The D Programming Language Specification

D is a small imperative language. It has 32-bit integer constants, variables, and functions, and a minimal set of statements: assignment, if, while, and return. There are no modules, classes, objects, or external variables. There are no header files or libraries, but there are two predefined functions, get and put, which provide the ability to read and write integer values. Syntactically it is much like C. A program consists of a sequence of function definitions.

Grammar

Here is the basic grammar for D.

1. *program* ::= *functionDefinition* | *program functionDefinition*
2. *functionDefinition* ::= **`int`** *id* **( )** { *statements* }
  | **`int`** *id* **(** *parameters* **)** { *statements* }
  | **`int`** *id* **( )** { *declarations statements* }
  | **`int`** *id* **(** *parameters* **)** { *declarations statements* }
3. *parameters* ::= *parameter* | *parameters* **,** *parameter*
4. *parameter* ::= **`int`** *id*
5. *declarations* ::= *declaration* | *declarations declaration*
6. *declaration* ::= **`int`** *id* **;**
7. *statements* ::= *statement* | *statements statement*
8. *statement* ::= **{** *statements* **}** | *id* **=** *expr* **;**
  | **`if`** **(** *boolExpr* **)** *statement* | **`if`** **(** *boolExpr* **)** *statement* **`else`** *statement*
  | **`while`** **(** *boolExpr* **)** *statement* | **`return`** *expr* **;**
9. *expr* ::= *term* | *expr* **+** *term* | *expr* **-** *term*
10. *term* ::= *factor* | *term* ***** *factor*
11. *factor* ::= *int* | **(** *expr* **)** | *id* | *id* **( )** | *id* **(** *exprList* **)**
12. *exprList* ::= *expr* | *exprList* **,** *expr*
13. *boolExpr* ::= *relExpr* | **!** (*relExpr*)
14. *relExpr* ::= *expr* **==** *expr* | *expr* **>** *expr*

See next page for additional productions that are rewrites of the above list.

Rewritten productions to fix left recursion problems.

Instead of 1, use 15 and 16.

15. *program* ::= *functionDefinition programTail* **EOF**
16. *programTail* ::= *functionDefinition programTail* | ε

Instead of 3, use 17 and 18.

17. *parameters* ::= *parameter parametersTail*
18. *parametersTail* ::= **,** *parameter parametersTail* | ε

Instead of 5, use 19 and 20.

19. *declarations* ::= *declaration declarationsTail*
20. *declarationsTail* ::= *declaration declarationsTail* | ε

Instead of 7, use 21 and 22.

21. *statements* ::= *statement statementsTail*
22. *statementsTail* ::= *statement statementsTail* | ε

Instead of 9, use 23 and 24.

23. *expr* ::= *term exprTail*
24. *exprTail* ::= + *term exprTail* | - *term exprTail* | ε

Instead of 10, use 25 and 26.

25. *term* ::= *factor termTail*
26. *termTail* ::= * *factor termTail* | ε

Instead of 12, use 27 and 28.

27. *exprList* ::= *expr exprListTail*
28. *exprListTail* ::= **,** *expr exprListTail* | ε

It is also possible to replace the tail recursive statements above with iterations. For example:

parameters ::= *parameter* (**,** *parameter*)*
declarations ::= *declaration* (*declaration*)*

Features implemented in the Scanner class

There are two undefined nonterminals in the grammar: *id* and *int*. An integer, *int*, consists of 1 or more digits (0-9) and denotes a decimal integer. An identifier, *id*, must begin with a letter, and consists of 1 or more letters, digits, and underscores. Upper- and lower-case letters are distinct, thus aa, AA, Aa, and aA are four different identifiers.

Comments, blanks, and other whitespace are ignored except as needed to separate adjacent syntactic tokens. A comment begins with the token // and continues to the end of the line.

The keywords in the grammar (**`int, if`**, etc.) are reserved and may not be used as identifiers.

General comments

A program consists of one or more function definitions.

All functions should have distinct names, and one of the functions must be named main (not enforced). A program is executed by evaluating main().

All integer values are 32-bit, two's complement numbers (not enforced).

D includes binary arithmetic operators +, -, and *. There is no division operator or unary + or - operators. The value -n can be computed by evaluating 0-n.

A *boolExpr* is a logical expression, which may only be used as a condition in an if or while statement. Logical expressions do not have integer values and cannot be stored in variables.

In conditional statements, each else is paired with the nearest previous unpaired if.

All local variables must be declared at the beginning of a function, and each declaration introduces a single variable. The local variables and parameters in a function must have distinct names (not enforced), and their scope extends over the entire function definition.

All functions are integer-valued, including main.

Function execution must terminate by executing a return statement. It is an error to "fall off" the end of the list of statements that make up a function body (not enforced).

There are two predefined functions that provide integer input and output (not implemented).

get() yields the next integer value from the standard input. If the next non-whitespace characters in the input do not form an integer constant, execution of get() is not defined.

put(x) yields the value of x and, as a side effect, prints that value on the next line of the standard output. Like all function calls, put(x) is an expression, so it may not be used as a statement. The statement x=put(x); may be used to print the value of a variable.