RTFM-core: Course in Compiler Construction

Marcus Lindner, Andreas Lindner, and Per Lindgren
Luleå University of Technology
{marcus.lindner,andreas.lindner,per.lindgren}@ltu.se

ABSTRACT

The course in Compiler Construction is part of the Computer Science masters program at Luleå University of Technology (LTU). Since the fall of 2014, the course is given by the Embedded Systems group. This paper outlines the course syllabus and its relation to CPS/IoT and embedded systems in general. In particular, the course introduces domain specific language design with the outset from the imperative RTFM-core language. Students are exposed to design choices for the language, spanning from programming model, compiler design issues, back-end tools, and even run-time environments. The intention is to give a holistic perspective and motivate the use of compilation techniques towards robust, efficient, and verifiable (embedded) software. Of course, developing basic skills is not overlooked and as part of the laboratory assignments, students extend the minimalistic Object Oriented language RTFM-cOOre and develop the compiler accordingly targeting the RTFM-core language as an intermediate representation. As the RTFM-core/-cOOre compilers are implemented using OCaml/Menhir, the students are also exposed to functional languages and to their advantages in the context of compiler construction. However, for their own development they may choose alternative design tools and languages. This gives us the opportunity to review and correlate achievements and efficiency to the choice of tools and languages and it is an outset for future course development.

1. INTRODUCTION

Robustness, real-time properties, and resource efficiency are key requirements to- and properties of- embedded devices of the CPS/IoT era. In this paper, we present the newly designed course syllabus of the compiler construction course at LTU, based on the domain specific language approach RTFM-core (alternatively referred to as -core in the following). The language is geared towards facilitating the software development for lightweight embedded devices, such as embedded sensors, actuators, and controllers common to CPS and IoT applications. The paper discusses a new approach for teaching the important topic of compiler construction in computer science studies and shows how the concepts are presented using the example of a language specialized for embedded real-time development. The design and compilation of the language is presented in context of showing its capabilities for specification and verification of real-time properties.

The -core compiler produces highly efficient implementations amenable for static verification of real-time properties and resource utilization. The programming model of -core is reactive, based on the familiar notions of concurrent tasks and (single-unit) resources. The language is kept minimalistic, capturing the static task, communication, and resource structure of the system. Whereas C-source can be arbitrarily embedded in the model (and/or externally referenced and linked) the step to mainstream development is minimal and smooth transition of legacy code is possible. A prototype compiler implementation for the -core language has been developed (implemented in the OCaml language [1], using the modern Menhir tool [2] for parser generation). The compiler generates C-code output that compiled together with the RTFM-kernel primitives [3] runs on bare metal. The RTFM-kernel guarantees deadlock-lock free execution and efficiently exploits the underlying interrupt hardware for static priority scheduling and resource management under the Stack Resource Policy [4]. This allows a plethora of well-known and state of the art methods for static verification (response time analysis, stack memory analysis, etc.) to be readily applied.

Another target for the C-code output of the -core compiler is the thread based run-time system RTFM-RT [5]. Different from the RTFM-kernel, it executes -core programs on mainstream and widespread thread based operating systems like Windows, Linux, and Mac OS X.

A common construction of a compiler is shown in Figure 1. Its front-end creates an internal representation of the input text. The structure used for that is called abstract syntax tree (AST). Subsequently, the back-end emits the output text from this AST.

1.1 Goal

Our primary goal is to demonstrate the potential of compilation techniques to address domain specific challenges. Settings of CPS/IoT typically involve requirements on robustness, reactive real-time performance, and power efficiency of the target devices. To this end, the specific design choices for the -core language, spanning from programming model, compiler design issues, back-end tools, and even run-time

This work is based on an earlier work: RTFM-core: Course in Compiler Construction, in Workshop on Embedded and Cyber-Physical Systems Education, WESE, (c) ACM, 2014. http://dx.doi.org/10.1145/2829957.2829962.

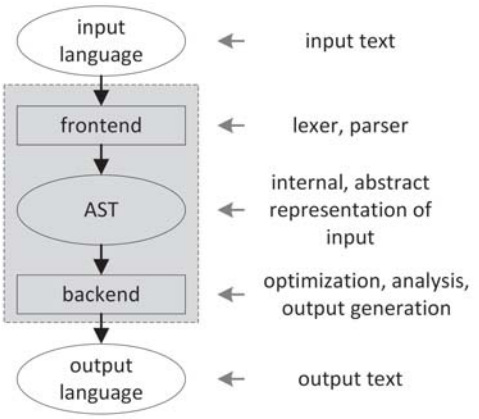

Figure 1: Common Compiler Architecture.

environments are motivated and scrutinized. The -core compiler allows the students direct hands-on experience. Due to its simplicity, the discussed problems and solutions can be presented and demonstrated in terms of their concrete implementations. The complete compiler including common definitions (`Common.ml`), error handling (`Error.ml`), command line parsing (`Cmd.ml`), source code lexing and parsing (`Lexer.mll/Parser.mly`), AST representation and pretty printing (`AST.ml`), and `Main.ml` amount in total only to a few hundred lines of code.

1.2 Method

We strongly believe that the succinctness of representation due to the functional language (OCaml) for the compiler implementation can be of great help to convey principles and facilitate understanding. To this end, our compiler implementation relies only on the standard libraries of OCaml, which are well designed, excessively proven by use, and very well documented. With that said, we are aware of alternative and additional libraries (like Core/Batteries), which may allow even more succinct, efficient, and elegant solutions. However, simplicity was the key for our decision and introducing additional libraries adds to the instep for understanding. One exception is the choice of Menhir. While being (to a high degree) backwards compatible to ocamlyacc, Menhir offers more flexibility and better error reporting and is to be considered as the preferred choice for new developments (as stated, e.g., in the excellent Real World OCaml textbook [6]).

As the RTFM-core compiler is developed using OCaml/Menhir, the students are exposed to the advantages of functional languages in the context of compiler construction, here used to implement an imperative language. For their own development, they may chose alternative design tools and languages (such as ANTLR/Java [7]). If doing so, they themselves have to cover the ground of re-implementing the necessary parts of the -cOOre compiler into their language of choice. A seemingly daunting task, however, the -cOOre implementation is even smaller than the -core compiler, so it is indeed feasible for the determined student.

1.3 Content

Of course, key content of the course aims at developing basic skills in the area of compiler construction and is not overlooked. The main part of the laboratory assignments is dedicated to manifesting theories and principles by hands on experience into extending a compiler for the (minimalistic) RTFM-cOOre Object Oriented language. W.r.t. code generation, the students target the RTFM-core language as an intermediate program representation. To this end, we step aside of the tradition of focusing low-level, register allocation methods, stack layout schemes, etc. This is a very deliberate choice based on the strong belief that students are more likely to deploy compilation techniques at much higher level (rather than new back-ends to non-supported target architectures or to compete with/improve on existing compiler back-ends).

Nevertheless, back-end issues and compiler optimizations are not left out of the course. On the contrary, they are important and discussed, but not in the traditional way. Programs in -core and the OO -cOOre (fronted) language are purely static. This hugely facilitates analysis and methods to program specialization (omitting costly pointer dereferencing during run-time) are introduced. Targeting lightweight embedded systems RAM is likely a sparse resource in comparison to FLASH/program memory storage. (E.g., the Cortex M3 LPC 1769 features 16/32 kb of RAM, and 512 kb of FLASH.) Hence, the duplication of code (due to specialization) can be argued for and is favorable not only w.r.t. speed (CPU cycles), but also register pressure (for the intermediate addresses during dereferencing), for function calls (omitting parameters where target addresses are known), etc. Also w.r.t. power consumption, the reduced number of clock cycles allows the processor to return to low-power operation earlier. In the case of off-chip memory referencing, memory operations per-se may also be a factor to the overall power consumption.

1.4 Student Qualification

Students are expected to have good knowledge of imperative and object-oriented programming, discrete mathematics (functions and relations, set theory, state automata), and algorithms (searching and sorting, common data structures like queues, stacks, lists, trees, and graphs), as well as basic skills in real-time systems and micro-computer engineering (specifically stack-based assembly programming and analysis of automatically generated assembly code from C).

1.5 Syllabus Overview

In brief, lecture 1 gives an introduction to compilers in general and the challenges specific to CPS/IoT settings, such as resource constrained targets, support for reactivity/concurrency, and resource management as well as analyzability w.r.t. to resource usage, e.g., memory and power, as well as real-time properties.

Lectures 2-10 cover basics of compiler construction, all the way from parsing and static analysis (well-formedness etc.) to code generation. Theory is followed closely by practice. The students work on extending the RTFM-cOOre language and compiler. The distinctive characteristic of both -cOOre and -core is their static nature. Hence, models can be put to static analysis and offer a highly predictable behavior, required in many CPS/IoT settings.

Lectures 11-15 introduce our prior, current, and future research directions related to the RTFM-lang development at LTU. RTFM-lang is in this context the collection of languages, run-time systems, and tools for analysis, developed

to address problems targeting challenges of embedded, real-time, and multi-core/multi-cpu software. In this way, the challenges stated in the course are revisited and students see how their own developments relate to the current state-of-the-art research in the field of embedded real-time systems. In contrast to standard courses in compiler technology, their own developed compiler is not just a mere toy compiler for a toy language, but rather a useful extension (and abstraction) to an already useful domain specific language. The challenges and concepts discussed towards the end of the course can all be related to their own developments, which hopefully may inspire students to dig deeper into the field.

2. SYLLABUS, LECTURE BY LECTURE

Each subsection amounts to a lecture or a seminar of the course. In this paper, we focus on the context and learning outcomes, not the detailed content.

The course is given as weekly lectures and seminars (15*90 minutes), accompanied by 6 laboratory assignments. Examination and grading is done on the evaluation of laboratory work and a final exam (given at the end of the course).

Required tools are to a large extent installed and configured by the students (which also adds to their skill set). Course material (background readings and references, slides, examples, as well as installation and configuration instructions) is offered online and made publicly available [8]. This allows students, engineers, and researchers worldwide to dig in. The RTFM-core compiler is intended to be free to use in non-commercial settings. In this way, the course material serves multiple purposes, on the one hand for educational purposes and on the other hand for engineers and researchers that like a deeper insight into the design choices behind the -core and -cOOre languages and their supporting tools (compilers and run-time systems).

2.1 Introduction

2.1.1 Background

The first lecture gives an introduction to compiler construction at large, covering language design (static and dynamic semantics), lexing and grammar rules formulation, intermediate (internal representation) compilation steps (well-formedness, etc), and code generation. The RTFM-core language and its compiler `rtfm-core` are used as a running example, relating each introduced concept to its concrete -core counterpart.

2.1.2 The RTFM-core language

The driving challenges of embedded software are presented and concepts of reactivity/concurrency, resource management, and real-time/resource analysis are covered in the context of -core and the RTFM-kernel. Key characteristics of -core are its static task, communication, and resource structure. This lends models/programs in -core exceptionally well to compile time analysis, as well as to compilation into efficient executables.

The complete tool chain, from -core source to bare metal executables, is demonstrated. Furthermore, driving challenges to multi-core and multi-CPU targets are presented and concepts of threads and parallelism are covered in the context of the run-time system RTFM-RT. It executes -core models by exploiting the available parallelism of the thread-based hosting environments Mac OS X, Linux, and Windows.

We choose RTFM-RT as the preferred run-time environment for the compiler construction course. However, students may choose to evaluate their developments under the bare metal target at will.

2.1.3 Assignment 1:RTFM-core

The students set up the tool-chain for RTFM-core. It consists of the OCaml/Menhir installation, a development environment (Eclipse OcaIDE plugin), and the graphviz toolsuite for visualizing task sets and resource utilization of -core programs.

Optionally, the back-end tools for executing RTFM-core on bare metal are installed. Internally we target the LPC1769 ARM-Cortex M3 MCU under the Eclipse based LPCXpresso environment.

First trials with the -core compiler are applied by the students and they explore the compiler options for source code analysis, as well as the debugging options provided by the back-end RTFM-RT and (optionally) debugging facilities provided by gdb under either RTFM-RT or the bare metal RTFM-kernel.

In order to finish the assignment, the students solve a set of control questions together with a given programming problem in -core. A set of given examples determines the control questions on syntax and semantics. The correctness of operation of the programming problem is shown by exploiting the compilers debugging options.

2.2 -core Compiler 1

2.2.1 OCaml from scratch 1

In order to efficiently work on the -core compiler, a gentle introduction to OCaml is given. For additional references, we suggest the Real World OCaml book (available free as web content [6]). It also includes a chapter on parsing using Menhir.

2.2.2 Lexing and Parsing

This lecture gives a brief introduction to the definition of a language (grammar), different parsing methods, and their strengths. Regular expressions, Menhir, and LR(1) parsing is more elaborately presented. We first look at some "textbook" examples and later go in further detail on the `core` source files `Lexer.ml`, `Parser.ml`, and `AST.ml` and the AST output from `core [- d_ast]`. In particular, nested parsing for inline C-code, single and multi-line comments require a gentle introduction.

Finally, we define a new statement `halt` to the RTFM-core language and see what necessary adaptations it requires to the compiler front-end.

2.3 -core Compiler 2

2.3.1 From front-end to back-end

Following the previous introduction to the compiler (front-end), we now follow the newly added `halt` statement through the succeeding compilation stages, making necessary additions all the way from compiler to code emission.

2.3.2 Assignment 2: Making a statement

Students define their own statement/built-in function, argue its use, and implement it in the compiler. For passing

the assignment, the compiler needs to emit corresponding code. For a higher grade, the added functionality needs to be supported in the graphical output. For the highest grade, dynamic semantics needs to be implemented and shown in DEBUG mode of the RTFM-RT or logged by the RTFM-kernel.

2.4 -cOOre 1

RTFM-cOOre is a minimalistic OO front-end to the -core language.

2.4.1 Syntax and Parsing of RTFM-cOOre

A grammar similar to MiniJava [9] (without inheritance at this point) is given to the students. MiniJava is popular in teaching compiler construction and has been the outset for the course previously given at LTU, which allows the re-use of some bread-and-butter material. Here we cover precedences and fixities (and exemplify on the expressions of RTFM-cOOre). We look at the dangling else problem [10].

A suggested AST is presented (in OCaml) and a set of example programs is introduced (good suit/bad suit that should be accepted/rejected by the parser).

2.4.2 Assignment 3: Parsing of -cOOre

The students start implementing their parsers for -cOOre.

2.5 -cOOre 2

2.5.1 Static Semantics

Here, we discuss the importance of compile time (static) analysis to verify correctness, resolving ambiguities (e.g., overloading of operators), and to guarantee soundness. Problems of unknown values and the general *halting* problem [11] are discussed.

2.5.2 Well-formedness of -cOOre

We discuss the type system of -cOOre and introduce inference rules for (a partially incomplete) well-formedness checking. Suggested data structures and OCaml for type checking is discussed (i.e, data types, associative lists/maps for environments). Simplicity is preferred to performance.

2.5.3 Assignment 4: Well-formedness

The students complete the well-formedness inference rules and their parser should now reject/pass the corresponding test suits.

2.6 -cOOre 3

Code generation is introduced.

2.6.1 Dynamic semantics of -cOOre

We present a mapping from the OO model to tasks and resources of the -core language. The mapping supports complete state encapsulation and state integrity, even in a concurrent setting. Through the implemented prototype compiler we demonstrate that -cOOre code can be (safely) executed by the RTFM-RT run-time. The prototype compiler is accessible to the students as a reference to which they can compare their compiler's behavior.

2.6.2 Code generation

We look in detail at the transformations required to go from -cOOre to -core.

2.6.3 Assignment 5: Code generation

At this point, the students start to implement code generation for the input language and iteratively enlarge the supported subset until the input language is covered.

2.7 -cOOre 4

We give a general introduction to object orientation [12] and inheritance and discuss alternative approaches.

2.7.1 Syntactic extensions

Syntax for inheritance for RTFM-cOOre is given.

2.7.2 Well-formedness criteria

The criteria for well-formedness is partially given in the form of inference rules. Checking it involves computing the transitive hull for cycles. A general algorithm in OCaml is presented and discussed.

2.7.3 Assignment 6: The -cOOre compiler

Completing the RTFM-cOOre compiler involves

1. completing all prior assignments (1-5),
2. implement parsing and type checking for the extended RTFM-cOOre language,
3. optionally completing the well-formedness inference rules to allow for inheritance and subtyping, and
4. optionally implement code generation including inheritance.

2.8 -cOOre 5

At this point, we discuss alternative approaches to implement inheritance and possibilities for optimization.

In particular, we discuss options for specialization of code, reducing or even completely removing the overhead of pointer dereferencing. A prototype implementation is presented and put to use, showcasing the benefits of specialization on a set of illustrative examples.

2.9 -cOOre + -core = TRUE

The RTFM-core language allows C-code to be embedded in tasks and functions. In this way, -core is a complete language (allowing to interact with- and operate on- the environment, e.g., memory and registers of the underlying platform or perform calls to the hosting environment).

In this lecture, we discuss the integration of native -core constructs into -cOOre models by introducing interface declarations. A prototype implementation is presented and the implementation is described.

2.10 Module systems

The -core language allows the inclusion and linking of external C code. However, external code is not seen as part of the -core model (as inclusion and linking is done outside the `core` compiler). In this lecture, we show an extension to a simplistic module system and its implication to scoping rules of the compiler. Furthermore, we discuss how -cOOre could be extended in a similar fashion.

2.11 Dynamic Object Instantiation

Programs/models in -core/-cOOre are purely static, with the advantage of analyzability and efficiency. This fits well

in the context of embedded systems, like CPS and IoT applications, where nodes are typically light-weight, resource constrained, and operating under real-time requirements. However, general purpose applications may be more conveniently expressed by allowing objects to be dynamically allocated. We discuss the implications both to compiler design and the supporting run-time environment. Specifically, we introduce the concepts of reference counting, copying collectors, and region based memory managers.

2.12 Memory Protection

The static semantics of -cOOre guarantees the property of complete state encapsulation, while the dynamic semantics enforces state integrity. Thus, in the ideal case of error free run-time systems, error free compilation of all software, and error free hardware, memory protection can be considered unnecessary. The question is whether we can trust this? In practice no, unless all involved software and hardware has been formally defined and proven correct (which is not yet the case). And even then, correctness of hardware operations is only provided to a degree of probability. Moreover, whenever allowing arbitrary C-code to be included (section 2.9), the risk of violating memory consistency is evident.

2.13 Mixed Criticality Systems

Criticality of functions in a mixed criticality setting may range from implying collateral damage, mission failure, or even just imply degraded quality of the perceived service. For many reasons, cost, power, space, or convenience, it may prove advantageous to share resources (e.g., CPU platform) between functions operating under different requirements. Traditionally, this is approached by separation, sand-boxing partitions into virtual execution environments. While relatively straightforward, this requires hardware support and a managing hypervisor. Thus, in the context of light-weight systems this is in many cases not a feasible solution.

Alternatively with support for memory protection (Section 2.12), we discuss the outsets from a language perspective. We review previous work on defining sections of criticality directly in the RTFM-language [13]. Such models, executed on bare metal by the RTFM-kernel primitives, are robust against overload and memory allocation failures caused by non-critical functions. Critical functions on their hand are implemented statically (hence have no memory allocations, thus cannot fail). Access to hardware is exclusive to the the critical parts. Thus, the degree of trust is only limited by the correctness of the tool chain, the kernel implementation, and the underlying hardware.

2.14 CompCert

RTFM-core relies heavily on correctness of the C code compilation. This is currently also the case for RTFM-cOOre, but the generated code is under full control of the compiler allowing solutions where we generate assembly or even machine code directly. Another approach is to turn to general C compilers that provide at least some degree of trust. To this end, the CompCert C compiler (ccomp, [14]) has an outstanding position. The complex compilation stages from preprocessing to the generation of abstract assembly syntax is purely defined in Coq [15] and has been verified through machine checked proofs. The compiler itself is built from automatically extracted code (generated from the Coq definitions) together with wrapping code (written in OCaml) for file I/O, compiler options, and conversion from abstract assembly to text.

In this lecture, we review our recent work on extending the CompCert C compiler to generate RTFM-kernel executables for the modern ARM Cortex family by porting the already existing ARMv3 back-end to the ARMv6/7m *thumb2* instruction set. Additionally, we discuss our added language built-ins that gives the programmer full control of the memory consistency and show how they are used to guarantee integrity of critical sections scheduled by the RTFM-kernel.

2.15 RTFM-4-FUN

In the last lecture, we showcase how the RTFM-kernel and the -core language can be used as intermediate representations in the context of model based design using Function Blocks. We review our recent work on mapping device level models in the IEC 61499 standard [16] to the task and resource model of **RTFM** [17]. Our mapping allows properties of resource usage and real-time behavior to be verified at compile time. The mapping is implemented by the RTFM-4-FUN compiler, which translates IEC 61499 (XML format) models to C code and RTFM-kernel primitives. Currently, Service Interface Function Block (SIFB) semantics is undefined in IEC 61499, hence verification of IEC 61499 models builds on mere assumptions or external models of their behavior. We show how the RTFM-4-FUN compiler can be altered in a straightforward way to produce -core/-cOOre code. This allows arbitrary mixing of **RTFM** models with function blocks and provides both semantics and a language for implementation of SIFBs. In this way, we provide a single semantic underpinning to the reasoning on IEC 61499 models at device level.

2.16 Course wrap-up

An additional lecture is given, wrapping up the course and address questions. Together with the students, we share and discuss our experiences and opinions interactively. Additional course evaluation is performed according to the university's rules and regulations.

3. METHODS AND TOOLS

Students get access to a git archive from all sources (including updates and bug-fixes). The process of setting up and use an Eclipse environment (with git and OCaml support) is explained and help is available. We anticipate this to greatly facilitate both our work (supervising the students) and their own developments. Furthermore, it also gives the students exposure and experience to modern teamwork environments and engineering processes.

3.1 The -core compiler

The RTFM-core language (grammar is depicted in Figure 2) is designed from the outset of simplicity, with a clear focus on constructs for concurrency and real-time operations.

The `CCode` terminal denotes the presence of embedded C-code in the language, occurring either at the top level or inside a `ISR/Task/Func/Reset` statement. Each `ISR/Task` is associated with a static integer priority, while `Func`'s merely facilitate modularization. The `claim` is recursively defined (allowing nesting of resource claims).

The students are exposed to the -core compiler implementation in OCaml. They are given the complete source for the

```
SYNTAX  Top ::= #> CCode <#
              | ISR Id Int{Stmt}
              | Task Id Int{Stmt}
              | Func Id( CCode){Stmt}
              | Reset {Stmt}
              | Top Top

SYNTAX  Stmt ::= #> CCode <#
               | claim Id{Stmt}
               | pend Id ;
               | sync Id( CCode) ;
               | Stmt Stmt
```

Figure 2: Grammar for RTFM-core.

working compiler and we scrutinize its design and implementation.

Listing 1 depicts the Menhir (yacc like) grammar, which is succinct and easily understandable (while being close to the EBNF given) and listing 2 its corresponding AST. This gives us an outset to discuss the principles for parsing and trade-offs (strength vs. complexity) and demonstrate it in context of the Menhir LR(1) tool.

```
prog:
  | top* EOF                        {Some (Prog ($1))}

top:
  | CCODE                           {TopC ($1)}
  | ISR ID INTVAL LC stmt* RC       {Isr (HARD, $2, $3, $5)}
  | TASK ID INTVAL LC stmt* RC      {Isr (SOFT, $2, $3, $5)}
  | FUNC ID ID PARAMS LC stmt* RC {Func ($2, $3, $4, $6)}
  | RESET LC stmt* RC               {Reset ($3)}

stmt:
  | CCODE                           {ClaimC ($1)}
  | CLAIM ID LC stmt* RC            {Claim ($2, $4)}
  | PEND ID SC                      {Pend ($2)}
  | SYNC ID PARAMS SC               {Sync ($2, $3)}
```

Listing 1: OCaml Menhir parser `Parser.mly`.

```
type stmt =
  | Claim     of string * stmt list
  | Pend      of string
  | Sync      of string * string
  | ClaimC    of string

type isr_type = HARD | SOFT

type top =
  | TopC      of string
  | Isr       of isr_type * string * int * stmt list
  | Func      of string * string * string * stmt list
  | Reset     of stmt list

type prog =
  | Prog      of top list
```

Listing 2: OCaml `AST.ml`.

The lexing part is more challenging. Listing 3 depicts an excerpt of the `Lexer.mll`. Besides introducing regular expressions and ordinary lexing rules, we cover nested parsing (for the multi-line comments and C inlining), as well as error reporting.

```
{
 (* tokens and dependencies *)
 type token =
   | PEND
... etc
   | ID of string
... etc
   | CCODE of string
   | PARAMS of string
   | SC
... etc

 open Parser
 open Lexing
... etc
 exception SyntaxError of string
}

(* regular expressions (regexps) *)
let white   = [' ' '\t']+
... etc
let enter_c = "#>"
let exit_c  = "<#"
let params  = ( [^ '*' ')'] [^ ')']* )?

(* lexing rules *)
rule lex = parse
  | "pend"                {PEND}
... etc
  | enter_c               {set_info lexbuf;
                           c (Buffer.create 100) lexbuf}
  | white                 {lex lexbuf}
  | newline               {next_line lexbuf; lex lexbuf}
  | "//"                  {set_info lexbuf;
                           comment lexbuf}
  | "(*"                  {set_info lexbuf;
                           comments 0 lexbuf}
  | '('(params as p) ')'  {PARAMS (p)}
  | eof                   {EOF}
  | _                     {raise Parser.Error}

and comment = parse
  | newline               {next_line lexbuf; lex lexbuf}
  | eof                   {EOF}  (* // at last line is OK*)
  | _                     {comment lexbuf;}

and comments level = parse
  | "*)"                  {if level = 0 then lex lexbuf
                           else comments (level-1) lexbuf}
  | "(*"                  {comments (level+1) lexbuf}
  | newline               {next_line lexbuf;
                            comments level lexbuf}
  | _                     {comments level lexbuf}
  | eof                   {bol lexbuf;
                           raise (SyntaxError("(*..."))}

and c buf = parse
  | exit_c                {CCODE (Buffer.contents buf)}
  | newline               {next_line lexbuf;
                           Buffer.add_string buf
                             (Lexing.lexeme lexbuf);
                           c buf lexbuf}
  | enter_c               {raise (SyntaxError("#>␣nested"))}
  | _                     {Buffer.add_string buf
                             (Lexing.lexeme lexbuf);
                           c buf lexbuf}
  | eof                   {bol lexbuf;
                           raise (SyntaxError("#>..."))}
```

Listing 3: OCaml lexer `Lexer.mll`.

Common problems to lexing and parsing are brought up and the design choices for the -core compiler are discussed. E.g., by choice of simplicity, params and C code are stored and processed literally, hence is not *understood* by the compiler. This is fine, since potential errors are spotted and reported by the back-end C compiler. In this context, -core can be seen as an advanced pre-processor for the C language.

Similarly, we cover the processing of the AST into target C code and the well-formedness checks done at this stage (validating -core specific constructs). Compiler command line processing and other commodity parts of the compiler are left to the students to investigate and understand on their own hand.

3.1.1 -core programs and run-time

The students have access to the source code of a set of

example -core programs, as well as run-time systems executing -core models on bare metal, POSIX environments (OSX/Linux), and Win32 platforms. Exposure is ensured both through demonstrations (at lectures throughout the course) and through their own lab developments.

3.2 The -cOOre compiler

The basic -cOOre language (Figure 3) and corresponding compiler source is given to the students. The grammar is defined such that a 1-1 mapping to Menhir grammar and AST is possible. Likely a more elegant grammar could be presented, but it may somewhat complicate the mapping to concrete rules for parsing, so we opted for simplicity in this case.

Each program defines a `Root` class with a `Reset` construct. `Root` and its inner objects are instantiated at compile time, while `Reset` is invoked by the run-time on startup. The static structure allows a simple mapping to -core (`Task`, `ISR`, and `Reset` constructs).

An object instance o amounts to a -core resource r_o and a method m amounts to a `Func` construct that executes statements s within a -core **`claim`** `r_o { s }`. In this way, state integrity is ensured by enforcing mutual exclusion between methods executing on a common object. Method invocations and pending of tasks amount to corresponding -core **`sync`**/**`pend`** statements.

The simple design allows for compile time instantiation of the complete model with zero overhead for class arguments, as well as complete specialization of methods.

```
SYNTAX  Prog ::= ClassDefs

SYNTAX  ClassDef ::= class Id < ClassArgs > {ClassDecls}

SYNTAX  ClassArg ::= PType Id
                   | PType Id(MSigs)

SYNTAX  PType ::= int
                | bool
                | char
                | byte
                | void

SYNTAX  MArg ::= PType Id

SYNTAX  ClassDecl ::= PType Id := Expr ;
                    | Id < Params > Id ;
                    | PType Id(MArgs){Stmts}
                    | Task Id Int{Stmts}
                    | ISR Id Int{Stmts}
                    | Reset {Stmts}

SYNTAX  Expr ::= Pend Id
               | Pend Id . Id
               | Id
               | Id(Params)
               | Id . Id
               | Id . Id(Params)
               | Int
               | Bool
               | Char
               | RT_rand (Expr)

SYNTAX  Stmt ::= Expr ;
               | PType Id := Expr ;
               | Id := Expr ;
               | return Expr ;
               | RT_sleep (Expr) ;
               | RT_printf (String, Params) ;
               | RT_printf (String) ;

SYNTAX  ClassDecls ::= List{ClassDecl, ""}

... ClassArgs, ClassArgs, Params, MArgs, MSigs, Stmts
```

Figure 3: Grammar for RTFM-cOOre.

3.3 Assignments

Assignments were carried out in groups (typically 2 students in each group). Given the -core and -cOOre language definitions and corresponding implementations, the students first make themselves comfortable with the -core and -cOOre languages.

3.3.1 Assignment 1: core/-cOOre first experiences

As a first entry point, the tools are available online and executing through a web interface. This allows students (and people around the globe) to access and play with the languages without installing any code/tools locally. Actually, executing -core/-cOOre models is a bit trickier to offer as an online service, since (at least in the -core language) user's may enter arbitrary C code in the models and thus executing these remotely imply security issues. To this end, precompiled tools (compilers and run-time systems) are offered for local installation under OSX/Linux and Win32 environments.

As part of the warmup, students install the compilers from source and see that they get expected output (in comparison to the pre-compiled and online compilers).

3.3.2 Assignment 2: Extending -core

The second assignment is to extend the -core language with an additional `halt` statement (following the lecture). This involves the lexer, parser, AST, and code generation. Students then suggest and implement their own -core statement.

3.3.3 Assignment 3: Extending -cOOre

The third assignment is related to more complex parsing problems. An extended grammar (not depicted in this paper) is given to the students and they need to deal with Menhir to solve occurring ambiguities. The grammar introduces operations to the primitive data types and introduce (single dimensional) arrays.

3.3.4 Assignment 4: Well-formedness of -cOOre

The students implement an additional pass for well-formedness check for -cOOre models, encoding scopes/environments (e.g., through associative lists in OCaml). More advanced features, such as tracking errors back to source, can be implemented by the students towards earning a higher grade.

3.3.5 Assignment 5: Code generation

The students implement code generation for the extended -cOOre language. More advanced features, such as ensuring safe array indexing, can be implemented towards earning a higher grade.

3.3.6 Assignment 6: OO addition to -cOOre

The first 5 assignments follow closely the lecture structure and students should now be able to work more independently on extending the language with single inheritance OO features. The requisite for pass is a working front-end and well-formedness check. Code generation with complete specialization can be done towards earning a higher grade.

3.3.7 Bonus Assignment

Once the requirements for passing the course are fulfilled, students may revisit prior labs and earn extra points for final evaluation. Alternative concepts that are discussed from lecture 8 and on can be worked on, in particular, module system and dynamic object instantiation seems within reach to ambitious students.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we present the new syllabus for the compiler construction course of the Computer Science masters program at the Luleå University of Technology. The course aims to focus domain specific language design and compilation techniques for resource constrained devices common to CPS and IoT applications in the embedded systems domain. Concepts of lexing, parsing, well-formedness, and code generation are covered, but put in a broader concept, taking the programming model and even the underlying run-time system into consideration. In this way, we expect that students gain good fundamental skills as well as valuable insights in the potential of compilation techniques to solve domain specific problems.

Until now, the new course was given just one time during the fall of 2014. Among the 16 enrolled students there was a large diversity of backgrounds. Some of them had no background in real-time systems at all. We also encountered students without prior knowledge about functional programming.

The course showed a very good throughput. All but one group completed the mandatory assignments. The students took the chance to individualize their course assignments in various directions. Two groups used Haskell and one group used SCALA for their implementation of the -cOOre compiler, thus they had to implement the whole compiler from scratch. Another group opted for developing editor support instead of working on the language design.

The main reason for avoiding OCaml for the compiler implementation was prior knowledge in Haskell and SCALA of the students due to previous courses at the university. Implementing the whole compiler from scratch was considered easier than learning the differences in both languages, even though both follow the functional paradigm.

Prior exposure to functional programming as a prerequisite would be beneficial for the compiler construction course. A more thorough introduction to functional programming with OCaml based on the excellent introductory slides by Stephen A. Edwards [18] could then be another option for improvement.

In contrast to our suggestion to use the Eclipse plugin, almost all students worked with the extendable highlighting text editor Sublime. Due to the simplicity of the source languages -core/-cOOre and their compilers, a full-fledged development environment was considered too distractive by the students. A simple text editor with text highlighting and command line tools have been enough to focus on the topics.

Next time given, we plan to discuss a proposal for -cOOre allowing objects to be dynamically instantiated. A prototype implementation of a simplistic (non-optimizing) reference counting collector will be given, as well as the corresponding syntactic additions and new inference rules for well-formedness will be highlighted.

Recent work at LTU investigated how resources in the -core language can be protected by exploiting underlying hardware. As a result, memory protection support has been integrated in -core and its compiler, which will be presented in upcoming course cycles to improve the topics of Section 2.12.

Acknowledgments

The authors acknowledge Svenska Kraftnät (Swedish national grid) and EU ARTEMIS JU (EMC2) for funding this research.

5. REFERENCES

[1] OCaml. (webpage) Last accessed 2014-07-28. [Online]. Available: http://ocaml.org

[2] Menhir. (webpage) Last accessed 2014-08-07. [Online]. Available: http://gallium.inria.fr/~fpottier/menhir/

[3] J. Eriksson, F. Haggstrom, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives." in *SIES*. IEEE, 2013, pp. 110–113.

[4] T. Baker, "A stack-based resource allocation policy for realtime processes," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, Dec. 1990, pp. 191 –200.

[5] A. Lindner, M. Lindner, and P. Lindgren, "RTFM-RT: a threaded runtime for RTFM-core - towards execution of IEC 61499," in *Emerging Technologies Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE, 2015.

[6] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml: Functional programming for the masses*, 1st ed. O'Reilly Media, 11 2013.

[7] ANTLR (ANother Tool for Language Recognition). (webpage) Last accessed 2014-08-07. [Online]. Available: http://www.antlr.org

[8] RTFM: Real-Time For the Masses. (webpage) Last accessed 2014-07-28. [Online]. Available: http://www.rtfm-lang.org

[9] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. Cambridge University Press, 10 2002.

[10] P. W. Abrahams, "A final solution to the dangling else of algol 60 and related languages," *Commun. ACM*, vol. 9, no. 9, pp. 679–682, Sep. 1966.

[11] A. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceeding of the London Mathematical Society*, 1936.

[12] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*, 1st ed. Addison-Wesley Professional, 6 1992.

[13] P. Lindgren, D. Pereira, J. Eriksson, M. Lindner, and L. M. Pinho, "RTFM-lang Static Semantics for Systems with Mixed Criticality," in *Ada-Europe 2014: 19th International Conference on Reliable Software Technologies*, 2014.

[14] CompCert. (webpage) Last accessed 2014-08-06. [Online]. Available: http://compcert.inria.fr

[15] The Coq Proof Assistant. (webpage) Last accessed 2014-08-06. [Online]. Available: http://www.lix.polytechnique.fr/coq/

[16] "International Standard IEC 61499: Function Blocks - Part 1, Architecture," Geneva, Switzerland: Int. Electrotech. Commission, 2005.

[17] P. Lindgren, M. Lindner, A. Lindner, J. Eriksson, and V. Vyatkin, "RTFM-4-FUN," in *SIES*. IEEE, 2014.

[18] S. A. Edwards, "An introduction to objective caml," 2010. [Online]. Available: http://www.cs.columbia.edu/~sedwards/classes/2012/w4115-spring/ocaml.pdf