlab++;
    }
    hilitelabel(newlab);
  }

  private void labdown() {
    int newlab = currlab -1;
    while ((newlab > -1) && (label(newlab) == null)) {
      newlab--;
    }
    hilitelabel(newlab);
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("GO")) {
      setlabel();
    } else if (e.getActionCommand().equals("UP")) {
      labup();
    } else if (e.getActionCommand().equals("DOWN")) {
      labdown();
    } else if (e.getActionCommand().equals("CLOSE")) {
      dispose();
    }
  }

  private static int index(int i[], int key) {
    int j;
    for (j=0; j<i.length; j++) {
      if (key == i[j]) {
        return j;
      }
    }
    return -1;
  }

}
```

B.2 ScanError.java

```
package projects.javascheme.tools.scan;

//import projects.javascheme.tools.RGScheme.error.SchemeError;

class ScanError extends Error {

  public ScanError(String e) {
    super(e);
  }

}
```

B.3 syntax.java

```
package projects.javascheme.tools.scan;

import java.util.*;

class syntax {

  public static String keywords[] = {"and", "begin", "case", "cond", "define",
                                     "else", "if", "lambda", "let",
                                     "letrec", "or", "`"};
  public static String allkeys[] =  {"=>", "and", "begin", "case", "cond",
                                     "define", "delay", "do", "else", "if",
                                     "lambda", "let", "let*", "letrec", "or",
                                     "quasiquote", "quote", "set!", "unquote",
                                     "unquote-splicing"};

  ////////////////////////////////////////////////////////////////////////
  // Defined essential Scheme procedures : input types
  ////////////////////////////////////////////////////////////////////////

  private static String not_handled[] =
  {"set-car!", "set-cdr!", "assq", "assv", "assoc", "string-fill!",
   "vector-set!", "vector-fill!", "force", "make-string", "string->number",
  "make-vector", "list->string", "apply", "map", "for-each", "do",
   "=>", "delay", "quasiquote", "set!", "unquote", "unquote-splicing"};

  private static String one_any[] =
  {"not", "boolean?", "pair?", "null?", "list?", "symbol?", "number?",
   "complex?", "real?", "rational?", "integer?", "char?", "string?",
   "vector?", "procedure?", "quote"};
  private static String two_any[] =
  {"eqv?", "eq?", "equal?", "cons"};
  private static String many_any[] =
  {"list", "string", "vector", "let*"};

  private static String one_char[] =
  {"char->integer", "char-upcase", "char-downcase"};
  private static String two_char[] =
  {"char?", "char<?", "char>?", "char<=?", "char>=?", "char-ci<?", "char-ci>?",
   "char-ci<=?", "char-ci>=?"};

  private static String one_vector[] =
  {"vector-length", "vector->list"};

  public static String one_pair[] =
  {"car", "cdr", "caar", "cadr", "cdar", "cddr", "caaar", "caadr", "cadar",
   "caddr", "cdaar", "cdadr", "cddar", "cdddr", "caaaar", "caaadr", "caadar",
   "caaddr", "cadaar", "cadadr", "caddar", "cadddr", "cdaaar", "cdaadr",
   "cdadar", "cdaddr", "cddaar", "cddadr", "cdddar", "cddddr"};

  private static String one_number[] =
  {"exact?", "inexact?", "zero?", "positive?", "negative?", "odd?", "even?",
   "abs", "numerator", "denominator", "floor", "ceiling", "truncate", "round",
   "exp", "log", "sin", "cos", "tan", "asin", "acos", "atan", "sqrt",
   "real-part", "imag-part", "magnitude", "angle", "exact->inexact",
   "inexact->exact", "number->string"};
  private static String two_number[] =
  {"quotient", "remainder", "modulo", "rationalize", "atan", "expt",
   "make-rectangular", "make-polar"};
  private static String many_number[] =
  {"=", "<", ">", "<=", ">=", "max", "min", "+", "*", "-", "/", "gcd", "lcm"};

  private static String one_string[] =
  {"string->symbol", "string-length", "string->list", "string-copy"};
  private static String two_string[] =
  {"string?", "string-ci=?", "string<?", "string>?", "string<=?", "string<=?",
   "string-ci<?", "string-ci>?", "string-ci<=?", "string-ci<=?"};
  private static String many_string[] =
  {"string-append"};
```

```
private static String one_list[] =
{"length", "append", "reverse", "list->vector"};

private static String one_int[] =
{"integer->char"};

////////////////////////////////////////////////////////////////////////
// Defined essential Scheme procedures : return types
////////////////////////////////////////////////////////////////////////

public static String rt[][] =
{{"quote", "car", "cdr", "caar", "cadr", "cdar", "cddr", "caaar", "caadr",
  "cadar", "caddr", "cdaar", "cdadr", "cddar", "cdddr", "caaaar", "caaadr",
  "caadar", "caaddr", "cadaar", "cadadr", "caddar", "cadddr", "cdaaar",
  "cdaadr", "cdadar", "cdaddr", "cddaar", "cddadr", "cdddar", "cddddr"},
 // any
 {"exact?", "inexact?", "zero?", "positive?", "negative?", "odd?", "even?",
  "=", "<", ">", "<=", ">=", "not", "boolean?", "pair?", "null?", "list",
  "symbol?", "number?", "inexact->exact", "complex?", "real?", "rational?",
  "integer?", "char?", "string?", "vector?", "procedure?", "eqv?", "eq?",
  "equal?", "string?", "string-ci=?", "string<?", "string>?", "string<=?",
  "string<=?", "string-ci<?", "string-ci>?", "string-ci<=?", "string-ci<=?",
  "char?", "char<?", "char>?", "char<=?", "char>=?", "char-ci<?",
  "char-ci>?", "char-ci<=?", "char-ci>=?"}, // boolean
 {"string->symbol"}, // symbol
 {"char-upcase", "char-downcase", "integer->char"}, // char
 {"vector", "list->vector"}, // vector
 {"cons"}, // pair
 {"quotient", "remainder", "modulo", "rationalize", "atan", "expt", "abs",
  "numerator", "denominator", "floor", "ceiling", "truncate", "round", "exp",
  "log", "sin", "cos", "tan", "asin", "acos", "atan", "sqrt", "real-part",
  "imag-part", "magnitude", "angle", "exact->inexact", "make-rectangular",
  "make-polar", "max", "min", "+", "*", "-", "/", "gcd", "lcm"}, // number
 {"string", "number->string", "string-append", "string-copy"}, // string
 {}, // procedure
 {"append", "reverse", "list", "vector->list", "string->list"}, // list
 {"length", "vector-length", "char->integer","string-length"}, // integer
};

////////////////////////////////////////////////////////////////////////
// Simple static checking
////////////////////////////////////////////////////////////////////////
// Postcondition : returns true if code is correctly parenthesized scheme
// code
public static boolean checkParen(String code) {
  int parencount = 0;
  String token = Parser.nextToken(code);
  while (token != null) {
    code = code.substring(token.length());
    switch (token.charAt(0)) {
    case '(' : parencount++; break;
    case ')' : parencount--; break;
    }
    token = Parser.nextToken(code);
  }
  return (parencount == 0);
}

// Precondition : code is a Vector of single scheme values, conditioned
// Postcondition : returns an empty string if all applications of scheme
// procedures have the correct number of variables and keyword syntax is
// correct, or a warning message if any, or throws an error otherwise.
public static String checkKeys(Vector code) throws ScanError {
  String result = "";
  int i;
  for (i=0; i<code.size(); i++) {
    String expr = (String)code.elementAt(i);
    Vector v = Parser.split(expr);
    if (v.size() > 0) {
      String proc = (String)v.elementAt(0);
      if (inarray(not_handled, proc)) {
        result = result + "Warning : SCAN does not handle " + proc + "/n";
```

```
        } else {
          if (schemedef(proc)) {
            if (inarray(keywords, proc)) {
              if (proc.equals("case")) {
               if (v.size() < 2) {
                 throw new ScanError("No key in case statement : " + expr);
               }
               int j;
               for (j=2; j<v.size(); j++) {
                 String r = (String)v.elementAt(j);
                 if (Parser.atomp(r)) {
                   throw new
                     ScanError("No datum in case clause statement." + expr);
                 }
                 String data = (String)Parser.split(r).elementAt(0);
                 if (Parser.atomp(data)) {
                   if (!data.equals("else")) {
                     throw new
                      ScanError("No datum in case clause statement." + expr);
                   }
                 }
                 Vector datums = Parser.split(data);
                 int k;
                 for (k=0; k<datums.size(); k++) {
                   String q = (String)datums.elementAt(k);
                   if (!Parser.atom(q)) {
                     throw new ScanError("Invalid datum in clause : " + q);
                   }
                 }
               }
              } else if (proc.equals("cond")) {
               int j;
               for (j=1; j<v.size(); j++) {
                 String r = (String)v.elementAt(j);
                 if (Parser.atomp(r)) {
                   throw new ScanError("Improper clause in cond : " + r);
                 } else {
                   if ((((String)Parser.split(r).elementAt(0))
                      .equals("else")) && (j < v.size() - 1)) {
                     throw new
                      ScanError("Else clause not last in cond : " + expr);
                   }
                 }
               }
               Vector w = Parser.split((String)v.lastElement());
               if (((String)w.elementAt(0)).equals("else") &&
                   (w.size() < 2)) {
                 throw new ScanError("Else clause has no subexpressions.");
               }
              } else if (proc.equals("define")) {
               if (v.size() < 2) {
                 throw new ScanError("Improper definition : " + expr);
               }
               if (Parser.atom((String)v.elementAt(1))) {
                 if (v.size() != 3) {
                   throw new ScanError("" + v.size() + "Improper code definition
: " + expr);
                 }
               } else {
                 if (!lambda.isDef(expr)) {
                   throw new
                     ScanError("Improper lambda syntactic sugar : " + expr);
                 }
               }
              } else if (proc.equals("if")) {
               if (!((v.size() == 3) || (v.size() == 4))) {
                 throw new ScanError("Improper if definition : " + expr);
               }
              } else if (proc.equals("lambda")) {
               if (!lambda.isVal(expr)) {
                 throw new
                   ScanError("Poorly formatted lambda expression : " + expr);
```

```
            }
           } else if (proc.equals("let") || proc.equals("let*") ||
                      proc.equals("letrec")) {
            if (v.size() < 3) {
              throw new ScanError("Improper let definition : " + expr);
            } else {
              Vector l = new Vector();
              Vector w = Parser.split((String)v.elementAt(1));
              int j;
              for (j=0; j<w.size(); j++) {
                Vector x = Parser.split((String)w.elementAt(j));
                if (x.size() != 2) {
                  throw new ScanError("Improper let definition : " + expr);
                }
                String r = (String)x.elementAt(0);
                if (!isIdent(r)) {
                  throw new ScanError("Improper binding variable : " + r);
                }
                if (l.indexOf(r) > -1) {
                  throw new
                   ScanError("Repeated binding definition : " + r);
                }
                l.addElement(r);
              }
            }
           } else if (proc.equals("quote")) {
            if (v.size() != 1) {
              throw new ScanError("quote got too many parameters.");
            }
           }
          } else {
            int x = mu.vi(type(proc), 0);
            if ((x > -1) && (x != v.size() - 1)) {
             throw new
              ScanError(proc + " got " + (v.size()-1) + " params.");
            }
          }
        }
       }
       String test = checkKeys(mu.subVector(v, 1, v.size()));
       if (test != null) {
         result = result + test;
       }
     } else {
       if (inarray(allkeys, expr)) {
         throw new ScanError("Improper use of keyword " + expr + ".");
       }
     }
   }
   return result;
}

// Postcondition : if s is a scheme defined procedure, a Vector
// specifying input types is returned;
// first element : if > 0, no. of elements, else any number
// following elements : element types (see predicate), or any type
public static Vector type(String s) {
  Vector result = new Vector();
  if (inarray(one_any, s)) {
    result = nr(1, -1);
  } else if (inarray(two_any, s)) {
    result = nr(2, -1, -1);
  } else if (inarray(many_any, s)) {
    result = nr(-1, -1);
  } else if (inarray(one_char, s)) {
    result = nr(1, 4);
  } else if (inarray(two_char, s)) {
    result = nr(2, 4, 4);
  } else if (inarray(one_vector, s)) {
    result = nr(1, 5);
  } else if (inarray(one_pair, s)) {
    result = nr(1, 6);
```

```
    } else if (inarray(one_number, s)) {
      result = nr(1, 7);
    } else if (inarray(two_number, s)) {
      result = nr(2, 7, 7);
    } else if (inarray(many_number, s)) {
      result = nr(-1, 7);
    } else if (inarray(one_string, s)) {
      result = nr(1, 8);
    } else if (inarray(two_string, s)) {
      result = nr(1, 8, 8);
    } else if (inarray(many_string, s)) {
      result = nr(-1, 8);
    } else if (inarray(one_list, s)) {
      result = nr(1, 10);
    } else if (inarray(one_int, s)) {
      result = nr(1, 11);
    } else if (s.equals("list-tail") || s.equals("list-ref")) {
      result = nr(2, 10, 11);
    } else if (s.equals("memq") || s.equals("memv") || s.equals("member")) {
      result = nr(2, -1, 10);
    } else if (s.equals("symbol->string")) {
      result = nr(1, 3);
    } else if (s.equals("string-ref")) {
      result = nr(2, 8, 11);
    } else if (s.equals("string-set")) {
      result = nr(3, 8, 11, 4);
    } else if (s.equals("substring")) {
      result = nr(3, 8, 11, 11);
    } else if (s.equals("vector-ref")) {
      result = nr(2, 5, 11);
    }
    return result;
  }

  private static Vector nr(int a, int b) {
    Vector v = new Vector();
    v.addElement(new Integer(a));
    v.addElement(new Integer(b));
    return v;
  }
  private static Vector nr(int a, int b, int c) {
    Vector v = nr(a, b);
    v.addElement(new Integer(c));
    return v;
  }
  private static Vector nr(int a, int b, int c, int d) {
    Vector v = nr(a, b, c);
    v.addElement(new Integer(d));
    return v;
  }


  ////////////////////////////////////////////////////////////////////////
  // Restriction checking
  ////////////////////////////////////////////////////////////////////////
  // Precondition : c is of type 1 or 6
  // Postcondition : returns false only if val is determined to be an invalid
  // argpos'th argument for procname
  public static boolean proccheck(String procname, int argpos, constraint c) {
    if (inarray(keywords, procname)) {
      return true;
    } else if (inarray(not_handled, procname)) {
      return true;
    } else if (lambda.isVal(procname)) {
      return true;
    } else {
      Vector t = type(procname);
      if (t.size() > 0) {
        int v = mu.vi(t, 0);
        if ((v == -1) || (argpos <= v)) {
          if (v == -1) {
            v = mu.vi(t, 1);
```

```
        } else {
          v = mu.vi(t, argpos);
        }
        if (v == -1) {
          return true;
        }
        if (c.type == 1) {
          predicate p;
          switch (v) {
          case 10 : p = new predicate(6); break;
          case 11 : p = new predicate(7); break;
          default : p = new predicate(v);
          }
          return p.accepts(c.constant);
        } else { // Potential bug for overlapping types : pair,list; num,int
          int it = c.pred.type;
          return
            ((it == v) ||
             ((it == 6) && (v == 10)) || ((it == 10) && (v == 6)) ||
             ((it == 7) && (v == 11)) || ((it == 11) && (v == 7)));
        }
      }
    }
  }
  return false;
}

////////////////////////////////////////////////////////////////////////
// Used in generate.rules
////////////////////////////////////////////////////////////////////////
// Postcondition : if s is a scheme defined procedure,
// return value specifies type of return by procedure (see predicate),
// or -1 if indeterminate
public static int rettype(String s) {
  int i;
  for (i=1; i<rt.length; i++) {
    if (inarray(rt[i], s)) {
      return i;
    }
  }
  return -1;
}

////////////////////////////////////////////////////////////////////////
// Miscellaneous procedures
////////////////////////////////////////////////////////////////////////
// Postcondition : returns true only if teststr is in arr
public static boolean inarray(String arr[], String teststr) {
  int i;
  for (i=0; i<arr.length; i++) {
    if (teststr.equals(arr[i])) {
      return true;
    }
  }
  return false;
}
public static boolean known(String var) {
  return (schemedef(var) || (constant(var)));
}

public static boolean schemedef(String var) {
  return ((mu.arrayHas(not_handled, var)) || (mu.arrayHas(keywords, var)) ||
          (defined(var)));
}

public static boolean defined(String var) {
  return (inarray(not_handled, var) || inarray(one_any, var) ||
          inarray(two_any, var) || inarray(many_any, var) ||
          inarray(one_char, var) || inarray(two_char, var) ||
          inarray(one_vector, var) || inarray(one_pair, var) ||
          inarray(one_number, var) || inarray(two_number, var) ||
          inarray(many_number, var) || inarray(one_string, var) ||
```

```
          inarray(two_string, var) || inarray(many_string, var) ||
          inarray(one_list, var) || inarray(one_int, var));
}

public static boolean constant(String var) {
  int types[] = {1, 2, 3, 4, 8, 7};
  int i;
  for (i=0; i<types.length; i++) {
    predicate p = new predicate(types[i]);
    if (p.accepts(var)) {
     return true;
    }
  }
  return false;
}

////////////////////////////////////////////////////////////////////////
// checks if s is an identifier under Scheme lexical structure rules
////////////////////////////////////////////////////////////////////////

private static String pecid[] = {"+", "-", "..."};

public static boolean isIdent(String s) {
  if (s.length() < 1) {
    return false;
  } else if (mu.arrayHas(pecid, s)) {
    return true;
  } else if (!isInit(s.charAt(0))) {
    return false;
  } else {
    int i;
    for (i=1; i<s.length(); i++) {
     if (!isSub(s.charAt(i))) {
       return false;
     }
    }
    return true;
  }
}

private static boolean isInit(char c) {
  String vals = "abcdefghijklmnopqrstuvwxyz!$%&*/:<=>?^_~";
  return (vals.indexOf(Character.toLowerCase(c)) > -1);
}

private static boolean isSub(char c) {
  if (isInit(c)) {
    return true;
  } else {
    String vals = "0123456789+-.@";
    return (vals.indexOf(Character.toLowerCase(c)) > -1);
  }
}

////////////////////////////////////////////////////////////////////////
// checks if s represents a number under Scheme lexical structure rules
////////////////////////////////////////////////////////////////////////

public static boolean isNumber(String s) {
  return (isNumR(2, s) || isNumR(8, s) || isNumR(10, s) || isNumR(16, s));
}

private static boolean isNumR(int R, String s) {
  if (s.length() > 4) {
    if (isPrefixR(R, s.substring(0, 4)) &&
        isComplexR(R, s.substring(4))) {
     return true;
    }
  }
  if (s.length() > 2) {
    if (isPrefixR(R, s.substring(0, 2)) &&
        isComplexR(R, s.substring(2))) {
```

```
      return true;
    }
  }
  return ((R == 10) && isComplexR(R, s));
}

private static boolean isComplexR(int R, String s) {
  // <real R>
  if (isRealR(R, s)) {
    return true;
  }
  // <real R> @ <real R>
  int i;
  for (i=0; i<s.length(); i++) {
    if (s.charAt(i) == '@') {
      if (isRealR(R, s.substring(0, i)) && isRealR(R, s.substring(i+1))) {
        return true;
      }
    }
  }
  if (s.length() > 0) {
    if (s.charAt(s.length() - 1) == 'i') {
      // <real R> +/- <ureal R> i
      for (i=0; i<s.length()-1; i++) {
        if ((s.charAt(i) == '+') || (s.charAt(i) == '-')) {
          if (isRealR(R, s.substring(0, i)) &&
              isUrealR(R, s.substring(i+1, s.length()-1))) {
            return true;
          }
        }
      }
      // <real R> +/- i
      if (s.length() > 1) {
        if ((s.charAt(s.length() - 2) == '+') ||
            (s.charAt(s.length() - 2) == '-')) {
          if (isRealR(R, s.substring(0, s.length() - 2))) {
            return true;
          }
        }
      }
      if ((s.charAt(0) == '+') || (s.charAt(0) == '-')) {
        // +/- <ureal R> i
        if (isUrealR(R, s.substring(1, s.length() - 1))) {
          return true;
          }
        // +/- i
        if (s.length() == 2) {
          return true;
        }
      }
    }
  }
  return false;
}

private static boolean isRealR(int R, String s) {
  if (isUrealR(R, s)) {
    return true;
  }
  if (s.length() > 0) {
    if (isSign(s.substring(0,1))) {
      if (isUrealR(R, s.substring(1))) {
        return true;
      }
    }
  }
  return false;
}

private static boolean isUrealR(int R, String s) {
  if (isUintR(R, s)) {
    return true;
```

```
    }
    int i;
    for (i=0; i<s.length(); i++) {
      if (s.charAt(i) == '/') {
        if (isUintR(R, s.substring(0, i)) && isUintR(R, s.substring(i+1))) {
          return true;
        }
      }
    }
    return isDecimalR(R, s);
  }

  private static boolean isDecimalR(int R, String s) {
    if (R != 10) {
      return false;
    }
    int i, k;
    // <uinteger 10> <suffix>
    for (i=0; i<s.length(); i++) {
      if (isUintR(R, s.substring(0,i)) && isSuffix(s.substring(i))) {
        return true;
      }
    }
    if (s.length() > 1) {
      // . <digit 10>+ #* <suffix>
      if (s.charAt(0) == '.') {
        if (isDigitR(R, s.charAt(1))) {
          if (isSuffix(s.substring(forwchar(s, forwdig(10, s, 1), '#')))) {
            return true;
          }
        }
      }
      // <digit 10>+ . <digit 10>* #* <suffix>
      if (isDigitR(R, s.charAt(0))) {
        k = forwdig(10, s, 1);
        if (k < s.length()) {
          if (s.charAt(k) == '.') {
            if (isSuffix(s.substring(forwchar(s, forwdig(10, s, k+1), '#')))) {
              return true;
            }
          }
        }
      }
      // <digit 10>+ #+ . #* <suffix>
      if (isDigitR(R, s.charAt(0))) {
        k = forwdig(10, s, 1);
        if (k < s.length()) {
          if (s.charAt(k) == '#') {
            k = forwchar(s, k+1, '#');
          }
          if (k < s.length()) {
            if (s.charAt(k) == '.') {
              k = forwchar(s, k+1, '#');
              if (isSuffix(s.substring(k))) {
                return true;
              }
            }
          }
        }
      }
    }
    return false;
  }

  private static boolean isUintR(int R, String s) {
    if (s.length() > 0) {
      if (isDigitR(R, s.charAt(0))) {
        return (forwchar(s, forwdig(R, s, 1), '#') == s.length());
      }
    }
    return false;
  }
```

```
private static boolean isPrefixR(int R, String s) {
  switch (s.length()) {
  case 0 : return (R == 10);
  case 2 : return (isRadixR(R, s) || (isExactness(s) && (R == 10)));
  case 4 :
    return ((isRadixR(R, s.substring(0, 2)) &&
             (isExactness(s.substring(2, 4)))) ||
            (isExactness(s.substring(0, 2)) &&
             (isRadixR(R, s.substring(2, 4)))));
  default :
    return false;
  }
}

private static boolean isSuffix(String s) {
  if (s.equals("")) {
    return true;
  }
  if (isExponentMarker(s.charAt(0))) {
    if (s.length() > 1) {
      if (isDigitR(10, s.charAt(1))) {
        if (forwdig(10, s, 2) == s.length()) {
          return true;
        }
      }
      if (isSign(s.substring(1, 2))) {
        if (s.length() > 2) {
          if (isDigitR(10, s.charAt(2))) {
            if (forwdig(10, s, 3) == s.length()) {
             return true;
            }
          }
        }
      }
    }
  }
  return false;
}

private static boolean isExponentMarker(char c) {
  String vals = "esfdl";
  return (vals.indexOf(c) > -1);
}

private static boolean isSign(String s) {
  return (s.equals("+") || s.equals("-") || s.equals(""));
}

private static boolean isExactness(String s) {
  return (s.equals("#e") || s.equals("#i") || s.equals(""));
}

private static boolean isRadixR(int R, String s) {
  switch (R) {
  case 2 : return (s.equals("#b"));
  case 8 : return (s.equals("#o"));
  case 10 : return (s.equals("#d") || s.equals(""));
  case 16 : return (s.equals("#x"));
  default : return false;
  }
}

private static boolean isDigitR(int R, char c) {
  String vals;
  switch (R) {
  case 2 : vals = "01"; break;
  case 8 : vals = "01234567"; break;
  case 10 : vals = "0123456789"; break;
  case 16 : vals = "0123456789abcdef"; break;
  default : return false;
  }
```

```
    return (vals.indexOf(c) > -1);
  }

  private static int forwchar(String s, int i, char c) {
    int k = i;
    if (k < s.length()) {
      boolean flag = true;
      while (flag) {
        flag = false;
        if (s.charAt(k) == c) {
          k++;
          if (k < s.length()) { flag = true; }
        }
      }
    }
    return k;
  }

  private static int forwdig(int R, String s, int i) {
    int k = i;
    if (k < s.length()) {
      boolean flag = true;
      while (flag) {
        flag = false;
        if (isDigitR(R, s.charAt(k))) {
          k++;
          if (k < s.length()) { flag = true; }
        }
      }
    }
    return k;
  }

}
```

B.4 Parser.java

```
package projects.javascheme.tools.scan;

import java.util.*;
import projects.javascheme.tools.RGScheme.error.SchemeError;
import projects.javascheme.tools.RGScheme.lang.PrimProcPPrint;
import projects.javascheme.tools.RGScheme.data.SchemeObject;
import projects.javascheme.tools.RGScheme.eval.Environment;

// uses pattern, labelrestriction

class Parser {

  // Postcondition : returns the next token if possible, otherwise null;
  // tokens are comments, strings, characters, whitespace (contiguous),
  // parentheses (including "#("), ', variable names and constants
  // (booleans and numbers)
  public static String nextToken(String partial) {
    String delim = ";\"#\\\t\n\r ()\'";
    StringTokenizer st = new StringTokenizer(partial, delim, true);
    if (st.hasMoreTokens()) {
      String s = st.nextToken();
      if (s.equals(";")) {
        String t = "";
        while (!t.equals("\n")) {
          if (st.hasMoreTokens()) {
            t = st.nextToken();
            s = s + t;
          } else {
            return s;
          }
        }
      } else if (s.equals("\"")) {
        if (st.hasMoreTokens()) {
          String t = "";
          while (!t.equals("\"")) {
            if (st.hasMoreTokens()) {
              t = st.nextToken();
              s = s + t;
            } else {
              return null;
            }
          }
        } else {
          return null;
        }
      } else if (s.equals("#")) {
        if (partial.length() > 1) {
          switch (partial.toLowerCase().charAt(1)) {
          case '\\' :
            if (partial.toLowerCase().startsWith("#\\space")) {
              return partial.substring(0, 7);
            } else if (partial.toLowerCase().startsWith("#\\newline")) {
              return partial.substring(0, 9);
            } else {
              return partial.substring(0, 3);
            }
          case '(' : case 't' : case 'f' :
            return partial.substring(0, 2);
          case 'e' : case 'i' : case 'b' : case 'o' : case 'd' : case 'x' :
            String t = st.nextToken();
            if (t.length() == 1) {
              if (st.hasMoreTokens()) {
               return s + t;
              } else {
               return null;
              }
            } else {
              return null;
            }
```

```
        default : return null;
        }
      } else {
        return null;
      }
    } else if (cw(s, 0)) {
      if (st.hasMoreTokens()) {
        String t = st.nextToken();
        while (cw(t, 0)) {
          s = s + t;
          if (st.hasMoreTokens()) {
            t = st.nextToken();
          } else {
            return s;
          }
        }
      }
    }
    return s;
  }
  return null;
}

// Postcondition : returns the next scheme value in partial, ignoring
// leading whitespace, if possible, otherwise, null (includes comments
// nextval("(+ 4 5) (* 2 3)))") => "(+ 4 5)"
// nextval("(car ") => null
public static String nextval(String partial) {
  partial = rmws(partial);
  String s = nextToken(partial);
  if (s != null) {
    if (s.equals("(")) {
      int j = 1;
      partial = partial.substring(1);
      while (partial.length() > 0) {
        String t = nextToken(partial);
        partial = partial.substring(t.length());
        s = s + t;
        if (t.equals("(")) {
          j++;
        } else if (t.equals(")")) {
          j--;
        }
        if (j == 0) {
          return s;
        }
      }
      return null;
    } else if (s.equals(")")) {
      return null;
    } else if (s.equals("'")) {
      String t = nextval(partial.substring(1));
      if (t != null) {
        return "'" + t;
      } else {
        return null;
      }
    } else {
      return s;
    }
  } else {
    return null;
  }
}

// Precondition : code is syntatically correct code
// Postcondition : returns an array of integers indicating the starting
// character of each token
public static int[] token(String code) {
  Vector tokens = new Vector();
  String t = nextToken(code);
  int i = 0;
```

```
  while (t != null) {
    switch (t.charAt(0)) {
    case '\'' :
      tokens.addElement(new Integer(i));
      i++;
      code = code.substring(1);
      t = nextToken(code);
    case ';' : case '\t' : case '\n' : case '\r' : case ' ' : case ')' :
      break;
    case '"' : case '#' : case '(' :
    default :
      tokens.addElement(new Integer(i));
    }
    code = code.substring(t.length());
    i = i + t.length();
    t = nextToken(code);
  }
  int result[] = new int[tokens.size()];
  for (i=0; i<tokens.size(); i++) {
    result[i] = ((Integer)tokens.elementAt(i)).intValue();
  }
  return result;
}

// Postcondition : comments are removed
public static String uncomment(String pre) {
  String post = "";
  while (pre.length()> 0) {
    String s = nextToken(pre);
    pre = pre.substring(s.length());
    if (s.charAt(0) != ';') {
      post = post + s;
    }
  }
  return post;
}

// Precondition : pre is a syntactically valid piece of scheme code
// without comments
// Postcondition : whitespace in front and end are removed, all multiple
// whitespace is replaced by ' ', inserts ' ' before all '(' and after all
// ')' except no whitespace directly after '(' and ''' or before ')', all
// non-literal characters converted to lowercase
// (+( +  1 2)(+ 3 5  ) ) => (+ (+ 1 2) (+ 3 5))
public static String condition(String pre) {
  Vector tokens = new Vector();
  String s;
  // Tokenize pre, delete whitespace
  while (pre.length() > 0) {
    s = nextToken(pre);
    if (!cw(s, 0)) {
      tokens.addElement(s);
    }
    pre = pre.substring(s.length());
  }
  String post = "";
  int i;
  boolean addspace = false;
  for (i=0; i<tokens.size(); i++) {
    s = (String)tokens.elementAt(i);
    if (!s.equals(")")) {
      if (addspace) {
        post = post + " ";
      }
    }
    post = post + s;
    addspace = !((s.equals("(")) || (s.equals("'")));
  }
  return post;
}

// Precondition : s is a valid Scheme expression on which condition()
```

```
// has been called (or would return the same value had condition() been
// called on it)
// Postcondition :
// 1. converts lambda syntactic sugar
// 2. (quote #x) is converted to '#x
// 3. (list ...) and '( ... ) is expanded to (cons ...)
// 4. '(1 . 2) => (cons 1 2)
// 5. (caar #x) expands to (car (car x)) and so on
// 6. (let* ((a 1) (b 2) ...) (...)) to (let ((a 1)) (let ((b 2)) (...)))
// 7. #\ => #\space
// returns a Vector with 1st element a String containing the converted code,
// 2nd element a int[] of token starters
public static Vector transform(String s, int startloc) {
  String r, t;
  int i, j, k;
  Vector v, w, x, y, z;
  Vector result = new Vector();
  w = brsplit(s);
  if (w.size() > 1) {
    j = 0;
    y = new Vector();
    t = "";
    for (i=0; i<w.size(); i++) {
      r = (String)w.elementAt(i);
      x = transform(r, j);
      j = j + r.length() + 1;
      t = t + " " + (String)x.elementAt(0);
      y = mu.joinVectors(y, mu.i2v((int[])x.elementAt(1)));
    }
    result.addElement(t.substring(1));
    result.addElement(mu.v2i(y));
    return result;
  }
  int tokens[] = new int[0];
  int offset = 0;
  if (s.length() == 0) { // EMPTY STRING
    result.addElement("");
    result.addElement(tokens);
    return result;
  } else if (s.equals("#\\ ")) {
    result.addElement("#\\space");
    tokens = new int[1];
    tokens[0] = startloc;
    result.addElement(tokens);
    return result;
  } else if (s.charAt(0) == '\'') {
    if (atomp(s.substring(1))) { // 'ABC
      tokens = new int[1];
      tokens[0] = startloc;
      if (s.equals("'()")) { // '()
        result.addElement("nil");
      } else {
        result.addElement(s);
      }
      result.addElement(tokens);
      return result;
    } else { // '(A B C)
      s = "(quote " + s.substring(1) + ")";
      offset = -6;
    }
  }
  pattern p;

  v = split(s);
  String indicator = "";
  if (v.size() > 0) {
    indicator = (String)v.elementAt(0);
  } else { // A
    tokens = new int[1];
    tokens[0] = startloc;
    result.addElement(s);
    result.addElement(tokens);
```

```
  return result;
}

String u[] = new String[28];
System.arraycopy(syntax.one_pair, 2, u, 0, 28);

Vector token = new Vector();
Vector tokenv = new Vector();
Vector tokent = new Vector();

// token contains strings of tokens to be joined together to form the
// string result

// tokenv contains the offsets of the tokens within the original string
// except for -1 : not in original, -2, non applicable offset
// tokent contains the data type containing the offsets
// 1 : Integer, 2 : int[], 3 : Vector of tokenv and tokent

if (indicator.equals("define")) {
  if (lambda.isDef(s)) { // LAMBDA
    lambda l = new lambda(s, false);
    if (Parser.atom((String)v.elementAt(1))) {
      String q[] = {"(", "define", " ", l.name, " ", "(", "lambda", " ",
                    l.varstr};
      i = startloc + 8 + l.name.length();
      int temp[] = {startloc, startloc + 1, -2, startloc + 8, -2, i + 1,
                    i + 2, -2, i + 9};
      for (k=0; k<9; k++) {
        token.addElement(q[k]);
        tokenv.addElement(new Integer(temp[k]));
        tokent.addElement(new Integer(1));
      }
      i = i + l.varstr.length() + 10;
      for (k=0; k<l.body.size(); k++) {
        token.addElement(" ");
        tokenv.addElement(new Integer(-2));
        tokent.addElement(new Integer(1));
        String bexp = (String)l.body.elementAt(k);
        token.addElement(bexp);
        tokenv.addElement(new Integer(i));
        tokent.addElement(new Integer(1));
        i = i + bexp.length() + 1;
      }
      for (k=0; k<2; k++) {
        token.addElement(")");
        tokenv.addElement(new Integer(-2));
        tokent.addElement(new Integer(1));
      }
    } else { // syntactic sugar
      String q[] = {"(", "define", " ", l.name, " ", "(", "lambda", " "};
      i = startloc + 9;
      int temp[] = {startloc, startloc + 1, -2, i, -2, -1, -1, -2};
      for (k=0; k<8; k++) {
        token.addElement(q[k]);
        tokenv.addElement(new Integer(temp[k]));
        tokent.addElement(new Integer(1));
      }
      i = i + l.name.length() + 1;
      w = split(l.varstr);
      if (atom(l.varstr)) {
        token.addElement(l.varstr);
        tokenv.addElement(new Integer(i + 2));
        tokent.addElement(new Integer(1));
      } else {
        token.addElement("(");
        tokenv.addElement(new Integer(-1));
        tokent.addElement(new Integer(1));
        for (k=0; k<w.size(); k++) {
          r = (String)w.elementAt(k);
          token.addElement(r);
          tokenv.addElement(new Integer(i));
          tokent.addElement(new Integer(1));
```

```
            i = i + r.length() + 1;
            if (k < (w.size() - 1)) {
             token.addElement(" ");
             tokenv.addElement(new Integer(-2));
             tokent.addElement(new Integer(1));
            }
          }
          token.addElement(")");
          tokenv.addElement(new Integer(-2));
          tokent.addElement(new Integer(1));
        }
        i = startloc + ((String)v.elementAt(1)).length() + 9;
        for (k=0; k<l.body.size(); k++) {
          token.addElement(" ");
          tokenv.addElement(new Integer(-2));
          tokent.addElement(new Integer(1));
          String bexp = (String)l.body.elementAt(k);
          token.addElement(bexp);
          tokenv.addElement(new Integer(i));
          tokent.addElement(new Integer(1));
          i = i + bexp.length() + 1;
        }
        for (k=0; k<2; k++) {
          token.addElement(")");
          tokenv.addElement(new Integer(-2));
          tokent.addElement(new Integer(1));
        }
      }
    } else {
      token.addElement("(");
      tokenv.addElement(new Integer(0));
      tokent.addElement(new Integer(1));
      i = startloc + 1;
      for (j=0; j<v.size(); j++) {
        r = (String)v.elementAt(j);
        token.addElement(r);
        tokenv.addElement(new Integer(i));
        tokent.addElement(new Integer(1));
        if (j < v.size() - 1) {
          token.addElement(" ");
          tokenv.addElement(new Integer(-2));
          tokent.addElement(new Integer(1));
        }
        i = i + r.length() + 1;
      }
      token.addElement(")");
      tokenv.addElement(new Integer(-2));
      tokent.addElement(new Integer(1));
    }
  } else if (indicator.equals("quote")) {
    p = new pattern("(quote #x)");
    if (p.match(s)) {
      t = p.lastmatch("#x");
      if (atom(t)) { // (QUOTE ABC)
        tokens = new int[1];
        tokens[0] = startloc;
        result.addElement("'" + t);
        result.addElement(tokens);
        return result;
      } else { // (QUOTE (A B C)) and (QUOTE (A B . C))
        w = split(t);
        i = startloc + 8;
        boolean dotnotation = false;
        for (j=0; j<w.size(); j++) {
          r = (String)w.elementAt(j);
          if (r.equals(".")) {
            dotnotation = true;
            token.addElement(w.elementAt(j + 1));
            tokenv.addElement(new Integer(i + 2));
            tokent.addElement(new Integer(1));
            j = w.size();
          } else {
```

```
            String q[] = {"(", "cons", " ", r, " "};
            int temp[] = {-1, -1, -2, i, -2};
            for (k=0; k<5; k++) {
             token.addElement(q[k]);
             tokenv.addElement(new Integer(temp[k]));
             tokent.addElement(new Integer(1));
            }
          }
          i = i + r.length() + 1;
        }
        if (!dotnotation) {
          token.addElement("nil");
          tokenv.addElement(new Integer(-1));
          tokent.addElement(new Integer(1));
          i = 0;
        } else {
          i = 2;
        }
        for (j=i; j<w.size(); j++) {
          token.addElement(")");
          tokenv.addElement(new Integer(-2));
          tokent.addElement(new Integer(1));
        }
      }
    }
  } else if (indicator.equals("list")) { // (LIST A B C)
    w = split(s);
    w.removeElementAt(0);
    i = startloc + 6;
    for (j=0; j<w.size(); j++) {
      r = (String)w.elementAt(j);
      String q[] = {"(", "cons", " ", r, " "};
      int temp[] = {-1, -1, -2, i, -2};
      for (k=0; k<5; k++) {
        token.addElement(q[k]);
        tokenv.addElement(new Integer(temp[k]));
        tokent.addElement(new Integer(1));
      }
      i = i + r.length() + 1;
    }
    token.addElement("nil");
    tokenv.addElement(new Integer(-1));
    tokent.addElement(new Integer(1));
    for (j=0; j<w.size(); j++) {
      token.addElement(")");
      tokenv.addElement(new Integer(-2));
      tokent.addElement(new Integer(1));
    }
  } else if (indicator.equals("let*")) {
    String lets = (String)v.elementAt(1);
    w = split(lets);
    i = startloc + 6;
    for (j=0; j<w.size(); j++) {
      r = (String)w.elementAt(j);
      String q[] = {"(", "let", " ", "(", r, ")", " "};
      int temp[] = {-1, -1, -2, -1, i, -2, -2};
      for (k=0; k<7; k++) {
        token.addElement(q[k]);
        tokenv.addElement(new Integer(temp[k]));
        tokent.addElement(new Integer(1));
      }
      i = i + r.length() + 1;
    }
    Vector values = mu.subVector(v, 2, v.size());
    r = (String)values.elementAt(0);
    token.addElement(r);
    tokenv.addElement(new Integer(i+1));
    tokent.addElement(new Integer(1));
    i = i + r.length() + 2;
    for (j=1; j<values.size(); j++) {
      token.addElement(" ");
      tokenv.addElement(new Integer(-2));
```

```
    tokent.addElement(new Integer(1));
    r = (String)values.elementAt(j);
    token.addElement(r);
    token.addElement(new Integer(i+1));
    tokent.addElement(new Integer(1));
    i = i + r.length() +  1;
   }
   for (j=0; j<w.size(); j++) {
    token.addElement(")");
    tokenv.addElement(new Integer(-2));
    tokent.addElement(new Integer(1));
   }
  } else if (mu.arrayHas(u, indicator)) { // (C####R A)
   p = new pattern("(#proc #arg)");
   if (p.match(s)) {
    r = p.lastmatch("#proc");
    i = startloc + r.length();
    for (j=1; j<i-1; j++) {
      String q[] = {"(", "c" + r.charAt(j) + "r", " "};
      int temp[] = {-1, -1, -2};
      for (k=0; k<3; k++) {
        token.addElement(q[k]);
        tokenv.addElement(new Integer(temp[k]));
        tokent.addElement(new Integer(1));
      }
    }
    token.addElement(p.lastmatch("#arg"));
    tokenv.addElement(new Integer(i + 2));
    tokent.addElement(new Integer(1));
    for (j=i-2; j>0; j--) {
      token.addElement(")");
      tokenv.addElement(new Integer(-2));
      tokent.addElement(new Integer(1));
    }
   }
  } else { // all others
   w = split(s);
   token.addElement("(");
   tokenv.addElement(new Integer(0));
   tokent.addElement(new Integer(1));
   i = startloc + 1;
   for (j=0; j<w.size(); j++) {
    r = (String)w.elementAt(j);
    token.addElement(r);
    tokenv.addElement(new Integer(i));
    tokent.addElement(new Integer(1));
    if (j < w.size() - 1) {
      token.addElement(" ");
      tokenv.addElement(new Integer(-2));
      tokent.addElement(new Integer(1));
    }
    i = i + r.length() + 1;
   }
   token.addElement(")");
   tokenv.addElement(new Integer(-2));
   tokent.addElement(new Integer(1));
  }

  for (i=0; i<token.size(); i++) {
   if (mu.vi(tokenv, i) > -1) {
    x = transform((String)token.elementAt(i), mu.vi(tokenv, i));
    token.setElementAt(x.elementAt(0), i);
    tokenv.setElementAt((int[])x.elementAt(1), i);
    tokent.setElementAt(new Integer(2), i);
   }
  }

  tokenv.setElementAt(new Integer(0), 0);
  tokent.setElementAt(new Integer(1), 0);
  s = "";
  for (i=0; i<token.size(); i++) {
   s = s + (String)token.elementAt(i);
```

```
  }
  tokens = collapse(tokenv, tokent);
  v = new Vector();
  for (i=0; i<tokens.length; i++) {
    switch (tokens[i]) {
    case 0 :
      v.addElement(new Integer(startloc));
      break;
    case -1 :
      v.addElement(new Integer(-1));
    case -2 :
      break;
    default :
      v.addElement(new Integer(offset + tokens[i]));
    }
  }
  result.addElement(s);
  result.addElement(mu.v2i(v));
  return result;
}

// Used by transform to collapse the tree of tokens into a flat int[]
private static int[] collapse(Vector v, Vector vtype) {
  int i;
  Vector ints = new Vector();
  for (i=0; i<vtype.size(); i++) {
    switch (((Integer)vtype.elementAt(i)).intValue()) {
    case 1 : // Integer
      ints.addElement(v.elementAt(i));
      break;
    case 2 : // int[]
      int j;
      int temp[] = (int[])v.elementAt(i);
      for (j=0; j<temp.length; j++) {
        ints.addElement(new Integer(temp[j]));
      }
      break;
    case 3 : // Vector
      Vector w = (Vector)v.elementAt(i);
      v.setElementAt(collapse((Vector)w.elementAt(0),
                              (Vector)w.elementAt(1)),
                     i);
      vtype.setElementAt(new Integer(2), i);
      i--;
      break;
    }
  }
  return mu.v2i(ints);
}

// Precondition : expr is a valid Scheme expression on which condition()
// has been called (or would return the same value had condition() been
// called on it)
// Postcondition : returns a Vector containing the result of split(expr)
// then the set restrictions; except that (quote #x) is converted to '#x
// (list ...) is expanded to (cons ...)
// nextlayer("(+ (+ 5 e) (* a u))") => {"+", "(+ 5 e)", "(* a u)", r_type0,
// r_1, r_2}
// nextlayer("3") => {}
// nextlayer("(quote (list (quote a) (quote b)))") =>
// {"quote", "(cons 'a (cons 'b nil))"
public static Vector nextlayer(String expr, int startloc) {
  Vector result = split(expr);
  int i = result.size();
  if (i > 0) {
    int k = startloc + 1;
    int l = k + ((String)result.elementAt(0)).length();
    labelrestriction r = new labelrestriction(k, l);
    result.addElement(r);
    int j;
    for (j=1; j<i; j++) {
      k = l + 1;
```

```
      l = k + ((String)result.elementAt(j)).length();
      r = new labelrestriction(j, k, l);
      result.addElement(r);
    }
  }
  return result;
}

// Precondition : s is conditioned code
// Postcondition : asks the user for undefined values, returns the
// new string with the definitions, and the offset of the original code
public static Vector undef(String s, Environment env) {
  Vector result = new Vector();
  Vector v = brsplit(s);
  v = undef_helper(v, new Vector(), env);
  Vector found = (Vector)v.elementAt(0); // requested definitions
  Vector values = (Vector)v.elementAt(1);
  if (found.size() == 0) {
    result.addElement(s);
    result.addElement(new Integer(0));
    result.addElement(new Integer(0));
  } else {
    String r = "";
    int i;
    for (i=0; i<found.size(); i++) {
      r = r + "(define " + (String)found.elementAt(i) + " " +
        (String)values.elementAt(i) + ") ";
    }
    r = r + s;
    result.addElement(r);
    result.addElement(new Integer(r.length() - s.length()));

  }
  return result;
}

private static Vector undef_helper(Vector v, Vector known, Environment env) {
  Vector newknown = mu.joinVectors(new Vector(), known);
  Vector newfound = new Vector();
  Vector newvalues = new Vector();
  int i;
  for (i=0; i<v.size(); i++) {
    Vector u = undef_helper((String)v.elementAt(i),
                            mu.joinVectors(newknown, newfound), env);
    newfound = mu.joinVectors((Vector)u.elementAt(0), newfound);
    newvalues = mu.joinVectors((Vector)u.elementAt(1), newvalues);
    newknown = mu.joinVectors((Vector)u.elementAt(2), newknown);
  }
  Vector res = mu.nv(newfound);
  res.addElement(newvalues);
  res.addElement(newknown);
  return res;
}

// Precondition : s is conditioned code, v is a vector of scheme symbols
// that are known.
// Postcondition : returns a Vector of 3 Vectors; 1. newly defined symbols,
// 2. the new values, 3. symbols defined in s
private static Vector undef_helper(String s, Vector known, Environment env)
     throws ScanError{
  Vector res = new Vector();
  Vector newdefs = new Vector();
  Vector newvals = new Vector();
  Vector defs = new Vector();
  Vector w, u, t;
  int i;

  if (atomp(s)) {
    if ((mu.vectFind(known, s) == -1) && (!syntax.known(s))) {
      String r;
      try {
        r = PrimProcPPrint.eval(env.lookup(s));
```

```
    } catch (SchemeError e) {
      throw new ScanError(s + " " + e.getErrorMessage());
    }
    newdefs.addElement(s);
    newvals.addElement(r);
    if ((mu.vectFind(known, r) == -1) && (!syntax.known(r))) {
      u = undef_helper(r, mu.joinVectors(newdefs, known), env);
      newdefs = mu.joinVectors((Vector)u.elementAt(0), newdefs);
      newvals = mu.joinVectors((Vector)u.elementAt(1), newvals);
    }
  }
} else {
  w = split(s);
  String r = (String)w.elementAt(0);
  if (r.equals("define")) {
    if (lambda.isDef(s)) {
      lambda l = new lambda(s, false);
      defs.addElement(l.name);
      u = undef_helper(l.body,
                  mu.joinVectors(mu.joinVectors(known, defs),
                                 l.vars), env);
    } else {
      defs.addElement(w.elementAt(1));
      u = undef_helper((String)w.elementAt(2), known, env);
    }
    newdefs = (Vector)u.elementAt(0);
    newvals = (Vector)u.elementAt(1);
  } else if (r.equals("let")) {
    t = split((String)w.elementAt(1));
    u = new Vector();
    for (i=0; i<t.size(); i++) {
      Vector let = split((String)t.elementAt(i));
      u.insertElementAt(let.elementAt(0), 0);
      Vector x = undef_helper((String)let.elementAt(1), known, env);
      newdefs = mu.joinVectors((Vector)x.elementAt(0), newdefs);
      newvals = mu.joinVectors((Vector)x.elementAt(1), newvals);
    }
    u = undef_helper(mu.subVector(w, 2, w.size()),
                  mu.joinVectors(u, known), env);
    newdefs = (Vector)u.elementAt(0);
    newvals = (Vector)u.elementAt(1);
  } else if (r.equals("letrec")) {
    t = split((String)w.elementAt(1));
    u = new Vector();
    for (i=0; i<t.size(); i++) {
      Vector let = split((String)t.elementAt(i));
      u.insertElementAt(let.elementAt(0), 0);
    }
    u = mu.joinVectors(u, known);
    for (i=0; i<t.size(); i++) {
      Vector let = split((String)t.elementAt(i));
      Vector x = undef_helper((String)let.elementAt(1), u, env);
      newdefs = mu.joinVectors((Vector)x.elementAt(0), newdefs);
      newvals = mu.joinVectors((Vector)x.elementAt(1), newvals);
    }
    u = undef_helper(mu.subVector(w, 2, w.size()),
                  mu.joinVectors(u, known), env);
    newdefs = (Vector)u.elementAt(0);
    newvals = (Vector)u.elementAt(1);
  } else if (r.equals("lambda")) {
    lambda l = new lambda(s, true);
    u = undef_helper(l.body, mu.joinVectors(known, l.vars), env);
    newdefs = (Vector)u.elementAt(0);
    newvals = (Vector)u.elementAt(1);
  } else {
    u = undef_helper(w, known, env);
    newdefs = (Vector)u.elementAt(0);
    newvals = (Vector)u.elementAt(1);
  }
}
res.addElement(newdefs);
res.addElement(newvals);
```

```
  res.addElement(defs);
  return res;
}

////////////////////////////////////////////////////////////////////
// Parser utilities
////////////////////////////////////////////////////////////////////

// Postcondition : returns true only if c is a character that may be
// part of a variable name (any char except ' ', '(' and ')')
public static boolean vchar(char c) {
  return (!Character.isWhitespace(c) && (c != '(') && (c != ')'));
}

// Postcondition : returns true only if the ith character of s is
// whitespace
public static boolean cw(String s, int i) {
  if (s == null) {
    return false;
  } else if (i >= s.length()) {
    return false;
  } else {
    return Character.isWhitespace(s.charAt(i));
  }
}

// Precondition : expr is a proper expression with no leading whitespace
// Postcondition : returns true only if s is an atom (no more parens)
public static boolean atom(String s) {
  return ((split(s).size() == 0) && (!s.equals("()")));
}

// Precondition : expr is a proper expression with no leading whitespace
// Postcondition : returns true only if s is an atom or ()
public static boolean atomp(String s) {
  return (split(s).size() == 0);
}

// Precondition : s is a sequence of proper expressions, conditioned
// Postcondition : returns a Vector containing the expressions
public static Vector brsplit(String s) {
  return split("(" + s + ")");
}

// Precondition : expr is a proper expression, conditioned
// Postcondition : returns a Vector containing strings of the
// next layer of Scheme expressions within the first expression
// split("(+ (+ 5 e) (* a u))") => {"+", "(+ 5 e)", "(* a u)"}
// split("3") => {}
public static Vector split(String expr) {
  Vector result = new Vector();
  if (expr.length() > 0) {
    if (expr.charAt(0) == '(') {
      expr = expr.substring(1);
      String next = nextval(expr);
      while (next != null) {
        result.addElement(next);
        expr = rmws(expr);
        expr = expr.substring(next.length());
        next = nextval(expr);
      }
      if (!(rmws(expr)).equals(")")) {
        return new Vector();
      }
    }
  }
  return result;
}

// Postcondition : removes leading whitespace from a
private static String rmws(String a) {
  while (cw(a, 0)) {
```

```
      a = a.substring(1);
    }
    return a;
  }

}
```

B.5 generate.java

```
package projects.javascheme.tools.scan;

import java.util.*;

// uses mu, Parser, pattern, predicate, constraint, labelrestriction

class generate {

  // Precondition : s is uncommented, conditioned and transformed code
  // Postcondition : returns a vector of 2 elements, the first of which
  // is a vector containing the constraints generated from s, the second
  // containing the restrictions which describe each label
  public static Vector gencon (String s) {
    Vector vals = Parser.brsplit(s);
    Vector code = new Vector();
    Vector restrictions = new Vector();
    int i, j, k;
    j = 0;
    for (i=0; i<vals.size(); i++) {
      String val = (String)vals.elementAt(i);
      k = j + val.length();
      restrictions.addElement(new labelrestriction(j, k));
      Vector tree = generate.gentree(val, j);
      j = k + 1;
      k = tree.size();
      restrictions = mu.joinVectors(restrictions, mu.subVector(tree, k/2, k));
    }
    int labels[] = new int[s.length()];
    for (i=0; i<s.length(); i++) {
      labels[i] = -1;
    }
    for (i=0; i<restrictions.size(); i++) {
      labelrestriction r = (labelrestriction)restrictions.elementAt(i);
      labels[r.start] = i;
    }
    Vector lrs = new Vector();
    j = 0;
    for (i=0; i<vals.size(); i++) {
      lrs.addElement(new Integer(j));
      labelrestriction lrn =
        (labelrestriction)restrictions.elementAt(labels[j]);
      lrn.type = -1;
      restrictions.setElementAt(lrn, labels[j]);
      j = j + ((String)vals.elementAt(i)).length() + 1;
    }
    Vector constraints = rules(vals, lrs, restrictions, labels, new Vector(),
                               new Vector());
    Vector result = new Vector();
    result.addElement(constraints);
    result.addElement(restrictions);
    return result;
  }

  // Precondition : s is a valid, conditioned expression
  // Postcondition : returns a vector, the first half of which are all
  // the possible scheme expressions (labels) in the original expressions,
  // the second half of which are restrictions which describe each label
  public static Vector gentree(String s, int startloc) {
    Vector landr = Parser.nextlayer(s, startloc);
    int i = landr.size();
    int j = startloc + 1;
    Vector layer = mu.subVector(landr, 0, (i / 2));
    Vector restr = mu.subVector(landr, (i / 2), i);
    int x = layer.size();
    for (i=0; i<x; i++) {
      String exp = (String)layer.elementAt(i);
      Vector newV = gentree(exp, j);
      j = j + exp.length() + 1;
      int k = newV.size();
```

```
    layer = mu.joinVectors(layer, mu.subVector(newV, 0, (k / 2)));
    restr = mu.joinVectors(restr, mu.subVector(newV, (k / 2), k));
  }
  return mu.joinVectors(layer, restr);
}

// Applies the rules to exprs, returns the constraints
private static Vector rules(Vector exprs, Vector labres, Vector res,
                            int[] lr, Vector d, Vector dl) {
  return (Vector)rules(exprs, labres, res, lr, d, dl, true).elementAt(0);
}

// Applies the rules to exprs, returns constraints, definitions and
// definition labels
private static Vector rules(Vector exprs, Vector labres, Vector res,
                            int[] lr, Vector d, Vector dl,
                            boolean passdefs) {
  int i, j, k;
  Vector constraints = new Vector();
  Vector defs = mu.joinVectors(new Vector(), d);
  Vector deflabs = mu.joinVectors(new Vector(), dl);

  for (i=0; i<exprs.size(); i++) {
    String expr = (String)exprs.elementAt(i);
    int startloc = mu.vi(labres, i);
    String lab = ls(lr[startloc]);
    constraint c = new constraint();

    if (expr.equals("()")) {
     c.settype1("nil", lab);
     constraints.addElement(c);
    } else if (Parser.atom(expr)) {
      if (syntax.known(expr)) {
        c.settype1(expr, lab);
        constraints.addElement(c);
      } else {
        int varlab = mu.vectFind(defs, expr);
        if (varlab > -1) {
          int l = lr[mu.vi(deflabs, varlab)];
          c.settype2(ls(l), lab);
          constraints.addElement(c);
        }
      }
    } else {
      Vector v = Parser.split(expr);
      String indicator = (String)v.elementAt(0);
      if (indicator.equals("cons")) {
        Vector w = mu.nv(new Integer(startloc + 1));
        w.addElement(new Integer(startloc + 6));
        w.addElement(new Integer(startloc +
                                 ((String)v.elementAt(1)).length()  + 7));
        c.settype1("(cons " + ls(lr[startloc + 6]) + " " +
                   ls(lr[startloc + 7 + ((String)v.elementAt(1)).length()])
                   + ")", lab);
        constraints.addElement(c);
        constraints = mu.joinVectors(constraints,
                                     rules(v, w, res, lr, defs,
                                           deflabs));
      } else if (indicator.equals("car")) {
        Vector w = mu.nv(new Integer(startloc + 1));
        w.addElement(new Integer(startloc + 5));
        pattern p = new pattern("(cons #lx #ly)");
        constraint sub = new constraint();
        sub.settype2("#lx", lab);
        c.settype5(p, ls(lr[startloc + 5]), sub);
        constraints.addElement(c);
        constraints = mu.joinVectors(constraints,
                                     rules(v, w, res, lr, defs,
                                           deflabs));
      } else if (indicator.equals("cdr")) {
        Vector w = mu.nv(new Integer(startloc + 1));
        w.addElement(new Integer(startloc + 5));
```

```
    pattern p = new pattern("(cons #lx #ly)");
    constraint sub = new constraint();
    sub.settype2("#ly", lab);
    c.settype5(p, ls(lr[startloc + 5]), sub);
    constraints.addElement(c);
    constraints = mu.joinVectors(constraints,
                              rules(v, w, res, lr, defs,
                                   deflabs));
  } else if (indicator.equals("if")) {
    String r = (String)v.elementAt(1);
    int l1, l2loc, l3loc;
    k = startloc + 4;
    l1 = lr[k];
    k = k + r.length() + 1;
    l2loc = k;
    l3loc = -1;
    constraint sub = new constraint();
    sub.settype2(ls(lr[l2loc]), lab);
    c.settype7(ls(l1), sub);
    constraints.addElement(c);
    if (v.size() == 4) {
      c = new constraint();
      l3loc = k + ((String)v.elementAt(2)).length() + 1;
      sub = new constraint();
      sub.settype2(ls(lr[l3loc]), lab);
      c.settype5(new pattern("\\#f"), ls(l1), sub);
      constraints.addElement(c);
      c = new constraint();
      sub = new constraint();
      sub.settype2(ls(lr[l3loc]), lab);
      c.settype5(new pattern("nil"), ls(l1), sub);
      constraints.addElement(c);
    }
    Vector w = Parser.split(r);
    if (w.size() == 2) {
      String pred = (String)w.elementAt(0);
      String testval = (String)w.elementAt(1);
      if (predicate.known(pred) && Parser.atom(testval)) {
        // (pred? val)
        k = lr[startloc + 6 + pred.length()];
        constraints =
         mu.joinVectors(constraints,
                        if_helper((String)v.elementAt(2), l2loc, lr,
                                  testval, k, pred, true));
        if (v.size() == 4) {
         constraints =
           mu.joinVectors(constraints,
                          if_helper((String)v.elementAt(3), l3loc, lr,
                                    testval, k, pred, false));
        }
      }
      constraints = mu.joinVectors(constraints,
                                rules(mu.nv(r),
                                      mu.nv(new Integer(startloc+4)),
                                      res, lr, defs,
                                      deflabs));
      w = mu.subVector(v, 2, v.size());
      k = startloc + 5 + r.length();
      Vector u = mu.nv(new Integer(k));
      if (v.size() == 4) {
        u.addElement(new Integer(k + ((String)v.elementAt(2)).length() +
                                 1));
      }
      Vector newdefs = mu.joinVectors(defs, new Vector());
      Vector newdeflabs = mu.joinVectors(deflabs, new Vector());
      j = mu.vectFind(defs, testval);
      if (j > -1) {
        newdefs.removeElementAt(j);
        newdeflabs.removeElementAt(j);
      }
      constraints = mu.joinVectors(constraints,
                                rules(w, u, res, lr, newdefs,
```

```
                                    newdeflabs));
    } else {
      k = startloc + 1;
      w = mu.nv(new Integer(k));
      for (j=0; j<v.size() - 1; j++) {
        k = k + ((String)v.elementAt(j)).length() + 1;
        w.addElement(new Integer(k));
      }
      constraints = mu.joinVectors(constraints,
                              rules(v, w, res, lr, defs,
                                    deflabs));
    }
  } else if (indicator.equals("lambda")) {
    String r = (String)v.elementAt(1);
    Vector w = Parser.split(r);
    String varlabs;
    Vector e = new Vector();
    Vector f = new Vector();
    if (w.size() > 0) {
      String t = (String)w.elementAt(0);
      k = startloc + 9;
      e.addElement(t);
      f.addElement(new Integer(k));
      varlabs = "(" + ls(lr[k]);
      labelrestriction lrn = (labelrestriction)res.elementAt(lr[k]);
      lrn.type = -1;
      res.setElementAt(lrn, lr[k]);
      for (j=1; j<w.size(); j++) {
        k = k + t.length() + 1;
        lrn = (labelrestriction)res.elementAt(lr[k]);
        lrn.type = -1;
        res.setElementAt(lrn, lr[k]);
        t = (String)w.elementAt(j);
        if (!t.equals(".")) {
         e.addElement(t);
         f.addElement(new Integer(k));
         varlabs = varlabs + " " + ls(lr[k]);
        } else {
         varlabs = varlabs + " .";
        }
      }
      varlabs = varlabs + ")";
    } else {
      if (Parser.atom(r)) {
        varlabs = ls(lr[startloc + 1] + 1);
      } else {
        varlabs = "()";
      }
    }
    k = startloc + r.length() + 9;
    Vector a = new Vector();
    Vector b = new Vector();
    for (j=2; j<v.size(); j++) {
      r = (String)v.elementAt(j);
      a.addElement(r);
      b.addElement(new Integer(k));
      k = k + r.length() + 1;
    }
    c.settype1("(lambda " + varlabs + " " +
              ls(lr[startloc + expr.length() -
                  ((String)v.lastElement()).length() - 1]) + ")",
              lab);
    constraints.addElement(c);
    constraints = mu.joinVectors(constraints,
                            rules(a, b, res, lr,
                                  mu.joinVectors(defs, e),
                                  mu.joinVectors(deflabs, f)));
  } else if (indicator.equals("define")) {
    String var = (String)v.elementAt(1);
    int varlab = startloc + 8;
    int valloc = startloc + var.length() + 9;
    c.settype2(ls(lr[valloc]), ls(lr[varlab]));
```

```
    constraints.addElement(c);
    if (lambda.isDef(expr)) {
      defs.addElement(var);
      deflabs.addElement(new Integer(varlab));
      constraints = mu.joinVectors(constraints,
                              rules(mu.nv(v.elementAt(2)),
                                    mu.nv(new Integer(valloc)),
                                    res, lr, defs, deflabs));
    } else {
      constraints = mu.joinVectors(constraints,
                              rules(mu.nv(v.elementAt(2)),
                                    mu.nv(new Integer(valloc)),
                                    res, lr, defs, deflabs));
      defs.addElement(var);
      deflabs.addElement(new Integer(varlab));
    }
  } else if (indicator.equals("let")) {
    c.settype2(ls(lr[startloc + expr.length() -
                ((String)v.lastElement()).length() - 1]), lab);
    constraints.addElement(c);
    c = new constraint();
    String temp = (String)v.elementAt(1);
    Vector lets = Parser.split(temp);
    k = startloc + 7;
    Vector a = new Vector();
    Vector b = new Vector();
    for (j=0; j<lets.size(); j++) {
      String r = (String)lets.elementAt(j);
      Vector w = Parser.split(r);
      a.addElement(w.elementAt(0));
      b.addElement(new Integer(k));
      int l = k + 1 + ((String)w.elementAt(0)).length();
      labelrestriction lrn = (labelrestriction)res.elementAt(lr[k-1]);
      lrn.type = -1;
      res.setElementAt(lrn, lr[k-1]);
      lrn = (labelrestriction)res.elementAt(lr[k]);
      lrn.type = -1;
      res.setElementAt(lrn, lr[k]);
      lrn = (labelrestriction)res.elementAt(lr[l]);
      lrn.type = -1;
      res.setElementAt(lrn, lr[l]);
      c.settype2(ls(lr[l]), ls(lr[k]));
      k = k + r.length() + 1;
      constraints.addElement(c);
      c = new constraint();
      constraints = mu.joinVectors(constraints,
                              rules(mu.nv(w.elementAt(1)),
                                    mu.nv(new Integer(l)),
                                    res, lr, defs,
                                    deflabs));
    }
    Vector w = mu.subVector(v, 2, v.size());
    Vector u = new Vector();
    k = startloc + temp.length() + 6;
    for (j=0; j<w.size(); j++) {
      u.addElement(new Integer(k));
      k = k + ((String)w.elementAt(j)).length() + 1;
    }
    constraints = mu.joinVectors(constraints,
                             rules(w, u, res, lr,
                                 mu.joinVectors(defs, a),
                                 mu.joinVectors(deflabs, b)));
  } else if (indicator.equals("letrec")) {
    c.settype2(ls(lr[startloc + expr.length() -
                ((String)v.lastElement()).length() - 1]), lab);
    constraints.addElement(c);
    c = new constraint();
    String temp = (String)v.elementAt(1);
    Vector lets = Parser.split(temp);
    k = startloc + 10;
    Vector a = new Vector();
    Vector b = new Vector();
```

```
    Vector letvals = new Vector();
    Vector lvlocs = new Vector();
     for (j=0; j<lets.size(); j++) {
       String r = (String)lets.elementAt(j);
        Vector w = Parser.split(r);
        a.addElement(w.elementAt(0));
        b.addElement(new Integer(k));
       int l = k + 1 + ((String)w.elementAt(0)).length();
       labelrestriction lrn = (labelrestriction)res.elementAt(lr[k-1]);
       lrn.type = -1;
       res.setElementAt(lrn, lr[k-1]);
       lrn = (labelrestriction)res.elementAt(lr[k]);
       lrn.type = -1;
       res.setElementAt(lrn, lr[k]);
       lrn = (labelrestriction)res.elementAt(lr[l]);
       lrn.type = -1;
       res.setElementAt(lrn, lr[l]);
        c.settype2(ls(lr[l]), ls(lr[k]));
       k = k + r.length() + 1;
        constraints.addElement(c);
        c = new constraint();
       letvals.addElement(w.elementAt(1));
       lvlocs.addElement(new Integer(l));
     }
    defs = mu.joinVectors(defs, a);
    deflabs = mu.joinVectors(deflabs, b);
    constraints = mu.joinVectors(constraints,
                              rules(letvals, lvlocs, res, lr,
                                  defs, deflabs));
    Vector w = mu.subVector(v, 2, v.size());
    Vector u = new Vector();
    k = startloc + temp.length() + 9;
    for (j=0; j<w.size(); j++) {
      u.addElement(new Integer(k));
      k = k + ((String)w.elementAt(j)).length() + 1;
    }
    constraints = mu.joinVectors(constraints,
                              rules(w, u, res, lr, defs, deflabs));
  } else if (indicator.equals("and")) {
    Vector w = new Vector();
    k = startloc + 5;
    constraint sub;
    int cur = 0;
    for (j=1; j<v.size() - 1; j++) {
      sub = new constraint();
      cur = lr[startloc + 1] + j;
      sub.settype2(ls(cur), lab);
      c.settype5(new pattern("nil"), ls(cur), sub);
      constraints.addElement(c);
      c = new constraint();
      sub = new constraint();
      sub.settype2(ls(cur), lab);
      c.settype5(new pattern("\\#f"), ls(cur), sub);
      constraints.addElement(c);
      c = new constraint();
      w.addElement(new Integer(k));
      k = k + ((String)v.elementAt(j)).length() + 1;
    }
    w.addElement(new Integer(k));
    if (v.size() == 1) {
      c.settype1("#t", lab);
    } else {
      cur = lr[startloc + 1] + v.size() - 1;
      c.settype2(ls(cur), lab);
    }
    constraints.addElement(c);
    constraints = mu.joinVectors(constraints,
                              rules(mu.subVector(v, 1, v.size()),
                                  w, res, lr, defs, deflabs));
  } else if (indicator.equals("begin")) {
    Vector w = new Vector();
    k = startloc + 7;
```

```
  for (j=1; j<v.size(); j++) {
    w.addElement(new Integer(k));
    k = k + ((String)v.elementAt(j)).length() + 1;
  }
  c.settype2(ls(lr[mu.vi(w, w.size() - 1)]), lab);
  constraints.addElement(c);
  Vector x = rules(mu.subVector(v, 1, v.size()),
                w, res, lr, defs, deflabs, true);
  constraints = mu.joinVectors(constraints, (Vector)x.elementAt(0));
  defs = (Vector)x.elementAt(1);
  deflabs = (Vector)x.elementAt(2);
} else if (indicator.equals("case")) {
  constraints = mu.joinVectors
    (constraints,
     rules(mu.nv((String)v.elementAt(1)),
          mu.nv(new Integer(startloc + 6)), res, lr,
          defs, deflabs));
  String key = ls(lr[startloc + 6]);
  int cl = startloc + ((String)v.elementAt(1)).length() + 7;
  for (j=2; j<v.size(); j++) {
    labelrestriction lrn = (labelrestriction)res.elementAt(lr[cl]);
    lrn.type = -1;
    res.setElementAt(lrn, lr[cl]);
    String clause = (String)v.elementAt(j);
    Vector clexprs = Parser.split(clause);
    String data = (String)clexprs.elementAt(0);
    Vector u = new Vector();
    Vector x = new Vector();
    int cle = cl + data.length() + 2;
    for (k = 1; k<clexprs.size(); k++) {
      String r = (String)clexprs.elementAt(k);
      u.addElement(r);
      x.addElement(new Integer(cle));
      cle = cle + r.length() + 1;
    }
    if (data.equals("else")) {
      c.settype2(ls(lr[cl + 1] + clexprs.size() - 1), lab);
      constraints.addElement(c);
      c = new constraint();
      constraints = mu.joinVectors(constraints,
                              rules(u, x, res, lr, defs,
                                   deflabs));
    } else {
      Vector datums = Parser.split(data);
      Vector tempdefs = mu.joinVectors(defs, mu.nv(v.elementAt(1)));
      int datumloc = cl + 2;
      for (k=0; k<datums.size(); k++) {
       lrn = (labelrestriction)res.elementAt(lr[datumloc]);
       lrn.type = -1;
       res.setElementAt(lrn, lr[datumloc]);
       String datum = (String)datums.elementAt(k);
       c.settype1(datum, ls(lr[datumloc]));
       constraints.addElement(c);
       c = new constraint();
       constraint sub = new constraint();
       if (clexprs.size() == 1) {
         sub.settype1("#t", lab);
       } else {
         sub.settype2(ls(lr[cl + 1] + clexprs.size() - 1), lab);
       }
       c.settype5(new pattern(datum), key, sub);
       constraints.addElement(c);
       c = new constraint();
       constraints = mu.joinVectors
         (constraints,
          rules(u, x, res, lr, tempdefs,
               mu.joinVectors(deflabs,
                         mu.nv(new Integer(datumloc)))));
       datumloc = datumloc + datum.length() + 1;
      }
    }
    cl = cl + clause.length() + 1;
```

```
  }
} else if (indicator.equals("cond")) {
  int cl, e;
  cl = startloc + 6;
  for (j=1; j<v.size(); j++) {
    String clause = (String)v.elementAt(j);
    Vector w = Parser.split(clause);
    Vector u = new Vector();
    e = cl + 1;
    for (k=0; k<w.size(); k++) {
      labelrestriction lrn = (labelrestriction)res.elementAt(lr[e]);
      lrn.type = -1;
      res.setElementAt(lrn, lr[e]);
      u.addElement(new Integer(e));
      e = e + ((String)w.elementAt(k)).length() + 1;
    }
    Vector x = rules(w, u, res, lr, defs, deflabs, true);
    constraints = mu.joinVectors(constraints, (Vector)x.elementAt(0));
    defs = (Vector)x.elementAt(1);
    deflabs = (Vector)x.elementAt(2);
    constraint sub = new constraint();
    sub.settype2(ls(lr[cl + 1] + w.size() - 1), lab);
    c.settype7(ls(lr[cl + 1]), sub);
    constraints.addElement(c);
    c = new constraint();
    cl = cl + clause.length() + 1;
  }
} else if (indicator.equals("else")) {
  Vector w = new Vector();
  k = startloc + 7;
  for (j=1; j<v.size(); j++) {
    w.addElement(new Integer(k));
    k = k + ((String)v.elementAt(j)).length() + 1;
  }
  c.settype2(ls(lr[mu.vi(w, w.size() - 1)]), lab);
  constraints.addElement(c);
  Vector x = rules(v, w, res, lr, defs, deflabs, true);
  constraints = mu.joinVectors(constraints, (Vector)x.elementAt(0));
  defs = (Vector)x.elementAt(1);
  deflabs = (Vector)x.elementAt(2);
} else if (indicator.equals("or")) {
  Vector w = new Vector();
  k = startloc + 4;
  for (j=1; j<v.size(); j++) {
    constraint sub = new constraint();
    int cur = lr[startloc + 1] + j;
    sub.settype2(ls(cur), lab);
    c.settype7(ls(cur), sub);
    constraints.addElement(c);
    c = new constraint();
    w.addElement(new Integer(k));
    k = k + ((String)v.elementAt(j)).length() + 1;
  }
  constraints = mu.joinVectors(constraints,
                          rules(mu.subVector(v, 1, v.size()),
                              w, res, lr, defs, deflabs));
} else { // all other procedure applications
  constraint sub = new constraint();
  sub.settype2("#value", lab);
  c.settype5(new pattern("(lambda #vars #value)"),
            ls(lr[startloc + 1]), sub);
  constraints.addElement(c);
  String vstr = "(";
  if (v.size() > 1) {
    vstr = vstr + ls(lr[startloc + 1] + 1);
    for (j=2; j<v.size(); j++) {
      vstr = vstr + " " + ls(lr[startloc + 1] + j);
    }
  }
  vstr = vstr + ")";
  for (j=0; j<v.size() - 1; j++) {
    c = new constraint();
```

```
            c.settype8(vstr, ls(lr[startloc + 1]), j);
            constraints.addElement(c);
          }
          Vector w = new Vector();
          k = startloc + 1;

          for (j=0; j<v.size(); j++) {
            w.addElement(new Integer(k));
            k = k + ((String)v.elementAt(j)).length() + 1;
          }
          constraints = mu.joinVectors(constraints, rules(v, w, res,
                                                          lr, defs, deflabs));
        }
      }
    }
    Vector result = mu.nv(constraints);
    result.addElement(defs);
    result.addElement(deflabs);
    return result;
  }

  private static Vector if_helper(String code, int startloc, int lr[],
                                  String val, int vallab, String p,
                                  boolean andornot) {
    Vector result = new Vector();
    if (Parser.atom(code)) {
      if (code.equals(val)) {
        constraint c = new constraint();
        if (andornot) {
          c.settype3(ls(vallab), p, ls(lr[startloc]));
        } else {
          c.settype4(ls(vallab), p, ls(lr[startloc]));
        }
        result.addElement(c);
      }
    } else {
      Vector v = Parser.split(code);
      int i, j;
      j = startloc + 1;
      for (i=0; i<v.size(); i++) {
        String r = (String)v.elementAt(i);
        result = mu.joinVectors(result,
                                if_helper(r, j, lr, val, vallab, p, andornot));
        j = j + r.length() + 1;
      }
    }
    return result;
  }

  public static String ls(int i) {
    return "$" + i;
  }

}
```

B.6 dbg.java

```
package projects.javascheme.tools.scan;

import java.util.*;

public class dbg {
  // Debugger class

  // Constructor
  public dbg() {
  }
  // Precondition : vars is a Vector of variable names
  // Postcondition : returns a Vector of pairs of similar names, each pair
  // only listed once
  public static Vector closevars(Vector vars) {
    Vector similar = new Vector();
    int i, j, k, n;
    n = vars.size();
    for (i=0; i<n-1; i++) {
      for (j=i+1; j<=n-1; j++) {
        String a = (String)vars.elementAt(i);
        String b = (String)vars.elementAt(j);
        if (close(a, b) == true) {
          similar.addElement(a + " & " + b);
        }
      }
    }
    return similar;
  }

  // Precondition : returns true if a and b are dissimilar but close
  private static boolean close(String a, String b) {
    if (syntax.known(a) && syntax.known(b)) {
      return false;
    }
    if ((a.length() == 1) || (b.length() == 1)) {
      return false;
    }
    a = a.toLowerCase();
    b = b.toLowerCase();
    // Checks if two variable names are exactly the same (case-insensitive)
    if (a.equals(b)) {
      return false;
    } else {
      if (a.length() == b.length()) {
        int i, m, n;
        n = a.length();
        m = 0;
        Vector difs = new Vector();
        for (i=0; i<n; i++) {
          if (a.charAt(i) != b.charAt(i)) {
            m++;
            difs.addElement(new Integer(i));
          }
        }
        if (m > 2) {
          return false;
        } else if (m == 2) {
          i = mu.vi(difs, 0);
          if (mu.vi(difs, 1) == i + 1) {
            if ((a.charAt(i) == b.charAt(i+1)) &&
                (a.charAt(i+1) == b.charAt(i))) {
              return true;
            }
          }
        } else if (m == 1) {
          i = mu.vi(difs, 0);
          char c1, c2;
          c1 = a.charAt(i);
          c2 = b.charAt(i);
```

```
          if (isline(c1) && (isline(c2))) {
            return true;
          }
          if (iso(c1) && (iso(c2))) {
            return true;
          }
          if (isl(c1) && (isl(c2))) {
            return true;
          }
        }
      } else if (Math.abs(a.length() - b.length()) == 1){
        String na, nb;
        if (a.length() > b.length()) {
          na = b;
          nb = a;
        } else {
          na = a;
          nb = b;
        }
        int i;
        for (i=0; i<na.length(); i++) {
          if ((na.substring(0,i).equals(nb.substring(0,i))) &&
              (na.substring(i).equals(nb.substring(i+1)))) {
            return true;
          }
        }
      }
      return false;
    }
  }

  private static boolean isline(char a) {
    return ((a == '-') || (a == '_'));
  }

  private static boolean isl(char a) {
    return ((a == 'l') || (a == '1'));
  }

  private static boolean iso(char a) {
    return ((a == 'o') || (a == '0'));
  }

}
```

B.7 labelrestriction.java

```
package projects.javascheme.tools.scan;

import java.util.*;

// uses mu, pattern, constraint

class labelrestriction {

  // Used to determine type restriction on a label
  // type = 0 means no restriction
  // type > 0 means the label is the type'th argument

  int type;
  int start;
  int end;

  public labelrestriction(int a, int b) {
    type = 0;
    start = a;
    end = b;
  }

  public labelrestriction(int argloc, int a, int b) {
    type = argloc;
    start = a;
    end = b;
  }

  // Precondition : constrs is a Vector of constraints, restrs is a Vector
  // of labelrestrictions
  // Postcondition : returns a String describing any typechecking errors that
  // are found
  public static String setcheck(Vector constrs, Vector restrs, int tokens[]) {
    String result = "";
    int i, j, k;
    for (i=0; i<restrs.size(); i++) {
      if (tokens[i] > -1) {
        labelrestriction r = (labelrestriction)restrs.elementAt(i);
        String label = generate.ls(i);
        if (r.type == 0) { // First argument in an expression
          Vector v = findMembers(constrs, label);
          for (j=0; j<v.size(); j++) {
            String p = ((constraint)v.elementAt(j)).constant;
            if (lambda.isVal(p)) {
              int m = lambda.minVars(p);
              int n = lambda.maxVars(p);
              boolean flag = false;
              if (i + m >= restrs.size()) {
               flag = true;
              } else {
               for (k=1; k<=m; k++) {
                 labelrestriction vl = (labelrestriction)
                   restrs.elementAt(i+k);
                 if (vl.type != k) {
                   flag = true;
                 }
               }
               if ((n > -1) && (i + n + 1 < restrs.size())) {
                 if (((labelrestriction)restrs.elementAt(i+n+1)).type ==
                     n + 1) {
                   result = result + p + " at " + label +
                     " is applied to more than " + n + " elements.\n";
                 }
               }
              }
              if (flag) {
               result = result + "Application of " + p + " at " + label +
                 " does not have the minimum " + m + " elements.\n";
              }
```

```
            } else if (!syntax.schemedef(p)) {
              result = result + "Label " + label +
               " is an illegal application of " + p + "\n";
            }
          }
        } else if (r.type > 0) {
          Vector v = findMembers(constrs, label);
          // The procedure to which the current label is an argument is
          // of is the index of the label, i, less the argument position, type
          Vector procs = findMembers(constrs, generate.ls(i - r.type));
          for (j=0; j<v.size(); j++) {
            constraint c = (constraint)v.elementAt(j);
            for (k=0; k<procs.size(); k++) {
              String p = ((constraint)procs.elementAt(k)).constant;
              if (!syntax.proccheck(p, r.type, c)) {
               if (c.type == 1) {
                 result = result + "Label " + label + ", argument " +
                   Integer.toString(r.type) + " to " + p +
                   " may receive an illegal input : " + c.constant + "\n";
               } else {
                 result = result + "Label " + label + ", argument " +
                   Integer.toString(r.type) + " to " + p +
                   " may receive an illegal input of type " +
                   c.pred.pred.substring(0, c.pred.pred.length() - 1) +
                   " : " + c.constant + "\n";
               }
              }
            }
          }
        }
      }
    }
    return result;
  }

  // Precondition : v is a Vector
  private static String vecMems(Vector v) {
    String s = "";
    if (v.size() > 0) {
      s = ((constraint)v.elementAt(0)).constant;
      int i;
      for (i=1; i<v.size(); i++) {
        s = s + " " + ((constraint)v.elementAt(i)).constant;
      }
    }
    return s;
  }

  // Precondition : constrs is a Vector of constraints
  // Postcondition : Returns the constraints in constrs that indicate that
  // a value is a member of set l (type 1 or 6 only)
  private static Vector findMembers(Vector constrs, String l) {
    Vector v = new Vector();
    int i;
    for (i=0; i<constrs.size(); i++) {
      constraint c = (constraint)constrs.elementAt(i);
      if ((c.type == 1) || (c.type == 6)) {
       if (c.setm.equals(l)) {
         v.addElement(c);
       }
      }
    }
    return v;
  }

  public String toString() {
    return "(" + start + ", " + end + ")";
  }

}
```

B.8 constraint.java

```
package projects.javascheme.tools.scan;

import java.util.*;

// uses pattern, predicate

class constraint {

  // types
  // 1. constant is a member of setm
  // 2. set (name constant) is a subset of setm
  // 3. set and pred is a subset of setm
  // 4. set and !pred is a subset of setm
  // 5. given pat is a member of setm, subconstraint of type 1, 2
  // subconstraint contains only one variable from pat
  // 6. an unknown value of type accepted by pred is a member of setm
  // 7. given a non-nil value is a member of setm, subconstraint of type 1, 2
  // 8. constant is a list of arguments; if a lambda value is a member of setm,
  // and it takes an var'th argument, the appropriate argument from constant
  // will be assigned to the var'th argument set
  // 9. a scheme defined procedure of known type as applied to arguments
  // in constant is a member of setm (eg (+ $3 $4) is a member of $1)

  int type;
  String constant, setm;
  pattern pat;
  predicate pred;
  constraint subconstraint;
  int var;

  public constraint() {
  }

  public constraint(constraint c) {
    type = c.type;
    setm = c.setm;
    switch (type) {
    case 3 :
    case 4 :
    case 6 :
      pred = new predicate(c.pred.pred);
    case 1 :
    case 2 :
    case 8 :
    case 9 :
      constant = c.constant; break;
    case 5 :
      pat = new pattern(c.pat.initvalue);
    case 7 :
      subconstraint = new constraint(c.subconstraint);
      break;
    }
  }

  public void settype1(String c, String setname) {
    type = 1;
    constant = c;
    setm = setname;
  }

  public void settype2(String s, String setname) {
    type = 2;
    constant = s;
    setm = setname;
  }

  public void settype3(String s, String p, String setname) {
    type = 3;
    constant = s;
```

```
  pred = new predicate(p);
  setm = setname;
}

public void settype4(String s, String p, String setname) {
  type = 4;
  constant = s;
  pred = new predicate(p);
  setm = setname;
}

public void settype5(pattern p, String setname, constraint newconst) {
  type = 5;
  pat = new pattern(p.initvalue);
  setm = setname;
  subconstraint = new constraint(newconst);
}

public void settype6(String cst, String p, String s) {
  type = 6;
  constant = cst;
  pred = new predicate(p);
  setm = s;
}

public void settype7(String setname, constraint newconst) {
  type = 7;
  setm = setname;
  subconstraint = new constraint(newconst);
}

public void settype8(String s, String setname, int v) {
  type = 8;
  constant = s;
  setm = setname;
  var = v;
}

public void settype9(String c, String setname) {
  type = 9;
  constant = c;
  setm = setname;
}

// returns a new constraint of c1 and c2 imply something new
public static constraint use(constraint c1, constraint c2) {
  constraint t = new constraint();
  switch (c1.type) {
  case 1 :
    switch (c2.type) {
    case 2 :
      if (c1.setm.equals(c2.constant)) {
        t.settype1(c1.constant, c2.setm);
        return t;
      }
      break;
    case 3 :
      if ((c1.setm.equals(c2.constant)) && (c2.pred.accepts(c1.constant))) {
        t.settype1(c1.constant, c2.setm);
        return t;
      }
      break;
    case 4 :
      if ((c1.setm.equals(c2.constant)) && (c2.pred.rejects(c1.constant))) {
        t.settype1(c1.constant, c2.setm);
        return t;
      }
      break;
    case 5 :
      if (c1.setm.equals(c2.setm) && c2.pat.match(c1.constant)) {
        t = new constraint(c2.subconstraint);
        if (c2.pat.pv.contains(t.constant)) {
```

```
        String r = c2.pat.lastmatch(t.constant);
        if (!r.startsWith("$") && (t.type == 2)) {
          t = new constraint();
          t.settype1(r, c2.subconstraint.setm);
        } else {
          t.constant = r;
        }
      } else {
        t.setm = c2.pat.lastmatch(t.setm);
      }
      return t;
    }
    break;
  case 7 :
    if (c1.setm.equals(c2.setm)) {
      predicate p = new predicate("null?");
      if (p.rejects(c1.constant)) {
        return new constraint(c2.subconstraint);
      }
    }
    break;
  case 8 :
    if (c1.setm.equals(c2.setm)) {
      if (lambda.isValStr(c1.constant)) {
        lambda l = new lambda(c1.constant, true);
        if (l.matchVars(c2.constant)) {
          String mem = (String)l.matchvars.elementAt(c2.var);
          if (mem.startsWith("$")) {
           t.settype2(mem, (String)l.vars.elementAt(c2.var));
          } else {
           t.settype1(mem, (String)l.vars.elementAt(c2.var));
          }
          return t;
        }
      } else if (syntax.rettype(c1.constant) > -1) {
        t.settype9("(lambda () ("+c1.constant+" "+
                  c2.constant.substring(1)+")", c1.setm);
        return t;
      }
    }
    break;
  }
  break;
case 2 :
  switch (c2.type) {
  case 1 :
    if (c2.setm.equals(c1.constant)) {
      t.settype1(c2.constant, c1.setm);
      return t;
    }
    break;
  case 2 :
    if (c1.constant.equals(c2.setm)) {
      t.settype2(c2.constant, c1.setm);
      return t;
    } else {
      if (c2.constant.equals(c1.setm)) {
        t.settype2(c1.constant, c2.setm);
        return t;
      }
    }
    break;
  case 3 :
    if (c1.constant.equals(c2.setm)) {
      t.settype3(c2.constant, c2.pred.pred, c1.setm);
      return t;
    }
    break;
  case 4 :
    if (c1.constant.equals(c2.setm)) {
      t.settype4(c2.constant, c2.pred.pred, c1.setm);
      return t;
```

```
      }
      break;
    case 6 :
      if (c2.setm.equals(c1.constant)) {
        t.settype6(c2.constant, c2.pred.pred, c1.setm);
        return t;
      }
      break;
    }
    break;
  case 3 :
    switch (c2.type) {
    case 1 :
      if ((c2.setm.equals(c1.constant)) && (c1.pred.accepts(c2.constant))) {
        t.settype1(c2.constant, c1.setm);
        return t;
      }
      break;
    case 2 :
      if (c2.constant.equals(c1.setm)) {
        t.settype3(c1.constant, c1.pred.pred, c2.setm);
        return t;
      }
      break;
    case 6 :
      if ((c2.setm.equals(c1.constant)) &&
          (c1.pred.pred.equals(c2.pred.pred))) {
        t.settype6(c2.constant, c2.pred.pred, c1.setm);
        return t;
      }
      break;
    }
    break;
  case 4 :
    switch (c2.type) {
    case 1 :
      if ((c2.setm.equals(c1.constant)) && (c1.pred.rejects(c2.constant))) {
        t.settype1(c2.constant, c1.setm);
        return t;
      }
      break;
    case 2 :
      if (c2.constant.equals(c1.setm)) {
        t.settype4(c1.constant, c1.pred.pred, c2.setm);
        return t;
      }
      break;
    case 6 :
      if ((c2.setm.equals(c1.constant)) &&
          (!c1.pred.pred.equals(c2.pred.pred))){
        t.settype6(c2.constant, c2.pred.pred, c1.setm);
        return t;
      }
      break;
    }
    break;
  case 5 :
    switch (c2.type) {
    case 1 :
      if (c1.setm.equals(c2.setm) && c1.pat.match(c2.constant)) {
        t = new constraint(c1.subconstraint);
        if (c1.pat.pv.contains(t.constant)) {
          String r = c1.pat.lastmatch(t.constant);
          if (!r.startsWith("$") && (t.type == 2)) {
            t = new constraint();
            t.settype1(r, c1.subconstraint.setm);
          } else {
            t.constant = r;
          }
        } else {
          t.setm = c1.pat.lastmatch(t.setm);
        }
```

```
      return t;
    }
    break;
  case 6 :
    if (c1.pat.match(c2.constant) && c1.setm.equals(c2.setm)) {
      t = new constraint(c1.subconstraint);
      if (c1.pat.pv.contains(t.constant)) {
        String r = c1.pat.lastmatch(t.constant);
        if (!r.startsWith("$") && (t.type == 2)) {
          t = new constraint();
          t.settype1(r, c1.subconstraint.setm);
        } else {
          t.constant = r;
        }
      } else {
        if (c1.pat.pv.contains(t.setm)) {
          t.setm = c1.pat.lastmatch(t.setm);
        }
      }
      return t;
    }
    break;
  case 9 :
    if (c1.setm.equals(c2.setm) && c1.pat.match(c2.constant)) {
      String r = c1.pat.lastmatch(c1.subconstraint.constant);
      t.settype6(r, predicate.
                pstr[syntax.rettype((String)Parser.split(r)
                                .elementAt(0))],
                c1.subconstraint.setm);
      return t;
    }
    break;
  }
  break;
case 6 :
  switch (c2.type) {
  case 2 :
    if (c1.setm.equals(c2.constant)) {
      t.settype6(c1.constant, c1.pred.pred, c2.setm);
      return t;
    }
    break;
  case 3 :
    if ((c1.setm.equals(c2.constant)) &&
        (c2.pred.pred.equals(c1.pred.pred))) {
      t.settype6(c1.constant, c1.pred.pred, c2.setm);
      return t;
    }
    break;
  case 4 :
    if (c1.setm.equals(c2.constant)) {
      t.settype6(c1.constant, c1.pred.pred, c2.setm);
      return t;
    }
    break;
  case 5 :
    if (c2.pat.match(c1.constant) && c1.setm.equals(c2.setm)) {
      t = new constraint(c2.subconstraint);
      if (c2.pat.pv.contains(t.constant)) {
        String r = c2.pat.lastmatch(t.constant);
        if (!r.startsWith("$") && (t.type == 2)) {
          t = new constraint();
          t.settype1(r, c2.subconstraint.setm);
        }
      } else {
        if (c2.pat.pv.contains(t.setm)) {
          t.setm = c2.pat.lastmatch(t.setm);
        }
      }
      return t;
    }
    break;
```

```
    case 7 :
      if (c1.setm.equals(c2.setm)) {
        if (c1.pred.type != 1) {
          return new constraint(c2.subconstraint);
        }
      }
      break;
    }
    break;
  case 7 :
    switch (c2.type) {
    case 1 :
      if (c1.setm.equals(c2.setm)) {
        predicate p = new predicate("null?");
        if (p.rejects(c2.constant)) {
          return new constraint(c1.subconstraint);
        }
      }
      break;
    case 6 :
      if (c1.setm.equals(c2.setm)) {
        if (c2.pred.type != 1) {
          return new constraint(c1.subconstraint);
        }
      }
      break;
    }
    break;
  case 8 :
    if (c2.type == 1) {
      if (c2.setm.equals(c1.setm)) {
        if (lambda.isValStr(c2.constant)) {
          lambda l = new lambda(c2.constant, true);
          if (l.matchVars(c1.constant)) {
            String mem = (String)l.matchvars.elementAt(c1.var);
            if (mem.startsWith("$")) {
             t.settype2(mem, (String)l.vars.elementAt(c1.var));
            } else {
             t.settype1(mem, (String)l.vars.elementAt(c1.var));
            }
            return t;
          }
        } else if (syntax.rettype(c2.constant) > -1) {
          t.settype9("(lambda () ("+c2.constant+" "+
                    c1.constant.substring(1)+")", c2.setm);
          return t;
        }
      }
    }
    break;
  case 9 :
    if (c2.type == 5) {
      if (c1.setm.equals(c2.setm) && c2.pat.match(c1.constant)) {
        String r = c2.pat.lastmatch(c2.subconstraint.constant);
        t.settype6(r, predicate.
                  pstr[syntax.rettype((String)Parser.split(r)
                                      .elementAt(0))],
                  c2.subconstraint.setm);
        return t;
      }
    }
    break;
  }
  return null;
}

// uses the constraints in cs and applies use to all pairs, including any
// newly generated constraints, and returns all (old and new) constraints
public static Vector propagate(Vector cs) {
  Vector result = mu.subVector(cs, 0, cs.size());
  Vector csStrings = new Vector();
  int i, j, k, l;
```

```
    boolean changed = true;
    constraint t, a, b;
    l = 0;
    for (i=0; i<result.size(); i++) {
      csStrings.addElement(result.elementAt(i).toString());
    }
    while (changed) {
      changed = false;
      k = result.size();
      for (i=0; i<l; i++) {
        for (j=l; j<k; j++) {
          a = (constraint)result.elementAt(i);
          b = (constraint)result.elementAt(j);
          t = constraint.use(a,b);
          if (t != null) {
            if (!csStrings.contains(t.toString())) {
              csStrings.addElement(t.toString());
              result.addElement(t);
              changed = true;
            }
          }
        }
      }
      for (i=l; i<k-1; i++) {
        for (j=i+1; j<k; j++) {
          a = (constraint)result.elementAt(i);
          b = (constraint)result.elementAt(j);
          t = constraint.use(a,b);
          if (t != null) {
            if (!csStrings.contains(t.toString())) {
              csStrings.addElement(t.toString());
              result.addElement(t);
              changed = true;
            }
          }
        }
      }
      l = k;
    }
    return result;
  }

  public String toString() {
    switch (type) {
    case 1 : return constant + " is a member of S(" + setm + ")";
    case 2 : return "S(" + constant + ") is a subset of S(" + setm + ")";
    case 3 : return "S(" + constant + ") and " + pred.pred +
               " is a subset of S(" + setm + ")";
    case 4 : return "S(" + constant + ") and not " + pred.pred +
               " is a subset of S(" + setm + ")";
    case 5 : return "If " + pat.initvalue + " is a member of S(" + setm +
               "), " + subconstraint.toString();
    case 6 : return constant + " of type " +
               pred.pred.substring(0, pred.pred.length() - 1) +
               ", is a member of S(" + setm + ")";
    case 7 : return "If a non-nil value is a member of S(" + setm +
               "), " + subconstraint.toString();
    case 8 : return "If a lambda value is a member of " + setm + ", arg " +
               (var + 1) + " of " + constant + " is assigned.";
    case 9 : return constant + " is a member of S(" + setm + ")";
    default : return "";
    }
  }

}
```

B.9 predicate.java

```
package projects.javascheme.tools.scan;
// uses mu, pattern

class predicate {

  public static String pstr[] = {"null?", "boolean?", "symbol?", "char?",
                                 "vector?", "pair?", "number?", "string?",
                                 "procedure?", "list?", "integer?"};

  String pred;
  int type;
  // type 1 : null
  // type 2 : boolean
  // type 3 : symbol
  // type 4 : char
  // type 5 : vector
  // type 6 : pair
  // type 7 : number
  // type 8 : string
  // type 9 : procedure
  // type 10 : list
  // type 11 : integer
  // type 99 : other

  public predicate(String p) {
    pred = p;
    type = 99;
    int i;
    for (i=0; i<pstr.length; i++) {
      if (p.equals(pstr[i])) {
        type = i + 1;
      }
    }
  }

  public predicate(int i) {
    if ((i >= 1) && (i <= pstr.length)) {
      type = i;
    } else {
      type = 99;
    }
  }

  // Precondition : s is a valid scheme expression
  // Postcondition : returns true only if the original predicate applied
  // to s returns true, and type is not 99
  public boolean accepts(String s) {
    if (s.charAt(0) == '\'') {
      if ((type == 2) || (type == 4) || (type == 5) || (type == 7) ||
          (type == 8) || (type == 11)) {
        s = s.substring(1);
      }
    }
    pattern p;
    switch (type) {
    case 1 : return (s.equals("nil") || s.equals("()") || s.equals("'()"));
    case 2 : return (s.equals("#t") || s.equals("#f"));
    case 3 :
      if (s.charAt(0) == '\'') {
        return syntax.isIdent(s.substring(1));
      } else {
        return false;
      }
    case 4 :
      p = new pattern("(\\#\\\\#c)");
      if (p.match("(" + s + ")")) {
        String c = p.lastmatch("#c");
        return ((c.length() == 1) || (c.equals("space")) ||
                (c.equals("newline")));
```

```
      } else {
        return false;
      }
    case 5 : return ((s.startsWith("(vector ")) ||
                     ((s.startsWith("#(")) && (s.endsWith(")"))));
    case 6 : return ((s.startsWith("(cons ")) ||
                     ((s.startsWith("\'(")) && (s.endsWith(")"))));
    case 7 : return syntax.isNumber(s);
    case 8 : return ((s.charAt(0) == '"') && (s.endsWith("\"")));
    case 9 : return lambda.isVal(s);
    case 10 :
      p = new pattern("(cons #a #b)");
      if (p.match(s)) {
        String b = p.lastmatch("#b");
        return ((b.equals("nil")) || accepts(b));
      } else {
        return ((s.startsWith("\'(")) && (s.endsWith(")")));
      }
    case 11 :
      if (syntax.isNumber(s)) {
        int i = s.indexOf('.');
        if (i == -1) {
          return true;
        } else {
          String q = s.substring(i+1);
          while (q.startsWith("0")) {
            q = q.substring(1);
          }
          if (q.length() > 0) {
            String dig = "123456789";
            return (dig.indexOf(q.charAt(0)) == -1);
          } else {
            return true;
          }
        }
      } else {
        return false;
      }
    default : return false;
    }
  }

  // Precondition : s is a valid scheme expression
  // Postcondition : returns true only if the original predicate applied
  // to s returns false, and type is not 99
  public boolean rejects(String s) {
    if (type == 99) {
      return false;
    } else {
      return !accepts(s);
    }
  }

  // Postcondition : returns true only if s is a known predicate type
  public static boolean known(String s) {
    return mu.arrayHas(pstr, s);
  }

}
```

B.10 lambda.java

```
package projects.javascheme.tools.scan;

import java.util.*;

class lambda {

  String name;
  String varstr;
  Vector vars;
  String bodystr;
  Vector body;
  int type;
  // 1.
  // (define (a) 1)
  // (define a (lambda () 1))
  // 2.
  // (define (a . b) 1)
  // (define a (lambda b 1))
  // 3.
  // (define (a b c) 1)
  // (define a (lambda (b c) 1))
  // 4.
  // (define (a b c . d) 1)
  // (define a (lambda (b c . d) 1))
  Vector matchvars;

  // Precondition : if isVal, s must be a valid lambda value
  // if not isVal, s must be a valid lambda definition
  // <formals> = <varname>, <varlist>, (<varname> ... <varname> . <lastvar>)
  public lambda(String s, boolean isVal) {
    String c = Parser.condition(s);
    Vector u, v;
    int i;
    u = Parser.split(c);
    if (isVal) {  // (lambda <formals> <body>)
      name = "";
      varstr = (String)u.elementAt(1);
      if (Parser.atomp(varstr)) {
        if (varstr.equals("()")) {
          type = 1;
          vars = new Vector();
        } else {
          type = 2;
          vars = mu.nv(varstr);
        }
      } else {
        v = Parser.split(varstr);
        type = 3;
        vars = new Vector();
        for (i=0; i<v.size(); i++) {
          String var = (String)v.elementAt(i);
          if (var.equals(".")) {
            type = 4;
          } else {
            vars.addElement(var);
          }
        }
      }
      bodystr = c.substring(9 + varstr.length(), c.length() - 1);
      body = mu.subVector(u, 2, u.size());
    } else {
      String a = (String)u.elementAt(1);
      if (Parser.atom(a)) {  // (define <name> <lambdavalue>)
        name = a;
        lambda l = new lambda((String)u.elementAt(2), true);
        varstr = l.varstr;
        vars = l.vars;
        bodystr = l.bodystr;
        body = l.body;
```

```
      type = 1.type;
    } else {
      v = Parser.split(a);
      name = (String)v.elementAt(0);
      if (v.size() == 1) {
        type = 1;
        varstr = "()";
        vars = new Vector();
      } else {
        if (((String)v.elementAt(1)).equals(".")) {
          type = 2;
          varstr = (String)v.elementAt(2);
          vars = mu.nv(varstr);
        } else {
          varstr = "(" + a.substring(2 + name.length());
          vars = new Vector();
          type = 3;
          for (i=1; i<v.size(); i++) {
            String var = (String)v.elementAt(i);
            if (var.equals(".")) {
             type = 4;
            } else {
             vars.addElement(var);
            }
          }
        }
      }
      bodystr = c.substring(9 + a.length(), c.length() - 1);
      body = mu.subVector(u, 2, u.size());
    }
  }
}

// Precondition : s is a list; e.g. (), (1 2)
// Postcondition : matches the variables in the list s to the variables
// in vars and returns true if there is a valid match.
public boolean matchVars(String s) {
  Vector v = Parser.split(s);
  switch (type) {
  case 1 :
    matchvars = new Vector();
    return (v.size() == 0);
  case 2 :
    String r = "nil";
    int i;
    for (i=v.size() - 1; i>=0; i--) {
      r = "(cons " + (String)v.elementAt(i) + " " + r + ")";
    }
    matchvars = mu.nv(r);
    return true;
  case 3 :
    if (v.size() == vars.size()) {
      matchvars = v;
      return true;
    } else {
      return false;
    }
  case 4 :
    if (v.size() < vars.size() - 1) {
      return false;
    } else {
      matchvars = mu.subVector(v, 0, vars.size() - 1);
      r = "nil";
      for (i=v.size() - 1; i>=vars.size() - 1; i--) {
        r = "(cons " + (String)v.elementAt(i) + " " + r + ")";
      }
      matchvars.addElement(r);
      return true;
    }
  }
  return false;
}
```

```
// Postcondition : returns true only if s is a valid lambda value
public static boolean isVal(String s) {
 int i, j;
  Vector v = Parser.split(s);
  if (v.size() > 2) {
    if (((String)v.elementAt(0)).equals("lambda")) {
      String r = (String)v.elementAt(1);
      v = Parser.split(r);
      if (v.size() == 0) {
        return ((Parser.atom(r) && syntax.isIdent(r)) || (r.equals("()")));
      }
      i = v.indexOf(".");
      if (i > -1) {
        if ((i == 0) || (i != v.size() - 2)) {
          return false;
        }
        r = (String)v.elementAt(v.size() - 1);
        if (!(Parser.atom(r) && syntax.isIdent(r))) {
          return false;
        }
        for (j=0; j<i; j++) {
          r = (String)v.elementAt(j);
          if (!(Parser.atom(r) && syntax.isIdent(r))) {
            return false;
          }
        }
      } else {
        for (j=0; j<v.size(); j++) {
          r = (String)v.elementAt(j);
          if (!(Parser.atom(r) && syntax.isIdent(r))) {
            return false;
          }
        }
      }
      return true;
    }
  }
  return false;
}

// Postcondition : returns true only if s has the structure of a
// valid lambda value; differs from isVal in that the lambda arguments
// do not have to be valid identifiers
public static boolean isValStr(String s) {
 int i, j;
  Vector v = Parser.split(s);
  if (v.size() > 2) {
    if (((String)v.elementAt(0)).equals("lambda")) {
      String r = (String)v.elementAt(1);
      v = Parser.split(r);
      if (v.size() == 0) {
        return (Parser.atom(r) || (r.equals("()")));
      }
      i = v.indexOf(".");
      if (i > -1) {
        if ((i == 0) || (i != v.size() - 2)) {
          return false;
        }
        r = (String)v.elementAt(v.size() - 1);
        if (!Parser.atom(r)) {
          return false;
        }
        for (j=0; j<i; j++) {
          r = (String)v.elementAt(j);
          if (!Parser.atom(r)) {
            return false;
          }
        }
      } else {
        for (j=0; j<v.size(); j++) {
          r = (String)v.elementAt(j);
```

```
          if (!Parser.atom(r)) {
            return false;
          }
        }
      }
      return true;
    }
  }
  return false;
}

// Postcondition : returns true only if s is a valid lambda definition
public static boolean isDef(String s) {
  String c = Parser.condition(s);
  Vector elts = Parser.split(c);
  if (elts.size() < 3) {
    return false;
  }
  if (!((String)elts.elementAt(0)).equals("define")) {
    return false;
  }
  String name = (String)elts.elementAt(1);
  Vector body = mu.subVector(elts, 2, elts.size());
  if (Parser.atom(name)) {
    if (elts.size() == 3) {
      String val = (String)body.elementAt(0);
      return (lambda.isVal(val));
    }
  } else { // syntatic sugar
    Vector v = Parser.split(name);
    if (v.size() == 0) {
      return false;
    }
    int a = v.indexOf(".");
    if (a > -1) {
      if (a != v.size() - 2) {
        return false;
      }
      v.removeElementAt(a);
    }
    for (a=0; a<v.size(); a++) {
      String r = (String)v.elementAt(a);
      if (!(Parser.atom(r) && syntax.isIdent(r))) {
        return false;
      }
    }
    return true;
  }
  return false;
}

// Precondition : val is a valid lambda value
// Postcondition : returns the maximum number of variables to the lambda
// or -1 if there is no restricition
public static int maxVars(String val) {
  lambda l = new lambda(val, true);
  switch (l.type) {
  case 1 : case 3 :
    return l.vars.size();
  case 2 : case 4 :
    return -1;
  }
  return -1;
}

// Precondition : val is a valid lambda value
// Postcondition : returns the minimum number of variables to the lambda
// or -1 if there is no restricition
public static int minVars(String val) {
  lambda l = new lambda(val, true);
  switch (l.type) {
  case 1 : case 3 :
```

```
      return l.vars.size();
    case 2 :
      return -1;
    case 4 :
      return (l.vars.size() - 1);
    }
    return -1;
  }

}
```

B.11 pattern.java

```
package projects.javascheme.tools.scan;

import java.util.*;

// uses Parser

class pattern {

  String initvalue;
  Vector pv;
  Vector pm;
  boolean lambdamatch;
  boolean listmatch;
  int matchtype;

  // The pattern matches character for character until \ or # is encountered.
  // The character after \ is matched. # indicates that a variable is next.
  // The name of that variable is assumed to be the next character (NOT
  // whitespace or parenthesis) up till the next whitespace or parenthesis
  // character. This will match the next scheme value in the string to be
  // matched.

  // Precondition : p is a pattern that forms a proper expression;
  public pattern(String p) {
    String nexts = "\\";
    int i, j;
    initvalue = p;
    lambdamatch = false;
    listmatch = false;
    pv = new Vector();
    i = 0;
    for (j=0; j<p.length(); j++) {
      if (p.charAt(j) == '\\') {
        nexts = nexts + p.charAt(j+1);
        j++;
      } else {
        if (p.charAt(j) == '#') {
          pv.addElement(nexts);
          nexts = "\\";
          i = j;
          while (Parser.vchar(p.charAt(j+1))) {
            j++;
          }
          pv.addElement(p.substring(i, j+1));
        } else {
          nexts = nexts + p.charAt(j);
        }
      }
    }
    pv.addElement(nexts);
  }

  // postcondition : i = 1 sets pattern to match any lambda value;
  // i = 2  sets pattern to match any list
  public pattern(int i) {
    if (i == 1) {
      initvalue = "lambdavalue";
      lambdamatch = true;
      listmatch = false;
      pv = new Vector();
      pv.addElement("#vars");
      pv.addElement("#body");
    } else if (i == 2) {
      initvalue = "listvalue";
      lambdamatch = false;
      listmatch = true;
      pv = new Vector();
      pv.addElement("#list");
    }
```

```
}

// Precondition : c is a proper expression, conditioned
// Postcondition : true if c matches pattern p, false otherwise
public boolean match(String c) {
  int i, j, k;
  String current, temp;
  pm = new Vector();
  if (lambdamatch) { // match lambda value
    if (!lambda.isVal(c)) {
      return false;
    } else {
      lambda l = new lambda(c, true);
      pm.addElement(l.varstr);
      pm.addElement(l.bodystr);
    }
  } else if (listmatch) { // match list
    predicate pred = new predicate("list?");
    // tests for (cons 1 (cons 2 nil))
    // tests for `(1 2)
    if (pred.accepts(c)) {
      pattern p = new pattern("(cons #a #b)");
      if (p.match(c)) {
        matchtype = 0;
        pm.addElement(c);
      } else {
        matchtype = 1;
        Vector v = Parser.split(c.substring(1));
        String lst = "nil";
        pattern q = new pattern(2);
        for (j=v.size()-1; j>=0; j--) {
          String vs = (String)v.elementAt(j);
          if (q.match("`" + vs)) {
            vs = q.lastmatch("#list");
          }
          lst = "(cons " + vs + " " + lst + ")";
        }
        pm.addElement(lst);
      }
    } else {
      // need to test for (list 1 2)
      Vector v = Parser.split(c);
      if (v.size() > 1) {
        if (((String)v.elementAt(0)).equals("list")) {
          matchtype = 2;
          String lst = "nil";
          for (j=v.size()-1; j>0; j--) {
            lst = "(cons " + (String)v.elementAt(j) + " " + lst + ")";
          }
          pm.addElement(lst);
        } else {
          return false;
        }
      } else {
        return false;
      }
    }
  } else { // match pattern
    i = 0;
    for (j=0; j<pv.size(); j++) {
      current = (String)pv.elementAt(j);
      if (i < c.length()){
        k = i;
      } else {
        return false;
      }
      if (current.charAt(0) == '#') {
        temp = Parser.nextval(c.substring(i));
        if (temp == null) {
          return false;
        }
        i = i + temp.length();
```

```
        } else {
          i = current.length() + k - 1;
          if (c.length() < i) {
            return false;
          }
          temp = c.substring(k, i);
          if (!temp.equals(current.substring(1))) {
            return false;
          }
        }
        pm.addElement(temp);
      }
    }
    return true;
  }

  // Precondition : match() has been called successfully
  // Postcondition : if v is in p, the corresponding value from c is
  // returned, empty string otherwise
  public String lastmatch(String v) {
    if (pv.contains(v)) {
      return (String)pm.elementAt(pv.indexOf(v));
    } else {
      return "";
    }
  }

}
```

B.12 mu.java

```
package projects.javascheme.tools.scan;

import java.io.*;
import java.util.*;

class mu {

  // returns the elements from a to b in v
  public static Vector subVector(Vector v, int a, int b) {
    Vector result = new Vector();
    int i;
    for (i=a; i<b; i++) {
      result.addElement(v.elementAt(i));
    }
    return result;
  }

  // returns a vector that contains the elements of a and b
  public static Vector joinVectors(Vector a, Vector b) {
    Vector c = new Vector();
    int i;
    for (i=0; i<a.size(); i++) {
      c.addElement(a.elementAt(i));
    }
    for (i=0; i<b.size(); i++) {
      c.addElement(b.elementAt(i));
    }
    return c;
  }

  // Adds the object b to a
  public static void vectAdd(Vector a, Object b[]) {
    int i;
    for (i=0; i<b.length; i++) {
      a.addElement(b[i]);
    }
  }

  // Finds a similar (.toString() returns the same String) object in a Vector
  public static int vectFind(Vector v, Object o) {
    String r = o.toString();
    int j;
    for (j=v.size()-1; j>=0; j--) {
      if (r.equals(v.elementAt(j).toString())) {
        return j;
      }
    }
    return -1;
  }

  // returns true if v contains a copy of o
  public static boolean arrayHas(Object v[], Object o) {
    String r = o.toString();
    int j;
    for (j=0; j<v.length; j++) {
      if (r.equals(v[j].toString())) {
        return true;
      }
    }
    return false;
  }

  // converts a vector of int to int[]
  public static int[] v2i(Vector ints) {
    int i;
    int intarr[] = new int[ints.size()];
    for (i=0; i<ints.size(); i++) {
      intarr[i] = ((Integer)ints.elementAt(i)).intValue();
    }
```

```
    return intarr;
  }

  // converts an int[] to vector of int
  public static Vector i2v(int ints[]) {
    int i;
    Vector intvect = new Vector();
    for (i=0; i<ints.length; i++) {
      intvect.addElement(new Integer(ints[i]));
    }
    return intvect;
  }

  // returns the value of the Integer at element i of v
  public static int vi(Vector v, int i) {
    return ((Integer)v.elementAt(i)).intValue();
  }

  // Returns a new vector containing only o
  public static Vector nv(Object o) {
    Vector v = new Vector();
    v.addElement(o);
    return v;
  }

}
```