Lengyelforma

Imfix	Postfix	Prefix	✍: ....
			....

Kifejezés lengyelformára hozása	
(1+2)*(3-4) ⇒	

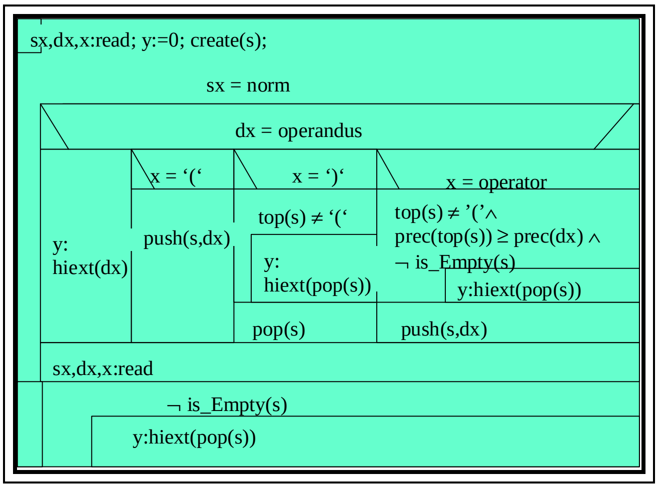
Lengyelformára hozás lépései

verem (s)	*eredmény* (y)	*bemenet* (x)	
		$\boxed{(}$ 1 + 2) * (3 - 4)	*ha nyitózárójel, „,,*
		($\boxed{1}$ + 2) * (3 - 4)	
		(1 $\boxed{+}$ 2) * (3 - 4)	
		(1 + $\boxed{2}$) * (3 - 4)	
		(1 + 2 $\boxed{)}$ * (3 - 4)	
		(1 + 2) $\boxed{*}$ (3 - 4)	
		(1 + 2) * $\boxed{(}$ 3 - 4)	
		(1 + 2) * ($\boxed{3}$ - 4)	
		(1 + 2) * (3 $\boxed{-}$ 4)	
		(1 + 2) * (3 - $\boxed{4}$)	
		(1 + 2) * (3 - 4 $\boxed{)}$	

Lengyelforma kiértékelése

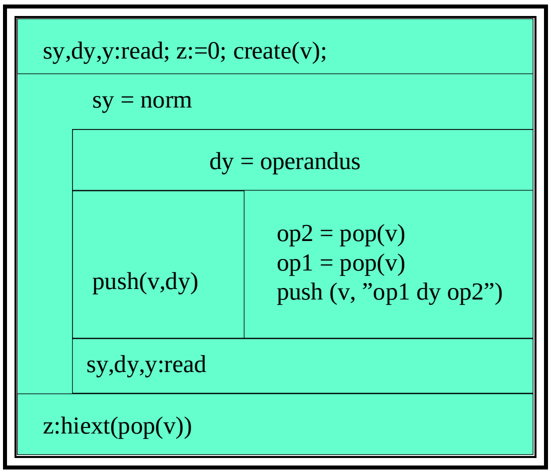
verem (v)	*eredmény* (z)	*bemenet* (y)	
		$\boxed{1}$ 2 + 3 4 - *	*ha a következő szimbólum operandus,, „,*
		1 $\boxed{2}$ + 3 4 - *	
		1 2 $\boxed{+}$ 3 4 - *	
		1 2 + $\boxed{3}$ 4 - *	
		1 2 + 3 $\boxed{4}$ - *	
		1 2 + 3 4 $\boxed{-}$ *	
		1 2 + 3 4 - $\boxed{*}$	
		1 2 + 3 4 - *	

Lengyelforma létrehozása



```
sx,dx,x:read; y:=0; create(s);
sx = norm
    dx = operandus
        y:
        hiext(dx)
    x = ‘(‘
        push(s,dx)
    x = ‘)‘
        top(s) ≠ ‘(‘
            y:
            hiext(pop(s))
        pop(s)
    x = operator
        top(s) ≠ ’(’∧
        prec(top(s)) ≥ prec(dx) ∧
        ¬ is_Empty(s)
            y:hiext(pop(s))
        push(s,dx)
    sx,dx,x:read
¬ is_Empty(s)
    y:hiext(pop(s))
```

Lengyelforma kiértékelése



```
sy,dy,y:read; z:=0; create(v);
sy = norm
    dy = operandus
        push(v,dy)
        op2 = pop(v)
        op1 = pop(v)
        push (v, ”op1 dy op2”)
    sy,dy,y:read
z:hiext(pop(v))
```

Implementáció - I. ☹

x:	elemek sorozata
y:	elemek sorozata
s:	elemeket tároló verem (template)
v:	integereket tároló verem (template)

✎

✍: ……………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………

Implementáció - II. ☺

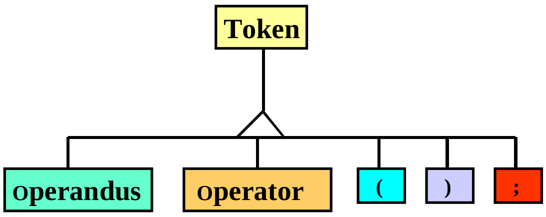
x:	elemekre mutató pointerek sorozata
y:	elemekre mutató pointerek sorozata
s:	elemeket tároló verem (template)
v:	integereket tároló verem (template)

✎

✍: ……………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………

Token osztály

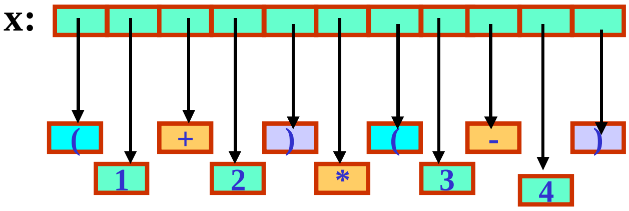
✎



✍: ……………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………

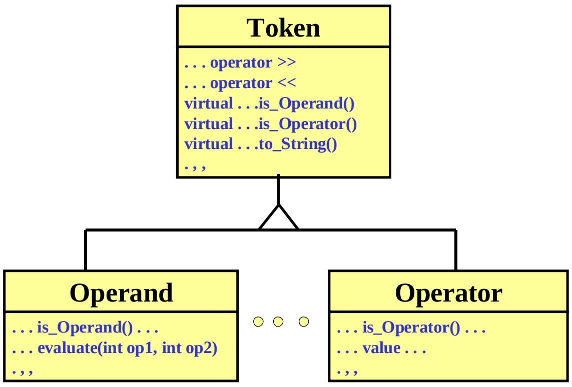
x:=sequence(Token*)

✎



✍: ……………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………

Származtatás, virtuális függvények



✍: ……………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………
……………………………………………

Token.h

```
class Token {
   friend  ostream& operator<<(ostream&, Token*&);
   friend  istream& operator>>(istream&, Token*&);
public:
   virtual ~Token(void);
   virtual bool is_LEFTP() {return false;};
   virtual bool is_RIGHTP() {return false;};
   virtual bool is_Operand() {return false;};
   virtual bool is_Operator() {return false;};
   virtual bool is_END() {return false;};
   virtual int priority() {return 0;};
   virtual string to_String() {return "";};
   virtual string class_Name() {return "Token:";};
protected:
   Token();
};
```

✍: ....

....

....

....

....

....

....

....

....

....

....

....

....

Operand.h

```
class Operand: public Token {
public:
   Operand(int v) {val=v;};
   int value() {return val;};
   bool is_Operand() {return true; };
   string to_String() ;
   string class_Name() {return "Operand: "; };
protected:
   int val;
};
```

✍: ....

....

....

....

....

....

....

....

Operator.h

```
class Operator: public Token {
public:
   Operator(char o) {op=o;};
   bool is_Operator() {return true; };
   int priority() ;
   int  evaluate(const int,const int);□
   string to_String() ;
   string class_Name() {return "Operator: "; };
protected:
   char op;
};
```

✍: ....

....

....

....

....

....

....

....

....

LEFTP.h *(RIGHTP.hasonlóan elkészíthető)*

```
class LEFTP: public Token {
public:
   bool is_LEFTP() {return true; };
   string to_String() {return "("; };
   string class_Name()  return "Left parentheses:"; };
};
```

✍: ....

....

....

....

....

END.h

```
class END: public Token {
public:
   bool is_END() {return true;};
   string to_String() {return ";"; };
   string class_Name() {return "End of expression:"; };
};
```

✍: ....

....

....

....

....

Sequence.h *(Lásd előző előadások anyaga)*

```
Sequence<Token*> x;
```

Stack.h *(Lásd előző előadások anyaga)*

```
Stack<Token*> s; Stack<int> v;
```

Implementációk

Operand.cpp	

#include <string>
using namespace std;
#include "Token.h"
#include "Operand.h"
#include "Stack.h"
	✍: ....
	....
	....

string Operand::to_String() {
string digit[10]={"0","1","2","3","4","5","6","7","8","9"};
    string s;
    Stack<string> v;
    if(val==0)   {
        v.push("0");
    }
    for(int i=val;i!=0;i=i/10){
        v.push(digit[i%10]);
    }
    for(;!v.empty();s=s+v.pop()) {}
    return s;
};
	✍: ....
	....
	....
	....
	....
	....
	....
	....
	....

Operator.cpp	

#include<iostream>
#include <string>
using namespace std;
#include "Token.h"
#include "Operator.h"
	✍: ....
	....
	....

string Operator::to_String() {
    string ret;
    ret=op;
    return ret;
};
	✍: ....
	....
	....

int Operator::priority() {
    switch(op) {
        case('+'):case('-'):
            return 1;
        case('*'): case('/'):
            return 2;
        default:
            return 3;
    }
};
	✍: ....
	....
	....
	....
	....
	....
	....

int Operator::evaluate(const int a,const int b) {
    switch(op) {
    case('+'):
        return(a+b);
    case('-'):
        return(a-b);
    case('*'):
        return(a*b);
    case('/'):
        return(a/b);
    default:
        exit(1);   //Baj van!
    }
}
	✍: ....
	....
	....
	....
	....
	....
	....
	....
	....

Token.cpp

```
#include<iostream>
#include <string>
using namespace std;
#include "Sequence.h"
#include "Token.h"
#include "LEFTP.h"
#include "RIGHTP.h"
#include "END.h"
#include "Operand.h"
#include "Operator.h"
```

✍: ....
....
....
....
....
....
....

```
ostream& operator <<(ostream& s, Token*& t) {
   s << t->to_String();  ;
   return s;
}
```

✍: ....
....

```
istream& operator >>(istream& s, Token*& t) {
   char ch;
   int intval;
   s >> ch;
   switch(ch) {
         case ('+'): case ('-'): case ('*'): case ('/'):
            t=new Operator(ch);
            break;
         case ('('):
            t=new LEFTP();
            break;
         case (')'):
            t=new RIGHTP();
            break;
         case (';'):
            t=new END();
            break;
         case ('0'):
          , . .
         case ('9'):
            s.putback(ch);
            s >> intval;
            t=new Operand(intval);
         break;
         default:
            cout << "Illegal element: " << ch << endl;
            s >> ch;
   }
  return s;
}
```

✍: ....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....

Lengyel.cpp

```
#include <string>
using namespace std;
#include "Token.h"
#include "LEFTP.h"
#include "RIGHTP.h"
#include "END.h"
#include "Operand.h"
#include "Operator.h"
#include "Stack.h"
#include "Sequence.h"
Token* next_Token();[1]
int main(){
```

✍: ....
....
....
....
....
....
....
....
....

[1] Alternatív megoldás a "**Token**"-ek beolvasására

//A szintaktikusan helyes infix kifejezés beolvasása

```
    Sequence<Token*> x;
    Sequence<Token*> y;
    Stack<Token*> s;
    Token* t;
    while(!t->is_END()){
        x.hiext(t);
        t=next_Token();
    }
    x.print();
```

: ....

....

....

....

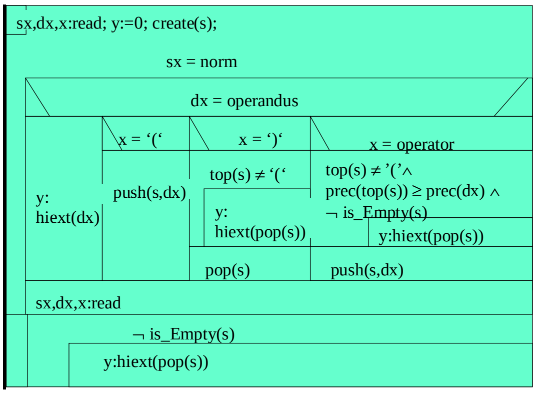
....

....

....

//A kifejezés átalakítása lengyelformára

```
    x.first();
    while(!x.eol()){
        t=x.current();
        if(t->is_Operand()) {
            y.hiext(t);
        } else if (t->is_LEFTP()) {
            s.push(t);
        } else if(t->is_RIGHTP()) {
            while(!s.top()->is_LEFTP()) {
                y.hiext(s.pop());
            }
            s.pop();
        } else if (t->is_Operator()) {
            while(!s.empty() &&
                  s.top()->priority() >=  s.top()->priority() &&
                  !s.top()->is_LEFTP()) {
                  y.hiext(s.pop());
            }
            s.push(t);
        } else {
            cout << "Szintaktikai hiba?" << endl;
        }
        x.next();
    }
    for(;!s.empty();y.hiext(s.pop())){}
    y.print();
```



: ....

....

....

....

....

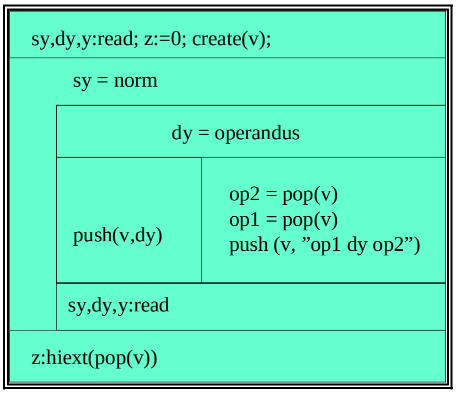
....

....

....

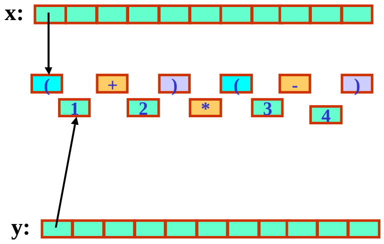
//A lengyelforma kiértékelése

```
    Stack<int> v;
    y.first();
    while(!y.eol()) {
        t=y.current();
        if (t->is_Operand()) {
             v.push((dynamic_cast<Operand*>(t))->value());
        } else {
            v.push(dynamic_cast<Operator*>(t)-
>evaluate(v.pop(),v.pop()));
        }
        y.next();
    }
    cout << "A kifejezes erteke: " << v.pop() << endl;
```



//A dinamikusan lefoglalt tárterület felszabadítása

```
    for( x.first();!x.eol(); x.next())
    {
    delete x.current();
    }
    char barmi;
    cin >> barmi;
    return 0;
}
```



Típuskényszerítés		✍: ....
upcast	**downcast**	
✎	✎	
v.push((dynamic_cast<Operand*>(t))->value());		✍: ....
v.push(dynamic_cast<Operator*>(t)-		✍: ....

C++ programok

Token.h

```
#ifndef TOKEN_H
#define TOKEN_H
#include<iostream>
#include <string>
class Token {
    friend  ostream& operator<<(ostream&, Token*&);
    friend  istream& operator>>(istream&, Token*&);

public:
    virtual ~Token(void);
    virtual bool is_LEFTP() {return false;};
    virtual bool is_RIGHTP() {return false;};
    virtual bool is_Operand() {return false;};
    virtual bool is_Operator() {return false;};
    virtual bool is_END() {return false;};
    virtual int priority() {return 0;};
    virtual string to_String() {return "";};
    virtual string class_Name() {return "Token:";};
protected:
    Token();
};
#endif
```

Token.cpp

```
#include<iostream>
#include <string>
using namespace std;
#include "Sequence.h"
#include "Token.h"
#include "LEFTP.h"
#include "RIGHTP.h"
#include "END.h"
#include "Operand.h"
#include "Operator.h"
ostream& operator <<(ostream& s, Token*& t) {
   s << t->to_String();  ;
   return s;
}
istream& operator >>(istream& s, Token*& t) {
   char ch;
   int intval;
   s >> ch;
   switch(ch)
   {
         case ('+'):
         case ('-'):
         case ('*'):
         case ('/'):
            t=new Operator(ch);
            break;
         case ('('):
            t=new LEFTP();
            break;
         case (')'):
            t=new RIGHTP();
            break;
         case (';'):
            t=new END();
            break;
         case ('0'):
         case ('1'):
         case ('2'):
         case ('3'):
         case ('4'):
         case ('5'):
         case ('6'):
         case ('7'):
         case ('8'):
         case ('9'):
            s.putback(ch);
            s >> intval;
            t=new Operand(intval);
         break;
         default:
            cout << "Illegal element: " << ch << endl;
            s >> ch;
   }
  return s;
}
```

Operand.h

```
#ifndef OPERAND_H
#define OPERAND_H
#include <string>
using namespace std;
#include "Token.h"

class Operand: public Token {
public:
    Operand(int v) {val=v;};
    int value() {return val;};
    bool is_Operand() {return true; };
    string to_String();
    string class_Name() {return "Operand: "; };
protected:
    int val;
};
#endif
```

Operand.cpp

```
#include <string>
using namespace std;

#include "Token.h"
#include "Operand.h"
#include "Stack.h"

string Operand::to_String() {
    string digit[10]={"0","1","2","3","4","5","6","7","8","9"};
    string s;
    Stack<string> v;
    if(val==0) {
        v.push("0");
    }
    for(int i=val;i!=0;i=i/10){
        v.push(digit[i%10]);
    }
    for(;!v.empty();s=s+v.pop()) {}
    return s;
};
```

Operator.h

```
#ifndef OPERATOR_H
#define OPERATOR_H
#include <string>
using namespace std;
#include "Token.h"

class Operator: public Token {
public:
    Operator(char o) {op=o;};
    bool is_Operator() {return true; };
    int priority();
    int evaluate(const int,const int);
    string to_String();
    string class_Name() {return "Operator: "; };
protected:
    char op;
};
#endif
```

Operator.cpp

```
#include<iostream>
#include <string>
using namespace std;
#include "Token.h"
#include "Operator.h"

string Operator::to_String() {
    string ret;
    ret=op;
    return ret;
};
int Operator::priority() {
    switch(op) {
        case('+'):
        case('-'):
            return 1;
        case('*'):
        case('/'):
            return 2;
        default:
            return 3;
    }
};
int Operator::evaluate(const int a,const int b) {
    switch(op) {
    case('+'):
        return(a+b);
    case('-'):
        return(a-b);
    case('*'):
        return(a*b);
    case('/'):
        return(a/b);
    default:
        exit(1)    // Baj van!
    }
}
```

LEFTP.h

```
#ifndef LEFTP_H
#define LEFTP_H
#include <string>
using namespace std;

class LEFTP: public Token {
public:
    bool is_LEFTP() {return true; };
    string to_String() {return "("; };
    string class_Name() {return "Left parentheses:"; };
};
#endif
```

RIGHTP.h

```
#ifndef RIGHTP_H
#define RIGHTP_H
#include <string>
using namespace std;
#include "Token.h"

class RIGHTP: public Token {
public:
    bool is_RIGHTP() {return true; };
    string to_String() {return ")"; };
    string class_Name() {return "Right parentheses:"; };
};
#endif
```

END.h

```
#ifndef END_H
#define END_H
#include <string>
using namespace std;
#include "Token.h"

class END: public Token {
public:
    bool is_END() {return true;};
    string to_String() {return ";"; };
    string class_Name() {return "End of expression:"; };
};
#endif
```

Sequence.h

```
#ifndef SEQUENCE_H
#define SEQUENCE_H
template <class Element>
class Sequence{
public:
    enum Exceptions{EMPTYSEQUENCE};
    Sequence():_first(0),_last(0),_current(0) {};
    ~Sequence();
    void hiext(const Element&);
    Element lopop();
    void first() {_current = _first;}
    void next() {_current = _current->next;}
    Element current() {return _current->val;}
    bool eol() {return (_current == 0);}
    bool empty() {return (_first == 0);}
    void print();
private:
    Sequence(const Sequence<Element>&);
    Sequence& operator=(const Sequence<Element>&);
    struct Node{
        Element val;
        Node *next;
        Node *prev;
        Node(const Element &v, Node *n, Node *p):val(v),next(n), prev(p) {}
    };
    Node *_first;
    Node *_last;
    Node *_current;
};
template <class Element>
Sequence<Element>::~Sequence() {
    Node *p;
    while(_first!=0){
        p=_first;
        _first=_first->next;
        delete p;
    }
}
template <class Element>
void Sequence<Element>::hiext(const Element &e){
    Node *p=new Node(e,0,_last);
    if(_last) {
        _last->next=p;
    }
    _last=p;
    if(_first==0) {
        _first=p;
    }
}
template <class Element>
Element Sequence<Element>::lopop() {
    if(_first==0) throw EMPTYSEQUENCE;
    Element retval=_first->val;
    Node *p=_first;
    _first=_first->next;
    delete p;
    if(_first) {
        _first->prev=0;
    }else{
        _last=0;
    }
    return retval;
}
template <class Element>
void Sequence<Element>::print() {
    Element t;
    first();
    while(!eol()) {
        t=current();
        cout << t << " ";
        next();
    }
    cout << endl;
}
#endif
```

Poland.cpp

```
#include <string>
using namespace std;
#include "Token.h"
#include "LEFTP.h"
#include "RIGHTP.h"
#include "END.h"
#include "Operand.h"
#include "Operator.h"
#include "Stack.h"
#include "Sequence.h"

Token* next_Token();

int main() {

//Kifejezés beolvasása
Sequence<Token*> x;
    Token* t;
    t=next_Token();
    while(!t->is_END()){
        x.hiext(t);
        t=next_Token();
    }
    x.print();

//Kifejezés lengyelformára hozása
Stack<Token*> s;
Sequence<Token*> y;
    x.first();
    while(!x.eol()){
        t=x.current();
        if(t->is_Operand()) {
            y.hiext(t);
        } else if (t->is_LEFTP()) {
            s.push(t);
        } else if(t->is_RIGHTP()) {
            while(!s.top()->is_LEFTP()) {
                y.hiext(s.pop());
            }
            s.pop();
        } else if (t->is_Operator()) {
            while(!s.empty() &&
                  s.top()->priority() >= s.top()->priority() &&
                  !s.top()->is_LEFTP()) {
                  y.hiext(s.pop());
            }
            s.push(t);
        } else {
            cout << "Szintaktikai hiba?" << endl;
        }
        x.next();
    }
    for(;!s.empty();y.hiext(s.pop())){}
    y.print();

    while(!s.empty()) {
        y.hiext(s.pop());
    }
    y.print();
//Lengyelforma kiértékelése
    Stack<int> v;
    y.first();
    while(!y.eol()) {
        t=y.current();
        if (t->is_Operand()) {
            v.push((dynamic_cast<Operand*>(t))->value());
        } else {
            v.push(dynamic_cast<Operator*>(t)->evaluate(v.pop(),v.pop()));
        }
        y.next();
    }
    cout << "A kifejezes erteke: " << v.pop() << endl;

//Tárterület felszabadítása
    x.first();
    while(!x.eol()) {
        delete x.current();
        x.next();
    }
    char barmi;
    cin >> barmi;
    return 0;
}

Token* next_Token()
{
    char ch;
    int intval;
    cin >> ch;
    Token* t;
    switch(ch)
    {
            case ('+'): case ('-'):case ('*'):case ('/'):
                t=new Operator(ch);
                break;
            case ('('):
                t=new LEFTP();
                break;
            case (')'):
                t=new RIGHTP();
                break;
            case (';'):
                t=new END();
                break;
            case ('0'):
            case ('1'):
           . . .
            case ('8'):
            case ('9'):
                cin.putback(ch);
                cin >> intval;
                t=new Operand(intval);
            break;
            default:
                cout << "Illegal element: " << ch << endl;
                cin >> ch;
    }
   return t;
}
```