SPML Language Reference Manual

Jayesh Kataria, jvk2104@columbia.edu
Dhivya Khrishnan, dsk2121@columbia.edu
Stefano Pacifico,sp2562@columbia.edu
Hye Seon Yi, hsy2105@columbia.edu

March 5, 2007

1. Introduction

This document is a reference manual for SPML (Simple PDF Manipulation Language), a language for Simple PDF Manipulations.

2. Lexical Conventions

2.1 Tokens

Tokens are divided in the following classes:

Identifiers
Keywords
Constants
String Literals
Operators
Other Separators

Blanks, horizontal and vertical tabs, newlines, formfeeds and comments are considered as "white space". They have no semantic meaning, and are used only as token separators.

The input stream is divided into tokens where each token is a group of valid characters that could constitute a token.

2.2 Comments

Comments are defined as a stream of characters that begin with the sequence **/*** and end with the sequence ***/** . Comments cannot be nested and cannot occur inside a string or character literals.

2.3 Identifiers

An identifier is a sequence of letters and digits. The first letter must be a character between [A-Z] or [a-z]. The remaining characters can be from [A-Z] or [a-z] or digits from [0-9] or the underscore (_) character.

2.4 Keywords

The following identifiers have been reserved as keywords and cannot be used otherwise

break	**close**	**continue**	**create**	**else**
extractpage	**getauthor**	**highlight**	**if**	**in**
int	**merge**	**open**	**pdf**	**print**
return	**setauthor**	**start**	**string**	**totextfile**
void	**while**			

2.5 Constants

The different kinds of constants supported by SPML are :

2.5.1 Integer Constants

SPML only supports integer constants to the base 10 i.e. decimal integers. Constants in the octal or hexadecimal number system are not supported. All integer constants are signed constants. Unsigned constants are not supported by SPML.

2.5.2 String Constants

String constants, also called string literals, are sequences of characters enclosed in double quotes for example `"abc\n"`.
The escape sequence `'\n'` denotes a new line and it can be used in any string constants.

2.6 Punctuations

These are the punctuation symbols in SPML.

```
;     statement end
,     argument separator
" "   string constant
[]    range or array index delimiter
{}    function body or block delimiter
()    function argument list
```

3. Type Specifiers

3.1 Basic Types

In SPML, there are four basic types available.

3.1.1 int

int is the primitive type for representing integer numbers. **int** type is compatible with Java **int** type, which means each **int** is 4 byte-long. When an **int** variable is declared, the variable is initialized to 0 automatically.

3.1.2 string

string is the primitive type for representing character strings. **string** type is compatible with Java String type, which can hold strings of variable length.

3.1.3 pdf

pdf is the primitive type representing PDF documents

3.2 Composite types

Composite types are composed of the basic types. **array** is a composite type which can store or reference a variable length of data items with one same basic type. An array is declared the following syntax is used.

```
<basic type> <variable name> [(length)?];
```

The basic type can be **int**, **string**, **pdf**. Users can set a random name for a variable name as long as the name follows the rules for identifiers. Brackets are used to specify the length of the array. The length is optional so that an array without data items can be declared. An array with no data item can be used as a holder for dynamic array creation, for another array, or for returning an array type for a function.

4. Expressions

Different types of expression are supported, as reported below.

4.1 Arithmetic Expressions

Expression including arithmetic operators. An example is `a+b-c*d/f`.
Additionally, the + operator can be applied on PDF documents. It is used to combine two PDF documents into a new one. The following is an example to combine PDF documents referenced by `p1` and `p2` into one PDF document referenced by `p3`.

```
pdf p1, p2, p3;
p3 = p1 + p2;
```

4.2 Conditional Expressions / Relational Expressions

Conditional Expressions are those involving comparison operators such as <, <=, >, >=, == and !=.

They can be used inside of if and while loops in SPML in order to test conditions. The associativity of these operators is from left to right. All of these operators are binary operators that work on two operands. Examples are `a<b, b>=d`.

4.3 Logical Expressions

Logical Expressions can only take the values true or false. The logical operators supported by the language are &&, || and !. && represents the AND logic, || represents the OR logic and ! is the NOT operator. These operators can be used with relational expressions. An example is

```
if(a<b && b>c)
```

The same can be applied to the while loop as well. ! has the highest precedence, followed by && and then by ||.

4.4 Expression Evaluation

Expressions are all evaluated left to right. In case of logical expressions, if the evaluation of the first n elements of the expression are enough for evaluating the whole expression itself, the n+1th element will not be evaluated.

5. Operators

5.1 Unary Operators

Unary operators affect the expressions on their right.

5.1.1 Minus sign: - *expression*

The result of the minus sign **-** operator is the negative value of *expression*. *expression* must be of type **int**.

5.1.2 Logical negation: ! *expression*

The result of the logical negation operator *!* is the constant 1 of type **int** if *expression* value is 0; constant 0 of type **int** if *expression* value is otherwise. *Expression* must be of type **int.**

5.1.3 Variable length: length expression

The result of the array length operator **length** is an **int** value of the number of elements contained in *expression* if *expression* is of type **array.**
It returns the number of pages contained in the PDF document referred by *expression*, if *expression* is of type **pdf**.

5.2 Multiplicative Operators

Multiplicative operators associate expressions left-to-right.

5.2.1 Multiplication: *expression1* * *expression2*

The multiplication operator ***** returns an **int** value of the product between the values of *expression1* and *expression2* elements. *expression1* and *expression2* must be of type **int**.

5.2.2 Division: *expression1* / *expression2*

The division operator **/** returns the integer quotient of type **int** between the values of *expression1* and *expression2*. *expression1* and *expression2* must be of type **int**. Division by 0 is not allowed and will cause a runtime error.

5.3 Additive Operators

Additive operators associate expressions left-to-right.

5.3.1 Addition: *expression1* + *expression2*

The addition operator + returns the sum of the values of *expression1* and *expression2* if both *expression1* and *expression2* are of type **int**. It returns an object of type **pdf** being the concatenation of the PDF files referred by *expression1* and *expression2* if both *expression1* and *expression2* are of type **pdf**. Any PDF file referred by *expression1* or *expression2* not being accessible at runtime will cause a runtime error.
If one of *expression1* and *expression2* is of type **string** and the other one of type **int,** the expression of type **int** is converted into a string to be added to the other expression of type **string.** For example, `s2` will contain a string "string1" after running the code below.

```
int i = 1;
String s1 = "string";
String s2 = s1 + i;
```

If both of *expression1* and *expression2* are of type **string**, it returns a **string** which is their concatenation.
No other type combination is allowed.

5.3.2 Subtraction: *expression1* – *expression2*

The subtraction operator – returns an **int** value of the difference between the values of *expression1* and *expression2*. *expression1* and *expression2* must be of type **int.**

5.4 Relational Operators

5.4.1 Greater than: *expression1* > *expression2*
5.4.2 Less than: *expression1* < *expression2*
5.4.3 Greater than or equal: *expression1* >= *expression2*
5.4.4 Less than or equal: *expression1* <= *expression2*

The greater than >, greater than or equal >=, less than <, less than or equal <= operators return constant 1 or 0 of type **int** if the relation between *expression1* and *expression2* is respectively true or false, in the numerical order and both *expression* and *expression2* are of type **int.**
They return constant 1 or 0 of type **int** if the relation between *expression1* and *expression2* is respectively true or false in the alphabetical order and both *expression1* and *expression2* are of type **string.**
No other type combination is allowed.

5.5 Equality Operators

5.5.1 Equality: *expression1* == *expression2*
5.5.2 Inequality: *expression1* != *expression2*

The equality == and inequality **!=** operators return constant 1 or 0 of type **int** if *expression1* and *expression2* have the same value and both are of type **int.**
They return constant 1 or 0 of type **int** if *expression1* and *expression2* refer to the same PDF file or not, and both *expression1* and *expression2* are of type **pdf**.
They return constant 1 or 0 of type **int** if *expression1* and *expression2* are the same string or not, and both *expression1* and *expression2* are of type **string**.

5.6 Logical operators

5.6.1 Logical AND: *expression1* **&&** *expression2*

The logical AND operator **&&** returns constant 1 of type **int** if *expression1* and *expression2* value evaluation is different from 0 for both. It returns 0 otherwise.

5.6.2 Logical OR: *expression1* || *expression2*

The logical OR operator || returns constant 0 of type **int** if *expression1* and *expression2* value evaluation is 0 for both. It returns 1 otherwise.

5.7 Other operators

5.7.1 Merge PDF: **merge** *expression1 expression2*

The merge PDF files operator **merge** returns **pdf** type object referring to a pdf file containing the fusion of the pages of the PDF documents referred by *expression1* and *expression2*. *expression1* and *expression2* must be of type **pdf** and must refer to a single page PDF documents only.

5.7.2 In: *expression1* **in** *expression2*

The element of operator **in** returns an array of type **pdf** elements referring to all the PDF documents contained in the directory as for the path indicated in *expression2*; in this case *expression1* must be the **pdf** type keyword and *expression2* must be of type **string**.
It returns an array of type **int** elements containing the line number where *expression1* occurs in *expression2*; in this case *expression1* must be of type **string** and *expression2* must be of type **pdf**. The PDF file referred by *expression2* not being opened will result in a runtime error.
It returns constant 1 of type **int** if *expression1* is a substring of *expression2,* 0 if otherwise. In this case both *expression1* and *expression2* must be of type **string.**
No other combinations of types are allowed.

5.7.3 Extract page from PDF: **extractpage** *expression1 expression2*

The extract a page from PDF operator **extractpage** returns an object of type **pdf** referring to a new PDF document made of the page number *expression2* extracted from the PDF file referred by *expression1*. *expression1* must be of type **pdf**; *expression2* must be of type **int**.

5.8 Operator Precedence

These are the operator precedence in SPML from the highest to the lowest. The operators with the same precedence are placed in the same line.

```
[]
- ! length
* /
+ -
in
< > <= >= == !=
&& ||
=
,
```

All binary operators associate left-to-right whereas the unary operators associate right-to-left.

6. Statements

Below are the descriptions of the statements in SPML.

6.1 Variable Declarations

These statements are used to declare the variables to be used by the program. The declarations are terminated with a semicolon and declarations can be separated by the comma operator. Below are the examples of variable declarations.

```
int p1, p2, p3;
int p1;
```

Array declarations are also supported. An example is

```
int a[10];
```

Declarations of array and variables and assignment of variables can be given in one statement.

```
int a, b[5], c=10;
```

The above declaration consists of a variable of type **int**, an **array** of five **int** and an **int** assigned with a value 10. These three can be given in any order. Declarations of other types are that of **pdf** and **string**.

6.2 PDF statements

The following statements involving **pdf** type variables are supported.

6.2.1 open statement

open statement is used to open an existing PDF file. The first argument must be a **pdf** variable that should have been declared before this statement is used. The second argument is the absolute path of the PDF file and it must be of type **string**. The syntax of the open statement is as follows:

```
open pdf_variable "C:\p1.pdf";
```

By executing this statement we make the `pdf_variable` refer to the corresponding PDF file.

6.2.2 close statement

The syntax of save statement is:

```
close pdf_variable;
```

This statement is used to save the PDF document file referred by `pdf_variable`.

6.2.3 **highlight** statement

highlight statement underlines every occurrence of the word in a PDF document. The first argument is of **pdf** type referencing a PDF document. The second argument is of **string** type representing a word or phrase to be highlighted. An example of highlight statement is:

```
highlight pdf_variable "hightlighted_word";
```

This underlines every `hightlighted_word` in the PDF document referred by `pdf_variable.`

6.2.4 extractpage statement

This statement is used to extract a page of a PDF document file into a text file

```
Open pdf_variable "C:\p1.pdf";
pdf p = extractpage pdf_variable 1;
```

The statement extracts page 1 of `p1.pdf` which is referred by `pdf_variable` into the **pdf** variable `p`.

6.2.5 create statement

The create statement is used to create a new PDF document file. The first argument is of type **pdf** referring to a file object which will be created. The second argument is of type **string** which represents the path for the file object. For example,

```
create pdf_1_variable "C:\test.pdf";
```

The statement creates a new PDF document named test.pdf and makes `pdf_1_variable` refer to it.

6.2.6 setauthor statement

This is used to set the author of a PDF document file. The first argument is of type **pdf** which refers to a PDF document. The second argument is of type **string** which represents the name of an author for the PDF document. An example is is:

```
setauthor pdf_variable "Jill";
```

The statement sets the author of the PDF file referred by `pdf_variable` to Jill.

6.2.7 getauthor statment

This is used to get the author of a PDF document file. The first argument is of type **pdf** which refers to a PDF document. Below is an example.

```
string t;
t = getauthor pdf_variable;
```

The author of a pdf file pointed by pdf_variable is returned into a string variable t.

6.2.8 totextfile statement

This statement produces a text file version of a PDF document. The first argument is of type **pdf** which refers to a PDF document. The second argument is of type **string** which represents a path for a text file created. Below is an example:

```
pdf file1;
. . .
totextfile file1 "/path/to/textfile";
```

This example creates a text file "`path/to/textfile`" which contains the contents of a PDF document referred by `file1`.

6.3 Assignment statements

Assignment statements enable the programmer to define or redefine a symbol. The right side of an assignment statement can be an arithmetic expression. The syntax is the following:

```
Variable  = expression;
```

This is an example where the value 5 is assigned to the variable `x` .

```
int x = 5;
```

In SPML, the type of the operands on both sides of the assignment operator = must be the same. For example:

```
p1 = p2 + p3;
```

In this case both the type of `p1` (left side) and the evaluation of `p2` and `p3` (right side) should be one of type **int**, **string**, or **pdf**. Thus, type conversion is not allowed in SPML.

6.4 Conditional statements

Conditional statements define control flow based on a condition tested. SPML supports the **if..else** conditional construct. The syntax is

```
if(condition)
{
      (statement)*
}

else
{
      (statement)*
}
```

The `condition` must be an expression with an **int** value. If the expression evaluates to a value different from 0, the `if` block is executed. Otherwise, the `else` block is executed.

6.5 Iteration statements

Iteration statements are used to execute a block of statements more than once instead of replicating code. SPML supports the **while** iteration statement, the syntax of which is the same as C.

```
while(condition)
{
      (statement)*
}
```

If the `condition` evaluates to a value different from 0 then the while block is executed. Otherwise, the while block is not executed. `condition` must be an expression with an **int** value.

6.6 Print statements

Print statements are used to display the value of an expression on the screen. The syntax is:

```
print (int_var | string_var);
```

The following example prints '1' on the screen.

```
int i = 1;
print i;
```

6.7 Jump statement

Jump statements are used to jump to a different set of statements altogether. They affect the control flow immediately without any condition check as in iterative or conditional statements.

return : used in functions to return a single value or void.
continue : used to continue with the next iteration in a while loop construct.
break : used to break out of a while loop immediately.

7. Functions

There are two types of functions: language-defined functions and user defined functions. Users are able to define functions.

7.1 Keywords

These are three keywords related to functions.

7.1.1 return

A function returns to its caller by the **return** statement. All user defined functions need to have at least one return statement. However, return statement can be omitted in the **start()** function.

7.1.2 start

start() is a language-defined function which is the execution point in SPML programs. In other words, the program cannot be executed without a start function.

7.1.3 void

void is placed to symbolize non-existence in function declaration. If a return type is void, there is no return value of the function. If void is used as an argument, there is no argument needed to call the function.

7.2 Function Declarations

A function declaration has four sub parts, namely: a return type, a function name, a list of arguments, a body which consists of statements enclosed by curly brackets.

```
<return type> <function name> (argument list) {body}
```

A return type can be any of the basic types available, which are **int**, **string**, **pdf, array** composite type and **void**. A user can choose an arbitrary name for a user-defined function and the name should follow the definition given before for identifiers. An argument list can contain a number of arguments separated by a comma. An argument should be declared as a type followed by its name. The body contains statements executed when the function is called is followed.

For example,

```
int foo(pdf p, int x)
{
     x = x+1;
     return x;
}
```

This is the declaration of function `foo`, whose return type is an **int** and which takes a **pdf** type variable as the first argument and an **int** variable as the second argument. Then, the arguments are local variables within `foo` function body. In the example, the **int** variable `x`, which was passed at its calling, is incremented. Since the return type of `foo` function is of type **int**, `foo` function should have at least one return statement which returns an **int** value. In the example, `x` is returned.

7.3 Function Calls

User-defined functions can be called in any function including the calling function itself , allowing recursive calls. A sound function invocation matches exactly the name, number and type of arguments as for the function declaration. No name overloading is allowed. Also, to call a function, the function should be declared in advance. When a function is called with a list of arguments, the arguments are evaluated before calling the function from left to right. When the function returns a value, the returned value can be retrieved. However, this is optional. Thus, the function call follows the notation below.

```
(<variable> =)? <function name> (argument list);
```

Below are presented two examples of correct invocation of the function `foo` declared in 4.2:

```
pdf p;
```

```
int y;
foo(p, y);
int r = foo(p, y);
```

7.4 Argument Evaluation

Function arguments are evaluated following the *applicative approach:* left to right before the execution of the function.

8. Scope

A program should be compiled at once which means that all source code should be in one file. Thus, there is one kind of scope in SPML, the lexical scope of an identifier, that is the region of a program in which the identifier is recognized.

8.1 Lexical Scope

Identifiers in different name space do not interfere with each other. In SPML, there are two separate name spaces available: variables name space and functions name space. To clarify the explanation of the lexical scope, the meaning of a block should be defined. A block is a set of statements grouped by curly brackets, {}. Thus, a block of statements is executed like a single statement syntactically.
An identifier can be declared once in a block within a name space. Identifiers can be used several times in its block. If the same name of an identifier is declared outside the block, the identifier in the outer block will be undermined in the current block. Thus, SPML is following static scoping, which means the life of an identifier begins with its declaration and ends at the end of its block.

9. Samples of SPML code and programs

```
/* Declaration of a function */
int foo()
{
  print "\ninside foo";
  return 1;
}

/* main function of the program: this program illustrates a simple
definition of a function and a basic manipulation of PDF documents.
*/
start()
{
  pdf p1,p2,p3;

  /* open two existing PDF files */
  p1 = "C:\pdf1.pdf";
  p2 = "C:\pdf2.pdf";
```

```
    /* create a third PDF file */
    p3 = create "C:\pdf3.pdf";

    /* assign to the third file the concatenation of the first two */
    p3 = p1+p2;

    /* highlight the string “hello world” in the second file */
    highlight p2 "hello world";

    close p3;

    int i;
    i = foo();
  }
```

10.ANTLR Grammar

```
// Lexer
class SimpleLexer extends Lexer;
options {
    k = 3;
    testLiterals = false;
    exportVocab = SPML;
    charVocabulary = '\3'..'\377';
}

// punctuation
LPAREN : '(';
RPAREN : ')';
LCURLY : '{';
RCURLY : '}';
LBRACKET : '[';
RBRACKET : ']';
COMMA : ',';
DQUOTE : '"';
SEMI : ';';
NEWLINE : ( '\n' | '\r'
          | ('\r' '\n') => '\r' '\n')
          {newline(); $setType(Token.SKIP);};
WS : ( ' ' | '\t' )+
    { $setType(Token.SKIP); };

// string constant
STRCON : '"'! (~('"' | '\n'))* '"'! ;

// operators
```

```
NOT : '!';
PLUS : '+';
MINUS : '-';
TIMES : '*';
DIV : '/';
ASSIGN : '=';
EQ : "==";
INEQ : "!=";
AND : "&&";
OR : "||";
GT : '>';
GEQ : ">=";
LT : '<';
LEQ : "<=";
COLON : ":";

// keywords (alphabetical order)
BREAK : "break";
CLOSE : "close";
CREATE : "create";
CONTINUE : "continue";
ELSE : "else";
EXTRACTPAGE : "extractpage";
GETAUTHOR : "getauthor";
HIGHLIGHT : "highlight";
IF : "if";
INT : "int";
IN : "in";
LENGTH : "length";
MERGE : "merge";
PDF : "pdf";
PRINT : "print";
RETURN : "return";
SETAUTHOR : "setauthor";
START : "start";
STRING : "string";
TOTEXTFILE : "totextfile";
VOID : "void";
WHILE : "while";

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

// number
NUMBER : (DIGIT)+;
```

```
protected LETTER : ('a'..'z' | 'A'..'Z');
protected DIGIT : '0'..'9';

// comment
MLCOMMENT : "/*" (options {greedy = false;} : .)* "*/"
            { $setType(Token.SKIP); };

// Parser
class SimpleParser extends Parser;

options {
     k = 2;
     buildAST = true;
     exportVocab = SPML;
}

// expressions
expr : conditional_expr ( AND conditional_expr
                        | OR conditional_expr)*
     ;

conditional_expr : plus_minus_expr ( GT plus_minus_expr
                                   | GEQ plus_minus_expr
                                   | LT plus_minus_expr
                                   | LEQ plus_minus_expr
                                   | EQ plus_minus_expr
                                   | INEQ plus_minus_expr)*
                 ;

plus_minus_expr : times_div_expr ( PLUS times_div_expr
                                 | MINUS times_div_expr)*
                ;

times_div_expr : ((NOT)? (PLUS|MINUS)? (ID (array_expr)?) | NUMBER)
                   ( TIMES ((NOT)? (PLUS|MINUS)? (ID (array_expr)?)
                                 | NUMBER)
                   | DIV ((NOT)? (PLUS|MINUS)? (ID (array_expr)?)
                                 | NUMBER) )*
               | LPAREN expr RPAREN
               ;

array_expr : LBRACKET NUMBER RBRACKET
           ;
blank_array : LBRACKET RBRACKET
            ;

// main function statement
```

```
program : (function)* main (function)*
        ;

main : startprog body
     ;
startprog : START LPAREN RPAREN
          ;
body : LCURLY (stmt)* RCURLY
     ;

// function statement
func_prototypes : (VOID|type)functionprog SEMI
                ;
function : (VOID | type) functionprog body
         ;

functionprog : ID LPAREN
                  ((type ID (blank_array)?)
                        (COMMA type ID (blank_array)?)*)*
                  RPAREN
             ;

type : INT | STRING | PDF
     ;

function_call : ID LPAREN (expr (COMMA expr)*)? RPAREN
              ;
function_call_stmt : function_call SEMI
                   ;

// type statements
int_stmt : INT ID int_tail SEMI
         ;
int_tail : ((array_expr)|(int_assign))?
                  (COMMA ID ((array_expr)|(int_assign))?)*
         ;
int_assign : ASSIGN expr
           ;

// only pdf has blank array assignments
pdf_stmt : PDF ID pdf_tail SEMI
         ;
pdf_tail : ((array_expr)
           |(pdf_assign)
           |(blank_array))?
               (COMMA ID ((array_expr)
                         |(pdf_assign)|(blank_array))?)*
```

```
        ;
pdf_assign : ASSIGN pdf_assign_types
           ;
pdf_assign_types : STRCON
                 | expr
                 ;

string_stmt : STRING ID string_tail SEMI
            ;
string_tail : ((array_expr)|(string_assign))?
                   (COMMA ID ((array_expr)|(string_assign))?)*
            ;
string_assign : ASSIGN expr
              | ASSIGN STRCON
              ;

decl_stmt : int_stmt
          | pdf_stmt
          | string_stmt
          ;

// operation statements
close_stmt : CLOSE ID
           ;
create_stmt : CREATE (STRCON|ID)
            ;
merge_stmt : MERGE ID COMMA ID
           ;
highlight_stmt : HIGHLIGHT ID (STRCON|ID)
               ;
extractpage_stmt : EXTRACTPAGE ID NUMBER
                 ;
set_stmt : SETAUTHOR ID (STRCON|ID)
         ;
get_stmt : GETAUTHOR ID
         ;

op_stmts : close_stmt SEMI
         | highlight_stmt SEMI
         | set_stmt SEMI
         | print_stmt SEMI
         | totext_stmt SEMI
         ;

// other statements
conditional_stmt : IF expr (stmt|body)
                    (options {greedy=true;} : ELSE (stmt|body))?
```

```
                  ;
iteration_stmt : WHILE expr (stmt|body)
                 ;
jump_stmt : RETURN (expr)? SEMI
          | CONTINUE SEMI
          | BREAK SEMI
          ;

length_stmt: LENGTH ID
             ;
print_stmt : PRINT STRCON
             ;
totext_stmt : TOTEXTFILE ID
            ;

in_stmt : (STRCON|PDF) IN (STRCON|ID)
        ;
// id assigned statements (ie) assignment, getauthor, extractpage
id_assign_stmt : ID ASSIGN ((function_call) => function_call
               | array_expr
               | STRCON
               | expr
               | get_stmt
               | extractpage_stmt
               | length_stmt
               | merge_stmt
               | create_stmt
               | in_stmt) SEMI
               ;

// all statements
//id_assign_stmt has some of the pdf statements
//   - cause they need the assign operator in them
stmt : id_assign_stmt
     | decl_stmt
     | conditional_stmt
     | iteration_stmt
     | jump_stmt
     | op_stmts
     | function_call_stmt
     ;
```