

Форма `set!` реализует оператор присваивания и выглядит следующим образом:

`(set! <идентификатор> <выражение>)`

Значение выражения замещает старое значение переменной, обозначенной идентификатором. Например, `(set! x 5)` присваивает переменной `x` новое значение 5.

Форма `set!` не возвращает значение. По аналогии с C++, можно считать, что это функция типа `void`. Обратите внимание, что в отличие от формы `(define x 5)`, форма `set!` не создает новую переменную, а только изменяет уже существующую.

В C++ эквивалент формы `set!` записывается с помощью оператора присваивания, например, `x=5`.

В файле `set.ss` записаны примеры применения формы `set!`, в файле `set.cpp` эквивалентная программа на C++.

```
; set
(define a 7)
(define(g x a)
  (set! a 5)
  (+ x a)
)
(g 1 0)
a
(set! a 8)
a
;_____
```

```
//set.cpp
#include "mlisp.h"
extern double a;
double g(double x, double a);

double a = 7.;
double g(double x, double a){
  a = 5.;
  return x + a;
}
int main(){
  display(g(1. , 0.));newline();
  display(a);newline();
  a = 8.;
  display(a);newline();
  std::cin.get();
}
```

```
    return 0;  
}
```

Форма `let` представлена в МИКРОЛИСПе двумя вариантами, существенно различающимися способом применения. Форму `let`, содержащую определения локальных переменных, будем называть блоком. Форму `let` без локальных переменных будем называть «легкий `let`».

Общая форма блока такова:

```
(let(  
    (<пер1> <выр1>)  
    (<пер2> <выр2>)  
    ...  
    (<перN> <вырN>)  
)  
  <тело>  
)
```

Первая часть блока представляет собой список пар вида имя-выражение. Когда блок вычисляется, для каждого имени создается локальная переменная, которая инициализируется значением соответствующего выражения. Область видимости локальных переменных распространяется только на тело блока. Инициализаторы в нее не входят.

Тело блока имеет такую же структуру, как и тело процедуры. Значение блока - это значение последнего выражения.

МИКРОЛИСП накладывает ограничения на контекст применения блоков:

- блок не может быть вложен в другой блок;
- блок можно записать только в теле процедуры;
- в теле процедуры может быть только один блок;
- блок должен быть последним выражением тела процедуры.

Область видимости локальных переменных вложена в область видимости параметров процедуры, а вместе с ней и в глобальную область видимости.

В файле `block.ss` записана программа, использующая блок, а в файле `block.cpp` эквивалентная программа на C++ .

```
; block  
(define(f x)
```

```

(set! x (- x (/ 100 101)))
(let(
  (a (sin x))
  (b (log x))
  (x 0)
  )
  (set! x (* 5 a b))
  x
)b )
(f 1)

```

```

//block.cpp
#include "mlisp.h
double f(double x);

double f(double x){
  x = x - 100./101.;
  {//let
    double
      a(sin(x)),
      b(log(x)),
      x(0.);
    f = 5. * a * b;
    return x;
  }//let
}
int main(){
  display(f(1.));newline();
  std::cin.get();
  return 0;
}

```

Взаимодействие областей видимости имен демонстрирует программа `scopes.ss` и ее эквивалент `scopes.cpp`.

```

;scopes
(define a 7)
(define(f f)
  (set! f (- f (/ 100 101)))
  (let(
    (a (sin f))
    (b (log f))
    (f 0)

```

```

    )
    (set! f (* 5 a b))
  f
)
)
(f 1)

```

```

//scopes.cpp
#include "mlisp.h"
extern double a;
double f(double f);

double a = 7.;
double f(double f){
  f = f - 100./101.;
  {//let
    double
      a(sin(f)),
      b(log(f)),
      f(0.);
    f = 5. * a * b;
    return f;
  }//let
}
int main(){
  display(f(1.));newline();
  std::cin.get();
  return 0;
}

```

Правильная структура программы на С++ будет обеспечена только, если эквивалент блока заключен в фигурные скобки. При отсутствии таких скобок параметр `f` и локальная переменная `f` окажутся в одной области видимости, компилятор С++ выдаст сообщение об ошибке. Эту ситуацию демонстрирует файл `nobraces.cpp`.

```

//nobraces.cpp
#include "mlisp.h"
extern double a;
double f(double f);

double a = 7.;
double f(double f){
  f = f - 100./101.;

```

```

//??? {//let
    double
        a(sin(f)),
        b(log(f)),

    f = 5. * a * b;
    return f;
//??? }//le t
}
int main(){
    display(f(1.));newline();
    std::cin.get();
    return 0;
}

```

Легкий let имеет форму:

(let())<тело>)

МИКРОЛИСП разрешает использовать легкий let в любом контексте, где может быть записано простое выражение и, в частности, числовая константа. Один из таких контекстов – это форма if. По правилам ЛИСПа следствие и альтернатива if должны быть простыми выражениями. Легкий let позволяет снять это ограничение.

В C++ легкий let переписывается с помощью оператора последовательности (оператор ,) .

В файле easylet.ss записана программа, использующая легкий let, а в файле easylet.cpp ее эквивалент на C++.

```

; easylet
(define a 7)
(define b 8)
"#1"
(+ (let()(display "a=")(display a)(newline)a)
   b)
;
;_____
"#2"
(if(< a b) a
    (let()(display "b=")(display b)(newline)b)
)

//easylet.cpp
#include "mlisp.h"
extern double a;
extern double b;

```

```

double a = 7.;
double b = 8.;

int main(){
  ##1
  display(
    (display("a="),display(a),newline(),a) +
    b
    );newline();
  ##2
  display(
    (a<b ? a
      : (display("b="),display(b),newline(),b)
    )
    );newline();

  std::cin.get();
  return 0;
}

```

Правила трансляции требуют заключать эквивалент легкого `let` в скобки. Файлы `nobrackets1.cpp` и `nobrackets2.cpp` демонстрируют печальные последствия нарушения этого требования.

```

// nobrackets 1.cpp
#include "mlisp.h"
extern double a;
extern double b;
double a = 7.;
double b = 8.;

int main(){
  ##1
  display(
    /*(*/display("a="),display(a),newline(),a/*)*/ +
    b
    );newline();

  std::cin.get();
  return 0;
}

```

```

// nobrackets 2.cpp
#include "mlisp.h"
extern double a;
extern double b;

```

```

double a = 7.;
double b = 8.;

int main(){
//#2
  display(
    (a<b ? a
      : /*(*display("b="),display(b),newline(),b/*)*/
    )
    );newline();

  std::cin.get();
  return 0;
}

```

Программа выдала неправильный результат потому, что приоритет оператора последовательности ниже приоритета условного оператора. Скобки устраняют конфликт приоритетов.

Аналогичный конфликт может возникнуть при трансляции формы cond с составной ветвью else.

Выражение

```

(cond((< a b) a)
      (else(display "b=")(display b)(newline)b)
)

```

следует транслировать так

```

(a<b ? a
  :(display("b="),display(b),newline(),b)
//      ^
)

```

Но в случае простой ветви else скобки не требуются.

```

(a<b ? a
  : b
)

```

Еще один вариант конфликта приоритетов устраняют скобки, обрамляющие эквивалент формы cond.

Выражение

```

(+ 1(cond((< e 0)2)(else 3)))

```

следует транслировать так

```

1. + ( e < 0. ? 2. : 3. )
//      ^

```

Если скобки убрать, то результат будет неправильный, поскольку приоритет оператора сложения выше приоритете условного оператора.

Список приоритетов операторов C++ можно найти в [4].