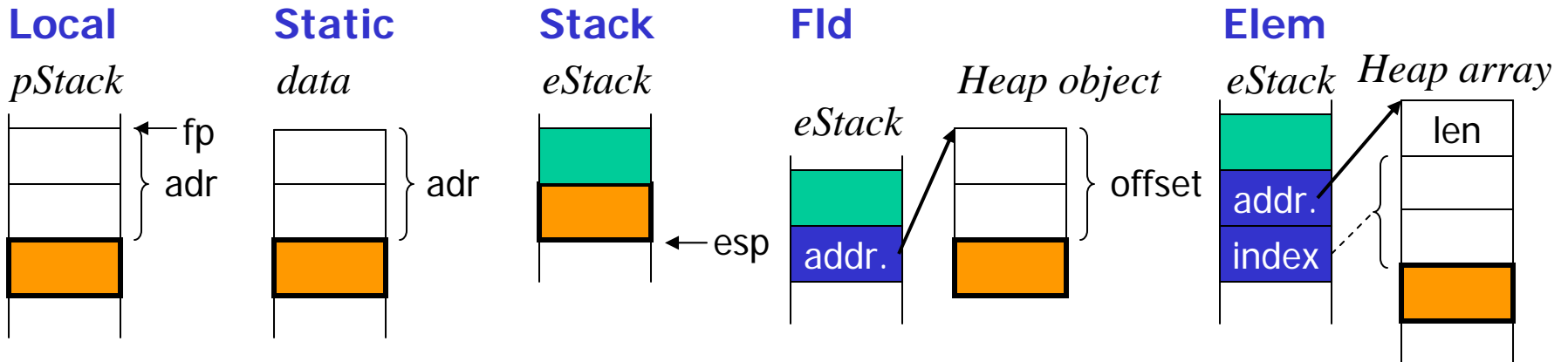


Klasse *Item*

```

class Item {
    public enum Kind {           // Mögliche Item-Arten
        Con, Local, Static, Stack, Fld, Elem, Meth
    }
    public Kind 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(ident);  
  
    if (sym == period) {  
        scan();  
  
        check(ident);  
  
    } else if (sym == lbrack) {  
        scan();  
  
        Expr();  
  
        check(rbrack);  
    }  
}
```

Semantische Aktionen (1)

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

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

        Item y = Expr();

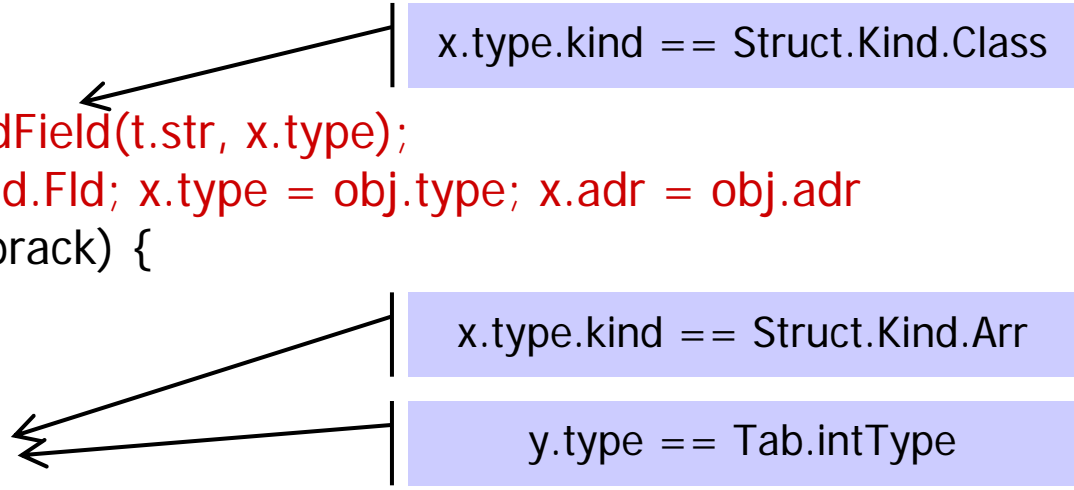
        x.kind = Item.Kind.Elem; x.type = x.type.elemType;
        check(rbrack);
    }
    return x;
}
```

Semantische Aktionen (1)

```

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

```



- `x.type.kind == Struct.Kind.Class`
- `x.type.kind == Struct.Kind.Arr`
- `y.type == Tab.intType`

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



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

Klasse *Code* – Hilfsmethode *load()*

```
void load(Item x) {
    switch (x.kind) {
        case Con:
            loadConst(x.val); break;
        case Local:
            if (0 <= x.adr && x.adr <= 3) { put(load_0 + x.adr); }
            else { put(load); put(x.adr); }
            break;
        case Static:
            put(getstatic); put2(x.adr); break;
        case Stack:
            break; // nothing to do (already loaded)
        case Fld:
            put(getfield); put2(x.adr); break;
        case 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 == number) {
        scan();
        x = new Item(t.val);
    } else {
        x = Designator();
    }
    return x;
}

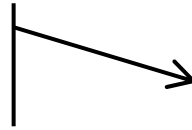
```

```

Item Term() {
    Item x = Factor();
    while (sym == 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 == 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(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 Local:  
            if (0 <= x.adr && x.adr <= 3) { put(store_0 + x.adr); }  
            else { put(store); put(x.adr); }  
            break;  
        case Static:  
            put(putstatic); put2(x.adr); break;  
        case Fld:  
            put(putfield); put2(x.adr); break;  
        case Elem:  
            if (x.type == Tab.charType) { put(bastore); }  
            else { put(astore); }  
            break;  
        default:  
            parser.error(NO_VAR);  
    }  
}
```

Klasse *Struct* – Hilfsmethoden



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

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

Klasse *Struct* – Hilfsmethoden



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

```
boolean assignableTo(Struct dest) {  
    return this.equals(dest) ||  
           (this == Tab.nullType && dest.isRefType()) ||  
           (this.kind == Kind.Arr && dest.kind == Kind.Arr &&  
            dest.elemType == Tab.noType); // for function len()  
}
```

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; {...} }
```

