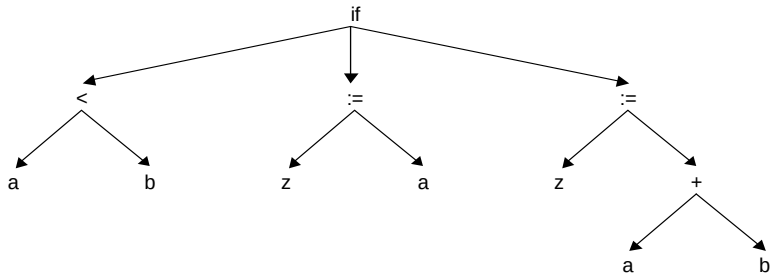
Překlad do syntaktického stromu

Příklad stromu reprezentujícího příkaz

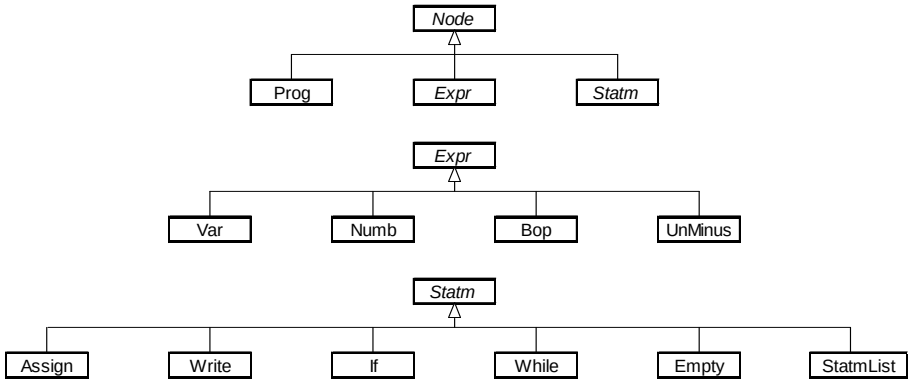
```
if a<b then z:=a else z:=a+b
```



Přehled uzlů

konstrukce jazyka	typ uzlu	položky uzlu
identifikátor proměnné	Var	adresa proměnné příznak, zda jde o proměnnou ve výrazu
číslo, identifikátor konstanty	Numb	hodnota čísla
výraz1 operátor výraz2	Bop	operátor ukazatel na výraz1 ukazatel na výraz2
- výraz	UnMinus	ukazatel na výraz
proměnná := výraz	Assign	ukazatel na proměnnou ukazatel na výraz
write výraz	Write	ukazatel na výraz
if podm then příkaz1 else příkaz2	If	ukazatel na podmínku ukazatel na příkaz1 ukazatel na příkaz2 (nebo *null*, chybí-li část else)
while podmínka do příkaz	While	ukazatel na podmínku ukazatel na příkaz
prázdný příkaz	Empty	
seznam příkazů	StmList	ukazatel na příkaz ukazatel na zbytek seznamu (nebo *null*, je-li zbytek prázdný)
program	Prog	ukazatel na seznam příkazů

Hierarchie tříd uzlů



Deklarace tříd v C++

```
class Node {
public:
   virtual Node *Optimize() {return this;}
   virtual void Translate() = 0;
   virtual ~Node() {}
};

class Expr : public Node {
};

class Statm : public Node {
};

class Var : public Expr {
   int addr;
   bool rvalue;
public:
   Var(int, bool);
   virtual void Translate();
};
```

```
class Numb : public Expr {
   int value;
public:
   Numb(int);
   virtual void Translate();
   int Value();
};

class Bop : public Expr {
   Operator op;
   Expr *left, *right;
public:
   Bop(Operator, Expr*, Expr*);
   virtual ~Bop();
   virtual Node *Optimize();
   virtual void Translate();
};

class UnMinus : public Expr {
   Expr *expr;
public:
```



```
   UnMinus(Expr *e);
   virtual ~UnMinus();
   virtual Node *Optimize();
   virtual void Translate();
};

class Assign : public Statm {
   Var *var;
   Expr *expr;
public:
   Assign(Var*, Expr*);
   virtual ~Assign();
   virtual Node *Optimize();
   virtual void Translate();
};

class Write : public Statm {
   Expr *expr;
public:
   Write(Expr*);
   virtual ~Write();
   virtual Node *Optimize();
```



```
   virtual void Translate();
};

class  If : public Statm {
   Expr *cond;
   Statm *thenstm;
   Statm *elsestm;
public:
   If(Expr*, Statm*, Statm*);
   virtual ~If();
   virtual Node *Optimize();
   virtual void Translate();
};

class While : public Statm {
   Expr *cond;
   Statm *body;
public:
   While(Expr*, Statm*);
   virtual ~While();
   virtual Node *Optimize();
   virtual void Translate();
```



```
};

class  StatmList : public Statm {
   Statm *statm;
   StatmList *next;
public:
   StatmList(Statm*, StatmList*);
   virtual ~StatmList();
   virtual Node *Optimize();
   virtual  void Translate();
};

class Empty : public Statm {
   virtual void Translate() {}
};

class Prog : public Node {
   StatmList *stm;
public:
   Prog(StatmList*);
   virtual ~Prog();
   virtual Node *Optimize();
```

```
   virtual void Translate();
};
```


Atributová překladová gramatika

Přidružení atributů k symbolům

druh symbolu	symbol	dědičné atributy	syntetiz. atributy
neterminální symbol	Program, SložPříkaz, ZbPříkazů, Příkaz, Podmínka, Výraz, Term, Faktor		su
	ZbVýrazu, ZbTermu	du	su
	Dekl, DeklKonst, DeklProm, ZbDeklKonst, ZbDeklProm		
	RelOp		sop
vstupní symbol	ident		sid
	číslo		shod
speciální výstupní symbol	*deklKonst* *deklProm*	did, dhod did	

Atributová překladová gramatika

Překlad deklarací
Stejný, jako při překladu do jazyka zásobníkového počítače

Překlad jednoduchých příkazů

14:	Příkaz → ident := Výraz	Příkaz.su := *new Assign*(*adrVar*(Příkaz.dts,ident.sid), Výraz.su)
15:	Příkaz → write Výraz	Příkaz.su := *new Write*(Výraz.su)
19:	Příkaz → ε	Příkaz.su := *new Empty*

Překlad složeného příkazu

11:	SložPříkaz → begin Příkaz ZbPříkazů end	SložPříkaz.su := *new StmList*(Příkaz.su,ZbPříkazů.su)
12:	ZbPříkazů → ; Příkaz ZbPříkazů	ZbPříkazů^0.su := *new StmList*(Příkaz.su,ZbPříkazů^1.su)
13:	ZbPříkazů → ε	ZbPříkazů.su := *null*
18:	Příkaz → SložPříkaz	Příkaz.su := SložPříkaz.su

Atributová překladová gramatika

Překlad příkazu cyklu

17:	Příkaz → while Podmínka do Příkaz	Příkaz^0.su := *new While*(Podmínka.su, Příkaz^1.su)

Překlad podmíněného příkazu

16:	Příkaz^0 → if Podmínka then Příkaz^1 ČástElse	Příkaz^0.su := *new If*(Podmínka.su, Příkaz^1.su, ČástElse.su)
20:	ČástElse → else Příkaz	ČástElse.su := Příkaz.su
21:	ČástElse → ε	ČástElse.su := *null*

Atributová překladová gramatika

Překlad výrazů

22:	Podmínka → Výraz[1] RelOp Výraz[2]	Podmínka.su := *new Bop*(RelOp.sop,Výraz[1].su,Výraz[2].su)
23:	RelOp → =	RelOp.sop := *Eq*
...	...	
29:	Výraz → Term ZbVýrazu	ZbVýrazu.du := Term.su Výraz.su := ZbVýrazu.su
30:	Výraz → - Term ZbVýrazu	ZbVýrazu.du := *new UnMinus*(Term.su) Výraz.su := ZbVýrazu.su
31:	ZbVýrazu → + Term ZbVýrazu	ZbVýrazu[1].du := *new Bop*(*Plus*, ZbVýrazu[0].du,Term.su) ZbVýrazu[0].su := ZbVýrazu[1].su
32:	ZbVýrazu → - Term ZbVýrazu	ZbVýrazu[1].du := *new Bop*(*Minus*, ZbVýrazu[0].du,Term.su) ZbVýrazu[0].su := ZbVýrazu[1].su
33:	ZbVýrazu → ε	ZbVýrazu.su := ZbVýrazu.du

Atributová překladová gramatika

Překlad výrazů

34:	Term → Faktor ZbTermu	ZbTermu.du := Faktor.su Výraz.su := ZbTermu.su
35:	ZbTermu → * Faktor ZbTermu	ZbTermu[1].du := *new Bop*(*Times*, ZbTermu[0].du,Faktor.su) ZbTermu[0].su := ZbTermu[1].su
36:	ZbTermu → / Faktor ZbTermu	ZbTermu[1].du := *new Bop*(*Divide*, ZbTermu[0].du,Faktor.su) ZbTermu[0].su := ZbTermu[1].su
37:	ZbTermu → ε	ZbTermu.su := ZbTermu.du
38:	Faktor → ident	Faktor.su := *VarOrConst*(ident.sid)
39:	Faktor → číslo	Faktor.su := *new Numb*(číslo.shod)
40:	Faktor → (Výraz)	Faktor.su := Výraz.su

Atributová překladová gramatika

Překlad výrazů

Pomocná funkce:

```
Expr *VarOrConst(char *id)
{
   int v;
   DruhId druh = idPromKonst(id,&v);
   switch (druh) {
   case IdProm:
      return new Var(v, true);
   case IdKonst:
      return new Numb(v);
   }
}
```


Rekurzívní sestup

```
Prog *Program()
{
   Dekl();
   return new Prog(SlozPrikaz());
}

void Dekl()
{
   switch (Symb) {
   case kwVAR:
      DeklProm();
      Dekl();
      break;
   case kwCONST:
      DeklKonst();
      Dekl();
      break;
   default:
      ;
   }
```

```
}

void DeklKonst()
{
   char id[MaxLenIdent];
   int hod;
   CtiSymb();
   Srovnani_IDENT(id);
   Srovnani(EQ);
   Srovnani_NUMB(&hod);
   deklKonst(id, hod);
   ZbDeklKonst();
   Srovnani(SEMICOLON);
}

void ZbDeklKonst()
{
   if (Symb == COMMA) {
      char id[MaxLenIdent];
      int hod;
      CtiSymb();
      Srovnani_IDENT(id);
```



```
      Srovnani(EQ);
      Srovnani_NUMB(&hod);
      deklKonst(id, hod);
      ZbDeklKonst();
   }
}

void DeklProm()
{
   char id[MaxLenIdent];
   CtiSymb();
   Srovnani_IDENT(id);
   deklProm(id);
   ZbDeklProm();
   Srovnani(SEMICOLON);
}

void ZbDeklProm()
{
   if (Symb == COMMA) {
      char id[MaxLenIdent];
      CtiSymb();
```



```
      Srovnani_IDENT(id);
      deklProm(id);
      ZbDeklProm();
   }
}

StatmList *SlozPrikaz()
{
   Srovnani(kwBEGIN);
   Statm *p = Prikaz();
   StatmList *su = new StatmList(p, ZbPrikazu());
   Srovnani(kwEND);
   return su;
}

StatmList *ZbPrikazu()
{
   if (Symb == SEMICOLON) {
      CtiSymb();
      Statm *p = Prikaz();
      return new StatmList(p, ZbPrikazu());
   }
```



```
   return 0;
}

Statm *Prikaz()
{
   switch (Symb) {
   case IDENT: {
      Var *var = new Var(adrProm(Ident),false);
      CtiSymb();
      Srovnani(ASSGN);
      return new Assign(var, Vyraz());
   }
   case kwWRITE:
      CtiSymb();
      return new Write(Vyraz());
   case kwIF: {
      CtiSymb();
      Expr *cond = Podminka();
      Srovnani(kwTHEN);
      Statm *prikaz = Prikaz();
      return new If(cond, prikaz, CastElse());
   }
```

```
   case kwWHILE: {
      Expr *cond;
      CtiSymb();
      cond = Podminka();
      Srovnani(kwDO);
      return new While(cond, Prikaz());
   }
   case kwBEGIN:
      return SlozPrikaz();
   default:
      return new Empty;
   }
}

Statm *CastElse()
{
   if (Symb == kwELSE) {
      CtiSymb();
      return Prikaz();
   }
   return 0;
}
```



```
Expr *Podminka()
{
   Expr *left = Vyraz();
   Operator op = RelOp();
   Expr *right = Vyraz();
   return new Bop(op, left, right);
}

Operator RelOp()
{
   switch (Symb) {
   case EQ:
      CtiSymb();
      return Eq;
   case NEQ:
      CtiSymb();
      return NotEq;
   case LT:
      CtiSymb();
      return Less;
   case GT:
```



```
      CtiSymb();
      return Greater;
   case LTE:
      CtiSymb();
      return LessOrEq;
   case GTE:
      CtiSymb();
      return GreaterOrEq;
   default:
      Chyba("neocekavany symbol");
   }
}

Expr *Vyraz()
{
   if (Symb == MINUS) {
      CtiSymb();
      return ZbVyrazu(new UnMinus(Term()));
   }
   return ZbVyrazu(Term());
}
```



```
Expr *ZbVyrazu(Expr *du)
{
   switch (Symb) {
   case PLUS:
      CtiSymb();
      return ZbVyrazu(new Bop(Plus, du, Term()));
   case MINUS:
      CtiSymb();
      return ZbVyrazu(new Bop(Minus, du, Term()));
   default:
      return du;
   }
}

Expr *Term()
{
   return ZbTermu(Faktor());
}

Expr *ZbTermu(Expr *du)
{
   switch (Symb) {
```

```
   case TIMES:
      CtiSymb();
      return ZbTermu(new Bop(Times, du, Faktor()));
   case DIVIDE:
      CtiSymb();
      return ZbVyrazu(new Bop(Divide, du, Faktor()));
   default:
      return du;
   }
}

Expr *Faktor()
{
   switch (Symb) {
   case IDENT:
      char id[MaxLenIdent];
      Srovnani_IDENT(id);
      return VarOrConst(id);
   case NUMB:
      int hodn;
      Srovnani_NUMB(&hodn);
      return new Numb(hodn);
```



```
   case LPAR: {
      CtiSymb();
      Expr *su = Vyraz();
      Srovnani(RPAR);
      return su;
   }
   default:
      Chyba("Neocekavany symbol");
   }
}
```


Optimalizace syntaktického stromu

Syntaktický strom vytvořený funkcemi rekurzivního sestupu budeme optimalizovat těmito úpravami:

- Uzel binární operace *Bop* specifikující operaci *op*, jehož oba operandy jsou konstanty *Numb*(n_1) a *Numb*(n_2), nahradíme uzlem konstanty *Numb*(*n*), kde *n* je výsledek operace n_1 *op* n_2.
- Uzel unární operace *UnMinus*, jehož operandem je konstanta *Numb*(*n*), nahradíme uzlem konstanty *Numb*(-*n*).
- Uzel podmíněného příkazu *If* obsahující ukazatel na konstantní podmínku *Numb*(1) nahradíme uzlem, který je kořenem podstromu reprezentujícího příkaz z části *then*.
- Uzel podmíněného příkazu *If* obsahující ukazatel na konstantní podmínku *Numb*(0) nahradíme uzlem, který je kořenem podstromu reprezentujícího příkaz z části *else*.
- Uzel příkazu cyklu *While* obsahující ukazatel na konstantní podmínku *Numb*(0) nahradíme uzlem, který je kořenem podstromu reprezentujícího příkaz tvořící tělo cyklu.

Optimalizace syntaktického stromu

```
Node *Bop::Optimize()
{
   Numb *l = dynamic_cast<Numb*>(left->Optimize());
   Numb *r = dynamic_cast<Numb*>(right->Optimize());
   if (!l || !r) return this;
   int res;
   int leftval = l->Value();
   int rightval = r->Value();
   switch (op) {
   case Plus:
      res = leftval + rightval;
      break;
   case Minus:
      res = leftval - rightval;
      break;
   case Times:
      res = leftval * rightval;
      break;
   case Divide:
      res = leftval / rightval;
```

```
      break;
   case Eq:
      res = leftval == rightval;
      break;
   case NotEq:
      res = leftval != rightval;
      break;
   case Less:
      res = leftval < rightval;
      break;
   case Greater:
      res = leftval > rightval;
      break;
   case LessOrEq:
      res = leftval <= rightval;
      break;
   case GreaterOrEq:
      res = leftval >= rightval;
      break;
   }
   delete this;
   return new Numb(res);
```



```
}

Node *UnMinus::Optimize()
{
   expr->Optimize();
   Numb *e = dynamic_cast<Numb*>(expr);
   if (!e) return this;
   e = new Numb(-e->Value());
   delete this;
   return e;
}

Node *Assign::Optimize()
{
   expr = (Var*)(expr->Optimize());
   return this;
}

Node *Write::Optimize()
{
   expr = (Expr*)(expr->Optimize());
   return this;
```



```
}

Node *If::Optimize()
{
   cond = (Expr*)(cond->Optimize());
   thenstm = (Statm*)(thenstm->Optimize());
   elsestm = (Statm*)(elsestm->Optimize());
   Numb *c = dynamic_cast<Numb*>(cond);
   if (!c) return this;
   Node *res;
   if (c->Value()) {
      res = thenstm; thenstm = 0;
   } else {
      res = elsestm; elsestm = 0;
   }
   delete this;
   return res;
}

Node *While::Optimize()
{
   cond = (Expr*)(cond->Optimize());
```



```
   body = (Statm*)(body->Optimize());
   Numb *c = dynamic_cast<Numb*>(cond);
   if (!c) return this;
   if (!c->Value()) {
      delete this;
      return new Empty;
   }
   return this;
}

Node *StatmList::Optimize()
{
   StatmList *s = this;
   do {
      s->statm = (Statm*)(s->statm->Optimize());
      s = s->next;
   }
   while (s);
   return this;
}

Node *Prog::Optimize()
```

```
{
   stm = (StatmList*)(stm->Optimize());
   return this;
}
```


Překlad stromu do jazyka zásobníkového počítače

```
void Var::Translate()
{
   Gener(TA, addr);
   if (rvalue)
      Gener(DR);
}

void Numb::Translate()
{
   Gener(TC, value);
}

void Bop::Translate()
{
   left->Translate();
   right->Translate();
   Gener(BOP, op);
}

void UnMinus::Translate()
```



```
{
   expr->Translate();
   Gener(UNM);
}

void Assign::Translate()
{
   var->Translate();
   expr->Translate();
   Gener(ST);
}

void Write::Translate()
{
   expr->Translate();
   Gener(WRT);
}

void If::Translate()
{
   cond->Translate();
   int a1 = Gener(IFJ);
```



```
   thenstm->Translate();
   if (elsestm) {
      int a2 = Gener(JU);
      PutIC(a1);
      elsestm->Translate();
      PutIC(a2);
   } else
      PutIC(a1);
}

void While::Translate()
{
   int a1 = GetIC();
   cond->Translate();
   int a2 = Gener(IFJ);
   body->Translate();
   Gener(JU, a1);
   PutIC(a2);
}

void StatmList::Translate()
{
```

```
   StatmList *s = this;
   do {
      s->statm->Translate();
      s = s->next;
   } while (s);
}

void Prog::Translate()
{
   stm->Translate();
   Gener(STOP);
}
```