Demonstration Abstract: BNF Converter

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

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Languages, Design

Keywords

Compiler Construction, Parser Generator, Grammar, BNF, Abstract Syntax, Pretty Printer, Document Automation

Abstract

We will demonstrate BNFC (the BNF Converter) [7, 6], a multi-lingual compiler tool. BNFC takes as its input a grammar written in LBNF (Labelled BNF) notation, and generates a compiler front-end (an abstract syntax, a lexer, and a parser). Furthermore, it generates a case skeleton usable as the starting point of back-end construction, a pretty printer, a test bench, and a LaTeX document usable as language specification.

The program components can be generated in Haskell, Java, C and C++ and their standard parser and lexer tools. BNFC itself was written in Haskell.

The methodology used for the generated front-end is based on Appel's books on compiler construction [3, 1, 2]. BNFC has been used as a teaching tool in compiler construction courses at Chalmers. It has also been applied to research-related programming language development, and in an industrial application producing a compiler for a telecommunications protocol description language [4].

BNFC is freely available under the GPL license at its website and in the testing distribution of Debian Linux.

1. DEMO OVERVIEW

The demo will consists of a brief explanation of the LBNF source format, followed by instructions on how to compile LBNF source format into a front-end in Haskell, Java, C, and C++. The rest of the demonstration will consist of explaining the generated code for Haskell.

2. GOALS AND LIMITS

The central goals of BNFC are

- to minimize the effort needed for compiler front-end construction
- to encourage clean and simple language design
- to make front-end definitions independent of implementation language and thus portable

The LBNF grammar formalism can be learnt in a few minutes by anyone who knows ordinary BNF. The main addition is that each grammar rule has a label, which is used as a constructor of a syntax tree. No semantic actions other than tree construction are allowed. Therefore the formalism is declarative and portable, and a pretty-printer can be derived from the same grammar as the parser. In addition to syntactic rules, LBNF provides a regular expression notation for defining lexical structure, and some pragmatic declarations defining features such as comments.

Since semantic actions are banned, BNFC can only describe languages that are context-free. The lexer must be finite-state and neatly separated from the parser. Even though these requirements are widely propagated in compiler text books, many real-world languages have features that do not quite conform to them. However, practice has shown that such problems can often be overcome by preprocessing. For example, layout syntax can be handled in BNFC by adding a processing level between the lexer and the parser.

3. AN EXAMPLE GRAMMAR

We will now give a short example to give a taste of what the language implementer has to supply, and what BNFC generates. The example grammar is a subset of the Prolog, known as *pure Prolog*.

```
Db       . Database ::= [Clause] ;
Fact     . Clause ::= Predicate ;
Rule     . Clause ::= Predicate ":-" [Predicate] ;
APred    . Predicate ::= Atom ;
CPred    . Predicate ::= Atom "(" [Term] ")" ;
TAtom    . Term ::= Atom ;
VarT     . Term ::= Var ;
Complex  . Term ::= Atom "(" [Term] ")" ;

terminator Clause "." ;
separator nonempty Predicate "," ;
separator nonempty Term "," ;

token Var  ((upper | '_') (letter | digit | '_')*) ;
token Atom (lower (letter | digit | '_')*) ;

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


Haskell'04, September 22, 2004, Snowbird, Utah, USA.
ACM 1-58113-850-4/04/0009.

The grammar shows a couple of things that go beyond the basic idea of labelled BNF rules and regular expressions: a special syntax `[C]` for polymorphic lists, to avoid cluttering the AST:s with monomorphic lists, as well as shorthands for defining the concrete syntax of a list in terms of its terminator or separator. In addition, LBNF has a notion of precedence levels expressed by integer indices attached to nonterminals. A complete reference of the LBNF language can be found on the BNFC website [6].

4. COMPILING A GRAMMAR

Assuming that the grammar of the previous section is contained in a file named `Prolog.cf`, a Haskell front-end is compiled by issuing the following command:

```
bnfc -m -haskell Prolog.cf
```

This command generates the following file:

- `AbsProlog.hs`: Algebraic datatypes for the AST:s
- `LexProlog.x`: Alex [5] lexer (v1.1 and v2.0)
- `ParProlog.y`: Happy [8] parser
- `PrintProlog.hs`: pretty printer
- `SkelProlog.hs`: AST traversal skeleton
- `TestProlog.hs`: test bench (a program that parses a file and displays the AST and the pretty-printed program)
- `Makefile`: an easy way to compile the test bench
- `DocProlog.tex`: language documentation in LaTeX

For C and C++, similar files are generated but with slightly different names. For Java, many more files are generated, because the abstract syntax definition consists of separate classes for each nonterminal and constructor, following the methodology of Appel [2].

Depending on target language, the generated code is 10–100 times the size of the LBNF source. Yet it isn't hopelessly ugly or low-level, but looks rather similar to hand-written code that follows the chosen compiler writing discipline.

5. RELATED WORK

Cactus [9], uses an EBNF-like notation to generate front ends in Haskell and C. Cactus is more powerful than BNFC, which makes its notation more complex. Cactus does not generate pretty printers and language documents.

The Zephyr language [10] is portable format for abstract syntax translatable into SML, Haskell, C, C++, Java, and SGML, together with functions for displaying syntax trees. Zephyr does not support the definition of concrete syntax.

6. WHEN TO USE BNFC

BNFC has proved useful as a compiler teaching tool. It encourages clean language design and declarative definitions. But it also lets the teacher spend more time on back-end construction and/or the theory of parsing than traditional compiler tools, which require learning tricky and complicated notations.

BNFC also scales up to full-fledged language definitions. Even though real-world languages already have compilers generating machine code, it can be difficult to extract abstract syntax from them. A BNFC-generated parser, case skeleton, and pretty printer is a good starting point for programs doing some new kind of transformation or translation on an existing language.

However, the clearest case for BNFC is the development of new languages. It is easy to get started: just write a few lines of LBNF, run `bnfc`, and apply the `Makefile` to create a test bench. Adding or changing a language construct is also easy, since changes only need to be done in one file. When the language design is complete, the implementor perhaps wants to change the implementation language; no work is lost, since the front-end can be generated in a new target language, Finally, when the language implementation is ready to be given to users, a reliable and human-readable language definition is ready as well.

7. BIO SECTION

Markus Forsberg is an PhD student at the Swedish Graduate School of Language Technology (GSLT) positioned at the department of Computing Science at Chalmers University of Technology and the University of Gothenburg. Aarne Ranta is an associative professor at the same department.

Forsberg and Ranta started the development of the BNF Converter in 2002, as a tool generating Haskell. It was retargeted to C, C++, an Java in 2003 by Michael Pellauer (at Chalmers). Later contributors are Björn Bringert, Peter Gammie and Antti-Juhani Kaijanaho.

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

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

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

[7] 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.

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

[9] 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/`.

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