A Translational BNF Grammar Notation (revision 0.4)

By Paul B Mann, first published in 2006, revised in 2011.

Abstract

BNF grammar notation came into existence about 1960 for the specification of programming languages. It was first used for the automatic generation of parsers about 1972. BNF was later replaced with EBNF offering regular expression notation in the right side of grammar rules. EBNF is powerful, however, it describes only the recognition phase, which is only 1/3 of the process of language translation. The second phase is the construction of an abstract-syntax tree and the third phase is the creation of an instruction code sequence. Some parser generators automate the construction of an AST, but none, that I know of, automate the output of instruction codes. Certainly if these second and third phases are to be automated, a suitable notation is required in the grammar. This paper proposes a notation that permits the construction of the AST in the correct order and the creation of instruction codes. In effect, the complete translation process can be described in the grammar and correct translators generated automatically. A working system has been implemented and tested with good results. The generator is called LRSTAR and the new grammar notation is called TBNF.

Keywords

Parser, grammar, grammarware, BNF, EBNF, syntax, LALR, LR, LL, language recognition, AST, intermediate code, compiler compiler, parser generator.

Introduction

The first requirement for TBNF is notation for constructing an abstract-syntax tree (AST) and making sure the AST is built in the correct order so that traversal of the AST produces the output of instruction codes in the desired sequence. I use a simple expression grammar as an example. Consider the simple expression grammar:

```
{ '+'  '-' } <<
{ '*'  '/' } <<

Goal     -> Stmt... <eof>

Stmt     -> Target '=' Exp ';'

Target   -> <identifier>

Exp      -> Primary
         -> Exp '+' Exp
         -> Exp '-' Exp
         -> Exp '*' Exp
         -> Exp '/' Exp

Primary  -> <identifier>
         -> <integer>
         -> '(' Exp ')'
```

AST Construction Notation

This grammar describes a syntax but says nothing about the creation of an AST. We attach to the end of certain rules a notation as follows:

```
+> node_name
```

Which means "when this rule is recognized, make a node in the AST with this name". And if the LR parser attaches the nodes to the AST in a bottom-up order, we will get a complete tree with a root node, intermediate nodes and leaf nodes, just like we want. Here is the improved notation:

```
{ '+'  '-' } <<
{ '*'  '/' } <<

Goal     -> Stmt... <eof>

Stmt     -> Target '=' Exp ';'    +> store

Target   -> <identifier>          +> target(1)

Exp      -> Primary
         -> Exp '+' Exp           +> add
         -> Exp '-' Exp           +> sub
         -> Exp '*' Exp           +> mul
         -> Exp '/' Exp           +> div

Primary  -> <identifier>          +> ident(1)
         -> <integer>             +> int(1)
         -> '(' Exp ')'
```

Note: the (1) notation means, "Attach the value found in the parse stack at relative position one to the new node being added to the AST." We need to have these values in the AST.

So far this is good, however, this notation is lacking one thing. The resultant AST will reflect the order of the input source code, which may not be the order we want when traversing the AST. In this case the AST order created from the rule:

```
Stmt     -> Target '=' Exp ';'    +> store
```

Will be the equivalent of:

```
+ store
  + Target(1)
  + Exp
```

The problem with this is that the tree traversal process would normally traverse the Target first and the Exp second. This is the opposite of what we want when generating intermediate code. This is the order we really want in the AST:

```
+ store
  + Exp
  + Target(1)
```

This allows the traversal to generate code for the Exp before generating the code for storing the result in the Target. Another symbol is needed here to indicate, "Reverse the order of nodes for this rule, when adding these nodes to the AST." We use the symbol (~>) instead of the (->) for this as follows:

```
Stmt     ~> Target '=' Exp ';'    +> store
```

Now we have a notation which can automate the second phase of a language translator, the construction of the AST in the correct order. When given this input source line:

```
x = (1+2/a) * (3-4/b);
```

Our generated parser will build an AST that looks like this:

```
+ root
  + store
    + mul
    | + add
    | | + int (1)
    | | + div
    | |   + int (2)
    | |   + ident (a)
    | + sub
    |   + int (3)
    |   + div
    |     + int (4)
    |     + ident (b)
    + target (x)
```

Exactly what we want.

Creating Instruction Codes From the AST

Now comes the third phase, creating output codes from the AST. We need a notation in the grammar attached to each node name which indicates the instruction we want to emit and the values for the computations. Consider that each node in the AST will be traversed at least two times, once when going down the tree and once when going back up the tree. Some nodes will experience an intermediate "pass over" traversal. These are the parent nodes when a parent has more than one child. For example, consider an argument list: (a, b, c), and its AST:

```
+ arg_list
  + arg (a)
  + arg (b)
  + arg (c)
```

If we think of a “pass over” traversal that passes over the parent node (arg_list) when going from one child to the next, then we have two passes over the “arg_list” node, one when going from (a) to (b) and another when going from (b) to (c). If we have a notation to generate a “(“ at the top-down pass, a “,” at the “pass over” pass, and a “)” at the bottom-up pass, when we traverse the AST we will get this output:

```
(a,b,c)
```

We could represent this in the grammar notation with the following:

```
ArgList -> `(` Arg/`,`... `)`    *> arg_list     emit ( "(", ",", ")" )
Arg     -> <identifier>          *> arg (1)      emit ( "%s" )
```

During top-down, emit “(“. During pass over, emit “,”. During bottom-up, emit “)”. During the bottom-up pass, emit as a string the value attached to the node (a, b, or c). Take a look at the complete TBNF grammar for the simple expression grammar:

```
{ '+'  '-' } <<
{ '*'  '/' } <<

Goal     -> Stmt... <eof>

Stmt     ~> Target '=' Exp ';'    *> store        emit (,,"STORE\n")

Target   -> <identifier>          *> target(1)    emit (,,"LADR %s\n")

Exp      -> Primary
         -> Exp '+' Exp           *> add          emit (,,"ADD\n")
         -> Exp '-' Exp           *> sub          emit (,,"SUB\n")
         -> Exp '*' Exp           *> mul          emit (,,"MUL\n")
         -> Exp '/' Exp           *> div          emit (,,"DIV\n")

Primary  -> <identifier>          *> ident(1)     emit (,,"LOAD %s\n")
         -> <integer>             *> int(1)       emit (,,"LOAD %s\n")
         -> '(' Exp ')'
```

When we process the input statement below:

```
x = (1+2/a) * (3-4/b);
```

We get the same AST as shown above:

```
+ root
 + store
   + mul
   | + add
   | | + int (1)
   | | + div
   | |   + int (2)
   | |   + ident (a)
   | + sub
   |   + int (3)
   |   + div
   |     + int (4)
   |     + ident (b)
   + target (x)
```

And this intermediate code:

```
LOAD 1
LOAD 2
LOAD a
DIV
ADD
LOAD 3
LOAD 4
LOAD b
DIV
SUB
MUL
LADR x
STORE
```

This is exactly what we want. This output is suitable for a stack based interpreter, which could execute these instructions directly or generate object code.

The Set of TBNF Symbols Used in This Paper

These are the symbols required by TBNF:

`->` Production arrow. An arrow indicates a new rule starts here (on the right side of the arrow).

`~>` Reverse production arrow. An arrow indicating a new rule whose nodes will be placed in the AST in reverse order.

`=>` Call a parse action. This means call a parse-action function at parse time when the input matches this rule in the grammar.

`+>` Make a node. This means make a node in the AST.

`*>` Make a node and call a node-action function during AST traversal of this node.

`(1)` Parse stack position. (1) refers to the symbol in parse stack position 1, the first symbol in the right side of the rule. (n) would refer to the n^{th} position in the rule.

`(a,b,c)` Arguments. Arguments are used for Parse Actions and Node Actions.

`&0` Counter indicator. When the AST processor enters a node, it increments a counter for the node and puts in on a stack. The '&0' indicates the current count for the node taken from the stack.. A '&1' means the counter for the parent node on the stack and '&2' means the counter of the grandparent. This provides a unique number for labels.

A More Complete TBNF Grammar

```
/* Input Tokens. */

<error>     => error()  // call error handler.
<ident>     => lookup() // symbol-table lookup.
<integer>   => lookup() // symbol-table lookup.

/* Operator precedence. */

{ '==' '!=' }  <<
{ '+'  '-'  }  <<
{ '*'  '/'  }  <<

/* Rules. */

Goal    -> Pgm... <eof>

Pgm     -> program <ident> '{' Stmt... '}'    *> program(2)    emit ("PROGRAM %s\n",,"END\n")

Stmt    ~> Target '=' Exp ';'                 *> store         emit (,,"STORE\n")
        -> if RelExp Then endif               *> if            emit ("if&0:\n",,"endif&0:\n")
        -> if RelExp Else endif               *> if            emit ("if&0:\n",,"endif&0:\n")
        -> if RelExp Then2 Else2 endif        *> if            emit ("if&0:\n",,"endif&0:\n")

Then    -> then Stmt...                       *> then          emit ("BR NZ,endif&1\nthen&1:\n",,)
Else    -> else Stmt...                       *> else          emit ("BR Z,endif&1 \nelse&1:\n",,)
Then2   -> then Stmt...                       *> then          emit ("BR NZ,else&1 \nthen&1:\n",,)
Else2   -> else Stmt...                       *> else          emit ("BR endif&1   \nelse&1:\n",,)

Target  -> <ident>                            *> target(1)     emit (,,"LADR %s\n")

Exp     -> Primary
        -> Exp '+' Exp                        *> add           emit (,,"ADD\n")
        -> Exp '-' Exp                        *> sub           emit (,,"SUB\n")
        -> Exp '*' Exp                        *> mul           emit (,,"MUL\n")
        -> Exp '/' Exp                        *> div           emit (,,"DIV\n")

RelExp  -> Exp '==' Exp                       *> eq            emit (,,"EQ\n")
        -> Exp '!=' Exp                       *> ne            emit (,,"NE\n")

Primary -> <ident>                            *> ident(1)      emit (,,"LOAD %s\n")
        -> <integer>                          *> int(1)        emit (,,"LOAD %s\n")
        -> '(' Exp ')'
```

Sample Source Code Input

```
program test
{
     if a == 0
     then
        if x == 0
        then b = 10;
        else b = 20;
        endif
     else
        if x == 1
        then b = 30;
        else b = 40;
        endif
     endif
}
```

AST Constructed by the Parser

```
+ root
  + program (test)
    + if
      + eq
      | + ident (a)
      | + int (0)
      + then
      | + if
      |   + eq
      |   | + ident (x)
      |   | + int (0)
      |   + then
      |   | + store
      |   |   + int (10)
      |   |   + target (b)
      |   + else
      |     + store
      |       + int (20)
      |       + target (b)
      + else
        + if
          + eq
          | + ident (x)
          | + int (1)
          + then
          | + store
          |   + int (30)
          |   + target (b)
          + else
            + store
              + int (40)
              + target (b)
```

Intermediate Code Output from the AST Traversal

```
          PROGRAM test
if1:
          LOAD a
          LOAD 0
          EQ
          BR NZ,else1
then1:
if2:
          LOAD x
          LOAD 0
          EQ
          BR NZ,else2
then2:
          LOAD 10
          LADR b
          STORE
          BR endif2
else2:
          LOAD 20
          LADR b
          STORE
endif2:
          BR endif1
else1:
if3:
          LOAD x
          LOAD 1
          EQ
          BR NZ,else3
then3:
          LOAD 30
          LADR b
          STORE
          BR endif3
else3:
          LOAD 40
          LADR b
          STORE
endif3:
endif1:
          END
```

Bibliography

[1] Klint, P., Lammel, R., Verhoef, C.
"Toward an Engineering Discipline for Grammarware."
ACM Transactions on Software Engineering and Methodology,
Vol. 14, No. 3, July 2005, Pages 331-380.

[2] Ford, B.
"Parsing Expression Grammars: A Recognition-Based Syntactic Foundation."
POPL'04, January 14-16, 2004, Venice, Italy,

[3] Derk, M. D.
"Toward a Simpler Method of Operational Semantics for Language Definition."
ACM SIGPLAN Notices, Vol. 40, Issue 5 (May 2005), pages 39-44.

[4] Ledgard, H. F.
"A Human Engineered Variant of BNF."
ACM SIGPLAN Notices, Vol. 15, Issue 10 (Oct 1980), pages 57-62.

Author

Paul B Mann
HighperWare.com