PDL: PATENT DRAFTING LANGUAGE

Project Proposal

Dean Alderucci
da2116@columbia.edu
February 9, 2010

1. Introduction

PDL allows users to perform all aspects of the patent drafting process using a high level language that is specific to the intricacies of patent documents. The language has built-in features that permit common but tedious and error prone tasks to be offloaded to the computer. Yet, the language is also general enough that users may customize functionality to suit their individual preferences.

By creating libraries in PDL, users can readily reuse previous work across many patents, and share improvements in design and analytic methodologies with others.

2. Background

Patents are grants of exclusive rights in inventions. The owner of a patent can exclude others from, or charge others for, making or using the patented invention.

Patents are created via documents. An inventor describes his invention to a patent lawyer, who drafts a document that describes the operational details of the invention and its variations. The document is often dozens of pages long, and lawyers typically charge between $10,000 and $25,000 to draft it. The document is sent to the patent office, which usually demands more drafting before the patent is granted. Once granted, the document defines exactly what is, and what isn't, protected by the patent. Most technology companies require many related patents. For example, several related technologies for a new suite of products may give rise to dozens of patents. A set of such related patents can cost hundreds of thousands of dollars in legal fees.

Patents have a great deal in common with mission-critical software. A patent is a complex, functional document written by a human, and is drafted solely to accomplish particular goals - protect particular variations of an invention. A patent does not accomplish what the author intended, only what the author actually wrote. A patent that appears correct, but that is written imprecisely or is even slightly incorrect, can completely fail to accomplish its goals. Even small errors in a single patent can cost millions of dollars in lost revenue opportunity or increased competitive activity. An error propagated across a set of related patents can be even more costly.

3. Goals of the Language

In general PDL allows the author of a patent to define abstract constructs (not simply blocks of text) that exist in a patent, analyze and manipulate those constructs during the drafting process, and generate the text of a patent as defined by those constructs.

More specifically, a program written in PDL embodies a patent's design. Therefore, the program, and thus the design, can be easily revisited and rewritten during the design phase of a patent, and at other phases of the patent's life cycle. For example, a PDL program may be

- written at the early stages of the design phase

- run to conduct an analysis of the patent's design
- amended and run several times to perfect the design by removing various errors and poor design choices
- run to generate the text that forms a patent
- reused in whole or in part in sets of related patents and by others

4. Sample Program

```
<<1.  Create first claim set >>
LIST steps1 = {process step 1, process step 2, process step 3}
LIST steps2 = {step A, step B, step C, step C}
LIST steps3 = {sorting step X, sorting step Y, sorting step Z}
TEXT prefix = {A method which includes all of the following steps:}
CLAIM claim1, claim2, claim 3
ADD COMPONENTS claim1 prefix, steps1
ADD COMPONENTS claim1 prefix, steps2
ADD COMPONENTS claim1 prefix, steps3

CLAIMSET s1 = claim1, claim2, claim3 <<add the three CLAIMs to s1 >>

<<2.  Create 2nd claim set from 1st claim set.  Each claim in the 2nd set
is 'analogous' to (has common components with) the corresponding claim
in the 1st set, except for certain claims in the 1st set which have
predefined 'bad steps'.>>

TEXT prefix2 = {A computer with software programmed to perform any or
all of the following steps:}
TEXT prefix3 = {A disk containing software programmed to perform any
or all of the following steps:}
LIST badSteps = {step A, step M}
CLAIMSET s2
```

```
c = EACH s1 <<c assumes the value of each CLAIM in CLAIMSET s>>
[
     IF NOT c CONTAINS badSteps [
          CLAIM newClaim = c
          IN newClaim SUBSTITUTE prefix2 FOR prefix
          ADD s2 newClaim
     ]
     OTHERWISE <<create an analogous claim in a very different way>>
[
          CLAIM newClaim
          newSteps = COMPONENTS(2) c
          newSteps = newSteps + {, final step}
          ADD COMPONENTS newClaim prefix3,  newSteps
          ADD s2 newClaim
     ]
]
```

<< **3. Perform analysis on the two sets.** >>

```
INTEGER numSteps1 = 0, numSteps2 = 0
CLAIM c
c = EACH s1 [ numSteps1 =  numSteps1 + LENGTH COMPONENTS(2) c]
c = EACH s2 [ numSteps2 =  numSteps2 + LENGTH COMPONENTS(2) c]

TEXT diagnosticMessage = { is the longest claim set, and so is most
vulnerable.}
IF numSteps1 > numSteps2
     diagnosticMessage = {The first set} + diagnosticMessage
OTHERWISE
     diagnosticMessage = {The second set} + diagnosticMessage
OutputAnalysis diagnosticMessage
```

<< **4. Output the two sets as text, which will be included in a patent.** >>

```
OUTPUT s1, s2
```

5. PDL Syntax Examples

The language is intended to be used by patent lawyers, who have technical training but typically do not have significant software development experience. Accordingly, the syntax of PDL has fewer of the characteristics that are common in higher level languages but would not be familiar to patent lawyers.

Comments

```
<< This is a comment.  >>
```

Declaration

```
TEXT t1, t2
TEXT t3 = {This is the assigned value of t3.}
```

Conditionals and expressions

```
IF (v1 < v3) AND ( (v1 = v2) OR (v1 > v4))
[    statements     << A block is surrounded by '[' and ']' >>  ]
OTHERWISE
     statements
```

```
Built In Functions and Operations
IF (t3 CONTAINS {short}) OR (t3 CONTAINS t2)

IN t3 SUBSTITUTE processor FOR microprocessor

ADD l1 {MAY}
```

PDL: PATENT DRAFTING LANGUAGE

Appendix – Extended Syntax Examples and Defined Types

Comments

```
<< This is a comment.  >>
<< This is another comment.
   Comments may be multiple lines.
   << and comments may be nested>>
>>
```

Declaration

```
<< Declaration without assignment>>
<<The end of line denotes the end of the declaration (and most PDL
statements)>>
TEXT t1, t2

<< Declaration with assignment>>
TEXT t3 = {This is the assigned value of t3.
Assignments can begin and end with braces, which is helpful for
multiline text literals. }
```

Conditionals and expressions

```
IF v1 = v2     statement1    <<Parentheses are optional>>
IF v1 = v2 OR v1 = v3 statement2 <<There is a precedence to Boolean
operators>>
IF (v1 < v3) AND ( (v1 = v2) OR (v1 > v4))
[    statement1     << A block is surrounded by '[' and ']' >>
     statement2  ]
OTHERWISE
     statement3
```

Loops

```
DOWHILE (v1 > v3) statement1
```

Functions

```
<<Declaring a function>>
MACRO foo
RETURNS TEXT
INPUTS TEXT t1, t2, INTEGER i
[    statement1
     statement2
]

<<usage of functions>>
TEXT t1 = foo t2, t3, 7
IF foo t4, t5, 0 = {January}
     statement1

<<Declaring a function with no return value>>
MACRO foo2
RETURNS NOTHING
INPUTS INTEGER i
[ statement1
statement2]

<<usage of functions with no return value>>
foo2 7
```

INTEGER – a signed integer.

```
INTEGER i = 3 + 4 - 2
i = i * 2 + 1
```

DECIMAL – a floating point number.

```
DECIMAL d = 3
d = d * 1.4 + 0.1
```

TRUTH– a Boolean which can have a value of either TRUE or FALSE. TRUTH values are returned by various conditional statements, such as in IF and WHILE statements. However variables of type TRUTH may also be declared.

```
TRUTH t1 = TRUE, t2 = NOT t1
TRUTH t3 = v1 = 3 OR v2 <= 7   <<note: nested assignments are not permitted >>
```

TEXT – a string. String literals are enclosed in braces. TEXT is essentially the 'default' type in many operations (since so much patent work involves text).

```
<< Assignment >>
TEXT t1 = {This is a short block of text}
TEXT t2 = {
Note that all newlines and white space between the braces are part of the value.

This is a new paragraph in the block of text t2.}

<< Concatenation >>
TEXT t3 = t1 + { February } + t2

<< defined operator - CONTAINS>>
IF (t3 CONTAINS {short}) OR (t3 CONTAINS t2)
     t3 = {a new assigned value for t3}

<<Replace throughout the string>>
IN t3 SUBSTITUTE processor FOR microprocessor
```

LIST – an array of one or more elements of any data type. Each element in a given LIST is of the same data type.

```
<< Assignment >>
<< By default, the elements of a LIST are type TEXT >>
LIST l1 = {January, February, March, Aprile}
l1(4) = {April}

<<Can specify the type for the elements of the list >>
INTEGER LIST l2 = {1, 2, 3, 4}
DECIMAL LIST d = {1, 3, 4.4, 9.0}

<< defined operator – CONTAINS returns TRUE or FALSE>>
IF (l1 CONTAINS {January}) OR (t2 CONTAINS 3) statement1

<<defined operator – ADD appends an element to the end of the list>>
ADD l1 {MAY}

<< defined operator – LENGTH returns an INTEGER>>
if LENGTH l2 = 7  statement2

<<Built in iterators over the LIST>>
x = EACH d <<x assumes the value of each element in list d>>
[    statements (involving x)  <<block statements are executed>>
]
```

HIERARCHY – a (directed) tree data structure, in which each vertex in the tree holds a value of any data type.
Computer science terminology is replaced with "patent-ese" as follows:

tree → HIERARCHY
vertex /node → TYPE
(proper) descendant relationship between vertices → SUBTYPE relationship between TYPEs. SUBTYPE may refer to the relationship and occasionally (with slight abusive of terminology) will refer to the vertex which is a child of another

vertex, as in "create a SUBTYPE of a TYPE".
root → BASE

<<In assignment of a HIERARCHY variable, the first value in the assignment is the root vertex. Nested braces denote children of the previous vertex >>

<<By default, elements of a HIERARCHY are type TEXT>>
HIERARCHY computers = {computer {server, personal computer {Mac, PC, Linux System} embedded computer {watch}}}
HIERARCHY computers2 = {computer {server, personal computer, embedded computer}}

SUBTYPES computers2 {personal computer} = {Mac, PC, Linux System}
TEXT baseName = BASE computers <<assigns value of the root vertex >>

<<Built in iterators over the HIERARCHY>>
c = EACH computers <<c assumes the value of each vertex>>
[*statements (involving c)* <<and the statements in the block are executed>>
]

CLAIM – a data structure that represents a 'claim' of a patent.
A variable of type CLAIM has
a DEPENDENT field which indicates whether the claim is intended to depend on another claim (i.e. inherit features from that other claim)
a TYPE field storing an optional 'handle' for use in identifying custom kinds / categories of CLAIMs
a COMPONENTS field storing a LIST of arbitrary content.
a FORMATTER field storing a 'formatter' function that outputs the final text of the claim (i.e. for including in the patent document).
This architecture allows many different types of claims with any structure to be created, and structures can be refined and improved by others.

```
<< For simplicity, a CLAIM variable can be assigned as if it were a
TEXT variable, in which case the text is stored as the sole element in
the variable's COMPONENTS field, and default values are used for the
remaining fields.>>
CLAIM claim1, claim2 = {A computer which includes: a monitor,
keyboard, a semiconductor storage device and a microprocessor
connected to the storage device.}
COMPONENTS claim1 =  {this is the first element, this is the second
element}
FORMATTER claim1 = foo <<set the FORMATTER field of claim1 to foo>>
TEXT t1 = TEXTOF claim1  <<this calls claim1's FORMATTER function>>
```

CLAIMSET – a compound data structure that includes:
a HIERARCHY of vertices of type CLAIM
a function defining a numbering scheme applied to each vertex
a formatting function to generate the complete text of the set of CLAIMs.

```
CLAIM bestclaim, claim1, claim2
CLAIMSET s = {claim1, claim2, bestclaim} <<like HIERARCHY assignment>>
FORMATTER s = foo <<set the FORMATTER field of s to foo>>

<<Built in iterators over type CLAIMSET>>
c = EACH s <<c assumes the value of each CLAIM in CLAIMSET s>>
[
     TEXT total = {}, total2 = {}
     IF c CONTAINS {computer}   total = total + TEXTOF c
     OTHERWISE   total2 = total2 + TEXTOF c

     CLAIM newClaim = foo(c, total, total2) <<create a claim >>
     DEPENDENT c = newClaim  <<add it as a dependent claim depending
on c>>
     additional statements
]
```