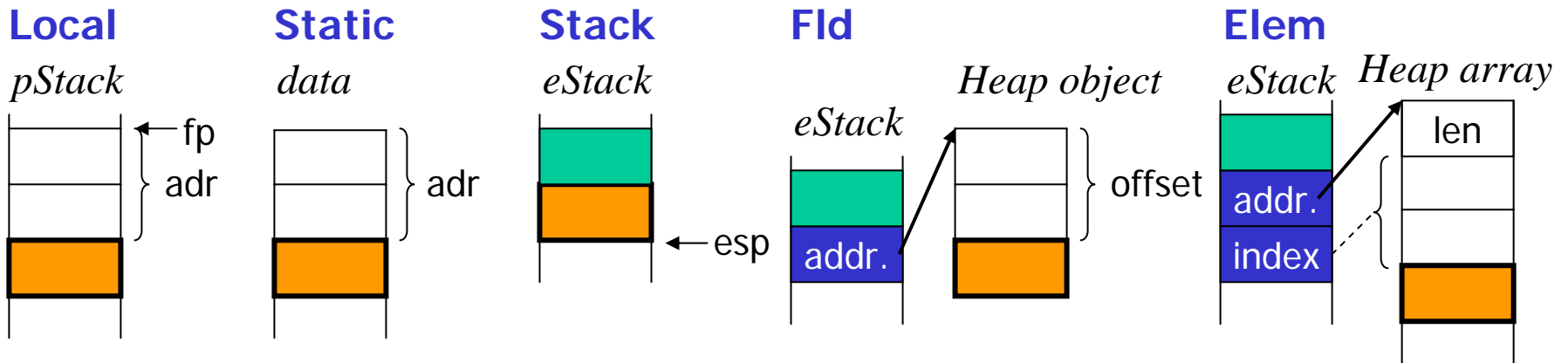


# Klasse *Item*



```
class Item {  
    public static final int // Mögliche Item-Arten  
        Con=0, Local=1, Static=2, Stack=3, Fld=4, Elem=5, Meth=6;  
  
    public int kind; // Item-Art  
    public Struct type; // Typ des Operanden  
    public int val; // Con: Wert  
    public int adr; // Local, Static, Fld, Meth: Adresse  
    public Obj obj; // Meth: Methodenobjekt aus Symbolliste  
}
```





# Beispiel-Grammatik

**Assignment** = Designator "=" Expr .

**Expr** = Term { "+" Term } .

**Term** = Factor { "\*" Factor } .

**Factor** = number | Designator.

**Designator** = ident [ "." ident | "[" Expr "]" ] .



# Semantische Aktionen (1)

```
void Designator() {  
    check(Token.ident);  
  
    if (sym == Token.period) {  
        scan();  
  
        check(Token.ident);  
  
    } else if (sym == Token.lbrack) {  
        scan();  
  
        Expr();  
  
        check(Token.rbrack);  
    }  
}
```



# Semantische Aktionen (1)

```
Item Designator() {
    check(Token.ident);
    Item x = new Item(Tab.find(t.str));
    if (sym == Token.period) {
        scan();

        check(Token.ident);
        Obj obj = Tab.findField(t.str, x.type);
        x.kind = Item.Fld; x.type = obj.type; x.adr = obj.adr
    } else if (sym == Token.lbrack) {
        scan();

        Item y = Expr();

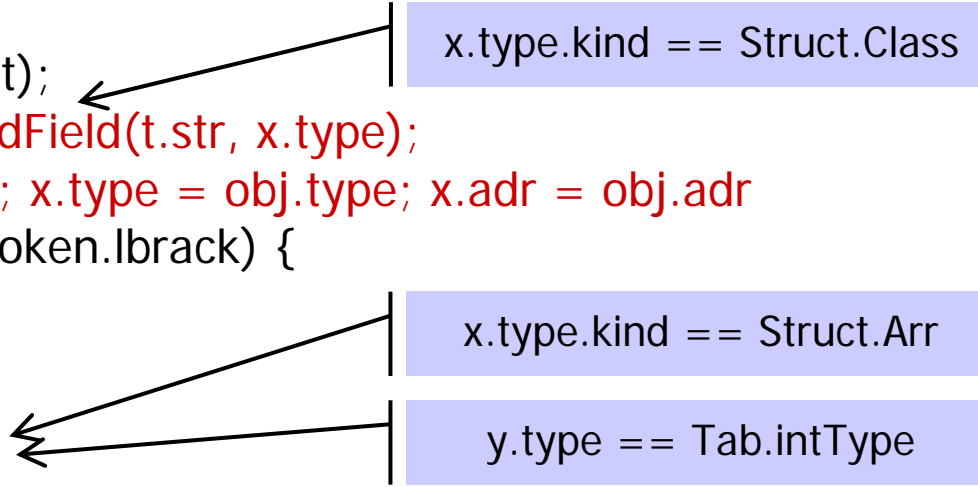
        x.kind = Item.Elem; x.type = x.type.elemType;
        check(Token.rbrack);
    }
    return x;
}
```

# Semantische Aktionen (1)

```

Item Designator() {
  check(Token.ident);
  Item x = new Item(Tab.find(t.str));
  if (sym == Token.period) {
    scan();
    Code.load(x);
    check(Token.ident);
    Obj obj = Tab.findField(t.str, x.type);
    x.kind = Item.Fld; x.type = obj.type; x.adr = obj.adr
  } else if (sym == Token.lbrack) {
    scan();
    Code.load(x);
    Item y = Expr();
    Code.load(y);
    x.kind = Item.Elem; x.type = x.type.elemType;
    check(Token.rbrack);
  }
  return x;
}

```



# Klasse *Item* – Konstruktor *Item(Obj)*



```
Item(Obj o) {
    type = o.type;
    val = o.val;
    adr = o.adr;
    switch (o.kind) {
        case Obj.Con:
            kind = Con; break;
        case Obj.Var:
            if (o.level == 0) kind = Static; else kind = Local; break;
        case Obj.Meth:
            kind = Meth; obj = o; break;
        default:
            Parser.error("NO_ITEM");
    }
}
```

# Klasse *Code* – Hilfsmethode *load()*

```
void load(Item x) {  
    switch (x.kind) {  
        case Item.Con:  
            if (x.type == Tab.nullType) put(const_0);  
            else loadConst(x.val); break;  
        case Item.Local:  
            if (0 <= x.adr && x.adr <= 3) put(load_0 + x.adr);  
            else { put(load); put(x.adr); } break;  
        case Item.Static:  
            put(getstatic); put2(x.adr); break;  
        case Item.Stack:  
            break; // nothing to do (already loaded)  
        case Item.Fld:  
            put(getfield); put2(x.adr); break;  
        case Item.Elem:  
            if (x.type == Tab.charType) put(baload);  
            else put(aload); break;  
        default:  
            Parser.error("NO_VAL");  
    }  
    x.kind = Item.Stack;  
}
```

# Semantische Aktionen (2)

```

Item Factor() {
    Item x;
    if (sym == Token.number) {
        scan();
        x = new Item(Item.Con, t.val, Tab.intType);
    } else {
        x = Designator();
    }
    return x;
}

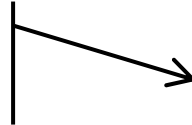
```

```

Item Term() {
    Item x = Factor();
    while (sym == Token.times) {
        scan();
        Code.load(x);
        Item y = Factor();
        Code.load(y);
        Code.put(Code.mul);
    }
    return x;
}

```

x.type == Tab.intType &&  
y.type == Tab.intType





# Semantische Aktionen (3)

```

Item Expr() {
    Item x = Term();
    while (sym == Token.plus) {
        scan();
        Code.load(x);
        Item y = Term();
        Code.load(y);
        Code.put(Code.add);
    }
    return x;
}

```

x.type == Tab.intType &&  
y.type == Tab.intType

```

void Assignment() {
    Item x = Designator();
    check(Token.assign);
    Item y = Expr();
    Code.assign(x, y);
}

```

y.type.assignableTo(x.type)



# Klasse *Code* – Hilfsmethode *assign*

```
void assign(Item x, Item y) {  
    load(y);  
    switch (x.kind) {  
        case Item.Local:  
            if (0 <= x.adr && x.adr <= 3) put(store_0 + x.adr);  
            else { put(store); put(x.adr); } break;  
        case Item.Static:  
            put(putstatic); put2(x.adr); break;  
        case Item.Fld:  
            put(putfield); put2(x.adr); break;  
        case Item.Elem:  
            if (x.type == Tab.charType) put(bastore);  
            else put(astore); break;  
        default:  
            Parser.error("NO_VAR");  
    }  
}
```

# Klasse *Struct* – Hilfsmethoden



```
boolean isRefType() {  
    return kind == Class || kind == Arr;  
}
```

```
boolean typeEqual(Struct other) {  
    if (kind == Arr) {  
        return other.kind == Arr && elemType.typeEqual(other.elemType);  
    } else {  
        return this == other;    // must be same type Obj node  
    }  
}
```

# Klasse *Struct* – Hilfsmethoden



```
boolean compatibleWith(Struct other) {  
    return this.typeEqual(other) ||  
        (this == Tab.nullType && other.isRefType()) ||  
        (other == Tab.nullType && this.isRefType());  
}
```

```
boolean assignableTo(Struct dest) {  
    return this.typeEqual(dest) ||  
        (this == Tab.nullType && dest.isRefType()) ||  
        (this.kind == Arr && dest.kind == Arr &&  
        dest.elemType == Tab.noType); // für len-Funktion  
}
```

Beispiel:  $b.x = \text{iarr}[5] + i * n$

**Deklaration: program A**

```

final int max = 12;           // Konstante
char c; int i;              // globale Variablen
class B { int x, y; }       // innere Klasse mit Feldern
{ void foo () int[] iarr; B b; int n; {...} }
  
```

