DaMPL

Language Reference Manual

Bernardo Abreu    bd2440
Felipe Rocha      flt2107
Henrique Grando   hp2409
Hugo Sousa        ha2398
# Contents

1. Getting Started .................................................................................................................. 4  
2. Syntax Notations ............................................................................................................... 4  
3. Lexical Conventions .......................................................................................................... 4  
3.1. Line Structure ............................................................................................................... 4  
3.1.1. Logical Lines .......................................................................................................... 4  
3.1.2. Physical Lines ......................................................................................................... 4  
3.1.3. Comments ............................................................................................................... 4  
3.1.4. Blank Lines ............................................................................................................. 4  
3.1.5. Whitespace between tokens .................................................................................. 5  
3.2. Identifiers and Keywords ............................................................................................ 5  
3.2.1. Identifiers .............................................................................................................. 5  
3.2.2. Keywords ............................................................................................................... 5  
3.3. Literals ......................................................................................................................... 5  
3.3.1. String Literals ....................................................................................................... 5  
3.3.2. Integer Literals ...................................................................................................... 5  
3.3.3. Floating Point Literals .......................................................................................... 5  
3.3.4. Boolean Literals .................................................................................................... 5  
4. Statements ....................................................................................................................... 6  
5. Expressions ...................................................................................................................... 6  
5.1. primary-expression ..................................................................................................... 6  
5.2. Postfix-expression ...................................................................................................... 6  
5.2.1. Reference operator .............................................................................................. 7  
5.3. unary-expression ....................................................................................................... 7  
5.3.1. not operator .......................................................................................................... 7  
5.3.2. - operator ............................................................................................................. 7  
5.4. multiplicative-operators ............................................................................................. 7  
5.4.1. *, / and % operators ............................................................................................ 7  
5.5. additive-operators ..................................................................................................... 7  
5.5.1. + and – operators ................................................................................................ 8  
5.6. Relational operators .................................................................................................... 8  
5.6.1. <, >, <= and >= operators .................................................................................. 8  
5.7. Equality operator ........................................................................................................ 8  
5.7.1. == operator ........................................................................................................... 8  
5.8. AND operator ............................................................................................................. 8  
5.8.1. and operator ......................................................................................................... 9  
5.9. OR operator ............................................................................................................... 9  
5.9.1. or operator ............................................................................................................ 9  
5.10. Assignment operator ............................................................................................... 9  
5.10.1. = operator .......................................................................................................... 9  
6. Variables ........................................................................................................................ 9  
6.1. Types ........................................................................................................................ 9  
6.2. Conversion .................................................................................................................. 10  
7. Tuples ............................................................................................................................. 10
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.</td>
<td>Control Structures</td>
<td>11</td>
</tr>
<tr>
<td>8.1.</td>
<td>If</td>
<td>11</td>
</tr>
<tr>
<td>8.2.</td>
<td>While</td>
<td>11</td>
</tr>
<tr>
<td>8.3.</td>
<td>For</td>
<td>12</td>
</tr>
<tr>
<td>9.</td>
<td>Functions</td>
<td>12</td>
</tr>
<tr>
<td>9.1.</td>
<td>Definitions</td>
<td>12</td>
</tr>
<tr>
<td>9.2.</td>
<td>Calls</td>
<td>13</td>
</tr>
<tr>
<td>10.</td>
<td>Arrays</td>
<td>14</td>
</tr>
<tr>
<td>11.</td>
<td>Tables</td>
<td>14</td>
</tr>
<tr>
<td>12.</td>
<td>Dictionaries</td>
<td>15</td>
</tr>
<tr>
<td>13.</td>
<td>Include</td>
<td>16</td>
</tr>
<tr>
<td>14.</td>
<td>Program</td>
<td>16</td>
</tr>
</tbody>
</table>
1. Getting Started

DaMPL (Data Manipulation Programming Language) is a scripting language designed for high-level applications that require easy and robust data manipulation. Its features aim to define and manipulate information (either defined by the user or obtained from external sources) in a clear and concise way.

In this reference manual the reader is able to find a detailed description of the structure of DaMPL programs and explanations the main points of the language's features.

Within the language, there are many features to input and output differently structures data-types, as well as particular functions of the standard modules that manipulate specific file extensions, such as ".csv" and "xml", which are well known and extensively used files to store data.

DaMPL is implemented as translated to C (for more details, see the GNU C Reference Manual) language, using gcc. As a constantly evolving language, the implementation details are likely to change, therefore, the main focus on this manual is to provide a complete documentation.

2. Syntax Notations

For the next items of this manual, syntactic notations will be written in italic, and literal word and characters will be written as regular text. A definition of a new syntactic notation will be as follows:

\[ \text{notation-name:} \]
\[ <\text{possibility-1}> \]
\[ \ldots \]
\[ <\text{possibility-n}> \]

3. Lexical Conventions

3.1. Line Structure
A program in DaMPL is divided into a number of logical lines.

3.1.1. Logical Lines
The end of a logical line is represented by the token SEMICOLON. A logical line is constructed from one or more physical lines.

3.1.2. Physical Lines
A physical line is a sequence of characters terminated by an end-of-line sequence.

3.1.3. Comments
There are one-line and multi-line comments in DaMPL. One-line comments start with a sequence of two bar characters (//). Multi-line comments start with a sequence of a slash character followed by an asterisk character (/*) and end with a sequence of an asterisk character followed by a slash character (*).  

3.1.4. Blank Lines
A physical line consisting of only spaces, tabs, formfeeds and comments are ignored.
3.1.5. Whitespace between tokens
The whitespace characters space, tab and formfeed can be used interchangeably to separate tokens.

3.2. Identifiers and Keywords
3.2.1. Identifiers
An identifier in DaMPL can be defined as a sequence of letters and digits. It starts by a letter or underscore. Upper and Lower case letters are different.

3.2.2. Keywords
The following identifiers are keywords, or reserved words, of the language.

    and   for   include   tuple
    break fun not while
    continue if or
    else in return

3.3. Literals
Literals are notations for representing constant values.
    literal:
    string-literal
    integer-literal
    floating-point-literal
    boolean-literal

3.3.1. String Literals
A string literal is a sequence of characters surrounded by double quotes, as in " . . . ".

3.3.2. Integer Literals
A integer literal is a sequence of digits.

3.3.3. Floating Point Literals
A floating literal consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the e/E and the exponent (not both) may be missing.

3.3.4. Boolean Literals
A boolean literal is described by the following definition.
    boolean-literal:
    True
    False
4. Statements

statement:
  \( \text{lhs} = \text{statement} ; \)
  \( \text{expression} ; \)
  \( \text{control-structure} \)
  \( \text{fun-call} \)

statements:
  \( \text{statement} \)
  \( \text{statement statements} \)

5. Expressions

The precedence of expression operators is the same as the order of the subsections within this section. Operators in the same subsection have the same precedence. The associativity (left or right) is specified for each operator within a subsection. An expression is defined as follows:

expression:
  primary-expression
  unary-expression
  binary-expression

5.1. primary-expression

A primary expression is a literal, an identifier or a parenthesized expression.

primary-expression:
  literal
  identifier
  \((\text{expression})\)

5.2. Postfix-expression

The operators in a postfix expression group left to right

postfix-expression:
  primary-expression
  postfix-expression[\text{expression}] 
  postfix-expression[\text{<empty>}] 
  postfix-expression{\text{expression}} 
  postfix-expression{\text{<empty>}} 
  postfix-expression.\text{identifier} 
  postfix-expression(\text{argument-expression-list}) 

argument-expression-list:
  \text{expression} 
  \text{expression, argument-expression-list}
5.2.1. Reference operator

Dictionaries, Tables and Arrays elements can be accessed through a postfix-expression[expression], where postfix-expression: identifier for any of these types.

5.3. unary-expression

A unary expression group right to left.

unary-expression:
unop primary-expression

unop: one of
not -

5.3.1. not operator

The operand must have a boolean type. It will return the negated value.

5.3.2. - operator

The operand must have an integer type. Change the sign of the integer literal

5.4. multiplicative-operators

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

multiplicative-expression:
expression
multiplicative-expression * expression
multiplicative-expression / expression
multiplicative-expression % expression

5.4.1. *, / and % operators

The operands for all these operators must have arithmetic type (int or float).

The * returns the multiplication result in the equivalent arithmetic type of the operands. In case of the operands being of different types, the int gets promoted to float.

The / returns the result of the division as a float. The right operand can't be zero.

The % returns the remainder of the division as an int. The right operand can't be zero.

5.5. additive-operators

The additive operators + and - group left-right.
additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression

5.5.1. + and – operators
The operands for these operators must have arithmetic type. In case the operands differ in type, the int is promoted to float.
The + returns the sum result in the equivalent arithmetic type of the operands.
The – returns the subtraction result in the equivalent arithmetic of the operands.

5.6. Relational operators
The relational operators group left-to-right.

relational-expression:
  additive-expression
  relational-expression < additive-expression
  relational-expression > additive-expression
  relational-expression <= additive-expression
  relational-expression >= additive-expression

5.6.1. <, >, <= and >= operators
The operands for these operators must have arithmetic type. In case the operands differ in type, the int is promoted to float.
The < operator returns true if the left operand is less than the right operand and false if not.
The > operator returns true if the left operand is greater than the right operand and false if not.
The <= operator returns true if the left operand is less than or equal to the right operand and false if not.
The >= operator returns true if the left operand is greater than or equal to the right operand and false if not.

5.7. Equality operator

Equality-expression:
  Relational-expression
  Equality-expression == relational-expression

5.7.1. == operator
The == operator tests if both operands are equal, i.e., have the same value.

5.8. AND operator
The and operator groups left-to-right.

\[
\text{and-expression:} \\
\text{equality-expression} \\
\text{and-expression and equality-expression}
\]

5.8.1. and operator

The operands for the and operator must be of type boolean. The and operator returns a boolean value: true if both operands are true; false otherwise.

5.9. OR operator

The or operator groups left-to-right.

\[
\text{or-expression:} \\
\text{and-expression} \\
\text{or-expression or and-expression}
\]

5.9.1. or operator

The operands for the or operator must be of type boolean. The or operator returns a boolean value: true if at least one of the operands is true; false otherwise.

5.10. Assignment operator

The = operator groups right-to-left.

\[
\text{assignment-expression:} \\
\text{or-expression} \\
\text{postfix-expression = assignment-expression}
\]

5.10.1. = operator

The assignment operator expects the operands’ types to be the same. The assignment operator returns the right operand value.

6. Variables

DaMPL is not a typed language, i.e. you don't have to declare a variable along with its type, since it will be determined through type inference. To each variable is associated a value, an underlying type and an id (that is related to the position in memory where the variable data is stored).

Variable names (notation: identifier) must start with a lowercase letter or an underscore, and may contain letters, numbers and underscore.

6.1. Types
A type determines how the value stored in a variable is going to be interpreted. The types in DaMPL are the following: `int`, `float`, `str`, `bool`.

6.2. Conversion

Variables can be converted from one type to another using the constructors (notation: `fun-cal`) offered for each type:

- `int (identifier)`
- `float (identifier)`
- `str (identifier)`
- `bool (identifier)`

7. Tuples

Tuples are associations of values, where each item has a label name. These are useful to deal with data rows as we'll see later on this manual.

A tuple name (notation: `tuple-identifier`) must always begin with an uppercase letter, and may contain characters or numbers. Tuple labels (notation: `tuple-label`) must always begin with a lowercase letter, and may contain letters, numbers and underscore.

```
tuple-definition:
    tuple tuple-identifier{ tuple-label-list };

tuple-expression:
    tuple-identifier{ expression-list }

tuple-item-identifier:
    identifier$tuple-label
```

Where
```
tuple-label-list:
    tuple-label
    tuple-label, tuple-label-list
```
```
expression-list:
    expression
    expression, expression-list
```

Note: `tuple-item-identifier` can be used also on the LHS of an assignment

Example

```
// This defines a new tuple name
tuple Person{name, age};

// This instantiates one variable as a tuple Person
```
a = Person;
a$name = "Steve";
a$age = 30;

// Or using a simpler notation
b = Person{"John", 23};
	note that using a tuple does not implies any variable type assumption. The following line could be added to the example above.

c = Person{1, 1.2};

8. Control Structures

control-structure:

if

while

for

iterable:

identifier

string-expression

array-expression

table-expression

dict-expression

iterator_variable:

identifier

8.1. If

If statements are used for conditional execution and are defined as follows:

if

binary

while

for

if-else:

if binary-expression { statements }

if binary-expression { statements } else { statements }

if binary-expression { statements } else if-else

It evaluates the binary-expression and executes the first group of statements if this binary-expression is true. If the binary-expression is false and the else exists, the statements or if-else that follows the else are executed.

8.2. While

While statements are used for repeated execution of a statement as long as a binary expression is true. They are defined as follows:
while:
    while binary-expression { loop-statements }

loop-statements:
    statements
    break;
    continue;

This repeatedly tests the expression and, if it is true, executes the loop-statements. If the expression is false, the loop terminates.

A break executed in the loop-statements terminates the loop, while a continue executed in it skips the rest of the statements and goes back to testing the expression.

8.3. For
For statements are used to iterate through a iterable. They are defined as follows:

for:
    for iterator-variable in iterable { loop-statements }

loop-statements:
    statements
    break;
    continue;

For each item provided by the iterable, this item is assigned to the iterator-variable using the standard rules for assignments, and then the loop-statements are executed. When the items are exhausted, which means that the for has gone through all the items on the iterable, the loop terminates.

A break statement executed in loop-statements terminates the loop, while a continue statement executed in it skips the rest of the suite and continues with the next item, or terminates the loop if there was no next item.

9. Functions
9.1. Definitions
DaMPL allows the user to define their own functions in order to manipulate data structures in the way that best fits their needs.

Function names (which also use notation identifier, just like variable names) must start with a lowercase character or an underscore, and may contain characters, numbers and underscore.

In order to start a new function definition, the user needs to use the keyword fun, as shown below:

fun-def:
fun identifier( parameter-def-list ) { function-body }

Where

parameter-def-list:
  identifier
  identifier, parameter-def-list

And function-body is a sequence of logical lines, the last one being usually (but not
mandatorily) a return statement.

function-body:
  statements
  return expression;

The return keyword tells the function what is the expression that will be evaluated and
returned to the function caller.

Examples:

fun myFunction123(arg1, arg2) {
  return (arg1 + arg2);
}

fun anotherFunction1() {
  print("Hello World");
}

9.2. Calls

For calling a function, the user simply has to use the function name and the
desired arguments.

fun-call:
  identifier( parameter-call-list )

Where

parameter-call-list:
  expression
  optional-parameter-call-list
  expression, parameter-call-list

optional-parameter-call-list:
  identifier = expression
  identifier = expression, optional-parameter-call-list

When a function is called, the parameter call list will try to be matched with the
parameter definition list. Additionally, in case there are more parameters in the function
calling than in the function definition, the extra parameters will be stored in a special
variable called _opt which is of type Dictionary (see Dictionaries). These extra
parameters must obligatorily be specified with the = token, and their identifiers will be used to index the _opt dictionary. Note that _opt exists for every function, whether it uses extra parameters or not. In case it doesn't, _opt will be empty.

10. Arrays

Arrays holds values of same-type (allowed internal types: integer, floating point, boolean or string), which can be accessed by its zero-indexed positions.

array-expression:
  [ array-items ]
array-items:
  <empty>
    integer-items
    floating-point-items
    boolean-items
    string-items
<x>-items:
  <x>-literal
  <x>-literal, <x>-items

Example:

a = [];
a[] = 1; // This adds 1 to the end of a
a[] = 2;
a[] = 3;
a[] = 4;

// The following would have the same result
b = [1,2,3,4];
b[0]; //equals 1
b[1:3]; // equals [2,3]

11. Tables

Tables holds same-tuple-label instances. To define a table the user needs to specify the tuple that defines the structure of the table.

table-instantiation:
  tuple-identifier[]

  table-indexing:
  identifier$tuple-label
In order to add an element (tuple) to the table, we use brackets, as follows:

Example:

```plaintext
tuple Foo{fa,fb,fs}; //defines a tuple Foo
a = Foo{1,2,"abc"};  //Creates a tuple a
b = Foo{2,3,"abd"};  //Creates a tuple b
t = Foo[];           //Instantiates a table that stores Foo tuples
t[] = a;             //adds tuple a to table t
t[] = b;             //adds tuple b to table t
t$fa;               //returns all fa elements in the tuples stored ([1,2])
t[0];               //returns the first tuple of the table ({1,2,"abc"})
```

12. Dictionaries

Dictionaries are data-types that works as maps. They are basically lists of data that can be indexed using different data-types. The item used to index the list is called key and the returned data after the indexing is called value. Therefore, we can see a dictionary as a list of pairs key, value.

Keys can be integer, floating point or string. Values can be any basic variable type, arrays or dictionaries.

A dictionary is defined as shown below:

```plaintext
dict-expression:
   { }                      { dict-items }

dict-key:
   integer-literal
   floating-point-literal
   string-literal

dict-value:
   identifier
   expression
   array-expression
   dict-expression

dict-items:
   dict-key:dict-value, dict-items
   dict-key:dict-value
```

Example:
d = {"a":1, "b":2, 20:{1:2, 1.1:4}};

d["a"]; //equals 1

d[0]; //equals 44

d[] = 44;

13. Include

A file of DaMPL code can gain access to another file with a include statement. This statement allows for the first file to access the content of the second one as if its code were present on the same file as the first.

\[
\text{include-statement:}\n\quad \text{include string-expression;}
\]

14. Program

\[
\text{program-statements:}\n\quad \text{program-statement}
\quad \text{program-statements}
\]

\[
\text{program-statement:}\n\quad \text{statement}
\quad \text{include-statement}
\]