COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 3: BNFC

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

January 27, 2022

Review

Last time we looked at BNF grammars and parsing, and took a pretty long look at precedence and associativity.

Today, we see how all those things come together to help us generate parsers using BNFC. We'll also see how we can hook into the BNFC generated code and use it's parsing facilities.

BNFC

BNFC stands for Backus-Naur Form Converter. It is called this because it converts BNF grammars into parsers. It was developed as a tool to help with prototyping programming language grammars and parsers.

The usefulness of BNFC comes from the fact that it abstracts the nasty bits of parser generators away so we can focus on developing grammars.

Interface

BNFC is just an executable command-line application. It's interface is very simple. It just takes in a grammar file, and outputs the code for an executable parser.

Throughout this class, you will rarely need to invoke BNFC yourself, as we have makefiles in the assignments that will do it for you. Today, we'll be executing it directly, and you'll see for yourself how simple it is.

Simple Grammar: LBNF

The grammar files that BNFC consumes are written in a language called LBNF, which stands for Labelled Backus Naur Form. It is exactly what it sounds like.

To understand LBNF, let's first make a grammar for simple addition/multiplication expressions using the BNF that we covered last time:

$$\begin{aligned} \langle exp \rangle \quad ::= \quad & \text{Integer} \\ | \quad & \langle exp \rangle \text{ `}*\text{'} \langle exp \rangle \\ | \quad & \langle exp \rangle \text{ `}+\text{'} \langle exp \rangle \end{aligned}$$

Simple Grammar: LBNF

Here's that grammar in LBNF.

```
EInt. Exp ::= Integer ;
EMul. Exp ::= Exp "*" Exp ;
EAdd. Exp ::= Exp "+" Exp ;
```

Simple Grammar: LBNF

Side by side:

```
EInt. Exp ::= Integer ;
EMul. Exp ::= Exp "*" Exp ;
EAdd. Exp ::= Exp "+" Exp ;
```

$$
\begin{array}{lll}
\langle exp \rangle & ::= & \text{Integer} \\
& | & \langle exp \rangle \text{ `}*\text{'} \langle exp \rangle \\
& | & \langle exp \rangle \text{ `}+\text{'} \langle exp \rangle
\end{array}
$$

Simple Grammar: LBNF

There's four main differences between BNF and LBNF:

- LBNF requires every rule to have a single expansion on the right hand side.
- Every rule has a unique label.
- Non-terminal literals are enclosed in double quotes.
- Rules are terminated by semi-colons.

Simple Grammar: LBNF

Another important feature of BNFC is it includes several handy, pre-defined basic types for terminals in a grammar. Here are those, straight from the LBNF reference guide.

- Type `Integer` of integers, defined `digit+`
- Type `Double` of floating point numbers, defined `digit+ '.' digit+ ('e' '-'? digit+)?`
- Type `Char` of characters (in single quotes), defined
  `'\'' ((char - ["'\\"]) | ('\\' ["'\\tnrf"])) '\''`
- Type `String` of strings (in double quotes), defined
  `'"' ((char - ["\"\\"]) | ('\\' ["\"\\tnrf"]))* '"'`
- Type `Ident` of (Haskell) identifiers, defined `letter (letter | digit | '_' | '\'')*`

Simple Grammar: LBNF

This is what enables the rule labelled `EInt` in our grammar, which just says any Integer is a valid expression:

```
EInt. Exp ::= Integer ;
EMul. Exp ::= Exp "*" Exp ;
EAdd. Exp ::= Exp "+" Exp ;
```

Without further ado, let's feed this grammar to BNFC.

BNFC Output

```
> bnfc Expr.cf
3 rules accepted

Use Alex 3 to compile LexExpr.x.
writing new file ./AbsExpr.hs
writing new file ./PrintExpr.hs
writing new file ./LexExpr.x
writing new file ./ParExpr.y
writing new file ./TestExpr.hs
writing new file ./ErrM.hs
writing new file ./SkelExpr.hs
writing new file ./DocExpr.txt
```

We just give BNFC our grammar file, and it generates this list of files. Let's briefly discuss what each of these files is and which you need to care about.

BNFC Output

The most important files:

- `AbsExpr.hs`: Contains the algebraic data type that represents the language's abstract syntax tree. One of the most important files.
- `PrintExpr.hs`: Contains pretty-printing functions for all of the elements of your language. Handy for printing error messages, debug statements and such.
- `TestExpr.hs`: An executable program that uses the generated parser and lexer to parse inputs and report errors. the language.

BNFC Output

The less important files:

- `LexExpr.x`: The lexer definition file that can be fed to `alex` to generate a lexer for your language.
- `ParExpr.y`: The parser definition file that can be fed to `happy` to generate a parser for your language.
- `ErrM.hs`: Contains the definition of the error monad that BNFC uses for parse errors.
- `SkelExpr.hs`: Example "skeleton" program that pattern matches on the AST.
- `DocExpr.txt`: Automatically generated documentation for the language.

BNFC Output

The less important files:

- `LexExpr.x`: The lexer definition file that can be fed to `alex` to generate a lexer for your language.
- `ParExpr.y`: The parser definition file that can be fed to `happy` to generate a parser for your language.
- `ErrM.hs`: Contains the definition of the error monad that BNFC uses for parse errors.
- `SkelExpr.hs`: Example "skeleton" program that pattern matches on the AST.
- `DocExpr.txt`: Automatically generated documentation for the language.

BNFC Output

To use Java, add the `--java` flag to the BNFC command:

```
> bnfc --java Expr.cf
3 rules accepted

   (Tested with JLex 1.2.6)
   (Parser created only for category Exp)
   (Tested with CUP 0.11b)
writing new file ./expr/Absyn/Exp.java
writing new file ./expr/Absyn/EInt.java
writing new file ./expr/Absyn/EAdd.java
writing new file ./expr/Absyn/EMul.java
...
```

Similar files with similar levels of importance are generated.

BNFC Output

We would like BNFC to generate makefiles for us so we can easily get executable parsers. To do that, add the `-m` flag.

Using the Parsers

If we generate makefiles from BNFC, we can get executable parsers by just typing `make` after running BNFC. The test application that gets generated accepts input on standard input, so we can easily test things in our grammar:

```
> echo "1 * 2 + 2" | ./TestExpr

Parse Successful!

[Abstract Syntax]

EMul (EInt 1) (EAdd (EInt 2) (EInt 2))

[Linearized tree]

1 * 2 + 2
```

Ambiguity

Our grammar is ambiguous! Remember how we fix that? Precedence levels! Let's update our grammar to encode the precedence rules as follows: literals $>$ multiplication $>$ addition. We would do this in normal BNF as follows:

$$\begin{aligned} \langle Exp0 \rangle ::= & \langle Exp0 \rangle \text{ `+' } \langle Exp0 \rangle \\ | & \langle Exp1 \rangle \end{aligned}$$

$$\begin{aligned} \langle Exp1 \rangle ::= & \langle Exp1 \rangle \text{ `}*\text{' } \langle Exp1 \rangle \\ | & \langle Exp2 \rangle \end{aligned}$$

$$\begin{aligned} \langle Exp2 \rangle ::= & \text{ Integer} \\ | & \text{ `(' } \langle Exp0 \rangle \text{ `)'} \end{aligned}$$

Ambiguity

We call rules like the following *coercions*:

$$\langle Exp0 \rangle ::= \langle Exp1 \rangle$$

$$\langle Exp1 \rangle ::= \langle Exp2 \rangle$$

$$\langle Exp2 \rangle ::= \text{`(' } \langle Exp0 \rangle \text{ `)'}$$

Because they effectively *coerce* non-terminals into other non-terminals. We use them to enforce precedence rules. Semantically, they have no meaning and we don't care to capture these coercions in the abstract syntax trees. LBNF has nice built-in facilities for precedence levels and coercions.

Ambiguity

Here is how we might enforce the precedence levels in LBNF:

```
EAdd. Exp0 ::= Exp0 "+" Exp0 ;
_.    Exp0 ::= Exp1 ;

EMul. Exp1 ::= Exp1 "*" Exp1 ;
_.    Exp1 ::= Exp2 ;

EInt. Exp2 ::= Integer ;
_.    Exp2 ::= "(" Exp0 ")" ;
```

The underscores mean we consider those rules to be semantic no-ops and we don't consider them parts of the abstract syntax. By convention, the numbers at the end of non-terminals should increase with precedence level.

Ambiguity

The above pattern is so common that BNFC has introduced a nice syntactic sugar for coercion rules. We can rewrite the above equivalently as:

```
EAdd. Exp0 ::= Exp0 "+" Exp0 ;
EMul. Exp1 ::= Exp1 "*" Exp1 ;
EInt. Exp2 ::= Integer ;

coercions Exp 2;
```

The `coercions Exp 2 ;` will create coercion rules for the `Exp` rules up to precedence level 2.

Ambiguity

The last ambiguous part of our grammar is the lack of associativity! The parser generators try to tell us this (as best they can):

```
> happy ParExp.y
shift/reduce conflicts:  2
```

In this case, we know that those are coming the lack of associativity in our operators, but how could we tell that if we didn't know that? This is where using `happy` comes in handy.

Ambiguity

If using `happy`, it will generator a `.info` file that explains your parser as a state machine. Let's look at what it says for our ambiguous language:

```
...
state 16 contains 1 shift/reduce conflicts.
state 17 contains 1 shift/reduce conflicts.
...
```

Ambiguity

Here's state 16 (17 is similar):

```
State 16

    Exp0 -> Exp0 . '+' Exp0                    (rule 4)
    Exp0 -> Exp0 '+' Exp0 .                    (rule 4)

    ')'             reduce using rule 4
    '+'             shift, and enter state 13
                (reduce using rule 4)

    %eof            reduce using rule 4
```

The shift/reduce error is shown after the `'+'`. Meaning the parser does not know whether it should *shift* (push) the or *reduce* (pop) if it encounters a `+`.

Ambiguity

By default, most parser generators will choose to shift rather than reduce. This is good, because that is usually what we want them to do.

If the info file isn't enough, you can also generate a *debugging* parser using `happy`. This requires you to generate the parser yourself with the command `happy -da ParExpr.y` and then remake the test parser with `make`. The degbugging parser will output a nice trace of its actions during parsing:

```
> echo "1 + 1 + 1" | ./TestExpr
state: 0,   token: 5,   action: shift, enter state 4
state: 4,   token: 4,   action: reduce (rule 3), goto state 5
state: 5,   token: 4,   action: reduce (rule 8), goto state 9
state: 9,   token: 4,   action: reduce (rule 7), goto state 11
...
```

Ambiguity

In this case, we know from experience that we can just fix this by making our addition and multiplication operators left-associative as follows:

```
EAdd. Exp0 ::= Exp0 "+" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EInt. Exp2 ::= Integer ;

coercions Exp 2;
```

Ambiguity

Now if we recompile the parser, we don't have any shift/reduce errors!

```
> make
alex --ghc LexExpr.x
happy --array --info --ghc --coerce ParExpr.y
Grammar info written to: ParExpr.info
ghc  TestExpr
[1 of 6] Compiling AbsExpr          ( AbsExpr.hs, AbsExpr.o )
[2 of 6] Compiling LexExpr          ( LexExpr.hs, LexExpr.o )
[3 of 6] Compiling ParExpr          ( ParExpr.hs, ParExpr.o )
[4 of 6] Compiling PrintExpr        ( PrintExpr.hs, PrintExpr.o )
[5 of 6] Compiling SkelExpr         ( SkelExpr.hs, SkelExpr.o )
[6 of 6] Compiling Main             ( TestExpr.hs, TestExpr.o )
Linking TestExpr ...
```

Live Coding AST Evaluation

The rest of this lecture will demonstrate how we can extend the generated code to do interesting things with an abstract syntax tree generated by the parsers. Since most projects in this course will come with boilerplate that does the lexing/parsing for you, the key piece here will be how exactly we do the evaluation in each language.

Useful Resources

- The LBNF Reference
- BNFC Docs

The End