
# Mini: an enough Programming Language

Authors: El Youssef Ahmad
Nimer Rakan

Project Supervisor: Professor Fadi Zaraket

Course: Advanced Programming Practices (EECE 637)

## Introduction:

In order to get used to working with people that we're not familiar with, and gain useful experience and insight for the professional software developer's world, we decided to let our group be chosen randomly. As luck had it, we ended up as a group of two people that had never spoken to each other before this course. However, we had a major interest in common and we both wanted to see what we were capable of doing compared to the others. After identifying each others' strength and fields of interest in the software world we were ready to start working.

## Task 0:

Task 0 was basically the introduction to the project's work. Neither of us knew what was an svn repository, what were its' functions and utilities and how it was supposed to be used. The lecture we had about it in class cleared up the questions we had, and after re-reading the slides provided and reading some documentation we found online, we had a solid idea about what it was, how to use it, and why it was useful. After that we created a project on Google's svn server, Google Code, and tried out it's functionalities, even though we didn't know how to use it well yet, over time we got the hang of it.

After having finished this part we had to choose what kind of programming language we wanted to implement. We had to choose between a mathematically oriented programming language and a programming language that was accessible to non-programmers. We ended up choosing a language in between that we called MPPL: Mathematical Pseudo Programming Language. We wrote the objectives and the target audience of the language and wrote down the grammar of the language.

## Task 1:

That was the task that took us the most time. After a lot of hesitation between the choice of Mini or Piano, we decided that it would be more interesting for us to choose Mini.

We set up the grammar of Mini on paper, based on our MPPL proposed grammar but with modifications. After that we re-read the slides about antlr and went to the antlr website looking for documentation. There was a lot about writing antlr grammar, so writing our grammar without the C++ code was easy and straightforward. It is after that part that things started to get complicated, neither of us had worked or programmed on an operating system different than Windows and without an integrated development environment. We downloaded Cygwin to simulate a Unix environment on Windows, took the time expected to familiarize ourselves with it and after many visits to your office and many questions we were able to generate a Parser and Lexer in C++. We created a main function that reads from the input file we created for the beginning of this task and parses it. After fixing (many) bugs, the antlr grammar that we created was able to parse successfully a Mini program.

After that it was time to think about a suitable data structure that will hold a representation of our input. Thinking that it will be good programming practice and experience for us, we decided to write a directed acyclic graph class ourselves. The graph that we wrote had a very low abstraction layer, it was an array of linked lists, and we worked with it as a such: If we wanted to insert a node we had to create a new Linked list and add it to the array and to add an edge we had to insert a node into the linked list of the origin node. So for every operation on the graph we had to know what was happening everywhere in the graph because we needed to pass indices to tell the graph where to store things. We added the actions to the grammar and everything was working (mostly) fine until we got to the expression parsing. As you can see in our grammar the expression use a lot of recursion, and keeping track of the indices with all this recursion was extremely hard. So we dropped the graph, wrote a new one, and restarted implementing the grammar actions. After a lot of work and debugging, everything was working. And we did the makefile before the debugging part but that was pretty easy but very useful in the debugging process.

## Task 2:

To write the aspect we had to familiarize ourselves with the aspectC++ and the dot syntax, that was the easy part. The ac++ and ag++ compilers for windows were not working under Cygwin. We tried to debug them and rebuild them, but unfortunately we're not that good yet. So we downloaded Ubuntu and got a Mac (not for the project specifically) and started working on them. The aspect written was used before the execution of DFS to open the dot file and write the introductory line. Another one was after the execution of the visit function that marks the node as visited, the aspect code gets the name of the node and writes to the dot file, the label, color and shape of the node depending on it's name and then visits the adjacent nodes and writes the relation between them (a->d) in the dot file. The last one is an advice after the execution of the DFS traversal, it appends the '}' symbol to the dot file when all recursive functions return and the graph is entirely traversed. The aspect not only helped us get a concrete look of what is happening, but also helped us in the debugging. Debugging gave us such a hard time mostly because of the recursive nature of the expressions, but it was worth it, the parser can easily parse expressions such as

2*4/2 –5 *6 +-2 for example and much more as can be seen in the examples provided.

## Task 3:

This task was not hard, a little time consuming but once you see the result it is worth it. In addition to simplifying the understanding of our code to readers, documenting it gives the coder a better understanding of what's going on and may help in finding bugs along the way and help in trimming unnecessary parts of the code, and this is because it forced us to reread a very large number of lines of code to be able to document every "important" element of it. As for the grammar (.g) documentation we started by modifying the doxyfile by adding the .g extension to the extensions that it should read and trying to document the grammar as such, however that did not work, then we tried to insert C++ documentation inside the .g grammar file however when Doxygen tried to parse the code it did not find it. Therefore, unfortunately the Mini*.* files are left undocumented by us. The documentation is automatically generated by the top level makefile and placed inside the doc folder.

## Task 4:

Up to this task all our work was done in one folder, and since we were new to the Unix environment, this helped a lot. It sufficed sometimes to just include the current directory '.' and the necessary files would be included and things would work. At this task all dependencies had to be studied and all makefiles had to take care of these dependencies. Before that there was one makefile and dependencies between the rules were pretty straightforward, this task gave us a solid understanding of how makefiles operate and how to automate the build of a large program. To do that there is a top level make file that builds the source code and generates the documentation, to build the source code the makefile enters the src/ directory and calls the makefile that is in it. This make file enters the directories in the subfolders (Graph, Mini, Main) if one of their component on which others depend has changed and calls the makefile that is inside it which will build the target, move object files to .Objs directory and if necessary builds a static or shared object library and moves them to the lib folder.

Note that the top level makefile builds and runs the program in order for the user to see the graph.

## Task 5:

In this task, we were faced with a choice: use the traversal function and convert as we go along the traversal or create two adapter classes for each translation. At first we wrote the traverse and translate functions that worked well, however we felt that it would be better to use more design patterns than a single factory Class. So we went ahead and wrote the adaptors class. That was no easy task since our functions (mainly the ones used in Mini.g) and the ProgGraph functions were not that similar. On small programs the Graph Adapters work, however when parsing a lot of bugs appear, after some nights of work on that, there were still many bugs that needed to be fixed, since we do not have anymore time we had to settle with the translator functions , however please try to check the GraphAdapter.h file even though it's not been used a lot of work has been done on it.

## Task 6:

We want to check the input and see if it's consistent with the values we gave it, to do so all we had to do was look for assignments in the graph representing the program and extracting the values in it, then we get the value of the input from the text file and adjust it according to the assignment. At the end of every assignment we will have a computed value of the variable and the value it's supposed to have according to the text file we compare them and return the corresponding result.

## Conclusion:

What we have learned and applied during this course and project:

-Becoming familiar with programming in the Unix environment

-Using Cygwin under Windows to simulate a Unix environment

-How Parsers work

-Writing a grammar with antlr and parsing an input accordingly

-Integrating antlr with C++

-Using subversion

-Constructing complex graphs

-Aspect C++ or adding functionality to the code without altering it

-Automating the build process

-Using the dot tool

-Generating automated documentation with Doxygen

-Using Design Patterns and how and when to apply them to real situations

-Software programming with competition and fixed deadlines

-Programming with people you didn't know