Appendix C

PP Final Project

The project of Programming Paradigms module consists of the development of a complete compiler for a (source) language that you may define yourself, as long as certain minimal criteria are met. The target machine for compilation is a realistic, though non-existent processor called SPROCKELL, for which a hardware-level simulator is provided in HASKELL. The corresponding machine language is called SPRIL (see the Tooling menu on BLACKBOARD).

Read this appendix carefully to understand what is expected.

- §C.1 describes the source language requirements
- §C.2 describes the possible options for the compilation process
- §C.3 describes what you should do to test your compiler
- §C.4 describes the requirements for the end result (software and report)
- §C.5 describes how the project will be assessed

Global schedule for the project.

- *Block 7:* Choice of language features, sample programs
- *Block 8:* Syntax definition, elaboration (type checking and other analyses),
- *Block 9:* Code generation and testing
- *Block 10:* Report and submission
- *Final deadline:* Friday 7 July 2017. The rules for late submission apply.

During Blocks 7–10, there are regular scheduled hours of assistance. You will be asked to show your progress and discuss your choices at least once during this period.

C.1 Source language

We go through a list of possible language features, for each feature mentioning whether supporting it in your language is optional or mandatory. §C.5 gives a summary and explains how the choice of features influences the expected grade for the project. Some of the optional features have not been treated during the lectures; however, EC Chapter 7 contains extensive pointers on how to implement them.

C.1.1 Data types

One of the first choices to make is which data types your language should support. The list below is a very short summary; section 4.2 of EC has given you an overview, and Chapter 7 of EC is devoted to implementation details.

- *Basic types: integers and booleans (mandatory)* In Blocks 3–5 of the CC strand you have already gained experience with these two most essential of basic types.

- *Arrays (encouraged).* Arrays make it much more easy to write non-trivial programs in your language. They provide an interesting challenge for the compiler, since it is no longer a priori clear what the size of a value is: obviously, this depends on the number of elements (the array length) as well as the size of the elements themselves. The array length may be fixed as part of the type (as was the case in early versions of PASCAL, for instance), but the may also be determined at run time, when the array is actually created (as is the case in JAVA). Run-time *changes* in array length are hard to support and may be ignored here. Note that, if the length is not fixed at compile time, you have to put array values on the heap, which raises the issue of heap management (see below).

- *Strings (optional).* Though very useful for pragmatic reasons, strings are a second-rate concept in most programming languages. Indeed, they can be seen as arrays of characters, but with their own typical constants and operators. Clearly, at the very least, the string type itself should not dictate the length of its values, making strings comparable to the trickier version of arrays mentioned above.

- *Enumerated types (optional).* These are user-defined scalar types (where the word *scalar* means that the values are singular and not composed). For the compiler, the challenge in supporting enumerated types lies in the choice of suitable syntax and the enforcement of a typing discipline; for code generation, values of enumerated types can be treated just like integer numbers.

- *Records and variants (optional).* A record is a combination of fields, each of which may have its own type. A variant, on the other hand, is a *choice* between different structures. A prime example of the latter are algebraic datatypes in HASKELL.

- *Pointers (optional).* A pointer is the address of a data value, rather than the value itself. The introduction of pointers to a language makes it unavoidable to place values on the heap, as their lifetimes are typically not limited to the lexical scope in which they are created. This again brings up the notion of heap management; see below.

- *Objects (optional).* Objects as in JAVA and other object-oriented languages are essentially pointers to records; for the compiler, they combine the challenges of both. Another layer of complexity is added if one also supports subtyping; in a way, this adds the notion of variants on top of the pointer/record combination.

Heap management. As soon as you have data values that are of unknown size (at compile time), you cannot put them into the allocation record any more; instead, you have to use the heap. This means that you need a mechanism to decide which locations on the heap are actually free at the moment the data value is created. This mechanism, called *heap management*, is a topic we have not covered at all during the course; see, however, §6.6.1 of EC for an extensive discussion.

More interesting yet is the notion of *automatic garbage collection*, as implemented in JAVA. §6.6.2 of EC covers this topic. Traditionally, the programmer had to explicitly deallocate heap memory when it is no longer used for anything; this is a famous source of very tricky errors, as a programmer may either forget to do this, or do it before the value has become unreachable. One of the big advantages of JAVA-like languages is that this task is taken out of the hands of the programmer.

Although you can't avoid having to allocate memory if you decide to implement dynamically-sized data values, for the purpose of this course you may ignore deallocation altogether. This means that your memory will slowly fill up if you repeatedly create such values, even if you abandon them afterwards. This phenomenon is called *memory leak*.

(On the other hand, would it not be extremely cool to write your own garbage collector?)

C.1.2 Simple expressions and variables (mandatory)

Your language should support appropriate expressions for all the data types you have chosen to implement. For the mandatory basic types, this comes down to the set of operators encountered in, e.g., the simple PASCAL fragment you have worked with in Blocks 4 and 5 of the CC strand; for the type extension, the kinds of operators strongly depend on the type itself. However, for every type you should at least offer:

- Denotations for primitive values of the type. For instance, for array types there should be a way to construct an array (something like `[1,2,3]` for an array of integers), and likewise for strings, records etc.
- Opreators for equality and inequality of values of the type. For instance, it should be possible to compare different values of the same array or record type. Such a comparison should be *content-based* rather than *identity-based*; in other words, it is *not* enough to compare the memory addresses where two (say) array values reside, they must be compared element by element.

Your language should support typed variables. Variable types may be given part of their declaration or inferred by their usage, but the type of every variable should be fixed throughout its scope and determined at compile-time. In other words, your language should be strongly typed.

Your language should support local scopes. It is not sufficient to have a single, global scope level only: within your language, it should be possible to reuse the variable name declared elsewhere with the same or a different type, at which point the two instances have nothing to do with one another. The reuse of a variable name in an *inner* scope is optional; that is, you may impose a restriction such as the one in JAVA that forbids the reuse of names in nested scopes.

Variables should also be checked for initialisation; that is, at compile time you should ensure that every use of a variable occurs after the variable has received an initial value. The initial value may be assigned explicitly (as for local variables in JAVA) or may be a default value of the type (as for instance variables in JAVA), but it should be well-defined.

C.1.3 Basic statements: Assignments, if and while (mandatory)

These language features are well-known and do not really need explanation. You are encouraged to look around a bit at other languages than the ones you know, to get inspiration about possible syntax and variations. For instance, besides **`while`**-constructs many languages also offer **`repeat`**, and there are widely differing variants of **`for`**-statements.

You may choose to allow assignments as expressions as in JAVA. For instance, in JAVA the following is legal:

```
int a = 3 - b = (c = 1) + 2;
```

which simultaneously assigns `1` to `c`, `3` to `b` and `0` to `a`.

C.1.4 Concurrency (mandatory)

For concurrency, you should add features to your language that make it possible to start and stop processes (a.k.a. threads), and to lock and unlock them. These should be compiled to the special SPRIL instructions provided for this purpose (see BLACKBOARD). Each process runs on a dedicated SPROCKELL; in other words, the architecture does not support dynamic thread creation.

Starting and stopping threads can either be offered (in your source language) on the basis of a `fork`/`join`-construct (see, e.g., `https://en.wikipedia.org/wiki/Fork-join_model`) or on the basis of a `parbegin`/`parend`-construct — essentially a block in which the statements should not be executed in sequence but in parallel. It is sufficient to offer concurrency only on the global level, and not within loops or procedures.

Inter-process communication in SPRIL is exclusively done through `WriteInstr`- and `ReadInstr`/`Receive`-instructions and an atomic `TestAndSet`, which all access a global, shared memory. This means that you have to distinguish between variables that are shared between threads, and hence should be stored in global memory, and variables that are local to a thread, which can therefore be stored locally. This distinction can either be made on the basis of an analysis (during the elaboration phase) about which

variables are accessed by more than one thread; or you may choose to build this distinction into the syntax of your language.

Because the global memory is the only medium of communication between SPROCKELL-instances, starting a thread must be encoded in that way as well. The demo program in `DemoMultipleSprockells.hs` shows how this can be done: essentially, a SPROCKELL is made to poll a predetermined global memory address until a non-zero value appears there; this value is taken to be the start address of the code that is to be executed.

To show the correct functioning of the concurrency feature of your language, you should provide at least the following test programs:

- An implementation of Peterson's algorithm for mutual exclusion (`https://en.wikipedia.org/wiki/Peterson%27s_algorithm`)
- An elementary banking system, consisting of several processes trying to transfer money simultaneously. Your implementation should ensure that concurrent transfers do not mess up the state of the bank account.

C.1.5 Procedures/functions (optional)

The principle of procedures and functions is well known in programming, and has been extensively discussed in the last blocks of the CC strand. There are several variants you may choose to support:

- Procedures only, functions only, or both. Functions have the slight additional complication that the return value should be stored in the activation record.
- Nested procedure definitions. These bring the added complication of access links or global displays, to access variables in enclosing scopes.
- Call-by-reference parameters. These necessitate a level of indirection: not the value itself is placed in an activation record, but the address in memory where the value is stored.

C.1.6 Exception handling (optional)

From JAVA you know the principle that any run-time error is visible within your program as an object of a subclass of `Exception` (strictly speaking, of `Throwable`). This is a very important feature of a language, because it allows you to deal with unpredictable errors in a way that does not completely clutter up your code.

Less powerful but still very useful variants of exception handling also exist. For instance, you may choose not to distinguish between different kinds of errors, or only use a single value from a predefined range to distinghuish them. Also, it is possible to restrict exception handling to a set of predefined exceptions and not allow the user to define and throw his own. In any case, however, an exception handling mechanism involves

- The detection of the error
- The notification of the error (in JAVA: **`throw`**)
- The handling of the error (in JAVA: **`catch`**).

If you choose to include exception handling in your language, you have to take care of the syntactic aspect (how should a programmer specify the handler, and how does he define and throw his own exceptions if that is supported) as well as the code generation part (detection and handling).

C.1.7 Optimisations (optional)

Though in the CC lecture we have not devoted a lot of attention to possible optimisations, the EC book has several chapters on this topic. You are encouraged to build one or more optimisation phases into your compiler. Candidates are, for instance

- Local value numbering (§8.4.1 of EC)
- Loop unrolling (§8.5.2 of EC)
- Inline substitution (§8.7.1 of EC)

C.2 The compilation process

In the CC strand you have exclusively used ANTLR as parser generator, but you are aware that there are other choices; in fact, you have also learned how to program and generate a parser in HASKELL.

C.2.1 JAVA front-end and code generation

If you choose this option, you have to fully apply the ANTLR-based skills you learned during the CC strand of the module:

- Write lexer+parser rules in ANTLR;
- Write a tree listener, using symbol table and attribute-like rules, that allow you to perform the elaboration phase of the front-end;
- Write a tree listener or tree visitor for the code generation.

In the last step, instead of generating ILOC, you have to generate SPRIL and emit the code as a HASKELL file that essentially contains a list of SPRIL instructions.

C.2.2 JAVA front-end, HASKELL code generation

If you choose this option, you use ANTLR to generate a parse tree, which you then hand over as a HASKELL data structure and process further in HASKELL.

- Write a lexer+parser in ANTLR;
- Write a tree listener in JAVA, using symbol table and attribute-like rules, that allow you to perform the elaboration phase of the front-end;
- Write a tree listener to emit the parse tree as a HASKELL file containing a single data structure;
- Write a HASKELL function that converts this data structure into a sequence of SPRIL instructions.

In the third step, it makes sense to abstract away the parse tree's irrelevant details (such as brackets, commas and other spurious syntactic structure), turning it into an abstract syntax tree (AST). In HASKELL, such an AST can be captured very naturally as an instance of an algebraic data type.

C.2.3 HASKELL front-end and code generation

If you choose this option, you may entirely ignore ANTLR and write your compiler in HASKELL altogether.

- Write a lexer+parser in HASKELL that generate an AST, encoded as an algebraic data type;
- Perform elaboration (scope and type checking) in HASKELL, on the basis of the generated AST;
- Write a HASKELL function that converts the AST into a sequence of SPRIL instructions.

C.2.4 SPROCKELL extensions

The SPRIL language and its simulator are all implemented in HASKELL. This makes it straightforward to extend the setup. In doing so, you also practice your FP skills and this can again be rewarded by some bonus points — see Table C.2. Whether or not a bonus will be awarded depends on the complexity of your extension; therefore, if you want to take advantage of this, please consult the teaching assistants.

Ideas for extension are:

- To add new instructions that ease the task of code generation.
- To change the channel mechanism that handles requests from parallel SPROCKELLs to the global memory, for instance by adding randomness.

C.3 Testing

You can increase your confidence in the correctness of a compiler by applying a series of tests. Of course, testing can only ever show the presence of errors, not their absence. Nevertheless, careful testing is useful and necessary in situations where formal verification is not possible. The advantage of testing is that it can be carried out independently of the way the compiler is specified and constructed.

Ideally, the tests should consist of all possible programs. Unfortunately, in most languages an infinite number of programs can be written, so all we can hope to do is to judiciously select a subset of all programs, called a test set, that is in some way representative of the language. A test set should not only contain correct programs, but also programs that contain errors, so that we can see how the compiler handles incorrect input. Errors in a program can occur in the:

- syntax (e.g. spelling errors in the lexical syntax, or language-construct errors)
- context constraints (e.g. declaration, scope and type errors)
- semantics (e.g. run-time errors)

Each of these classes of errors can be tested for separately.

C.3.1 Testing for syntax errors

This class of errors is about the first stages of compilation: scanning and parsing. For every language feature, you can consider the following (types of) test cases (the more comprehensive the better):

- One or more instances of that feature that should be syntactically correct. For such test cases, apart from checking that they pass the scanner and lexer, you can also check that the parse tree has the expected shape.
- One or more instances of that feature that should be syntactically incorrect. For such test cases, you can check that they are indeed rejected by the scanner or lexer.

For testing this class of errors, you do not have to invoke the full compiler, but just the scanning and lexing phases.

C.3.2 Testing for contextual errors

This class of errors is about scoping and typing: the things you check during the elaboration phase. What you should test strongly depends on the scoping and typing rules of your language. You may consider the following (types of) test cases (the more comprehensive the better):

- In a context where an identifier can appear (for instance: inside a block or procedure body), a test case where that identifier has been declared;
- In a context where an identifier can appear, a test case where that identifier has not been declared;
- For contexts that expect a certain type (for instance: an expression operand, **`if`**- or **`while`**-condition or procedure parameter), a test case where a simple expression of that (correct) type is used;
- For contexts that expect a certain type, a test case where a simple expression of a wrong type is used.

For testing this class of errors, you do not have to invoke the full compiler, but just the scanning, lexing and elaboration phases.

C.3.3 Testing for semantic errors

This class of errors is about run-time behaviour. Typically, you should include programs that are supposed to run correctly and of which the expected outcome is known, as well as programs that are known to contain a run-time error. You may consider the following (types of) test cases:

- Simple algorithms that calculate some value, for instance the number of days of the month February in any particular year (involving a test for leap years); whether or not a given number is prime. If you have implemented arrays, the scope for algrithms of the above kind becomes much larger.

- An algorithm that you expect to run into an infinite loop. Note that this is problematic to test automatically, as by definition your test will not terminate if the behaviour is as expected. In JUNIT, there is a way around this: the `@Test`-annotation has a parameter `timeout` that you can set to avoid tests that do not terminate; e.g.

```
@Test(timeout = 1000)
public void testSomething() {
    while (true) ;
}
```

  will cause the method `testSomething` to terminate after 1000 milliseconds and flag an error.

- Algorithms that you expect to generate a run-time error, for instance division by zero.

In the test suite for this class of errors, ideally every feature of your language should occur at least once in a correctly running program (meaning that you actually test whether correct code is generated for that feature, at least in the particular context of your test). The test involves both running the compiler, including code generation, and running the generated code on the SPROCKELL virtual machine.

C.3.4 Automatic versus manual testing

All the tests above are essentially *system tests* (the system under test is your compiler). This is not the same as a *unit test*, where you focus on part of your system; in JAVA, typically a single class.

During the CC strand you have written and executed a lot of JUNIT tests, which as the name implies are mostly unit tests. The great thing about a test support library like JUNIT is that it runs a large test suite automatically, so it becomes very easy to test often and test thoroughly. JUNIT *can* also be used for system testing; however, in some cases setting up a system test is a lot of work, and a manual test is preferable. This may very well be the case for some of the tests described above, especially where you have to run a combination of JAVA, HASKELL and SPROCKELL.

> *For the PP project, if you have a reasonable collection of test programs of the categories described above, but they can only be run manually, this translates to the default 0 for the testing aspect (in a range from $-1,5$ to $+1,5$, see Table C.3). Automated tests are a bonus but not a requirement.*

C.4 The final product

The product you should submit for the final project consists the developed software and a printable report. These should be uploaded to BLACKBOARD in a single zip-file. The general conditions for late submission apply.

C.4.1 Software

The submitted zip-file should contain the following elements (in a well-structured directory hierarchy):

- *The report*, in PDF-format.
- *A `README`-file* with instructions on using the compiler. Such instructions may include, but are not lomited to, an overview of the directories and files necessary for execution and the required steps for installation and invocation. Upon following these instructions, an end user should be able to obtain and invoke use a working compiler. *If this requirement is not met, the submission will not be graded; non-compiling programs are not accepted.*
- *Full grammars* as well as the code files generated therefrom by your chosen parser generator (ANTLR or otherwise).
- *Source code* of all classes programmed by you, in a single directory hierarchy, also including sources you reused from the CC laboratory. The code should meet the following criteria:

  - Compiles and executes without errors*
  - Contains documentation (in the form of JAVADOC or HASKELL comments)*
  - Meets common coding standards, for instance regarding naming, package structure, and visibility of fields.

  The criteria marked * are necessary to score more than 5,0 for the project.

- *Auxiliary code* of all predefined classes and libraries, insofar they are not part of the standard JAVA/HASKELL runtime environment.
- *Results of all tests.* See §C.3 for a discussion on what to test. For *correct* test programs, the result should consist of
  - The source code of the program itself
  - The generated SPRIL code
  - Some test runs

  For *incorrect* test programs, the result shoud consist of
  - The source code of the program itself
  - The output generated by the compiler for the program (i.e., the error messages)

Again, take care that your code compiles and runs after following the instructions in the enclosed `README`-file. *We should not have to change anything in your source code!* Typical cases where this goes wrong are: names and paths of files or other URLs, like host machines and servers. *Test this before submission.*

C.4.2 Report

The report should give insight in how the language has been defined, and how any problems occurring during the construction of the compiler were solved. The report should contain the following parts; for each part, list who of you was responsible, or whether the responsibility was shared equally.

- *Front page.* Clearly list the authors, including (for each student) first and last name and student number.
- *Summary* of the main features of your programming language (max. 1 page).
- *Problems and solutions.* Problems you encountered and how you solved them (max. 2 pages).
- *Detailed language description.* A systematic description of the features of your language, for each feature specifying
  - Syntax, including one or more examples;
  - Usage: how should the feature be used? Are there any typing or other restrictions?
  - Semantics: what does the feature do? How will it be executed?
  - Code generation: what kind of target code is generated for the feature?

  You may make use of your grammar specification as a basis for this description, but note that not every grammar rule necessarily corresponds to a language feature.
- *Description of the software:* Summary of the JAVA classes and/or HASKELL files you implemented; for instance, for symbol table management, type checking, code generation, error handling, etc. In your description, rely on the concepts and terminology you learned during the course, such as synthesised and inherited attributes, parse tree properties, tree listeners and visitors.
- *Test plan and results.* Overview of the test programs (which themselves are part of the submitted software, as described in §C.4.1). Here you should convince the reader that you have done systematic and extensive testing using correct and faulty test programs, and document how he can re-run the tests.
- *Conclusions*. Your personal evaluation of the language you have defined, as well asl the module as a whole. This is the right place to put your personal thoughts about what you have learned, what you liked and did not like about the module. *The content of the conclusion will not be graded; feel free to write whatever you like. However, the absence of a critical evaluation may decrease your grade.*

Appendices. In addition to the above, your report should also contain the following appendices:

- *Grammar specification.* The complete listing of your grammar(s), in the input format of your chosen parser generator (ANTLR or otherwise).
- *Extended test program.* The listing of one (correct) extended test program, as well as the generated target code for that program and one or more example executions showing the correct functioning of the generated code.

Check the readability of your listings, if necessary by putting them into landscape mode. If you used tabs, make sure the tab stops are the same for your editor and your printout.

(The reason to include listings in your report of files that are also provided in your zip-file is primarily to make it easier to assess your work. The idea is that the report can be used as the basis of the assessment; ideally, it should not be necessary to study your code separately.)

C.5 Assessment

The grade for the final project depends on:

- The language features supported in your programming language (§C.5.1);
- The degree to which you have applied your HASKELL skills (§C.5.2);
- The quality of the final product (code and report) (§C.5.3).

Concretely, the final grade is calulated as a basic grade with modifications; the basic grade is determined by the chosen language features, the modifications by the other two parameters. *The final grade for the project cannot exceed a 10; if the calculated grade is higher — which is possible and has occurred multiple times in the past — it will be "rounded" down to a 10.*

C.5.1 Assessment of the language features.

Table C.1 shows how the *basic grade* of the final project is determined. It should be noted that the basic grade for a language offering the mandatory features only is 6.5; this is sufficient for the final project. We hope you will tackle at least one extension; after all, the basic language is essentially what you already addressed in Block 5 of the CC strand (though you have to generate a different instruction set now), plus rudimentary concurrency features from the CP strand.

Table C.1: Basic grade determined by language features.

Language feature	*Max*
Mandatory part	6.5
Procedures/functions	+0.5
— also nested	+0.5
— with call-by-reference	+0.5
Exception handling	+1.0
Other extensions	+1.0
Arrays	+1.0
Strings	+0.5
Enumerated types	+0.5
Records	+0.5
Pointers	+1.0
Objects	+1.5
Optimisations	+2.0

For a language feature to be awarded (full) points, test program(s) must be included — see §C.3 and §C.5.3.

The suggested (though not enforced) order in which you should consider extensions is:

1. Arrays
2. Procedures/functions, preferably nested, with or without call by reference
3. Strings
4. Other extensions

Additional statement kinds such as **`switch`**-statements, **`for`**-statements or **`repeat/until`**-statements are regarded as "more of the same" and do not yield extra points; however, they can be used as reasons to round the final grade up.

The "optimisation" entry in Table C.1 should be read as a maximum: to earn this maximum, you should either implement two of the simpler optimisation strategies, or a more complex one, and also demonstrate their effect on a test program.

C.5.2 Use of HASKELL

The final project is meant to integrate the CC, FP and CP strands of the module. CP is integrated through the concurrency feature of your language; FP is integrated because you are at minimum asked to generate code in the form of a `.hs`-file that can serve as input to the SPROCKELL simulator built especially for the module. However, you are encouraged to go beyond this, by choosing one of the options described in §C.2.

Table C.2 describes how this choice of options can influence your grade.

Table C.2: Grade modifications based on HASKELL usage

Activity	*Max*
JAVA code generation of SPRIL (default)	+0.0
HASKELL code generation	+0.5
HASKELL front-end	+1.0
Extending SPROCKELL	+0.5

C.5.3 Quality of the final product

Not only the choice of language features and compilation process can affect your grade, but also how well you work them out. Aspects of quality are:

- How well have you tested your product? This regards the quality of your tests; see §C.3.
- How well have you structured and documented your grammar (the syntax specification underlying your language) and your code (the hand-written classes that perform the elaboration and code generation phases)? As a computer scientist, writing readable and well-documented code should become second nature; this is one opportunity to show your skills. Consult §C.4.2 to see what you are expected to hand in.
- How well have you written up your results? This regards the quality and completeness of the report you handed in; see §C.4.2.

Each of these aspects will be assessed separately, and can modify you grade either positively (if the aspect is covered particularly well) or negatively (if it is ignored or done badly). The ranges of modification are listed in Table C.3.

Table C.3: Grade modifications based on quality of code and report

Quality aspect	*Range*
Quality of testing	-1.5 . . . +1.5
All tests automatically executable	+0.5
Structure/readability of grammar	-1.0 . . . +1.0
Structure/readability of code	-2.0 . . . +1.0
Completeness/readability of report	-2.0 . . . +1.0

C.5.4 Example scenarios

Here are two examples of how concrete projects may be graded.

John and Mary are hard-core programming freaks, and they just loved the intricacies of concurrent programming. HASKELL, however, never conquered their hearts and minds. They cram in as many features as they can, but forget to test them well, so that in the end some things are not working properly. Also, they completely forgot to work on the report, but they have a hard deadline in the form of holidays (John is planning to visit Sicily with his parents and Mary has planned a trekking tour on Iceland with some friends) so pulling some all-nighters or late submission is not an option.

Their final grade is calculated as

- Mandatory part, functions, ref-parameters, objects, optimisation; $6.5 + 1.0 + 0.5 + 1.5 + 1.0 = 10.5$
- Exception handling was planned but didn't work out: $+0.0$
- JAVA code generation of SPRIL: $+0.0$
- Good grammar, poor quality code, really poor tests, minimal report: $+0.7 - 1.5 - 1.5 - 1.5 = -3.8$
- Final grade: $10.5 + 0.0 + 0.0 - 3.8 = 6.7$

Alice and Bob loved the FP side of the module. They choose to implement functions without call by reference and arrays as part of their language, to define the grammar of their language in ANTLR but to do code generation in HASKELL; and also to extend the SPROCKELL architecture. In addition, Alice is a control freak who is only happy when she can extensively test everything, but she is not hot on documenting code she herself already understands, whereas Bob writes beautifully and loves producing a well-written report. However, they do not plan well and in the end submit the result a day late.

Their final grade is calculated as

- Mandatory part, functions, arrays; $6.5 + 1.0 + 1.0 = 8.5$
- HASKELL code generation and SPROCKELL extension: $+0.5 + 0.5 = +1.0$
- Badly structured code, good tests, well-written report: $-1.0 + 1.3 + 0.8 = +1.1$
- Late submission: -1.0
- Final grade: $8.5 + 1.0 + 1.1 - 1.0 = 9.6$

C.5.5 Feel like a challenge?

If you do not find this project challenging enough or the requirements unnecessarily restrictive, if you do not want to be constrained by the imperative straightjacket or if you want to propose your own variation: this is possible in principle, but do contact the teacher(s) in time.