Università degli Studi di Pisa

Laurea Specialistica in Informatica

Advanced Programming

Final Term Paper

Start Date: 6/06/2017
Submission deadline: 26/06/2017 (send a single PDF file to attardi@di.unipi.it)

Rules:

The paper must be produced personally by the student, signed implicitly via his mail address.

You are allowed to discuss with others the general lines of the problems, provided that each student eventually formulates his own solution. Each student is expected to understand and to be able to explain his solution.

You are allowed to consult documentation from any source, provided that references are mentioned.

It is not considered acceptable:

- **to consult or setup an online forum, to request help of consultants in producing the paper**
- to develop code or pseudo-code with others
- to use code written by others
- to let others use someone's code
- to show or to examine the work of other students.

Violation of these rules will result in the cancellation of the exam and a report to the Presidente del Consiglio di Corso di Studio.

For the programming exercises you can choose a programming language among C++, C# and Java.

The paper must:

1. be in a single PDF file, formatted readably (**font size ≥ 10 pt** with suitable margins, single column), of **no more than 10 numbered pages**, including code: for each extra page one point will be subtracted from the score.
2. include the student name
3. provide the solution and the code for each exercise separately, referring to the code of other exercises if necessary. **Do not include in an exercise code only needed for a later exercise**.
4. cite references to literature or Web pages from where information was taken.

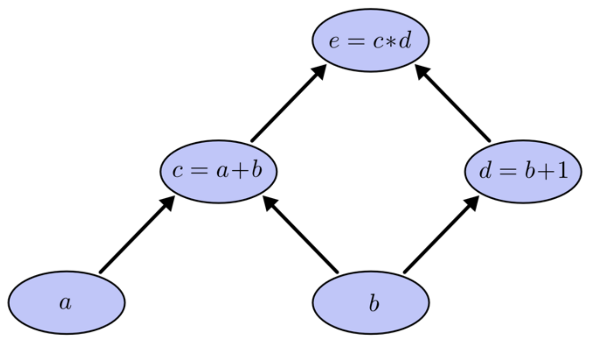
Introduction

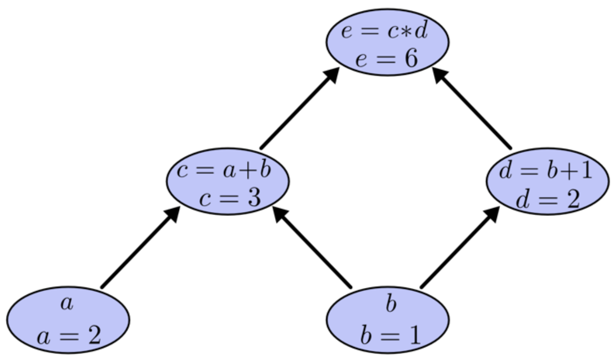
We will develop a computational graph formalism like the one used by Machine Learning libraries such as Theano or TensorFlow. A computational graph consists of nodes and edges. There are two kinds of nodes:

1. Input nodes, which are placeholders for input data.
2. Computational nodes, which take values from nodes on the incoming edges and compute a function on those values.

Edges connect nodes forming a Directed Acyclic Graph (DAG). The computation expressed by a graph is compiled into executable code in a target language that gets data from inputs and performs the operations expressed in the graph.

For example, the graph below on the left represents a computation with two input placeholders (a, b) and three computational nodes. When the graph is fed with a=2 and b=1, the computational nodes carry out the computation, producing the output in the top node as shown in the right graph.





This graph deals with inputs that are matrices of floats with 2 dimensions. Notice that a scalar can be represented as a matrix of dimensions [1, 1], and a vector of length *n* with a matrix of dimensions [1, *n*] or [*n*, 1].
The library compiles a graph into executable code in a target language.

Exercise 1

Design a hierarchy of classes to represent nodes and graph objects. The classes should provide suitable constructors for building nodes and graphs.

Exercise 2

Implement a recursive descent parser, which takes a JSON representation of a graph and builds the corresponding graph. For example, the above computational graph is represented as:

```
{
    ”a”: {“type”: “input”, “shape”: [1,1]},
    ”b”: {“type”: “input”, “shape”: [1,1]},
    ”c”: {“type”: “comp”, “op”: “sum”, “in”: [“a”, ”b”]},
    ”d”: {“type”: “comp”, “op”: “sum”, “in”: [“b”, [[1]]]},
    ”e”: {“type”: “comp”, “op”: “mult”, “in”: [“c”, ”d”]}
}
```

Notice that not all the operations are possible, for example summing two vectors of different shapes. The *Graph* class should provide the methods: *isDAG*, and *isValid* to check whether all the variables are defined and if all the operations are computable. For checking the compatibility, there should be a table specifying the signature for each function, for example:

```
sum: ([n, m], [n, m]) -> [n, m]
mult: ([n, m], [m, k]) -> [n, k]
```

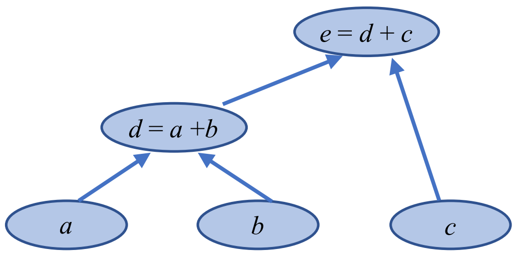
Exercise 3

Use polymorphism to implement a compiler that transforms a graph into executable code in a programming language. Use a template for each function, that represents the code to be generated for the body of the function. The resulting code should correspond to a function, which takes as arguments variables corresponding to the inputs of the graph and returns an array with all the outputs of the graph.

Provide the code generated for the example in the Introduction.

Exercise 4

Extend the code generator in order to perform code optimizations. In particular, the generator should identify cases where multiple operations can be fused into one, for example:



should be compiled into code that performs the addition $a + b + c$ with a single nested loop. Hint: use a template to represent the transformation.

Exercise 5

Discuss the technique of C++ template metaprogramming and the technique of LINQ expression trees.