Labelled BNF: A High-Level Formalism for Defining Well-Behaved Programming Languages

Markus Forsberg and Aarne Ranta
Department of Computing Science
Chalmers University of Technology and the University of Gothenburg
SE-412 96 Gothenburg, Sweden
{markus,aarne}@cs.chalmers.se

June 6, 2003

Abstract

This paper introduces the grammar formalism *Labelled BNF* (LBNF), and the compiler construction tool *BNF Converter*. Given a grammar written in LBNF, the BNF Converter produces a complete compiler front end (up to, but excluding, type checking), i.e. a lexer, a parser, and an abstract syntax definition. Moreover, it produces a pretty-printer and a language specification in LaTeX, as well as a template file for the compiler back end.

A language specification in LBNF is completely declarative and therefore portable. It reduces dramatically the effort of implementing a language. The price to pay is that the language must be "well-behaved", i.e. that its lexical structure must be describable by a regular expression and its syntax by a context-free grammar.

Keywords

compiler construction, parser generator, grammar, labelled BNF, abstract syntax, pretty printer, document automation

1 Introduction

This paper defends an old idea: a programming language is defined by a BNF grammar [12]. This idea is usually not followed for two reasons. One reason is that a language may require more powerful methods (consider, for example, languages with layout rules). The other reason is that, when parsing, one wants to do other things already (such as type checking etc). Hence the idea of extending pure BNF with semantic actions, written in a general-purpose programming language. However, such actions destroy declarativity and portability. To describe the language, it becomes necessary to write a separate document, since the BNF no longer defines the language. Also the problem of synchronization arises: how to guarantee that the different modules—the lexer, the parser, and the document, etc.—describe the same language and that they fit together?

The idea in LBNF is to use BNF, with construction of syntax trees as the only semantic action. This gives a unique source for all language-related modules, and it also solves the problem of synchronization. Thereby it dramatically reduces the effort of implementing a new language. Generating syntax trees instead of using more complex semantic actions is a natural phase of *multi-phase compilation*, which is recommended by most modern-day text books about compiler construction (e.g. Appel[1]). BNF grammars are an ingredient of all modern compilers. When designing LBNF, we tried to keep it so simple and intuitive that it can be learnt in a few minutes by anyone who knows ordinary BNF.

Of course, the are some drawbacks with our approach. Not all languages can be completely defined, although surprisingly many can (see Section 5.1). Another drawback is that the modules generated are not quite as good as handwritten. But this is a general problem when generating code instead of handwriting it: a problem shared by all compilers, including the standard parser and lexer generation tools.

To use LBNF descriptions as implementations, we have built the *BNF Converter*[4]. Given an input LBNF grammar, the BNF Converter produces a lexer, a parser, and an abstract syntax definition. Moreover, it produces a pretty-printer and a language specification in LaTeX. Since all this is generated from a *single source*, we can be sure that the documentation corresponds to the actual language, and that the lexer, parser and abstract syntax fit seamlessly together.

The BNF Converter is written in the functional programming language Haskell[14], and its target languages are presently Haskell, the associated compiler tools Happy[10] and Alex[3], and LaTeX. Happy is a parser generator tool, similar to YACC[7], which from a BNF-like description builds an LALR(1) parser. Alex is a lexer generator tool, similar to Lex[9], which converts a regular expression into a finite-state automaton. Over the years, Haskell and these tools have proven to be excellent devices for compiler construction, to a large extent because of Haskell's algebraic data types and a convenient method of syntax-directed translation via pattern matching; yet they do not quite remove the need for repetitive and low-level coding. The BNF Converter can be seen as a high-level front end to these tools. However, due to its declarative nature, LBNF does not crucially depend on the target language, and it is therefore possible to redirect the BNF Converter as a front end to another set of compiler tools. This has in fact recently been done for Java, CUP [6], and JLex [2][1]. The only essential difference between Haskell/Happy/Alex and Java/CUP/JLex or C/YACC/Lex is the target language included in the parser and lexer description.

2 The LBNF Grammar Formalism

As the first example of LBNF, consider a triple of rules defining addition expressions with "1":

```
EPlus. Exp ::= Exp "+" Num ;
ENum.  Exp ::= Num ;
NOne.  Num ::= "1" ;
```

Apart from the *labels*, `EPlus`, `ENum`, and `NOne`, the rules are ordinary BNF rules, with terminal symbols enclosed in double quotes and nonterminals written without quotes. The labels serve as *constructors* for syntax trees.

From an LBNF grammar, the BNF Converter extracts an *abstract syntax* and a *concrete syntax*. The abstract syntax is implemented, in Haskell, as a system of datatype definitions

```
data Exp = EPlus Exp Exp | ENum Num
data Num = NOne
```

(For other languages, including C and Java, an equivalent representation can be given in the same way as in the Zephyr abstract syntax specification tool [16]). The concrete syntax is implemented by the lexer, parser and pretty-printer algorithms, which are defined in other generated program modules.

2.1 LBNF in a nutshell

Briefly, an LBNF grammar is a BNF grammar where every rule is given a label. The label is used for constructing a syntax tree whose subtrees are given by the nonterminals of the rule, in the same order.

More formally, an LBNF grammar consists of a collection of rules, which have the following form (expressed by a regular expression; Appendix A gives a complete BNF definition of the notation):

Ident "`.`" Ident "`::=`" (Ident | String)* "`;`" ;

The first identifier is the *rule label*, followed by the *value category*. On the right-hand side of the production arrow (`::=`) is the list of production items. An item is either a quoted string (*terminal*) or a category symbol (*non-terminal*). A rule whose value category is C is also called a *production* for C.

Identifiers, that is, rule names and category symbols, can be chosen *ad libitum*, with the restrictions imposed by the target language. To satisfy Haskell, and C and Java as well, the following rule is imposed

An identifier is a nonempty sequence of letters, starting with a capital letter.

[1] Work by Michael Pellauer at Chalmers

LBNF is clearly sufficient for defining any context-free language. However, the abstract syntax that it generates may often become too detailed. Without destroying the declarative nature and the simplicity of LBNF, we have added to it four *ad hoc* conventions, which are described in the following subsection.

2.2 LBNF conventions

2.2.1 Predefined basic types

The first convention are predefined basic types. Basic types, such as integer and character, can of course be defined in a labelled BNF, for example:

```
Char_a. Char ::= "a" ;
Char_b. Char ::= "b" ;
```

This is, however, cumbersome and inefficient. Instead, we have decided to extend our formalism with predefined basic types, and represent their grammar as a part of lexical structure. These types are the following, as defined by LBNF regular expressions (see 3.3 for the regular expression syntax):

`Integer` of integers, defined `digit+`

`Double` of floating point numbers, defined `digit+ '.' digit+ ('e' '-'? digit+)?`

`Char` of characters (in single quotes), defined `'\'' ((char - ["'\\"]) | ('\\' ["'\\nt"])) '\''`

`String` of strings (in double quotes), defined `'"' ((char - ["\"\\"]) | ('\\' ["\"\\nt"]))* '"'`

`Ident` of identifiers, defined `letter (letter | digit | '_' | '\'')*`

In the abstract syntax, these types are represented as corresponding types. In Haskell, we also need to define a new type for Ident:

```
newtype Ident = Ident String
```

For example, the LBNF rules

```
EVar. Exp ::= Ident ;
EInt. Exp ::= Integer ;
EStr. Exp ::= String ;
```

generate the abstract syntax

```
data Exp = EVar Ident | EInt Integer | EStr String
```

where `Integer` and `String` have their standard Haskell meanings. The lexer only produces the high-precision variants of integers and floats; authors of applications can truncate these numbers later if they want to have low precision instead.

Predefined categories may not have explicit productions in the grammar, since this would violate their predefined meanings.

2.2.2 Semantic dummies

Sometimes the concrete syntax of a language includes rules that make no semantic difference. An example is a BNF rule making the parser accept extra semicolons after statements:

```
Stm ::= Stm ";" ;
```

As this rule is semantically dummy, we do not want to represent it by a constructors in the abstract syntax. Instead, we introduce the following convention:

A rule label can be an underscore _, which does not add anything to the syntax tree.

Thus we can write the following rule in LBNF:

```
_ . Stm ::= Stm ";" ;
```

Underscores are of course only meaningful as replacements of one-argument constructors where the value type is the same as the argument type. Semantic dummies leave no trace in the pretty-printer. Thus, for instance, the pretty-printer "normalizes away" extra semicolons.

2.2.3 Precedence levels

A common idiom in (ordinary) BNF is to use indexed variants of categories to express precedence levels:

```
Exp3 ::= Integer ;
Exp2 ::= Exp2 "*" Exp3 ;
Exp  ::= Exp  "+" Exp2 ;
Exp  ::= Exp2 ;
Exp2 ::= Exp3 ;
Exp3 ::= "(" Exp ")" ;
```

The precedence level regulates the order of parsing, including associativity. Parentheses lift an expression of any level to the highest level.

A straightforward labelling of the above rules creates a grammar that does have the desired recognition behavior, as the abstract syntax is cluttered with type distinctions (between `Exp`, `Exp2`, and `Exp3`) and constructors (from the last three rules) with no semantic content. The BNF Converter solution is to distinguish among category symbols those that are just indexed variants of each other:

> A category symbol can end with an integer index (i.e. a sequence of digits), and is then treated as a type synonym of the corresponding non-indexed symbol.

Thus `Exp2` and `Exp3` are indexed variants of `Exp`.

Transitions between indexed variants are semantically dummy, and we do not want to represent them by constructors in the abstract syntax. To do this, we extend the use of underscores to indexed variants. The example grammar above can now be labelled as follows:

```
EInt.   Exp3 ::= Integer ;
ETimes. Exp2 ::= Exp2 "*" Exp3 ;
EPlus.  Exp  ::= Exp  "+" Exp2 ;
_.      Exp  ::= Exp2 ;
_.      Exp2 ::= Exp3 ;
_.      Exp3 ::= "(" Exp ")" ;
```

Thus the datatype of expressions becomes simply

```
data Exp = EInt Integer | ETimes Exp Exp | EPlus Exp Exp
```

and the syntax tree for `2*(3+1)` is

```
ETimes (EInt 2) (EPlus (EInt 3) (EInt 1))
```

Indexed categories *can* be used for other purposes than precedence, since the only thing we can formally check is the type skeleton (see the section 2.3). The parser does not need to know that the indices mean precedence, but only that indexed variants have values of the same type. The pretty-printer, however, assumes that indexed categories are used for precedence, and may produce strange results if they are used in some other way.

2.2.4 Polymorphic lists

It is easy to define monomorphic list types in LBNF:

```
NilDef.  ListDef ::= ;
ConsDef. ListDef ::= Def ";" ListDef ;
```

However, compiler writers in languages like Haskell may want to use predefined polymorphic lists, because of the language support for these constructs. LBNF permits the use of Haskell's list constructors as labels, and list brackets in category names:

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

As the general rule, we have

`[`C`]`, the category of lists of type C,

`[]` and `(:)`, the Nil and Cons rule labels,

`(:[])`, the rule label for one-element lists.

The third rule label is used to place an at-least-one restriction, but also to permit special treatment of one-element lists in the concrete syntax.

In the LaTeX document (for stylistic reasons) and in the Happy file (for syntactic reasons), the category name `[X]` is replaced by `ListX`. In order for this not to cause clashes, `ListX` may not be at the same time used explicitly in the grammar.

The list category constructor can be iterated: `[[X]]`, `[[[X]]]`, etc behave in the expected way.

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

2.3 The type-correctness of LBNF rules

It is customary in parser generators to delegate the checking of certain errors to the target language. For instance, a Happy source file that Happy processes without complaints can still produce a Haskell file that is rejected by Haskell. In the same way, the BNF converter delegates some checking to Happy and Haskell (for instance, the parser conflict check). However, since it is always the easiest for the programmer to understand error messages related to the source, the BNF Converter performs some checks, which are mostly connected with the sanity of the abstract syntax.

The type checker uses a notion of the *category skeleton* of a rule, which is a pair

$$(C, A \dots B)$$

where C is the unindexed left-hand-side non-terminal and $A \dots B$ is the sequence of unindexed right-hand-side non-terminals of the rule. In other words, the category skeleton of a rule expresses the abstract-syntax type of the semantic action associated to that rule.

We also need the notions of a *regular category* and a *regular rule label*. Briefly, regular labels and categories are the user-defined ones. More formally, a regular category is none of `[`C`]`,`Integer`, `Double`, `Char`, `String` and `Ident`. A regular rule label is none of `_`, `[]`, `(:)`, and `(:[])`.

The type checking rules are now the following:

A rule labelled by `_` must have a category skeleton of form (C, C).

A rule labelled by `[]` must have a category skeleton of form $([C],)$.

A rule labelled by `(:)` must have a category skeleton of form $([C], C[C])$.

A rule labelled by `(:[])` must have a category skeleton of form $([C], C)$.

Only regular categories may have productions with regular rule labels.

Every regular category occurring in the grammar must have at least one production with a regular rule label.

All rules with the same regular rule label must have the same category skeleton.

The second-last rule corresponds to the absence of empty data types in Haskell. The last rule could be strengthened so as to require that all regular rule labels be unique: this is needed to guarantee error-free pretty-printing. Violating this strengthened rule currently generates only a warning, not a type error.

3 LBNF Pragmas

Even well-behaved languages have features that cannot be expressed naturally in its BNF grammar. To take care of them, while preserving the single-source nature of the BNF Converter, we extend the notation with what we call *pragmas*. All these pragmas are completely declarative, and the pragmas are also reflected in the documentation.

3.1 Comment pragmas

The first pragma tells what kinds of *comments* the language has. Normally we do not want comments to appear in the abstract syntax, but treat them in the lexical analysis. The comment pragma instructs the lexer generator (and the document generator!) to treat certain pieces of text as comments and thus to ignore them (except for their contribution to the position information used in parser error messages).

The simplest solution to the comment problem would be to use some default comments that are hard-coded into the system, e.g. Haskell's comments. But this definition can hardly be stated as a condition for a language to be well-behaved, and we could not even define C or Java or ML then. So we have added a comment pragma, whose regular-expression syntax is

"`comment`" String String? ";"

The first string tells how a comment begins. The second, optional, string marks the end of a comment: if it is not given, then the comment expects a newline to end. For instance, to describe the Haskell comment convention, we write the following lines in our LBNF source file:

```
comment "--" ;
comment "{-" "-}" ;
```

Since comments are treated in the lexical analyzer, they must be recognized by a finite state automaton. This excludes the use of nested comments unless defined in the grammar itself. Discarding nested comments is one aspect of what we call well-behaved languages.

The length of comment end markers is restricted to two characters, due to the complexities in the lexer caused by longer end markers.

3.2 Internal pragmas

Sometimes we want to include in the abstract syntax structures that are not part of the concrete syntax, and hence not parsable. They can be, for instance, syntax trees that are produced by a type-annotating type checker. Even though they are not parsable, we may want to pretty-print them, for instance, in the type checker's error messages. To define such an internal constructor, we use a pragma

```
"internal" Rule ";"
```

where `Rule` is a normal LBNF rule. For instance,

```
internal EVarT. Exp ::= "(" Ident ":" Type ")";
```

introduces a type-annotated variant of a variable expression.

3.3 Token pragmas

The predefined lexical types are sufficient in most cases, but sometimes we would like to have more control over the lexer. This is provided by *token pragmas*. They use regular expressions to define new token types.

If we, for example, want to make a finer distinction for identifiers, a distinction between lower- and upper-case letters, we can introduce two new token types, `UIdent` and `LIdent`, as follows.

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

The regular expression syntax of LBNF is specified in the Appendix. The abbreviations with strings in brackets need a word of explanation:

`["abc7%"]` denotes the union of the characters `'a' 'b' 'c' '7' '%'`

`{"abc7%"}` denotes the sequence of the characters `'a' 'b' 'c' '7' '%'`

The atomic expressions `upper`, `lower`, `letter`, and `digit` denote the character classes suggested by their names (letters are isolatin1). The expression `char` matches any character in the 8-bit ASCII range, and the "epsilon" expression `eps` matches the empty string.[2]

[2]If we want to describe full Java, we must extend the character set to Unicode. This is currently not supported by Alex, however.

3.4 Entry point pragmas

The BNF Converter generates, by default, a parser for every category in the grammar. This is unnecessarily rich in most cases, and makes the parser larger than needed. If the size of the parser becomes critical, the *entry points pragma* enables the user to define which of the parsers are actually exported:

```
entrypoints (Ident ",")* Ident ;
```

For instance, the following pragma defines `Stm` and `Exp` to be the only entry points:

```
entrypoints Stm, Exp ;
```

4 BNF Converter code generation

4.1 The files

Given an LBNF source file `Foo.cf`, the BNF Converter generates the following files:

- `AbsFoo.hs`: The abstract syntax (Haskell source file)
- `LexFoo.x`: The lexer (Alex source file)
- `ParFoo.y`: The parser (Happy source file)
- `PrintFoo.hs`: The pretty printer (Haskell source file)
- `SkelFoo.hs`: The case Skeleton (Haskell source file)
- `TestFoo.hs`: A test bench file for the parser and pretty printer (Haskell source file)
- `DocFoo.tex`: The language document (LaTeXsource file)
- `makefile`: A makefile for the lexer, the parser, and the document

In addition to these files, the user needs the Alex runtime file `Alex.hs` and the error monad definition file `ErrM.hs`, both included in the BNF Converter distribution.

4.2 Example: `JavaletteLight.cf`

The following LBNF grammar defines a small C-like language, Javalette Light[3].

```
Fun.      Prog     ::= Typ Ident "(" ")" "{" [Stm] "}" ;
SDecl.    Stm      ::= Typ Ident ";"  ;
SAss.     Stm      ::= Ident "=" Exp ";"  ;
SIncr.    Stm      ::= Ident "++" ";"  ;
SWhile.   Stm      ::= "while" "(" Exp ")" "{" [Stm] "}" ;
ELt.      Exp0     ::= Exp1 "<" Exp1 ;
EPlus.    Exp1     ::= Exp1 "+" Exp2 ;
ETimes.   Exp2     ::= Exp2 "*" Exp3 ;
EVar.     Exp3     ::= Ident ;
EInt.     Exp3     ::= Integer ;
EDouble.  Exp3     ::= Double ;
TInt.     Typ      ::= "int" ;
TDouble.  Typ      ::= "double" ;
[].       [Stm]    ::= ;
(:).      [Stm]    ::= Stm [Stm] ;

-- coercions
_.  Stm      ::= Stm ";" ;
_.  Exp      ::= Exp0 ;
_.  Exp0     ::= Exp1 ;
_.  Exp1     ::= Exp2 ;
```

[3]It is a fragment of the language Javalette used at compiler construction courses at Chalmers University

```
_.  Exp2     ::= Exp3 ;
_.  Exp3     ::= "(" Exp ")" ;

-- pragmas
internal ExpT. Exp ::= Typ Exp ;
comment "/*" "*/" ;
comment "//" ;
entrypoints Prog, Stm, Exp ;
```

4.2.1 The abstract syntax `AbsJavaletteLight.hs`

The abstract syntax of Javalette generated by the BNF Converter is essentially what a Haskell programmer would write by hand:

```
data Prog =
   Fun Typ Ident [Stm]
  deriving (Eq,Show)

data Stm =
   SDecl Typ Ident
 | SAss Ident Exp
 | SIncr Ident
 | SWhile Exp [Stm]
  deriving (Eq,Show)

data Exp =
   ELt Exp Exp
 | EPlus Exp Exp
 | ETimes Exp Exp
 | EVar Ident
 | EInt Integer
 | EDouble Double
 | ExpT Typ Exp
  deriving (Eq,Show)

data Typ =
   TInt
 | TDouble
  deriving (Eq,Show)
```

4.2.2 The lexer `LexJavaletteLight.x`

The lexer file (in Alex) 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) and for comments. Here is a fragment with the definitions characteristic of Javalette.

```
{ %s =  ^( | ^) | ^{ | ^} | ^; | ^= | ^+ ^+ | ^< | ^+ | ^*}

"tokens_lx"/"tokens_acts":-
 <>       ::= ^/^/ [.]* ^n
 <>       ::= ^/ ^* ([^u # ^*] | ^* [^u # ^/])* (^*)+ ^/

 <>       ::= ^w+
 <pTSpec> ::=  %s %{ pTSpec p = PT p . TS    %}
 <ident>  ::= ^l ^i*   %{ ident  p = PT p . eitherResIdent TV %}
 <int>    ::= ^d+      %{ int    p = PT p . TI    %}
 <double> ::= ^d+ ^. ^d+ (e (^-)? ^d+)? %{ double  p = PT p . TD %}

 eitherResIdent :: (String -> Tok) -> String -> Tok
 eitherResIdent tv s = if isResWord s then (TS s) else (tv s) where
   isResWord s = elem s ["double","int","while"]
```

The lexer file moreover defines the token type `Tok` used by the lexer and the parser.

4.2.3 The parser `ParJavaletteLight.y`

The parser file (in Happy) has a large number of token definitions (which we find it extremely valuable to generate automatically), followed by parsing rules corresponding closely to the source BNF rules. Here is a fragment containing examples of both parts:

```
%token
 '('      { PT _ (TS "(") }
 ')'      { PT _ (TS ")") }
 'double' { PT _ (TS "double") }
 'int'    { PT _ (TS "int") }
 'while'  { PT _ (TS "while") }

L_integ  { PT _ (TI $$) }
L_doubl  { PT _ (TD $$) }

%%

Integer : L_integ  { (read $1) :: Integer }
Double  : L_doubl  { (read $1) :: Double }

Stm :: { Stm }
Stm : Typ Ident ';'                      { SDecl $1 $2 }
  | Ident '=' Exp ';'                    { SAss $1 $3 }
  | Ident '++' ';'                       { SIncr $1 }
  | 'while' '(' Exp ')' '{' ListStm '}' { SWhile $3 (reverse $6) }
  | Stm ';'                              { $1 }

Exp0 :: { Exp }
Exp0 : Exp1 '<' Exp1 { ELt $1 $3 }
  | Exp1 { $1 }
```

The exported parsers have types of the following form, for any abstract syntax type `T`,

```
[Tok] -> Err T
```

returning either a value of type `T` or an error message, using a simple error monad. The input is a token list received from the lexer.

4.2.4 The pretty-printer `PrintJavaletteLight.hs`

The pretty-printer consists of a Haskell class `Print` with instances for all generated datatypes, taking precedence into account. The class method `prt` generates a list of strings for a syntax tree of any type.

```
instance Print Exp where
  prt i e = case e of
    ELt    exp0 exp -> prPrec i 0 (concat [prt 1 exp0 , ["<"] , prt 1 exp])
    EPlus  exp0 exp -> prPrec i 1 (concat [prt 1 exp0 , ["+"] , prt 2 exp])
    ETimes exp0 exp -> prPrec i 2 (concat [prt 2 exp0 , ["*"] , prt 3 exp])
```

The list is then put in layout (identation, newlines) by a *rendering* function, which is generated independently of the grammar, but written with easy modification in mind.

4.2.5 The case skeleton `SkelJavaletteLight.hs`

The case skeleton can be used as a basis when defining the compiler back end, e.g. type checker and code generator. The same skeleton is actually also used in the pretty printer. The case branches in the skeleton are initialized to show error messages saying that the case is undefined.

```
transExp :: Exp -> Result
transExp x = case x of
  ELt exp0 exp    -> failure x
  EPlus exp0 exp  -> failure x
  ETimes exp0 exp -> failure x
```

4.2.6 The language document `DocJavaletteLight.tex`

We show the main parts of the generated JavaletteLight document in a typeset form. The grammar symbols in the document are produced by LaTeX macros, with easy modification in mind.

The lexical structure of JavaletteLight

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩ + '.'⟨*digit*⟩ + ('*e*''-'?⟨*digit*⟩+)? i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in JavaletteLight are the following:

double	**int**	**while**

The symbols used in JavaletteLight are the following:

(	)	{
}	;	=
++	$<$	+
*		

Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of JavaletteLight

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Prog*⟩	::=	⟨*Typ*⟩ ⟨*Ident*⟩ **( ) {** ⟨*ListStm*⟩ **}**
⟨*Stm*⟩	::=	⟨*Typ*⟩ ⟨*Ident*⟩ **;**
	|	⟨*Ident*⟩ = ⟨*Exp*⟩ **;**
	|	⟨*Ident*⟩ ++ **;**
	|	**while (** ⟨*Exp*⟩ **) {** ⟨*ListStm*⟩ **}**
	|	⟨*Stm*⟩ **;**
⟨*Exp0*⟩	::=	⟨*Exp1*⟩ $<$ ⟨*Exp1*⟩
	|	⟨*Exp1*⟩
⟨*Exp1*⟩	::=	⟨*Exp1*⟩ + ⟨*Exp2*⟩
	|	⟨*Exp2*⟩
⟨*Exp2*⟩	::=	⟨*Exp2*⟩ * ⟨*Exp3*⟩
	|	⟨*Exp3*⟩
⟨*Exp3*⟩	::=	⟨*Ident*⟩
	|	⟨*Integer*⟩
	|	⟨*Double*⟩
	|	**(** ⟨*Exp*⟩ **)**

$$
\begin{array}{lcl}
\langle \mathit{ListStm} \rangle & ::= & \epsilon \\
 & | & \langle \mathit{Stm} \rangle \ \langle \mathit{ListStm} \rangle
\end{array}
$$

$$
\langle \mathit{Exp} \rangle \ ::= \ \langle \mathit{Exp0} \rangle
$$

$$
\begin{array}{lcl}
\langle \mathit{Typ} \rangle & ::= & \textbf{int} \\
 & | & \textbf{double}
\end{array}
$$

4.2.7 The `makefile`

The makefile is used 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.

4.2.8 The test bench file `TestJavaletteLight.hs`

The test bench file can be loaded in the Haskell interpreter hugs to run the parser and the pretty-printer on terminal or file input. The test functions display a syntax tree (or an error message) and the pretty-printer result from the same tree.

4.3 An optimization: left-recursive lists

The BNF representation of lists is right-recursive, following Haskell's list conctructor. Right-recursive lists, however, are an inefficient way of parsing lists in an LALR parser. The smart programmer would implement a pair of rules such as JavaletteLight's

```
[].    [Stm] ::= ;
(:).   [Stm] ::= Stm [Stm] ;
```

not in the direct way,

```
ListStm : {- empty -} { [] }
  | Stm ListStm { (:) $1 $3 }
```

but under a left-recursive transformation:

```
ListStm : {- empty -} { [] }
  | ListStm Stm { flip (:) $1 $2 }
```

Then the smart programmer would also be careful to reverse the list when it is used:

```
Prog : Typ Ident '(' ')' '{' ListStm '}' { Fun $1 $2 (reverse $6) }
```

As reported in the Happy manual, this transformation is vital to avoid running out of stack space with long lists. Thus we have implemented the transformation in the BNF Converter for pairs of rules of the form

```
[].  [C] ::= ;
(:). [C] ::= C x [C] ;
```

where `C` is any category and `x` is any sequence of terminals (possibly empty).

There is another important parsing technique, recursive descent, which cannot live with left recursion at all, but loops infinitely with left-recursive grammars (cf. e.g. [1]). The question sometimes arises if, when designing a grammar, one should take into account what method will be used for parsing it. The view we are advocating is that the designer of the grammar should in the first place think of the abstract syntax, and let the parser generator perform automatic grammar transformations that are needed by the parsing method.

5 Discussion

5.1 Results

LBNF and the BNF Converter[4] were introduced as a teaching tool at the fourth-year compiler course in Spring 2003 at Chalmers. The goal was, 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 in a compiler course. The students of the course had as a project to build a compiler in small groups, and grading was based on how much (faultless) functionality the compiler had, e.g. how many language features and how many back ends. The first results were encouraging: a majority (12/20) of the groups that finished their compiler used the BNF Converter. They all were able to produce faultless front ends and, in average, more advanced back ends than the participants of the previous year's edition of the course. In fact, 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—far from the old times when the parser was more than 50 % of a student compiler.

One worry about using the LBNF in teaching was that students would not really learn parsing, but just to write grammars. We found that this concern is not relevant when comparing LBNF with a parser tool like Happy and YACC: 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 the LR parser works. The lexer was a bigger concern, though: since all of the token types needed for the project were predefined types in LBNF, the students did not need to write a single regular expression to finish their compiler! An obvious solution to this is to add some more exotic token types to the project specification.

The main conclusion drawn from the teaching experiment was that the tool should be ported to C and Java, so that the students who don't use Haskell would have the same facilities as those who do.

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 [8] as reference, has been written[4]. The length of the LBNF source file is approximately the same as the length of the specification. Here is a word count comparison between the source file and what is generated:

```
$ wc C.cf
  288    1248   10203 C.cf

$ wc ?*C.* makefile
  287     707    5635 AbsC.hs
  518    1795   23062 DocC.tex
   72     501    2600 LexC.x
  477    2675   13761 ParC.y
  423    3270   18114 PrintC.hs
  336    1345    9178 SkelC.hs
   22     103     677 TestC.hs
    7      22     320 makefile
 2142   10418   73347 total
```

Another real-world example is the object-oriented specification language OCL [17][5]. And of course, the BNF Converter has been implemented by using modules generated from an LBNF grammar of LBNF (see the Appendix).

5.2 Well-behaved languages

A language that can be defined in LBNF is one whose syntax is context-free.[6] Its lexical structure can be described by a regular expression. Modern languages, like Java and C, are close to this ideal; Haskell, with its layout syntax and infix declarations, is a little farther. To rescue the maximum of existing Haskell or some other language would be a matter of detail handwork rather than general principles; and we have opted for keeping the LBNF formalism simple, sacrificing completeness.

[4]Work by Ulf Persson at Chalmers

[5]Work by Kristofer Johannisson at Chalmers

[6]Due to the parser tool used by the BNF converter, it moreover has to be LALR(1)-parsable; but this is a limitation not concerning LBNF as such.

We do not need to sacrifice *semantic completeness*, however: languages usually have a well-behaved subset that is enough for expressing everything that is expressible in the language. When designing new languages—and even when using old ones—we find it a virtue to avoid exotic features. Such features are often included in the name of user-friendliness, but for *new* users, they are more often an obstacle than a help, since they violate the users' expectations gained from other languages.

5.3 Related work

The BNF Converter belongs largely to the YACC [7] tradition of compiler compilers, since it compiles a higher-level notation into the YACC-like notation of Happy, and since the parser is the most demanding part of a language front-end implementation. Another system on this level up from YACC is Cactus [11], which uses an EBNF-like notation to generate a Happy parser, an Alex lexer, and a datatype definition for abstract syntax. Cactus, unlike the BNF Converter, aims for completeness, and it is indeed possible to define Haskell 98 (without layout rules) in it [5]. The price to pay is that the notation is less simple than LBNF. Moreover, because of Cactus's higher level of generality, it is no longer possible to extract a pretty-printer from a grammar. Nor does Cactus generate documentation.

For abstract syntax alone, the Zephyr definition language [16] defines a portable format and translations into program code in SML, Haskell, C, C++, Java, and SGML. Zephyr also generates functions for displaying syntax trees in these languages. But it does not support the definition of concrete syntax.

A survey of compiler tools on the web and in the literature tells that their authors almost invariably opt for expressivity rather than declarativity. The situation is different with grammar tools used in linguistic: there the declarativity and *reversibility* (i.e. usability for both parsing and generation) of grammar formalisms is highly valued. A major example of this philosophy are Definite Clause Grammars (DCG) [13]. In practice, DCGs are implemented as an embedded language in Prolog, and thereby some features of full Prolog are sometimes smuggled into grammars to improve expressivity; but this is usually considered harmful since it destroys declarativity and reversibility.

5.4 Future work

In addition to the obvious task of writing LBNF back ends to other languages than Haskell, there are many imaginable ways to extend the formalism itself. One direction is to connect LBNF with the Grammatical Framework GF [15]. GF is a rich grammar formalism originally designed to describe natural languages. LBNF was originally a spin-off of GF, customizing a subset of GF to combine with standard compiler tools. The connection between LBNF and and GF is close, with the difference that GF makes an explicit distinction between abstract and concrete syntax. Consider an LBNF rule describing multiplication:

```
Mult. Exp2 ::= Exp2 "*" Exp3 ;
```

This rule is in GF divided into two judgements: an abstract syntax function definition, and a concrete syntax linearization rule,

```
fun Mult : Exp -> Exp -> Exp ;
lin Mult e1 e2 = {s = parIf P2 e1 ++ "*" ++ parIf P3 e2 ; p = P2} ;
```

Precedence is treated as a parameter that regulates the uses of parentheses. In GF, the user can define new parameter types, and thus the precedences `P2` and `P3`, as well as the function `parIf`, are defined in the source code instead of being built in into the system, as in LBNF. GF moreover includes higher-order abstract syntax and dependent types, and a GF grammar can therefore define the type system of a language.

6 Conclusion

We see Labelled BNF as a natural step to a yet higher level in the development that led machine programmers to create assemblers, assembler programmers to create Fortran and C, and C programmers to create YACC and Lex. A high-level notation always hides details that can be considered well-understood and therefore uninteresting; this lets the users of the new notation to concentrate on new things. At the same time, it creates quality by eliminating certain errors. Inevitably, it also precludes some smart decisions that a human would make if hand-writing the generated code.

It would be too big a claim to say that LBNF can replace tools like YACC and Happy. It can only replace them if the language to be implemented is simple enough. Even though this is not always the case with legacy programming languages, there is a visible trend towards simple and standardized, "well-behaved" languages, and LBNF has proved useful in reducing the effort in implementing such languages.

References

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

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

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

[4] M. Forsberg and A. Ranta. Bnf converter site. Program and documentation, `http://www.cs.chalmers.se/~markus/BNFC/`, 2002.

[5] T. Hallgren. The Haskell 98 grammar in Cactus, 2001. `http://www.cs.chalmers.se/~hallgren/CactusExample/`.

[6] S. E. Hudson. CUP Parser Generator for Java, 1999. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

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

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

[9] M. E. Lesk. Lex — a lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, N.J., 1975.

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

[11] 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/`.

[12] P. Naur. Revised Report of the Algorithmic Language Algol 60. *Comm. ACM*, 6:1–17, 1963.

[13] 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.

[14] S. Peyton Jones and J. Hughes. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://www.haskell.org`, February 1999.

[15] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*. To appear.

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

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

7 Appendix: LBNF Specification

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of LBNF

Identifiers

Identifiers $\langle Ident \rangle$ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ ', reserved words excluded.

Literals

String literals $\langle String \rangle$ have the form "x", where x is any sequence of characters.

Character literals $\langle Char \rangle$ have the form 'c', where c is any single character.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in LBNF are the following:

char	**comment**	**digit**
entrypoints	**eps**	**internal**
letter	**lower**	**token**
upper		

The symbols used in LBNF are the following:

;	.	::=
[	]	_
(	:	)
|	−	*
+	?	{
}	,	

Comments

Single-line comments begin with −−.
Multiple-line comments are enclosed with { − and − } .

The syntactic structure of LBNF

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

$$\langle Grammar \rangle \quad ::= \quad \langle ListDef \rangle$$

$$\begin{array}{lll} \langle ListDef \rangle & ::= & \epsilon \\ & | & \langle Def \rangle \; ; \; \langle ListDef \rangle \end{array}$$

$$\begin{array}{lll} \langle ListItem \rangle & ::= & \epsilon \\ & | & \langle Item \rangle \; \langle ListItem \rangle \end{array}$$

$$\begin{array}{lll}
\langle \mathit{Def} \rangle & ::= & \langle \mathit{Label} \rangle \ \textbf{.} \ \langle \mathit{Cat} \rangle \ \textbf{::=} \ \langle \mathit{ListItem} \rangle \\
& | & \textbf{comment} \ \langle \mathit{String} \rangle \\
& | & \textbf{comment} \ \langle \mathit{String} \rangle \ \langle \mathit{String} \rangle \\
& | & \textbf{internal} \ \langle \mathit{Label} \rangle \ \textbf{.} \ \langle \mathit{Cat} \rangle \ \textbf{::=} \ \langle \mathit{ListItem} \rangle \\
& | & \textbf{token} \ \langle \mathit{Ident} \rangle \ \langle \mathit{Reg} \rangle \\
& | & \textbf{entrypoints} \ \langle \mathit{ListIdent} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Item} \rangle & ::= & \langle \mathit{String} \rangle \\
& | & \langle \mathit{Cat} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Cat} \rangle & ::= & [\ \langle \mathit{Cat} \rangle \ ] \\
& | & \langle \mathit{Ident} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Label} \rangle & ::= & \langle \mathit{Ident} \rangle \\
& | & \textbf{_} \\
& | & [\,] \\
& | & \textbf{(:)} \\
& | & \textbf{(: [ ])}
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Reg2} \rangle & ::= & \langle \mathit{Reg2} \rangle \ \langle \mathit{Reg3} \rangle \\
& | & \langle \mathit{Reg3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Reg1} \rangle & ::= & \langle \mathit{Reg1} \rangle \ | \ \langle \mathit{Reg2} \rangle \\
& | & \langle \mathit{Reg2} \rangle - \langle \mathit{Reg2} \rangle \\
& | & \langle \mathit{Reg2} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Reg3} \rangle & ::= & \langle \mathit{Reg3} \rangle \ \textbf{*} \\
& | & \langle \mathit{Reg3} \rangle \ + \\
& | & \langle \mathit{Reg3} \rangle \ \textbf{?} \\
& | & \textbf{eps} \\
& | & \langle \mathit{Char} \rangle \\
& | & [\ \langle \mathit{String} \rangle \ ] \\
& | & \{ \ \langle \mathit{String} \rangle \ \} \\
& | & \textbf{digit} \\
& | & \textbf{letter} \\
& | & \textbf{upper} \\
& | & \textbf{lower} \\
& | & \textbf{char} \\
& | & (\ \langle \mathit{Reg} \rangle \ )
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Reg} \rangle & ::= & \langle \mathit{Reg1} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{ListIdent} \rangle & ::= & \langle \mathit{Ident} \rangle \\
& | & \langle \mathit{Ident} \rangle \ \textbf{,} \ \langle \mathit{ListIdent} \rangle
\end{array}$$