Compiler Construction Using SYNTAX and LNT

Frédéric Lang

joint work with Hubert Garavel, Radu Mateescu, and Wendelin Serwe

INRIA – Laboratoire d'Informatique de Grenoble

Université Grenoble Alpes, CNRS, Grenoble INP

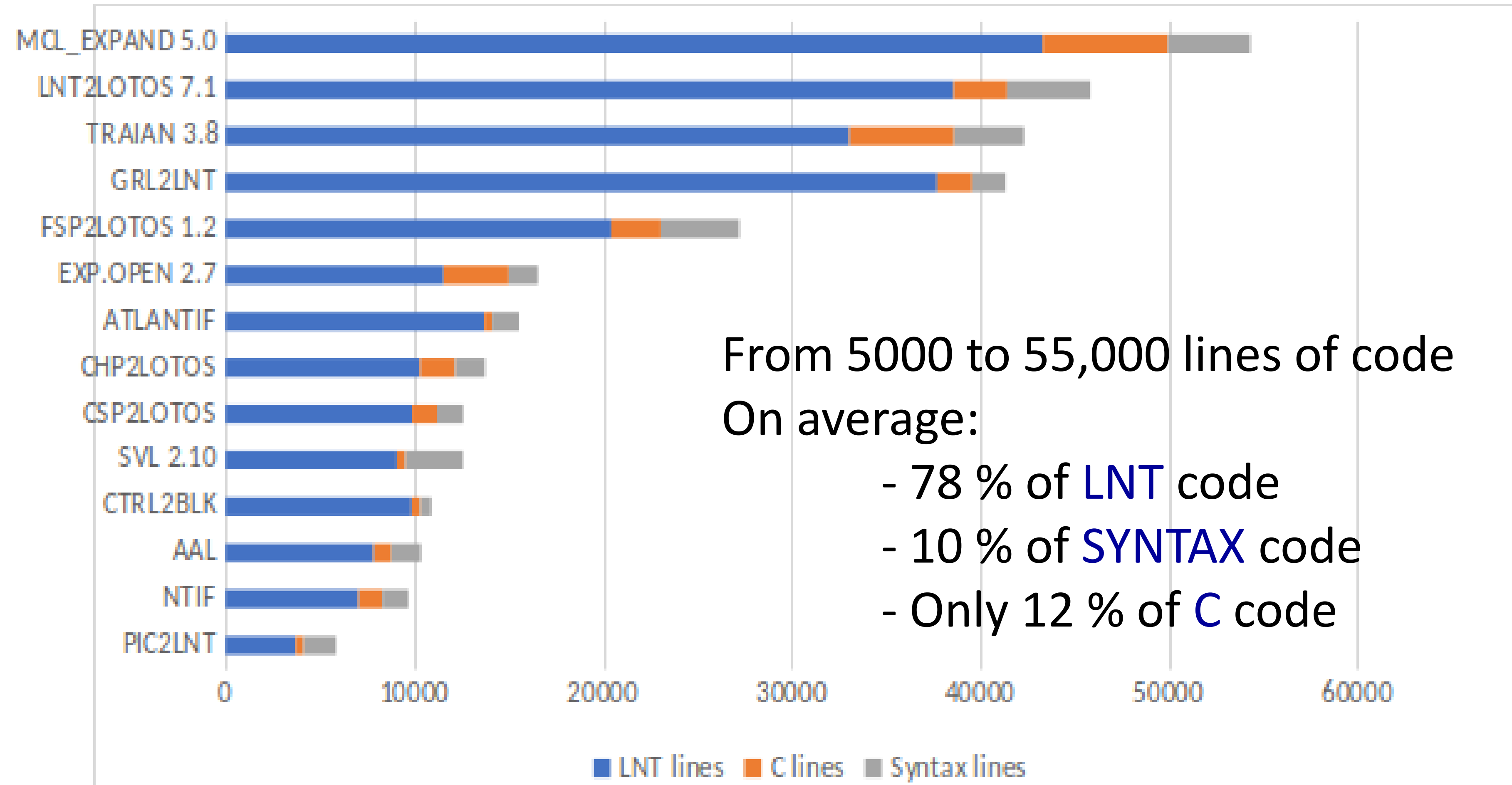
http://convecs.inria.fr

Compiler construction at CONVECS

- A compiler construction approach used by CONVECS for more than 20 years based on:
  - SYNTAX: a powerful compiler generator
  - LNT: a functional language that interfaces well with C
- Used in the development of our verification tools
  - Many languages $\Rightarrow$ many compilers
    e.g., concurrent processes, temporal logics, verification scripts
  - Memory and time efficiency are paramount
    $\Rightarrow$ Need to use C as our main development language
- Advantages:
  - Efficiency of SYNTAX
  - Efficiency & portability provided by the C language
  - Type discipline & safety of functional language
  - Constructor types, pattern-matching, static dataflow analysis

Compilers built using this approach

14 compilers developed at INRIA Grenoble since 2000 using TRAIAN and LNT



From 5000 to 55,000 lines of code

On average:

- 78 % of LNT code
- 10 % of SYNTAX code
- Only 12 % of C code

LNT: language for sequential & concurrent systems

- Derived from E-LOTOS (ISO 15437)
- Developed and used by CONVECS since 1995
- Each LNT specification is a set of modules
- Each module contains:
  - types
  - functions (instructions for sequential computations)

    Used for compilers
  - processes (behaviours for concurrent computations)
  - channels (types for process communications)

    Used for concurrent systems

- LNT types & functions = LNT data part

Overview of LNT data part

- A first-order functional language with an ADA-like imperative syntax
- Data types:
  - base types: **bool**, **nat**, **int**, **real**, **char**, **string**
  - free-constructor types used to define abstract trees
  - type combinator: **list**
- Instructions:
  - **in** / **out** / **in out** parameters
  - **return**
  - **require** (preconditions), **ensure** (postconditions), **assert**
  - local variable declarations
  - assignments to local variables and parameters
  - **if-then-else**, **case** (pattern matching)
  - **while** and **for** loops with **break**
  - **raise** and **trap** (exception handling)
- Easy connection to external C types and functions

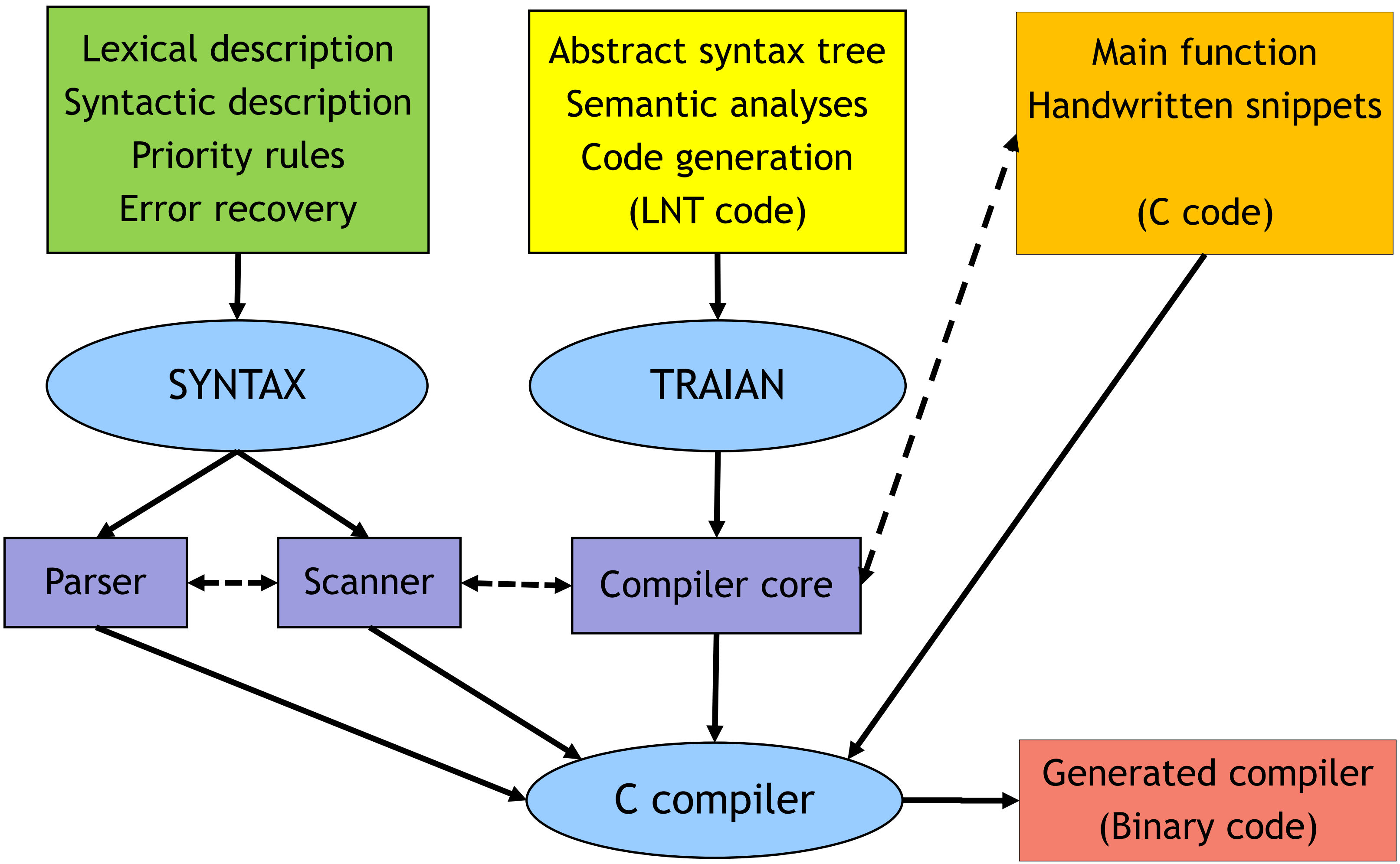
TRAIAN: a compiler for LNT data part

- Generates C code for LNT types & functions
- Developed since 1998

	1997	1998	2000		2016	2020		2022
TRAIAN releases		1.0	2.0	…	2.9	3.0	…	3.8

- Compiler written using SYNTAX + LNT (bootstrap)
  - 42,340 lines of code (TRAIAN 3.8)
  - Fast: TRAIAN compiles itself in one second
  - Strong static semantic checks: unused variables, uninitialized variables, dead code, etc.

Overview of the SYNTAX + TRAIAN compiler construction technology



Example: statements of a simple procedural language

Modular description using several files

- lang.lecl: lexical description (SYNTAX/LECL code)
- lang.tabc: syntax description & abstract tree construction (SYNTAX/TABC code)
- lang.lnt: abstract tree definition & traversals (LNT code)
- lang.tnt: definition of LNT external types (C code)
- lang.fnt: definition of LNT external functions (C code)

Example: external LNT types written in C

Excerpt of lang.lnt

```
type SYMBOL_TABLE is
    !external !implementedby "C_SYMTAB"
end type
```

Excerpt of lang.tnt

```
typedef struct  {
    char *name:
    C_TYPE *type;
} C_SYMTAB [MAX_ENTRIES];
```

Example: external LNT functions written in C

For functions with side effects, e.g., update of abstract tree (binding, type checking), code generation, error printing, etc.

Excerpt of lang.lnt

```
function PRINT_ERROR (S : STRING) is
  !external !implementedby "C_EXT_PRINT_ERROR"
end function
```

Excerpt of lang.fnt

```
void C_EXT_PRINT_ERROR (ERROR_MSG)
    char *ERROR_MSG;
{
  fprintf (stdout, "error : ");
  fprintf (stdout, ERROR_MSG) ;
  fprintf (stdout, "\n");
}
```

Example: abstract tree definition

Excerpt of lang.lnt

```
type ID is !implementedby "C_TYPE_ID"
  ID (SPELLING : STRING, LINE : NAT) !implementedby "C_ID"
end type

type EXPR is !implementedby "C_TYPE_EXPR"
  VAR (V : ID) !implementedby "C_VAR ",                          -- V
  INFIX (OP: ID, E1, E2 : EXPR) !implementedby "C_INFIX ",       -- E1  OP  E2
  …
end type

type STMT is !implementedby "C_TYPE_STMT"
  ASSIGN (V : ID, E : EXPR) !implementedby "C_ASSIGN",           -- V := E
  WHILE (E : EXPR, S0 : STMT) !implementedby "C_WHILE",          -- while E do S0
  IF (E : EXPR, S1, S2 : STMT) !implementedby "C_IF",            -- if E then S1 else S2
  …
end type
```

Example: syntax description & abstract tree construction

Excerpt of lang.tabc

```
$TABC_STMT (<stmt>) : C_TYPE_STMT ;
$TABC_ID (<id>) : C_TYPE_ID ;
$TABC_EXPR (<expr>) : C_TYPE_EXPR ;
```

attribute declarations

```
...
```

** BNF rules and attribute definitions*

```
<stmt> = <id> ":=" <expr> ;
$TABC_STMT (<stmt>)
    $TABC_STMT (<stmt>) = C_ASSIGN ($TABC_ID (<id>) ,  $TABC_EXPR (<expr>));
```

C statements that compute (synthesized) TABC attributes

```
<id> = %ID;
$TABC_ID (<id>)
    $TABC_ID (<id>) = C_ID ($pste ("%ID"), sxcurrent_parsed_line (0));
...
```

Example: lexical description

File lang.lecl

```
Classes
  SPACE        = SP + HT + NL + FF ;

Tokens
  Comments     = { SPACE | "-" "-" {^EOL}* EOL }+ ;

  %ID          = LETTER {["_"] (LETTER | DIGIT)}* ;

  %INT         = {DIGIT}+ ;
```

Abstract tree traversals
Example: type checking

Excerpt of lang.lnt

```
function CHECK_STMT (S : STMT, SYMB : SYMBOL_TABLE) : BOOL is
   case S var V : ID, E : EXPR, S0, S1, S2 : STMT, V_T, E_T : TYPE, CORRECT : BOOL in
     ASSIGN (V, E) ->
         V_T := CHECK_ID (V, SYMB); E_T := CHECK_EXPR (E, SYMB);
         CORRECT := (V_T == E_T) and (E_T != TYPE_ERROR);
         if not (CORRECT) then PRINT_ERROR ("type mismatch") end if;
         return CORRECT
   | IF_THEN_ELSE (E, S1, S2) ->
         E_T := CHECK_EXPR (E, SYMB);
         CORRECT := (E_T == BOOL_TYPE);
         if not (CORRECT) then PRINT_ERROR ("type mismatch") end if;
         return CORRECT and then CHECK_STMT (S1, SYMB) and then CHECK_STMT (S2, SYMB)
   | WHILE (E, S0) -> …
   end case
end function
```

Companion tool: Make-makefile

- Automatic generation of Makefiles
- Inputs:
  - The SYNTAX files present in the current directory (.tabc, .lecl, ...)
  - The LNT files present in the current directory
  - A custom configuration file Userfile
- Output: a specialized Makefile
- Supports several environments & cross compilation
  - Operating systems: Linux, macOS, Solaris, Windows
  - Processors: Intel or AMD x86 and x86-64, ARM (macOS)
  - Compilers: Gcc, Clang, Solaris cc, ...

Userfile for Make-makefile

PROGRAM=lang	Target program name
SXLEVEL=2 SXDIR=/common/Syntax SXMAIN=lang TABCFLAGS="-rhs 15" LNTLEVEL=1	SYNTAX flags
LNTDIR=/common/Traian LNTFLAGS= LNTMAIN=lang.lnt LNTFLAGS=-noindent	LNT flags
DEPEND="-I. -I$LNTDIR/incl" INCL='-I$(LNTDIR)/incl' LIB="-L \$(LNTDIR)/bin.\$(ARCH)/lotosnt_exceptions.o -lm"	C flags

Conclusion (1)

- Compiler construction based on INRIA tools:
  SYNTAX + LNT/TRAIAN
- Fast development
  - Simple and easy-to-learn technology
  - SYNTAX flexibility for parser generation:
    accept a large class of BNF grammars, powerful error recovery
- Maintainable and robust code
  - Readable LNT code
  - TRAIAN static checks: strong typing, detection of case exhaustivity,
    detection of uninitialized variables, detection of dead code, …
  - Direct manipulations of C pointers are avoided
  - Efficient generated code

Conclusion (2)

- Portability
  - Support for Linux, Mac, Solaris, and Windows
  - Standard C code generated
  - Straightforward interface with C
- Sustainability
  - SYNTAX is stable and mature
  - LNT / TRAIAN are stable and actively supported
- SYNTAX and TRAIAN are freely available:
  - https://sourcesup.renater.fr/projects/syntax
  - http://vasy.inria.fr/traian