Backus-Naur Form of the C Grammar (Recursive-Descent Parsing)

The C grammar in K&R 2nd Ed is fairly simple, only about 5 pages. Here it is, translated to BNF. Here ( ) groups, ? means optional, | is alternation, + means one or more, * means zero or more, space means sequence, and "x" means literal x. As a special abbreviation, x% means x ("," x)* -- that is, a non-null comma-separated list of x's.

I did this with the idea of writing a bare-bones recursive-descent parser for the language. Accordingly, I have eschewed left recursion, and in general have eschewed recursion as a method of iteration, preferring explicit iteration. I think the only recursion remaining is where recursion is really necessary. This resulted in the elimination of many nonterminals.

I don't know if I will actually carry the implementation as code through, though.

```
Discarded nonterminals: external-declaration struct-or-union
  struct-declaration-list specifier-qualifier-list struct-declarator-list
  enumerator-list init-declarator-list direct-declarator type-qualifier-list
  parameter-list identifier-list initializer-list direct-abstract-declarator
  labeled-statement expression-statement declaration-list statement-list
  primary-expression typedef-name selection-statement
  iteration-statement jump-statement argument-expression-list
  unary-operator asssignment-operator
Renamed symbols: compound-statement -> block
```

40 nonterminals; I discarded 25. Also, I turned typedef-name into a terminal.

Original grammar has a total of 65 nonterminals.

C grammar begins here:

```
Terminals:
  typedef-name integer-constant character-constant floating-constant
  enumeration-constant identifier

translation-unit: (function-definition | declaration)+

function-definition:
  declaration-specifiers? declarator declaration* block

declaration: declaration-specifiers init-declarator% ";"

declaration-specifiers:
  (storage-class-specifier | type-specifier | type-qualifier)+

storage-class-specifier:
  ("auto" | "register" | "static" | "extern" | "typedef")

type-specifier: ("void" | "char" | "short" | "int" | "long" | "float" |
  "double" | "signed" | "unsigned" | struct-or-union-specifier |
  enum-specifier | typedef-name)
```

```
type-qualifier: ("const" | "volatile")

struct-or-union-specifier:
  ("struct" | "union") (
    identifier? "{" struct-declaration+ "}" |
    identifier
  )

init-declarator: declarator ("=" initializer)?

struct-declaration:
  (type-specifier | type-qualifier)+ struct-declarator%

struct-declarator: declarator | declarator? ":" constant-expression

enum-specifier: "enum" (identifier | identifier? "{" enumerator% "}")

enumerator: identifier ("=" constant-expression)?

declarator:
  pointer? (identifier | "(" declarator ")") (
    "[" constant-expression? "]" |
    "(" parameter-type-list ")" |
    "(" identifier%? ")"
  )*

pointer:
  ("*" type-qualifier*)*

parameter-type-list: parameter-declaration% ("," "...")?

parameter-declaration:
  declaration-specifiers (declarator | abstract-declarator)?

initializer: assignment-expression | "{" initializer% ","? "}"

type-name: (type-specifier | type-qualifier)+ abstract-declarator?

abstract-declarator:
  pointer ("(" abstract-declarator ")")? (
    "[" constant-expression? "]" |
    "(" parameter-type-list? ")"
  )*
```

```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
   block |
   "if" "(" expression ")" statement |
   "if" "(" expression ")" statement "else" statement |
   "switch" "(" expression ")" statement |
   "while" "(" expression ")" statement |
   "do" statement "while" "(" expression ")" ";" |
   "for" "(" expression? ";" expression? ";" expression? ")" statement |
   "goto" identifier ";" |
   "continue" ";" |
   "break" ";" |
   "return" expression? ";"
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: (
    unary-expression (
      "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" |
      "^=" | "|="
    )
  )* conditional-expression

conditional-expression:
  logical-OR-expression ( "?" expression ":" conditional-expression )?

constant-expression: conditional-expression

logical-OR-expression:
  logical-AND-expression ( "||" logical-AND-expression )*

logical-AND-expression:
  inclusive-OR-expression ( "&&" inclusive-OR-expression )*

inclusive-OR-expression:
  exclusive-OR-expression ( "|" exclusive-OR-expression )*

exclusive-OR-expression:
  AND-expression ( "^" AND-expression )*

AND-expression:
  equality-expression ( "&" equality-expression )*

equality-expression:
  relational-expression ( ("==" | "!=") relational-expression )*

relational-expression:
  shift-expression ( ("<" | ">" | "<=" | ">=") shift-expression )*

shift-expression:
  additive-expression ( ("<<" | ">>") additive-expression )*
```

```
additive-expression:
  multiplicative-expression ( ("+" | "-") multiplicative-expression )*

multiplicative-expression:
  cast-expression ( ("*" | "/" | "%") cast-expression )*

cast-expression:
  ( "(" type-name ")" )* unary-expression

unary-expression:
  ("++" | "--" | "sizeof" ) * (
    "sizeof" "(" type-name ")"                      |
    ("&" | "*" | "+" | "-" | "~" | "!" ) cast-expression |
    postfix-expression
  )

postfix-expression:
  (identifier | constant | string | "(" expression ")") (
    "[" expression "]"            |
    "(" assignment-expression% ")" |
    "." identifier                |
    "->" identifier               |
    "++"                          |
    "--"
  )*

constant:
  integer-constant |
  character-constant |
  floating-constant |
  enumeration-constant

C grammar ends here.

Notes:
  Empty struct declarations (struct foo { }) are not legal in the grammar.

  Neither are empty enum declarations (enum foo { }) or empty declaration
  lists (int;).
```

Some comments in the book indicate that the book's expression grammar captures both precedence and associativity. This was a matter of some concern to me; making iteration happen with Kleene stars instead of recursion eliminates the information on associativity. But the book appears to be incorrect; its grammar captures precedence, but none of the *-expression nonterminals are right-recursive, and most of them are left-recursive. So if you parse according to the grammar, all your operators will associate from left to right.

The split between cast-expression and unary-expression exists mainly to try to keep you from incrementing or decrementing the results of casts, I think, but it is ineffective, because an extra set of parens is all you need. In other words, --(int)x doesn't parse with this grammar, but --((int)x) does.

There are obviously many constraints on the language that the grammar cannot express. In particular, constant-expression is subject to some constraints, and many operators require modifiable lvalues for one of their operands. It appears that some attempt to capture this has been made in this grammar, but it would require a much larger grammar to be successful.

There are also obviously many pieces of semantic information that the original grammar conveyed by the name of the nonterminal that this grammar does not convey.

I suspect this grammar still needs some work before I can use it for a recursive-descent parser. I'm worried about how to tell labels from variable names starting C statements (they are in separate namespaces, so the typedef-name trick won't work) and how to tell casts from parenthesized expressions.

For fun, I wrote the following, in the same language as the C grammar.

```
Grammar grammar begins here:

Terminals: identifier quoted-string blank-line

grammar:
  blank-line*
  terminals-decl
  blank-line+
  (definition blank-line+)*
  definition?

terminals-decl: "Terminals" ":" identifier*

definition: identifier ":" alternation-regex

alternation-regex: simple-regex ("|" simple-regex)*

simple-regex:
  (
    (identifier | quoted-string | "(" alternation-regex ")")
    ("+" | "*" | "?" | "%")*
  )*

Grammar grammar ends here.
```