DEMO

A Language for Practice Implementation

COMP 506, Spring 2019

1 Purpose

This document describes the DEMO programming language. DEMO was invented for instructional purposes; it has no real use aside from programming assignments in compiler-construction courses, such as COMP 506. In different years, we expect to use DEMO in different ways; thus, this document is separate from the handout that describes your specific assignment, and the document contains information that may seem irrelevant to your current assignment. It is, however, the ultimate definition of DEMO for your current assignment.

2 The DEMO Language

DEMO is a simple, Algol-like language. It is not intended as a replacement for any known programming language. In fact, by design it has few of the features that programmers find useful in writing large programs. It is intended to be simple enough to implement in a single semester, but powerful enough to illustrate many of the common features of Algol-like languages. It avoids many complications; each of its features was added to illustrate a specific problem that arises in the design and implementation of a compiler.

DEMO supports two data types: *integer* and *character*. Each of these types can be aggregated into vectors and arrays. You can assume that the underlying hardware supports integers with 32 bit, twos-complement arithmetic, and that four characters fit into a word. The type system is trivial; coercions from integer to character and back are not allowed.

DEMO supports a number of simple control structures: a compound statement, `while` and `for` loops, and an `if-then-else` construct. Limited input and output primitives are provided. At the moment, procedure calls are not defined in the language, although they are likely to be added in the future. Thus, a DEMO program is written as a single function.

2.1 Syntax

The following the context-free grammar describes the syntax of DEMO. It is written in a modified Backus-Naur Form [1, p. 87]. *Nonterminal symbols* are typeset in *slanted roman* text, while TERMINAL symbols are underlined and typeset in CAPITAL LETTERS.

Procedure → PROCEDURE NAME
{ *Decls Stmts* }

Decls → *Decls Decl* ;
| *Decl* ;

Decl → *Type SpecList*

Type → INT
| CHAR

SpecList → *SpecList* , *Spec*
| *Spec*

Spec → NAME
| NAME [*Bounds*]

Bounds → *Bounds* , *Bound*
| *Bound*

Bound → NUMBER : NUMBER

Stmts → *Stmts Stmt*
| *Stmt*

Stmt → *Reference* = *Expr* ;
| { *Stmts* }
| WHILE (*Bool*) { *Stmts* }
| FOR NAME = *Expr* TO
Expr BY *Expr* { *Stmts* }
| IF (*Bool*) THEN *Stmt*
| IF (*Bool*) THEN *Stmt*
ELSE *Stmt*
| READ *Reference* ;
| WRITE *Expr* ;

Bool → NOT *OrTerm*
| *OrTerm*

OrTerm → *OrTerm* OR *AndTerm*
| *AndTerm*

AndTerm → *AndTerm* AND *RelExpr*
| *RelExpr*

RelExpr → *RelExpr* LT *Expr*
| *RelExpr* LE *Expr*
| *RelExpr* EQ *Expr*
| *RelExpr* NE *Expr*
| *RelExpr* GE *Expr*
| *RelExpr* GT *Expr*
| *Expr*

Expr → *Expr* + *Term*
| *Expr* - *Term*
| *Term*

Term → *Term* * *Factor*
| *Term* / *Factor*
| *Factor*

Factor → (*Expr*)
| *Reference*
| NUMBER
| CHARCONST

Reference → NAME
| NAME [*Exprs*]

Exprs → *Expr* , *Exprs*
| *Expr*

This grammar is ambiguous. You will need to transform the grammar to make it suitable for the parsing technique that you use. You should consult reference materials for an explanation of how to transform a grammar for either LL(1) or LR(1) parser generators [1, Ch. 3].

2.2 Microsyntax

The following table specifies the spelling of the TERMINAL SYMBOLS in DEMO, except for NAME, NUMBER, and CHARCONST.

Terminal	Spelling
AND	and
BY	by
CHAR	char
ELSE	else
FOR	for
IF	if
INT	int
NOT	not
OR	or
PROCEDURE	procedure
READ	read
THEN	then
TO	to
WHILE	while
WRITE	write

Reserved Words

Terminal	Spelling
PLUS	+
MINUS	-
TIMES	*
DIVIDE	/
LT	<
LE	<=
EQ	==
NE	!=
GT	>
GE	>=

Operators

Terminal	Spelling
COLON	:
SEMICOLON	;
COMMA	,
BECOMES	=
LBRACE	{
RBRACE	}
LBRACKET	[
RBRACKET	]
LPAREN	(
RPAREN	)

Punctuation

Additional Notes

1. NAME is specified by the regular expression *Letter* (*Letter* | *Digit*)*, where *Letter* is any uppercase or lowercase English letter, from `a` to `z` and *Digit* is a digit between `0` and `9`. In `lex` or `flex`, we could write the rule for NAME as `[A-Za-z][A-Za-z0-9]*`
2. NUMBER is specified as the positive closure of *Digit*. In `lex` of `flex` notation, we could write NUMBER as `[0-9]+`, or `[0-9][0-9]*`.
3. CHARCONST contains a single character, surrounded by single quotes. Any valid ASCII character can appear between the quotes. Examples might be '`a`', '`b`', '`6`', and '`?`'. Since neither the forward single quote, `‘`, nor the backward single quote, `’`, have any other use in DEMO, the scanner should treat them as interchangeable.
4. In DEMO blanks are significant. Thus, "andor" is a valid NAME, while "and or" contains the terminals AND and OR. Reserved words, as shown in the table, cannot be used for any other purpose.
5. The characters "`//`" denote the start of a comment that runs until the end of the current input line.
6. Note that DEMO inherits the unfortunate spelling of the boolean equality operator (EQ) from C. EQ is spelled "`==`".

2.3 The Meaning of a DEMO Program

Program Structure A DEMO program consists of a header, followed by a set of declarations, followed by a set of executable statements. The header consists of the PROCEDURE keyword, followed by a procedure name. The declarations and executable statements are enclosed in curly braces ('{' and '}'). All statements, whether declaration or executable, end with a semicolon (';').

In the current incarnation of DEMO, a program consists of a single procedure. DEMO has no provision for either multiple procedures or procedure calls.[1]

Declarations DEMO supports two data types: integer and character. Integers occupy a single word in the ILOC virtual machine, while characters occupy $\frac{1}{4}$ of a word. Either integers or characters can be aggregated into arrays. Thus, an NAME may represent (1) an integer variable, (2) an integer array, (3) a character variable[2], or (4) a character array.

NAMEs must be declared in a *Decl* statement. A given NAME can only occur in one *Decl* statement. For example, an integer array `A`, with dimensions of [1,10] and [2,8] would be declared as

```
int A[1:10,2:8];
```

DEMO is case sensitive; that is, `a` and `A` are distinct NAMEs. Given the declaration shown above, `A` cannot appear in another declaration, while `a` would be legal.

Example

```
int a, b[1:10], c[1:10,1:100];
char d, e[1:10], f[1:10,1:100];
```

declares three integer variables, `a`, `b`, and `c`, as well as three character variables, `d`, `e`, and `f`. `a` and `d` are scalar values. `b` and `e` are both vectors with ten elements, starting at index 1. `c` and `f` are both two-dimensional arrays, with the given bounds.

Storage Assignment
Because the current version of DEMO only supports a single procedure, a DEMO compiler has unusual freedom in storage layout. All variables have the same lifetime as the current procedure—they are *local* variables in the sense that term is used in Algol-like languages.

- Scalar variables can be assigned either a location in data memory or in a virtual register.
- Aggregates (e.g., vectors and multi-dimensional arrays) must be assigned locations in data memory.

The ILOC virtual machine has separate data and program memory. It requires that integer references be word-aligned, but places no other restrictions on the use of data memory.

Executable Statements DEMO supports an assignment statement, a compound construct, three control structures, and two statements for handling input and output (I/O).

[1] As experience grows with COMP 506, we may add provisions for multiple procedures and procedure calls.

[2] A character variable holds exactly one character

Assignment
The assignment statement requires that its left-hand side and its right-hand side evaluate to the same type. DEMO does not support coercions, so a character LHS requires a character RHS and an integer LHS requires an integer LHS. Similarly, the RHSmust evaluate to a single value; if the LHS refers to an aggregate (vector or array), it must evaluate to a single element of that aggregate.

Compound Statement
The compound statement consists of a list of statements surrounded by curly braces, { and }. From the perspective of any surrounding control-flow construct, a compound statement is treated as a single statement; that is, if any statement in the compound statement executes, they all execute in an order equivalent to the order given in the source program.

The grammar allows compound statements to be nested, as in

```
{ a = 0; { b = 0; { c = 0; } } { d = 0; } e = 1; }
```

None of the blanks in this example compound statement are required. The blanks are, however, legal and were added to improve readability.

DEMO does not allow an empty statement or statement list. Thus, any of

```
x = 1; ;  // second semicolon is anempty statement
{ }       // empty statement list
{ ; }     // empty statement in a list
```

should produce a syntax error.

Control Structures
DEMO supports three control-flow constructs: a `for` loop, a `while` loop, and an `if-then-else` construct.

Boolean Values The `while` loop and the `if-then-else` require Boolean expressions, a *Bool* in the grammar. In DEMO, a Boolean expression is evaluated as an integer expression, with the value zero representing `false` and any non-zero value representing `true`.

For Loop The `for` loop is a standard iterative loop, with fairly typical semantics. The loop header reads as:

`for` iv `=` ex_1 `to` ex_2 `by` ex_3

The loop's index variable, iv takes on a series of values from ex_1 to ex_2 in increments of ex_3. The index variable must be an integer, and all of ex_1, ex_2, and ex_3 must evaluate to integer values. If $ex_1 > ex_2$, the statement list under loop's control does not execute.

While Loop The `while` loop provides a simple mechanism for iteration. It executes the statement list under its control until the controlling expression evaluates to `false` (see the earlier discussion of Boolean values). The order of execution of a while loop is always:

(1) evaluate the controlling expression;
(2) if the result is `true`, then execute the controlled statement and go back to step (1).

If-Then-Else The `if-then-else` construct allows conditional control of a statement. An `if-then-else` first evaluates the controlling expression to a Boolean value (see the earlier discussion of Boolean values). If the result is `true`, the code executes the statement under the `then` clause. If the result is `false` and the `then` clause has a matching `else` clause, the statement under the `else` clause executes.

The grammar for the `if-then-else` construct is ambiguous. It embodies the classic example of an ambiguous grammar. You should rewrite that portion of the grammar to resolve the "dangling else" ambiguity so that each `else` clause is matched with the most recent unmatched `then` clause.

Input/Output DEMO provides two simple I/O statements:

Read The `read` statement reads a single data item from the system's standard input stream (`stdin` in C terminology). The type of the value read is determined by the type of the *Reference*. Each value read from `stdin` must be separated from surrounding values by either blanks or a newline.

Write The `write` statement writes a single data item to the system's standard output stream (`stdout` in C terminology). The type of the value written is determined by the type of the *Expr*. Each value printed with a write statement appears on its own line; that is, the value is printed, followed by a newline.

Expression Evaluation DEMO expressions compute simple values of type integer or character. Since DEMO currently has no character operators, the only meaningful character expression is a single literal character, a CHARCONST. No coercions are allowed, so only integer subexpressions can occur in an integer expression.

In DEMO, all operators are left associative. The precedence of operators is specified by the following table, where larger numbers mean a higher priority.

Operator	Precedence
`( )`	6
`*`, `/`	5
`+`, `-`	4
`<`, `<=`, `==`, `>`, `>=`, `!=`	3
`not`	2
`and`, `or`	1

The relational operators, `<`, `<=`, `==`, `>`, `>=`, and `!=` can only be applied to operands of the same type. Thus, both `'a' <= 'b'` and `13 > 2` are legal, while `'a' != 1` is not. The latter example exhibits a type mismatch—an error.

The other operators, `+`, `-`, `*`, `/`, `not`, `and`, `or` can only be applied to operands of type integer. The use of integer values to represent Boolean values creates some unusual behavior. For example, `(a == b) or (c < d)` is equivalent to `(a == b) + (c < d)`, although the parentheses are required in the expression that uses `+` and are unneeded in the expression that uses `or`.

3 Example Program

```
// Example program in DEMO
// A simple bubblesort
// (Assumes data initialized by simulator)

procedure bsort {

  int DATA[0:10000];
  int i, upper, flag, temp;

  read upper; // # elements to sort

  if (upper > 10000) then {
      upper = 0; // a primitive quit
      write 1;  // a primitive message
  }

  for i = 0 to upper - 1 by 1 {
      read DATA[i];
  }

  flag = 1;
  while( flag == 1) {
      flag = 0;
      for i = 0 to upper - 2 by 1 {
          if (DATA[i] > DATA[i+1]) then {
             flag = 1;
             temp = DATA[i];
             DATA[i] = DATA[i+1];
             DATA[i+1] = temp;
          }
      }
  }

  for i = 0 to upper - 1 by 1 {
      write DATA[i];
  }

}
```

References

[1] Keith D. Cooper and Linda Torczon. *Engineering a Compiler, 2nd Edition.* Elsevier, Morgan Kaufmann, 2012.