Parcel Programming Language Final Report

Min Chae, Mike Ilardi, Blake Shaw, Lawrence Wang, Tao Wang

May 3, 2004

Contents

1 Parcel White Paper **4**

1.1 Introduction 4

1.2 Parcel Applications 4

1.3 The Cel 5

1.4 Syntax, Ease of Use, and Readability 5

1.5 Meaningful Representations 6

2 Parcel Tutorial **7**

2.1 A Basic Example 7

2.2 Sample Programs 10

2.2.1 Recursive Drawing 10

2.2.2 A Bar Graph 11

2.2.3 Julia Set Fractals 12

3 Parcel Reference Manual **16**

3.1 Lexical conventions 16

3.1.1 Comments 16

3.1.2 Identifiers 16

3.1.3 Numbers 16

3.1.4 String literals 16

3.1.5 List literals 16

3.2 Types 17

3.2.1 Scalars and lists 17

3.2.2 Cel and color 17

3.3 Structure 18

3.3.1 Statements . . . 18
3.3.2 Identifiers, constants, and expressions . . . 18
3.3.3 Functions . . . 20
3.3.4 Control . . . 21
3.4 Internal functions . . . 22
3.4.1 Input and output . . . 22
3.4.2 Math . . . 23
3.4.3 Cel . . . 24

4 Project Plan **26**
4.1 Team Responsibilities . . . 26
4.2 Project Log and Timeline . . . 27
4.3 Programming Guidelines . . . 27
4.4 Development Process . . . 28

5 Architecture Design **29**
5.1 Design of the Interpreter . . . 29

6 Systems Integration **31**

7 Project Baselining **32**

8 Testing **33**
8.1 Test Plan . . . 33
8.2 Test Cases . . . 33
8.3 Process Used to Fix Bugs . . . 36
8.4 Known Issues . . . 36

9 Lessons Learned **38**
9.1 Group . . . 38
9.2 Blake Shaw . . . 38
9.3 Min Chae . . . 38
9.4 Mike Ilardi . . . 39
9.5 Lawrence Wang . . . 39

9.6 Tao Wang 39

10 Parcel Code **40**

10.1 Code Listing 40

10.2 Makefile 40

10.3 parcelGrammar.g 41

10.4 parcelWalker.g 46

10.5 Cel.java 50

10.6 PDT.java 56

10.7 PST.java 74

10.8 ParcelFunctionLibrary.java 78

10.9 ParcelInterpreter.java 83

10.10ParcelLibrary.java 89

10.11pcl.java 94

10.12Testing Code 95

Chapter 1

Parcel White Paper

1.1 Introduction

Parcel is a two-dimensional graphical processing language designed to simplify the task of producing meaningful representations of data models and simulations for statistical, scientific, and artistic applications. The fundamental object in Parcel is the cel. A cel is defined proportionally to other cels; it can be anything from a pixel to the whole screen depending on context. Furthermore, cels can be nested and tiled within each other. This fundamental data structure allows for unique high-level, abstract drawing and animation, independent of the underlying graphical implementation. Cels are described in more detail below.

Parcel was conceived of as an efficient means for parsing and visualizing data, and performing simulations in a cel-based system. The basic data types, constructs, and language structure work to make Parcel simultaneously easy to learn and powerful in function. Cel-based visualization techniques allow for unprecedented flexibility and ease of use in applications ranging from graphical analysis of data to image processing and computational art in two dimensions. Parcel's built-in iterative functionality allows for the consideration of time in computational art and analysis of data. To this end Parcel natively supports a variety of static image and video formats for output. Input is just as simple, as Parcel provides easy means of reading and performing calculations, both iterative and recursive, on a multitude of file formats.

1.2 Parcel Applications

Parcel is designed from the ground up for rapid development of a variety of scientific, statistical, and artistic applications. Any simulation that fundamentally relies on a discrete grid will benefit greatly from the simplified routines and structure of Parcel. Furthermore, Parcel allows for data to be quickly parsed and visualized from a variety of different sources. And finally, Parcel is capable of creating detailed compelling images with only a few lines of code, a boon for computational artists.

Simulation is becoming a fundamental aspect of science because it allows scientists to see phenomena not testable by conventional experiments or predictable by pure theory. Computational scientists need simple tools that will allow them to visualize these phenomena. The goal of Parcel is to reduce the complexity of setting up a simulation. The underlying grid structure of cels allow for simulations to be written quickly and efficiently. Simply specify an equation governing a region of cels, iterate, and output to the screen or a file. For example, it's very simple to implement variations on Conway's Game of Life. All you need to do is define the cases for when a cel lives or dies, and then iterate. Parcel handles all of the setup for the simulation, allowing the user to focus on what's important, the rules and equations governing the simulation.

Parcel's parsing routines allow the user to quickly get a visual representation from a set of data. Parcel accepts input from image and text files, and is based on the cross-platform standard Java, integrating nicely into a Unix-based environment. Parcel's graphical engine allows beautiful color images to be created with minimal code. Furthermore, Parcel was designed to allow scalability, so an artist working on a piece can output a small preview to the screen and then render and output a large high-resolution version to file. For example, the cel-based structure of Parcel is perfect for generating fractals and other iterative computational art such as plotting the Lorenz Attractor.

1.3 The Cel

One of the major features of Parcel is its fundamental graphical object, the cel. Graphical constructions made from cels are defined relationally: cels' positions, proportions, and sizes are defined entirely relative to other cels. A cel's height-to-width ratio is also free to change. This relational definition of cels creates a cel hierarchy; each cel has a 'parent' relative to which it is defined. If a cel's relational properties are changed, the changes propagate down the tree to its descendants, making it easy to manipulate subgroups in a graphical construction by manipulating their parent cels.

The relational definition of cels makes resulting constructions screen-independent. The independence of cels from screen space allows the user to construct as many separate groups of cels as is necessary, like separate canvases, and then link them together as desired. For example, this makes it nearly as simple to display a grid of 100 copies of an image permuted over some attribute, as it would be to display a single instance of that image; all image resizing is completely automatic. The mapping of cel-image to screen space is deferred until the last possible moment, right before drawing, and handled by a display function that abstracts away the drawing details, requiring only a scaling factor.

1.4 Syntax, Ease of Use, and Readability

Parcel is a little language focused around rapid development of visualizations of complex data and simulations. Its syntax reflects this design goal: easy to learn, it provides developers with great flexibility to produce meaningful output, while abstracting commonly-used low-level functions. Parcel, like AppleScript, avoids cryptic symbols in favor of words that make their function clear. Users without any knowledge of traditional programming languages like C or Java can quickly learn to use Parcel to generate a wide variety of images and simulations.

With Parcel, developers can spend their time working out the details of the images they want to create, instead of wasting time re-implementing basic methods for handling input and output. With simple, easily-readable lines of code, developers can perform many complex functions such as grouping and ungrouping cels and manipulating complex groups of cels based on their attributes. A few simple lines of Parcel can create complex images from data or simulations in a fraction of the lines needed by a traditional programming language. Parcel is good news for Java programmers who do not have experience programming graphics, but want to continue to use the Java platform, since it can create Java applets and executables.

1.5 Meaningful Representations

It can be fairly simple, once a programmer knows how to manipulate graphics on screen, to translate data in text files into some sort of image. The ultra-naive approach is that the text itself can be thought of as a "graphical" representation of its meaning. The purpose of Parcel, however, is to make it easy to create meaningful images. By meaningful, we refer to representations that clearly show the underlying pattern of data, which make it simple to reveal hidden patterns, and to allow a user to discern the signal from the noise.

Imagine how difficult it would be to comprehend the mysteriousness of fractals without ever seeing an image of the Mandelbrot set or to see the complexity of cellular automata without seeing the sequential generation of each of its states. There exist a number of systems – physical, biological, chemical, mathematical – that are extremely hard, if not impossible, to analyze without a substantial visual representation. Parcel makes designing such representations simple.

Chapter 2

Parcel Tutorial

2.1 A Basic Example

```
set x to 5
set c to [blue, yellow, red, Color[255, 34, 255]]
```

This creates two variables, `x` and `c`. `x` is set to an integer value; `c` is set to a list of colors. Blue, yellow and red, along with a number of other colors, are built into the language as keywords. The last element in the list is a call to the built-in Color function that creates a user-defined color; the three integer parameters represent RGB values.

```
print x
print c
```

The print command prints a given object to the console.

```
set root to new cel
```

This is a special case of the set command that creates a new cel. All variables need to be created with set before they are used.

```
set color of root to green
```

This sets the color of the `root` cell to the built-in color green.

```
draw root to screen as [200, 200]
draw root to "example1.png" as [200, 200]
```

Cels have no intrinsic proportion or size; only when the `draw` command is executed are cels mapped to pixels for display. Cels can be drawn to the screen or to an image file. Figure 2.1 shows what the cel looks like.

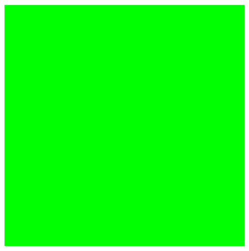

Figure 2.1: A green cel

```
divide root by x
set q to child [2,2] of root
set color of q to red
```

`divide` breaks up a cel into a grid of smaller "child" cels. These smaller cels can be accessed with the `child` keyword. Each child initially inherits its color from its parent. There are many options for determining the size and position of children relative to their parent. This example illustrates the simplest use of `divide`, which creates an `x` by `x` grid of equally sized cels.

In the next two lines, set is used to "name" a particular child of `root` as `q`. `q` is then colored white.

Figure 2.2 shows what the cel looks like if we drew it with the same proportions as above.

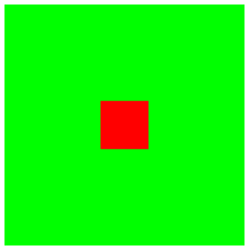

Figure 2.2: A green cel divided

```
foreach aCel in root:
    set color of aCel to randomItem[c].
```

Here we introduce some simple iteration. The `foreach` command iterates over every item in a list or cel. As `root` is a cel, `foreach` iterates over all of `root`'s 25 children. In each iteration one of the children is assigned to aCel, and given a color from the list `c` that we initialized earlier. The color is picked using the `randomItem` function, which returns a random item from a list.

Figure 2.3 shows what the cel looks like if we draw it with the same proportions as above.

```
define pickRandomColor[]:
    return Color[255*random[], 255*random[], 255*random[]].
```

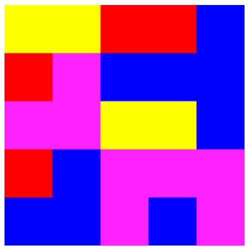

Figure 2.3: A cel divided and then randomly colored

Here we demonstrate function definition. The keyword `define` begins the definition, followed by the function name `pickRandomColor` and a colon. The function body, between the colon and the period that closes the block, is defined as returning a color with three random RGB values. The built-in function random returns a value between 0 and 1.

The function is then called like any other, by its name followed by its parameters in brackets. If there are no parameters, a set of empty brackets must follow the function identifier.

```
set distance to 0
set wonRace to 0
while wonRace is 0:
    print distance
        if random[] > 0.9:
            set distance to distance + 2
        . else: if random[] > 0.5:
            set distance to distance + 1..
        if distance > 10:
            set wonRace to 1..
print "hurray you won"
```

Here we demonstrate basic control flow. As in most programming languages, `while` repeats the contents of its body until its condition is false, and `if` executes its contents if its condition is true.

Note that there is no special ”else if” construction, so two periods are necessary to close both the `else` and the `if` blocks.

```
define fact[x]:
    if x is 1:
        set rs to 1
    . else:
        set rs to (x * fact[x-1]).
    return rs.
```

This example demonstrates a recursive function.

```
set pic to child [0, 0] of root
```

```
load image "lena.png" into pic

set list to []
load data "numbers.txt" into list
```

Parcel can load two types of data: images (GIF, PNG, or JPG, automatically detected), and numbers from a text file, delimited by newlines and tabs. Images are loaded directly into existing cels, creating a child cel for each pixel in the image, and lists of numbers are loaded into an existing list. For text files, each line is loaded as a new list item. Multiple numbers on the same line produce a sublist, and lone numbers produce a single value.

Figure 2.4 shows what the cel looks like if we draw it with the same proportions as above.

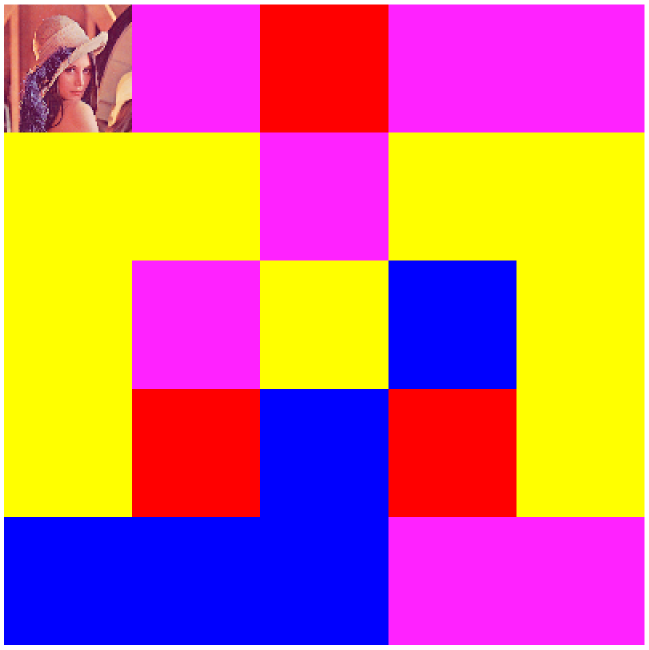

Figure 2.4: Loading an image into a cel

2.2 Sample Programs

2.2.1 Recursive Drawing

```
set root to new cel
set color of root to white
divide root by [[1,10,1],[1,10,1]]
foreach c in root:
divide c by [[1,10,1],[1,10,1]]
set color of child [1,1] of c to red.
set color of child [1,1] of root to blue
set child [1,1] of child [1,1] of root to root
```

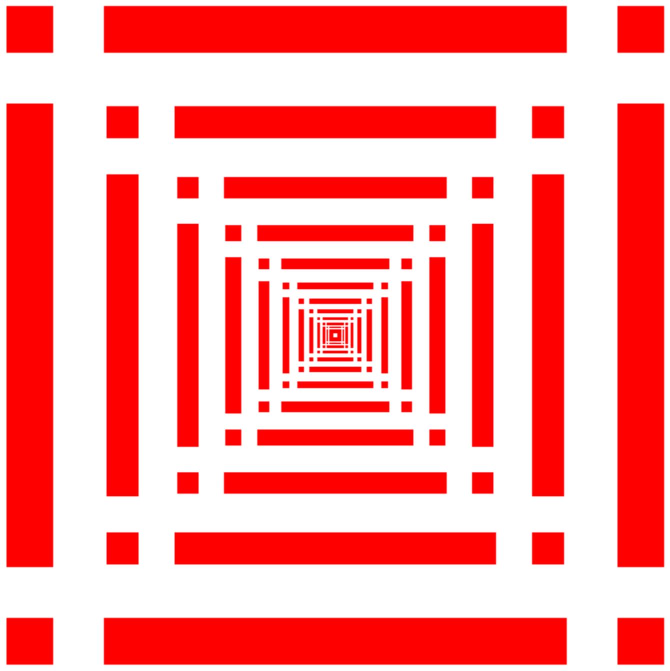

Figure 2.5: A Simple Recursive Drawing

```
draw root to "selfparent.png" as [500,500]
```

2.2.2 A Bar Graph

```
set numbers to []
load data "data.txt" into numbers

set max to 0
foreach num in numbers:
    print num
    if num > max:
        set max to num..

print max

set x to new cel
set color of x to yellow
set sz to size of numbers

print sz

divide x into rows by sz
for i from 0 to sz-1:
```

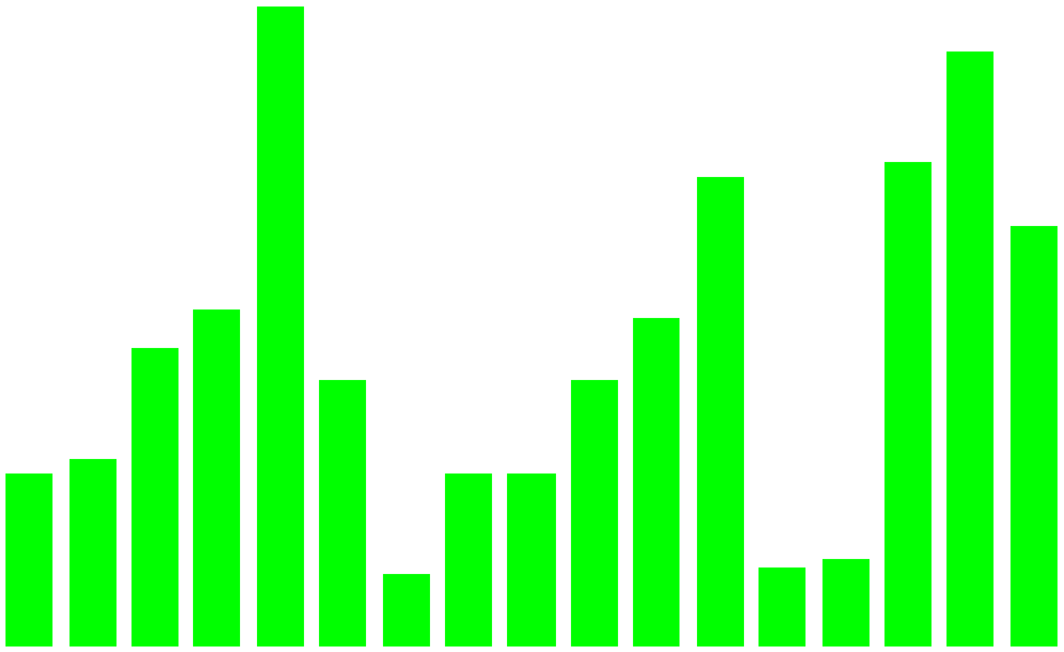

Figure 2.6: A Bar Graph

```
        set cn to child i of numbers
        divide child [i,0] of x by [[1,6,1],[cn, max-cn]]
        set color of child [1,0] of child [i,0] of x to green.

    draw x to screen as [400,600]
```

2.2.3 Julia Set Fractals

This Parcel script is capable of generating a wide variety of Julia set fractals.

```
set gridsizeCols to 1000
set gridsizeRows to 1000
set filename to "juliaFractal.png"

print (gridsizeCols + gridsizeRows)

set root to new cel
set colorSpace to new cel
set blowUpNumber to 100
set maxIts to 255
set br to random[]
set gr to random[]
set rr to random[]

divide root by [gridsizeRows, gridsizeCols]
```

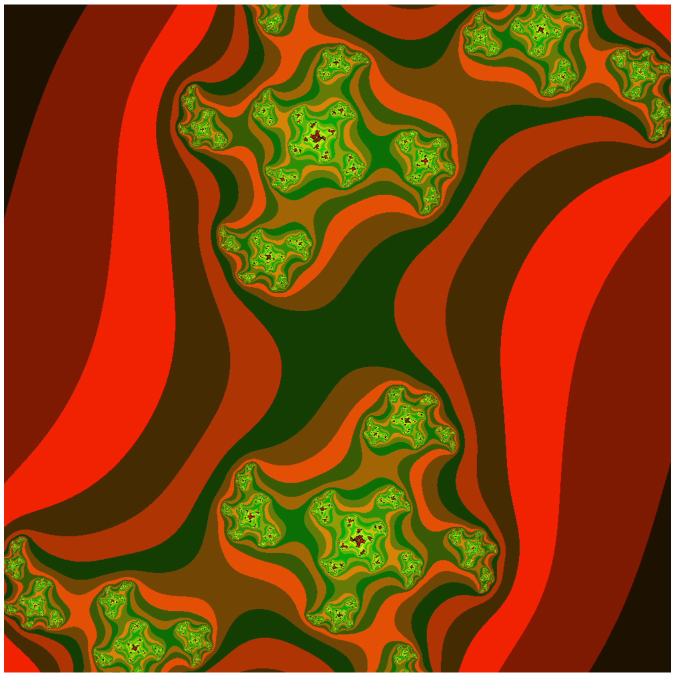

Figure 2.7: A Julia Set Fractal

```
define f[num]:
    return f4[num].

define f1[num]:
    set c to [0.422534, 0.525359]
    set num to iMultiply[num, num]
    return iPlus[num, c].

define f2[num]:
    set c to [0.422534, 0.525359]
    set num to iMultiply[iMultiply[num, num], num]
    return iPlus[num, c].

define f3[num]:
    set c to [0.422534, 0.525359]
    set num to iMultiply[num, iMultiply[num, num]]
    set num to (iPlus[num, num])
    return iPlus[num, c].

define f4[num]:
    set c to [0.422534, 0.525359]
    set num to iMultiply[iMultiply[num, num], iMultiply[num, num]]
    set num to (iPlus[num, num])
    return iPlus[num, c].
```

```
define iMultiply[num1, num2]:
    set x to first in num1
    set y to last in num1
    set a to first in num2
    set b to last in num2
    return [(x*a) - (y*b), (b*x) + (a*y)].


define iMinus[num1, num2]:
    return [first in num1 - first in num2, last in num1 - last in num2].

define iPlus[num1, num2]:
    return [(first in num1) + (first in num2), (last in num1) + (last in num2)].

define norm[numList]:
    set x to first in numList
    set y to last in numList
    return sqrt[(x*x) + (y*y)].

define colorValue[n]:
    set redc to (rr * 100 * n) mod 256
    set greenc to (gr * 10 * n) mod 256
    set bluec to (br * 1 * n) mod 256
    return Color[redc, greenc, bluec]
.

define getColor[numList]:
    set num to f[numList]
    set value to 0
    while (norm[num] < blowUpNumber) and (value < maxIts):
        set num to f[num]
        set value to (value + 1).
    set rColor to colorValue[value]
    return rColor.

print "Starting Fractal Generation"
print "Resolution: (" + gridsizeCols + ", " + gridsizeRows + ") "
print "Blowup Parameter: " + blowUpNumber
print "Max Iterations: " + maxIts

foreach aCel in root:
set x to (-1 + ((2 * (last in position of aCel)) / (1.0 * gridsizeCols)))
set y to (-1 + ((2 * (first in position of aCel)) / (1.0 * gridsizeRows)))
set num to [x, y]
set color of aCel to getColor[num]
set i to (last in position of aCel) + ((first in position of aCel) * gridsizeCols)
```

```
if ((100 * (i / (1.0 * gridsizeRows * gridsizeCols))) mod 1) is 0:
    print "Status : " + (100 * (i / (1.0 * gridsizeRows * gridsizeCols))).
if ((100 * (i / (1.0 * gridsizeRows * gridsizeCols))) mod 10) is 0:
    draw root to screen as [gridsizeCols/2, gridsizeRows/2]..

print "Fractal Generation Done"
draw root to screen as [gridsizeCols/2, gridsizeRows/2]
print "Writing Fractal to: " + filename
draw root to filename as [gridsizeCols, gridsizeRows]
```

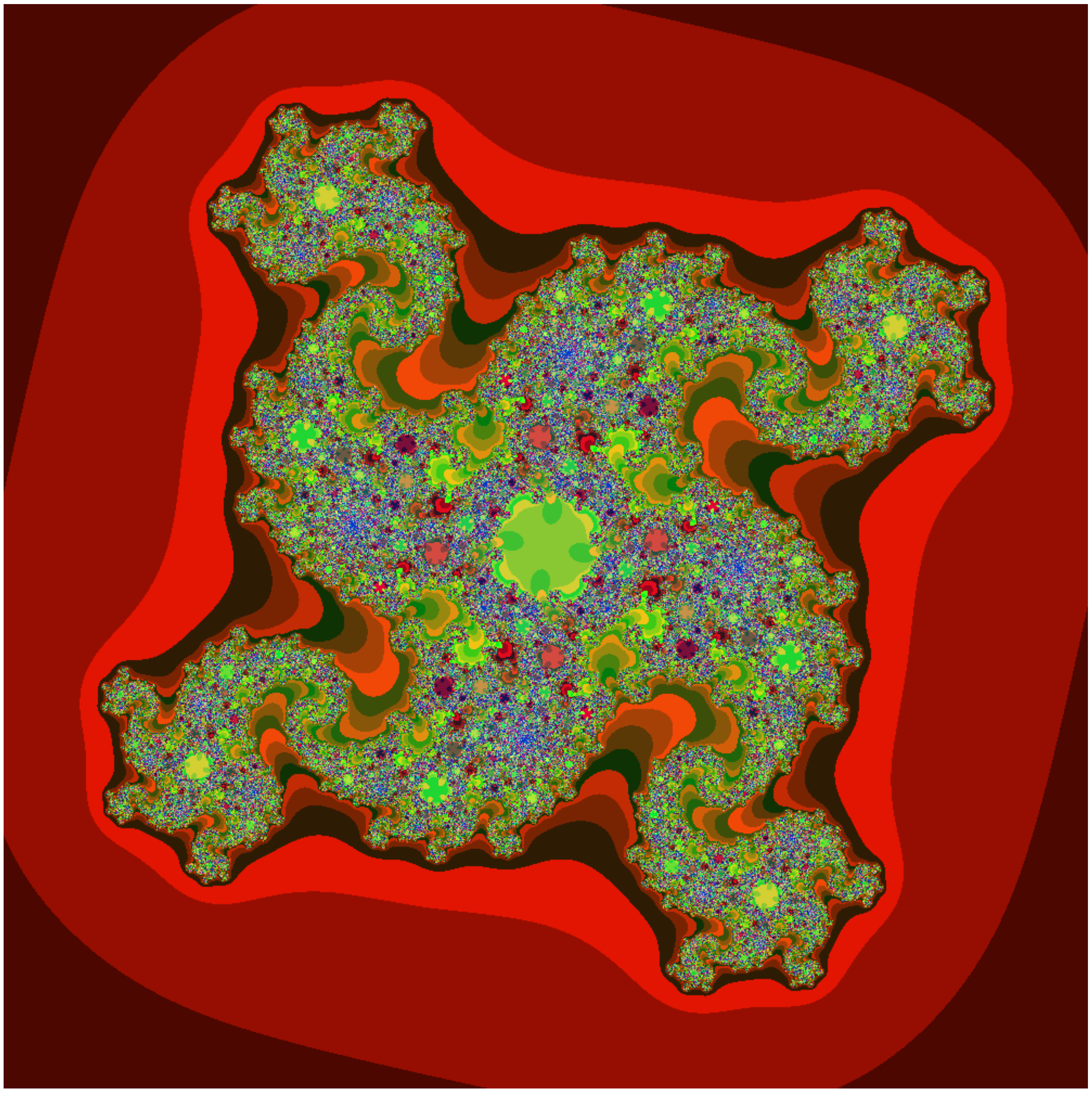

Figure 2.8: Another Julia Set Fractal

Chapter 3

Parcel Reference Manual

3.1 Lexical conventions

3.1.1 Comments

Multi-line comments begin with the characters `~{` and end with the characters `}~`. Single-line comments begin with “–”.

3.1.2 Identifiers

Identifiers are sequences of letters, numbers, and the underscore “_” character. Identifiers must begin with a letter.

3.1.3 Numbers

A number consists of the digits 0-9 and may include an optional decimal point “.”. There are no separate types for integers and floating point values and the Parcel interpreter automatically chooses the appropriate representation.

3.1.4 String literals

A string is a sequence of characters enclosed in double quotes `" "`.

3.1.5 List literals

List literals are a sequence of numbers, strings, identifiers, or other lists separated by commas “,” and enclosed in braces `[` and `]`.

3.2 Types

There are no explicit type declarations in Parcel; the language is dynamically typed. Data types and their values are determined at runtime, and their interaction is determined by the operator or function involved.

The types available in Parcel are:

- `integer` : 32-bit integers
- `double` : 64-bit IEEE floating format
- `string` : string of characters
- `list` : vector containing entries of any type
- `cel` : the primary graphical primitive
- `color` : RGB value
- `error` : used to indicate the presence of runtime errors

3.2.1 Scalars and lists

Integers, doubles, and strings fall under the category of scalars. They are considered single primitive values. This is in contrast to languages like C where strings are arrays of characters in memory.

Lists do not have a declared size and may grow and shrink as the user adds and removes elements. Lists may include scalar types, list types, cel and color types, other lists, or may be empty.

Parcel lacks an explicit boolean data type. True/false values are handled as in C: zero is false, and any non-zero value is true.

3.2.2 Cel and color

The cel is the primary graphical primitive of Parcel. It is the language's analog to a pixel in that cels have a single associated color, and all images produced and processed by Parcel programs will be made up of cels.

A cel, however, is sizeless. It can be scaled as small as a single pixel on the screen or as large as memory can accommodate. Instead of being arranged in an evenly-proportioned two-dimensional matrix, cel images are stored as hierarchies of cels. Cels may be divided with child cells defined proportionally to their parent.

3.3 Structure

3.3.1 Statements

A program in Parcel consists of a sequence of statements. Statements are terminated by newlines and include variable assignments, function calls, and control logic.

3.3.2 Identifiers, constants, and expressions

Identifiers are bound to values in the Parcel symbol table. These can be constants (either numbers or double-quoted strings) or any scalar, list, cel, or color variable.

Expressions are evaluated operations that return a value. Expressions include arithmetic, relational and logical operators, and function calls.

Variable assignment

An assignment takes on the form:

```
    set <left-value> to <right-value>
```

A valid `<left-value>` includes identifiers and expressions that return a child of a cel or the color of a cel.

A valid `<right-value>` includes any expression (including numbers, string and list literals) and the keywords "new cel" that create an instance of a cel with no parents and no children.

Operators

Because variable types are only resolved at runtime, Parcel handles operations on conflicting data types on the fly and in a way determined by the context of the operation.

Often, the variables involved will be cast to doubles in order for the operation to have any meaning. Here is how Parcel handles the casting of its data types to double:

- `integer` : double value of the integer
- `string` : attempts to parse the string as a double value
- `list` : double value of the size of the list
- `cel` : double value of the number of children
- `color` : double value of the sRGB value

Arithmetic operators Arithmetic operators use infix notation and take expressions as operands. They include addition "+", subtraction "-", multiplication "*", division "/", and modulus "%".

- When any operation involves a scalar variable and a list, the value returned is a new list with each member the result of the operation applied to the scalar and the corresponding member of the original list.
- The addition of two lists concatenates the two lists.
- The addition or subtraction of two colors returns the sum or difference of their sRGB values.
- The addition of a string with any data type concatenates the string with the string representation of that data type.
- The result of the division of two integers will be an integer if the dividend is a multiple of the divisor. Otherwise, the return value will be a float.
- If Parcel cannot find a way to return a meaningful value to an expression, a variable with type `error` is returned.

Relational operators Relational operators use infix notation and take expressions as operands. They include equals `=`, not equal to `!=`, greater than `>`, less than `<`, greater than or equal to `>=`, and less than or equal to `<=`. Relational operators return 0 if the relation is false and a non-zero value when the relation is true.

- When comparing to values of different types, each type is cast to a double value and then compared.
- When two list values are compared with `=` or `!=`, the operator returns true if and only if the relation on each corresponding member of both lists returns true.
- When two list values are compared with any other relational operator, it is the sizes of the two lists that are compared.

Logical operators Two logical operators, `and` and `or` use infix notation and take expressions as operands.

The `not` operator is a unary operator and takes a single expression as an operand.

The "of" and "in" operators Cel and list properties are accessed via the "of" and "in" operators.

```
<property_name> of <identifier>
```

The size of a list, for example, is referenced by:

```
size of list
```

A few properties that can be passed to the `of` and `in` operators are special in that return a proper `left-value` that can be `set`.

Cel:

- `child[<row>, <col>] of` : reference to the corresponding child cel
- `color of` : the color of this cel
- `first in` : the top-left-most child cel
- `last in` : the bottom-right-most child cel

List:

- `child <n> of` : reference to the `nth` member of the list
- `first in` : equivalent to `child 0 of`
- `last in` : equivalent to `child size of list of`

The remaining properties are read-only and may not be modified by the `set` command.

Cel:

- `size of` : the number of child cels
- `height of` : the number of rows of child cels
- `width of` : the number of columns of child cels
- `position of` : a tuple of the coordinates of this cel
- `parent of` : reference to the parent cel
- `superparent of` : reference to the root cel of the cel tree
- `neighbors of` : list of references to the adjacent cels

List:

- `size of` : the number of members in the list

3.3.3 Functions

Parcel includes a variety of built-in functions for general programming purposes and to facilitate cel manipulation. These functions are called with their own unique syntax and are discussed later in the reference manual. Furthermore, users may extend Parcel by writing their own functions, blocks of code that may be reused and called multiple times within the same program.

User-defined functions are called by invoking the identifier of the related function followed by a parameter list enclosed in braces.

```
randomItem[list]
```

If no parameters are passed, the braces are still necessary to identify the function call to the interpreter.

Function definitions

Functions can be defined within a Parcel program with the `define` keyword, followed by the function identifier, a parameter list enclosed in braces, a colon ":" to open the function block, a series of statements which may include a `return` statement, and a period "." to close the function block.

```
define fibonacci[n]:
    set r to 1
    if n != 0 and n != 1:
        set r to fibonacci[n-1] + fibonacci[n-2].
    return r.
```

Return statement

If a user-defined function should return a value to the calling block, the return statement must be placed at the end of a function definition.

3.3.4 Control

Control statements alter the normal flow of a program. In all cases, a control block begins with a keyword (either `if`, `while`, `for`, or `foreach`) and an `<expression>` to be evaluated. A colon ":" signals the opening of the control block, and a period "." terminates it.

Conditional statements

A conditional statement takes on the following form:

```
if <expression>:
    <statement>.
```

or

```
if <expression>:
    <statement>.
else:
    <statement>.
```

First, `<expression>` is evaluated. If its value is non-zero, then the first `<statement>` is evaulated. Otherwise, the second `<statement>` is evaluated. Afterwards, control is returned to calling block.

Iterative statements

Iterative statements are loops that enclose code to be repeatedly executed for a number of steps or until a condition is met.

while The while loop takes on the following form:

```
while <expression>:
    <statement>.
```

for and foreach The for loop takes on the following form:

```
for <identifier> from <number> to <number> by <number>:
    <statement>.
```

The `<identifier>` in this case is a variable local to the control block, which takes on the values denoted by the boundaries. By default, the increment is set to 1 unless the second `<number>` is less than the first in which case it is -1. The "by" part of the statement is optional and denotes the size of each step of the iteration.

The foreach loop iterates over elements of a list:

```
foreach <identifier> in <expression>:
    <statement>.
```

The `<expression>` must return a list and the `<identifier>` is a local variable that takes on the values of the members of that list.

Parallel for The `foreach` loop allows iteration over multiple lists at the same time. This is known as a parallel for and takes on the form:

```
foreach <id> in <expression> and <id> in <expression>:
    <statement>.
```

When iterating over two lists of unequal size, the loop terminates when the end of the shorter list or smaller range is reached.

3.4 Internal functions

3.4.1 Input and output

print

Output in Parcel is almost exclusively graphical. A command to print to standard output is included for functional and debugging purposes.

```
print <expression>
```

load

Parcel is able to read data from image files and (in a limited way) read data from text files. Image files are loaded and stored in cels. The target cel is divided into `m x n` child cels where `m` and `n` are the dimensions of the image. The color of each child cel corresponds to the color of that pixel in the image.

```
load image <expression> into <cel>
```

The `<expression>` must return a string that is a valid path to a supported image file (either GIF, JPEG, or PNG), and `<cel>` must have been already instantiated with `set`. Otherwise, the program halts on execution.

Data from text files are loaded into lists. The formatting of the text files is specific: each line will be loaded as a single member of the target list and each line may only contain a single number or numbers separated by tabs. As with image loading, `<list>` must have been already instantiated with `set`.

```
load data <expression> into <list>
```

This limitation is in place because Parcel is meant to specialize in graphical processing, and the language is meant to be small and precise. Other tools are available that can handle the text processing needed to take data and format it correctly for Parcel, which can then create the images to represent that data.

3.4.2 Math

Mathematical functions that are part of the Parcel language.

- `abs[n]` : absolute value
- `acos[n]` : arc cosine
- `asin[n]` : arc sine
- `atan[n]` : arc tangent
- `ceil[n]` : smallest integer greater than `n`
- `cos[n]` : cosine
- `floor[n]` : largest integer less than `n`
- `log[n]` : natural logarithm
- `max[m, n]` : the larger of `m` and `n`

- `min[m, n]` : the smaller of `m` and `n`
- `pow[m, n]` : m to the power of n
- `random[]` : pseudo-random number between 0 and 1
- `randomItem[list]` : random element from `list`
- `sin[n]` : sine
- `sqrt[n]` : square root
- `tan[n]` : tangent

3.4.3 Cel

draw

Prior to output, all cel dimensions are defined proportionally to the root cel of a cel tree. This allows the user to scale the actual up or down easily. Drawing is done by supplying a cel (it does not have to be a root cel), a target, and the dimensions of the final image.

```
    draw <cel> to <target> as <dimension>
```

The `draw` function will output the image of the cel and all child cels. The `<target>` may be the user's monitor (specified by the keyword `screen`) or a string representing the name of the file to write to. The file name must include a valid extension (either `.gif`, `.jpg`, or `.png`) and the file type is determined by this extension.

`<dimension>` is a tuple that specifies the pixel dimensions of the resulting image.

If the cel tree is unusually deep, processing for the `draw` function terminates once the resolution of a cel is determined to be less than a pixel.

divide

The `divide` function is easily the most complicated and robust function available in Parcel as it concerns the language's primary graphical primitive. The power of cels lies in the fact that they can be divided proportionally and organized in a hierarchical fashion.

```
    divide <cel> by <number>|<list>
```

In this case, `<list>` must be a list of two lists. The first list defines the division for the rows, and the second list defines the division for the columns.

```
    divide <cel> into rows|columns by <list>
```

If `<number>` is specified in the first case, the `<cel>` is divided into a `<number> x <number>` grid.

The `<list>` in each case defines the *proportions* of the child cels. Cels do not have associated sizes (until they're drawn, see above) and so cel dimensions are defined relationally.

Each list is a series of numbers, and each child cel's size in relation to its parent is the ratio of the corresponding number in the list to the sum of all numbers in the list. For example, a list of `[1, 2, 4]` defines three cels. The second cel is twice the size of the first, and the third cel is twice the size of the second.

Chapter 4

Project Plan

4.1 Team Responsibilities

Each member of the Parcel team holds an official title and is responsible for a particular section of code. This allocation of project responsibilities allows each of us to be in charge of a specific aspect of the development process while also being in charge of a certain section of the code for the language.

- Lawrence Wang – Compiler Architect
  - In charge of IO, graphics, and Cel related code
  - Cel.java
  - ParcelLibrary.java
- Blake Shaw – Project Manager
  - In charge of the interpreter and walker
  - ParcelLexer.java
  - ParcelParser.java
  - ParcelWalker.java
  - ParcelInterpreter.java
- Mike Ilardi – Systems Integrator
  - In charge of internal functions and the symbol table
  - PST.java
  - ParcelFunctionLibrary.java
- Tao Wang – Language Editor
  - In charge of operators and data structures
  - PDT.java
- Min Chae – Testing
  - In charge of test suite and sample programs

4.2 Project Log and Timeline

- 2/5/04 – Wiki created, brainstorming on website begins
- 2/16/04 – Settle on basics of Parcel, graphics based language, preliminary notion of cel and simple syntax
- 2/20/04 – Meeting to discuss the white paper
- 2/25/04 – White paper due
- 3/24/04 – Hello world program written (although without an interpreter for it)
- 3/26/04 – Meeting to settle grammar and features of Parcel, and lay out code structure for Interpreter, Parcel Data Type, and Cels
- 3/31/04 – Language Documentation Due
- 4/4/04 – Started work on Lexer/Parser/Walker, and preliminary aspects of backend as well
- 4/9/04 – Basic backend components start being integrated: Symbol Table, initial graphics library and Cel related code
- 4/9/04 – First demonstration of dividing cels and drawing them to the screen
- 4/16/04 – Cel code is optimized, Graphics Library and IO is done, The interpreter and walker are integrated with the backend. Only some work left to do on the Parcel Data Type code, and lots of testing and debugging.
- 4/23/04 – Meeting to practice the presentation. Each team member submits some example programs.
- 4/23/04 – Parcel function library finished
- 4/24/04 – Parcel Beta 0.1 released, close to full functionality
- 4/26/04 – Presentation
- 4/30/04 – Meeting to work on write up, and implement better error handling code.

4.3 Programming Guidelines

- General
  - Tab Width = 4
- Antlr
  - Lexer Tokens should be in upper case
  - Long grammar rules should be on their own line, with each production tabbed in under the rule, the first production with a colon, the rest with the `|` symbol, and then finally a semicolon tabbed in on a new line at the end.

  - Long embedded java code starts on its own line and is tabbed in one tab further than productions.

- Java
  - Clearly indent blocks by one tab.
  - The left brace used to open the block should come directly after the function definition or loop declaration. The closing right brace must vertically line up with the start of that block.
  - Multi-word variables and function names should be concatenated together with the first letter of every word except the first being in upper case.
  - Do not use braces when unnecessary (single-line `if` or `for` statements). Use minimal braces.
  - Use spaces between operators and operands.

- Parcel
  - Clearly indent blocks by one tab.
  - Periods which are used to close blocks should come after the last statement in the block on the same line.
  - Use spaces between operators and operands.

4.4 Development Process

Generally our development process involved meeting together and creating frameworks for code that needed to get written and then assigning members of the team to flesh out those sections. Then once the sections were completed, we would integrate them into the project and they would be tested.

We used a variety of tools during the Development Process:

- Java – We used Java 1.4.2 to create the backend of parcel.
- Antlr – We used Antlr 2.7.3 to create the front end for parcel.
- SubEthaEdit – SubEthaEdit is a simple collaborative text-editing application. During our meetings we would use SubEthaEdit to quickly prototype sections of code, or work on important sections of code as a group.
- Mac OS X Server – We used a server to hold latest versions of our code, with simple scripts to backup and save versions. In retrospect, using CVS would have been a smarter plan, however it was excellent being able to have a centralized server running SubEthaEdit, allowing us to see who has been working on which sections by color coding, and being able to work collaboratively with other team members over the internet.
- Email and Wiki – Staying synchronized is one of the hardest tasks for a team of 5 people. We used a wiki on a webserver and lots of email communication to keep people informed about important updates.

Chapter 5

Architecture Design

5.1 Design of the Interpreter

The Parcel interpreter consists of the following parts:

- pcl.java, which instantiates the various ANTLR-derived components and sends them the Parcel source file to be executed;
- a lexer, which breaks the source file into tokens;
- a parser, which builds the stream of tokens into an abstract syntax tree;
- a tree walker, which traverses the abstract syntax tree and performs the appropriate functions;
- the ParcelInterpreter class, which executes commands from the tree walker. The interpreter, as necessary, instantiates variables as PDT (Parcel Data Type) objects, stores and retrieves variables from a symbol table, and calls functions in the ParcelLibrary class, which manages data input and display output.

This set of relationships is displayed in figure 5.1.

The lexer, parser and walker are produced using ANTLR and the source files ParcelGrammar.g and ParcelWalker.g.

The PDT generic datatype implements integers, doubles, strings, lists, and cels. Dynamic typing is achieved by methods in PDT for each operation that perform type-checking via flags internal to each PDT instance, which are set upon assignment.

Error-handling is implemented during lexing and parsing (typographical errors, invalid characters), and during execution (type mismatches, array overflow).

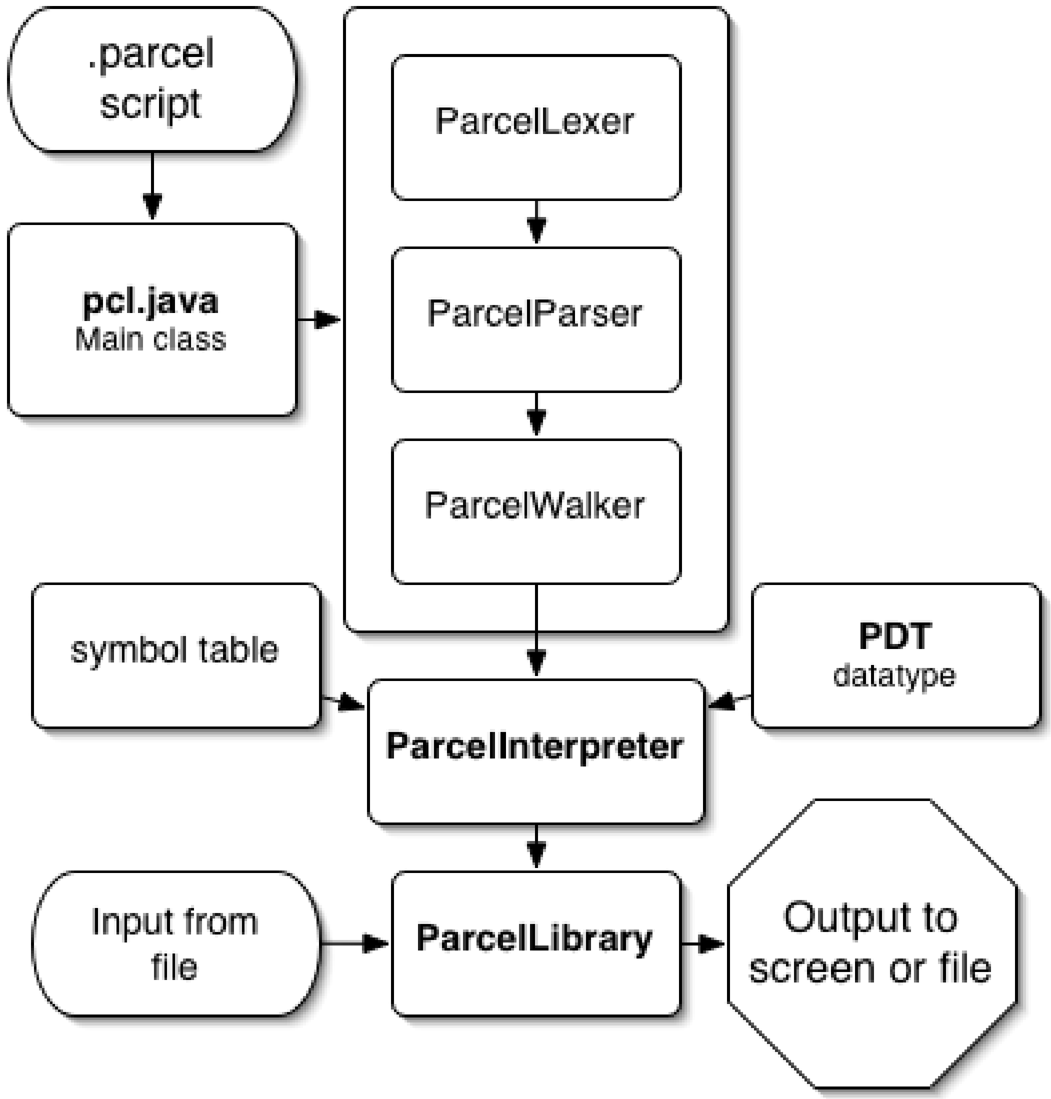

Figure 5.1: the Parcel Architecture

Chapter 6

Systems Integration

As a multi-platform development team, systems integration proved to be a particularly important aspect of the development of Parcel. Fortunately, the bulk of the work involved in the development of a functional cross-platform interpreter was nearly trivialized by the use of Java as the underlying software environment. Thusly, Parcel will run on any system capable of running version 1.4 of the Java Virtual Machine. However, as Java's purported platform independence is imperfect, problems may arise on certain systems. Particularly, Java's GUI components utilized are prone to exhibit unexpected behavior on some systems. Parcel has been tested on Mac OS X as well as the Win32 environment. Additionally, Parcel is expected to run on a variety of UNIX variants.

The Parcel interpreter expects to find UNIX-style line breaks in parcel source files. This is of particular import to those users running Parcel on a Windows-based system. While Windows systems use a carriage return, ASCII 13, and a line feed, ASCII 10, to represent a line break, UNIX-based systems, such as OSX, use only a line feed. Translating text files during development presented minor annoyances which were easily remedied through the use of a simple Perl script to strip the extra characters. Additionally, many UNIX systems provide commands to convert text files between the various formats. Furthermore, many text editors available for Windows, such as Emacs, make use of UNIX-style line breaks.

Parcel's lexer, parser, and tree-walker were implemented through Antlr, a compiler development tool designed to expedite the creation of the fundamental elements of a compiler or interpreter. The Antlr tool library was written in Java, which further eased the cross-platform development process. Other tools used by the Parcel development team include a server which maintained a code repository and a text editor, SubEthaEdit, for Mac OSX which allows multiple users to edit a document simultaneously. In terms of collaborative development, the latter of the mentioned tools proved invaluable as it provided a means for project members to work concurrently on a single section of code, thus greatly expediting the coding process.

Additionally, various shell scripts and Makefiles greatly simplified the development process. Those of us working in a UNIX environment made use of shell scripts and a Makefile (see appendix whatever) while those working in a Windows environment made use of batch files, which are functionally equivalent to shell scripts, in compiling and developing the Parcel interpreter.

Chapter 7

Testing

7.1 Test Plan

The overall approach of the testing process was to test the basic features first, and then to test more complex areas of the grammar. Our language was tested both at the development stage and the final stage using the prewritten testing suites. The main focus of testing was on the final stage of the language to make sure that all the features are working correctly. However, a substantial amount of testing was also involved frequently in the stage of developing the language by each implementer.

7.2 Test Cases

A few testing suites were written to check the features of our language. One testing suite was written for each statement in our language: set-statement, define-statement, do-statement, control-statements which again divided into for-statement, if-statement, while-statement. In each of these testing suites, everything, from the most basic features to the more complicated features, was tested to make sure the language worked correctly. After testing all of the features described in the grammar, we had to make sure that these features would work properly when combined in a more complex program. A few sample programs were written by each member for the purpose of demonstrating the key features and for discovering overlooked errors.

Here is the list of testing suites that were used for the fundamental features:

```
setTest.parcel: Used to test set-statement(assignment)
defTest.parcel: Used to test define-statement(function definition/call)
doTest.parcel: Used to test do-statement(draw, print, load, etc.)
ifTest.parcel: Used to test if-statement
forTest.parcel: Used to test for-statement
whileTest.parcel: Used to test while-statement
errorTest.parcel: Used to test error-handling
```

Here is one of the suites we used for testing:

```
~{
doTest.parcel: do-statement testing(draw, print, load, etc.)
Min Chae
}~


print "do_statement test"


print 10 > 12
print 10 < 12
print 10 is 12
print 10 is greater than 12 or 10 is less than 12
print 10>11 or 10>12 or 10>9

print ""

set root to new cel
set color of root to red
--draw root to screen as [600, 600]

print height of root
print width of root
print size of root

print ""

set i to 5
divide root by i
set child1 to child [0, 1] of root
set color of child1 to blue
--draw root to screen as [500, 500]


print position of root+" "+position of child1
print height of root+" "+height of child1
print width of root+" "+width of child1
print size of root+" "+size of child1

print ""

divide child1 into rows by [1, 2, 3, 4, 5]
set child2 to child [0, 0] of child1
set color of child2 to green

set nb to neighbors of child [2, 2] of root
foreach n1 in nb:
set color of n1 to black.
set color of child 1 of nb to white
--draw root to screen as [500, 500]
--draw child1 to screen as [500, 500]
--draw child2 to screen as [500, 500]
--draw child3 to screen as [500, 500]

print position of root+" "+position of child1+" "+position of child2
print height of root+" "+height of child1+" "+height of child2
print width of root+" "+width of child1+" "+width of child2
print size of root+" "+size of child1+" "+size of child2

draw root to "drawtest1.jpg" as [300, 300]

print ""

set root2 to new cel
set color of root2 to pink
divide root2 by 3
set child11 to child [0, 1] of root2
set color of child11 to yellow
divide child11 into columns by [5, 4, 3, 2, 1]

set child22 to child [0, 4] of child11
set color of child22 to green

draw root2 to screen as [500, 500]
--draw child11 to screen as [500, 500]
--draw child22 to screen as [500, 500]

print position of root2+" "+position of child11+" "+position of child22
print height of root2+" "+height of child11+" "+height of child22
print width of root2+" "+width of child11+" "+width of child22
print size of root2+" "+size of child11+" "+size of child22

--draw root2 to "drawtest2.jpg" as [400, 400]

print ""

set root3 to new cel
set color of root3 to red
divide root3 by [[1, 2, 3, 4], [4, 3, 2, 1]]
set color of child [3, 1] of root3 to blue
set color of child [1, 3] of root3 to yellow
--draw root3 to screen as [500, 500]
set z to new cel
load image "drawtest1.jpg" into z
set child [2, 2] of root3 to z
set z2 to new cel
load image "drawtest2.jpg" into z2
set child [2, 1] of root3 to z2

draw root3 to screen as [500, 500]
draw root3 to "drawtest3.jpg" as [500, 500]
```

7.3 Process Used to Fix Bugs

Throughout the testing phase, many bugs were found through the use of the testing code. Error messages generated by the Java Virtual Machine and Antlr libraries aided in debugging the Parcel interpreter. Additionally the graphical AST debugger helped in determining errors in the grammar. After making sure the grammar and generated tree were valid, we checked the more complex Parcel programs written by each member of the team. The majority of errors turned out to be problems in the Java code implementing our backend, and were therefore fairly easy to uncover.

The process used to fix bugs was fairly simple. Whenever bugs were found, descriptions of bugs and error messages were sent to the team via email. The member responsible for the module containing the offending code was typically able to fix the bug right away. However, more complicated bugs and language issues were discussed at team meetings where the underlying cause would be determined and a member would be assigned to implement a solution.

7.4 Known Issues

We had some issues with platform incompatibility between Macs and PCs. Sometimes, features would only work on one platform and not the other. Choice of editing tools also had an affect on the outcome. The PC users in our group had to make sure to save the parcel file as UNIX file format. Here are the known bugs:

Commenting

```
~{blah
}~
```

These statements work correctly. But if there is no newline right before the second commenting bracket, it gives an error. The following are incorrect:

```
~{blah}~
~{blah
   }~
```

Divide

```
divide by i+1
divide by (i+1)
```

These statements give an error or invalid outcome, because `divide` can only take a number or a variable as an argument.

Conditional Statements

```
if / while / for statement:
set i to (i+1).
define fn[i]:
return (i+1).
```

works correctly.

However,

```
if / while / for statement:
set i to i+1.

define fn[i]:
return i+1.
```

gives an error. Be sure to use parentheses when you use compound expression inside a function, control statement. The error comes from having a number at the end of a block. The period is not parsed correctly in that case.

Draw

works correctly on Macs, however seems to have a problem on PCs with refreshing the image to the screen. The image is not shown, unless you drag the image window back and forth.

is not

`is not` does not work properly when testing inequality of numbers. Use `!=` instead.

Chapter 8

Project Baselining

Language design proceeded in three phases over the life-span of the project: overall language goals, detailed language design, and language implementation. Deciding overall language goals was a matter of targeting the kinds of applications we wanted to see and implement. Detailed language design involved working out the basic syntax and data structures that would be required to meet our language goals. Language implementation was the actual coding of all the ANTLR grammar and all Java classes necessary for a working deliverable.

The first two phases, roughly corresponding to the delivery of the whitepaper and the reference manual, were undertaken without writing any actual code. Three design decisions that define the Parcel language were made at this time. First, Parcel would have a solely graphical angle and be a language specializing in image creation and processing. In the beginning, our target applications were fractals and cellular automaton.

Second, Parcel would be based on the cel, a graphical primitive described in more detail in the language reference manual. The properties of the cel were chosen to create an elegant, abstract analog to pure pixel processing.

Third, Parcel syntax would be closer to natural language than most programming languages. Two of the designers are fans of AppleScript, and since the applications we had in mind would appeal to both programmers and non-programmers, it was felt that the more natural syntax would make Parcel more accessible to a wider variety of users.

The design of Parcel changed little after the reference manual was set. Originally, it was envisioned that Parcel could be a jack-of-all-trades language like Perl. Regular expressions and more robust file input and output were a couple of features removed in order to meet project deadlines and to streamline the language. In the end, it was decided that these features were not essential to the overall vision of Parcel and that there were tools already available that would do any of these things much better than we could implement.

Chapter 9

Lessons Learned

9.1 Group

- Start early
- Spec things out early
- Code version management is important
- Jobs are flexible in small groups

9.2 Blake Shaw

Coordination is vital in a project involving many people. It is crucial that people are working with each other and not against each other. One of the best ways to do this is to assign ownership of certain aspects of the project to different team members and then encourage collaboration when different aspects overlap. It was a pleasure working with this team on Parcel. Each member contributed a unique style to every part of code that they were working on. My only regret is not setting up the code structure earlier, so each member of the team could have had better flexibility in choosing what parts of the project they wanted to work on.

9.3 Min Chae

I have learned a great deal about the teamwork while participating in this project. However, the most important thing that I learned was about the interaction between team members. Respecting other team members was the most important thing to the team project in my mind. Without the respect between the team members, it is impossible to have a good team work, and without the good team work, it is impossible to have a good outcome as a team. Having a faith in your team members was another important thing that I learned. I had faith in each members of our team and each member of our team showed each other great respect, and this was the biggest factor that drove me through the project with the energy and the excitement.

9.4 Mike Ilardi

Even the most carefully thought out piece of code is subject to hidden algorithmic errors and flaws in design based upon incorrect assumptions. Very often, and in spite of extensive testing, these errors will not reveal themselves for quite some time, luring the developer into a false sense of security before they rear their ugly heads, forcing the rethinking of the solution. Such is the nature of something as large and complex as a compiler or interpreter. The only real solution is to allot sufficient time for the majority of such errors to show themselves before the deadline.

9.5 Lawrence Wang

I learned that division of labor can be a blessing or a curse, depending on how it's done. In this project, my official title was Compiler Architect, but as the project went on, I found myself working more and more on elements of the Java backend, while our Project Manager, Blake, took over much of the task of developing the tree walker.

The advantages of division of labor are a clear sense of giving each person a task that doesn't overlap with anyone else's, making it easy to work independently, especially when this division is reinforced by separate source files for each group member. However, this division of labor must be made with a clear and precise understanding of the project as a whole; when that is lacking, the roles given are bound to change as the group's understanding of the project changes. In other words, a good initial spec is crucial, if not for the project itself, then at least for intelligently adapting group members' roles when new issues emerge, to keep everyone working together on the same wavelength.

9.6 Tao Wang

The best thing to take out of the Parcel project is abstraction in design. There are a number of languages that can do graphical processing, but none include the elegance of the cel data structure. It would have been easy to flood Parcel's standard library with methods to draw bezier curves, gaussian blurs, and any number of filters you can find in a modern image editor like Photoshop.

Instead, by some kind of inspiration when we were constructing Parcel, we did not ask what things we wanted to do easily, but what basic elements and tools would make what we wanted to do easy. That extra step took us a long way. As the language editor, I saw what originally was a plan for a glorified graphics library add-on to Java turn into a robust and precise language. That we can do things like Mike's image warp and Blake's fractals (and others like the game of life) in 20 lines of code or less using only the primitives we devised is a testament to the robustness of Parcel and the efficacy of abstract design.

Chapter 10

Parcel Code

10.1 Code Listing

```
Makefile
parcelGrammar.g
parcelWalker.g
Cel.java
PDT.java
PST.java
ParcelFunctionLibrary.java
ParcelInterpreter.java
ParcelLibrary.java
pcl.java
```

10.2 Makefile

```
default:
    java antlr.Tool parcelGrammar.g
    java antlr.Tool parcelWalker.g
    javac ParcelInterpreter.java
    javac pcl.java

run:
    java pcl -debug simple.parcel

clean:
    rm *.class

jar:
    jar cvmf mainClass parcel.jar *.class

runJar:
    java -jar parcel.jar -debug "simple.parcel"

debug:
    java pcl -debug errorTest.parcel
```

10.3 parcelGrammar.g

```
class ParcelParser extends Parser;
options {
    k=2;
    buildAST = true;
    exportVocab = ParcelVocab;
}

tokens {
PROGRAM;
STATEMENT;
STATEMENTS;
LIST;
LOGICAL_TEST;
CEL_LIST_OF;
FUNC_CALL;
EXPRESSION;
ELSE_STATEMENT;
LESS_OR_EQUAL;
GREATER_OR_EQUAL;
LOAD_OPTIONS;
DIVIDE_DIRECTION;
SETTABLE;
UNARY_MINUS;

}


program
    : statements EOF!
       {#program = #([PROGRAM,"parcel program"], program); }
    ;

statements
    : (statement)*
        {#statements = #([STATEMENTS,"statements"], statements); }
    ;

statement
```

```
    : set_stmt
    | do_statement
    | func_call
    | define_statement
    | control_statement
    ;

set_stmt
    : "set"^ settable "to"! (expr | "new"! "cel")
    ;

settable
    : ID
        {#settable = #([SETTABLE,"settable"], settable); }
    | "child"! expr "of"! cel
        {#settable = #([CEL_LIST_OF,"child of another cel"], settable); }
    | cel_property
    ;

cel_property
    : "color"^ "of"! expr
    ;

func_call
    : ID list
        {#func_call = #([FUNC_CALL,"func_call"], func_call); }
    ;

logical_tests
    : logical_test ("or"^ logical_test)*
        {#logical_tests = #([EXPRESSION,"expression"], logical_tests); }
    ;

logical_test
    : logical_test2 ("and"^ logical_test2)*
    ;

logical_test2
    : ("not"^)? logical_test3
    ;

logical_test3
    : expr (( "is"^ | "is"! "less"^ "than"! | "is"! "greater"^ "than"!
    | EQUAL^ | NOT_EQUAL^ | LESS_THAN^ | GREATER_THAN^ | LESS_OR_EQUAL^
    | GREATER_OR_EQUAL^) expr)?
    ;

expr
```

```
    : expr2
    | "true"
    | "false"
    ;

expr2
    : expr3 ((PLUS^ | MINUS^) expr3)*
    ;

expr3
    : expr4 ((MULTIPLY^ | DIVIDE^ | MOD^ | "mod"^) expr4)*
    ;

expr4
    : MINUS! cel
      {#expr4 = #([UNARY_MINUS,"unary minus"], expr4); }
    | cel
    ;

cel
    : ("height"^ | "width"^ | "size"^ | "position"^ | "neighbors"^ | "color"^) "of"! cel
    | ("parent"^ "of"! | "superparent"^ "of"! | "first"^ "in"! | "last"^ "in"!) cel
    | "child"! expr "of"! cel {#cel = #([CEL_LIST_OF,"child of another cel"], cel); }
    | atom
    ;

atom
    : NUM
    | ID
    | func_call
    | list
    | STRING_LITERAL
    | OPEN_PAREN! logical_tests CLOSE_PAREN!
    ;



do_statement
    : "print"^ logical_tests
    | "return"^ expr
    | load_statement
    | "draw"^ expr "to"! draw_output ("as"! list)?
    | "divide"^ expr divide_direction "by"! divide_size_relation
    ;

draw_output
    : "screen"
    | expr
    ;

load_statement
    : "load"^ load_type expr "into"! expr
    ;


load_type
    : "data"
    | "image"
    ;

divide_direction
    : "into"! "rows"
        {#divide_direction = #([DIVIDE_DIRECTION,"direction"], divide_direction); }
    | "into"! "columns"
        {#divide_direction = #([DIVIDE_DIRECTION,"direction"], divide_direction); }
    |
        {#divide_direction = #([DIVIDE_DIRECTION,"direction"], divide_direction); }
    ;

divide_size_relation
    : list
    | NUM
    | ID
    ;

string
    : STRING_LITERAL
    | ID
    ;

define_statement
    : "define"^ ID list COLON! statements PERIOD!
    ;

control_statement
    : while_statement
    | if_statement
    | for_statement
    | for_each_statement
    ;

while_statement
    : "while"^ logical_tests COLON! statements PERIOD!
    ;

list
```

```
    : OPEN_BRACKET! (expr (list_end)*)? CLOSE_BRACKET!
        {#list = #([LIST,"list"], list); }
    ;

list_end
    : COMMA! expr
    ;


if_statement
    : "if"^ logical_tests COLON! statements PERIOD! (else_statement)?
    ;

else_statement
    : "else"^ COLON! statements PERIOD!
    ;

for_statement
    : "for"^ ID for_statement_it COLON! statements PERIOD!
    ;

for_statement_it
    :"from"^ expr "to"! expr ("by"! expr)?
    ;

for_each_statement
    : "foreach"^ for_each_statement_it COLON! statements PERIOD!
    ;

for_each_statement_it
    : ID "in"^ expr ("and"! ID "in"! expr)*
    ;



class ParcelLexer extends Lexer;

options {
    k=2;
}

protected
ALPHA:    'a'..'z' | 'A'..'Z'
    ;

protected
DIGIT:  '0'..'9';
EQUAL:  '=';
```

```
NOT_EQUAL:  "!=";
COMMA: ',';
OPEN_PAREN: '(';
CLOSE_PAREN: ')';
OPEN_BRACKET:   '[';
CLOSE_BRACKET:  ']';
COLON:  ':';
PERIOD: '.';
LESS_THAN:  '<';
LESS_OR_EQUAL: "<=";
GREATER_THAN:  '>';
GREATER_OR_EQUAL: ">=";
PLUS:   '+';
MINUS:  '-';
MULTIPLY:   '*';
DIVIDE: '/';
MOD:    '%';


ID  options { testLiterals = true; }
        : ALPHA (ALPHA | DIGIT | '_')*
        ;

NUM:       (DIGIT)+ ('.' (DIGIT)+)? ;


WS:        (' ' | '\t' | ('\r' | '\n') {newline();})
                {$setType(Token.SKIP);}
    ;

COMMENT
    :    "--" (~('\n' | '\r'))* '\n'
            { newline(); } {$setType(Token.SKIP);}
    | '~' '{' ((~ ('~' | '{' | '\n' | '\r'))* '\n' { newline(); })* '}' '~'
            {$setType(Token.SKIP);}
    ;


STRING_LITERAL:        '"' (~('"'))* '"'
    ;
```

10.4 parcelWalker.g

```
{
import java.util.*;
}
```

```
class ParcelWalker extends TreeParser;
options
{
importVocab = ParcelVocab;
}

{

ParcelInterpreter pInt;

public ParcelWalker(ParcelInterpreter interp){
    new ParcelWalker();
    pInt = interp;
}

}

expr returns [PDT s]
{
Vector aVector;
String aString;
AST anAST;
PDT a, b, c, d;
s = new PDT();

}
        : #(PROGRAM a=expr) { s = a;}
    | #(STATEMENTS (statement:. { s = expr(#statement); } )*)
    | #("set" set_thing:. b=expr) {
        if (set_thing.getText().equals("color")){
            anAST = set_thing.getFirstChild();
            a = expr ( #anAST );
            a.setColor(b.getColor());
        } else if (set_thing.getText().equals("child of another cel")){
            a = expr( #set_thing );
            a.setCel(b);
        } else {
            s = pInt.assign(expr( #set_thing ), b);
        }
    }
    | #("print" a=expr) { s = pInt.print(a); }
    | #("draw" a=expr b=expr c=expr (color_option:.)?)
        { if(color_option==null){
             d = new PDT();
        } else {
            d = expr( #color_option );
        }
         s = pInt.draw(a, b, c, d);
        }
    | #(FUNC_CALL func_call_name:. b=expr)
        {
            aString = func_call_name.getText();
            s = pInt.function_call(aString, b);
        }
    | #("return" a=expr) { s = pInt.returnStatement(a); }
    | #("define" func_name:. func_params:. function_action:.)
        {
        aString = func_name.getText();
        aVector = new Vector();
        if((anAST = func_params.getFirstChild()) != null){
            aVector.add(anAST.getText());
            while((anAST = anAST.getNextSibling()) != null){
                aVector.add(anAST.getText());
            }
        }
         s = pInt.defineNewFunction(aString, aVector, #function_action);

        }
    | #("load" a=expr b=expr c=expr)
        {
        s = pInt.load(a, b, c);
        }
    | #("divide" a=expr b=expr c=expr) { s = pInt.divide(a, b, c);
                                          pInt.check(s, "divide"); }
    | #(DIVIDE_DIRECTION (direction_id:.)?)
        {
            if(direction_id != null){
                s = new PDT(direction_id.getText());
            } else {
                s = new PDT();
            }
        }
    | #("if" a=expr action:. (else_action:.)?)
        {
            if( pInt.evaluateLogical(a) ) {
                s = expr( #action );
            } else if(else_action != null){
                s = expr( #else_action );
            }
        }
    | #("else" a=expr) { s = a; }
    | #("for" iterator:. b=expr for_action:.)
        {
```

```
        aString = iterator.getText();
        pInt.initializeForLoop(aString, b);
        do {
            s = expr( #for_action );
        } while( pInt.forLoopContinue(aString, b) );
        pInt.endBlock();
    }
| #("from" { aVector = new Vector(); }
        (a = expr { aVector.add(a); })*
    ) { s = new PDT(aVector); }
| #("foreach" a=expr for_each_action:.)
    {
        pInt.startBlock();
        int counter = 0;
        while( pInt.forEachLoopContinue(a, counter) ){
            s = expr( #for_each_action );
            counter++;
        }
        pInt.endBlock();
    }
| #("in" { aVector = new Vector(); int fCount = 0; }
        (theToken:.
            { if(fCount == 0){
                aVector.add(new PDT(theToken.getText()));
                fCount = 1;
                } else {
                aVector.add( expr(#theToken) );
                fCount = 0;
                }
            }
        )*
    ) { s = new PDT(aVector); }
| #("while" while_test:. while_action:.)
    {
        pInt.startBlock();
        while( pInt.evaluateLogical(expr( #while_test )) ){
            s = expr( #while_action );
        }
        pInt.endBlock();
    }
| #(LIST { aVector = new Vector(); }
    (a = expr { aVector.add(a); })*
    ) { s = new PDT(aVector); }
| #(EXPRESSION a=expr) { s = a; }
| #("is" a=expr b=expr) { s = a.is(b); pInt.check(s, "is"); }
| #(EQUAL a=expr b=expr) { s = a.is(b); pInt.check(s, "equal"); }
| #("greater" a=expr b=expr) { s = a.greater(b); pInt.check(s, "greater");}
| #(GREATER_THAN a=expr b=expr) { s = a.greater(b); pInt.check(s, "greater");}
```

```
    | #(GREATER_OR_EQUAL a=expr b=expr) { s = a.greaterOrEqual(b);
         pInt.check(s, "greater or equal");}
    | #("less" a=expr b=expr) { s = a.less(b); pInt.check(s, "less");}
    | #(LESS_THAN a=expr b=expr) { s = a.less(b); pInt.check(s, "less");}
    | #(LESS_OR_EQUAL a=expr b=expr) { s = a.lessOrEqual(b);
         pInt.check(s, "less or equal");}
    | #(NOT_EQUAL a=expr b=expr) { s = a.notEqual(b); pInt.check(s, "not equal");}
    | #("and" a=expr b=expr) { s = a.and(b); pInt.check(s, "and");}
    | #("or" a=expr b=expr) { s = a.or(b);  pInt.check(s, "or");}
    | #(PLUS a=expr b=expr) {s = a.plus(b); pInt.check(s, "+"); }
    | #(MINUS a=expr b=expr) { s = a.minus(b); pInt.check(s, "-"); }
    | #(UNARY_MINUS a=expr) { s = a.times(new PDT(-1)); }
    | #(MULTIPLY a=expr b=expr) { s = a.times(b); pInt.check(s, "mult"); }
    | #(DIVIDE a=expr b=expr) { s = a.divideBy(b); pInt.check(s, "divide"); }
    | #(MOD a=expr b=expr) { s = a.mod(b); pInt.check(s, "mod");}
    | #("mod" a=expr b=expr) { s = a.mod(b); }
    | #("neighbors" a=expr) { s = a.neighbors(); pInt.check(s, "neighbors" ); }
    | #("height" a=expr) { s = a.height(); pInt.check(s, "height" ); }
    | #("width" a=expr) { s = a.width(); pInt.check(s, "width" ); }
    | #("size" a=expr) { s = a.size(); pInt.check(s, "size" ); }
    | #("position" a=expr) { s = a.position(); pInt.check(s, "position" ); }
    | #("first" a=expr) { s = a.first(); pInt.check(s, "first" ); }
    | #("last" a=expr) { s = a.last(); pInt.check(s, "last" ); }
    | #("color" a=expr) { s = a.color(); pInt.check(s, "color" ); }
    | #(CEL_LIST_OF a=expr b=expr) { s = b.getChild(a); pInt.check(s, "child" ); }
    | #(SETTABLE theSettable:ID) { s = pInt.getID(theSettable.getText()); }
    | theID:ID { s = pInt.getValue(theID.getText());
                 pInt.check(s, "Object not found");}
    | theNum:NUM { s = pInt.parseNumber(theNum.getText()); }
    | theStringLiteral:STRING_LITERAL
         { s = pInt.parseString(theStringLiteral.getText()); }
    | "screen" { s = new PDT("screen"); }
    | "data" { s = new PDT("data"); }
    | "image" { s = new PDT("image"); }
    | "cel" { s = new PDT(new Cel()); }
    ;
```

10.5 Cel.java

```
/*
Cel.java
Implementation of Cel data structure for Parcel
Lawrence Wang
*/
import java.util.*;
import java.awt.*;
import java.lang.*;

public class Cel {
    public static boolean ROWS = true;
    public static boolean COLS = false;

    private Cel parent;
    private int height, width, size;
    private Color color;

    public Vector children;
    private double[] rowProps, colProps;
    private boolean equalRows, equalColumns;
    private int indexInParent;

    public Cel() {
        new Cel(null, 0, Color.BLACK);
        children = new Vector();
    }

    public Cel(Cel parent, int index, Color c) {
        height = 1;
        width = 1;
        children = new Vector();
        this.parent = parent;
        this.indexInParent = index;
        this.color = c;
    }

    public void divide(int rows, double[] cols) {
        double coldenom = 0;
        // initialize column widths
           for (int i=0; i<cols.length; i++)
            coldenom += cols[i];
        colProps = new double[cols.length];
        for (int i=0; i<cols.length; i++)
            colProps[i] = cols[i]/coldenom;

        divide(rows, cols.length);
        equalRows = true;
        equalColumns = false;
    }

    public void divide(double[] rows, int cols) {
        double rowdenom = 0;
        // initialize row heights
        for (int i=0; i<rows.length; i++)
            rowdenom += rows[i];
        rowProps = new double[rows.length];
        for (int i=0; i<rows.length; i++)
            rowProps[i] = rows[i]/rowdenom;

        divide(rows.length, cols);
        equalRows = false;
        equalColumns = true;
    }

    public void divide(double[] rows, double[] cols){
        double rowdenom = 0, coldenom = 0;

        // initialize row heights
        for (int i=0; i<rows.length; i++)
            rowdenom += rows[i];
        rowProps = new double[rows.length];
        for (int i=0; i<rows.length; i++)
            rowProps[i] = rows[i]/rowdenom;

        // initialize column widths
           for (int i=0; i<cols.length; i++)
            coldenom += cols[i];
        colProps = new double[cols.length];
        for (int i=0; i<cols.length; i++)
            colProps[i] = cols[i]/coldenom;

        divide(rows.length, cols.length);
        equalRows = false;
        equalColumns = false;
    }

    public void divide(int divisions) {
        divide(divisions, divisions);
    }


    public void divide(int rows, int columns) {
        height = rows;
        width = columns;
        children.clear();
        for(int i=0; i<rows*columns; i++)
            children.add(color);
        equalRows = true;
        equalColumns = true;
    }

    /* Accessing Properties */
```

```
public Color getColor(){
    return color;
}

public void setColor(Color color) {
    this.color = color;
}

public int getHeight() {
    return height;
}

public int getWidth() {
    return width;
}

// formerly getSize()
public int numChildren() {
    return children.size();
}

/* Accessing Cels */

// setChild(cel, index) does not change the child's parent pointer,
// so this info may be out of date. the plus side is that the same
// cel can be the child of many different parents.
public Cel getParent() {
    return parent;
}

public Cel getRoot() {
    Cel parent = this.getParent();
    if (parent != null) {
        return parent.getRoot();
    }

    return this;
}

public int getIndexInParent() {
    return indexInParent;
}

public void setIndexInParent(int index) {
    indexInParent = index;
}
```

```
    /* Cel child methods */

/*    this method may be unusable since the children vector is
    a mix of cels and baby cels... unless you write getChildren()
    in PDT to mediate. is this method even necessary?
    public Vector getChildren() {
        return children;
    }
*/

    public int getIndex(int row, int col) {
           return row * width + col;
    }

    public int getRow(int index){
        return (int)(Math.floor(index / width));
    }

    public int getColumn(int index){
        return index % width;
    }

    public boolean isBaby(int row, int col) {
        return isBaby(getIndex(row, col));}
    public boolean isBaby(int index) {
        // is the target child a Cel or not?
        return !(children.elementAt(index) instanceof Cel);
    }

    public Cel getChild(int row, int col) {
        return getChild(getIndex(row, col));
    }

    public Cel getChild(int index) {
        return (Cel) children.elementAt(index);
    }

    public void setChild(Cel c, int row, int col) {
        setChild(c, getIndex(row, col));
    }

    public void setChild(Cel c, int index) {
        children.setElementAt(c, index);
    }

    public Color getBabyColor(int row, int col) {
        return getBabyColor(getIndex(row, col));}
    public Color getBabyColor(int index) {
```

```
    return (Color) children.elementAt(index);
}

public void setBabyColor(Color c, int row, int col) {
    setBabyColor(c, getIndex(row, col));}
public void setBabyColor(Color c, int index) {
    children.setElementAt(c, index);
}

public Cel matureBaby(int row, int col) {
    return matureBaby(getIndex(row, col));}
public Cel matureBaby(int index){
    Cel c = new Cel(this, index, getBabyColor(index));
    children.setElementAt(c, index);
    return c;
}

// return height of child as fraction of parent's height
public double getYProportion(int row) {
    if (equalRows)
        return (double)1/height;
    else
        return rowProps[row];
}

// return width of child as fraction of parent's width
public double getXProportion(int col) {
    if (equalColumns)
        return (double)1/width;
    else
        return colProps[col];
}

// return y position in parent cell, where top = 0 and bottom = 1
public double getYPosition(int row) {
    if (equalRows)
        return (double)row/height;
    else {
        double total = 0;
        for (int i=0; i<row; i++)
            total += rowProps[i];
        return total;
    }
}

// return x position in parent cell, where left = 0 and right = 1
public double getXPosition(int column) {
    if (equalColumns)
```

```
            return (double)column/width;
        else {
            double total = 0;
            for (int i=0; i<column; i++)
                total += colProps[i];
            return total;
        }
    }
}
```

10.6 PDT.java

```
/*
PDT.java
General data type for Parcel
Tao Wang
*/
import java.util.*;
import java.awt.*;
import java.awt.geom.*;

public class PDT {
    public static final int INT     = 0;
    public static final int DOUBLE  = 1;
    public static final int STRING  = 2;
    public static final int LIST    = 3;
    public static final int CEL     = 4;
    public static final int BABYCEL = 5;
    public static final int COLOR   = 6;
    public static final int ERROR   = 7;

    public static final PDT TRUE  = new PDT(1);
    public static final PDT FALSE = new PDT(0);
    public static final PDT ZERO  = new PDT(0);

    private int    intValue;
    private double doubleValue;
    private String stringValue;
    private Vector listValue;
    private Color  colorValue;
    private int    type;

    /* For a full-fledged cel, this is a pointer to it.
     * For a baby, this is a pointer to its parent. */
    private Cel    celValue;
```

```
/* the index of the cel in its parent. */
private int    indexInParent;

public PDT() {
}

public PDT(int tn, String message) {
    this.type = tn;
    stringValue = message;
}

public PDT(int i) {
    clearEverything();
    type = INT;
       intValue = i;
}

public PDT(double d) {
    clearEverything();
    type = DOUBLE;
    doubleValue = d;
}

public PDT(String s) {
    clearEverything();
    type = STRING;
    stringValue = s;
}

public PDT(Vector v) {
    clearEverything();
    type = LIST;
    listValue = v;
}

public PDT(Cel c) {
    clearEverything();
    type = CEL;
    celValue = c;
    indexInParent = c.getIndexInParent();
}

public PDT(Cel parent, int index) {
    clearEverything();
    if (parent.isBaby(index)) {
        type = BABYCEL;
        celValue = parent;
    }
    else {
        type = CEL;
        celValue = parent.getChild(index);
    }
    indexInParent = index;
}

public PDT(Color c) {
    clearEverything();
    type = COLOR;
    colorValue = c;
}

private void clearEverything() {
    intValue     = 0;
    doubleValue  = 0.0;
    stringValue  = null;
    listValue    = null;
    celValue     = null;
    indexInParent = 0;
    colorValue   = null;
    type         = -1;
}

public int getType() {
    return type;
}

public void setType(int t) {
    type = t;
}

public void setVector(Vector v) {
    type = LIST;
    listValue = v;
}

public String getTypeString(){
    switch (type) {
        case INT   : return "number";
        case DOUBLE: return "number";
        case STRING: return "string";
        case LIST  : return "list";
        case CEL   : return "cel";
        case COLOR : return "color";
        case ERROR : return "error";
        default    : return "null pdt object";
    }
}

/* Access */

public int getInt(){
    switch (type) {
        case INT   : return intValue;
        case DOUBLE: return (int) doubleValue;
        case STRING: return Integer.parseInt(stringValue);
        case LIST  : return listValue.size();
        case CEL   : return celValue.numChildren();
        case COLOR : return colorValue.getRGB();
        default    : return 0;
    }
}

public double getDouble() {
    switch (type) {
        case INT   : return (double) intValue;
        case DOUBLE: return doubleValue;
        case STRING: return Double.parseDouble(stringValue);
        case LIST  : return (double) listValue.size();
        case CEL   : return (double) celValue.numChildren();
        case COLOR : return (double) colorValue.getRGB();
        default    : return 0.0;
    }
}

public String getString() {
    String s = "";

    switch (type) {
        case INT   : return Integer.toString(intValue);
        case DOUBLE: return Double.toString(doubleValue);
        case STRING: return stringValue;
        case LIST  : for (int i = 0; i < listValue.size(); i++) {
                        s = s.concat( ((PDT) listValue.elementAt(i)).getString() );
                        s = s + " ";
                     }
                     return s;
        case CEL   : return celValue.toString();
        case COLOR : return colorValue.toString();
        case ERROR : return stringValue;
        default    : return null;
    }
}

public String getErrorMessage() {
    if (type == ERROR) {
        return stringValue;
    }

    return null;
}

public Vector getVector() {
    if (type != LIST) {
        return null;
    }

       return listValue;
}

public Cel getCel() {
    if (type == BABYCEL) {
    }
    else if (type == CEL)
        return celValue;
    else
        return null;

    return celValue;
}

public Color getColor() {
    if (type != COLOR) {
        return null;
    }

    return colorValue;
}

public void debugPDT() {
    System.out.println("--Debug PDT--");

    System.out.print("Type: ");
    switch (type) {
        case INT   : System.out.println("Int");    break;
        case DOUBLE: System.out.println("Double"); break;
        case STRING: System.out.println("String"); break;
        case LIST  : System.out.println("List");   break;
        case COLOR : System.out.println("Color");  break;
        case CEL   : System.out.println("Cel");    break;
    }

    System.out.print("Value: ");
    switch (type) {
        case INT   : System.out.println(getInt());    break;
        case DOUBLE: System.out.println(getDouble()); break;
        case STRING: System.out.println(getString()); break;
        case LIST  : System.out.println(getVector()); break;
        case COLOR : System.out.println(getColor());  break;
        case CEL   : System.out.println(getCel());    break;
    }
}

/* Logical */

public PDT is(PDT p) {
    boolean equal = true;

    if (type == p.getType()) {
        switch (type) {
            case INT   : if (getInt() == p.getInt())
                                return TRUE; else return FALSE;
            case DOUBLE: if (getDouble() == p.getDouble())
                                return TRUE; else return FALSE;
            case STRING: if (getString().equals(p.getString()))
                                return TRUE; else return FALSE;
            case COLOR : if (getColor().equals(p.getColor()))
                                return TRUE; else return FALSE;
            case LIST  : if (listValue.size() == p.getVector().size()) {
                                for (int i = 0; i < listValue.size(); i++) {
                                   PDT a = (PDT) listValue.elementAt(i);
                                   PDT b = (PDT) p.getVector().elementAt(i);
                                   if (a.notEqual(b) == TRUE) {
                                       return FALSE;
                                   }
                                }
                                return TRUE;
                         }
                         else {
                                return FALSE;
                         }
            case CEL   : if (this == p) return TRUE; else return FALSE;
            default    : return FALSE;
        }
    }

    if (getDouble() == p.getDouble()) {
        return TRUE;
    }

    return FALSE;
}

public PDT notEqual(PDT p) {
    if (is(p) == TRUE) {
        return FALSE;
    }

    return TRUE;
}

public PDT greater(PDT p) {
    if (type == p.getType()) {
        switch (type) {
            case INT   : if (getInt() > p.getInt())
                             return TRUE; else return FALSE;
            case DOUBLE: if (getDouble() > p.getDouble())
                             return TRUE; else return FALSE;
            case STRING: if (getString().compareTo(p.getString()) > 0)
                             return TRUE; else return FALSE;
            case COLOR : if (colorValue.getRGB() > p.getColor().getRGB())
                             return TRUE; else return FALSE;
            case LIST  : if (listValue.size() > p.getVector().size())
                             return TRUE; else return FALSE;
            case CEL   : if (celValue.numChildren() > p.getCel().numChildren())
                             return TRUE; else return FALSE;
            default    : return FALSE;
        }
    }

    if (getDouble() > p.getDouble()) {
        return TRUE;
    }

    return FALSE;
}

public PDT greaterOrEqual(PDT p) {
    boolean gt = (greater(p) == TRUE);
    boolean eq = (is(p) == TRUE);

    if (gt || eq) {
        return TRUE;
    }

    return FALSE;
}

public PDT less(PDT p) {
    if (greaterOrEqual(p) == TRUE) {
        return FALSE;
    }

    return TRUE;
}

public PDT lessOrEqual(PDT p) {
    if (greater(p) == TRUE) {
        return FALSE;
    }

    return TRUE;
}

public PDT and(PDT p) {
    if (greater(ZERO) == TRUE && p.greater(ZERO) == TRUE) {
        return TRUE;
    }

    return FALSE;
}

public PDT or(PDT p) {
    if (greater(ZERO) == TRUE || p.greater(ZERO) == TRUE) {
        return TRUE;
    }

    return FALSE;
}

public boolean isScalar() {
    if (type <= 2) {
        return true;
    }

    return false;
}

public boolean isList() {
    return (type == LIST);
}

public boolean isError() {
    return (type == ERROR);
}

/* Arithmetic */
```

```
public PDT plus(PDT p) {
    Vector v = new Vector();

    if (type == p.getType()) {
        switch (type) {
            case (INT)   : return new PDT(intValue + p.getInt());
            case (DOUBLE): return new PDT(doubleValue + p.getDouble());
            case (STRING): return new PDT(stringValue.concat(p.getString()));
            case (COLOR) : return new PDT(new Color(colorValue.getRGB() +
                               p.getColor().getRGB()));
            case (LIST)  : for (int i = 0; i < listValue.size(); i++) {
                            v.add(listValue.elementAt(i));
                           }
                           for (int i = 0; i < p.getVector().size(); i++) {
                               v.add(p.getVector().elementAt(i));
                           }
                           return new PDT(v);
            default      : break;
        }
    } else {

        // String + scalar => String
        if (isScalar() && p.isScalar()) {
            if (type == STRING || p.getType() == STRING)
                return new PDT(getString() + p.getString());
            return new PDT(getDouble() + p.getDouble());
        }
        else if (isScalar() && p.isList())
            return svAdd(this, p);
        else if (isList() && p.isScalar())
            return vsAdd(this, p);
    }

    return new PDT(ERROR, "Invalid argument: " + getTypeString() + " + " +
p.getTypeString());
}

public PDT svAdd(PDT scalar, PDT list) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add(scalar.plus( (PDT) list.getVector().elementAt(i) ));
       }

    return new PDT(v);
}
```

```
public PDT vsAdd(PDT list, PDT scalar) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add( ((PDT) list.getVector().elementAt(i)).plus(scalar) );
       }

    return new PDT(v);
}

/* Strings, cels, and lists are not valid arguments for minus. */

public PDT minus(PDT p) {
    if (type == p.getType()) {
        switch (type) {
            case (INT)   : return new PDT(intValue - p.getInt());
            case (DOUBLE): return new PDT(doubleValue - p.getDouble());
            case (COLOR) : return new PDT(new Color(colorValue.getRGB() -
                    p.getColor().getRGB()));
            default      : break;
        }
    } else {
        if (isScalar() && p.isScalar()) {
            return new PDT(getDouble() - p.getDouble());
        }
    }

    return new PDT(ERROR, "Invalid argument: " + getTypeString() + " - " +
        p.getTypeString());
}

public PDT times(PDT p) {
    if (type == p.getType()) {
        switch (type) {
            case (INT)   : return new PDT(intValue * p.getInt());
            case (DOUBLE): return new PDT(doubleValue * p.getDouble());
            default      : break;
        }
    } else {
        if (isScalar() && p.isScalar()) {
            return new PDT(getDouble() * p.getDouble());
        }

        if (isScalar() && p.isList()) {
            return svTimes(this, p);
        }

        if (isList() && p.isScalar()) {
            return vsTimes(this, p);
        }
    }

    return new PDT(ERROR, "Invalid argument: " + getTypeString() + " * " +
        p.getTypeString());
}

public PDT svTimes(PDT scalar, PDT list) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add(scalar.times( (PDT) list.getVector().elementAt(i) ));
       }

    return new PDT(v);
}

public PDT vsTimes(PDT list, PDT scalar) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add( ((PDT) list.getVector().elementAt(i)).times(scalar) );
       }

    return new PDT(v);
}

public PDT divideBy(PDT p){
    if (type == p.getType()) {
        switch (type) {
            case (INT)   : if (intValue < p.getInt() || intValue % p.getInt() != 0)
                            return new PDT(getDouble() / p.getDouble());
                           else
                               return new PDT(intValue / p.getInt());
            case (DOUBLE): return new PDT(doubleValue / p.getDouble());
            default      : break;
        }
    } else {
        if (isScalar() && p.isScalar()) {
            return new PDT(getDouble() / p.getDouble());
        }

        if (isScalar() && p.isList()) {
            return svDivideBy(this, p);
        }

        if (isList() && p.isScalar()) {
```

```
            return vsDivideBy(this, p);
        }
    }

    return new PDT(ERROR, "Invalid argument: " + getTypeString() + " / " +
        p.getTypeString());
}

public PDT svDivideBy(PDT scalar, PDT list) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add(scalar.divideBy( (PDT) list.getVector().elementAt(i) ));
       }

    return new PDT(v);
}

public PDT vsDivideBy(PDT list, PDT scalar) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add( ((PDT) list.getVector().elementAt(i)).divideBy(scalar) );
       }

    return new PDT(v);
}

public PDT mod(PDT p) {
    if (type == p.getType()) {
        switch (type) {
            case (INT)   : return new PDT(intValue % p.getInt());
            case (DOUBLE): return new PDT(doubleValue % p.getDouble());
            default      : break;
        }
    } else {
        if (isScalar() && p.isScalar()) {
            return new PDT(getDouble() % p.getDouble());
        }

        if (isScalar() && p.isList()) {
            return svMod(this, p);
        }

        if (isList() && p.isScalar()) {
            return vsMod(this, p);
        }
    }
    return new PDT(ERROR, "Invalid argument: " + getTypeString() + " mod " +
        p.getTypeString());
}

public PDT svMod(PDT scalar, PDT list) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add(scalar.mod( (PDT) list.getVector().elementAt(i) ));
       }

    return new PDT(v);
}

public PDT vsMod(PDT list, PDT scalar) {
    Vector v = new Vector();

       for (int i = 0; i < list.getVector().size(); i++) {
        v.add( ((PDT) list.getVector().elementAt(i)).mod(scalar) );
       }

    return new PDT(v);
}

/* Cel */

public PDT first() {
    if (isList()) {
        return (PDT) listValue.elementAt(0);
    }

    return this;
}

public PDT last() {
    if (isList()) {
        return (PDT) listValue.elementAt(listValue.size() - 1);
    }

    return this;
}

public PDT rest(){
    Vector v = new Vector();

    if (isList()) {
        for (int i = 1; i < listValue.size(); i++) {
            v.add(listValue.elementAt(i));
        }

        return new PDT(v);
    }

    return new PDT(ERROR, "Invalid argument: not a list.");
}

public PDT getElement(int index) {
    if (isList()) {

        if (index >= listValue.size()) {
            return new PDT(ERROR, "List index out of bounds.");
        }

        return (PDT) listValue.elementAt(index);
    }

    return new PDT(ERROR, "Invalid argument: not a list.");
}

public PDT height(){
    if (type == CEL) {
        return new PDT(celValue.getHeight());
    }

    return ZERO;
}
public PDT width(){
    if (type == CEL) {
        return new PDT(celValue.getWidth());
    }

    return ZERO;
}

public PDT size() {
    if(type == CEL)
        return new PDT(celValue.numChildren());
    else if (type == BABYCEL)
        return ZERO;
    else if (type == LIST)
        return new PDT(listValue.size());
    return new PDT();
}

// position of this cel in its parent
```

```
public PDT position() {
     Vector pos = new Vector();
    pos.add(new PDT(celValue.getRow(indexInParent)));
    pos.add(new PDT(celValue.getColumn(indexInParent)));
    return new PDT(pos);
}


public PDT neighbors() {
       Vector ne = new Vector();
    Cel parent = new Cel();
    if (type == BABYCEL)
        parent = celValue;
    else {
        parent = celValue.getParent();
        if (parent == null)
            return new PDT(ne);
    }
    int pwidth = parent.getWidth();
    int psize = parent.numChildren();
    int i = indexInParent;

    // N
    if(i > pwidth) {
        ne.add(new PDT(parent, i - pwidth));
        // NE
        if(i % pwidth < pwidth-1)
            ne.add(new PDT(parent, i - pwidth + 1));
            // Should we add a null pointer to the vector
            // if NE doesn't exist? This would keep the list
            // indices consistent (i.e. elementAt(1) would always
            // be the NE neighbor) everytime neighbors() is called

        // NW
        if(i % pwidth > 0)
            ne.add(new PDT(parent, i - pwidth - 1));
    }
    //S
    if(i < psize - pwidth){
        ne.add(new PDT(parent, i + pwidth));
        //SE
        if(i % pwidth < pwidth-1)
           ne.add(new PDT(parent, i + pwidth + 1));
        //SW
        if(i % pwidth > 0)
            ne.add(new PDT(parent, i + pwidth - 1));
    }
    //E
```

```
        if(i % pwidth < pwidth-1)
            ne.add(new PDT(parent, i + 1));

        //W
        if(i % pwidth > 0)
            ne.add(new PDT(parent, i - 1));

        return new PDT(ne);
    }

    public PDT parent() {
        if(type == CEL) {
            return new PDT(celValue.getParent());
        }

        if (type == BABYCEL) {
            return new PDT(celValue);
        }

        return new PDT();
    }

    public PDT superparent() {
        if(type == CEL){
            return new PDT(celValue.getRoot());
        }

        if (type == BABYCEL){
            return new PDT(celValue.getRoot());
        }

        return new PDT();
    }

    public PDT color() {
        if(type == CEL) {
            return new PDT(celValue.getColor());
        }

        if (type == BABYCEL) {
            return new PDT((Color) celValue.getBabyColor(indexInParent));
        }

        return new PDT();
    }

    public PDT getChild(PDT listOrIndex) {
        if(listOrIndex.getType()==LIST) {
```

```
        //a List is the tuple specifying which child to return
        if (type==CEL) {
            int row, col;
            row = ((PDT)(listOrIndex.getVector()).elementAt(0)).getInt();
            col = ((PDT)(listOrIndex.getVector()).elementAt(1)).getInt();

            if (celValue.isBaby(row, col)) {
                //returns a new PDT object which is a baby cel
                //having the parent cel in celValue, and the index for the
                //child in indexInParent
                return new PDT(celValue, celValue.getIndex(row, col));
            }
            else
                return new PDT(celValue.getChild(row, col));
        }
        else if (type == BABYCEL)
            System.err.println("the cel has no children");
    }
    else if(listOrIndex.getType()==PDT.INT) {
        if (type==CEL) {
            int i = listOrIndex.getInt();
              if (celValue.isBaby(i))
                return new PDT(celValue, i);
            else
                return new PDT(celValue.getChild(i));
        }
        else if (type==BABYCEL) {
        // fail
        }
        else if (type == LIST)
            return (PDT)(getVector().elementAt(listOrIndex.getInt()));
    }
    return new PDT( );
}

// divide into a square grid
public void divide(PDT divisions){
    if (type == BABYCEL) {
        type = CEL;
        celValue = celValue.matureBaby(indexInParent);
    }

    celValue.divide(divisions.getInt());
}

// lots of repeated code here, wonder if there's a smarter way
// to do this.
// parameters: any pair permutation of list & int
```

```
public void divide(PDT rows, PDT columns){


    if (rows.getType() == LIST && columns.getType() == LIST) {
        Vector rv = rows.getVector();
        double[] ra = new double[rv.size()];
        for (int i=0; i<ra.length; i++)
            ra[i] = ((PDT)rv.elementAt(i)).getDouble();
        Vector cv = columns.getVector();
        double[] ca = new double[cv.size()];
        for (int i=0; i<ca.length; i++)
            ca[i] = ((PDT)cv.elementAt(i)).getDouble();
        if (type == BABYCEL) {
            type = CEL;
            celValue = celValue.matureBaby(indexInParent);
        }
        celValue.divide(ra, ca);
    }
    else if (rows.getType() == INT && columns.getType() == LIST) {
        Vector cv = columns.getVector();
        double[] ca = new double[cv.size()];
        for (int i=0; i<ca.length; i++)
            ca[i] = ((PDT)cv.elementAt(i)).getDouble();
        if (type == BABYCEL) {
            type = CEL;
            celValue = celValue.matureBaby(indexInParent);
        }
        celValue.divide(rows.getInt(), ca);
    }
    else if (rows.getType() == LIST && columns.getType() == INT) {
        Vector rv = rows.getVector();
        double[] ra = new double[rv.size()];
        for (int i=0; i<ra.length; i++)
            ra[i] = ((PDT)rv.elementAt(i)).getDouble();

        if (type == BABYCEL) {
            type = CEL;
            celValue = celValue.matureBaby(indexInParent);
        }
        celValue.divide(ra, columns.getInt());
    }
    else if (rows.getType() == INT && columns.getType() == INT) {
        if (type == BABYCEL) {
            type = CEL;
            celValue = celValue.matureBaby(indexInParent);
        }
        celValue.divide(rows.getInt(), columns.getInt());
        }
    }

    public void setCel(PDT p) {
        setCel(p.getCel());
    }

     public void setCel(Cel c) {
        if (type == BABYCEL)
            celValue.children.setElementAt(c, indexInParent);
        type = CEL;
        celValue = c;
        celValue.setIndexInParent(indexInParent);
     }

    /*
    public void setChild(PDT list, PDT newChild) {
        int row = ((PDT)(listOrIndex.getVector()).elementAt(0)).getInt();
        int col = ((PDT)(listOrIndex.getVector()).elementAt(1)).getInt();
    }
    */
    public void setColor(Color c) {
        if(type == CEL){
            celValue.setColor(c);
        } else if (type == BABYCEL) {
            celValue.setBabyColor(c, indexInParent);
        }
    }

}
```

10.7 PST.java

```
/*
PST.java
The Parcel symbol table class
Mike Ilardi
*/
import java.util.*;
public class PST extends Hashtable {

    private PST currentLevel;  //the child we are currently working on
    private PST parent;  //this PST's parent.  If NULL we are the root
    //allows the symbol table to distinguish between functions and block
    private boolean isAFunction;
    //Constructor
```

```
PST() {

    super(300);

    currentLevel=this;
            isAFunction=false;
}


public void insertKey(String key, Object o) {
//first check for a more-global instance of the variable
Object ob;
PST p = currentLevel;
while ((ob = p.get(key)) == null && p.parent != null && p.isAFunction==false) {
    p = p.parent;
}

//if while recursing up the tree we hit a function
//call and had to stop, try checking the global level
if (ob==null && p.isAFunction==true){
    ob = get(key);
    p = this;
}


if (ob!=null)
    p.remove(key); //remove it if it exists anywhere
 else
    p=currentLevel; //if not found, go to local scope

    p.put(key,o); //insert the key and object
}



public Object getKey(String key) {
    PST p = currentLevel;
    Object o;
    while (  (o = p.get(key)) == null
             && p.parent != null
             && p.isAFunction==false) {
        p = p.parent;
    }
            //if while recursing up the tree we hit a function
            //call and had to stop, try checking the global level
            if (o==null && p.isAFunction==true){
                o = get(key);
            }



    return o;
}

  public void makeChildFunction() {
    PST t = currentLevel;
    currentLevel = new PST();
    currentLevel.parent = t;
            currentLevel.isAFunction=true;
}

public void makeChild() {
    PST t = currentLevel;
    currentLevel = new PST();
    currentLevel.parent = t;
}

public void removeChild() {
    currentLevel=currentLevel.parent;
}

public void debug(){
    try{
        System.out.println("--------------------");
        System.out.println("Symbol Table Entries");
        System.out.println("--------------------");
        String s;
        PST p = currentLevel;

        do {
            System.out.println("Level:");
            for(Enumeration e = p.keys(); e.hasMoreElements() ;){
                s = (String) e.nextElement();
                System.out.println(s + " = " + ((PDT) p.getKey(s)).getString());
            }
        } while ((p = p.parent) != null);
        System.out.println("--------------------");
    } catch (Exception e){
        // this throws an error and i cant figure out why
        //System.out.println("Caught in symbol table: " + e);
    }

}

public static void main( String args[] ) throws java.io.IOException {
```

```
PST symbol = new PST();

symbol.insertKey("x", new Integer(5));
System.out.println("table after inserting 5");
System.out.println(symbol.getKey("x"));
symbol.makeChild();
symbol.insertKey("x", new Integer(6));
System.out.println("table after making child and inserting 6");
System.out.println(symbol.getKey("x"));
symbol.removeChild();
System.out.println("table after removing child");
System.out.println(symbol.getKey("x"));


/*
System.out.println("Inserting x=4,y=5,z=7 at root");
symbol.insertKey("x",new Integer(4));
symbol.insertKey("y", new Integer(5));
symbol.insertKey("z", new Integer(7));
System.out.println("testing root symbol table");
System.out.println("x="+ symbol.getKey("x"));
System.out.println("y="+ symbol.getKey("y"));

System.out.println("----------making child");
symbol.makeChild();
System.out.println("x="+symbol.getKey("x"));
System.out.println("inserting d=10 at this level");
symbol.insertKey("d", new Integer(10));

System.out.println("------------making the child have a child");
symbol.makeChild();
System.out.println("x="+symbol.getKey("x"));
System.out.println("d="+symbol.getKey("d"));
System.out.println("Inserting h=30");
symbol.insertKey("h",new Integer(30));
System.out.println("Inserting d=40");
symbol.insertKey("d",new Integer(40));
System.out.println("d="+symbol.getKey("d"));


System.out.println("h="+symbol.getKey("h"));
System.out.println("z="+symbol.getKey("z"));

System.out.println("removing child--------------------");
symbol.removeChild();
System.out.println("h="+symbol.getKey("h"));
System.out.println("d="+symbol.getKey("d"));
```

```
        System.out.println("------------making the child have a child");
        symbol.makeChild();
        System.out.println("------------making the child have a child");
        symbol.makeChild();
        System.out.println("------------making the child have a child");
        symbol.makeChild();

        System.out.println("d="+symbol.getKey("d"));

        symbol.removeChild();
        symbol.removeChild();
        symbol.removeChild();
        symbol.removeChild();
        System.out.println("x="+ symbol.getKey("x"));
        System.out.println("y="+ symbol.getKey("y"));
        System.out.println("z="+ symbol.getKey("z"));
        */

    }
}
```

10.8 ParcelFunctionLibrary.java

```
/*
ParcelFunctionLibrary.java
Various library functions for use in Parcel programs
Mike Ilardi
*/
import java.awt.*;
import java.util.*;
import java.lang.*;
public class ParcelFunctionLibrary extends Hashtable{
    //http://java.sun.com/j2se/1.4.2/docs/api/index.html

    PDT pcopy;
    private String cname;
    private Random rndnum;
    public ParcelFunctionLibrary(){
        super(300);
        put("abs", new Integer(0));
        put("acos",new Integer(1));
        put("asin",new Integer(2));
        put("atan",new Integer(3));
        put("ceil",new Integer(4));
        put("cos",new Integer(5));
        put("floor",new Integer(6));
        put("log",new Integer(7));
        put("sin",new Integer(8));
        put("sqrt",new Integer(9));
        put("pow",new Integer(10));
        put("tan",new Integer(11));   //lib functions >11 cannot be
        put("Color",new Integer(12)); //applied recursively to lists.
        put("max",new Integer(13));   //they may, however, accept a list
        put("min", new Integer(14));  //as a parameter
        put("random",new Integer(15));
        put("randomItem",new Integer(16));
        //instantiate our random number generator and seed with time
        rndnum = new Random();

    }

public PDT callFunction(String name, PDT params){
     pcopy=new PDT (params.getVector());
     cname=name;
     //find integer associated with given library function using hash table
     Integer n = (Integer)get(name);
     if (n==null)return null;       //library function does not exist.  return null
     //main switch statement.  Can call itself recursively to operate on lists
     return callFunction2(n.intValue(),pcopy);
    }

    //code for all library functions.  calls itself recursively on lists
    private PDT callFunction2(int val, PDT params)
    {
        int i;

        if (params.getType()==PDT.LIST && params.getInt()==0) //create dummy params
        {                                                      // if none exist
            Vector tvector = new Vector();                    //  to avoid crashing
            tvector.add(new PDT("dummy"));                   //   the code below
            params.setVector(tvector);
        }


        Vector v = new Vector( params.getVector());

        //the following code recursively explores lists if we
        //have passed a list and are running a lib function on it
        //that makes sense to apply to every element of a list...

        //meaning, lib functions 0-11
        if (((PDT)v.elementAt(0)).getType()==PDT.LIST && val<12)
        {
            Vector v2 = new Vector(((PDT)v.elementAt(0)).getVector());
            //Apply function recursively to every element in list
            for (i=0;i<v2.size();i++)
            {
                Vector nv = new Vector();
                nv.add(v2.elementAt(i));
                v2.setElementAt(callFunction2(val,new PDT(nv)),i);
            }
            return new PDT(v2);
        }
        //done

        //we’re either not passing a list or we’re running a function that
        //specifically requires a list


        //below code does type-checking for first 13 function calls.
        //They all require either a double
        //or an integer
          if (val<12 &&  (((PDT)v.elementAt(0)).getType()!=PDT.DOUBLE  &&
              ((PDT)v.elementAt(0)).getType()!=PDT.INT ))
            {
                return new PDT(PDT.ERROR, "Invalid Arguments of type "+
                    ((PDT)v.elementAt(0)).getTypeString()  +" passed to function " +
                    cname + ".  Expecting a double or an integer");

            }

    switch(val)
        {

        case 0: //ABS
            if (((PDT)v.elementAt(0)).getType()==PDT.DOUBLE) //TYPE CHECKING
        return new PDT(Math.abs(((PDT) v.elementAt(0)).getDouble()));
            if (((PDT)v.elementAt(0)).getType()==PDT.INT) //TYPE CHECKING
                return new PDT(Math.abs(((PDT) v.elementAt(0)).getInt()));
        case 1: //ACOS
            if (((PDT)v.elementAt(0)).getType()==PDT.DOUBLE ||
                ((PDT)v.elementAt(0)).getType()==PDT.INT ) //TYPE CHECKING
        return new PDT(Math.acos(((PDT) v.elementAt(0)).getDouble()));
    case 2: //ASIN
        return new PDT(Math.asin(((PDT) v.elementAt(0)).getDouble()));
    case 3: //ATAN
        return new PDT(Math.atan(((PDT) v.elementAt(0)).getDouble()));
    case 4: //CEIL
        return new PDT((int)Math.ceil(((PDT) v.elementAt(0)).getDouble()));
    case 5: //COS
        return new PDT(Math.cos(((PDT) v.elementAt(0)).getDouble()));
    case 6: //FLOOR
```

```
    return new PDT(((PDT)(v.elementAt(0))).getInt());
case 7: //LOG
    return new PDT(Math.log(((PDT) v.elementAt(0)).getDouble()));
    case 8: //SIN
    return new PDT(Math.sin(((PDT) v.elementAt(0)).getDouble()));
case 9: //SQRT
    return new PDT(Math.sqrt(((PDT) v.elementAt(0)).getDouble()));
    case 10://POW
    return new PDT(Math.pow( ((PDT) v.elementAt(0)).getDouble(),
        ((PDT) v.elementAt(1)).getDouble()));
case 11://TAN
    return new PDT(Math.tan(((PDT) v.elementAt(0)).getDouble()));
    case 12://COLOR
        if (v.size()!=3) return new PDT(PDT.ERROR,
            "Function Color[] requires 3 arguments.");
        if((((PDT)v.elementAt(0)).getType()!=PDT.INT &&
            ((PDT)v.elementAt(0)).getType()!=PDT.DOUBLE) ||
            (((PDT)v.elementAt(1)).getType()!=PDT.INT &&
            ((PDT)v.elementAt(1)).getType()!=PDT.DOUBLE) ||
            (((PDT)v.elementAt(2)).getType()!=PDT.INT &&
            ((PDT)v.elementAt(2)).getType()!=PDT.DOUBLE))
            return new PDT(PDT.ERROR,
                "Invalid Arguments passed to function Color[].");

    return new PDT(new Color( ((PDT)v.elementAt(0)).getInt(),
        ((PDT)v.elementAt(1)).getInt(), ((PDT)v.elementAt(2)).getInt() ));
case 13://MAX
         if (((PDT)v.elementAt(0)).getType()==PDT.LIST) //If we pass it a list
                {     //scan list. find max element
                    Vector v2 = ((PDT)v.elementAt(0)).getVector();
                    int t=0; //type of max value
                    double m=Double.MIN_VALUE; //current max
                    t=PDT.ERROR;
                    for (i=0;i<v2.size();i++)
                    {
                        if (((PDT)v2.elementAt(i)).getDouble()>=m)
                        {
                            //store current max
                            m = ((PDT)v2.elementAt(i)).getDouble();
                            //store current max's type
                            t= ((PDT)v2.elementAt(i)).getType();
                        }
                    }
                    //return double if max is double
                    if (t==PDT.DOUBLE) return new PDT(m);
                    //return int if max is an int
                    if (t==PDT.INT) return new PDT((int)m);
                    return new PDT(PDT.ERROR,
                        "Invalid Arguments. A list passed to function max[]
                        must contain numerical data.");
                }
         //Deal with non-list cases
        if (((PDT)v.elementAt(0)).getType()==PDT.DOUBLE ||
            ((PDT)v.elementAt(1)).getType()==PDT.DOUBLE )//Type check
        {
            return new PDT(Math.max( ((PDT) v.elementAt(0)).getDouble(),
                ((PDT) v.elementAt(1)).getDouble() ));
        }
        if (((PDT)v.elementAt(0)).getType()==PDT.INT &&
            ((PDT)v.elementAt(1)).getType()==PDT.INT )
        {
            return new PDT(((int)Math.max( ((PDT) v.elementAt(0)).getDouble(),
                ((PDT) v.elementAt(1)).getDouble())));
        }

        return new PDT(PDT.ERROR,
            "Invalid Arguments.  Function max[] accepts either a list
            or a pair of arguments.");

    case 14://MIN
        if (((PDT)v.elementAt(0)).getType()==PDT.LIST) //If we pass it a list
        {     //scan list. find min element
            Vector v2 = ((PDT)v.elementAt(0)).getVector();
            int t=0; //type of min value
            double m=Double.MAX_VALUE; //current min
            for (i=0;i<v2.size();i++)
            {
                if (((PDT)v2.elementAt(i)).getDouble()<=m)
                {
                    m = ((PDT)v2.elementAt(i)).getDouble(); //store current max
                    t= ((PDT)v2.elementAt(i)).getType(); //store current max's type
                }
            }
            if (t==PDT.DOUBLE) return new PDT(m); //return double if min is double

            if (t==PDT.INT)  return new PDT((int)m); //return int if min is an int
            return new PDT(PDT.ERROR,
                "Invalid Arguments.  Function min[] accepts either a list or
                a pair of arguments.");
        }
        //Deal with non-list cases
        if (((PDT)v.elementAt(0)).getType()==PDT.DOUBLE ||
            ((PDT)v.elementAt(1)).getType()==PDT.DOUBLE )//Type check
        {
            return new PDT(Math.min( ((PDT) v.elementAt(0)).getDouble(),
                ((PDT) v.elementAt(1)).getDouble() ));
            }
            if (((PDT)v.elementAt(0)).getType()==PDT.INT &&
                ((PDT)v.elementAt(1)).getType()==PDT.INT )
            {
                return new PDT(((int)Math.min( ((PDT) v.elementAt(0)).getDouble(),
                    ((PDT) v.elementAt(1)).getDouble())));
            }
            return new PDT(PDT.ERROR,
                "Invalid Arguments.  Function min[] accepts either a list or
                a pair of arguments.");
    case 15://RANDOM
            double d = rndnum.nextDouble();
        return new PDT(rndnum.nextDouble());
    case 16://RANDOMITEM
        return (PDT)(((Vector)
            ((PDT)v.elementAt(0)).getVector()).elementAt(
                rndnum.nextInt(((Vector)((PDT)v.elementAt(0)).getVector()).size())));

    default: break;

    }

    return new PDT();
    }

}
```

10.9 ParcelInterpreter.java

```
/*
ParcelInterpreter
Handles calls from the walker
Blake Shaw
*/
import antlr.*;
import antlr.debug.misc.*;
import antlr.CommonAST;
import antlr.collections.AST;
import java.util.*;
import java.awt.Color;

public class ParcelInterpreter {
    public PST st;
    public ParcelLibrary pl;
    public ParcelFunctionLibrary pfl;
    public ParcelInterpreter(){
        st = new PST();
        pl = new ParcelLibrary();
        pfl = new ParcelFunctionLibrary();
        initializeST();
    }


    public void check(PDT s, String errorMessage){
        if(s.getType() == PDT.ERROR) {
            // print msg
                System.out.print("Error (" + errorMessage + "): ");
                System.out.println(s.getString());
            // quit
                System.exit(1);
        }
    }

    public void initializeST(){
        st.insertKey("black", new PDT(Color.BLACK));
        st.insertKey("blue", new PDT(Color.BLUE));
        st.insertKey("cyan", new PDT(Color.CYAN));
        st.insertKey("darkGray", new PDT(Color.DARK_GRAY));
        st.insertKey("gray", new PDT(Color.GRAY));
        st.insertKey("green", new PDT(Color.GREEN));
        st.insertKey("lightGray", new PDT(Color.LIGHT_GRAY));
        st.insertKey("magenta", new PDT(Color.MAGENTA));
        st.insertKey("orange", new PDT(Color.ORANGE));
        st.insertKey("pink", new PDT(Color.PINK));
        st.insertKey("red", new PDT(Color.RED));
        st.insertKey("white", new PDT(Color.WHITE));
        st.insertKey("yellow", new PDT(Color.YELLOW));
    }
    public PDT defineNewFunction(String name, Vector paramList, AST functionBody){
        Vector v = new Vector();
        v.add(paramList);
        v.add(functionBody);
        st.insertKey(name, v);
        return (new PDT());
    }

    public PDT function_call(String name, PDT theParams){
        PDT x = new PDT();

        if ((x = pfl.callFunction(name, theParams)) != null)
            return x;

        Vector functionInfo = (Vector) st.getKey(name);
        AST functionBody = (AST) functionInfo.elementAt(1);
        Vector paramNames = (Vector) functionInfo.elementAt(0);

        st.makeChildFunction();
        for(int i = 0; i<(theParams.getVector()).size(); i++){
            st.insertKey( (String) paramNames.elementAt(i),
                          (PDT)(theParams.getVector()).elementAt(i) );
        }

        ParcelWalker walker = new ParcelWalker(this);
        try {
            x = (PDT) walker.expr(functionBody);
        } catch (Exception e){
            System.out.println("Caught: " + e);
        }

        st.removeChild();
        return x;
    }

    /* Control */
    public boolean evaluateLogical(PDT s){
    // evaluates the logic in an if statement
        if(s.getInt() == 1){
            return true;
        }
    return false;
    }

    public PDT returnStatement(PDT r){
    // known issue:
    // limited to a single return statement
    // at the end of the block
    return r;
    }

    public PDT load(PDT type, PDT filename, PDT var){
        if((type.getString()).equals("image")){
            if(var.getType() != PDT.CEL)
                check(new PDT(PDT.ERROR, "Attempted to copy image into non-cel"),
                      "Load:");

            pl.load(filename.getString(), var);
        } else if((type.getString()).equals("data")){
            if(var.getType() != PDT.LIST)
                check(new PDT(PDT.ERROR, "Attempted to copy values into non-list"),
                      "Load:");
            pl.load(filename.getString(), var);
        }
    return (new PDT());
    }

    public PDT divide(PDT target, PDT direction, PDT relation){
        if (target.getType() != PDT.CEL && target.getType() != PDT.BABYCEL)
            return new PDT(PDT.ERROR, "Attempt to divide a non-cel");

        Vector v = new Vector();
        if (direction.getString().equals("rows"))
            target.divide(relation, new PDT(1));
        else if (direction.getString().equals("columns"))
            target.divide(new PDT(1), relation);
        else {
            if (relation.getType() == PDT.LIST) {
                // divide root by [[1, 2, 4], [1, 2, 4]]
                // divide root by [4, 5]
                v = relation.getVector();
                target.divide((PDT) v.elementAt(0), (PDT) v.elementAt(1));
            }
            else if(relation.getType() == PDT.INT)
                target.divide(relation);
        }
        return new PDT();
    }


    public boolean forEachLoopContinue(PDT iterated, int counter){
    // iterated is a list of information about which ids and expressions are used
    // in the for loop
    // the iterated list should always be of even length
    // iterated should have pairs of IDs and Expressions
    // eg. foreach i in aCel and j in bCel --> (i, aCel, j, bCel)
        Vector v = iterated.getVector();
        PDT x = new PDT();
        String var = "";
        for(int i=0; i< v.size(); i+=2){
            var = ((PDT) v.elementAt(i)).getString();
            x = (PDT) v.elementAt(i+1);
            if(x.getType() == PDT.LIST){
                if(counter >= x.getVector().size()){
                    return false;
                }
                x = (PDT) (x.getVector()).elementAt(counter);
                assign(new PDT(var), x);
            } else if(x.getType() == PDT.CEL){
                if(counter >= (x.size()).getInt()){
                    return false;
                }
                x = x.getChild(new PDT(counter));
                assign(new PDT(var), x);

            } else {


            }
        }
    return true;
    }

    public void initializeForLoop(String iterator, PDT iterationInfo){
        startBlock();
        int startValue = ((PDT) (iterationInfo.getVector()).elementAt(0)).getInt();
        st.insertKey(iterator, new PDT(startValue));
    }

    public void startBlock(){
        st.makeChild();
    }

    public void endBlock(){
        st.removeChild();
    }


    public boolean forLoopContinue(String iterator, PDT iterationInfo){
        //iterator is the name of the variable which is being used to iterate
        //iterationInfo returns information about the for loop in a list
        // should either be of length 2 or 3, 3 if the "by" option is specified
        // eg. for i from 1 to 5 by 2 --> (1, 5, 2)

        int endValue = ((PDT) (iterationInfo.getVector()).elementAt(1)).getInt();
        int i = ((PDT)st.getKey(iterator)).getInt();

        if (iterationInfo.getVector().size()==2) {
            if(i+1 <= endValue){
                i++;
                st.insertKey(iterator, new PDT(i));
                return true;
            }
        } else if (iterationInfo.getVector().size()==3) {
            int byNum = ((PDT) (iterationInfo.getVector()).elementAt(2)).getInt();
            if(i+byNum <= endValue){
                i += byNum;
                st.insertKey(iterator, new PDT(i));
                return true;
            }
        }
        return false;
    }

    public PDT print(PDT s){
        System.out.println(s.getString());
        return (new PDT());
    }

    public PDT assign(PDT name, PDT setValue){
        st.insertKey(name.getString(), setValue);
        return (new PDT());
    }

    public PDT draw(PDT node, PDT output, PDT res, PDT colorMapOptions){
        if((output.getString()).equals("screen")){
            pl.display(node.getCel(), ((PDT)(res.getVector()).elementAt(0)).getInt(),
                       ((PDT)(res.getVector()).elementAt(1)).getInt());
        } else {
            pl.display(node.getCel(), ((PDT)(res.getVector()).elementAt(0)).getInt(),
                       ((PDT)(res.getVector()).elementAt(1)).getInt(),
                       (output.getString()));
        }
        return (new PDT());
    }

    public PDT getID(String idname){
           return new PDT(idname);
    }

    public PDT getValue(String idname) {
        //Look up symbol table and return the correct data object
        //Right now just spits back a null data object


        PDT id = (PDT)st.getKey(idname);
        if (id == null){
            return new PDT(PDT.ERROR, "Value for " + idname + " is null");
        }
        return id;
    }

    public PDT parseNumber(String s){
        //parse number and return the correct data object
        //Right now just spits back an int
        if (s.indexOf('.') != -1)
```

```
            return (new PDT(Double.parseDouble(s)));

        return (new PDT(Integer.parseInt(s)));
    }

    public PDT parseString(String s){
        return (new PDT(s.substring(1, s.length()-1)));
    }

}
```

10.10 ParcelLibrary.java

```
/*
ParcelLibrary.java
I/O functions for Parcel
Lawrence Wang
*/
import java.util.*;
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.BufferedImage;
import javax.swing.*;
import javax.imageio.ImageIO;
import java.io.*;

public class ParcelLibrary {

    private JFrame frame;
    private DrawPanel panel;
    private boolean hasDrawn;

    public ParcelLibrary() {
        frame = new JFrame();
        panel = new DrawPanel();
        frame.getContentPane().add(panel);
        frame.addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                System.exit(0);
            }
        });
        hasDrawn = false;
        frame.show();
    }

    // loads: textfile as list(s), image as cel
    // any one-token rows will be inserted as bare values, not single-item lists.
    // for type, use PDT.CEL or PDT.LIST
    public void load(String filename, PDT target) {
        try {
            if (target.getType() == PDT.LIST) {
                BufferedReader br = new BufferedReader(new FileReader(filename));
                if (br == null) {
                    System.out.println("Reading from file failed.");
                    System.exit(1);
                }
                Vector list = new Vector();
                String next;
                while ((next = br.readLine()) != null) {
                    String[] tokens = next.split("\t");
                    if (tokens.length == 1)
                        list.add( new PDT(Double.parseDouble(tokens[0])) );
                    else {
                        Vector sublist = new Vector();
                        for (int i=0; i<tokens.length; i++)
                            sublist.add( new PDT(Double.parseDouble(tokens[i])) );
                        list.add(sublist);
                    }
                }
                target.setVector(list);
            }
            else if (target.getType() == PDT.CEL || target.getType() == PDT.BABYCEL) {
                File f = new File(filename);
                // ImageIO supports gif, jpg, png & detects image type automatically
                BufferedImage bi = ImageIO.read(f);
                if (bi == null) {
                    System.out.println("Reading from file failed.");
                    System.exit(1);
                }
                Cel root = new Cel();
                root.divide( bi.getHeight(), bi.getWidth() );
                for (int y=0; y<bi.getHeight(); y++)
                    for (int x=0; x<bi.getWidth(); x++)
                        root.setBabyColor(new Color(bi.getRGB(x,y)), y, x);
                target.setCel(root);
            }
        } catch (IOException e) {
            System.out.println("Loading Error: File not found");
            System.exit(1);
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
```

```
    public void display(Cel root, int width, int height, String filename) {
        hasDrawn = true;

        BufferedImage img = new BufferedImage(width, height,
                                              BufferedImage.TYPE_INT_ARGB);
        Graphics2D g = img.createGraphics();
        drawCel(root, g, 0, 0, width, height);
        try {
            ImageIO.write(img, "png", new File(filename));
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void display(Cel root, int width, int height) {
        hasDrawn = true;

        //BufferedImage img = new BufferedImage(width, height,
                                              BufferedImage.TYPE_INT_ARGB);
        //Graphics2D g = img.createGraphics();
        //drawCel(root, g, 0, 0, width, height);

        panel.setPreferredSize(new Dimension(width, height));
        frame.pack();

        // refresh but no progress indication
        panel.setup(root, width, height);
        panel.repaint();

        // progressive draw but no refresh
//        Graphics2D pg = (Graphics2D)panel.getGraphics();
//        drawCel(root, pg, 0, 0, width, height);

        //pg.drawImage(img, new AffineTransform(), frame);
    }

    // recursive function for drawing cels
    private void drawCel(Cel cel, Graphics2D g, double x, double y,
                         double width, double height) {
        // don't draw cels that are too small to be visible
        if (width < 0.3 || height < 0.3)
            return;

        if (cel.numChildren() > 0) {
            for (int r=0; r<cel.getHeight(); r++)
                for (int c=0; c<cel.getWidth(); c++) {
```

```
                if (cel.isBaby(r, c)) {
                    g.setColor(cel.getBabyColor(r, c));
                    g.fill(new Rectangle2D.Double(cel.getXPosition(c)*width + x,
                                                  cel.getYPosition(r)*height + y,
                                                  cel.getXProportion(c)*width,
                                                  cel.getYProportion(r)*height));
                }
                else {
                    drawCel(cel.getChild(r,c), g, cel.getXPosition(c)*width + x,
                                                  cel.getYPosition(r)*height + y,
                                                  cel.getXProportion(c)*width,
                                                  cel.getYProportion(r)*height);
                }
            }
    }
    // the following elseblock will only ever occur if displaying an empty root
    else {
        g.setColor(cel.getColor());
        g.fill(new Rectangle2D.Double(x, y, width, height));
    }
}

/* quick test */
public static void main(String[] args) {
       Color[] colors = {Color.BLUE, Color.WHITE, Color.GREEN,
                         Color.CYAN, Color.RED, Color.YELLOW,
                         Color.BLACK, Color.ORANGE};
       ParcelLibrary p = new ParcelLibrary();

     PDT list = new PDT(new Vector());
     p.load("data.txt", list);
     for (Enumeration e = list.getVector().elements(); e.hasMoreElements();)
         System.out.println(e.nextElement());


    int size = 4;
    PDT lena = new PDT(new Cel());
    p.load("lena.png", lena);
    PDT a = new PDT(new Cel());

    Vector v = new Vector();
    v.add(new PDT(1));
    v.add(new PDT(4));
    v.add(new PDT(6));
    v.add(new PDT(4));
    v.add(new PDT(1));
    a.divide(new PDT(v), new PDT(v));
    for(int i=0; i<a.getCel().numChildren(); i++) {
        if (Math.random()<0.2) {
            a.getCel().setBabyColor(new Color((int)(255*Math.random()),
                                              (int)(255*Math.random()),
                                              (int)(255*Math.random())), i);
        }
        else if (Math.random()<0.3) {
            a.getCel().setChild(lena.getCel(), i);
        }
        else {
            a.getChild(new PDT(i)).divide(new PDT((int)(Math.random()*20)));
            for (int j=0; j<a.getChild(new PDT(i)).getCel().numChildren(); j++)
                 a.getChild(new PDT(i)).getCel().setBabyColor(
                     new Color( (int)(255*Math.random()),
                     (int)(255*Math.random()), (int)(255*Math.random())), j);
        }
    }

    p.display(a.getCel(), 600, 600);
    p.display(a.getCel(), 600, 600, "hmm.png");
}

// override JPanel to draw.
// right now the cel is drawn from scratch
// every time, which is slow and wasteful
public class DrawPanel extends JPanel {
    public int width, height;
    public Cel cel = null;
    private Graphics2D g2 = (Graphics2D)this.getGraphics();

    public void setup(Cel c, int w, int h) {
        width = w;
        height = h;
        cel = c;
    }

    public void draw() {
        Graphics2D g2 = (Graphics2D)this.getGraphics();
        drawCel(cel, g2, 0, 0, width, height);
        g2.dispose();
    }

    public void paint(Graphics g) {
        //System.out.println("Paint called");
        if (cel != null)
            draw();
    }
}
```

```
}
```

10.11 pcl.java

```
import antlr.*;
import antlr.debug.misc.*;
import java.io.*;
public class pcl {
    public static void main(String[] args) throws Exception {

        boolean debugMode = false;
        String theFile = "";
        if(args.length == 1){
            theFile = args[0];
        } else if((args.length == 2) && (args[0].equals("-debug"))){
            debugMode = true;
            theFile = args[1];

        } else {
            System.out.println("Usage: java test nameOfFile");
            System.exit(1);
        }


        try{
            FileReader fr = new FileReader(theFile);
            ParcelLexer lexer = new ParcelLexer(fr);
            ParcelParser parser = new ParcelParser(lexer);

        try{
            parser.program();
        } catch (RecognitionException r){
            System.out.println("Parse Error: " + r);
            System.exit(1);
        }

        CommonAST tree = (CommonAST)parser.getAST();

        if(debugMode){

            System.out.println( "-----------Debug---------------" );
            System.out.println( tree.toStringList() );
            ASTFrame frame = new ASTFrame(args[1], tree);
            frame.setVisible(true);
            System.out.println( "-------------------------------" );
        }
        ParcelInterpreter interp = new ParcelInterpreter();
        ParcelWalker walker = new ParcelWalker(interp);
        PDT s = walker.expr(tree);
        } catch (ArrayIndexOutOfBoundsException e){
            System.out.println("Error: Index of list/cel out of bounds");
            //e.printStackTrace();
        } catch (IOException e){
            System.out.println("Error: File Not Found");
            //e.printStackTrace();
        } catch (Exception e){
            System.out.println("Generic Java Error: " + e);
            e.printStackTrace();
        }
    }
}
```

10.12 Testing Code

```
~{
defTest.parcel: function definition/function call testing
Min Chae
}~


print "define_statement / func_call test"

define fn1[]:
    print "function1".
fn1[]


define fn2[i, j] :
    print i+" "+j.
fn2[1, 2]

print ""

define fn3[i, j] :
    print i+"+"+j
    return i+j.

print fn3[1, 2]

set z to fn3[1, 2]
print z
```

```
set root to new cel
define fn4[c]:
    return size of c.
print fn4[root]

print ""

define fn5[i, j, c]:
    fn1[]
    set k to fn3[fn3[i, j], fn3[i, j]].
fn5[11, 22, 33]


print ""

set i to 10
define fn[x]:
    set x to 7
    print i.
print i
fn[i]
print i


set i to 10
define fn[i]:
    set i to 5
    print i.
print i
fn[i]
print i


define fact[i]:
    if i is 1:
        set z to (1).
    else:
        set z to (i*fact[i-1]).
    return z.
print fact[4]
~{
doTest.parcel: do-statement testing(draw, print, load, etc.)
Min Chae
}~
```

```
print "do_statement test"


print 10 > 12
print 10 < 12
print 10 is 12
print 10 is greater than 12 or 10 is less than 12
print 10>11 or 10>12 or 10>9

print ""

set root to new cel
set color of root to red
--draw root to screen as [600, 600]

print height of root
print width of root
print size of root

print ""

set i to 5
divide root by i
set child1 to child [0, 1] of root
set color of child1 to blue
--draw root to screen as [500, 500]


print position of root+" "+position of child1
print height of root+" "+height of child1
print width of root+" "+width of child1
print size of root+" "+size of child1

print ""

divide child1 into rows by [1, 2, 3, 4, 5]
set child2 to child [0, 0] of child1
set color of child2 to green

set nb to neighbors of child [2, 2] of root
foreach n1 in nb:
    set color of n1 to black.
set color of child 1 of nb to white
--draw root to screen as [500, 500]
--draw child1 to screen as [500, 500]
--draw child2 to screen as [500, 500]
--draw child3 to screen as [500, 500]
```

```
print position of root+" "+position of child1+" "+position of child2
print height of root+" "+height of child1+" "+height of child2
print width of root+" "+width of child1+" "+width of child2
print size of root+" "+size of child1+" "+size of child2

draw root to "drawtest1.jpg" as [300, 300]

print ""

set root2 to new cel
set color of root2 to pink
divide root2 by 3
set child11 to child [0, 1] of root2
set color of child11 to yellow
divide child11 into columns by [5, 4, 3, 2, 1]

set child22 to child [0, 4] of child11
set color of child22 to green

draw root2 to screen as [500, 500]
--draw child11 to screen as [500, 500]
--draw child22 to screen as [500, 500]

print position of root2+" "+position of child11+" "+position of child22
print height of root2+" "+height of child11+" "+height of child22
print width of root2+" "+width of child11+" "+width of child22
print size of root2+" "+size of child11+" "+size of child22

--draw root2 to "drawtest2.jpg" as [400, 400]

print ""

set root3 to new cel
set color of root3 to red
divide root3 by [[1, 2, 3, 4], [4, 3, 2, 1]]
set color of child [3, 1] of root3 to blue
set color of child [1, 3] of root3 to yellow
--draw root3 to screen as [500, 500]
set z to new cel
load image "drawtest1.jpg" into z
set child [2, 2] of root3 to z

set z2 to new cel
load image "drawtest2.jpg" into z2
set child [2, 1] of root3 to z2

draw root3 to screen as [500, 500]
draw root3 to "drawtest3.jpg" as [500, 500]~{
```

```
forTest.parcel: for-statement testing (for/foreach)
Min Chae
}~


print "for_statement test"


for i from 0 to 5:
    print i.

print ""

set l to 10
for i from 5 to l by 2:
    print i.


print ""


for i from 0 to 2:
    for j from 0 to 3:
        print "i="+i
        print "j="+j..


print ""

for i from 2 to 9 by 4:
    for j from 2 to 4:
        for k from 3 to 9 by 3:
            print i+","+j+","+k...


print ""

for i from 5 to 7:
    for j from 10 to 15 by 2:
        for k from 20 to 27 by 3:
            print i+" "+j+" "+k...


print ""
print "for_each_statement test"


foreach id in [1, 2, 3, 4]:
    print id.


set list1 to [3, 4, 5, 6, 7]
foreach id2 in [1, 2, 3] and id3 in list1:
    print id2
    print id3.


set list2 to ["a", "b", "c", "d", "e"]
foreach id in list1 and id2 in list2 and id3 in ["f", 10, "g", 11, "h"]:
    print id+","+id2+","+id3.
~{
ifTest.parcel: if-statement testing
Min Chae
}~

print "if_statement test"

set i to 10


if i is 10:
    print "true".

if i != 10:
    print "true".
else:
    print "false".

if i is 10 and i is 9:
    print "true".
else:
    print "false".


if i != 10 or i != 9:
    print "true".
else:
    print "false".


if i is 10 and i = 10:
    print "true2".
else:
    print "false2".

print ""
```

```
   set i to 10
   if i = 10:
        print "true".
   else: print "false".

if i is 9 and i = 9:
    print "true".
else:
    print "false".


if i != 9:
    print "true".
else:
    print "false".


if i > 9 and i is greater than 9:
    print "true".
else:
    print "false".

print ""

if i >= 9 and i >= 10:
    print "true".
else:
    print "false".


if i < 9 and i is less than 9:
    print "true".
else:
    print "false".


if i <= 9 and i <= 8:
    print "true".
else :
    print "false".

if i is less than 11 and i is less than 12:
    print "true".
else:
    print "false".
print ""

if i is 10:
    if i is less than 10:
        print "case1".
    else:
        print "case2"..


if i is 9:
    if i is less than 10:
        print "case1".
    else:
        print "case2"..
else:
    print "case3".

if i<8:
    print "case1".
else:
    if i is less than 9:
        print "case2".
    else:
        if i < 10:
            print "case3".
        else:
            if i is less than 11:
                print "case4".
            else:
                print "case5"....


print ""


print "i="+i
if i is less than 11 and i is less than 12 and i is greater than 9:
    print "true".
else:
    print "false".




if i is less than 9 or i is less than 10 or i is less than 11:
    print "true".
else:
    print "false".
```

```
if i  < 11 and i  <= 12 and i  < 9:
    print "true".
else:
    print "false".
~{
setTest.parcel: set-statement testing
Min Chae
}~

~{comment testing
}~

print "set_statement test"


set a to 11
print a


set a2 to -11
print a2


set b to 2.34123
print b


set b2 to -2.34123
print b2


set c to "hello parcel 123"
print c


set c to "hello"+"parcel"+123
print c


set d to 11-2.34123-12+42.56
set d2 to a-b-12+42.56
print 11-2.34123-12+42.56
print d
print d2
```

```
set d to 11-(2.34123-12)+42.56
set d2 to a-(b-12)+42.56
print 11-(2.34123-12)+42.56
print d
print d2


print 12/4
print 11/4
print 12.0/4.0

print " "

set d to 11/2.34123
set d2 to a/b
print 11/2.34123
print d
print d2

print " "

set d to 11*2.34123
set d2 to a*b
print 11*2.34123
print d
print d2

print " "

set d to (10/2.34123)*2.34123
print d
set d2 to 2.34123/(10*2.34123)
print d2


set e to 11 mod 3
set e2 to 11 % 3
print 11 % 3
print 11 mod 3
print e
print e2



set f to -23+3*4
print f
```

```
set f2 to 12*(4/2+2)
print f2


set g to "hello"+f2+"bye!"
print g


set i to (10+2)*3
set i2 to ((10+2)*3-6)*2
print i
print i2

set j to [1, 2, 3]
print child 0 of j
print child 2 of j


set j2 to ["a", "b", "c", "d", "e"]
print child 1 of j2
print child 3 of j2

~{
whileTest.parcel: while-statement testing
Min Chae
}~


print "while_statement test"

set i to 0
while i < 10:
    set i to i+1
    print i.

print ""

set i to -3
while i < 10:
    set i to i+2
    print i.

print ""

set i to 0
while i < 5:
    set i to i+1
    print i.

print ""

set i to 0
while i < 5:
    set j to 0
    while j < 3:
        print i
        print j
        set j to (j+1).
    set i to( i+1)
    print "i="+i.

print ""

set i to 0
while i<5:
    set j to 0
    while j<4:
        set k to 0
        while k<3:
            print "i j k = "+i+" "+j+" "+k
            set k to (k+1).
        set j to (j+1).
    set i to (i+1).
```