TDDD16 Compilers and Interpreters

TDDB44 Compiler Construction

Compiler Frameworks and Compiler Generators

A (non-exhaustive) survey

with a focus on open-source frameworks

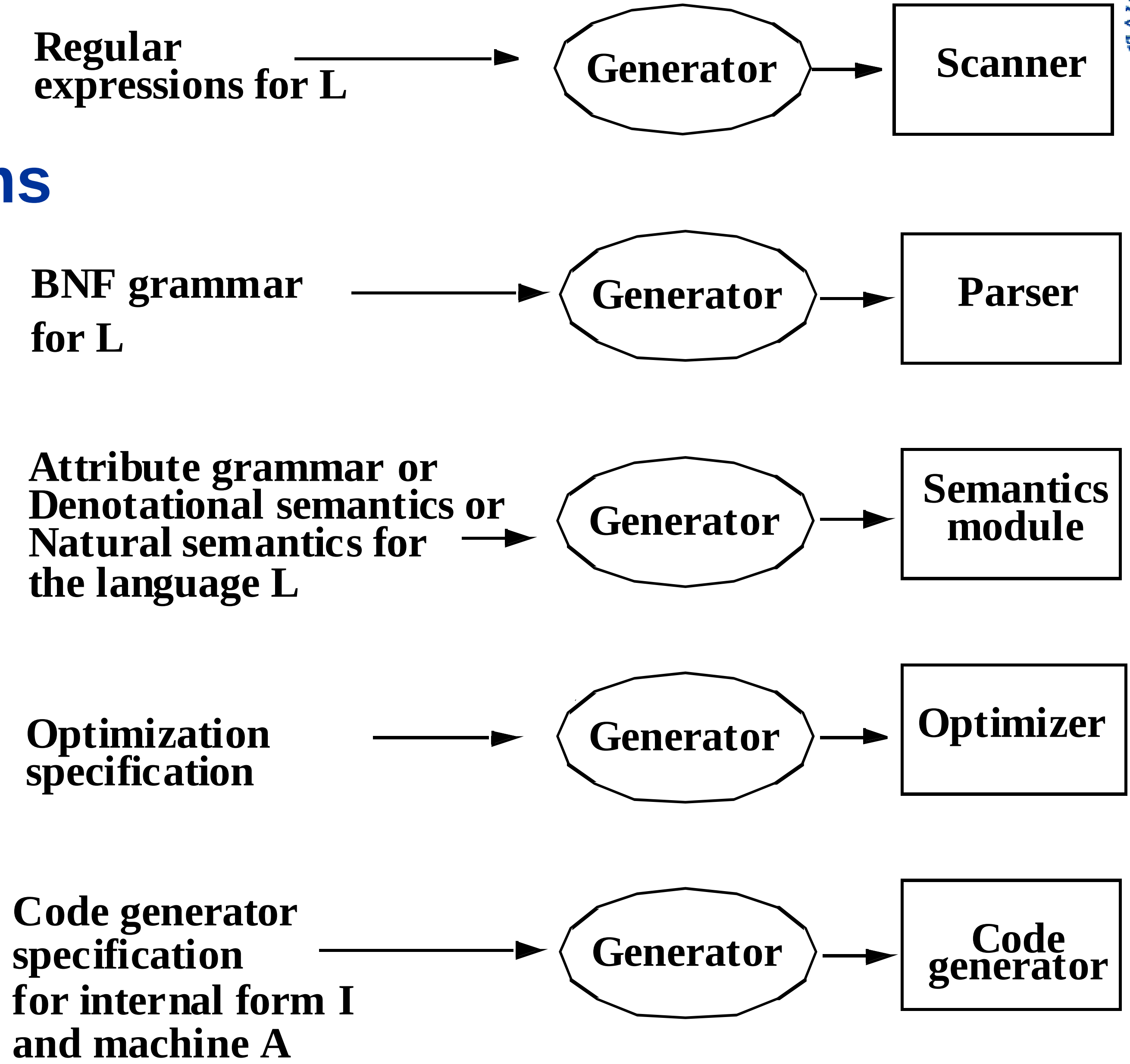
Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

Compiler Generators or CWS - Compiler Writing Systems

- A Compiler Generator or CWS is a program which, given a description of the source language and a description of the target language (object code description), produces a compiler for the source language as output.

- Different generators within CWS generate different phases of the compiler.

Compiler Generator Formalisms and generated Compiler Modules



TDDD16 Compilers and Interpreters

TDDB44 Compiler Construction

Syntax-Based Generators

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

Syntax-Based Generators

- LEX or FLEX– generates lexical analysers

- YACC or BISON – generates parsers

  - Compiler components that are not generated:
    - semantic analysis and intermediate code gen.
    - the optimisation phase
    - code generators

- Note: YACC produces parsers which are bad at error management

TDDD16 Compilers and Interpreters

TDDB44 Compiler Construction

Semantics-Based Generators

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

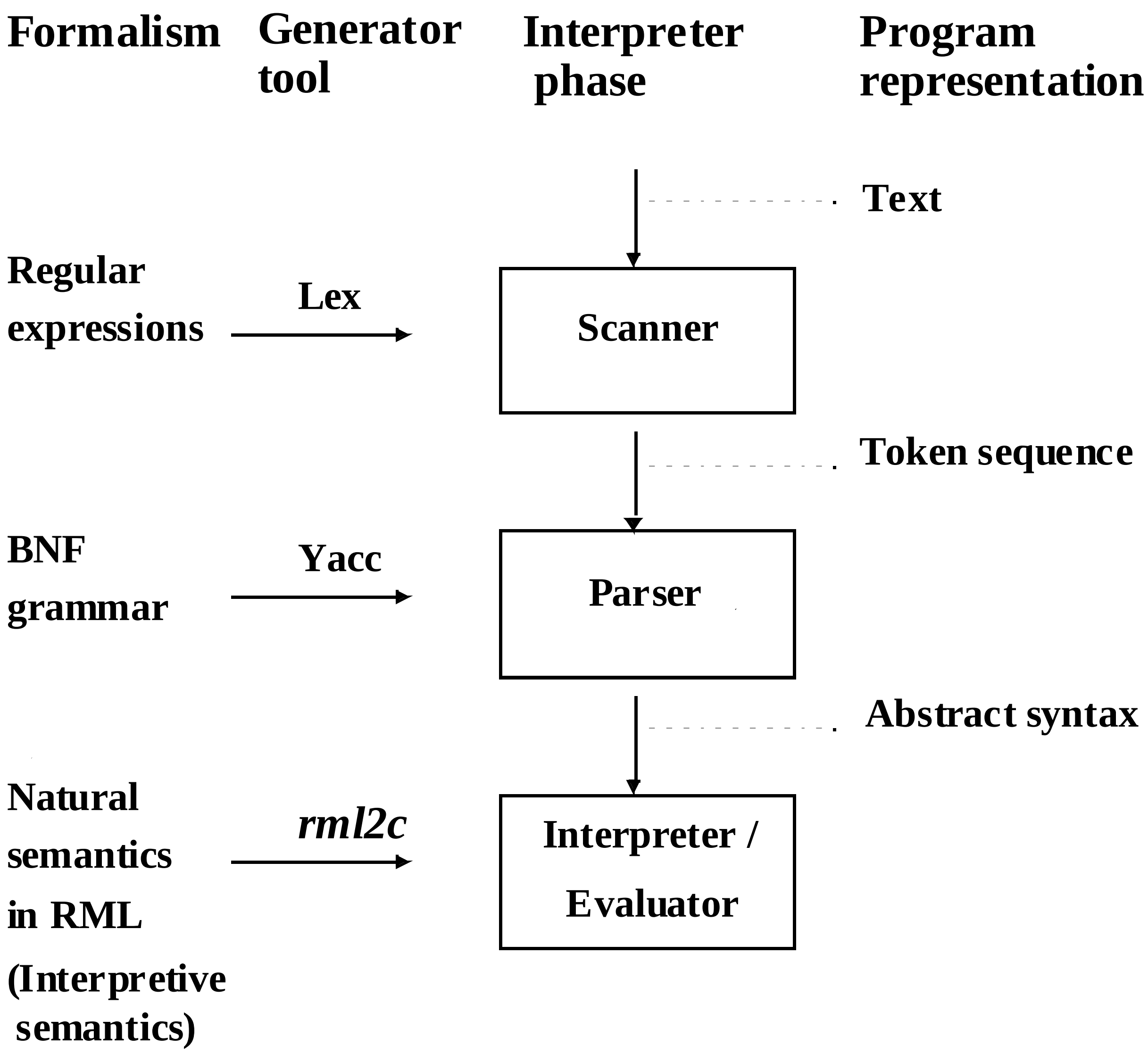
RML - A Compiler Generation System and Specification Language from Natural Semantics/Structured Operational Semantics

- **Goals**
  - Efficient code — comparable to hand-written compilers
  - Simplicity — simple to learn and use
  - Compatibility with "typical natural semantics" and with Standard ML
- **Properties**

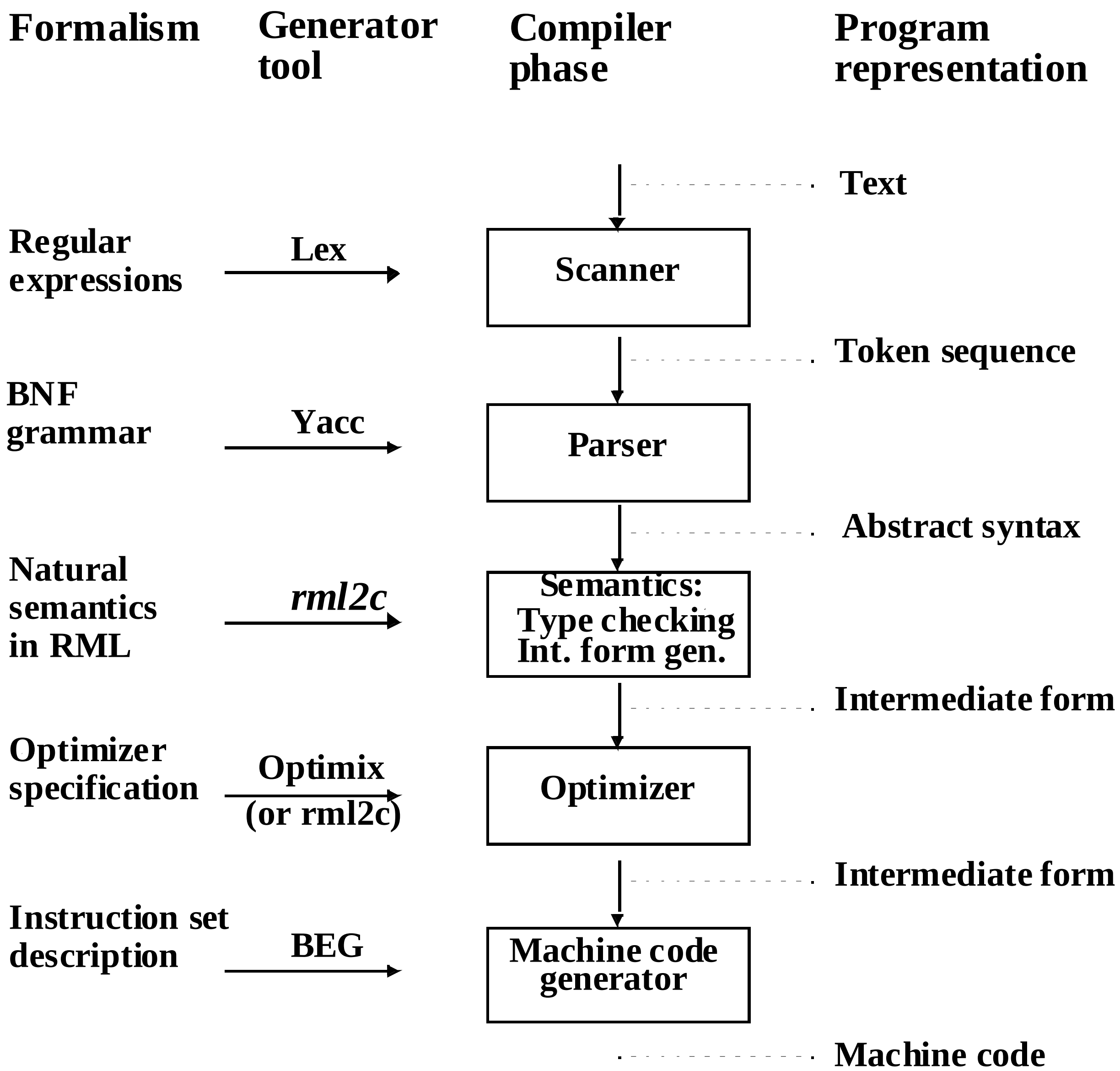
www.ida.liu.se/pelab/~rml

  - Deterministic
  - Separation of input and output arguments/results
  - Statically strongly typed
  - Polymorphic type inference
  - Efficient compilation of pattern-matching

Example Use: Generating an Interpreter Implemented in C, using rml2C



Example Use: Generating a Compiler Implemented in C



RML Syntax

- Goal: Eliminate phletora of special symbols usually found in Natural Semantics specifications

  Software engineering viewpoint: identifiers are more readable in large specifications
- A Natural semantics rule:
- $$\frac{H1\ T1 : R1 \ \ ..\ \ Hn\ Tn : Rn}{H\ \ T : R} \quad \text{if } \ \langle cond \rangle$$
- Typical RML rule:

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

Example: the Exp1 Expression Language

Typical expressions

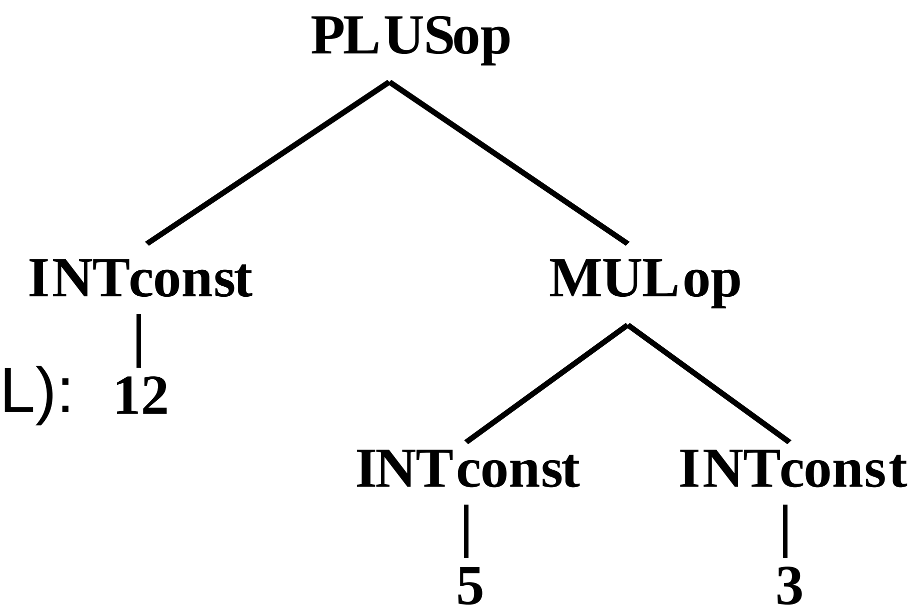
12 + 5*3

-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
```

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
```

Translational Semantics of the PAM language – Abstract Syntax to Machine Code

PAM language example:

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

Low level representation tree form

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

Arithmetic Expression Translation Semantics

Beginning of RML Relation trans_expr:

```
relation trans_expr:  Exp => Mcode list  =
axiom  trans_expr(INT(v))     => [MLOAD( N(v))]
axiom  trans_expr(IDENT(id)) => [MLOAD( I(id))]
....
```

Code template for simple subtraction expression:

```
<code for expression e1>
MB(MSUB (      e2))
```

and in assembly text form:

```
<code for expression e1>
SUB     e2
```

RML rule for simple (expr1 binop expr2):

```
rule trans_expr(e1) => cod1  &
     trans_expr(e2) => [MLOAD(operand2)]  &
     trans_binop(binop) => opcode  &
     list_append(cod1, [MB(opcode,operand2)]) => cod3
     -----------------------------------
     trans_expr(BINARY(e1,binop,e2) => cod3
```

The Complete trans_expr Relation

```
relation trans_expr:  Exp => Mcode list  =
```

(* Evaluation of expressions in the current environment *)

```
  axiom  trans_expr(INT(v)) => [MLOAD( N(v))]      (* integer constant *)
  axiom  trans_expr(IDENT(id)) => [MLOAD( I(id))]    (* identifier id *)
```

(* Arith binop: simple case, expr2 is just an identifier or constant *)

```
  rule   trans_expr(e1) => cod1  &
         trans_expr(e2) => [MLOAD(operand2)]  &  (* expr2 simple *)
         trans_binop(binop) => opcode  &
         list_append(cod1, [MB(opcode,operand2)]) => cod3
         -----------------------------------    (* expr1 binop expr2 *)
         trans_expr(BINARY(e1,binop,e2) => cod3
```

(* Arith binop: general case, expr2 is a more complicated expr *)

```
  rule   trans_expr(e1) => cod1  &
         trans_expr(e2) => cod2  &
         trans_binop(binop) => opcode  &
         gentemp => t1  &
         gentemp => t2  &
         list_append6(
           cod1,                          (* code for expr1 *)
           [MSTO(t1)],                    (* store expr1 *)
           cod2,                          (* code for expr2 *)=
           [MSTO(t2)],                    (* store expr2 *)
           [MLOAD(t1)],                   (* load expr1 value into Acc *)
           [MB(opcode,t2)] ) => cod3      (* Do arith operation *)
         -----------------------------------    (* expr1 binop expr2 *)
         trans_expr(BINARY(e1,binop,e2)) => cod3
end (* trans_expr *)
```

Help Relations
Called from trans_expr Relation

```
relation trans_binop:  BinOp => MBinOp  =
  axiom  trans_binop(PLUS) =>  MADD
  axiom  trans_binop(SUB)  =>  MSUB
  axiom  trans_binop(MUL)  =>  MMULT
  axiom  trans_binop(DIV)  =>  MDIV
end


relation gentemp: () => MTemp  =

  rule   tick => no
         ----------
         gentemp => T(no)
```

Some Applications of RML

- Small functional language with call-by-name semantics (mini-Freja, a subset of Haskell)

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

- Almost full Pascal with some C features (Petrol)
- Mini-ML including type inference
- Specification of Full Java 1.2
- Specification of Modelica 2.0

Additional Performance Comparison

Additional measurements on performed on a Fedora Core4 Linux machine (2007) with two AMD Athlon(TM) XP 1800+ processors at 1500 MHz and 1.5GB of memory

#primes	RML	SICStus	SWI	Maude MSOS Tool
8	0.00	0.05	0.00	2.92
10	0.00	0.10	0.03	5.60
30	0.02	1.42	1.79	226.7
40	0.06	3.48	3.879	-
50	0.13	-	11.339	-
100	1.25	-	-	-
200	16.32	-	-	-

Execution time in seconds. The – sign represents out of memory

Some RML and Semantics References

- Web page, open source: www.ida.liu.se/~rml
- Adrian Pop and Peter Fritzson. *An Eclipse-based Integrated Environment for Developing Executable Structural Operational Semantics Specifications*. in *3rd Workshop on Structural Operational Semantics*. 2006. Bonn, Germany. Elsevier Science. Electronic Notes in Theoretical Computer Science (ENTCS) No:175, Issue 1. p. 71-75
- Mikael Pettersson, *Compiling Natural Semantics*. Lecture Notes in Computer Science (LNCS). Vol. 1549. 1999: Springer-Verlag.
  (Based on PhD Thesis at PELAB, Linköping Univ, 1995)
- Gilles Kahn, *Natural Semantics*, in *Programming of Future Generation Computers*, Niva M., Editor. 1988, Elsevier Science Publishers: North Holland. p. 237-258.

Some Attribute-Grammar Based Tools

- JASTADD – OO Attribute grammars

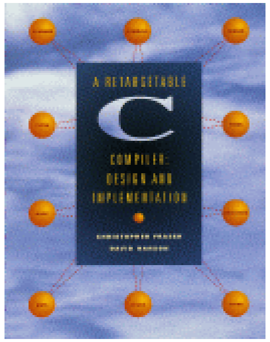
- Ordered Attribute Grammars.
  - Uwe Kastens, Anthony M. Sloane
    Generating Software from Specifications
    2007 (c) Jones and Bartlett Publishers Inc.
    www.jbpub.com

Primarily Back-End Frameworks and Generators

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

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
- 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 only
  - local common subexpr. elimination, constant folding
- Good choice for source-to-target compiling if a prototype is needed soon

GCC 4.x
Not a Generator – but wide-spread usage

- Gnu Compiler Collection (earlier: Gnu C Compiler)
- Compilers for C, C++, Fortran, Java, Objective-C, Ada …
  - sometimes with own extensions, e.g. Gnu-C
- Open-source, developed since 1985
- Very large
- 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
- Many target architectures
- Version 4.x (since ~2004) has strong support for retargetable code generation
  - Machine description in .md file
  - Reservation tables for instruction scheduler generation
- Good choice if one has the time to get into the framework

Open64 / ORC Open Research Compiler Framework

- Based on SGI Pro-64 Compiler for MIPS processor, written in C++, went open source in 2000
- 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: www.open64.net, ipf-orc.sourceforge.net
- 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

VHO
standalone inliner

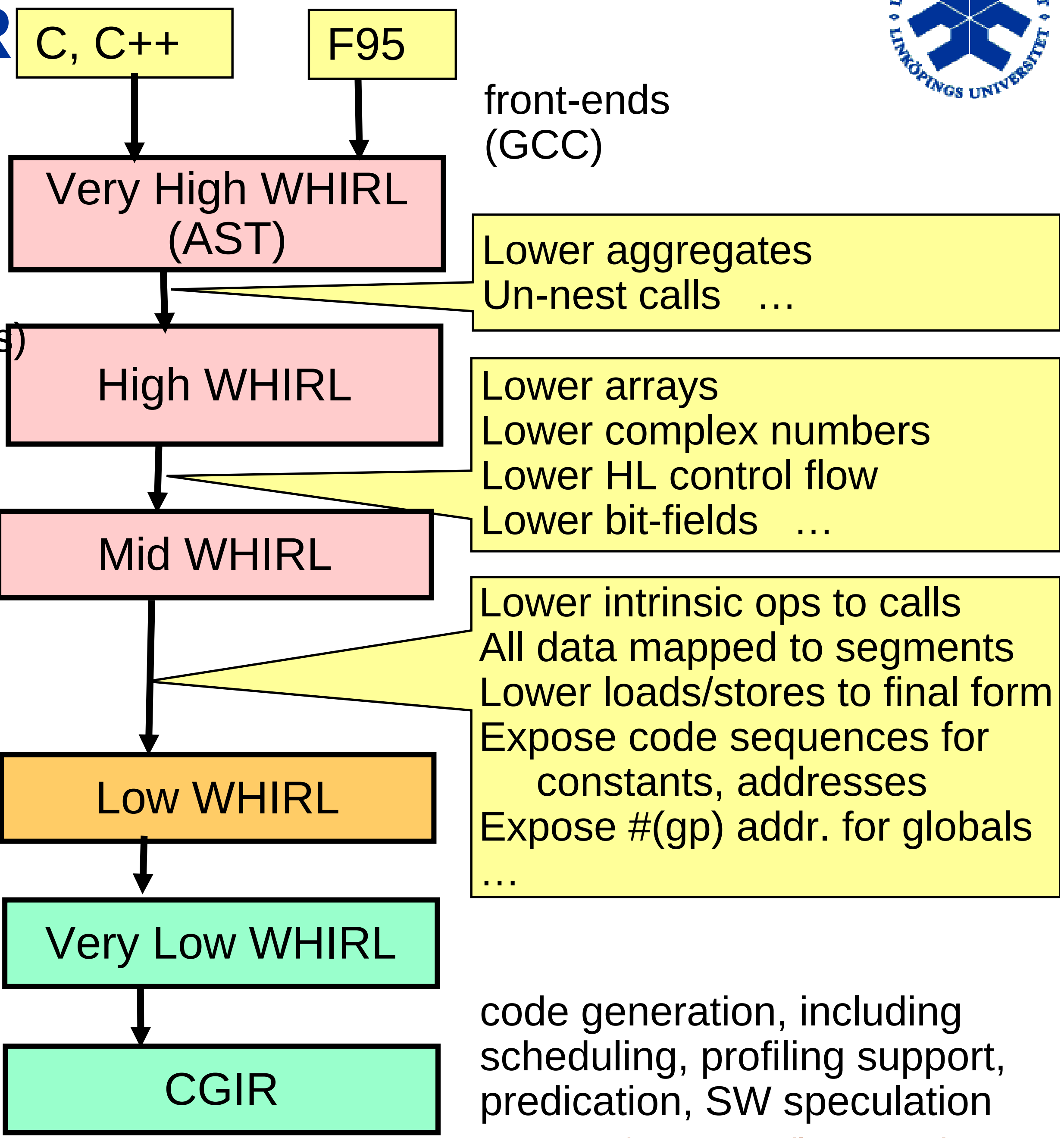
IPA (interprocedural analysis)
PREOPT
LNO (Loop nest optimizer)

WOPT (global optimizer,
uses internally an SSA IR)
RVI1 (register variable
identification)

RVI2

CG

CG



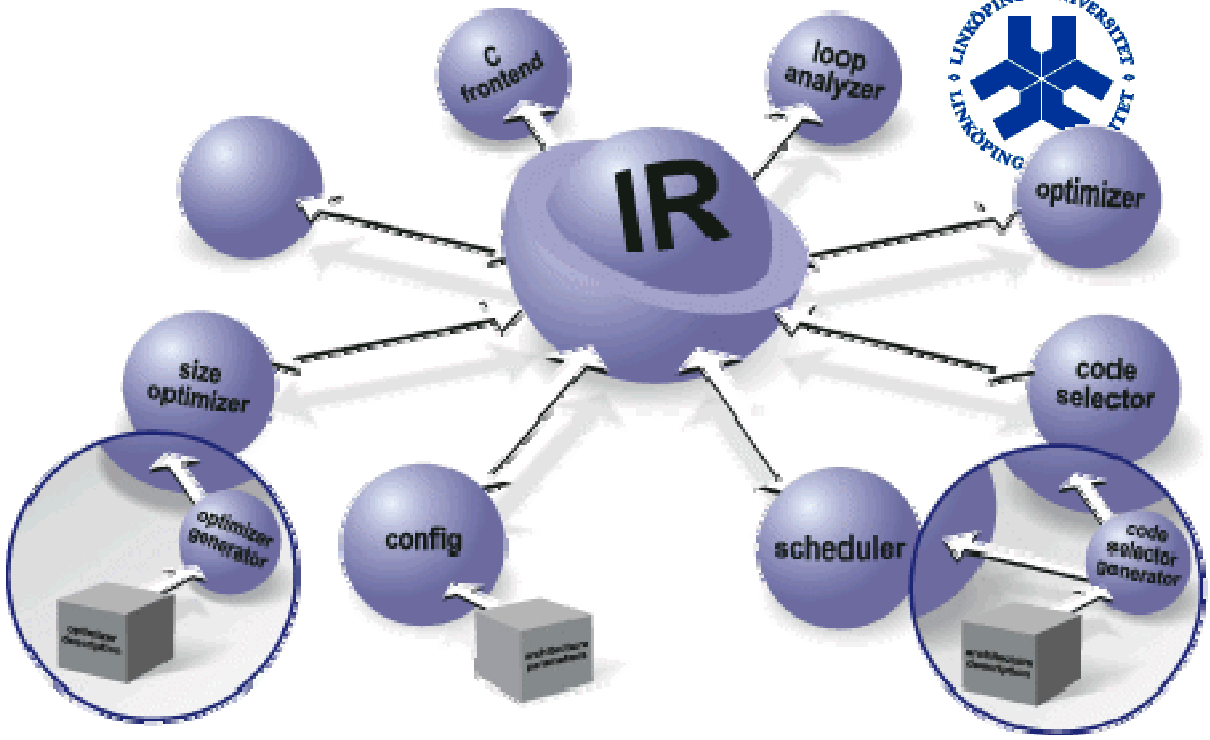
Open64 / ORC Open Research Compiler

- 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

TDDD16 Compilers and Interpreters

TDDB44 Compiler Construction



CoSy

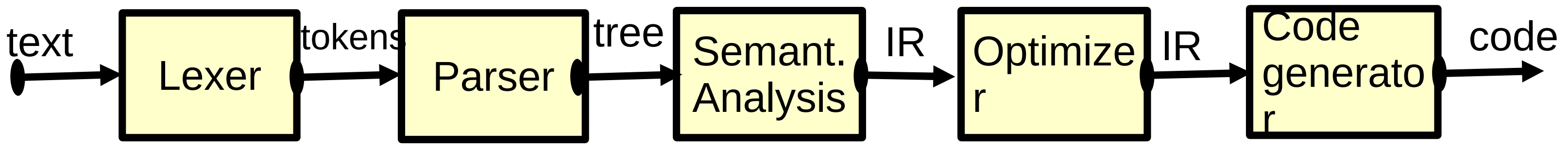
A commercial compiler framework
Primarily focused on backends

www.ace.nl

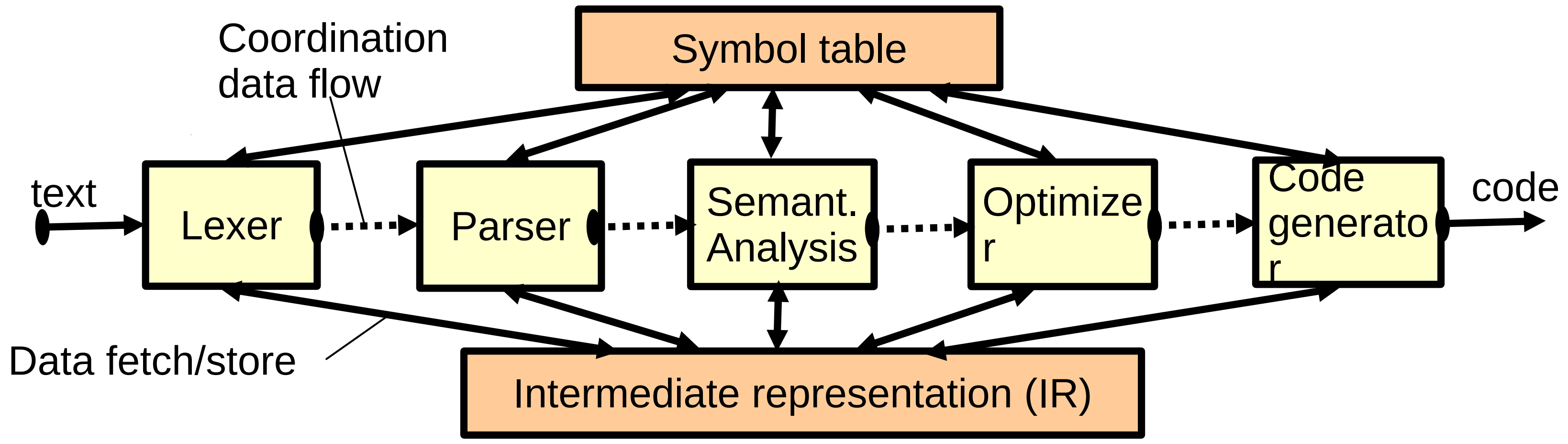
Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

Traditional Compiler Structure

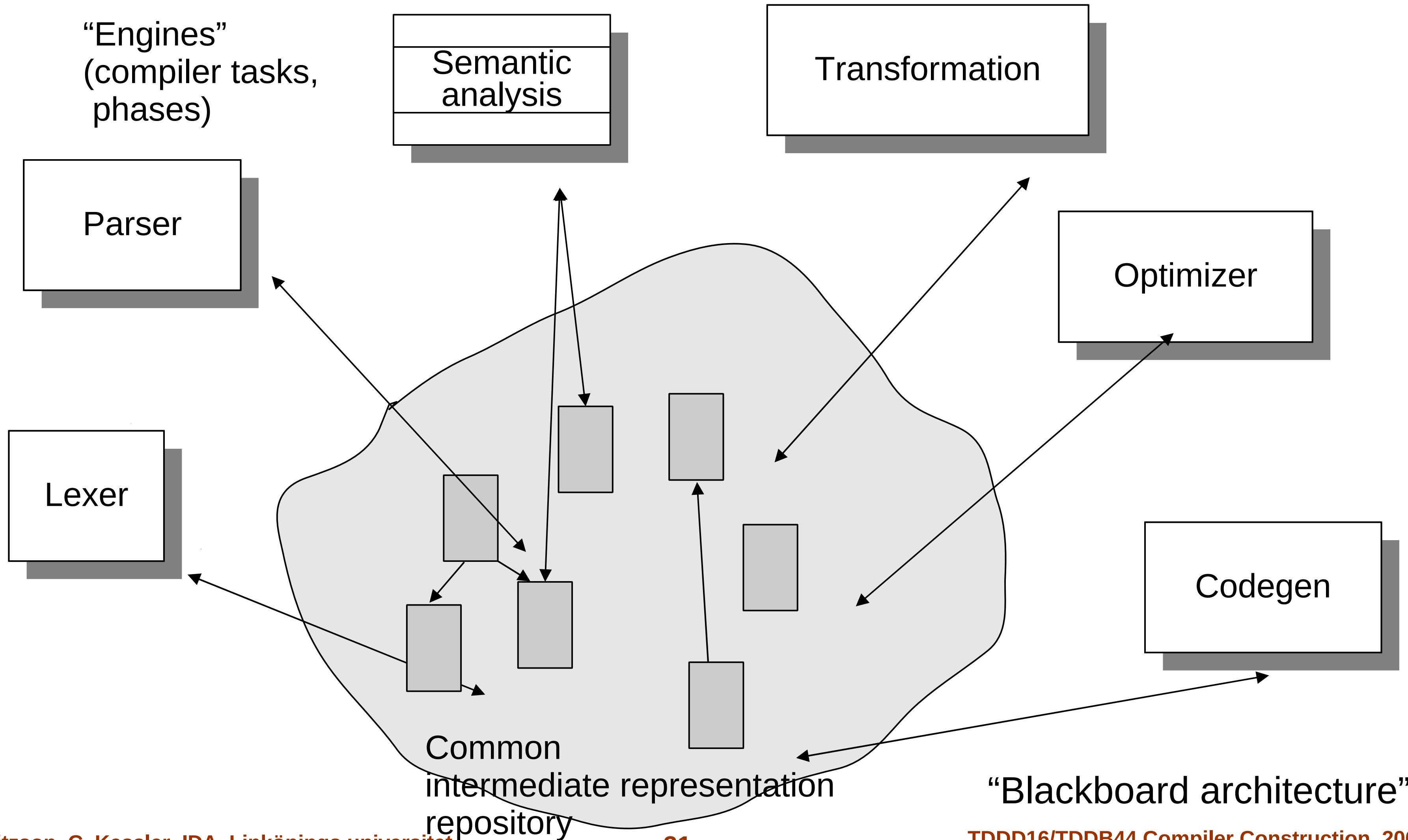
- Traditional compiler model: sequential process



- Improvement: Pipelining
  (by files/modules, classes, functions)
- More modern compiler model with shared symbol table and IR:

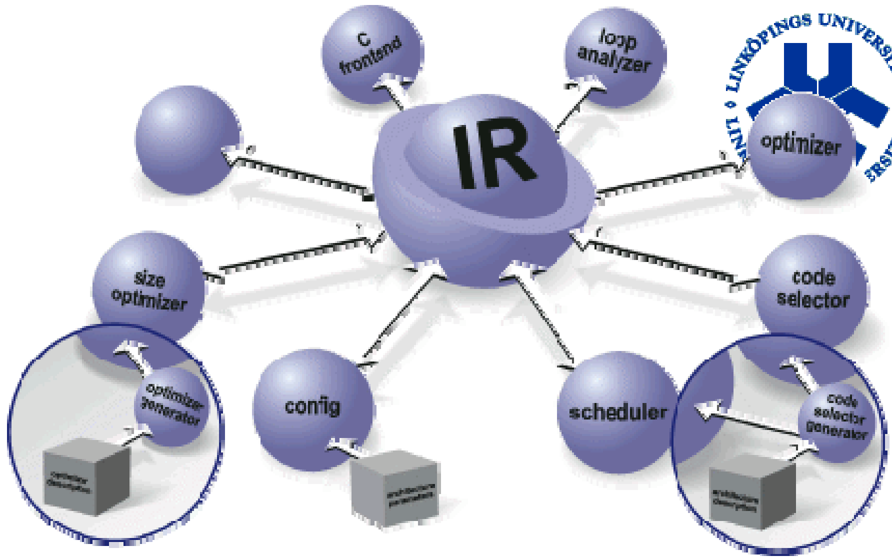


A CoSy Compiler with Repository-Architecture



"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)

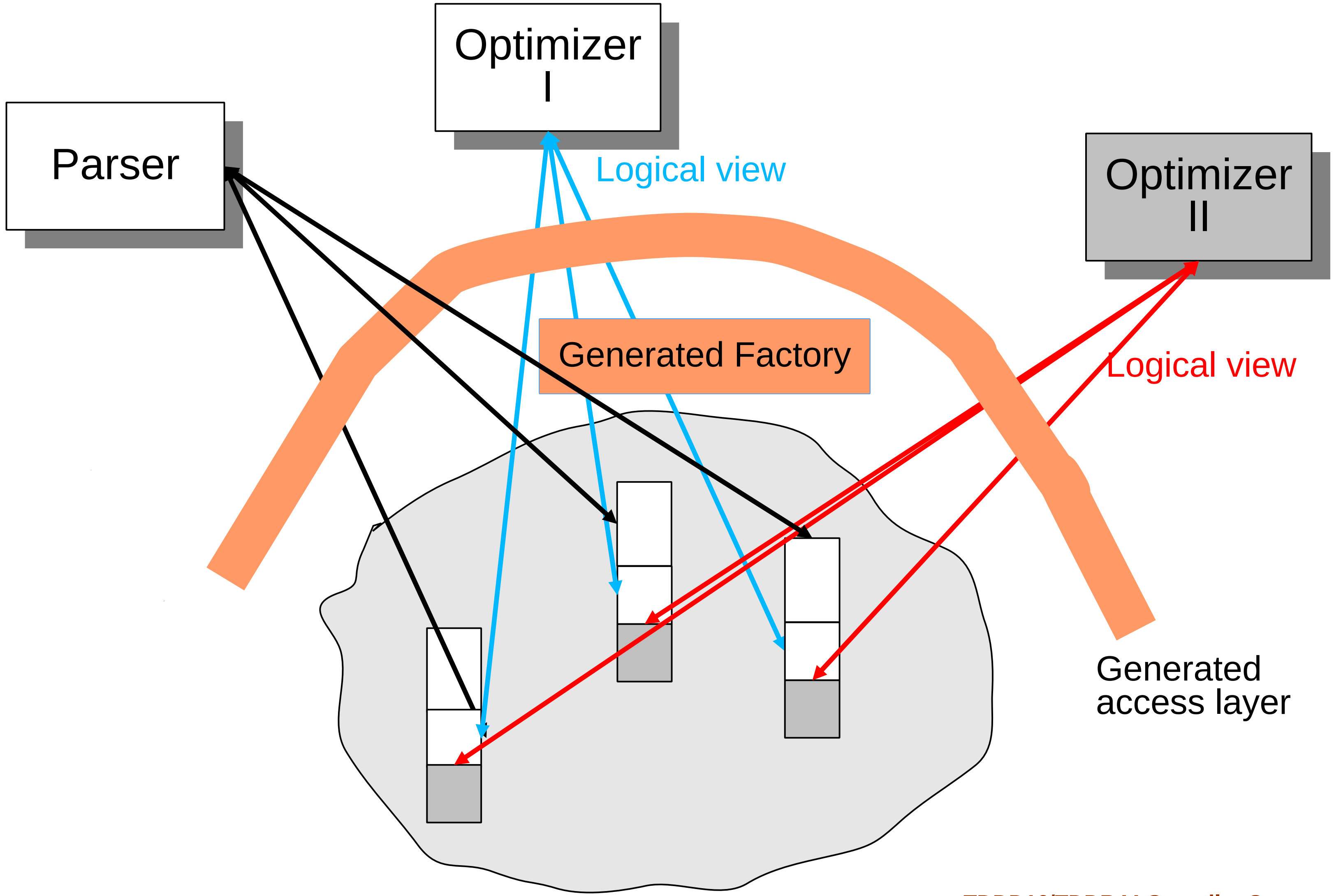
- Examples: Analyzers, Lowerers, Optimizers, Translators, Support

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



Example for CoSy EDL (Engine Description Language)

- 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 simply extensible
  - That's why it is expensive
  - Reconfiguration of a compiler within an hour

More Frameworks

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

More Frameworks…

- **LLVM** (Univ. of Illinois at Urbana Champaign)
  - llvm.org
  - ”Low-level virtual machine”, IR
  - compiles to several target platforms: x86, Itanium, ARM, Alpha, SPARC
  - Open source
- **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?
  - TDDC86 Compiler optimizations and code generation 6hp
  - TDDC18 Component-based software 4.5hp
  - Thesis project (Exjobb) at PELAB, 30 hp

R.U = (THANK YOU)$^{+}$. ((MERRY CHRISTMAS) . (HAPPY NEW YEAR))$^{+}$

Peter