Technical Report no. 2004-09

BNF Converter
Multilingual Front-End Generation from Labelled BNF Grammars

Michael Pellauer, Markus Forsberg and Aarne Ranta

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2004

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2004-09
ISSN 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Göteborg, Sweden, 2004

Abstract

The BNF Converter is a compiler-construction tool that uses a Labelled BNF grammar as the single source of definition to extract the abstract syntax, lexer, parser and pretty printer of a language. The added layer of abstraction allows it to perform multilingual code generation. As of version 2.0 it is able to output front ends in Haskell, Java, C or C++.

1 Introduction

Language implementors have long used generative techniques to implement parsers. However, with advances in language design the focus of the compiler front end has shifted from the parsing of difficult languages to the definition of a complex abstract-syntax-tree data structure. It is the common practise for modern implementors to use one tool to generate an abstract syntax tree, another to generate a lexer, and a third to generate a parser.

Yet this requires that the implementor learn three separate configuration syntaxes, and maintain disparate source files across changes to the language definition. The BNF Converter[1] is a compiler-construction tool based on the idea that from a single source grammar it is possible to generate both an abstract syntax tree definition, including a traversal function, and a concrete syntax, including lexer, parser and pretty printer.

The decoupling of the grammar description from the implementation language allows our tool to perform multilingual code generation. As of version 2.0 the BNF Converter is able to generate a front end in Haskell, Java, C, or C++. This continues the tradition of Andrew Appel [1, 2, 3], whose textbooks apply the same compiler methodology across three widely different target languages.

The BNF Converter Approach.

With the BNF Converter the user specifies a grammar using an enhanced version of Backus Naur Form called Labelled BNF (LBNF), described in Section 2. This grammar is language independent and serves as a single source for all language definition changes, increasing maintainability. After the user selects a target language it is used generate the following:

- Abstract syntax tree data structure
- Lexer and parser specification
- Pretty printer and traversal skeleton
- Test bench and Makefile

[1]Available from the BNF Converter website [7]

- Language documentation

This unified approach to generation offers many advantages. First of all, the increased level of abstraction allows our tool to check the grammar for problems, rather than attempting to check code written directly in an implementation language like C. Secondly, the components are generated to interoperate correctly together with no additional work from the user. Packages such as the abstract syntax and pretty printer can be supplied as development frameworks to encourage applications to make use of the new language.

Combined with BNF Converter 2.0's multiingual generation this facilitates interesting possibilities, such as using a server application written in C++ to pretty-print output that will be parsed by a Java application running on a PDA. The language maintainers themselves can experiment with implementing the same methodology over multiple languages, even creating a prototype language implementation in Haskell, then switching to C for development once the language definition has been finalized.

This paper gives an overview of the LBNF grammar formalism. We then compare the methodology the BNF Converter uses to produce code in Haskell, Java, C++, and C, highlighting some of the differences of generating a compiler in these languages. Finally, we conclude with a discussion of our practical experiences using the tool in education and language prototyping.

Language Describability.

The requirements that the BNF Converter puts on a language in order to describe it are simple and widely accepted: the syntax must be definable by a context-free grammar and the lexical structure by a regular expression. The parser's semantic actions are only used for constructing abstract syntax trees and can therefore not contribute to the definition of the language. Toy languages in compiler text books are usually designed to meet these criteria, and the trend in real languages is to become closer to this ideal.

Often it is possible to use preprocessing to turn a language that almost meets the criteria into one that meets them completely. Features such as layout syntax, for example, can be handled by adding a processing level between the lexer and the parser. Our experiences with real-world languages are discussed in Section 8.

2 The LBNF Grammar Formalism

The input to the BNF Converter is a specification file written in the LBNF grammar formalism. LBNF is an entirely declarative language designed to combine the simplicity and readability of Backus Naur Form with a handful of features to hasten the development of a compiler front-end.

Besides declarativity, we find it important that LBNF has its own semantics, instead of only getting its meaning through translations to Haskell, Java, C, etc. This means, among other things, that LBNF grammars are type checked on the source, so that semantic errors do not appear unexpectedly in the generated code. Full details on LBNF syntax and semantics are given in [8], as well as on the BNF Converter homepage [7].

2.1 Rules and Labels

At the most basic level, an LBNF grammar is a BNF grammar where every rule is given a *label*. The label is an identifier used as the constructor of syntax trees whose subtrees are given by the non-terminals of the rule; the terminals are just ignored. As a first example, consider a rule defining assignment statements in C-like languages:

```
SAssign. STM ::= Ident "=" EXP ;
```

Apart from the label `SAssign`, the rule is an ordinary BNF rule, with terminal symbols enclosed in double quotes and non-terminals written without quotes. A small, though complete example of a grammar is given in Section 2.4.

Some aspects of the language belong to its lexical structure rather than its grammar, and are described by regular expressions rather than by BNF rules. We have therefore added to LBNF two rule formats to define the lexical structure: *tokens* and *comments* (Section 2.2).

Creating an abstract syntax by adding a node type for every BNF rule may sometimes become too detailed, or cluttered with extra structures. To remedy this, we have identified the most common problem cases, and added to LBNF some extra conventions to handle them (Section 2.3).

Finally, we have added some *macros*, which are syntactic sugar for potentially large groups of rules and help to write grammars concisely, and some *pragmas*, such as the possibility to limit the entrypoints of the parser to a subset of nonterminals.

2.2 Lexer Definitions

The `token` definition format.

The `token` definition form enables the LBNF programmer to define new lexical types using a simple regular expression notation. For instance, the following defines the type of identifiers beginning with upper-case letters.

```
token UIdent (upper (letter | digit | '\_')*) ;
```

The type `UIdent` becomes usable as an LBNF nonterminal and as a type in the abstract syntax. Each token type is implemented by a `newtype` in Haskell, as a `String` in Java, and as a `typedef` to `char*` in C/C++.

Predefined token types.

To cover the most common cases, LBNF provides five predefined token types:

> `Integer`, `Double`, `Char`, `String`, `Ident`

These types have predefined lexer rules, but could also be defined using the regular expressions of LBNF (see [8]). In the abstract syntax, the types are represented as corresponding types in the implementation language; `Ident` is treated like user-defined token types. Only those predefined types that are actually used in the grammar are included in the lexer and the abstract syntax.

The `comment` definition format.

Comments are segments of source code that include free text and are not passed to the parser. The natural place to deal with them is in the lexer. A `comment` definition instructs the lexer generator to treat certain pieces of text as comments.

The comment definition takes one or two string arguments. The first string defines how a comment begins. The second, optional string marks the end of a comment; if it is not given then the comment is ended by a newline. For instance, the Java comment convention is defined as follows:

```
comment "//" ;
comment "/*" "*/" ;
```

2.3 Abstract Syntax Conventions

Semantic dummies.

Sometimes the concrete syntax of a language includes rules that make no semantic difference. For instance, the C language accepts extra semicolons after statements. We do not want to represent these extra semicolons in the abstract syntax. Instead, we use the following convention:

> If a rule has only one non-terminal on the right-hand-side, and this non-terminal is the same as the value type, then it can have as its label an underscore (_), which does not add anything to the syntax tree.

Thus, we can write the following rule in LBNF:

```
_ . STM ::= STM ";" ;
```

Precedence levels.

A common idiom in (ordinary) BNF is to use indexed variants of categories to express precedence levels, e.g. `EXP, EXP2, EXP3`. The precedence level regulates the order of parsing, including associativity. An expression belonging to a level n can be used on any level $< n$ as well. Parentheses lift an expression of any level to the highest level.

Distinctions between precedence levels and moving expressions between them can be defined by BNF rules, but we do not want these rules to clutter the abstract syntax. Therefore, we can use semantic dummies (`_`) for the transitions, together with the following convention:

> A category symbol indexed with a sequence of digits is treated as a type synonym of the corresponding non-indexed symbol.

A non-indexed symbol is treated as having the level 0. The following grammar shows how the convention works in a familiar example with arithmetic sums and products:

```
EPlus.  EXP  ::= EXP  "+" EXP2 ;
ETimes. EXP2 ::= EXP2 "*" EXP3 ;
EInt.   EXP3 ::= Integer ;
_.      EXP  ::= EXP2 ;
_.      EXP2 ::= EXP3 ;
_.      EXP3 ::= "(" EXP ")" ;
```

The indices also guide the pretty-printer to generate a correct, minimal number of parentheses.

The `coercions` macro provides a shorthand for generating the dummy transition rules concisely. It takes as its arguments the unindexed category and the highest precedence level. So the final three rules in the above example could be replaced with:

```
coercions EXP 3 ;
```

Polymorphic lists.

It is easy to define monomorphic list types in LBNF:

```
NilDEF.  ListDEF ::= ;
ConsDEF. ListDEF ::= DEF ";" ListDEF ;
```

But LBNF also has a *polymorphic list notation.* It follows the Haskell syntax but is automatically translated to native representations in Java, C++, and C.

```
[].  [DEF] ::= ;
(:). [DEF] ::= DEF ";" [DEF] ;
```

The basic ingredients of this notation are

> `[`C`]`, the category of lists of type C,
>
> `[]` and `(:)`, the Nil and Cons rule labels,
>
> `(:[])`, the rule label for one-element lists.

The list notation can also be seen as a variant of the Kleene star and plus, and hence as an ingredient from *Extended BNF* in LBNF.

Using the polymorphic list type makes BNF Converter perform an automatic optimization: *left-recursive lists*. Standard lists in languages like Haskell are right-recursive, but LR parsers favor left-recursive lists because they save stack space. BNF Converter allows programmers to define familiar right-recursive lists, but translates them into left-recursive variants in parser generation. When used in another construction, the list is automatically reversed. The code examples below, generated from the grammar in Section 2.4, show how this works in the different parser tools.

The `terminator` and `separator` macros.

The `terminator` macro defines a pair of list rules by what token terminates each element in the list. For instance,

```
terminator STM ";" ;
```

is shorthand for the pair of rules

```
[].  [STM] ::= ;
(:). [STM] ::= STM ";" [STM] ;
```

The `separator` macro is similar, except that the separating token is not expected after the last element of the list. The qualifier `nonempty` can be used in both macros to make the one-element list the base case.

2.4 Example Grammar

A small example LBNF grammar is given in Figure 1. It describes a language of boolean expressions, perhaps written as part of a larger grammar. In this small language a PROGRAM is simply a list of expressions terminated by semicolons. The expressions themselves are just logical AND and OR of true, false, or variable names represented by the LBNF built-in type `Ident`.

This example, though small, is representative because it uses both polymorphic lists and precedence levels (the AND operator having higher precedence than OR). We will use this single source example to explore BNF Converter's generation methodology across multiple implementation languages.

```
PROGRAM. PROGRAM ::= [EXP] ;

EOr.    EXP  ::=
           EXP "||" EXP1 ;
EAnd.   EXP1 ::=
           EXP1 "&&" EXP2 ;
ETrue.  EXP2 ::= "true" ;
EFalse. EXP2 ::= "false" ;
EVar.   EXP2 ::= Ident ;
terminator EXP ";" ;
coercions EXP 2 ;
```

Figure 1: LBNF Source code for all examples

3 Haskell Code Generation

The process the BNF Converter uses to generate Haskell code is quite straightforward. Here we will only present an overview of this process, for comparison with the methods used for Java and C. For a more complete look at this process see the documentation on the BNF Converter Homepage [7].

The Abstract Syntax.

Consider the example grammar given in Section 2.4.

The Haskell abstract syntax generated by the BNF Converter, shown in Figure 2A, is essentially what a Haskell programmer would write by hand, given the close relationship between a declarative grammar and Haskell's algebraic data types.

The Lexer and Parser.

The BNF Converter generates lexer and parser specifications for the Alex [6] and Happy [15] tools. The lexer file (omitted for space considerations) consists mostly of standard rules for literals and identifiers, but has rules added for reserved words and symbols (i.e. terminals occurring in the grammar), regular expressions defined in token definitions, and comments.

The Happy specification (Figure 2B) has a large number of token definitions, followed by parsing rules corresponding closely to the source BNF rules. Note the left-recursive list transformation, as defined in Section 2.3.

The Pretty Printer and Case Skeleton.

The pretty printer consists of a Haskell class `Print` with instances for all generated data types, taking precedence into account. The class method `prt` generates a list of strings for a syntax tree of any type (Figure 2C).

The list of strings is then put in layout (indentation, newlines) by a *rendering* heuristic, which is generated independently of the grammar. This function is designed to make C-like languages look good by default, but it is written with easy modification in mind.

The case skeleton (Figure 2D) is a simple traversal of the abstract syntax tree representation that can be used as a template when defining the compiler back end, e.g. type checker and code generator. The same methodology is also used to generate the pretty printer. The case branches in the skeleton are initialized to fail, and the user can simply replace them with something more interesting.

The Makefile and Test Bench.

The generated test bench file can be loaded in the Haskell interpreter `hugs` to run the parser and the pretty printer on terminal or file input. If parsing succeeds the test functions display a syntax tree, and the pretty printer linearization. Otherwise an error message is displayed.

A simple makefile is created to run Alex on the lexer, Happy on the parser, and LaTeX on the document, by simply typing `make`. The `make clean` command removes the generated files.

Translation Summary.

Overall, it is easy to represent an LBNF grammar as a Haskell data type—a straightforward translation between source productions and algebraic data types. Language implementors have long known that the similarities between algebraic data types and grammar specifications make functional programming a good choice for compilers.

4 Java Code Generation

Translating an LBNF grammar into an object-oriented language is less straightforward. Appel outlines two possible approaches to abstract syntax representation in *Modern Compiler Implementation in Java* [2].

In the first method, which Appel refers to as the "Object-Oriented method," there is one Java class for each rule in the language grammar. Each class inherits from a single superclass, and each class defines operations on itself. For instance, if our compiler were to translate to SPARC and Intel assembly code each class would have a method `toSPARC()` and `toIntel()` that would translate itself to the appropriate representation. The advantage of this method is that it is easy to add new language categories. The user may add new classes containing the appropriate methods without altering

A. Abstract Syntax

```
data PROGRAM = PROGRAM [EXP]
   deriving (Eq, Show)
data EXP =
    EOr EXP EXP
  | EAnd EXP EXP
  | ETrue
  | EFalse
  | EVar
   deriving (Eq, Show)
```

B. Happy Parser

```
PROGRAM :: { PROGRAM }
PROGRAM : ListEXP { PROGRAM (reverse $1) }
EXP :: { EXP }
EXP : EXP '||' EXP1 { EOr $1 $3 }
  | EXP1 { $1 }
EXP1 :: { EXP }
EXP1 : EXP1 '&&' EXP2 { EAnd $1 $3 }
  | EXP2 { $1 }
EXP2 :: { EXP }
EXP2 : 'true' { ETrue }
  | 'false' { EFalse }
  | Ident { EVar $1 }
  | '(' EXP ')' { $2 }
ListEXP :: { [EXP] }
ListEXP : {- empty -} { [] }
  | ListEXP EXP ';' { flip (:) $1 $2 }
```

C. Pretty Printer

```
instance Print PROGRAM where
  prt i e = case e of
   PROGRAM exp -> prPrec i 0
                   (concat [prt 0 exp])
instance Print EXP where
  prt i e = case e of
   EOr exp0 exp -> prPrec i 0
    (concat
      [prt 0 exp0 , ["||"] , prt 1 exp])
   EAnd exp0 exp -> prPrec i 1
    (concat
      [prt 1 exp0 , ["&&"] , prt 2 exp])
   ETrue  -> prPrec i 2
      (concat [["true"]])
   EFalse  -> prPrec i 2
      (concat [["false"]])
   EVar id -> prPrec i 2
      (concat [prt 0 id])
  prtList es = case es of
   [] -> (concat [])
   x:xs ->
    (concat
      [prt 0 x , [";"] ,
       prt 0 xs])
```

D. Case Skeleton

```
transPROGRAM :: PROGRAM -> Result
transPROGRAM x = case x of
  PROGRAM exp  -> failure x
transEXP :: EXP -> Result
transEXP x = case x of
  EOr exp0 exp  -> failure x
  EAnd exp0 exp  -> failure x
  ETrue  -> failure x
  EFalse  -> failure x
  EVar id  -> failure x
```

Figure 2: Haskell source code fragments generated from Figure 1

existing definitions. The disadvantage is that it can be hard to add new syntax tree traversals. Adding a function `toAlpha()` for instance, could result in editing hundreds of classes.

In the second "syntax separate from interpretations" method, there is still one Java class for each grammar rule, but now classes are simply empty data structures with no methods aside from a constructor. Translation functions are removed from the data structure, and traverse the tree by straightforward manner. With this method it is easy to add new traversals, and these functions can make better use of context information than single objects' methods. The disadvantage is that adding new language constructs requires editing all existing traversal functions to handle the new cases.

However, the BNF Converter, which makes the grammar the central point of all language changes, lessens this disadvantage. Additionally, since translation functions are now traversals, it is easy for our tool to generate skeleton functions as we do in Haskell and for the user to reuse the template in all transformations.[2] Therefore the BNF Converter uses this method in generating Java (and C++) abstract syntax.

Java Abstract Syntax Generation.

Let us return to our example of Boolean Expressions from earlier (Section 2.4). Given this grammar, the BNF Converter will generate the abstract syntax found in Figure 3A, following Appel's method.

There are several differences between this transformation and the Haskell version that should be highlighted. First, experienced Java programmers will quickly notice that all the generated classes are public, and in Java public classes must go into their own `.java` file, with class name matching the file name. Since it common to have hundreds of productions in an LBNF grammar, the user's source directory can quickly become cluttered, so Abstract Syntax classes are placed into a sub-package called `Absyn`, and thus must be kept in a file-system subdirectory of the same name, which the tool creates.

There is a second difference in the code in Figure 3A: names. Classes in Java have instance variables and parameters, and all of these require unique names (whereas in Haskell data structures the names are only optional). First, we realize that parameter names generally are not important—we can simply give them the name "p" plus a unique number. The names of instance variables, on the other hand, do matter. The BNF Converter converts the type name to lowercase and adds an underscore to prevent conflicts with reserved words. If there is more than one variable of a type then they are numbered. Thus, the classes `EPlus` and `ETimes` have members `exp_1` and `exp_2`.

Notice that Appel's method uses public instance variables, which may be regarded as bad style by object-oriented programmers today. We have chosen to remain with the original method, both to keep a higher correspondence to the textbook, and to ease the generation of the pretty printer and other traversals.

Finally recall that Java 1.4 does not support polymorphic lists. Generic types is supported in the Java 2 Platform, Standard Edition 1.5 release, also implemented in BNF Converter (see section 5). The BNF Converter Java 1.4 backend generates simple null-terminated linked lists for each list that

[2]Of course, if the user implements a translation and then modifies the language definition they must still change the implemented code to reflect the modifications. However, they can refer to the template function in order to locate the differences.

the grammar uses. These special classes are prefixed with "List," such as the class `ListEXP` above, which takes the place of Haskell's [EXP].

The Lexer and Parser.

The BNF Converter generates specification files for the JLex [4] and CUP [12] tools which create a lexer and parser in a manner similar to the Haskell version. The difference between the tools is mainly a matter of syntax. For example, CUP cannot work with strings directly but requires terminal symbols be defined for each language symbol or reserved word. Also, CUP does not refer to variables with $ variables like Bison, but rather by assigning names to all possibly-used values. Specifications equivalent to the Happy code in Figure 2B is shown in Figures 3B.

The Java Pretty Printer and Skeleton Function.

Similar to the Haskell version, the Java pretty printer linearizes the abstract syntax tree using some easily-modifiable heuristics. It follows the method Appel outlines, using Java's `instanceof` operator to determine which subclass it is dealing with, then down-casting and working with the public variables. For example, the code to pretty-print an EXP is found in Figure 3D.

However, the pretty printer alone is not enough to test the correctness of a parse. In the Haskell version the built-in `show` function is used to print out the abstract syntax tree so that the programmer can confirm its correctness. We could use Java's `toString()` method in a similar role, but this is not satisfying, as it is generally used for debugging purposes. Instead, the BNF Converter adds a second method to the pretty printer, similar to Haskell's `show` function, shown in Figure 3E.

Throughout both methods the generated code makes use of Java's `StringBuffer` class to efficiently build the result of the linearization.

This `instanceof` method is also used to generate a code skeleton. However, this method may seem awkward to many object-oriented programmers, who are often taught to avoid `instanceof` wherever possible.

Much more familiar is the *Visitor Design Pattern* [11]. In it each member of the abstract syntax tree implements an `accept` method, which then calls the appropriate method in the visiting class (double-dispatch).

There is no reason that these two methods cannot live side by side. Therefore the BNF Converter generates code skeletons using both Appel's method and a Visitor interface and skeleton (Figure 3F).

Most familiar Visitor Design Patterns use a Visitee-traversal algorithm. That is to say, visiting the top member of a list will automatically visit all the members of the list. However, the BNF Converter-generated pattern uses Visitor-traversal. This means that it is the Visitor's responsibility, when visiting a list, to visit all the members in turn. This is because certain algorithms that compilers want to implement are not compositional, so performing a transformation on a single member may be quite different than performing that transformation on a certain pattern of nodes. For example, during peephole analysis a compiler may wish to merge to subsequent additions into a single operation, but may want to leave single additions unchanged. In our experience, these types of algorithms are easier to implement if the Visitor itself is in control of the traversal.

The Test Bench and Makefile.

With the pretty printer defined it is trivial to define a test bench and makefile to compile the code. However, the lack of an interactive environment such as Haskell's `hugs` means that the user is not able to specify which parser is used. Instead the first-defined entry-point of the grammar is used by default. However it is easy for the user to specify another entry point directly in the test bench source code.

Translation Summary.

Overall, translating from a declarative grammar to an object-oriented abstract syntax definition is possible, however the translation introduces a number of new complications such as the names of instance variables. A comparison of Figure 2A and Figure 3A emphasizes the challenges of implementing a compiler in Java.

The BNF Converter tries to deal with these complications in a consistent way to ease the implementation of the rest of the compiler. Appel's syntax-separate-from-interpretations method introduces several conventions that object-oriented programmers may find confusing at first. However, in practice the ease of using the generated transformation templates should help users to quickly overcome these difficulties.

5 Java 1.5 Generation

The Java backend has been adapted to Java 1.5 by Björn Bringert at Computing Science, Chalmers. The main difference is *generic types.* Generic types ensure type safety without having to resort to monomorphic types. For example, the container types in Java 1.5 are parameterized by a type `T`. Compare this with Java 1.4 where all objects in a container are of type `Object`. Furthermore, some adaptions were needed to reflect these changes, in particular in the syntax tree traversal.

A. Abstract Syntax

```
public class PROGRAM {
  public ListEXP listexp_;
  public PROGRAM(ListEXP p1)
            { listexp_ = p1; }
}
public abstract class EXP {}
public class EAnd extends EXP {
  public EXP exp_1, exp_2;
  public EAnd(EXP p1, EXP p2)
     { exp_1 = p1; exp_2 = p2; }
}
public class EOr extends EXP {
  public EXP exp_1, exp_2;
  public EOr(EXP p1, EXP p2)
     { exp_1 = p1; exp_2 = p2; }
}
public class ETrue extends EXP {
  public ETrue() { }
}
public class EFalse extends EXP {
  public EFalse() { }
}
public class EVar extends EXP {
  public String ident_;
  public EVar(String p1)
               { ident_ = p1; }
}
public class ListEXP {
  public EXP exp_;
  public ListEXP listexp_;
  public ListEXP(EXP p1, ListEXP p2)
    { exp_ = p1; listexp_ = p2; }
}
```

B. CUP Parser

```
terminal _SYMB_0;    //   ||
terminal _SYMB_1;    //   &&
terminal _SYMB_2;    //   ;
terminal _SYMB_3;    //   (
terminal _SYMB_4;    //   )
terminal _SYMB_5;    //   false
terminal _SYMB_6;    //   true
terminal String _IDENT_;
PROGRAM ::= ListEXP:p_1  {:
    if (p_1 != null) p_1 = p_1.reverse();
      RESULT = new Absyn.PROGRAM(p_1); :}
;
EXP ::= EXP:p_1 _SYMB_0 EXP1:p_3  {:
     RESULT = new Absyn.EOr(p_1, p_3); :}
  | EXP1:p_1  {:  RESULT = (p_1); :}
;
EXP1 ::= EXP1:p_1 _SYMB_1 EXP2:p_3  {:
    RESULT = new Absyn.EAnd(p_1, p_3); :}
  | EXP2:p_1  {:  RESULT = (p_1); :}
;
EXP2 ::= _SYMB_6 {:
           RESULT = new Absyn.ETrue(); :}
  | _SYMB_5 {:
          RESULT = new Absyn.EFalse(); :}
  | _IDENT_:p_1 {:
         RESULT = new Absyn.EVar(p_1); :}
```

CUP Parser (continued)

```
  | _SYMB_3 EXP:p_2 _SYMB_4 {:
                      RESULT = (p_2); :}
;
ListEXP ::= /*empty*/{: RESULT = null; :}
  | ListEXP:p_1 EXP:p_2 _SYMB_2  {:
 RESULT = new Absyn.ListEXP(p_2, p_1); :}
;
```

C. Pretty Printer

```
  private static void
  pp(Absyn.EXP exp, int _i_) {
    if (exp instanceof Absyn.EOr) {
       Absyn.EOr eor = (Absyn.EOr) exp;
       if (_i_ > 0) render(_L_PAREN);
       pp(eor.exp_1, 0);
       render("||");
       pp(eor.exp_2, 1);
       if (_i_ > 0) render(_R_PAREN);
    }
    if (exp instanceof Absyn.EAnd) {
       Absyn.EAnd eand = (Absyn.EAnd) exp;
       if (_i_ > 1) render(_L_PAREN);
       pp(eand.exp_1, 1);
       render("&&");
    ...
```

D. Abstract Syntax Viewer

```
  private static void sh(Absyn.EXP exp)
  {
    if (exp instanceof Absyn.EOr) {
       Absyn.EOr eor = (Absyn.EOr) exp;
       render("(");
       render("EOr");
       sh(eor.exp_1);
       sh(eor.exp_2);
       render(")");
    }
    if (exp instanceof Absyn.EAnd) {
       Absyn.EAnd eand = (Absyn.EAnd) exp;
       render("(");
       render("EAnd");
    ...
```

E. Visitor Design Pattern

```
  public void visitEOr(Absyn.EOr eor) {
    /* Code For EOr Goes Here */
    eor.exp_1.accept(this);
    eor.exp_2.accept(this);
  }
  public void visitEAnd(Absyn.EAnd eand) {
    /* Code For EAnd Goes Here */
    ...
  public void
  visitListEXP(Absyn.ListEXP listexp) {
    while(listexp!= null) {
      /* Code For ListEXP Goes Here */
      listexp.listexp_.accept(this);
      listexp = listexp.listexp_;
      ...
```

Figure 3: Java source code fragments generated from Figure 1

6 C++ Code Generation

With the Java version implemented it was straightforward to add support for C++ generation, using Flex [10] and Bison [9]. This translation is similar to the Java version—the main difference being the additional complications of destructors and the separation of interface (.H) and implementation (.cpp) files. The details of this translation have been omitted for space considerations but may be found on the BNF Converter homepage [7].

7 C Code Generation

The Abstract Syntax.

The translation to C code is quite different than the other languages. It follows the methodology used by Appel in the C Version of his textbook [1].

In this methodology, each grammar category is represented by a C `struct`. Each struct has an enumerated type indicating which LBNF label it represents, and a `union` of pointers to all corresponding non-terminal categories. Our boolean-expressions example generates the structs shown in Figure 4A. Structs are originally named with an underscore, and `typdef` declarations clean up the code by making the original grammar name refer to a pointer to that struct.

Data structure instances are created by using constructor functions, which are generated for each struct (Figure 4B). These functions are straightforward to generate and take the place of the `new` operator and constructors in an object-oriented language.

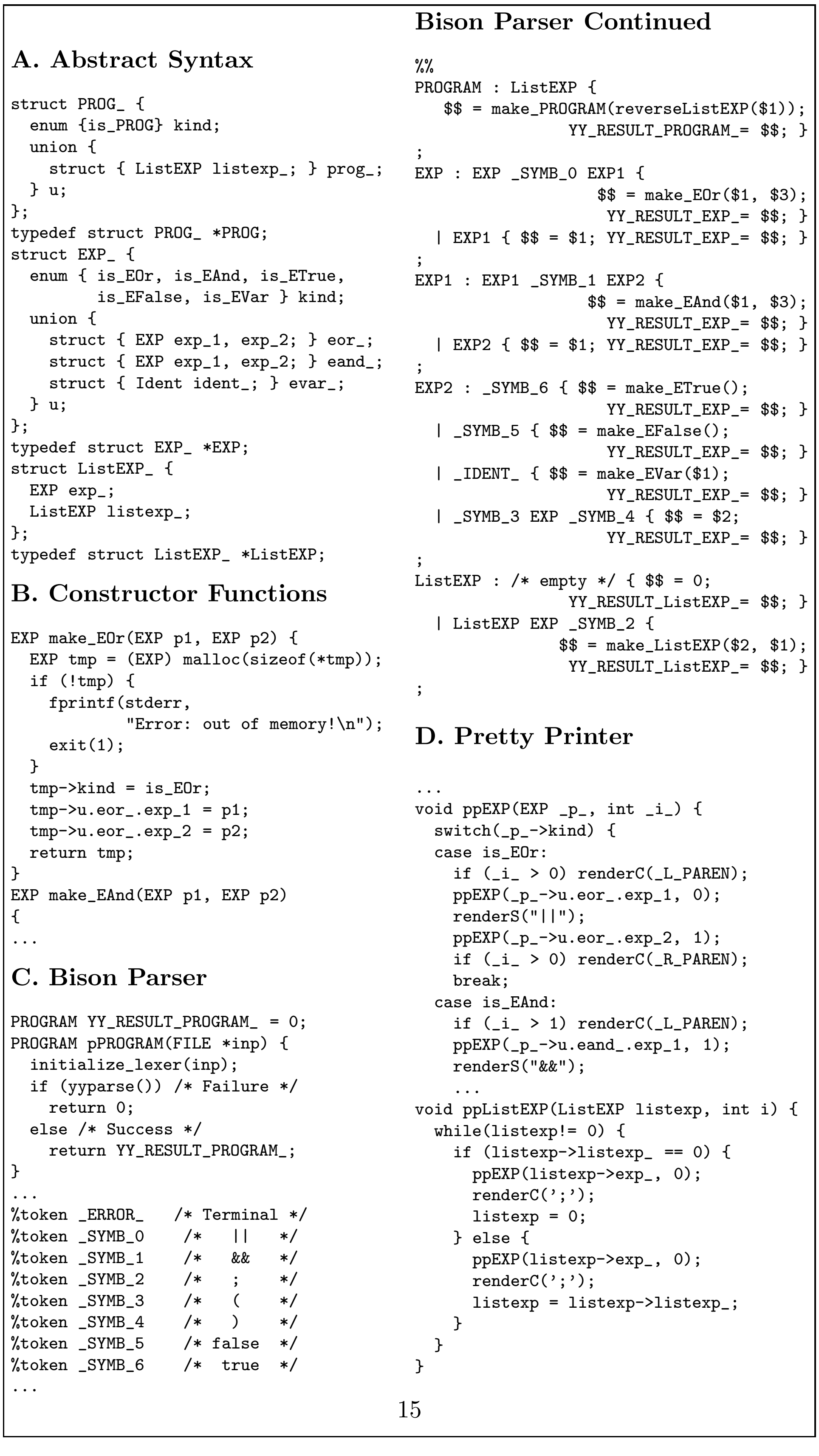
The Lexer and Parser.

The BNF Converter also generates a lexer specification file for Flex and a parser specification file for Bison. Figure 4C shows specification code equivalent to the examples in Figures 2B and 3B.

One complication is that there is no way to access the result of the parse without storing a global pointer to it. This means that every potential entry point production must store a pointer to the parse (the `YY_RESULT` variables in Figure 4C), in case they are the final successful category. Users can limit the performance impact of this by using the `entrypoints` pragma.

The Pretty Printer and Case Skeleton.

Any algorithm that wishes to traverse the tree must switch on the `kind` field of each node, then recurse to the appropriate members. For example, Figure 4E shows the pretty-printer traversal. The abstract syntax tree viewer and skeleton template are similar traversals.



A. Abstract Syntax

```
struct PROG_ {
  enum {is_PROG} kind;
  union {
    struct { ListEXP listexp_; } prog_;
  } u;
};
typedef struct PROG_ *PROG;
struct EXP_ {
  enum { is_EOr, is_EAnd, is_ETrue,
         is_EFalse, is_EVar } kind;
  union {
    struct { EXP exp_1, exp_2; } eor_;
    struct { EXP exp_1, exp_2; } eand_;
    struct { Ident ident_; } evar_;
  } u;
};
typedef struct EXP_ *EXP;
struct ListEXP_ {
  EXP exp_;
  ListEXP listexp_;
};
typedef struct ListEXP_ *ListEXP;
```

B. Constructor Functions

```
EXP make_EOr(EXP p1, EXP p2) {
  EXP tmp = (EXP) malloc(sizeof(*tmp));
  if (!tmp) {
    fprintf(stderr,
            "Error: out of memory!\n");
    exit(1);
  }
  tmp->kind = is_EOr;
  tmp->u.eor_.exp_1 = p1;
  tmp->u.eor_.exp_2 = p2;
  return tmp;
}
EXP make_EAnd(EXP p1, EXP p2)
{
...
```

C. Bison Parser

```
PROGRAM YY_RESULT_PROGRAM_ = 0;
PROGRAM pPROGRAM(FILE *inp) {
  initialize_lexer(inp);
  if (yyparse()) /* Failure */
    return 0;
  else /* Success */
    return YY_RESULT_PROGRAM_;
}
...
%token _ERROR_   /* Terminal */
%token _SYMB_0    /*   ||   */
%token _SYMB_1    /*   &&   */
%token _SYMB_2    /*   ;    */
%token _SYMB_3    /*   (    */
%token _SYMB_4    /*   )    */
%token _SYMB_5    /* false  */
%token _SYMB_6    /*  true  */
...
```

Bison Parser Continued

```
%%
PROGRAM : ListEXP {
   $$ = make_PROGRAM(reverseListEXP($1));
                YY_RESULT_PROGRAM_= $$; }
;
EXP : EXP _SYMB_0 EXP1 {
                   $$ = make_EOr($1, $3);
                    YY_RESULT_EXP_= $$; }
  | EXP1 { $$ = $1; YY_RESULT_EXP_= $$; }
;
EXP1 : EXP1 _SYMB_1 EXP2 {
                  $$ = make_EAnd($1, $3);
                    YY_RESULT_EXP_= $$; }
  | EXP2 { $$ = $1; YY_RESULT_EXP_= $$; }
;
EXP2 : _SYMB_6 { $$ = make_ETrue();
                    YY_RESULT_EXP_= $$; }
  | _SYMB_5 { $$ = make_EFalse();
                    YY_RESULT_EXP_= $$; }
  | _IDENT_ { $$ = make_EVar($1);
                    YY_RESULT_EXP_= $$; }
  | _SYMB_3 EXP _SYMB_4 { $$ = $2;
                    YY_RESULT_EXP_= $$; }
;
ListEXP : /* empty */ { $$ = 0;
                YY_RESULT_ListEXP_= $$; }
  | ListEXP EXP _SYMB_2 {
               $$ = make_ListEXP($2, $1);
                YY_RESULT_ListEXP_= $$; }
;
```

D. Pretty Printer

```
...
void ppEXP(EXP _p_, int _i_) {
  switch(_p_->kind) {
  case is_EOr:
    if (_i_ > 0) renderC(_L_PAREN);
    ppEXP(_p_->u.eor_.exp_1, 0);
    renderS("||");
    ppEXP(_p_->u.eor_.exp_2, 1);
    if (_i_ > 0) renderC(_R_PAREN);
    break;
  case is_EAnd:
    if (_i_ > 1) renderC(_L_PAREN);
    ppEXP(_p_->u.eand_.exp_1, 1);
    renderS("&&");
    ...
void ppListEXP(ListEXP listexp, int i) {
  while(listexp!= 0) {
    if (listexp->listexp_ == 0) {
      ppEXP(listexp->exp_, 0);
      renderC(';');
      listexp = 0;
    } else {
      ppEXP(listexp->exp_, 0);
      renderC(';');
      listexp = listexp->listexp_;
    }
  }
}
```

15

Figure 4: C source code fragments generated from Figure 1

Translation Summary.

While it is straightforward to generate a parser and a data structure to represent the results of a parse in C, the combination of pointers and unions (seen in Figures 4B and 4D) results in code that can be sometimes hard for the user to work with. We are currently looking into ways to make the generated code more friendly through the use of macros or other methods.

8 Discussion

Productivity Gains.

The source code of the Boolean expression grammar in Section 2.4 is 8 lines. The size of the generated code varies from 425 lines of Haskell/Happy/Alex to 1112 lines of C++/Bison/Flex. The generated code is not superfluously verbose, but similar to what would be written by hand by a programmer following Appel's methodology [1, 2, 3]. This amounts to a gain of coding effort by a factor of 50–100, which is comparable to the effort saved by, for instance, writing an LR parser in Bison instead of directly in C.[3] In addition to decreasing the number of lines, the single-source approach alleviates synchronization problems, both when creating and when maintaining a language.

The BNF Converter as a Teaching Tool.

The BNF Converter has been used as a teaching tool in a fourth-year compiler course at Chalmers University in 2003 and 2004. The goal is, on the one hand, to advocate the use of declarative and portable language definitions, and on the other hand, to leave more time for back-end construction. The generated code follows the format recommended in Appel's text books [1, 2, 3], which makes is coherent to use the tool as a companion to those books. The results are encouraging: the lexer/parser part of the compiler was estimated only to be 25 % of the work at the lowest grade, and 10 % at the highest grade—at which point the student compiler had to include several back ends. This was far from the times when the parser was more than 50 % of a student compiler. About 50 % of the laboration groups use Haskell as implementation language, the rest using Java, C, or C++. In 2004, when the BNF Converter was available for all these languages, 16 groups of the 19 accepted ones used it in their assignment. The main discouraging factor were initial problems with Bison versions: older versions than 1.875 do compile the generated Bison file, but the parser fails with all input.

[3]In the present example, the Flex and Bison code generated by the BNF Converter is 172 lines, from which these tools generate 2600 line of C.

In Autumn 2003, the BNF Converter was also used in a second-year Chalmers course on Programming Languages. It is replacing the previously-used parser combinator libraries in Haskell. The main motivation at this level is to teach the correspondence between parsers and grammars, and to provide a high-level parser tool also for programmers who do not know Haskell.

One concern about using the BNF Converter was that students would not really learn parsing, but just to write grammars. However, students writing their parsers in YACC are equally isolated from the internals of LR parsing as those writing in LBNF. In fact, as learning the formalism takes less time in the case of LBNF, the teacher can allocate more time for explaining how an LR parser works.

Real-World Languages.

Students in a compiler class usually implement toy languages. What about real-world languages? As an experiment, a complete LBNF definition of ANSI C, with [14] as reference, was written.[4] The result was a complete front-end processor for ANSI C, with the exception, mentioned in [14] of type definitions, which have to be treated with a preprocessor. The grammar has 229 LBNF rules and 15 token definitions (to deal with different numeral literals, such as octals and hexadecimals).

The BNF Converter has also been applied in an industrial application producing a compiler for a telecommunications protocol description language. [5]

Another real-world example is the object-oriented specification language OCL [20].[5] Finally, the BNF Converter itself is implemented by using modules generated from an LBNF grammar of the LBNF formalism.

A Case Study in Language Prototyping.

A strong case for BNF Converter is the prototyping of new languages. It is easy to add and remove language features, and to test the updated language immediately. Since standard tools are used, the step from the prototype to a production-quality front end is small, typically involving some fine-tuning of the abstract syntax and the pretty printer. We have a large-scale experience of this in creating a new version of the language GF (Grammatical Framework, [18]).

The main novelties added to GF were a module system added on top of the old GF language, and a lower-level language GFC, playing the role of "object code" generated by the GF compiler. The GF language has constructions mostly familiar from functional programming languages, and

[4]BSc thesis of Ulf Persson at Chalmers.

[5]Work by Kristofer Johannisson at Chalmers (private communication).

the size of the full grammar is similar to ANSI C; GFC is about half this size. We wrote the LBNF grammar from scratch, one motivation being to obtain reliable documentation of GF. This work took a few hours. We then used the skeleton file to translate the generated abstract syntax into the existing hand-written Haskell datatypes; in this way, we did not need to change the later phases of the existing compiler (apart from the changes due to new language features). In a couple of days, we had a new parser accepting all old GF files as well as files with the new language features. Working with later compilation phases suggested some changes in the new features, such as adding and removing type annotations. Putting the changes in place never required changing other things than the LBNF grammar and some clauses in the skeleton-based translator.

The development of GFC was different, since the language was completely new. The crucial feature was the symmetry between the parser and the pretty printer. The GF compiler generates GFC, but it also needs to parse GFC, so that it can use precompiled modules instead of source files. It was reassuring to know that the parser and the pretty printer completely matched. As a last step, we modified the rendering function of the GFC pretty printer so that it did not generate unnecessary spaces; GFC code is not supposed to be read by humans. This step initially created unparsable code (due to some necessary spaces having been omitted), which was another proof of the value of automatically generated pretty-printers.

In addition to the GF compiler written in Haskell, we have been working on GF-based applets ("gramlets") written in Java. These applications use precompiled GF. With the Java parser generated by the BNF Converter, we can guarantee that the GFC code generated by the Haskell pretty-printer can be read in by the Java application.

Related Work.

The BNF Converter adds a level of abstraction to the YACC [13] tradition of compiler compilers, since it compiles a yet higher-level notation into notations on the level of YACC. Another system on this level up from YACC is Cactus [16], which uses an EBNF-like notation to generate front ends in Haskell and C. Unlike the BNF Converter, Cactus aims for completeness, and the notation is therefore more complex than LBNF. It is not possible to extract a pretty printer from a Cactus grammar, and Cactus does not generate documentation.

The Zephyr definition language [19] defines a portable format for abstract syntax and translates it into SML, Haskell, C, C++, Java, and SGML, together with functions for displaying syntax trees. It does not support the definition of concrete syntax.

In general, compiler tools almost invariably opt for expressive power rather than declarativity and simplicity. The situation is different in linguistics, where the declarativity and *reversibility* (i.e. usability for both parsing and generation) of grammar formalisms are highly valued. A major example of this philosophy are Definite Clause Grammars (DCG) [17]. Since DCGs are usually implemented as an embedded language in Prolog, features of full Prolog are sometimes smuggled into DCG grammars; but this is usually considered harmful since it destroys declarativity.

9 Conclusions and Future Work

BNF Converter is a tool implementing the Labelled BNF grammar formalism (LBNF). Given that a programming language is "well-behaved", in a rather intuitive sense, an LBNF grammar is the only source that is needed to implement a front end for the language, together with matching LaTeX documentation. Since LBNF is purely declarative, the implementation can be generated in different languages: these currently include Haskell, Java, C++, and C, each with their standard parser and lexer tools. Depending on the tools, the size of the generated code is typically 50–100 times the size of the LBNF source.

The approach has proven to be useful both in teaching and in language prototyping. As for legacy real-world languages, complete definitions have so far been written for C and OCL. Often a language is almost definable, but has some exotic features that would require stronger tools. We have, however, opted to keep LBNF simple, at the expense of expressivity; and we believe that there are many good reasons behind a trend toward more and more well-behaved programming languages.

One frequent request has been a possibility to retain some of the position information in the abstract syntax tree, so that error messages from later compiler phases can be linked to the source code. This has been partly solved by extending the `token` pragma with the keyword `position` that enable position information to be retained in that particular token. However, further generalizations are needed at this point. Other requests are increased control of the generated abstract syntax and some means of controlling the output of the pretty-printing.

References

[1] A. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[2] A. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[3] A. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[4] E. Berk and C. Ananian. JLex: A Lexical Analyzer Generator for Java, 2000. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[5] C. Däldborg and O. Noreklint. ASN.1 Compiler. 2004. Master's Thesis, Department of Computing Science, Chalmers University of Technology.

[6] C. Dornan. Alex: a Lex for Haskell Programmers, 1997. `http://www.cs.ucc.ie/dornan/alex.html`.

[7] M. Forsberg, P. Gammie, M. Pellauer, and A. Ranta. BNF Converter site. Program and documentation, `http://www.cs.chalmers.se/~markus/BNFC/`, 2004.

[8] M. Forsberg and A. Ranta. Labelled BNF: a highlevel formalism for defining well-behaved programming languages. *Proceedings of the Estonian Academy of Sciences: Physics and Mathematics*, 52:356–377, 2003. Special issue on programming theory edited by J. Vain and T. Uustalu.

[9] Free Software Foundation. Bison - GNU Project, 2003. `http://www.gnu.org/software/bison/bison.html`.

[10] Free Software Foundation. Flex - GNU Project, 2003. `http://www.gnu.org/software/flex/flex.html`.

[11] E. Gamma, R. Hehn, R. Johnson, and J. Viissides. *Design Patterns*. Addison Wesley, 1995.

[12] S. E. Hudson. CUP Parser Generator for Java, 2003. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[13] S. C. Johnson. Yacc — yet another compiler compiler. Technical Report CSTR-32, AT & T Bell Laboratories, Murray Hill, NJ, 1975.

[14] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988. 2nd edition.

[15] S. Marlow. Happy, The Parser Generator for Haskell, 2001. `http://www.haskell.org/happy/`.

[16] N. Martinsson. Cactus (Concrete- to Abstract-syntax Conversion Tool with Userfriendly Syntax) . Master's Thesis in Computer Science, 2001. `http://www.mdstud.chalmers.se/~md6nm/cactus/`.

[17] F. Pereira and D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[18] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[19] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr Abstract Syntax Description Language. 1997. USENIX Association.

[20] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML.* Addison-Wesley, 1999.