TDDB44/TDDD55 Lecture 14:
Compiler Frameworks and Compiler Generators

A (non-exhaustive) survey
with a focus on open-source frameworks

Peter Fritzson, Christoph Kessler and Martin Sjölund

Department of Computer and Information Science
Linköping University

2018-12-17

LINKÖPING
UNIVERSITY

Overview

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

Part I

Syntax-Based Generators

Syntax-Based Generators

- Lex and Flex – generates lexical analysers.
  - Clones and/or open source alternatives exist for many programming languages. Wikipedia has a reasonable overview.
- Yacc and Bison – generates 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 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`

Part II

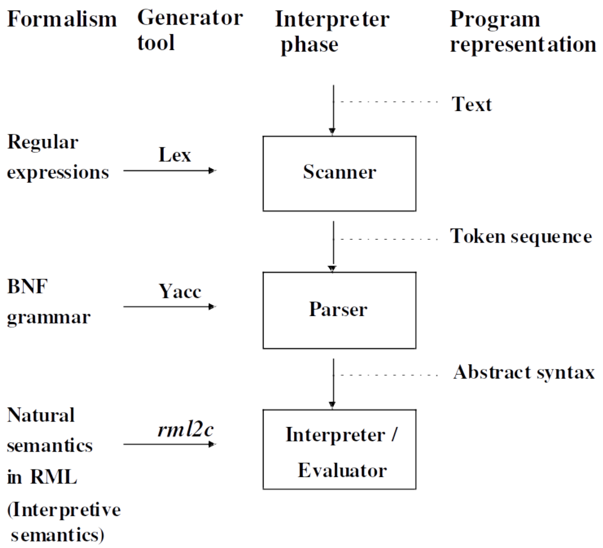
Semantics-Based Generators

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

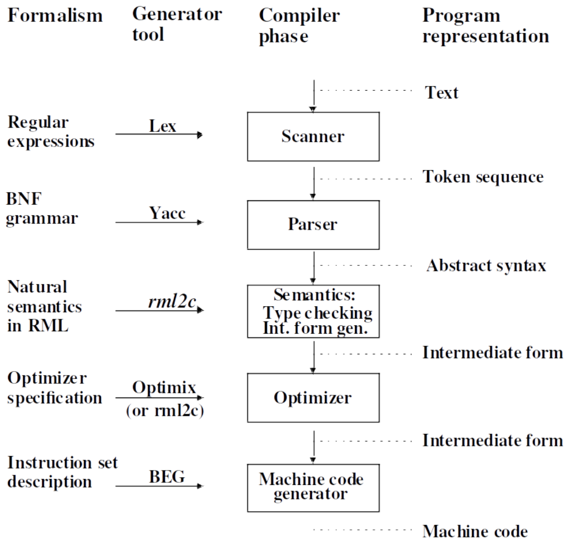
- Goals
  - Efficient code – comparable to hand-written compilers
  - Simplicity – simple to learn and use
  - Compatibility with “typical natural semantics/operational semantics” and with Standard ML
- Properties
  - Deterministic
  - Separation of input and output arguments
  - Statically strongly typed
  - Polymorphic type inference
  - Efficient compilation of pattern-matching

`www.ida.liu.se/pelab/~rml` – developed around 1999 and used in OpenModelica until 2014-10-25.

Generating an Interpreter Implemented in C, using rml2C



Generating a Compiler Implemented in C, using rml2C



RML Syntax

Goal: Eliminate phletora 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
```

Typical RML rule:

```
rule    NameX(H1,T1) => R1   &
         ...
        NameY(Hn,Tn) => Rn   &
        <cond>
        ______________________________

         RelationName(H,T) => R
```

LINKÖPING UNIVERSITY

Example: the Exp1 Expression Language

Typical expressions

```
12 + 5*3
-5 * (10 - 4)
```

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
```

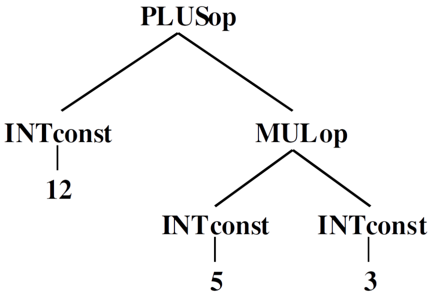


Figure: Abstract syntax tree of 12 + 5*3

Evaluator for Exp1

```
Relation eval: Exp => int =
```

Evaluation of an integer constant ival is the integer itself:

```
axiom eval( INTconst(ival) )
 => ival
```

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. (we have removed division below)

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

Simple Lookup in Environments Represented as Linked Lists

```
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.
     RML does not support fancy hash tables... *)
```

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

Mini-Freja Interpreter performance compared to Centaur/Typol:

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

Some Attribute-Grammar Based Tools

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

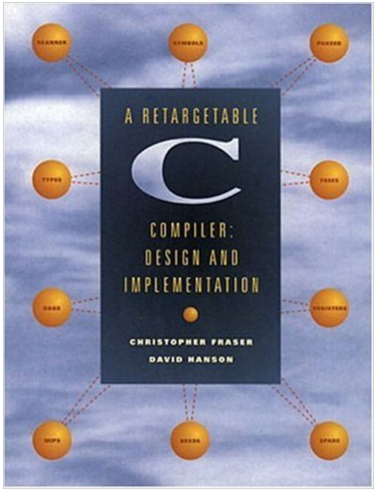
Part III

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, well tested, widely used
- Open source: `http://www.cs.princeton.edu/software/lcc`
- Textbook *A retargetable C compiler* [Fraser, Hanson 1995] contains complete source code
- One-pass compiler, fast



LCC (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 subexpr. elimination
  - constant folding
- Good choice for source-to-target compiling if a prototype is needed soon

GCC – Not a Generator, but wide-spread usage

- Gnu Compiler Collection (earlier: Gnu C Compiler)
- Compilers for C, C++, Fortran, Java, Objective-C, Ada, and more
  - sometimes with own extensions, e.g. Gnu-C
- Open-source, developed since 1985
- Very 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

- Version 4.x (since 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 language, 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: Now has 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).

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

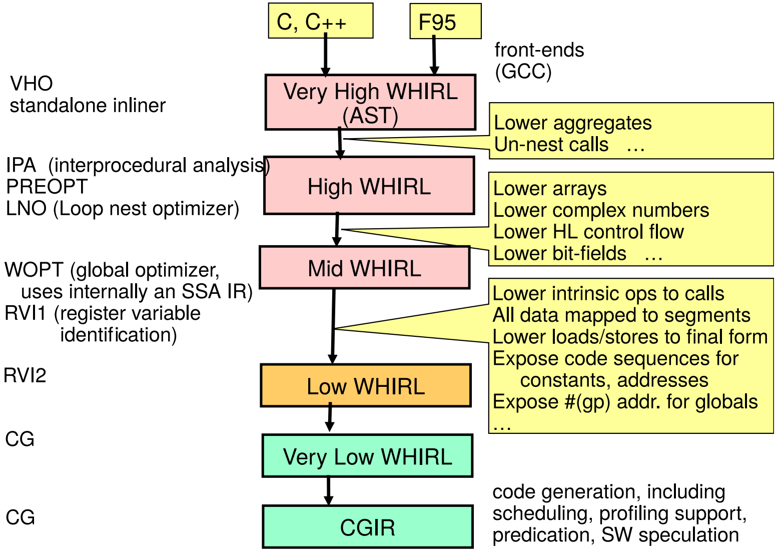
- “Low-level virtual machine”, IR. LLVM is a backend framework.
- Mainly accessed through an API, and is suitable for integration in an IDE (such as Apple’s XCode).
- Also comes with command-line tools, which can manipulate its IR (LLVM bitcode), including optimizing bitcode to produce an optimized bitcode 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, including for example the Julia language.

- 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: sourceforge.net/projects/open64/
- 6-layer IR:
  - WHIRL (VH, H, M, L, VL) – 5 levels of abstraction
    - All levels semantically equivalent
    - Each level a lower level subset of the higher form
  - and target-specific very low-level CGIR

ORC: Flow of IR

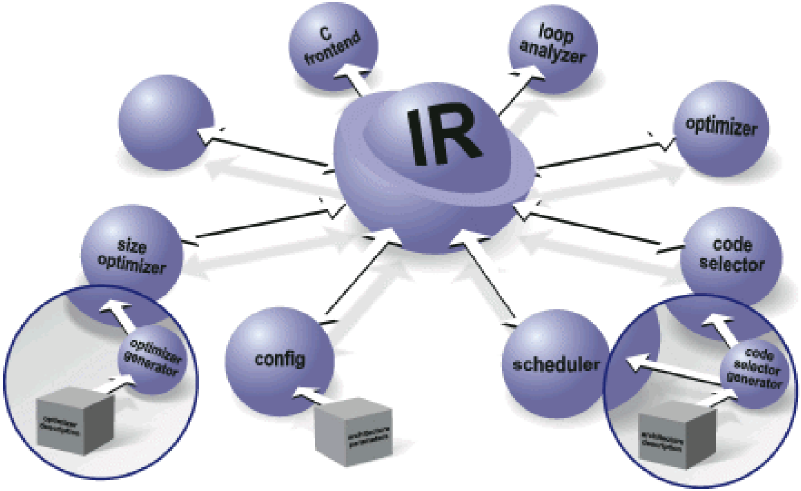

C, C++
F95
front-ends (GCC)
VHO
standalone inliner
Very High WHIRL (AST)
Lower aggregates
Un-nest calls …
IPA (interprocedural analysis)
PREOPT
LNO (Loop nest optimizer)
High WHIRL
Lower arrays
Lower complex numbers
Lower HL control flow
Lower bit-fields …
WOPT (global optimizer, uses internally an SSA IR)
RVI1 (register variable identification)
Mid WHIRL
Lower intrinsic ops to calls
All data mapped to segments
Lower loads/stores to final form
Expose code sequences for constants, addresses
Expose #(gp) addr. for globals
…
RVI2
Low WHIRL
CG
Very Low WHIRL
CG
CGIR
code generation, including scheduling, profiling support, predication, SW speculation

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

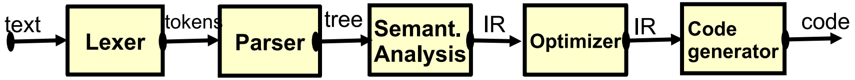
A commercial compiler framework primarily focused on backends



www.ace.nl

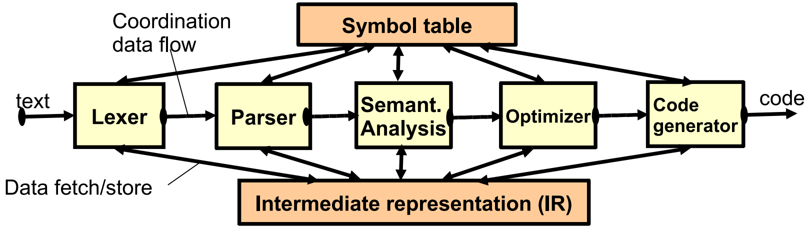
Traditional Compiler Structure

Figure: Traditional compiler model: sequential process

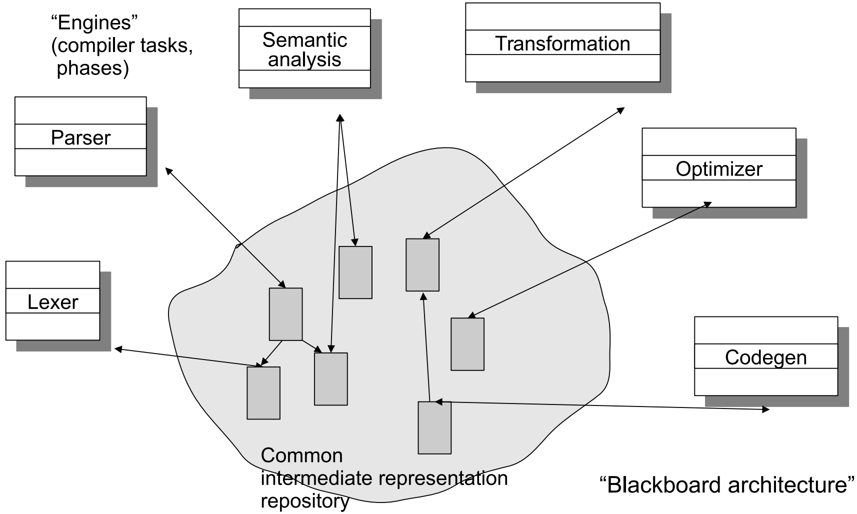


Improvement: Pipelining (by files/modules, classes, functions)

Figure: More modern compiler model with shared symbol table and IR



A CoSy Compiler with Repository Architecture


"Engines" (compiler tasks, phases)
Semantic analysis
Transformation
Parser
Optimizer
Lexer
Codegen
Common intermediate representation repository
"Blackboard architecture"

Engine

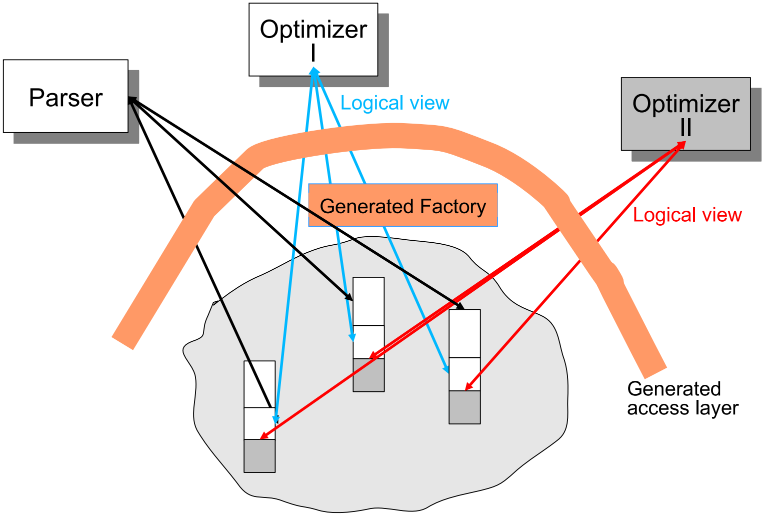
- Modular compiler building block
- Performs a well-defined task
- Focus on algorithms, not compiler configuration
- Parameters are handles on the underlying common IR repository
- Execution may be in a separate process or as subroutine call - *the engine writer does not know!*
- View of an engine class: 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

Composite Engines in CoSy

- 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 is expensive
  - Reconfiguration of a compiler within an hour

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

More Frameworks...

- Some influential frameworks of the 1990s
  - SUIF Stanford university intermediate format, `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 ...

The End (?)

> *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)

www.liu.se

LINKÖPING
UNIVERSITY