A Simple Approach for Building Compiler Front-ends

Hubert Garavel

INRIA – Université Grenoble Alpes and DEPEND

http://convecs.inria.fr

Compiler architecture

Compiler architecture

source program (input)

lexical analysis

syntactic analysis

← compiler front-end

abstract syntax tree

semantic analyses:

identifier binding,

type checking,

data-flow checking, etc.

code generation

← compiler back-end

target program (output)

Basic facts

- The back-end is usually the most complex part
  (20% front-end, 80% back-end ?)

- Compiler authors have strong views about which language to use for the back-end:
  - traditionally: C/C++ (but too low-level)
  - today: Haskell, Java, Ocaml, LNT, Python, Rust, etc.

- What about the front-end?

Front-end construction

- Compiler front-ends may be programmed manually — but this is boring and error-prone
- In practice, one uses compiler-generation systems
  - lexical and syntactic descriptions using BNF grammars
  - tools generate analyzers from these grammars
  - BNF grammars must not be ambiguous
  - BNF grammars must follow restrictions: LL, LR, LALR
- Examples of tools:
  - Lex/Yacc (and Flex/Bison), ANTLR, SYNTAX, etc.

Front-end vs back-end tradeoffs

- Fixing a programming language for the back-end restricts the choice of tools for the front-end
- Wikipedia: Comparison of parser generators
  - 22 tools listed for lexer generation, but
    only one for Haskell (resp. Eiffel, Go, Rust)
  - 98 tools listed for parser generation, but

    only one for Erlang (resp. Common Lisp)
    only two for Ada (resp. Haskell, Swift)
- Not all lexer/parser generators are equal:
  - different restrictions on the BNFs accepted (e.g., LL vs LR)
  - some give user-friendly explanation of conflicts in grammar
  - some have automatic recovery from lexical/syntactic errors

Three possible solutions

- 1. Front-end and back-end in the same language
  - restricted choice of tools for the front-end
- 2. Front-end and back-end in compatible languages
  - example: CADP compilers
    front-end written in SYNTAX, back-end written in LNT
    both SYNTAX and LNT generate C code
- 3. Front-end and back-end in different languages
  - front-end builds an abstract tree (XML or JSON file)
  - back-end reads this file, then does semantic analyses

The third solution

- Advantages for compiler writers:
  - they can choose "their" language for the back end
  - they can chose the "best" tool for the front-end
  - front- and back-end are separate software modules
  - front-end and back-end can be developed in parallel (once the XML/JSON structure has been specified)
- Drawbacks:
  - performance penalty for communicating through files instead of through memory

Remainder of this talk

- A working implementation of the third solution
- Front-end done using INRIA's SYNTAX tool
- Abstract tree in XML or JSON format
  - few parser generators export XML or JSON files
- Simple, efficient, widely applicable

Introduction to the SYNTAX compiler-generation system

SYNTAX

- Likely, the oldest INRIA software still in activity
- Undertaken in 1972 (Algol60 $\rightarrow$ PL/1 $\rightarrow$ C)
- Large software: 1618 files 468,000 lines of code
  - bootstrapped (SYNTAX written using SYNTAX)
  - now maintained by CONVECS, with the help of Pierre Boullier, the main author of SYNTAX
- Two-level interface:
  - higher level: "processors" BNF, CSYNT, LECL...
  - lower level: C code libraries ("managers")

The LECL processor

- LECL produces tables for a scanner automaton that recognizes the tokens of the language
- The input language is expressive (more than Lex)

Ada-like comments

```
Comments = "-" "-" {^EOL} EOL ;
```

C-like #include

```
Include = "#" &Is_First_Col {space} "include" {space}
          QUOTE {^"\n"}+ QUOTE {space} EOL @1 ; @2 ;
Tokens
    %Integer = {DIGIT}+;
    Priority shift > reduce;
    %Ident = LETTER [ {LETTER | DIGIT} ] ;
    Context All But %Ident, %Integer;
```

The BNF processor

- BNF verifies that a context-free grammar is correct (symbols are well defined, productive, etc.)

fragment of a grammar
for the LUSTRE language

<const> = "false" ;
<const> = "true" ;
<const> = %Integer ;
<const> = %Real ;
<exp> = <const> ;
<exp> = <exp> "and" <exp> ;
<exp> = <exp> "+" <exp> ;
<exp> = "pre" <exp> ;
<exp> = <exp> "->" <exp> ;
<exp> = "if" <exp> "then" <exp> "else" <exp> ;
<exp> = %Ident "(" <exp_list> ")" ;

The CSYNT processor

- CSYNT produces tables for a parser automaton that recognizes the language of the BNF grammar
  - ascending deterministic analysis : LR(1) or LALR(1)
  - optimisation techniques to reduce the automaton
- In case of Shift/Reduce or Reduce/Reduce conflicts
  - 1. one may let CSYNT use its predefined resolution strategies (e.g., Shift > Reduce)
  - 2. one may modify the grammar
  - 3. one may use syntactic predicates and/or actions
  - 4. one uses the PRIO processor

The PRIO processor

- PRIO removes conflicts (ambiguities) in a BNF grammar using higher-level strategies:

priority rules for a LUSTRE-like language

%left "or"

%left "and"

%nonassoc "not"

%left "+" "-"

%left "*" "/" "div" "mod"

%nonassoc "<" "<=" "=" ">=" ">" "<>"

<exp> = "-" <exp> ; **%prec** "not"

The RECOR processor

- Automatic recovery of errors:
  - lexical : insertion-destruction-permutation of characters
  - syntactic: insertion-destruction-permutation of tokens
- This is a key feature of SYNTAX

```
5 :     A 1,2 := B(3*(I + 4, J*/K)
          ↑   ↑              ↑   ↑
          0   1              2   3

**** Warning (0) : "(" is inserted before "1".
**** Warning (1) : ")" is inserted before ":=".
**** Warning (2) : ")" is inserted before ",".
**** Error (3)  :   "%IDENTIFIER" is inserted before "/".

6 :     if I = 1 then then go to UP ;
        ↑             ↑
        0             1

**** Warning (0) : ";" is inserted before "if".
**** Warning (1) : "then" is deleted.
```

The "semantic" processors

- SYNTAX has three semantic processors:
  - SEMACT
  - SEMAT
  - SEMC (formerly named TABC)
- The CONVECS team uses SEMC:
  - BNF extended with typed synthesized attributes (Yacc only support a single attribute per non-terminal)
  - these attributes are computed by C code fragments
  - it is good practice to keep these fragments short

Abstract tree construction

Functionalities of the front-end

- 1. Parse the input program using SYNTAX:
  - LECL description of the lexer
  - BNF/SEMC description of the parser
  - PRIO description of priority rules in the BNF grammar
  - RECOR description of error-recovery rules
- 2. Output the abstract tree in XML or JSON format

Standard vs simple approach

- Standard approach:
  - specify the abstract tree as a data structure in memory
  - build this data structure using synthesized attributes (e.g., Yacc or SYNTAX) while parsing the input program
  - traverse the data structure and dump it to a file in XML or JSON format

  Drawback: the data structure is described three times
- Simple approach:
  - directly output the XML or JSON file while parsing the input program

Preliminary remark

- The abstract tree cannot be written to disk "on-the-fly", while reading the input program

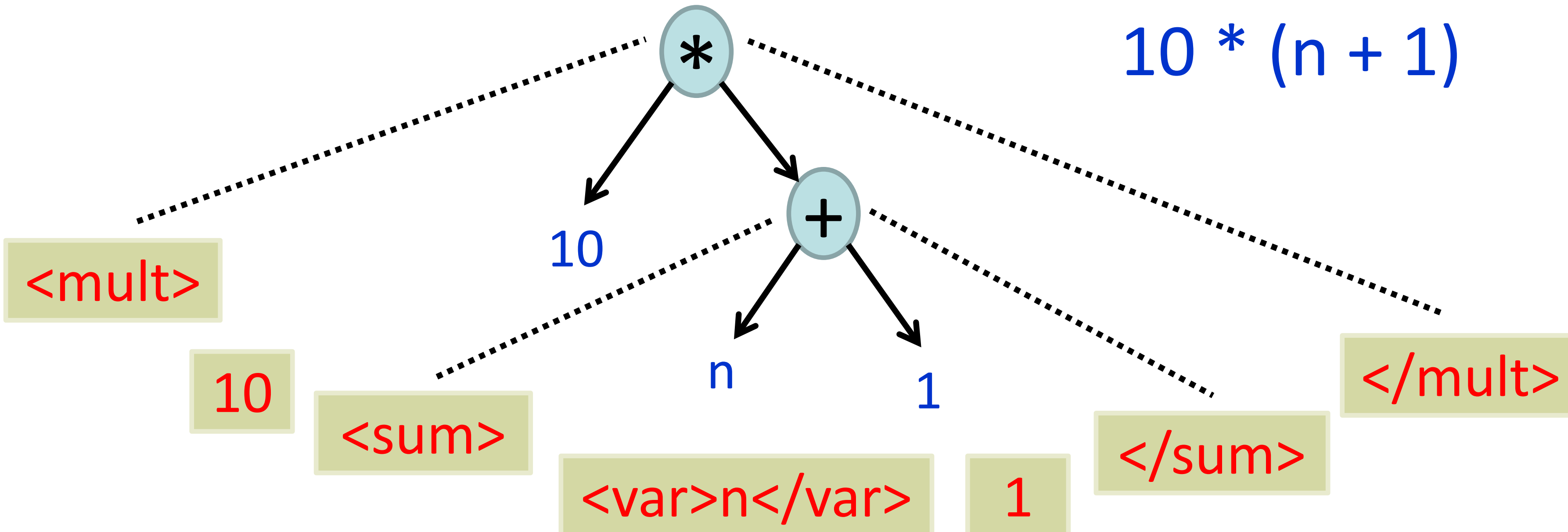
- Example:
  - input term: (n + 1)
  - output term: <sum><var>n</var>1</sum>

  One cannot write <sum> before having read "+"

  ⇒ unbounded lookahead is needed

  ⇒ XML output must be buffered in memory

Overview of the translation



- The abstract syntax tree is not built as a tree, but simply as the concatenation of many small text fragments (here, in XML) stored in memory
- These fragments are then dumped to a file

The new SXML library of SYNTAX

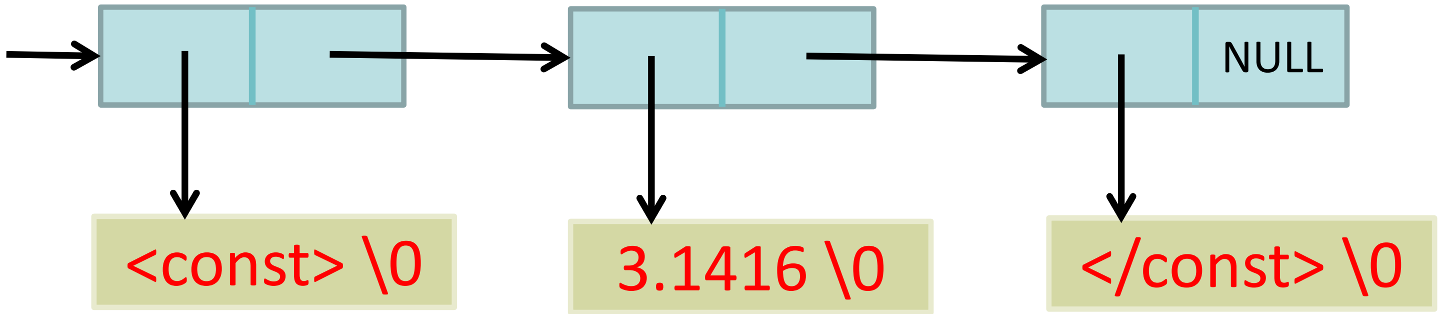
- SXML_TYPE_LIST: linked list whose elements are character strings (possibly of different lengths)

- SXML_PRINT: function that prints to a file the character strings contained in a linked list

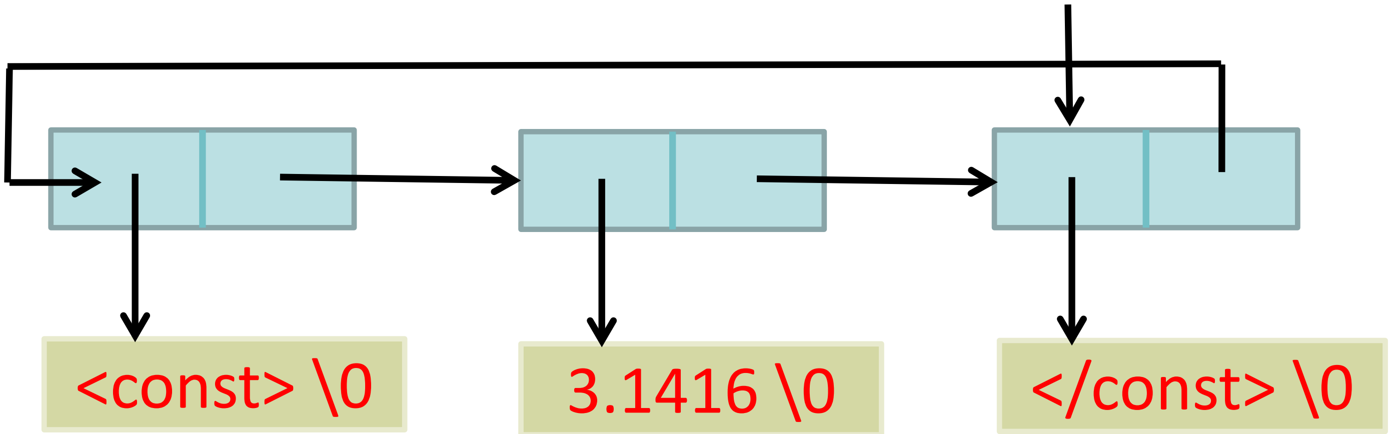
- SXML_T*, SXML_L*: concatenation functions taking one or many character strings and/or linked lists, and returning a linked list

Implementation of SXML

Straightforward implementation:



Clever implementation (concatenation in constant time):



LUSTRE example (1/4)

```
<Type> = "bool" ;
$LIST (<Type>)
        $LIST (<Type>) = SXML_T ("bool");
*------------------------------------------------------------------------
<Type> = "int" ;
$LIST (<Type>)
        $LIST (<Type>) = SXML_T ("int");
*------------------------------------------------------------------------
<Type> = "real" ;
$LIST (<Type>)
        $LIST (<Type>) = SXML_T ("real");
*------------------------------------------------------------------------
<Type> = %Ident ;
$LIST (<Type>)
        $LIST (<Type>) = SXML_T ($ptext ("%Ident"));
```

LUSTRE example (2/4)

```
<LocalDecls> = ;

$LIST (<LocalDecls>)

        $LIST (<LocalDecls>) = SXML_T ("<var></var>");

*----------------------------------------------------------------------------------------------------

<LocalDecls> = "var" <VarDeclList> ";" ;

$LIST (<LocalDecls>)

        $LIST (<LocalDecls>) = SXML_TLT ("<var>", $LIST (<VarDeclList>), "</var>");
```

LUSTRE example (3/4)

```
<Decl> = "function" <Header> <LocalDecls> <Equations> ";" ;

$LIST (<Decl>)

        $LIST (<Decl>) = SXML_TLLLT ("<function>",
                                     $LIST (<Header>),
                                     $LIST (<LocalDecls>),
                                     $LIST (<Equations>),
                                     "</function>");
```

LUSTRE example (4/4)

```
<Expr> = "not" <Expr> ;
$LIST (<Expr>)
        $LIST (<Expr>) = SXML_TLT ("<expr kind=\"not\">",
                            $LIST (<Expr>'), "</expr>");
*-------------------------------------------------------------------------------------
<Expr> = <Expr> "and" <Expr> ;
$LIST (<Expr>)
        $LIST (<Expr>) = SXML_TLLT ("<expr kind=\"and\">",
                            $LIST (<Expr>'), $LIST (<Expr>''),"</expr>");
*-------------------------------------------------------------------------------------
<Expr> = "if" <Expr> "then" <Expr> "else" <Expr> ;
$LIST (<Expr>)
        $LIST (<Expr>) = SXML_TLLLT ("<expr kind=\"if\">",
                            $LIST (<Expr>'), $LIST (<Expr>''), $LIST (<Expr>'''), "</expr>");
```

Conclusion

SYNTAX + SXML

- A concise solution to build compiler front-ends
- A single file for concrete and abstract syntaxes
  - no need to define the abstract syntax tree separately
- Already used for two compiler front-ends:
  - LUSTRE $\rightarrow$ XML
  - FORTRAN 77 $\rightarrow$ JSON (ongoing work)
- Also applicable to Yaml or other custom formats