NAME

- `opcount` - count operations in a cryptographic scheme

SYNOPSIS

```
opcount [{preamble} ...] [-t {postamble} ...] < {filename} | gp -q
```

DESCRIPTION

The `opcount` is a filter program that counts group operations in a cryptographic scheme written in C-like scheme description language (CSDL), and outputs some sequence of assignment statements of formulae for GP (PARI Calculator). It reads a CSDL code from the standard input, then writes the counting results to the standard output. If `{preamble} ...` are specified, it outputs the contents of the files `{preamble} ...` as it is in the specified order before the counting results. In the same way, if `-t {postamble} ...` are specified, it outputs the contents of the files after the counting results. It is preferable to prepare files of `preamble` and `postamble` so that the total output of the `opcount` constitutes a complete script for the GP calculator.

ALGEBRA OF THE OPERATION COSTS OF MULTI-BASE EXPONENTIATION

Consider the multi-base exponentiation $\prod_{i=0}^{n} g_i^{x_i}$ where $n \in \mathbb{N}_0$ and $x_0 \in \{0,1\}$. $x_0,\dots,x_n$ are variables represent elements in a commutative ring $\mathbb{L}$ with a unit, and called exponents. $g_0,\dots,g_n$ are variables represent elements in some group ($\mathbb{L}$-module), and called bases. When we design a cryptographic scheme, we do not need to care about the actual elements stored in the exponents or bases. Instead we assume that there exists such an element virtually (formal multi-base exponentiation).

For example, when $x_0 = 0$, we don't have to consider the element g_0, but we assume that there exists such an element virtually. The reason why x_0 is restricted in $\{0,1\}$ unlike the other x_i's is that this expression make it easy to handle both the multiplication and the exponentiation on $\mathbb{G}$.

If all bases are single variables of type $\mathbb{G}$, we write (n, x_0) to express the operation cost to calculate $\prod_{i=0}^{n} g_i^{x_i}$.

For example, the operation cost to calculate a variable g of type $\mathbb{G}$ without any exponentiation or multiplication can be written as $(0,1)$, because $g = \prod_{i=0}^{0} g_i^{x_i}$ where $g_0 = g$ and $x_0 = 1$. Of course,

$$(0,1) = 0.$$

To derive algebraic operations of costs of formal multi-base exponentiation, we consider a representation s.t. all multi-base exponentiation in an algebraic representation are replaced into corresponding operation costs.

For example, Let `mul` be the cost of multiplication of two variables of type $\mathbb{G}$. For $g,h,k \in \mathbb{G}$, we can consider the following correspondence.

actual operation	algebraic representation of cost	actual cost
$h \times k \to g$	$(0,1)\times(0,1)\to(0,1)$	`mul`.

Considering this correspondence, we can regard

$$(0,1)\times(0,1) = (0,1) + \texttt{mul}.$$

Similarly, we can consider such a correspondence for general multi-base exponentiation. That is

actual operation	algebraic representation of cost
$\left(h_0^{y_0}\prod_{i=1}^{m} h_i^{y_i}\right)\times\left(k_0^{z_0}\prod_{i=1}^{\ell} k_i^{z_i}\right)\to g_0^{x_0}\prod_{i=1}^{n} g_i^{x_i},$	$(m,y_0)\times(\ell,z_0)\to(n,x_0),$

where $n, m, \ell \in \mathbb{N}_0$, $x_0, y_0, z_0 \in \{0, 1\}$. Let

$$g_i = \begin{cases} h_0^{y_0} \times k_0^{z_0} & i = 0, \\ h_i & i \in \{1, \dots, m\}, \\ k_{i-m} & i \in m + \{1, \dots, \ell\}. \end{cases}$$

We can assume that

$$n = m + \ell,$$
$$x_0 = y_0 \vee z_0.$$

When $y_0 \wedge z_0 = 1$, we need one multiplication of bases to calculate the new base $g_0 = h_0 \times k_0$. Therefore we can derive

$$(m, y_0) \times (\ell, z_0) = (m + \ell, y_0 \vee z_0) + (y_0 \wedge z_0) \cdot \texttt{mul}.$$

In the same manner, we can derive the cost of an exponentiation of multi-base exponentiation.

actual operation	algebraic representation of cost
$\left(h_0^{y_0} \prod_{i=1}^{m} h_i^{y_i}\right)^z \to g_0^{x_0} \prod_{i=1}^{n} g_i^{x_i},$	$(m, y_0)^z \to (n, x_0),$

where, $n, m \in \mathbb{N}_0$, $x_0, y_0 \in \{0, 1\}$. Let

$$g_i = \begin{cases} h_i & y_0 = 0,\ i \in \{0, \dots, m\}, \\ h_{i-1} & y_0 = 1,\ i \in \{0, \dots, m+1\}, \end{cases}$$

where h_{-1} is an appropriate variable. we can assume that

$$n = m + y_0,$$
$$x_0 = 0$$

That is

$$(m, y_0)^z = (m + y_0, 0).$$

`opcount` counts this costs of multi-base exponentiation, then outputs them.

HOW TO BUILD

First of all, prepare the follows.

- UNIX-like development environment (`sh`,`mv`,`cp`,`rm`,`chmod`,`cat`,`sed`,`awk`,`perl`,`make`,...)
- yacc or bison parser generator
- lex or flex lexical analyzer generator
- clang or gcc c99/c++11 compiler
- boost C++ LIBRARIES
- PARI/GP computer algebra system (not necessary to build)
- source code of `opcount`

Change the current directory to the source code directory (which contains Makefile) and then, on the command line (shell), just type

```
make
```

`opcount` is confirmed to build correctly under the following environments with an appropriate configuration.

Environment	Version	Architecture	Note
FreeBSD	10.1	amd64	yacc/lex/clang
CentOS	6.8	x86_64	bison/flex/gcc

OPTIONS

- `-t` outputs the contents of the files specified after this option as a postamble.
- `-v` outputs the version number of `opcount` as a GP formula.
- `-p` automatically generate a postamble to display the results.

C-LIKE SCHEME DESCRIPTION LANGUAGE

C-like Scheme Description Language (CSDL) is a domain specific language to analyze data flow of pairing-based cryptographic schemes. Its syntax is similar to (its formal syntax is almost the same as) that of C language (C89) without preprocessor.

Comment of CSDL is roughly the same as C++, starts with `/*` and end with `*/`, or starts with `//` and extending to the next newline character.

The most syntactically significant difference between CSDL and C is interpretation of the ˆ operator.

In C language, the ˆ operator represents the bitwise xor operation, and has lower precedence than any arithmetic operator. While in CSDL, it represents the exponent operation like TeX, has higher precedence than any arithmetic operator and right-associativity. Use `xor` of `(+)` for xor operator (`xor_eq` or `(+)=` for xor assignment) instead.

In CSDL, a description of an algorithm must be placed inside a function such as

```
main(){
    ...
}
```

It is no matter that function named other than `main` is defined. Multiple functions can be defined if all functions have unique names in a file. For example:

```
setup(){ ... }
sign(){ ... }
verify(){ ... }
```

DATA TYPE

Other than the standard C data types, CSDL support the following data types.

Type name	Explanation
`group`	source group $\mathbb{G}$ of pairing
`group0`	source group $\mathbb{G}_0$ of asymmetric pairing
`group1`	source group $\mathbb{G}_1$ of asymmetric pairing
`group2`	source group $\mathbb{G}_2$ of asymmetric pairing
`target`	target group of pairing
`integer`	integer ((commutative subring of) the endomorphisms of the above groups)
`list`	list data type

To declare variables, write as follows (declaration statement).

```
group0   A,B,C ;
group1   X,Y,Z ;
integer alpha, beta, gamma, delta ;
target  gT ;
```

Variables can be initialized when they are declared (initialization statement). For example:

```
group1 W = X*Y*Z ;
```

Like C++ or C99, declaration or initialization statements can be placed in any lines unless a variable with the same name is already declared in the same scope level.

To assign a variable, write as follows (assignment statement).

```
X = Y^alpha * Z^beta ;
```

The left side must be a so called left value. Assignment such as

```
X^2 = Y^alpha * Z^beta ;
```

are illegal.

LIST VALUE AND LIST VARIABLE

There are very few syntactic differences between CSDL and C. Although there is no significant meaning to discuss the semantics of CSDL that the `opcount` assumes, since it has just a function to count operations, interpretation of the `,`(comma) operator is a major difference between them.

In the C language, the `,`(comma) operator is evaluated from left to right, then the evaluated value of the left is dropped. On the other hand, in CSDL, it is assumed that the left is not dropped.

Since the `,`(comma) operator has the lowest precedence in CSDL as in the C language, in order to operate a list value, it must be enclosed in parentheses like

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

If all elements in a list value are left values, then the list becomes a left value, i.e. assignable.

```
integer a, b, c, d, e, f ;
(a,b,c) = (d^2, e^3*f, e*d/f) ;
```

List can be nested.

```
integer a, b, c, d, f ;
((a,b),c) = ((d^2, d^3*f), d/f) ;
```

A list value can be assigned to a variable of type `list` (list variable).

```
integer a, b, c ;
list L = (a, b, c) ;
```

FUNCTION

Functions must be defined as in the C language. If the return type is not specified, it is assumed to be list. If there is a prototype declaration of a function, then the function is available after the declaration.

```
integer random() ;

setup(){
  group   g   ;
  integer msk = random() ;
  group   y   = g^msk ;
  list    pp  = (g,y) ;
  return  (pp,msk) ;
}
```

SPECIAL FORM

The following special forms are available. Notice that if there is a variable or a function of the same name of a special form, the form is not available in the scope. It frequently happens that a variable `e` is declared carelessly.

Special form	Assumed function
`target e (group1 g1, group2 g2)`	pairing
`group  Repeat (integer n, group  g)`	repetition
`target product(integer n, target t)`	product

Some special forms are converted into some functions assumed to represent the number of some operations. In such a case, the corresponding functions must be defined in the pari/gp script.

EXAMPLES

EXAMPLE 1

Suppose that `hello_world.csdl` is a text file which contains the following code.

```
group    hello ;
integer  world ;

main(){
  return hello^world ;
}
```

By inputting `hello_world.csdl` to `opcount`, the following output is obtained. (Note that the line followed by `$` represents the input of the command line.)

```
$ ./opcount < hello_world.csdl
call_main = (1)*GROUP_batch(1*1,0)+(1)*LIST_VARIABLE_assign ;
```

When a function is defined in csdl, `opcount` counts the operation cost of the function, then output a script to assign the value to the variable `call_`{function name}. In the above case, it outputs an assignment to `call_main`, since `main()` function is defined in the source. Here `GROUP_batch(`n`,`b`)` represents the operation cost of multi-base exponentiation of type `group`, i.e. (n, b). Although the computer algebra system PARI/GP cannot evaluate this script just as is, it can be handled with appropriate definitions in preamble e.g.

```
GROUP0_batch(n,b) = n * GROUP0_pow + (n-1+b) * GROUP0_mul ;
```

The `LIST_VARIABLE_assign` in the above output is a term generated by the `return` statement, so in the preamble, we can configure

```
LIST_VARIABLE_assign = 0 ;
```

to ignore this term. Suppose that `preamble` is a gp script as in the followings.

```
GROUP_batch(n,b)     = n * GROUP_pow   + (n-1+b) * GROUP_mul   ;
GROUP0_batch(n,b)    = n * GROUP0_pow  + (n-1+b) * GROUP0_mul  ;
GROUP1_batch(n,b)    = n * GROUP1_pow  + (n-1+b) * GROUP1_mul  ;
TARGET_batch(n,b)    = n * TARGET_pow  + (n-1+b) * TARGET_mul  ;
pairing_batch(n)     = n * Miller_Loop + (n-1)   * TARGET_mul + Final_Exponentiation ;

LIST_VARIABLE_assign = 0 ;
LIST_LITERAL_assign  = 0 ;
GROUP_assign         = 0 ;
GROUP0_assign        = 0 ;
GROUP1_assign        = 0 ;
TARGET_assign        = 0 ;
INTEGER_assign       = 0 ;
```

When `hello_world.csdl` is input to `opcount` with specifying `preamble` and `-p` option which generates a postamble to display the results automatically, and then the result is piped to gp, the following output is obtained.

```
$ ./opcount preamble -p < hello_world.csdl | gp -q
main = GROUP_pow
```

EXAMPLE 2

Suppose that `elgamal.csdl` is a text file which contains the following code.

```
group   g ;
integer random() ;

list setup(){
  integer x = random() ;
  group   y = g^x ;
  return  (x,y) ;
}

list enc(group y, group m){
  integer r = random() ;
  return  (g^r, m * y^r) ;
}

group dec(integer x, list c){
  group c1 , c2 ;
  (c1, c2) = c ;
  return c2/c1^x ;
}
```

In this example, an indistinct function `random()` is declared as a prototype. A function declared as a prototype is available in the definitions of other functions without its definition. By processing `elgamal.csdl` with `opcount` and `gp`, the following output is obtained.

```
  $ ./opcount preamble -p < elgamal.csdl | gp -q
  setup = GROUP_pow + call_random
  enc = 2*GROUP_pow + (GROUP_mul + call_random)
  dec = GROUP_pow + GROUP_mul
```

If a function defined or declared as a prototype is called in the definition of other function, it is counted as a `call_`{function name}. For example, the above `random()` function is counted as a `call_random`.

EXAMPLE 3

Suppose that `boneh_franklin.csdl` is a text file which contains the following code.

```
  group   g0 ;
  group   g1 ;
  group   hash2point(char * ID) ;
  integer random() ;

  setup(){
    integer x = random() ;
    group   y = g1^x ;
    return  (x,y) ;
  }

  keygen(char * ID, integer x){
    return hash2point(ID)^x ;
  }

  enc(target m, char * ID, group  y){
    integer r  = random() ;
    group   c1 = g0^r ;
    target  c2 = m * e(hash2point(ID),y^r) ;
    return (c1,c2) ;
  }

  dec(list c, group  d_ID){
    group  c1 ;
    target c2 ;
    (c1,c2) = c ;
    target m = c2/e(c1,d_ID) ;
    return m ;
  }
```

In this example, a cryptographic scheme is defined by using a special form `e(·,·)` (pairing). By processing `boneh_franklin.csdl` with `opcount` and `gp`, the following output is obtained. (Note that \\ at the end of a line represents continuation to the next line.)

```
$ ./opcount preamble -p < boneh_franklin.csdl | gp -q
setup = GROUP_pow + call_random
keygen = GROUP_pow + call_hash2point
enc = 2*GROUP_pow + (call_random + (call_hash2point + (TARGET_mul\\
 + (Miller_Loop + Final_Exponentiation))))
dec = (Miller_Loop + (Final_Exponentiation + TARGET_div))
```

Pairing is counted as an independent operation cost from that of the target group. A product of n pairings is counted as a `pairing_batch`(n).