Deklarierte Namen in MicroJava

Programm	`Program()`	
Konstanten	`ConstDecl()`	
Globale Variablen	`VarDecl()`	level = 0
Klassen	`ClassDecl()`	
Felder	`VarDecl()`	level = 1
Methoden	`MethodDecl()`	
Formale Parameter	`FormPars()`	
Lokale Variablen	`VarDecl()`	level = 1
Block	`Block()`	
Lokale Variablen	`InlineVarDecl()`	level = 2
Block	`Block()`	
...		level = 3+

Wird ein Name deklariert, wird er in die Symbolliste eingefügt

Objektknoten


```
class Obj {
  enum Kind {
    Con, Var, Type, Meth, Prog
  }

  Kind   kind;
  String name;
  Struct type;
  int    val;     // Con: value
  int    adr;     // Var, Meth: address
  int    level;   // Var: 0 → global, ≥1 → local
  int    nPars;   // Meth: number of parameters
  // Meth: parameters and local objects
  // Prog: global variables, constants,
  //       classes and methods
  Map<String, Obj> locals;
}
```

Strukturknoten und Scope-Knoten


```
class Struct {
  enum Kind {
    None, Int, Char, Arr, Class
  }
  Kind   kind;
  Struct elemType;  // Arr: element type
  Map<String, Obj> fields; // Class: list of fields

  int nrFields() { return fields.size(); } // Class
}


class Scope {
  Scope outer;   // next outer scope
  Map<String, Obj> locals; // list of objects in this scope
  int   nVars;   // number of variables in this scope
}
```

Symboltabelle


```
class Tab {
  public static final Struct
                  noType, intType, charType, nullType;
  public Obj noObj, chrObj, ordObj, lenObj;

  public Parser parser;  // target for errors
  public Scope curScope; // current scope
  private int curLevel;  // nesting level of current scope

  public Tab(Parser parser);
}

class TabImpl extends Tab {
  public void openScope();                  // shortcut for openScope(0)
  public void openScope(int nVars);         // local variables in outer scope
  public void closeScope();
  public Obj insert(Obj.Kind kind, String name, Struct type);
  public Obj find(String name);
  public Obj findField(String name, Struct type);
}
```

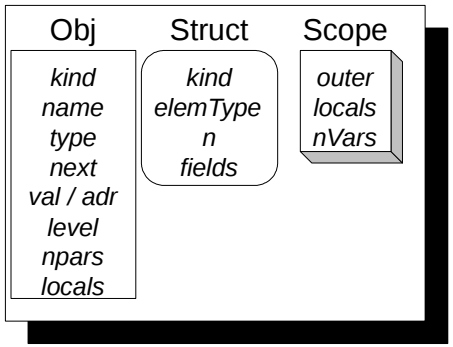
Beispiel: Symbollistenaufbau


```
program ABC ①
  char[] c;
  int max;
  char npp;
{
  int put ② (int x) ③ { ④
    var int x = x + 1; ⑤
    { ⑥
      print(x, 5);
      npp = 'C';
    }
    return x;
  } ⑦
} ⑧
```

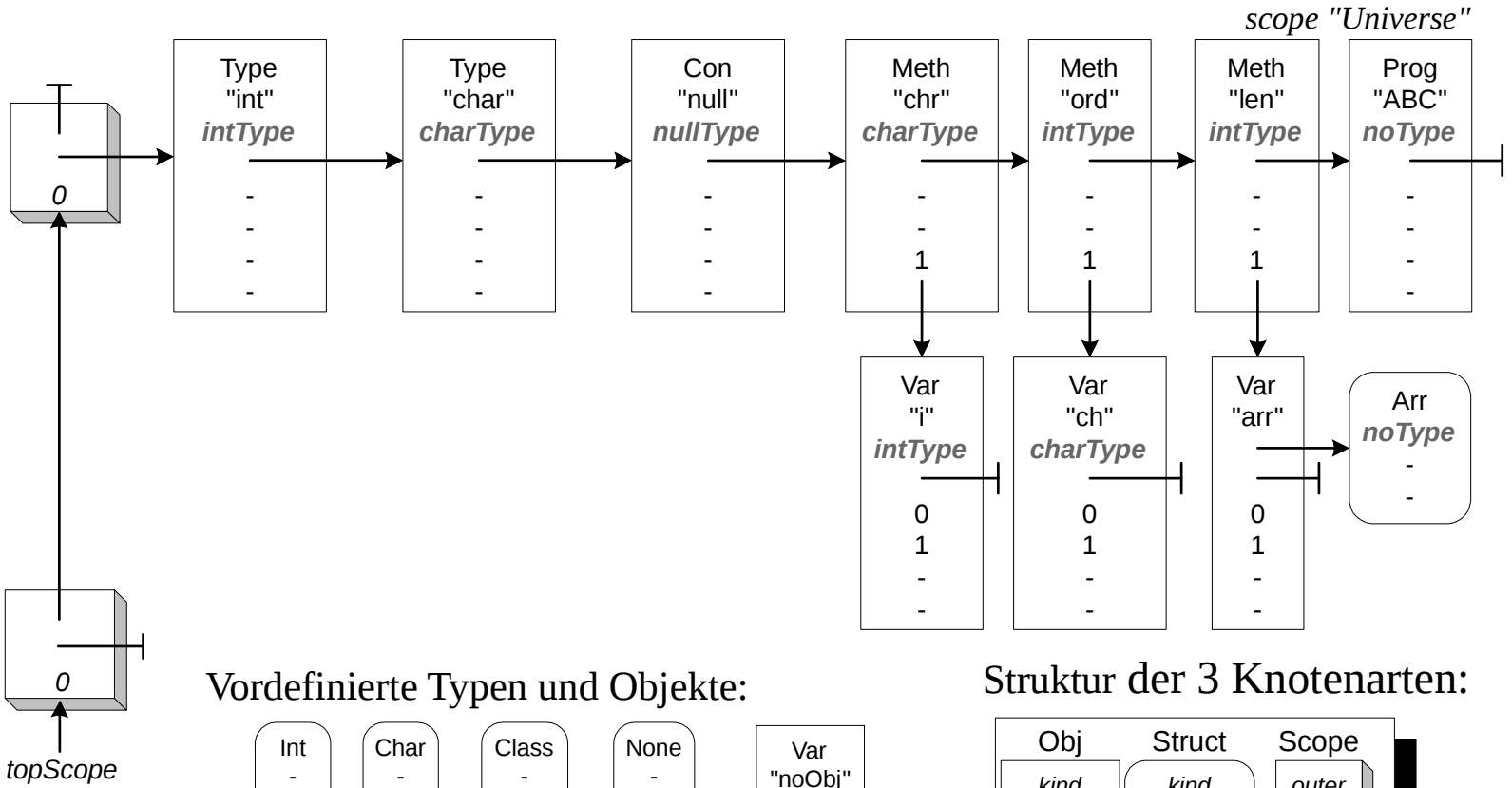
Beispiel: Symbollistenaufbau

```
program ABC ①
  char[] c;
  int max;
  char npp;
{
  int put ② (int x) ③ { ④
    var x = x + 1; ⑤
    { ⑥
      print(x, 5);
      npp = 'C';
    }
    return x;
  } ⑦
} ⑧
```

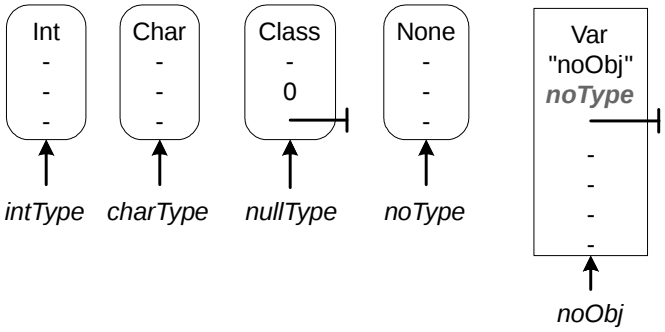
Struktur der 3 Knotenarten:



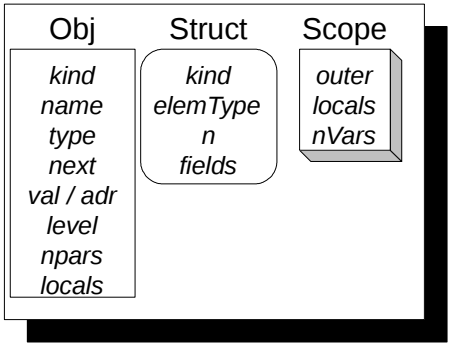
Beispiel: Bei Punkt ①



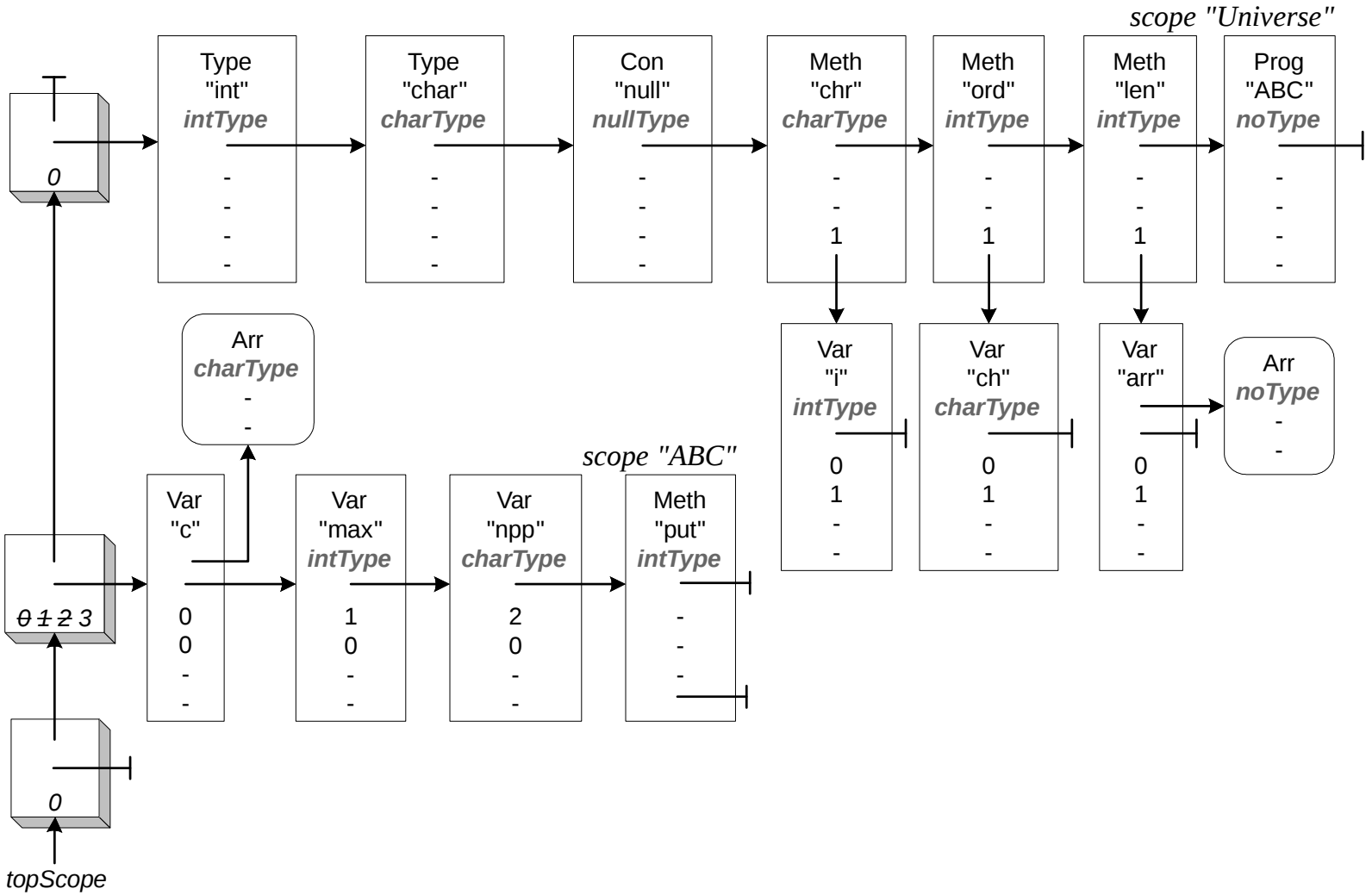
Vordefinierte Typen und Objekte:



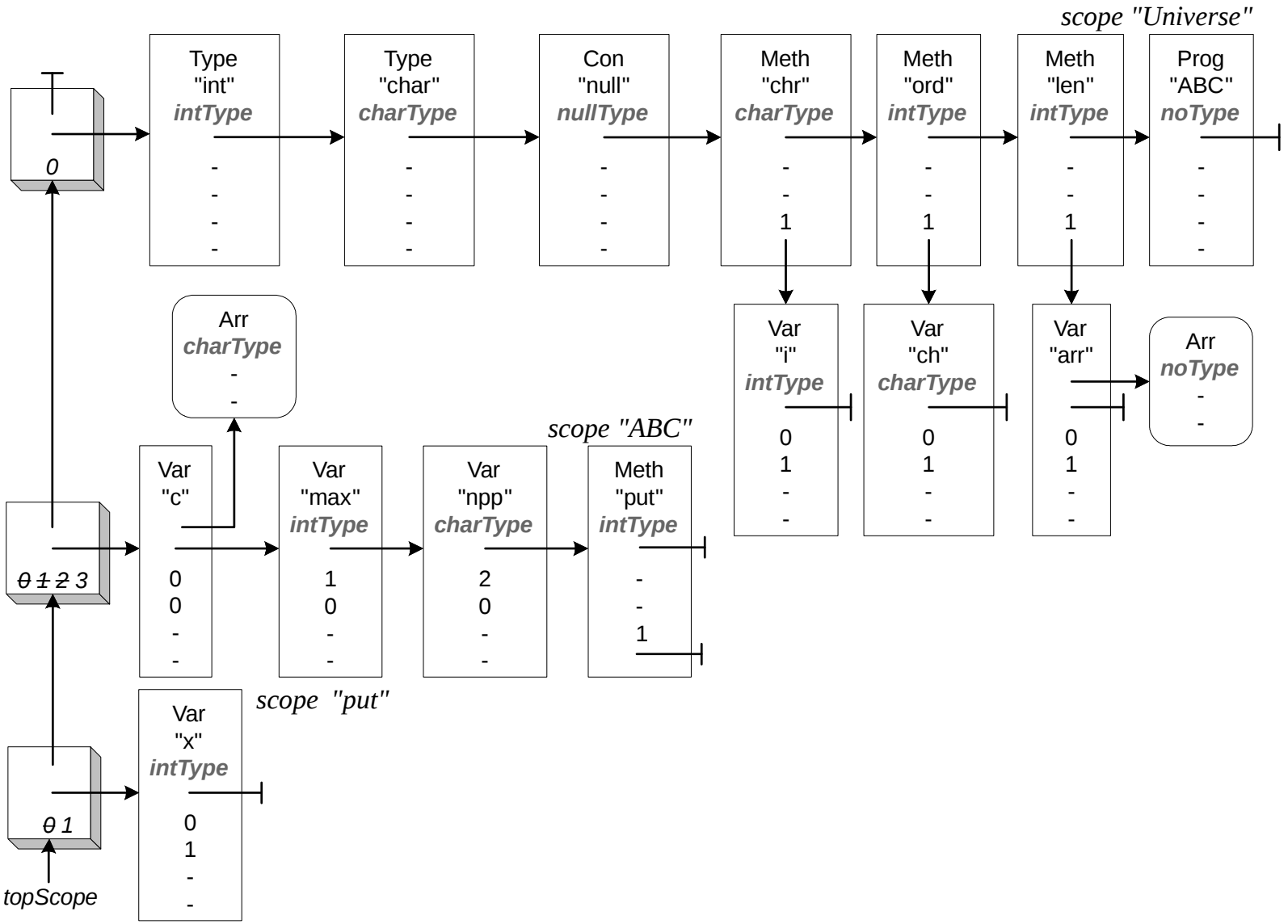
Struktur der 3 Knotenarten:



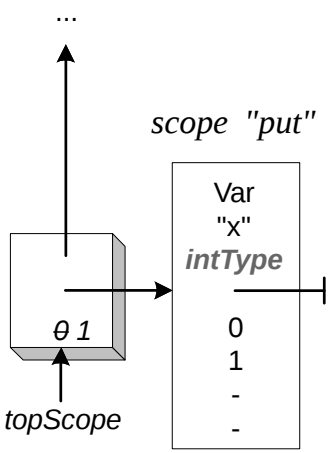
Beispiel: Bei Punkt ②



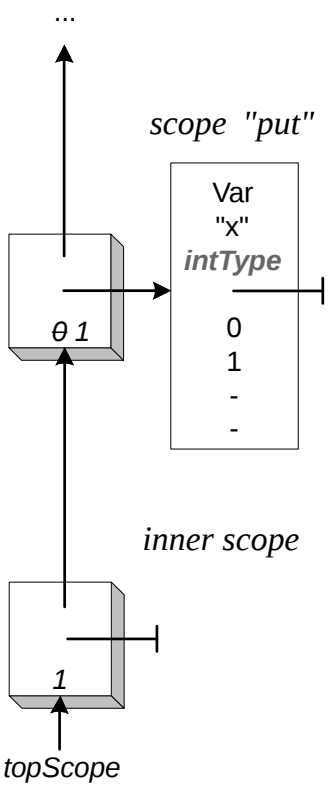
Beispiel: Bei Punkt ③



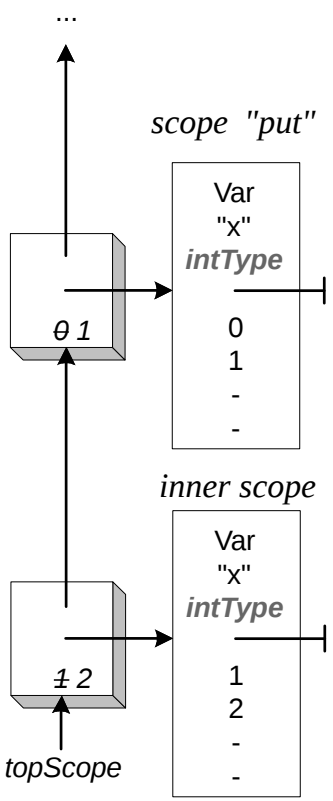
Beispiel: Bei Punkt ③



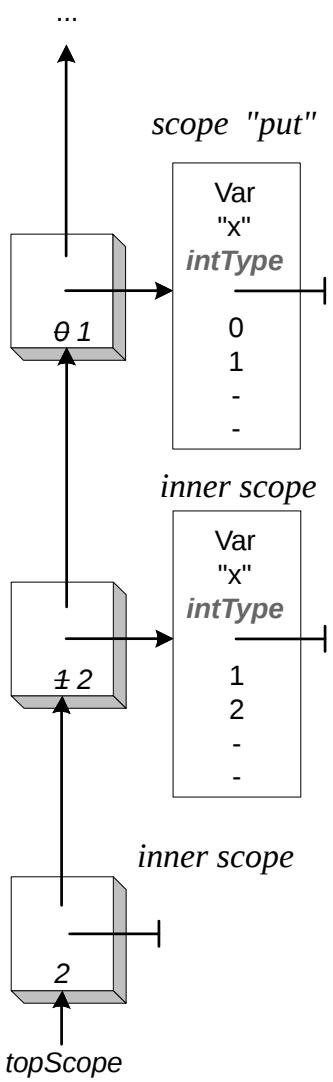
Beispiel: Bei Punkt ④



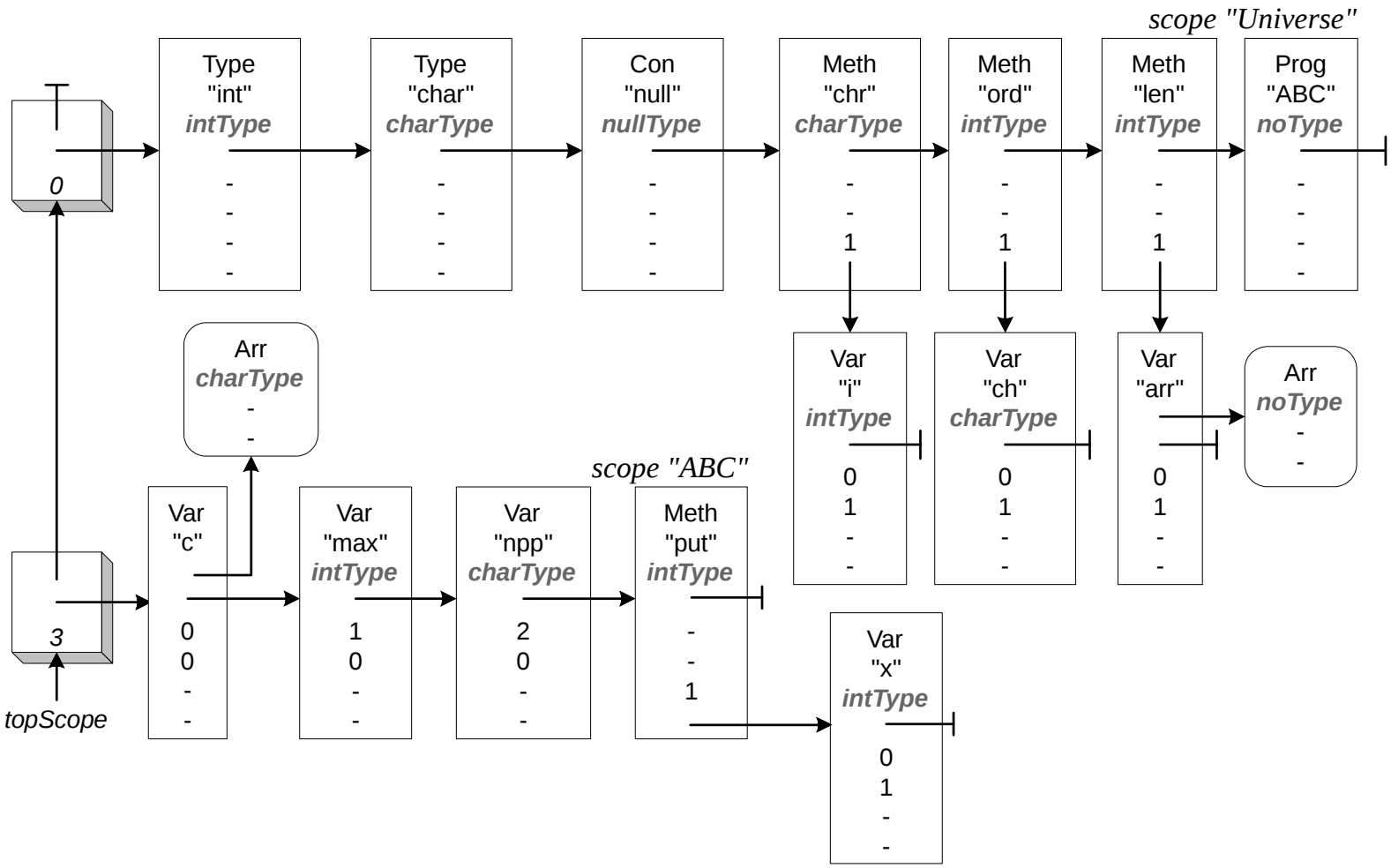
Beispiel: Bei Punkt ⑤



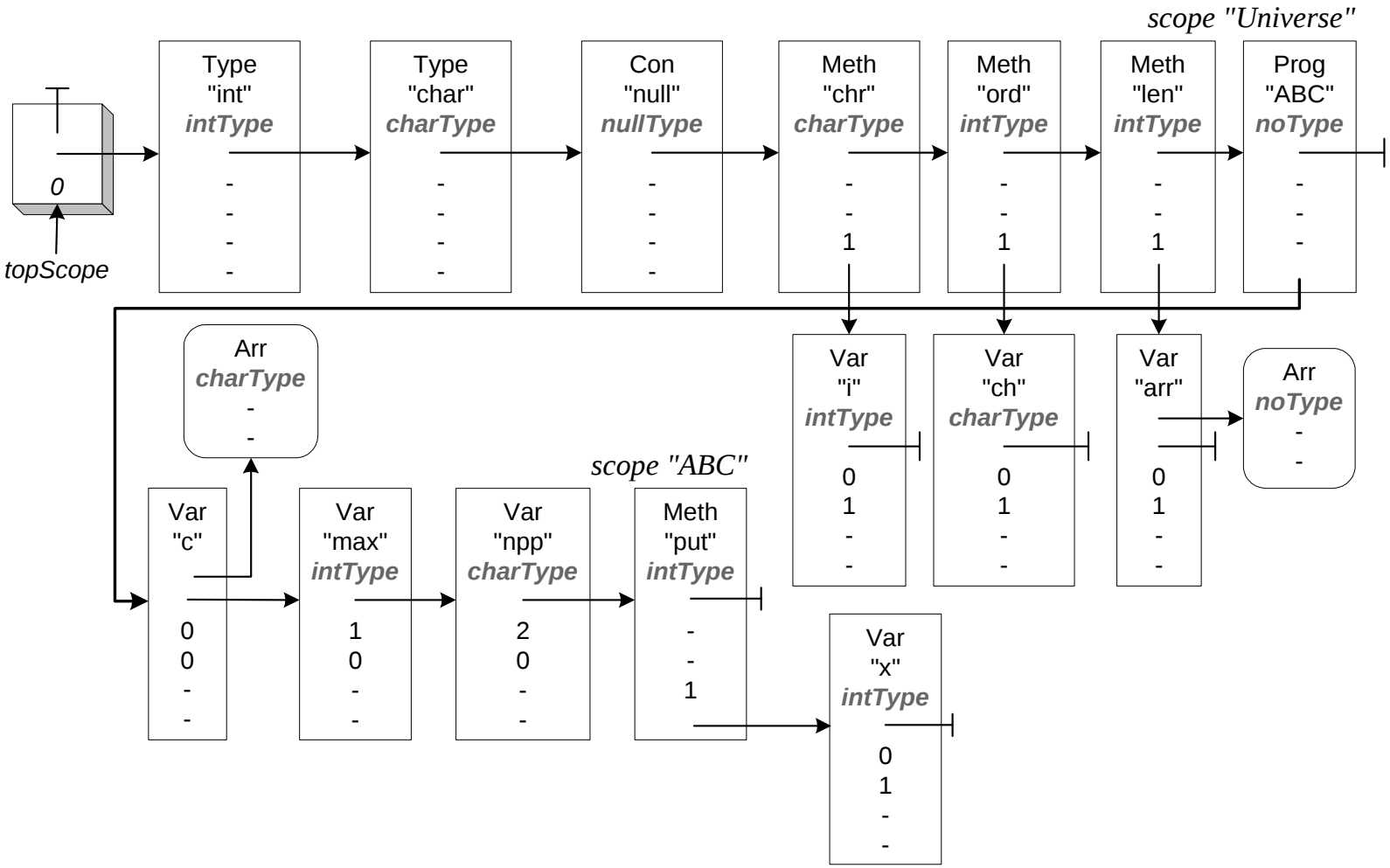
Beispiel: Bei Punkt ⑥



Beispiel: Bei Punkt ⑦



Beispiel: Bei Punkt ⑧



Füllen der Symbolliste


```
/** VarDecl = Type ident { "," ident } ";" . */
private void VarDecl() {
  Type();
  check(ident);

  while (sym == comma) {
    scan();
    check(ident);

  }
  check(semicolon);
}
```

Füllen der Symbolliste


```
/** VarDecl = Type ident { "," ident } ";" . */
private void VarDecl() {
  Struct type = Type();
  check(ident);
  tab.insert(Obj.Kind.Var, t.str, type);
  while (sym == comma) {
    scan();
    check(ident);
    tab.insert(Obj.Kind.Var, t.str, type);
  }
  check(semicolon);
}
```

Symbolliste verwenden


```
/** Type = ident [ "[" "]" ] . */
private void Type() {
  check(ident);




  if (sym == lbrack) {
    scan();
    check(rbrack);

  }

}
```

Symbolliste verwenden


```
/** Type = ident [ "[" "]" ] . */
private Struct Type() {
  check(ident);
  Obj o = tab.find(t.str);
  if (o.kind != Obj.Kind.Type) {
    error(NO_TYPE);
  }
  Struct type = o.type;
  if (sym == lbrack) {
    scan();
    check(rbrack);
    type = new Struct(type);
  }
  return type;
}
```

Klassen


```
/** ClassDecl = "class" ident "{" {VarDecl} "}" . */
private void ClassDecl() {
  check(class_);
  check(ident);


  check(lbrace);

  while (sym == ident) {
    VarDecl();
  }
  check(rbrace);


}
```

Klassen


```
/** ClassDecl = "class" ident "{" {VarDecl} "}" . */
private void ClassDecl() {
  check(class_);
  check(ident);
  Obj clazz = tab.insert(Obj.Kind.Type,
      t.str, new Struct(Struct.Kind.Class));
  check(lbrace);
  tab.openScope();
  while (sym == ident) {
    VarDecl();
  }
  check(rbrace);
  clazz.type.fields = tab.curScope.locals;
  tab.closeScope();
}
```

Block


```
/** Block = "{" { Statement } "}" . */
private void Block() {
        check(lbrace);

        Statements();




        check(rbrace);
}
```

Block


```
/** Block = "{" { Statement } "}" . */
private void Block() {
        check(lbrace);
        tab.openScope(tab.curScope.nVars());
        Statements();




        tab.closeScope();
        check(rbrace);
}
```

Block


```
/** Block = "{" { Statement } "}" . */
private void Block() {
        check(lbrace);
        tab.openScope(tab.curScope.nVars());
        Statements();
        if (tab.curScope.nVars() > MAX_LOCALS) {
                error(TOO_MANY_LOCALS);
        }

        tab.closeScope();
        check(rbrace);
}
```

Inline Variablen Deklaration


```
/** InlineVarDecl = Type ident [ "=" Expr ]. */
private void InlineVarDecl() {
        Type();
        check(ident);




        if (sym == assign) {
                scan();
                Expr();
        }


}
```

Inline Variablen Deklaration


```
/** InlineVarDecl = Type ident [ "=" Expr ]. */
private void InlineVarDecl() {
        StructImpl type = Type();
        check(ident);
        String name = t.str;

        // make sure name is unique in current scope
        if (tab.curScope.findLocal(name) != null) {
                error(DECL_NAME, name);
        }

        if (sym == assign) {
                scan();
                Expr();
        }

        tab.insert(Obj.Kind.Var, name, type);
}
```