Language Reference Manual: PDFml
A PDF Manipulation Language

Lingshi Huang - lh2561@columbia.edu
Patrick McGuire - pjm2119@columbia.edu
Miriam Melnick - mrm2198@columbia.edu

Professor: Stephen Edwards
Fall 2011

December 25, 2011

Contents

1 Introduction

PDFml is a functional programming language based on Common Lisp and O'Caml. It serves to allow the programmer to manipulate PDF documents. Available operations include split, merge, add page numbers, and more. PDFml files compile internally (using logic written in O'Caml) into Java code, which is then run on the native Java Virtual Machine (using a library called iTextPDF).

2 Lexical Conventions

There are five kinds of tokens: identifiers, keywords, constants, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* begin a comment, and the characters */ terminate one. There is no special format for single-line comments and there is no nesting of comments.

2.2 Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetical or _(underscore). Upper and lower case letters are considered different. Only the first thirty-one characters are significant.

2.3 Keywords

The following identifiers are reserved for use as keywords:

- null
- boolean
- true
- false
- number
- string
- name
- array
- dictionary
- stream
- file
- define

2.4 Constants

2.4.1 Null Constant

The null constant has its own type. It has no value besides null.

2.4.2 Boolean Constants

A boolean constant can have one of two values: true or false.

2.4.3 Integer Constants

An integer constant is a sequence of one or more digits, interpreted as decimal.

2.4.4 Real Constants

A real constant is a sequence of digits (interpreted as decimal) followed by either a decimal point or an 'e' with an optional sign ('+' or '-') and then one or more digits. These are double-precision floating point numbers.

2.4.5 String Constants

A string constant is a sequence of zero or more characters (in ASCII) surrounded by double quotes (").

2.5 lvalues

Lvalue can be any variable or object name. It can be in the following types: Boolean, Number (Integer, Real), String, Name, Array, Dictionary, Stream, List.

3 Types

- Null
- Boolean: All values other than #f evaluate to #t (true).
- Number (Integer, Real)
- String
- Name

```
get (Name Type)
/Type, /Kids and /Count
/*A name object is a basically a name that means something to a PDF Reader.
Keys in dictionaries are always name objects.*/
```

- Array

```
(array-set 0 (Integer 4) (Array a1)) —> returns Array
 /* to differentiate between string 4 and number 4 */
(array-get 0 a1) —> returns 4 of type Integer
(array-remove a1) —> returns Array
```

- Dictionary

```
(dict-add country capitol (Dict d1)) --> returns Dict
(dict-add USA Washington d1) --> returns Dict
(dict-find USA d1) --> returns (key value) pair
(dict-remove USA d1) --> returns Dict
(dict-keys d1) --> returns list of names that map to values
```

- Stream

```
(stream-set a1 (Stream s1)) --> returns Stream
/*Creates stream from array with default dict */
(stream-list s1) --> returns stream as list
(let (l1) (stream-list s1) (stream-set l1 s1)) -->
returns Stream with contents of s1
/* Creates stream from list with default stream dictionary */
(stream-dict s1) --> Returns dictionary of stream properties
```

- Function

```
(define func1 (Integer a) (Integer b) (a+b))
```

- File A file contains a number of reserved dictionaries. These include /Resources and /Contents. These dictionaries are automatically declared when the user initializes a file variable.
- List

```
(list-add (Integer 4) (List l1)) --> returns List
/*creates a list l1 and add 4 of type Integer to it.*/
(list-get 0 l1) --> returns value /*return the value in index 0*/
(list-remove l1) --> returns nothing /*removes l1*/
```

4 Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions.

4.1 Integer and Real

All Integers may be converted without loss of significance to Real. Conversion of Real to Integer takes place with truncation towards 0 (ex. 1.5 becomes 1). Erroneous results can be expected if the magnitude of the Integer result is less than 32,768 or greater than 32,767. Erroneous results can also be expected if the magnitude of the Real result is less than -10^{308} or greater than 10^{308}.

4.2 Integer and Boolean

Only Integers 0 and 1 can be converted to Boolean, where 0 converts to false, and 1 to true. Conversion of Boolean to Integer takes place in the opposite direction.

4.3 Other

Any other conversions except this need to be specified explicitly. For instance:

```
a = 4        /* a is an integer variable */
b = ``4''    /* b is a string variable */
a = <Integer> b
```

5 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions referred to as the operands of + (see Section 5.4) are those expressions defined in Sections 5.1-5.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. Function arguments bind more tightly than juxtaposition. For example,

```
int 3 * 2
```

means to multiply 3 by 2 and cast the result to int.

5.1 Primary Expressions

Primary expressions involving . , subscripting, and function calls group left to right.

5.1.1 Identifier

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration.

5.1.2 Constant

Null or a Boolean, Integer, Real, String, Dictionary, Stream, or File constant is a primary expression.

5.1.3 (expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

5.1.4 primary-expression expression-listopt

A function call is a primary expression followed by a possibly empty, space-delimited list of expressions which constitute the actual arguments to the function. The primary expression must be of type “function returning ...”,and the result of the function call is of type “...”. In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in PDFml is strictly by value. A function may change the values of its formal parameters, but these changes cannot possibly affect the values of the actual parameters. Recursive calls to any function are permissible.

5.2 Multiplicative Operators

The multiplicative operators *, /, and % group left-to-right.

5.2.1 expression * expression

The binary * operator indicates multiplication. If both operands are Integers, the result is an Integer. If at least one operand is a Real, the result is a Real. No other types are permitted.

5.2.2 expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

5.2.3 expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be Integers, and the result is an Integer.

5.3 Additive Operators

The additive operators + and - group left-to-right.

5.3.1 expression + expression

The result is the sum of the expressions. If both operands are Integer, the result is Integer. If both are Real, the result is Real If one is Integer and one is Real, the former is converted to Real and the result is Real. No other type combinations are allowed.

5.3.2 expression - expression

The result is the difference of the operands. The same type considerations as for + apply.

5.4 Relational Operators

The relational operators group left-to-right, but this fact is not very useful; "a <b <c" does not mean what it seems to.

5.4.1 expression <expression

5.4.2 expression >expression

5.4.3 expression ≤ expression

5.4.4 expression ≥ expression

The operators <(less than), >(greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the + operator.

5.5 Equality Operators

5.5.1 expression == expression

5.5.2 expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus "a <b == c <d" is 1 whenever a <b and c <d have the same truth-value). Equality operators (==, !=) usually apply to these types: INTEGER, REAL, NULL, BOOLEAN, STRING.

5.6 expression && expression

The && operator returns 1 if both its operands are non-zero, 0 otherwise. && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0. The operands need not have the same type, but each must have one of the fundamental types.

5.7 expression || expression

The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike | , || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero. The operands need not have the same type, but each must have one of the fundamental types.

5.8 Assignment Operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. For details on type conversion, see Section 4.

5.8.1 lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. When both operands are Integer, no conversion ever takes place; the value of the expression is simply stored into the object referred to by the lvalue.

5.8.2 lvalue =+ expression

5.8.3 lvalue =- expression

5.8.4 lvalue =* expression

5.8.5 lvalue =/ expression

5.8.6 lvalue =% expression

The behavior of an expression of the form "E1 =op E2" may be inferred by taking it as equivalent to "E1 = E1 op E2"; however, E1 is evaluated only once.

5.9 expression in expression

A pair of expressions separated by "in" is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right.

6 Control Structures

- If/Then/Else

```
(if (x > 5) ``big'' ``small'' )
```

- cond

```
(cond   ((x > 5) ``big'')
        (( x > 3) ``medium'')
        ((x > 0) ``positive'')
)
```

7 Scope Rules

PDFml is statically/lexically scoped. Objects and functions are accessible within their lexical scope. Identical identifiers mask ones in encompassing scopes.

8 Implicit Definitions

Variables declared implicitly will be taken as a base type of whatever the type required by the function. For example, if (merge x y) requires two files, and x is of type file, but y is undefined, y will be treated as an empty file). This cannot be applied to undefined functions