Perl 6 на высоких нагрузках

Как ускорить программу на Perl 6

Как ускорить программу на Perl 6 и сделать ее в 10 000 раз быстрее

Как ускорить программу на Perl 6 и сделать ее в 10 000 раз быстрее

Как ускорить программу на Perl 6 и сделать ее в 10 000 раз быстрее, привлекая принцип неопределенности

Берем

простую программу

Берем ресурсоемкую простую программу

```
say is_prime(@*ARGS[0]);

sub is_prime($n) {
    for 2 .. $n - 1 -> $d {
        return 0 unless $n % $d;
    }

    return 1;
}
```

```
$ perl6
```

```
$ perl6 is_prime.p6
```

```
$ perl6 is_prime.p6 37
```

```
$ perl6 is_prime.p6 37
1
```

```
$ perl6 is_prime.p6 38
0
```

```
$ perl6 is_prime.p6 15511
```

```
$ perl6 is_prime.p6 15511
1
```

Rakudo выполняет это 20 секунд :-(

Perl 6 может работать быстро!

Perl 6 может работать быстро!

Perl 6 может работать быстро!

$$T \cdot t = \mathrm{const}$$

Время разработки

$$T \cdot t = \text{const}$$

Время разработки

$$T \cdot t = \text{const}$$

Время выполнения

Boost it!

ОК, идем на boost.org :-)

```
struct p6grammar: public grammar<p6grammar> {
    template<typename ScannerT>
    struct definition {
        definition(p6grammar const& self) {
```

```
struct p6grammar: public grammar<p6grammar> {
    template<typename ScannerT>
    struct definition {
        definition(p6grammar const& self) {
            program
                = statement_list >> end_p;
```

```
struct p6grammar: public grammar<p6grammar> {
    template<typename ScannerT>
    struct definition {
        definition(p6grammar const& self) {
            program
                = statement_list >> end_p;
            statement_list
                = *statement;
```

```
struct p6grammar: public grammar<p6grammar> {
    template<typename ScannerT>
    struct definition {
        definition(p6grammar const& self) {
            program
                = statement_list >> end_p;
            statement_list
                = *statement;
            statement
                = statement_code
                  >> !statement_separator;
```

```
statement_code
    = comment
    | statement_control
    | sub_def
    | sub_call
    | expression;
```

```
statement_code
    = comment
    | statement_control
    | sub_def
    | sub_call
    | expression;
comment
    = ch_p('#')
      >> lexeme_d[*~ch_p('\n')];
```

```
statement_code
    = comment
    | statement_control
    | sub_def
    | sub_call
    | expression;
comment
    = ch_p('#')
      >> lexeme_d[*~ch_p('\n')];
sub_call
    = sub_name
      >> !sub_arguments;
```

```
struct p6grammar: public grammar<p6grammar> {
    template<typename ScannerT> struct
definition {
        definition(p6grammar const& self) {
            program
                = statement_list >> end_p;

            statement_list
                = *statement;

            statement
                = statement_code >> !
statement_separator[&a_statement];

            statement_code
                = comment
                | statement_control
                | sub_def
                //| (sub_call[&a_sub_call_start]
>> !statement_modifier)[&a_sub_call_end]
                | sub_call
                | expression;

            comment
                = ch_p('#') >>
lexeme_d[*~ch_p('\n')];

            statement_separator
                = ch_p(';');

            sub_call
                = (sub_name >> !sub_arguments)
[&a_sub_called];

            sub_name
                = builtin_sub_name
                | identifier[&a_sub_name]
[&a_sub_calling];

            builtin_sub_name
                = str_p("say")[&a_builtin_say]
                | str_p("print")
[&a_builtin_print]
                | str_p("return")
[&a_builtin_return];

            identifier
                = lexeme_d[+(alpha_p | '_') >>
*(alnum_p | '_')];

            sub_arguments
                = ch_p('(') >> !argument_list >>
ch_p(')')
                | argument_list;

            argument_list
                = list_p(argument, ch_p(',')
[&a_argument_list_sep]);
```

```
            argument
                = expression
                | variable;

            variable
                = array
                | scalar;

            scalar
                = constant
                | ch_p('$') >>
identifier[&a_scalar]
                | array_element;

            array
                = simple_array
                | global_array;

            simple_array
                = ch_p('@') >> (identifier >>
~eps_p('['))[&a_array];

            global_array
                = str_p("@*") >> (identifier >>
~eps_p('['))[&a_global_array];

            array_element
                = simple_array_element
                | global_array_element;

            simple_array_element
                = (ch_p('@') >>
array_element_name_and_index)[&a_array_element];

            global_array_element
                = (str_p("@*") >>
array_element_name_and_index)
[&a_global_array_element];

            array_element_name_and_index
                =
identifier[assign_a(array_element_name)]
                  >> ch_p('[') >> array_index
                  >> ch_p(']');

            array_index
                =
int_p[assign_a(array_element_index)];

            constant
                = real_p[&a_constant];

            sub_def
                = str_p("sub")[&a_sub_def_start]
                >> sub_name
                >> !
sub_arguments[&a_sub_def_arguments_end]
                >> sub_body[&a_sub_def_end];
```

```
             sub_body
                = ch_p('{')
[&a_sub_def_body_start] >> !statement_list >>
ch_p('}');

            statement_control
                = for_cycle
                | conditional_block;

            conditional_block
                = (str_p("if") |
str_p("unless"))[&a_condition_start] >>
expression[&a_condition_end] >>
code_block[&a_conditional_block_end];

            code_block
                = ch_p('{') >> statement_list >>
ch_p('}');

            for_cycle
                = str_p("for")
[&a_for_cycle_start]
                >> (range >> !(str_p("->") >>
ch_p('$') >> identifier[&a_for_cycle_variable]))
[&a_for_cycle_range]
                >>
for_cycle_body[&a_for_cycle_body];

            range
                = ch_p('(') >> range_content >>
ch_p(')')
                | range_content;

            range_content
                = expression >> str_p("..")
[&a_range_sep] >> expression;

            expression
                = ch_p('(')[&a_paren] >>
expression_content  >> ch_p(')')[&a_paren]
                | expression_content;

            expression_content
                = scalar >>
*(math_op[&a_math_op] >> expression)
                | sub_call;

            math_op
                = ch_p('+')
                | ch_p('-') >> eps_p(~ch_p('>'))
                | ch_p('*')
                | ch_p('/')
                | ch_p('%');

            for_cycle_body
                = sub_body;

            statement_modifier
                = (str_p("unless") |
str_p("if")) >> expression;
```

```
say is_prime(@*ARGS[0]);

sub is_prime($n) {
    for 2 .. $n - 1 -> $d {

        return 0 unless $n % $d;

    }

    return 1;
}
```

```
say is_prime(@*ARGS[0]);

sub is_prime($n) {
    for 2 .. $n - 1 -> $d {

        return 0 unless $n % $d;

    }

    return 1;
}
```

```
say is_prime(@*ARGS[0]);

sub is_prime($n) {
    for 2 .. $n - 1 -> $d {
        unless $n % $d {
            return 0;
        }
    }

    return 1;
}
```

Программа 1

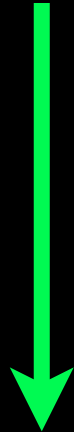
Проверяет аргумент на «простоту»

```
say is_prime(@*ARGS[0]);

sub is_prime($n) {
    for 2 .. $n - 1 -> $d {
        unless $n % $d {
            return 0;
        }
    }

    return 1;
}
```

```
./p++ < t/is_prime.p6
```

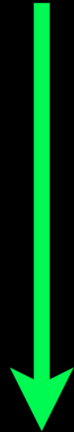
```
#include<iostream>
#include<string>

using namespace std;

typedef int variable_t;
typedef int scalar_t;
typedef int range_boundary_t;

variable_t sub_is_prime(scalar_t var_scalar_n) {
    for (range_boundary_t var_scalar_d = 2; var_scalar_d <=
var_scalar_n - 1; var_scalar_d++) {
    if(!(var_scalar_n % var_scalar_d)) {
    return 0;
}
    }
    return 1;
}

int main(int argn, char** var_global_array_ARGS) {
    cout << sub_is_prime(atoi(var_global_array_ARGS[1 + 0])) << "\n";
    return 0;
}
```

```
say is_prime(@*ARGS[0]);
```



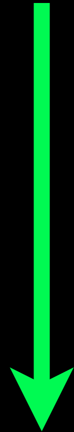
```
cout << sub_is_prime(
            atoi(var_global_array_ARGS[1 + 0])
        )
    << "\n";
```

```
sub is_prime($n) {
```



```
variable_t sub_is_prime(scalar_t var_scalar_n)
{
```

```
for 2 .. $n - 1 -> $d {
```



```
for (range_boundary_t var_scalar_d = 2;
     var_scalar_d <= var_scalar_n - 1;
     var_scalar_d++) {
```

```
$ time ./is_prime 15511
1
```

```
$ time ./is_prime 15511
1

real 0m0.002s
```

Программа 2

Проверяет аргумент на «простоту»
и сообщает первый найденный делитель

```
say is_prime(@*ARGS[0]);

sub is_prime($n) {
    for 2 .. $n - 1 -> $d {
        unless $n % $d
            say $d;
            return 0;
        }
    }

    return 1;
}
```

Программа 3

Находит простые числа, меньшие 1 000 000

```
for 2 .. 1000000 -> $n {
    if is_prime($n) {
        say $n;
    }
}

sub is_prime($n) {
    for 2 .. $n - 1 -> $d {
        unless $n % $d {
            return 0;
        }
    }
    return 1;
}
```

svn://svn.shitov.ru/p6c

http://perl6.ru/p6c

svn://svn.shitov.ru/p6c

http://perl6.ru/p6c

__END__

Андрей Шитов | andy@shitov.ru