idc

Intro

idc is a simplified variant of the dc calculator, intended to show o the spin literate programming system. It only operates on integers, and is thus mostly a stack-based interactive variant of C integer semantics. It only operates on 32-bit integers.

This program is structured in 2 pieces, a dialect and an application. The dialect holds generic data structures and general functions that could be provided by a library. The application consists of the main function and other specialized code.

Dialect

We use stacks to manage the data. We also use a multibranch loop as introduced by Dijkstra in *A Discipline of Programming* (see the do...od construct defined in chapter 4) and championed by Wirth in the Oberon version of *Algorithms and Data Structures* (see appendix C). We also use the stdio library, and define single-lookahead streams on top. This with a character set functionality is used in our numeric input function. Finally, we have a die function that handles error reporting and termination.

Stack

We define the stack as a global struct grouping together the bu er and the top pointer. This allows us to use the struct for namespacing, instead of ad hoc name conventions. The exception is in the initialization function, since we can't assign it inline, and we only call it once. We also hard-code the size of the stack, and treat overflow and underflow as fatal errors. This is purely for simplicity, in a real application, these would be treated as errors to be handled.

```
 1 #define SIZE 128
 2 struct {
 3      uint32_t bu▫er[SIZE];
 4      uint32_t * top;
 5 } stack = {0};
 6
 7 void stack_init() {
 8      stack.top = stack.bu▫er - 1;
 9 }
10
11 #define Top stack.top[0]
12 #define Next stack.top[-1]
13
14 void push(uint32_t x) {
15      if(stack.top - stack.bu▫er > SIZE)
16           die("overflow");
17
18      stack.top++;
19      Top = x;
20 }
21
22 void drop() {
```

```
23      if(stack.top < stack.bu□er)
24           die("underflow");
25
26      stack.top--;
27 }
```

Dloop

The Dijkstra loop provides a clean way to structure conditional loops. This allows you to still use the normal elsif keywords within the loop.

```
1 #define Dloop(condition) while(1) {if(condition)
2 #define Dorelse(condition) else if(condition)
3 #define Dend else break;}
```

stdio

We mainly use the stdio library for printf, getchar and EOF, and we use exit from stdlib. We also include stdint here, since we only use int for booleans.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
```

Stream Input

This single-lookahead stream also handles end-of-file conditions, freeing us from dealing with that throughout the code. Since we only are ever looking at a single stream of data, we simply use a global variable to hold the lookahead. This eases congestion with function parameters.

```
1 char c;
2
3 void next() {
4      int x = getchar();
5      if (x == EOF) exit(0);
6      c = x;
7 }
```

Charsets

Rather than using the library provided class facility that depends on locale and such, we provide a simpler, more flexible interface here.

The idea is that the only ordering allowed is byte ordering, and you can provide either explicitly enumerated sets, or a union of byte ranges, both encoded as strings. I've found this to make otherwise complicated parsing code quite tractable.

```
 1 int in(char c, char *set) {
 2      if(!set) return 0;
 3      while(*set) {
 4           if(c == *set) return 1;
 5           set++;
 6      }
 7      return 0;
 8 }
 9
10 int within(char c, char *rangeset) {
11      if(!rangeset) return 0;
12      while(rangeset[0] && rangeset[1]) {
13           if (rangeset[0] <= c
14                     && c <= rangeset[1])
15                return 1;
16           rangeset += 2;
17      }
18      return 0;
19 }
```

Numeric Input

We only do decimal for illustration purposes, but the implementation is easiliy extended. This also provides insight into how to use the ranged charset function within.

```
1 uint32_t num() {
2      uint32_t n = 0;
3      while(within(c, "09")) {
4           n *= 10;
5           n += c - '0';
6           next();
7      }
8      return n;
9 }
```

Errors

Having a die function cleans up a lot of code, since we can just assume functions worked successfully. This is only called if you

have complete control of the code. For example, when defining a library, the client code should be in charge of handling errors, so this would be inappropriate from within the library.

```
1 void die(char *s) {
2       fputs(s, stderr);
3       fputc('0, stderr);
4       exit(1);
5 }
```

Application

Main

We are using the Dijkstra loop to ease the control structure. In general I prefer to implicitly thread control through data structures, but due to the sparsity of the input, it makes sense to directly encode the control.

```
 1 int main() {
 2      stack_init();
 3      next();
 4      Dloop(c == 'q')
 5           return 0;
 6      Dorelse(c == 'p') {
 7           printf("%d0, Top);
 8           next();
 9      }
10      Dorelse(in(c, " 0))
11           next();
12      Dorelse(within(c, "09"))
13           push(num());
14      Dorelse(1) {
15           operate();
16           drop();
17           next();
18      }
19      Dend
20      return 1;
21 }
```

Operations

Note that each operation is expected to consume one character from the input stream, and to modify the second element in the stack. The main function drops the top element, and advances the stream.

```
 1 void operate() {
 2      switch(c) {
 3      case '+':
 4           Next += Top;
 5           break;
 6      case '-':
 7           Next -= Top;
```

```
 8           break;
 9      case '*':
10           Next *= Top;
11           break;
12      case '/':
13           Next /= Top;
14           break;
15      case 's':
16           Next <<= Top;
17           break;
18      case 'S':
19           Next >>= Top;
20           break;
21      default:
22           fputs("?", stderr);
23      }
24 }
```