LINKÖPINGS UNIVERSITET

Compiler Frameworks and Compiler Generators

A (non-exhaustive) survey

with a focus on open-source frameworks

Overview

- Part I – Syntax-Based Generators
- Part II – Semantics-Based Generators
- Part III – Primarily Back-End Frameworks and Generators
- Part IV – More Frameworks

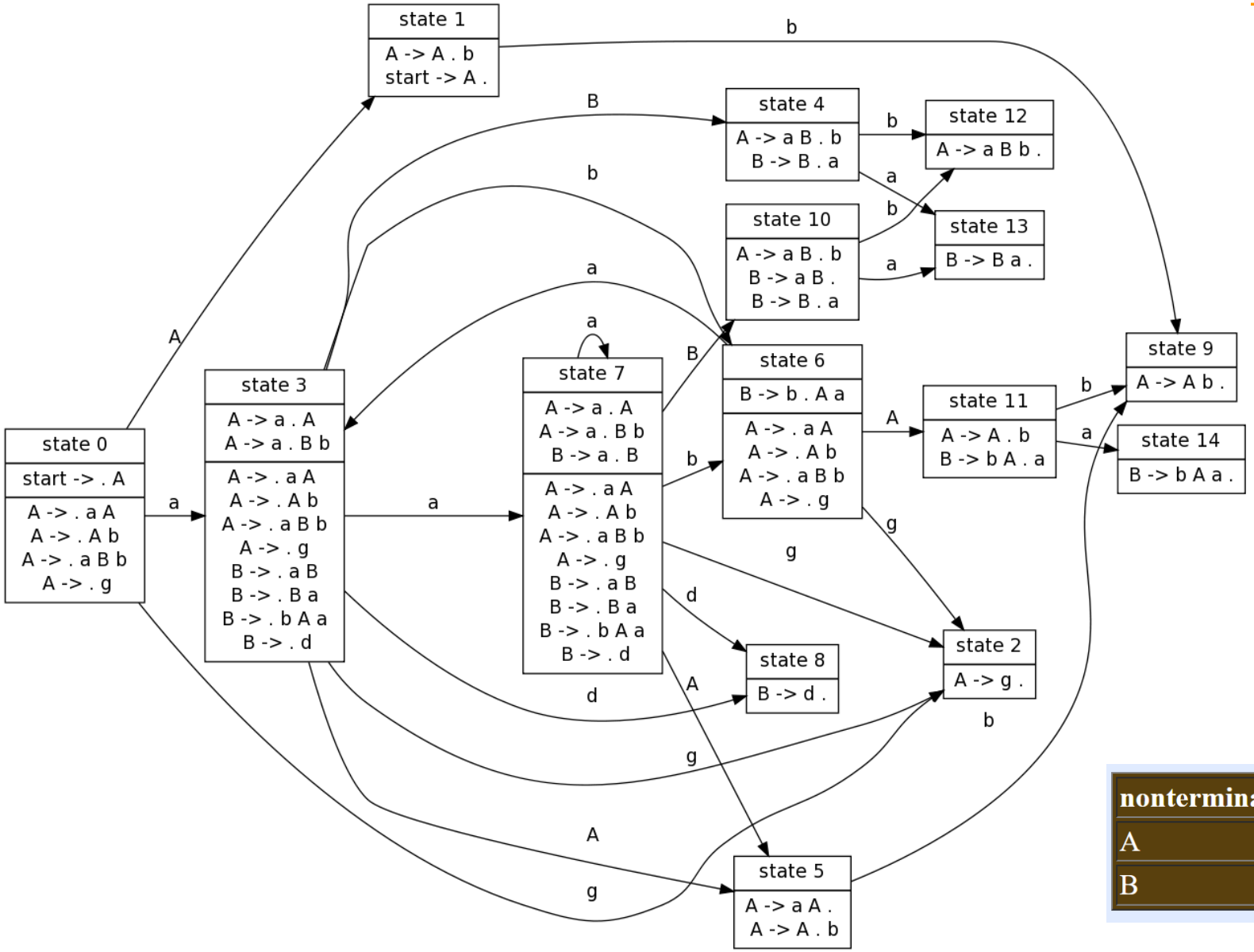
LINKÖPINGS UNIVERSITET

Part I

Syntax-Based Generators

Grammar analysis tool (I)

https://smlweb.cpsc.ucalgary.ca/start.html



Grammar

A → a A
| A b
| a B b
| g.
B → a B
| B a
| b A a
| d.

nonterminal	first set	follow set
A	a g	b a
B	a b d	b a

Grammar analysis tool (II)

https://smlweb.cpsc.ucalgary.ca/start.html

LR(0) Table

	$	d	a	b	g	A	B
0			s3		s2	s1	
1	acc	acc	acc	acc/s9	acc		
2	r(A → g)	r(A → g)	r(A → g)	r(A → g)	r(A → g)		
3		s8	s7	s6	s2	s5	s4
4			s13	s12			
5	r(A → a A)	r(A → a A)	r(A → a A)	r(A → a A)/s9	r(A → a A)		
6			s3		s2	s11	
7		s8	s7	s6	s2	s5	s10
8	r(B → d)	r(B → d)	r(B → d)	r(B → d)	r(B → d)		
9	r(A → A b)	r(A → A b)	r(A → A b)	r(A → A b)	r(A → A b)		
10	r(B → a B)	r(B → a B)	r(B → a B)/s13	r(B → a B)/s12	r(B → a B)		
11			s14	s9			
12	r(A → a B b)	r(A → a B b)	r(A → a B b)	r(A → a B b)	r(A → a B b)		
13	r(B → B a)	r(B → B a)	r(B → B a)	r(B → B a)	r(B → B a)		
14	r(B → b A a)	r(B → b A a)	r(B → b A a)	r(B → b A a)	r(B → b A a)		

SLR(1) Table

	$	d	a	b	g	A	B
0			s3		s2	s1	
1	acc			s9			
2	r(A → g)		r(A → g)	r(A → g)			
3		s8	s7	s6	s2	s5	s4
4			s13	s12			
5	r(A → a A)		r(A → a A)	r(A → a A)/s9			
6			s3		s2	s11	
7		s8	s7	s6	s2	s5	s10
8			r(B → d)	r(B → d)			
9	r(A → A b)		r(A → A b)	r(A → A b)			
10			r(B → a B)/s13	r(B → a B)/s12			
11			s14	s9			
12	r(A → a B b)		r(A → a B b)	r(A → a B b)			
13			r(B → B a)	r(B → B a)			
14			r(B → b A a)	r(B → b A a)			

The grammar is not LR(0) because

- shift/reduce conflict in state 1.
- shift/reduce conflict in state 5.
- shift/reduce conflict in state 10.

Grammar

```
A → a A
  | A b
  | a B b
  | g.
B → a B
  | B a
  | b A a
  | d.
```

The grammar is not SLR(1) because

- shift/reduce conflict in state 5.
- shift/reduce conflict in state 10.

Syntax-Based Generators

- Lex and Flex – generate lexical analyzers.
  - Clones and/or open-source alternatives exist for many programming languages. Wikipedia has a reasonable overview.
- Yacc and Bison – generate parsers
  - Can be used for syntax-directed translation
  - Usually syntax-directed translation is not used (if the compilation is not completely driven by the parser, it is something else)
  - Does not generate semantic analysis, intermediate code, optimization, or code generation
  - YACC/Bison produces parsers that are bad at error management
- Very many alternatives exist, with the grammar specification either using an API in the programming language, EBNF, or something else. Many parser generators (such as ANTLR) allow the user to adapt the error handling routines. Some also have IDE's that make debugging your grammar easier.
- https://en.wikipedia.org/wiki/Comparison_of_parser_generators

EBNF Evaluator

https://mdkrajnak.github.io/ebnftest/

LINKÖPINGS UNIVERSITET

EBNF Evaluator

github project

EBNF Grammar

```
program ::= 'PROGRAM', white_space, identifier, white_space,
          'BEGIN', white_space,
          { assignment, ";", white_space },
          'END.'
identifier ::= alphabetic_character, { alphabetic_character | digit }
number ::= [ "-" ], digit, { digit }
string ::= "'" , { all_characters }, "'"
assignment ::= identifier , ":=" , ( number | identifier | string )
alphabetic_character ::= "A" | "B" | "C" | "D" | "E" | "F" | "G"
                       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                       | "V" | "W" | "X" | "Y" | "Z" ;
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
white_space ::= (' ' | '\n')*
all_characters ::= (#'[A-Za-z][A-Za-z0-9_+ !]'* | #' '*)*
```

Test Input

```
PROGRAM DEMO1
BEGIN
  A:=3;
  B:=45;
  H:=-100023;
  C:=A;
  D123:=B34A;
  BABOON:=GIRAFFE;
  TEXT:='Hello world!';
END.
```

Test Output

```
The test input is valid.
```

Test Input

```
PROGRAM DEMO1
BEGIN
  A:=3;
  B:=45;
  H:=-100023;
  C:=A;
  D123:=B34A
  BABOON:=GIRAFFE;
  TEXT:='Hello world!';
END.
```

Test Output

```
Parse error at line 7 , column 13 :

  D123:=B34A
            ^
Expected one of:
";"
"9"
"8"
"7"
"6"
"5"
"4"
"3"
"2"
"1"
"0"
"Z"
"Y"
"X"
"W"
"V"
"U"
"T"
```

ANTLR example

http://lab.antlr.org/

https://www.antlr.org/tools.html

Lexer Parser Sample

```
1  parser grammar ExprParser;
2  options { tokenVocab=ExprLexer; }
3
4  program
5      : stat EOF
6      | def EOF
7      ;
8
9  stat: ID '=' expr ';'
10     | expr ';'
11     ;
12
13 def : ID '(' ID (',' ID)* ')' '{' stat* '}' ;
14
15 expr: ID
16     | INT
17     | func
18     | 'not' expr
19     | expr 'and' expr
20     | expr 'or' expr
21     ;
22
23 func : ID '(' expr (',' expr)* ')' ;
```

Input sample.expr

```
1  f(x,y) {
2      a = 3+foo;
3      x and y;
4  }
```

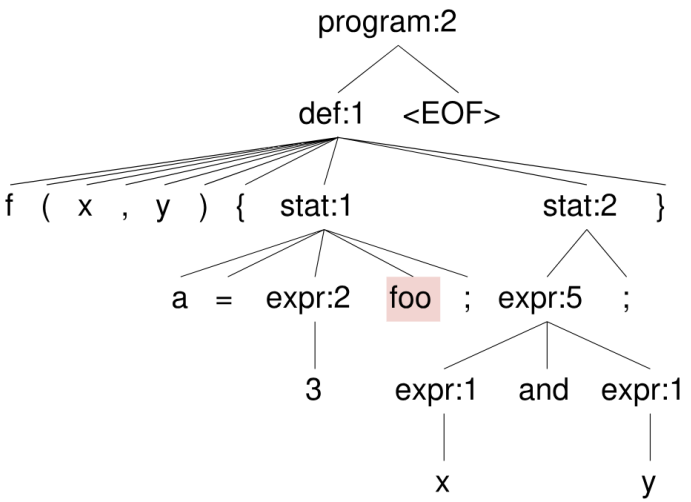
Start rule

program Run Show profiler

Parser console

```
2:9 token recognition error at: '+'
2:10 extraneous input 'foo' expecting ';'
```

Tree Hierarchy



LINKÖPINGS UNIVERSITET

Part II

Semantics-Based Generators

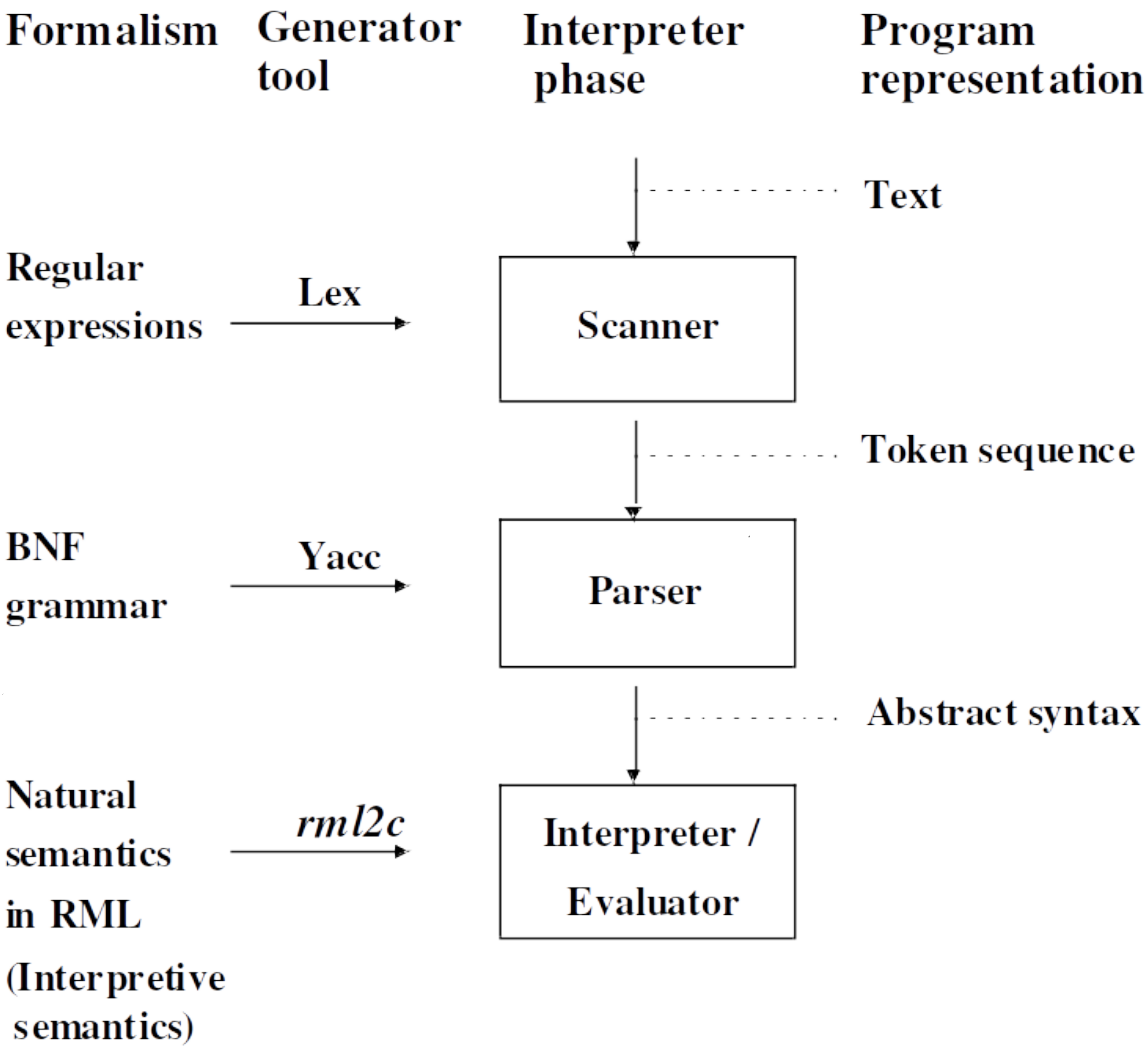
RML – Compiler Generation from NS

A Compiler Generation System and Specification Language from Natural Semantics/Structured Operational Semantics

- Goals
  - Efficient code – comparable to hand-written compilers
  - Simplicity – simple to learn and use
  - Compatibility with “typical natural semantics/operational semantics” and with Standard ML
- Properties
  - Deterministic
  - Separation of input and output arguments
  - Statically strongly typed
  - Polymorphic type inference
  - Efficient compilation of pattern-matching
- https://www.ida.liu.se/labs/pelab/rml/
  - developed around 1999 and used in OpenModelica until 2014-10-25

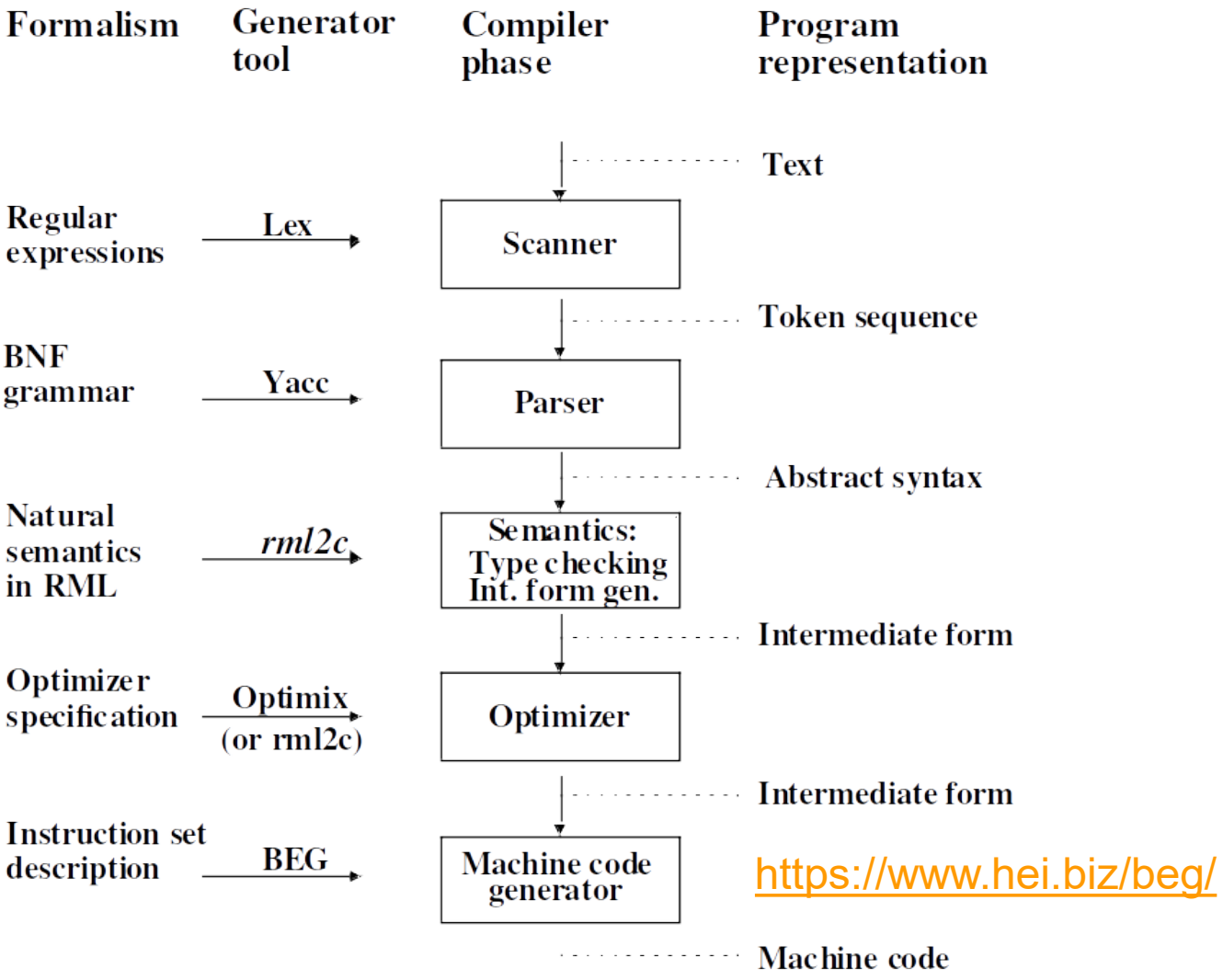
Generating an Interpreter

Generating an Interpreter Implemented in C, using rml2C



Generating a Compiler

Generating a Compiler Implemented in C, using rml2C



RML Syntax

- Goal: Eliminate plethora of special symbols usually found in Natural Semantics/Operational Semantics specifications
- Software engineering viewpoint: identifiers are more readable in large specifications
- A Natural/Operational semantics rule

```
H1 |- T1 : R1   . .   Hn  |- Tn : Rn
-------------------------------------
if <cond>
H |- T : R
```

- A typical RML rule

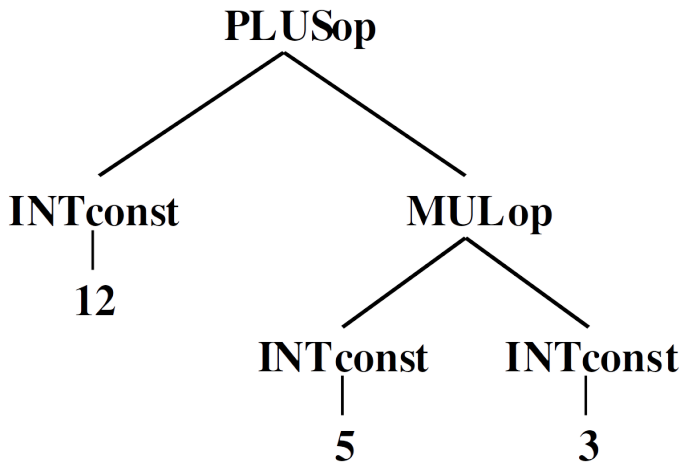
```
rule NameX ( H1 , T1 ) = > R1 &
     . . .
     NameY ( Hn , Tn ) = > Rn &
     < cond >
     ------------------------------
     RelationName ( H , T ) = > R
```

Example: The Exp1 Language Definition

- Typical expressions

  12 + 5 * 3

  -5 * (10 - 4)



Abstract Syntax Tree of

12 + 5*3

- Abstract syntax (defined in RML)

```
datatype Exp
= INTconst of int
| PLUSop of Exp * Exp
| SUBop of Exp * Exp
| MULop of Exp * Exp
| DIVop of Exp * Exp
| NEGop of Exp
```

Example: The Exp1 Evaluator

```
(* file exp1.rml *)

(* Abstract syntax of the language Exp1 *)

module exp1:
  datatype Exp =  INTconst of  int
                |  PLUSop    of  Exp * Exp
                |  SUBop     of  Exp * Exp
                |  MULop     of  Exp * Exp
                |  DIVop     of  Exp * Exp
                |  NEGop     of  Exp
  relation eval:  Exp => int
end
```

- Evaluation of an addition node PLUSop is v3, if v3 is the result of adding the evaluated results of its children e1 and e2.
- Subtraction, multiplication, division operators have similar specifications.

```
(* Evaluation semantics  of Exp1 *)

relation eval: Exp => int  =

  axiom eval( INTconst(ival) ) => ival    (* eval of an integer node *)
                                          (* is the integer itself *)

  (* Evaluation of an addition node PLUSop is v3, if v3 is the result of
   * adding the evaluated results of its children e1 and e2
   * Subtraction, multiplication, division operators have similar specs.
   *)

  rule  eval(e1) => v1  &  eval(e2) => v2  &  int_add(v1,v2) => v3
        ------------------------------------------------------------
        eval( PLUSop(e1,e2) ) => v3

  rule  eval(e1) => v1  &  eval(e2) => v2  &  int_sub(v1,v2) => v3
        ------------------------------------------------------------
        eval( SUBop(e1,e2) ) => v3

  rule  eval(e1) => v1  &  eval(e2) => v2  &  int_mul(v1,v2) => v3
        ------------------------------------------------------------
        eval( MULop(e1,e2) ) => v3

  rule  eval(e1) => v1  &  eval(e2) => v2  &  int_div(v1,v2) => v3
        ------------------------------------------------------------
        eval( DIVop(e1,e2) ) => v3


  rule  eval(e) => v1  &  int_neg(v1) => v2
        ----------------------------------
        eval( NEGop(e) ) => v2

end (* eval *)
```

Lookup in Environments

```
relation lookup : ( Env , Ident ) = > Value =

  (* lookup returns the value associated with an identifier.
   If no association is present, lookup will fail.
   Identifier id is found in the first pair of the list,
   and value is returned. *)
  rule id = id2
       --------------
       lookup((id2, value) :: _, id) = > value

  (* id is not found in the first pair of the list, and lookup will
   recursively search the rest of the list. If found, value is
   returned.* )
  rule not id = id2 & lookup(rest, id) = > value
       ----------------------------------------
       lookup((id2, _) :: rest, id) = > value
end

(* NOTE : Searching linked lists is slow, no fancy HT in RML *)
```

Translational Semantics of the PAM language Abstract Syntax to Machine Code

- PAM example program

```
read x, y;
while x <> 99 do
   ans := (x+1)-(y/2);
   write ans ;
   read  x, y
end
```

- Simple Machine Instruction set

```
LOAD   Load accumulator
STO    Store
ADD    Add
SUB    Subtract
MULT   Multiply
DIV    Divide
GET    Input a value
PUT    Output a value
J      Jump
JN     Jump on negative
JP     Jump on positive
JNZ    Jump on negative or zero
JPZ    Jump on positive or zero
JNP    Jump on negative or positive
LAB    Label ( no operation )
HALT    Halt execution
```

PAM Example Translation

- PAM example program

```
read x, y;
while x <> 99 do
   ans := (x+1)-(y/2);
   write ans ;
   read  x, y
end
```

- Translated machine code assembly text

```
    GET   x          STO   T2
    GET   y          LOAD  T1
L1  LAB              SUB   T2
    LOAD  x          STO   ans
    SUB   99         PUT   ans
    JZ    L2         GET   x
    LOAD  x          GET   y
    ADD   1          J     L1
    STO   T1   L2    LAB
    LOAD  y          HALT
    DIV   2
```

- Low level representation tree form

```
MGET(    I(x) )   MSTO(    T(2) )
MGET(    I(y) )   MLOAD(   T(1) )
MLABEL(  L(1) )   MB(MSUB,T(2) )
MLOAD(   I(x) )   MSTO(    I(ans) )
MB(MSUB,N(99))    MPUT(    I(ans) )
MJ(MJZ,  L(2) )   MGET(    I(x) )
MLOAD(   I(x) )   MGET(    I(y) )
MB(MADD,N(1) )    MJMP(    L(1) )
MSTO(    T(1) )   MLABEL(  L(2) )
MLOAD(   I(y) )   MHALT
MB(MDIV,N(2) )
```

Some Applications of RML

- Small functional language with call-by-name semantics (mini-Freja, a subset of Haskell)
- Almost full Pascal with some C features (Petrol)
- Mini-ML including type inference
- Specification of full Java 1.2
- Specification of Modelica 2.0

primes	Typol	RML	Typol/RML
3	13s	0.0026s	5000
4	72s	0.0037s	19459
5	1130s	0.0063s	179365

Mini-Freja Interpreter performance compared to Centaur/Typol

Some Attribute-Grammar Based Tools

- JastAdd – A meta-compilation system
  - https://jastadd.org
  - Supports Reference Attribute Grammars (RAGs)
  - Modelica tools – Modelon Impact (former JModelica.org)
  - Java compiler – ExtendJ

- Ordered Attribute Grammars
  - Uwe Kastens, Anthony M. Sloane. Generating Software from Specifications 2007
  - ©Jones and Bartlett Publishers Inc. www.jbpub.com

LINKÖPINGS UNIVERSITET

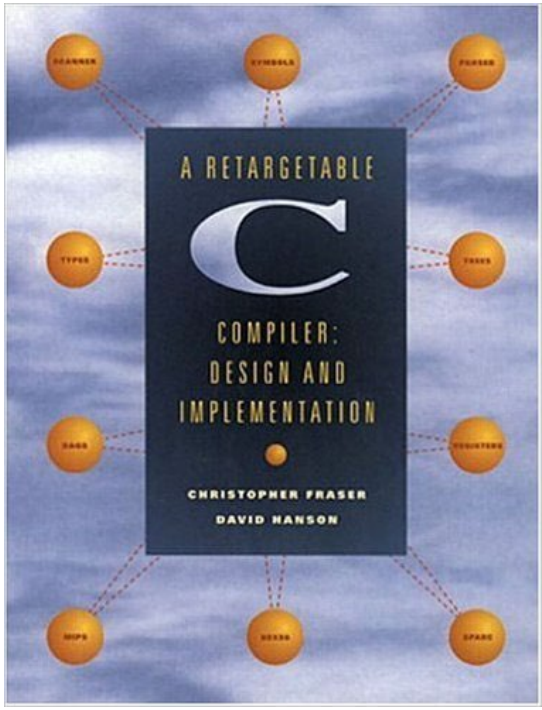
Part III

Primarily Back-End Frameworks and Generators

LCC (Little C Compiler)

Not really a generator, but uses IBURG

- Dragon-book style C compiler implementation in C
- Very small (20K Loc), well documented, tested, widely used
- Open source: http://www.cs.princeton.edu/software/lcc
- Textbook: *A retargetable C compiler* [Fraser, Hanson 1995] contains complete source code
- Fast, one-pass compiler



LLC (Little C Compiler)

- C frontend (hand-crafted scanner and recursive descent parser) with own C preprocessor
- Low-level IR
  - Basic-block graph containing DAGs of quadruples
  - No AST
- Interface to IBURG code generator-generator
- Example code generators for MIPS, SPARC, Alpha, x86 processors
- Tree pattern matching + dynamic programming
- Few optimizations
  - local common subexpression elimination
  - constant folding
- Good choice for source-to-target compiling if a quick prototype is needed

GCC – not a generator, but widely used

- Gnu Compiler Collection (earlier: Gnu C Compiler) https://gcc.gnu.org/
- Compilers for C, C++, Fortran, Java, Objective-C, Ada, and more
  - sometimes with own extensions, e.g. Gnu-C
- Open-source, developed since 1985
- Quite large (GCC 6.2.0 tarball is 835 MB)
- 3 IR formats (all language independent)
  - GENERIC: tree representation for whole function (also statements)
  - GIMPLE (simple version of GENERIC for optimizations) based on trees but expressions in quadruple form. High-level, low-level and SSA-low-level form.
  - RTL (Register Transfer Language, low-level, Lisp-like) (the traditional GCC-IR) only word-sized data types; stack explicit; statement scope
- Many optimizations

GCC – not a generator, but widely used

- Currently at version 14.2
- Since version 4.x (2004) has strong support for retargetable code generation
  - Machine description in .md file
  - Reservation tables for instruction scheduler generation

- Many target architectures
  - Note: GCC is not a cross-compiling compiler and does not include a linker. It compiles code for a set of languages, but only targets a single target platform. If you want to cross-compile code, you need to compile a linker and GCC targeting this platform (you have one GCC and linker toolchain installed for each target platform).
- Good choice if one has the time to get into the framework (and what you want is *a compiler, not a development environment*).
- Note: a new version numbering where 5.2 is really 4.10.2 and 6.0 is really 4.11.0 (in the old version numbering scheme).

LLVM - The LLVM Compiler Infrastructure Project

https://llvm.org/

Official LLVM *dragon* logotype. Inspired by the course book. Dragons, like LLVM, are powerful.

LLVM != “Low-Level Virtual Machine”

LLVM - The LLVM Compiler Infrastructure Project

- "Low-level virtual machine", IR. LLVM is a backend framework.
- Mainly accessed through an API and is suitable for integration in an IDE (such as Apple's XCode).
- Also comes with command-line tools, which can manipulate its IR (LLVM bitcode), including optimizing bitcode to produce an optimized bitcode file or generating an executable from bitcode.
- It includes:
  - Front-ends for C/C++/ObjC/OpenMP (`clang`), can use GCC as a frontend (`dragonegg`),
  - A debugger (`lldb`).
  - A C++ standard library.
  - An experimental linker (`lld`).

- Third parties add more frontends, for example the Julia language.

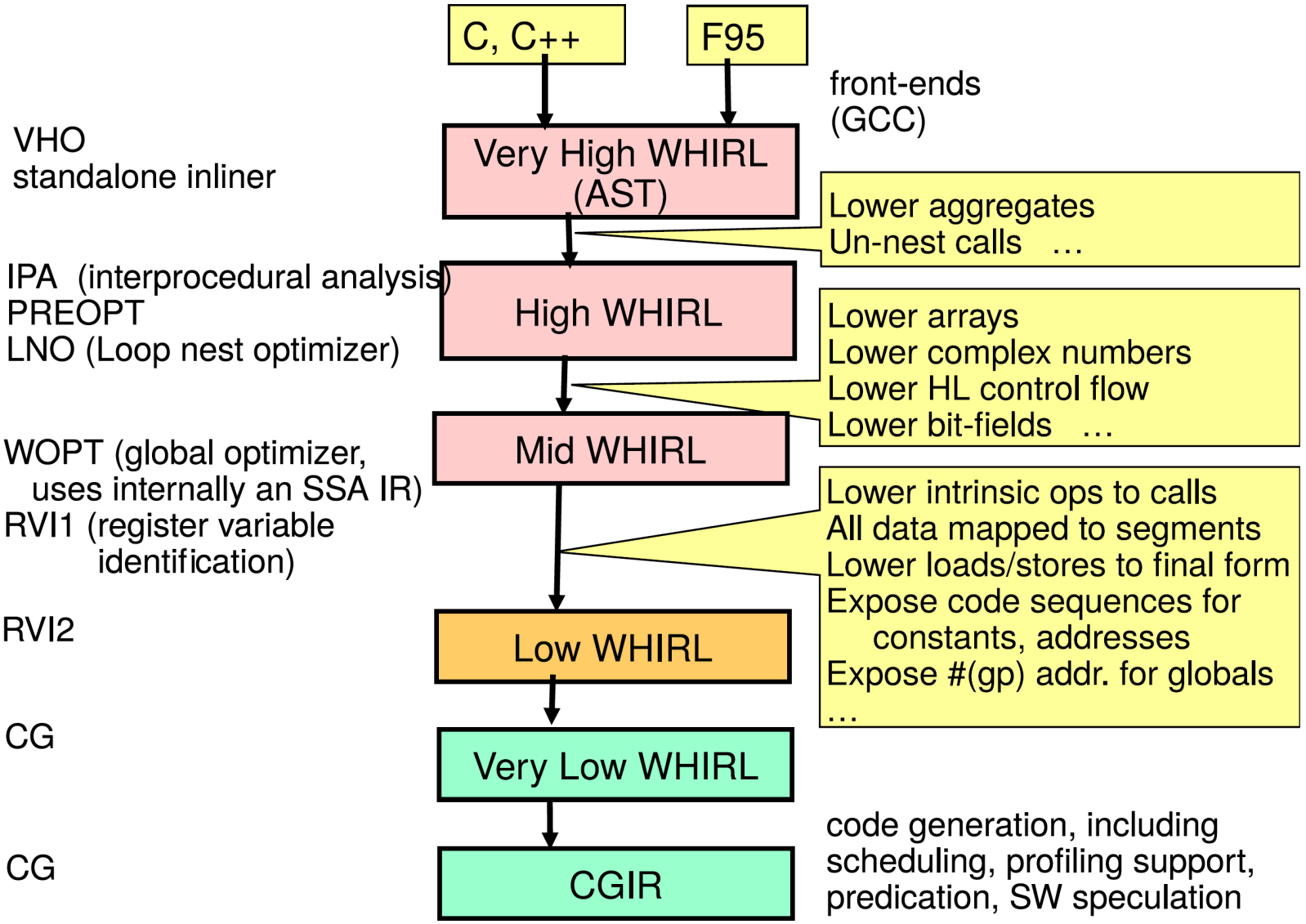
LLVM - The LLVM Compiler Infrastructure Project

- Compiles to several target platforms (see `llc --version`)
  - LLVM is a cross-compiling compiler.
  - You only need one copy of LLVM installed to generate code for all supported platforms.
  - You probably still need a linker for the target installed (`lld` is limited).
  - You will also need platform-specific headers for the compiler frontend and platform-specific libraries to link against.
- Open source (BSD-license), originally developed at Univ. of Illinois at Urbana Champaign.
- Note: Microsoft's Visual Studio can use `clang` as a front-end but uses their own backend and optimizations instead of LLVM.

Open64 / ORC Open Research Compiler Framework

- Based on SGI Pro-64 Compiler for MIPS processor, written in C++, went open source in 2000. Discontinued in 2011. Forked by Nvidia for optimizing CUDA code.
- Several tracks of development (Open64, ORC, ...)
- For Intel Itanium (IA-64) and x86 (IA-32) processors. Also retargeted to x86-64, Ceva DSP, Tensilica, XScale, ARM, ... "simple to retarget" (?)
- Languages: C, C++, Fortran95 (uses GCC as frontend), OpenMP and UPC (for parallel programming)
- Industrial strength, with contributions from Intel, Pathscale, ...
- Open source: https://github.com/open64-compiler/open64
- 6-layer IR:
  - WHIRL (VH, H, M, L, VL) – 5 levels of abstraction
    - All levels semantically equivalent
    - Each level is a lower-level subset of the higher form
  - and target-specific very low-level CGIR

ORC: Flow of IR

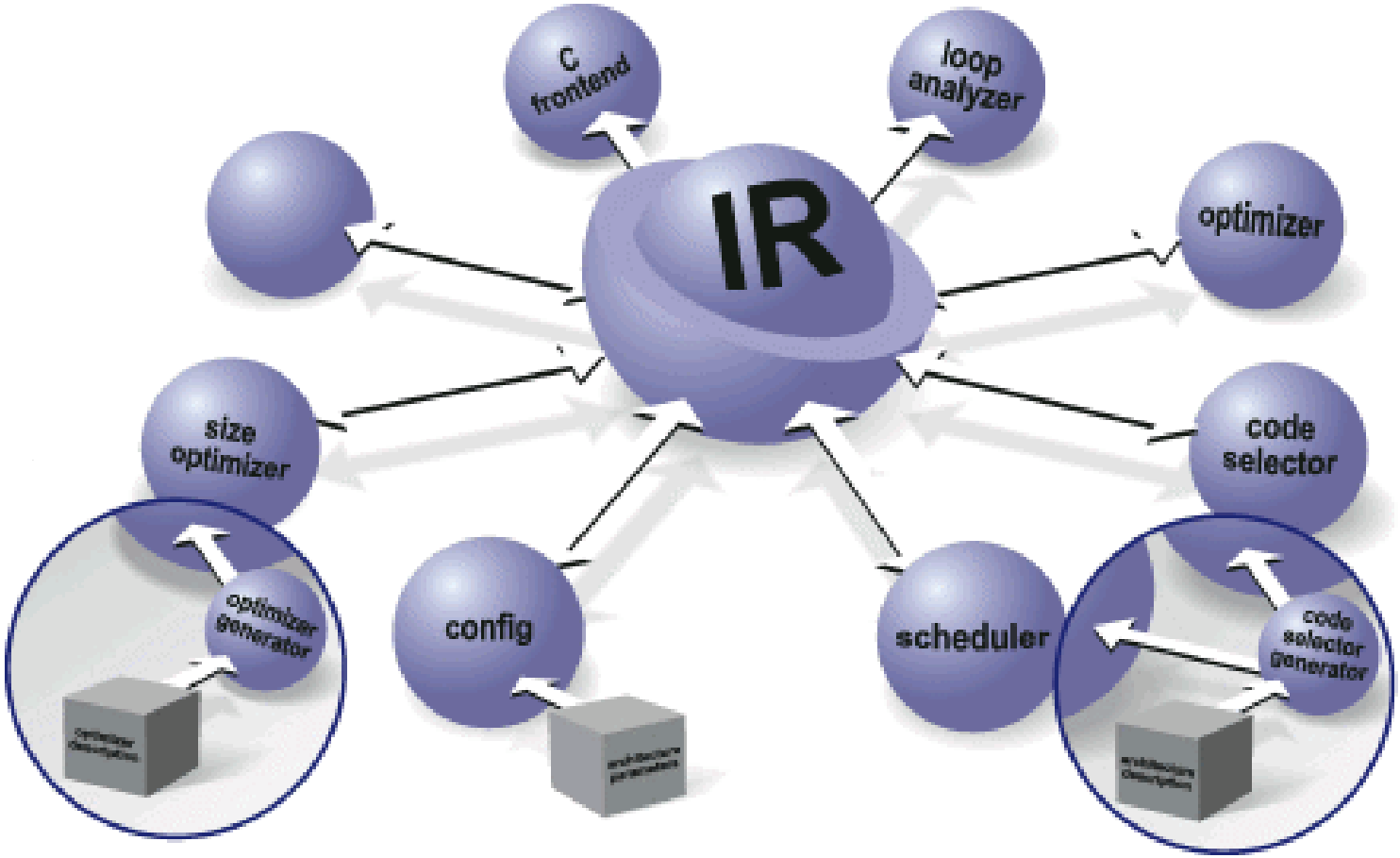


Open64 / ORC Open Research Compiler

- Multi-level IR
  - Translation by lowering
  - 😊Analysis / Optimization engines can work on the most appropriate level of abstraction
  - 😊 Clean separation of compiler phases
  - 😕Framework gets larger and slower
- Many optimizations, many third-party contributed components

CoSy – commercial compiler framework

- A commercial compiler framework primarily focused on backends
- CoSy is a registered trademark of ACE Associated Computer Experts bv



https://www.ace.nl

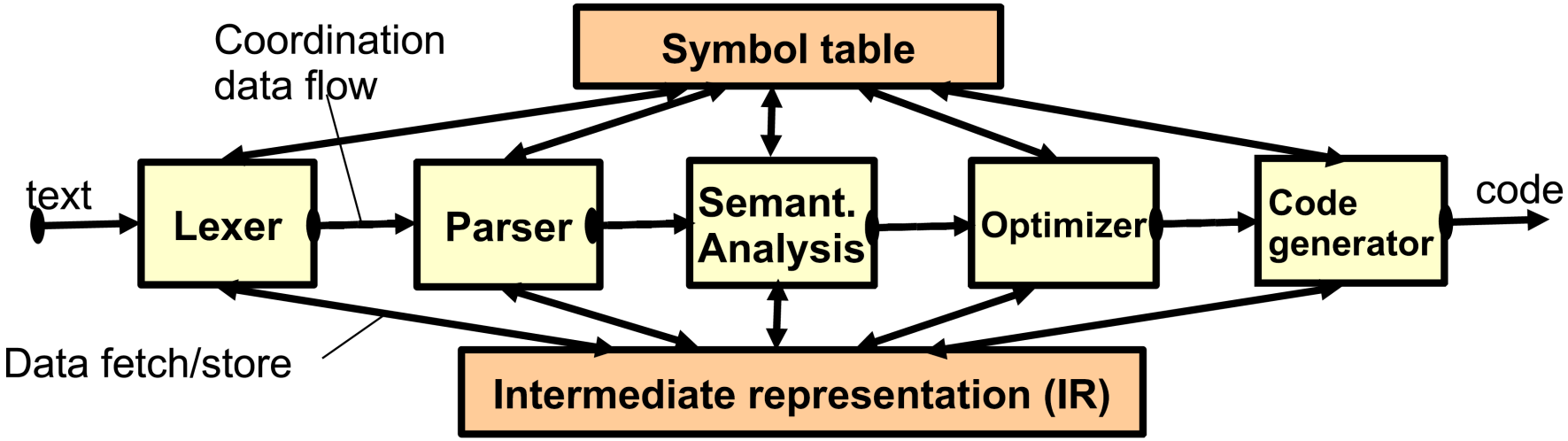
CoSy - features

- For the C family of languages, aimed at the embedded market
- Single IR, control flow based
  - IR is generated from a distributed description: every ‘engine’ can extend the IR with data structures
- More modular than any other compiler framework
- Extensible and flexible
  - Fixed point, arbitrary primitive data types, multiple memories, processor specific extensions
- Generated code generator
  - Supports VLIW, non-interlocked architectures, predicated execution, software pipelining, hardware loops, …

Traditional Compiler Structure

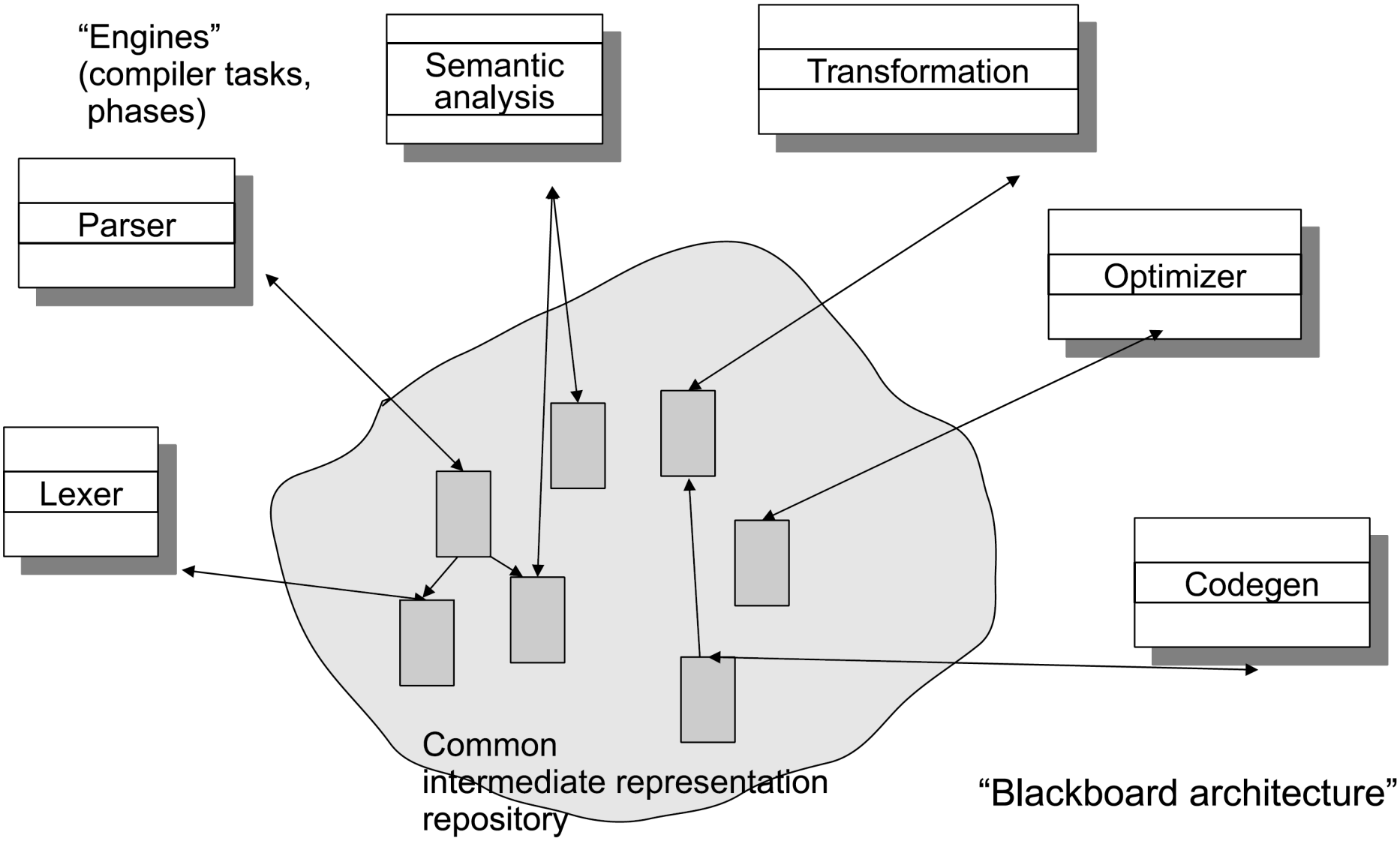
- Traditional compiler model: sequential process

- Improvement: Pipelining (by files/modules, classes, functions)



More modern compiler model with shared symbol table and IR

A CoSy Compiler with Repository Architecture



CoSy – Engine

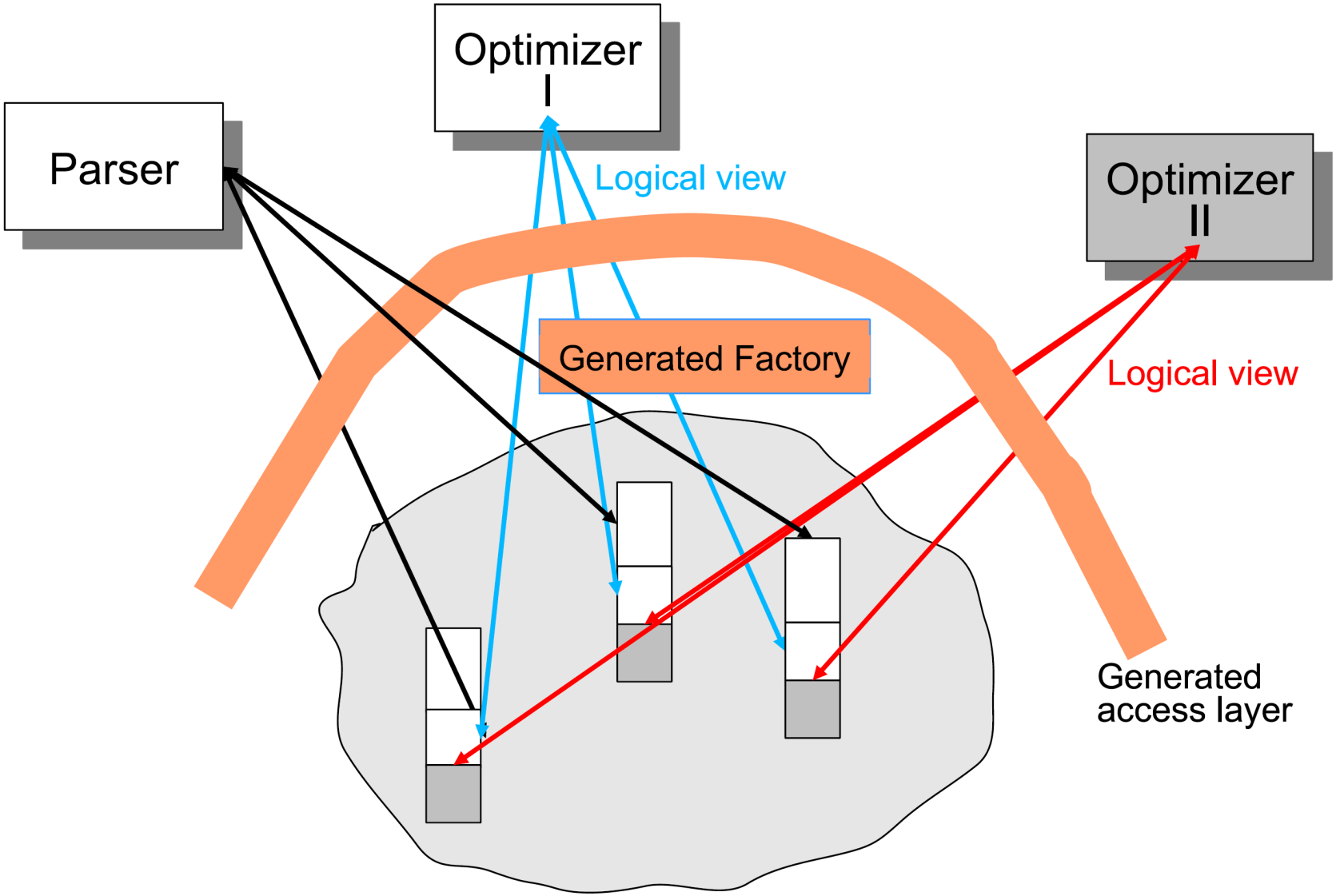
- Modular compiler building block
- Performs a well-defined task
- Focus on algorithms, not compiler configuration
- Parameters are handles on the underlying common IR repository
- Execution may be in a separate process or as subroutine call
  - *the engine writer does not know!*
- View of an engine class: the part of the common IR repository that it can access (scope set by access rights: read, write, create)
- Examples: Analyzers, Lowerers, Optimizers, Translators, Support

CoSy – Composite Engines

- Built from simple engines or from other composite engines *by combining engines in interaction schemes*
  - Loop, Pipeline, Fork, Parallel, Speculative, ….
- Described in EDL (Engine Description Language)
- View defined by the joint effect of constituent engines
- A compiler is nothing more than a large composite engine

```
ENGINE CLASS compiler ( IN u : mirUNIT ) {
  PIPELINE
  frontend ( u )
  optimizer ( u )
  backend ( u )
}
```

A CoSy Compiler



Composite Engines in CoSy

- Component classes (engine class)
- Component instances (engines)
- Basic components are implemented in C
- Interaction schemes (cf. skeletons) form complex connectors
  - SEQUENTIAL
  - PIPELINE
  - DATAPARALLEL
  - SPECULATIVE
- EDL can embed automatically
  - Single-call-components into pipes
  - p<> means a stream of p-items
  - EDL can map their protocols to each other (p vs p<>)

```
ENGINE CLASS optimizer ( procedure p )
{
  ControlFlowAnalyser cfa;
  CommonSubExprEliminator cse;
  LoopVariableSimplifier lvs;
  PIPELINE
    cfa ( p );
    cse ( p );
    lvs ( p );
}

ENGINE CLASS compiler ( file f )
{
  . . .
 Token token;
 Module m;
 PIPELINE
 // lexer takes file, delivers token stream
 lexer ( IN f, OUT token <>);
 // Parser delivers a module
 parser ( IN token <>, OUT m ) ;
 sema ( m );
 decompose ( m, p <> ) ;
 // here comes a stream of procedures
 // from the module
 optimizer ( p <> ) ;
 backend ( p <> ) ;
}
```

Evaluation of CoSy

- The outer call layers of the compiler are generated from view description specifications
  - Adapter, coordination, communication, encapsulation
  - Sequential and parallel implementation can be exchanged
  - There is also a non-commercial prototype [Martin Alt: On Parallel Compilation. PhD thesis, 1997, Univ. Saarbrücken]
- Access layer to the repository must be efficient (solved by generation of macros)
- Because of views, a CoSy-compiler is very easily extensible
  - That's why it was expensive
  - Reconfiguration of a compiler within an hour

LINKÖPINGS UNIVERSITET

Part IV

More Frameworks

More Frameworks …

- Cetus
  - http://cobweb.ecn.purdue.edu/ParaMount/Cetus/
  - C/C++ source-to-source compiler written in Java.
  - Open source
- Tools and generators
  - TXL source-to-source transformation system
  - ANTLR frontend generator
  - Xtext open-source software framework for developing programming languages and DSLs
    - generates not only a parser, but also a class model for the abstract syntax tree, as well as providing a fully featured, customizable Eclipse-based IDE.

More Frameworks …

- Some influential frameworks of the 1990s
  - SUIF Stanford university intermediate format, https://suif.stanford.edu
  - Trimaran (for instruction-level parallel processors) www.trimaran.org
  - Polaris (Fortran) UIUC
  - Jikes RVM (Java) IBM
  - Soot (Java)
  - GMD Toolbox / Cocolab Cocktail™ compiler generation tool suite
  - and many others ...

- And many more for the embedded domain ...

Continue the journey?

> *Now this is not the end.*
> *It is not even the beginning of the end.*
> *But it is, perhaps, the end of the beginning.*
>
> W. Churchill

- Do you like compiler technology? Learn more?
- Advanced Compiler Construction 9 hp (PhD-level)
- Thesis project (exjobb) at PELAB, 30/15/16 hp
- For more software engineering:
  - TDDE41 Software Architectures, 6 hp (VT), replaces component-based software
  - TDDE45 Software Design and Construction, 6 hp (HT), replaces Design Patterns
  - TDDE46 Software Quality, 6 hp (VT)

LINKÖPINGS UNIVERSITET

Bootstrapping of a Compiler

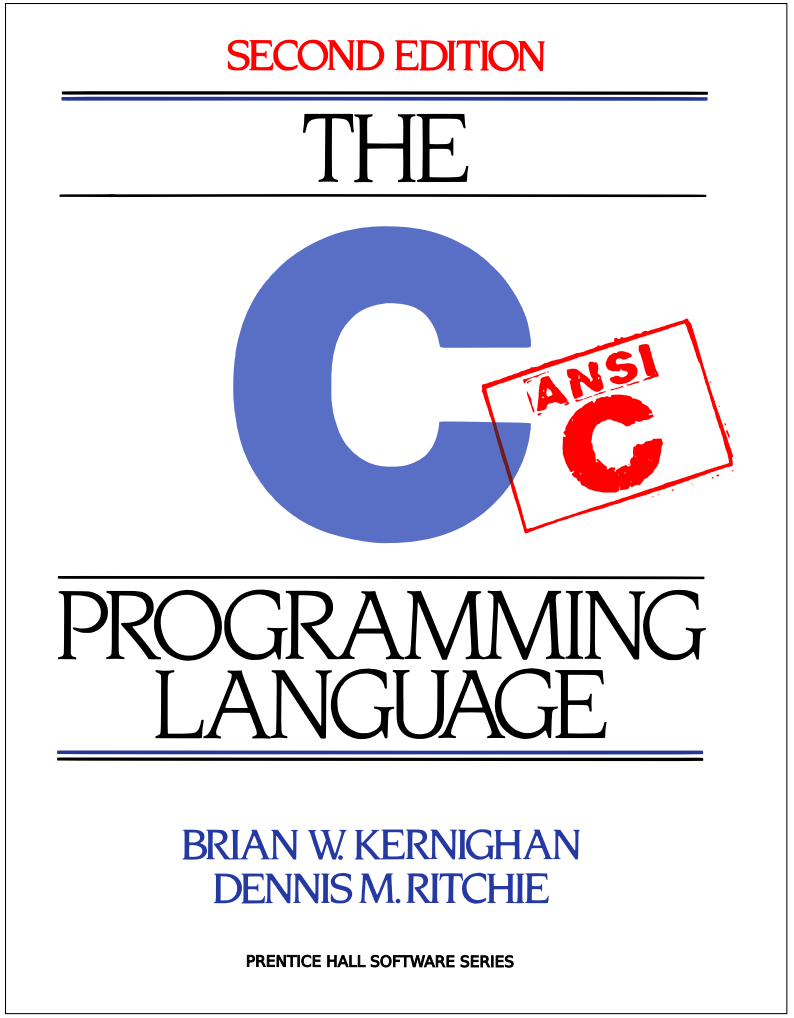
Optional Material

How to Implement a Compiler

- Implement your compiler in an existing language (easy).
- Writing your compiler in the language it is trying to compile itself (bootstrapping):
  - 😊 Another compiler already exists, with binaries for your build architecture.
  - 😐 Another compiler already exists, but no binaries for your build architecture (only 32-bit; your system is 64-bit; cross-compiling + bootstrapping).
  - 😕 No other compiler exists.

Example: Origins of C

- Started as the language B, a simple dialect of BCPL.
- The B compiler was implemented in TMG, a language for writing a compiler, itself written in PDP-7 assembler.
- The B compiler was then rewritten in B itself and compiled using the TMG version of the B compiler.
- The B compiler was then tweaked into “New B”, and eventually became the C language and compiler.



Bootstrapping Language x: Alternatives

Notation: ${}^{k}C_{x}^{o}$, a compiler C written in the language x which compiles the source language k into the object language o.

- Implement a small, stupid compiler for x_subset in another language y, producing native executables. This compiler is $A_1 = {}^{k}C_{x\_subset}^{native,unoptimized}$).
- Write a compiler in x_subset that can compile x_subset into C-code. Bootstrap your compiler using A_1. Then we get a compiler $A_2 = {}^{x\_subset}C_{x\_subset}^{C-code}$.
- Keep a tarball of translated C-code that produces an x_subset compiler. Compile this old, basic version of the compiler ($A_3 = {}^{x\_subset}C_{C}^{native,unoptimized}$ generated by A_2).
- Write an interpreter for x_subset . Feed it your compiler as input, ... (A_4)
- Or keep a tarball of bytecode for x that you can interpret. ($A_5 = {}^{x\_subset}C_{bytecode}^{native,unoptimized}$)
- Interpret x code with a human in the loop, being fed your compiler as input. (A_6)

Bootstrapping Language x: Step 2

- Compile a (subset) version of your compiler (B = ${}^{x}C_{x\_subset}^{native,unoptimized}$) using this other compiler (A_n).
  - This version might be incomplete (optimization modules disabled, etc., that A_n does not support).
- Compile a full version of your compiler (C = ${}^{x}C_{x}^{native}$), using (B = ${}^{x}C_{x\_subset}^{native,unoptimized}$).
- Compile an optimized, full version of your compiler (D = ${}^{x}C_{x}^{native}$) using (C = ${}^{x}C_{x}^{native}$), targeting (possibly cross-compiling) your host platform.

Rationale

- It is a proof that your language is powerful enough to do something useful.
- Why should I use your programming language if you yourself use C?
- Only need to learn one language to be a compiler developer.
- Improving the performance for the language also improves the performance of the compiler.

OpenModelica Bootstrapping History (1)

- Implementation of a Modelica compiler using rml2c
- Design of an early MetaModelica language version as an extended subset of Modelica, spring 2005.
- Implementation of a MetaModelica Compiler (MMC) which translates MetaModelica into RML intermediate form, spring-fall 2005.
- Automatically translating the whole OpenModelica compiler, 60 000 lines, from RML to MetaModelica.
- In parallel, developing MDT (Modelica Development Tooling), including debugger for MMC, 2005-2006.
- Switching to using this MetaModelica 1.0, the MMC compiler, and MDT for the OpenModelica compiler development, at that time 3-4 full-time developers. Fall 2006.
- Preliminary implementation of pattern-matching and exception handling in the OpenModelica compiler, to enable future bootstrapping. Spring-fall 2008.

OpenModelica Bootstrapping History (2)

- Continuation of the work on better support for pattern-matching compilation, support for lists, tuples, records (uniontypes), etc. in OpenModelica. Spring-fall 2009.
- Implementation of higher-order functions (used in MetaModelica), also in OpenModelica. Fall 2009, spring 2010.
- The bootstrapped compiler supporting most of MetaModelica 2.0, which includes standard Modelica. Fall 2010, spring 2011.
- Adding garbage collection. Fall 2012.
- Improving the build system, parallel builds. Reaching full testsuite coverage, good performance, and running the tests nightly. 2013.
- Removing support for MMC.
- Further adding, enhancing, and redesigning MetaModelica language features, based on usage experience, the Modelica design effort, and inspiration from functional languages and languages. Refactoring parts of the compiler to use the enhanced features.

OpenModelica Bootstrapping

- Start with a tarball of source-code (only code necessary for bootstrapping)
  - https://github.com/OpenModelica/OMBootstrapping
- This source-code was at one time generated by OMC compiled with RML/MMC.
- At some point, OMC was able to generate its own tarball.
- Then support for RML/MMC was dropped and new language features added to OMC (that RML/MMC did not support).
- At a later time, these new language features were used in the compiler itself (and a new tarball was generated).
- Parts of the compiler that are not used during bootstrapping can use new language features before a new tarball is generated.
- …

OpenModelica Cross-Compiling (ARM host, x86 build)

- Start with a tarball of source-code: https://github.com/OpenModelica/OMBootstrapping
- Bootstrap the x86 version of OpenModelica, save this somewhere. Make clean.
- `./configure –with-omc=path/to/x86/omc`
- Cross-compile the ARM version of OpenModelica using the x86 version of OMC to produce code.
- Note: OMC generates C-code, so you need a cross-compiler tool-chain installed.
- For gcc, a similar approach is used, but you then use the regular gcc to compile a version of gcc that runs on x86 but produces ARM executables (including assemblers and linkers).
- clang (LLVM) is able to produce assembly for multiple targets using the same compiler (but it does not integrate assemblers, linkers, or C++ run-times for these targets, so you usually need to install a gcc cross-compilation tool-chain anyway).

Thank you!

- Any questions?
- This Week
  - TDDE66 & TDDD55
    - Last Seminar: Exam preparation