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 Appendix B-CP).

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 how the project will be assessed
- §C.5 describes what you should do to test your compiler
- §C.4 describes the requirements for the end result (software and report)

Global schedule for the project.

- *Block 8:* Choice of language features, syntax definition, sample programs
- *Block 9:* Elaboration (type checking and other analyses), code generation and testing
- *Block 10:* Report and submission
- *Final deadline:* Friday 3 July 2015. The rules for late submission apply.

During Blocks 8–10, there are daily scheduled hours of assistance.

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.3 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.
- *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 make 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 then also brings the notion of *allocation* and especially *deallocation* to the stage. For the purpose the final project, you may choose to ignore this issue — meaning that your programs will have memory leaks. (On the other hand, would it not be extremely cool to write your own garbage collector?)
- *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.

C.1.2 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 ect.
- 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.

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 if 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)

C.1.4 Concurrency (mandatory)

C.1.5 Procedures/functions (optional)

C.1.6 Exception handling (optional)

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 a parser in HASKELL.

C.2.1 JAVA front-end and code generation

C.2.2 JAVA front-end, HASKELL code generation

C.2.3 HASKELL front-end and code generation

C.2.4 Optimizations

C.3 Assessment

The grade for the final project depends on:

- The language features supported in your programming language (§C.3.1);
- The degree to which you have applied your HASKELL skills (§C.3.2);
- The quality of the final product (code and report) (§C.3.3).

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.

C.3.1 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 tackled in Block 5 of the CC strand, plus rudimentary concurrency features from the CP strand.

Because the concurrency feature is not trivial, the requirement on its inclusion is actually "softer" than for the other mandatory features; as the table shows, you may sidestep it, but this will cost you two full points that you will have to compensate by including some other language features instead.

For a language feature to be awarded (full) points, test program(s) must be included.

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

1. Arrays
2. Procedures/functions, 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.

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

Language feature	*Max*
Mandatory part	6.5
No concurrency	-2.0
Procedures/functions	+1.0
— 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

C.3.2 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	+0.0
HASKELL code generation	+1.0
HASKELL front-end	+2.0
Extending SPROCKELL	+1.0

C.3.3 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 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.
- How well have you tested your product? You should be aware by now that the primary way to convince yourself and others that any program is correct is to test it thoroughly. Without systematic tests, your claim to correctness will be based only on a very few sample scenarios you happened to have tried out. In the case of compilers, testing consists of feeding your compiler a lot of different source programs, both wrong ones (which should not compile) and correct ones (which should compile and give rise to code that has the expected behaviour). More information on how to test your program can be found in §C.5.
- How well have you written up your results? As part of the final product, you should hand in a report documenting the compiler, where you collect and summarise information about the language, the structure of the compiler and the tests you have performed.

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

Language feature	*Range*
Structure/readability of grammar	-1.0 ... +1.0
Structure/readability of code	-2.0 ... +1.0
Quality of testing	-1.5 ... +1.5
Completeness/readability of report	-2.0 ... +1.0

C.3.4 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, optimization; $6.5+1.0+0.5+1.5+1.0=10.5$
- 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, but they did not spend much time on the CP strand and do not feel up to the task of including concurrency in their language. 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. 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, no concurrency, functions, arrays; $6.5-2.0+1.0+1.0=6.5$
- HASKELL code generation: $+1.0$
- Badly structured code, good tests, well-written report: $-1.0+1.3+0.8=+1.1$
- Late submission: -1.0
- Final grade: $6.5+1.0+1.1-1.0=7.6$

C.3.5 Feel like a challenge?

If you do 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.

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 guidelines 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 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 isntructions, 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. 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 4,0 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.* 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. two 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 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 aditor 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 is the basis of the assessment; ideally, it should not be necessary to study your code separately.)

C.5 Testing

This is based on chapter "Self-testing your compiler" van [Henk Alblas, Han Groen, Albert Nymeyer and Christiaen Slot, Student Language Development Environment (The SLADE Companion Version 2.8), University of Twente, 1998].

We can increase our confidence in the correctness of a compiler by applying a series of tests. Note that 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.5.1 Testing for syntax errors

C.5.2 Testing for contextual error

C.5.3 Testing for semantic errors