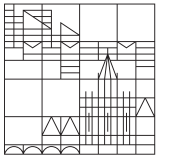
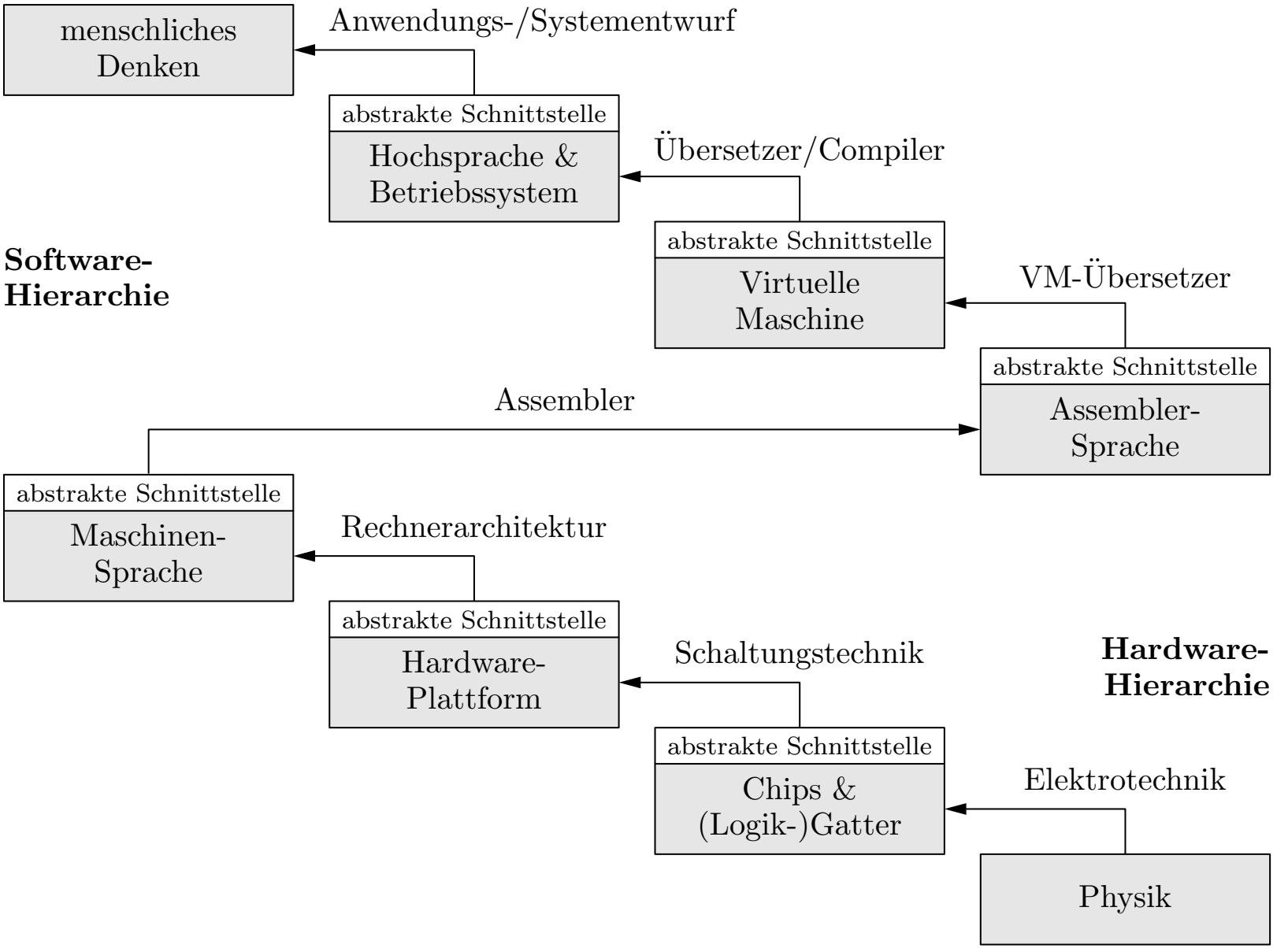
Universität Konstanz



# Rechnersysteme und -netze
# Kapitel 8
# Hochsprachen und Compiler

**Bastian Goldlücke**
Universität Konstanz
WS 2020/21

# Rechnersysteme: Plan der Vorlesung



# Erinnerung: Maschinensprache und Assembler

- **Die Hack-Maschinensprache**
  - A-Anweisungen (address instructions)
  - C-Anweisungen (compute instructions)

- **Assembler und Assemblersprache**
  - Physikalische und symbolische Programmierung
  - Maschinensprache und Assemblerprache
  - Opcodes, mnemonische Symbole (Mnemonics)
  - Die Hack-Assemblersprache
  - Typische Programmstrukturen in der Hack-Assemblersprache
  - Symbole und Symbolverwaltung
  - Programmübersetzung und Assemblerimplementierung
  - Beispiele für Assemblerprogramme: Linux und Windows

# Erinnerung: Virtuelle Maschine

- **Höhere Programmiersprachen und Übersetzung**
  - Direkte und zweistufige Übersetzung
  - Zwischensprache und virtuelle Maschine
  - Systembasierte und prozeßbasierte virtuelle Maschinen

- **Virtuelle Maschine des Hack-Systems**
  - Stapel(-speicher) und ihre Operationen
  - Stapelarithmetik (arithmetische und logische Operationen)
  - Speicherzugriff, Speicheraufteilung, Speichersegmente
  - Programmablauf (bedingte Anweisungen und Schleifen)
  - Objekt- und Arraybehandlung
  - Funktionsaufrufe, globaler Stapel zur Steuerung
  - Programmstart

# Höhere Programmiersprachen

**Programmieren**
**in verständlichen Sprachen**

- Maschinensprache besteht aus (Binär-)Zahlen und ist daher für einen Menschen nur sehr schwer verständlich (aber von einem Rechner direkt ausführbar).
- Assemblersprache ist immerhin symbolisch und daher für einen Menschen leichter verständlich, aber immer noch unbequem.
- Ende der 1940er Jahre schlug Grace Hopper vor, eine Programmiersprache zu entwickeln, die statt Zahlen und mnemonischen Symbolen Worte der englischen Sprache benutzt.
- Die Idee setzte sich zunächst nur langsam durch, führte aber schließlich zu höheren Programmiersprachen, wie wir sie heute kennen.

cecomhistorian.armylive.dodlive.mil

Grace Brewster Murray Hopper [1906–1992]

Schrieb den ersten Compiler (1952) und leistete wesentliche Vorarbeiten zur Entwicklung der Programmiersprache COBOL (“Grandma COBOL”).

# Programmiersprachen und -paradigmen

- Maschinensprache: „physikalische“ Programmierung
- Assemblersprache: symbolische Programmierung
- Fortran (formula translation): Formelübersetzung
- COBOL (common business oriented language),
  ALGOL (algorithmic language):
  **imperative Programmierung**, strukturierte Programme, Rekursion
- C, Pascal: **imperative Programmierung**
  größere Projekte möglich, gewerblicher Einsatz
- C++: **objektorientierte Programmierung**
- Java, C#: verbesserte **objektorientierte Programmierung**
- Lisp/Scheme/Haskell: **funktionale Programmierung**
- Prolog (programming in logic): **deklarative/logische Programmierung**

# Inhalt

# Inhalt

# Die Programmiersprache Jack

- Obwohl Jack eine echte, brauchbare Programmiersprache ist, wird sie in dieser Vorlesung nicht als Zweck gesehen.
- Vielmehr wird Jack zu Lehrzwecken verwendet:
  - wie man einen Übersetzer (compiler) implementiert,
  - wie Sprache und Übersetzer mit dem Betriebssystem zusammenarbeiten,
  - wie die oberste Ebene der Software-Hierarchie mit tieferen zusammenspielt.
- Die Programmiersprache Jack kann man in etwa einer Stunde lernen.

```
/** Hello World program. */
class Main {
   function void main() {
      /** Prints some text using the standard library. */
      do Output.printString("Hello World");
      do Output.println(); // New line
   }
}
```

Jack verfügt über:

- Java-ähnliche Syntax
- Kommentare
- Standardbibliothek

# Beispiel: Prozedurale Programmierung

- Ein Jack-Programm ist eine Sammlung von Jack-Klassen
- Eine Jack-Klasse ist eine Sammlung von Jack-Unterprogrammen
- Ein Jack-Unterprogramm ist
  - eine Funktion,
  - eine Methode oder
  - ein Konstruktor.

  (Beispiel rechts: nur Funktionen, da „objektlos“)
- Es muß genau eine Klasse mit dem Namen `Main` und in dieser Klasse genau eine Funktion mit dem Namen `main` geben.

```
class Main {

   /* Sums up 1 + 2 + 3 + …+ n */
   function int sum(int sum) {
      var int i, sum;
      let sum = 0;
      let i = 1;
      while (~(i > n)) {  // Java: (!(i > n))
         let sum = sum + i;
         let i = i + 1;
      }
      return sum;
   }

   function void main() {
      var int n, sum;
      let n = Keyboard.readInt(Ënter n: ``);
      let sum = Main.sum(n);
      do Output.printString("The result is: ");
      do Output.printInt(sum);
      do Output.println();
   }

} // Main
```

# Beispiel: Objektorientierte Programmierung 1

```
class BankAccount {
   static int nAccounts;

   // account properties
   field id;
   field String owner;
   field int balance;

   /* Constructs a new bank account. */
   constructor BankAccount new (String aOwner) {
      let id = nAccounts;
      let nAccount = nAccounts + 1;
      let owner = aOwner;
      let balance = 0;
      return this;
   }
   // … more BankAccount methods …

} // BankAccount
```

```
…
var int sum;
var BankAccount b, c;

let b = BankAccount.new("Joe");
…
```

- Mit Hilfe von Klassen kann man neue Datentypen definieren.
- Die Methoden der Klassen definieren die auf diesen Datentypen zulässigen Operationen.

- Eine Klasse, die einen neuen Datentyp definiert, muß mindestens einen `constructor` enthalten, mit dem ein Objekt des neuen Datentyps angelegt/erzeugt wird.

# Beispiel: Objektorientierte Programmierung 2

```
class BankAccount {
   static int nAccounts;

   // account properties
   field id;
   field String owner;
   field int balance;

   // constructor …(omitted)

   /** Deposits money in this account. */
   method void deposit (int amount) {
      let balance = balance + amount;
      return;
   }

   /** Withdraws money form this account. */
   method void withdraw (int amount) {
      if (balance > amount) {
         balance = balance - amount;
      }
      return;
   }
   // … more BankAccount methods …
} // BankAccount
```

```
…
var int sum;
var BankAccount b, c;

let b = BankAccount.new("Joe");
let c = BankAccount.new("Jane");

do b.deposit(5000);
let sum = 1000;
do b.withdraw(sum);
…
```

- Mit dem Konstruktor einer Klasse wird eine Instanz dieser Klasse / dieses Datentyps erzeugt.
- Mit den Methoden eines Datentypes kann der Zustand von Instanzen dieses Datentypes modifiziert werden.

# Beispiel: Objektorientierte Programmierung 3

```
class BankAccount {
   static int nAccounts;

   // account properties
   field id;
   field String owner;
   field int balance;

   // constructor …(omitted)

   /** Prints information about this account. */
   method void printInfo () {
      do Output.printInt(id);
      do Output.printString(owner);
      do Output.printInt(balance);
      return;
   }

   /** Disposes this account. */
   method void dispose () {
      do Memory.deAlloc(this);
      return;
   }

} // BankAccount
```

```
…
var int sum;
var BankAccount b, c;

let b = BankAccount.new("Joe");
// manipulation of b …

do b.printInfo();
do b.dispose();
…
```

- Jack verfügt über keine automatische Freigabe nicht mehr benötigten Speichers (garbage collection).
- Der Programmierer ist für die Freigabe nicht mehr benötigter Objekte verantwortlich (analog zu C/C++).

# Beispiel: Abstrakte Datentypen (Schnittstelle)

- Jack hat drei Basisdatentypen: `int`, `char`, `boolean`.
- Beispieldatentyp: Darstellung von Brüchen $\frac{n}{m}$ mit Ganzzahlen $n$ und $m$.

```
field int numerator, denominator;
constructor Fraction new (int a, int b);
method int getNumerator();
method int getDenominator();
method Fraction plus(Fraction other);

method void print();
```

```
// Fraction object properties
// returns a new Fraction object
// returns the numerator of this fraction
// returns the denominator of this fraction
// returns the sum of this fraction
// and another fraction, as a fraction
// print the fraction in the format
// "numerator/denominator"
```

```
// Computes the sum of 2/3 and 1/5.
class Main {
   function void main();
      var Fraction a, b, c;
      let a = Fraction.new(2,3);
      let b = Fraction.new(1,5);
      let c = a.plus(b); // compute c = a + b
      do c.print(); // should print "13/15"
      return;
   }}
```

- **API** (Schnittstelle) (application programming interface) Zusicherung von Funktionalität (public contract)
- Implementierung kann variieren, solange die Schnittstelle unverändert bleibt.

# Beispiel: Abstrakte Datentypen (Implementierung 1)

```
/** Provides the Fraction type and related services. */
class Fraction {
   field int numerator, denominator;

   constructor Fraction new (int a, int b) {
      let numerator = a; let denominator = b;
      do reduce();
      return this;
   }

   method void reduce () {
      // reduces the fraction -- see the book.
   }
   function int gcd (int a, int b) {
      // computes the greatest common divisor of a and b -- see the book.
   }

   method int getNumerator () {
      return numerator;
   }
   method int getDenominator () {
      return denominator;
   }
   // … more methods follow …
} // Fraction class
```

# Beispiel: Abstrakte Datentypen (Implementierung 2)

```
/** Provides the Fraction type and related services. */
class Fraction {

   // fields, constructor etc. from previous slide go here

   method Fraction plus (Fraction other) {
      var int sum;
      let sum = (numerator * other.getDenominator())
              + (other.getNumerator() * denominator);
      return Fraction.new(sum, denominator * other.getDenominator());
   }

   // More fraction-related methods come here: minus, times, div etc.

   /** Prints this fraction. */
   method void print () {
      do Output.printInt(numerator);
      do Output.printString("/");
      do Output.printInt(denominator);
   }

   /** Disposes this fraction. */
   method void dispose () {
      /** implementation omitted */ }
} // Fraction class
```

# Die Programmiersprache Jack

- Im folgenden werden wir die Elemente der Programmiersprache Jack genauer betrachten. Dazu gehören:
  - Syntax (Sprachelemente, Schlüsselworte etc.)
  - Datentypen (Basisdatentypen, abstrakte Datentypen etc.)
  - Arten und Sichtbarkeit von Variablen
  - Ausdrücke (expressions)
  - Anweisungen (statements)
  - Unterprogrammaufrufe (Funktionen, Methoden etc.)
  - Programmstruktur
  - Standardbibliothek

- Die vollständige Sprachdefinition findet man in dem Buch
  **The Elements of Computing Systems:**
  **Building a Modern Computer from First Principles**
  Noam Nisan & Shimon Schocken

# Syntax der Programmiersprache Jack 1

| | | |
|---|---|---|
| Leerzeichen und Kommentare | Leerzeichen, Tabulatoren, Neue-Zeile-Zeichen und Kommentare werden überlesen.<br>Die folgenden Kommentarformate werden unterstützt:<br>`//` Kommentar bis Zeilenende<br>`/*` Kommentar bis zum schließenden `*/`<br>`/**` Kommentar zur API-Dokumentation `*/` | |
| Symbole | `( )` Zur Gruppierung arithmetischer Ausdrücke und zum Einschließen von Parameter- und Argumentlisten<br>`[ ]` Zur Arrayindizierung<br>`{ }` Zur Gruppierung von Anweisungen und Programmteilen<br>`,` Separator für Listen von Variablen<br>`;` Abschluß einer Anweisung<br>`.` Klassenzugehörigkeit<br>`+ - * / & \| ~ < > =` Operatoren („~“ entspricht auch „!“) | |
| Schlüsselworte | `class`, `constructor`, `method`, `function` | Programmbausteine |
| | `int`, `boolean`, `char`, `void` | Basisdatentypen |
| | `var`, `static`, `field` | Variablendeklarationen |
| | `let`, `do`, `if`, `else`, `while`, `return` | Anweisungen |
| | `true`, `false`, `null` | Konstanten |
| | `this` | Objektreferenz |

# Syntax der Programmiersprache Jack 2

| | |
|---|---|
| Konstanten | Ganzzahlen (integer) müssen positiv sein und<br>in Standarddezimaldarstellung angegeben werden, z.B. `1984`.<br>Negative Zahlen wie z.B. `-13` sind keine Konstanten,<br>sondern Ausdrücke, die aus einer Konstante<br>und einem unären Operator bestehen.<br>Zeichenketten (strings) werden in zwei Anführungszeichen<br>(`"`, quote character) eingeschlossen und können beliebige Zeichen<br>außer Neue-Zeile-Zeichen und Anführungszeichen (`"`) enthalten.<br>(Diese Zeichen werden durch die Funktionen `String.newLine()` und `String.doubleQuote()` der Standardbibliothek geliefert.)<br>Boolesche Konstanten sind `true` und `false`.<br><br>Die Konstante `null` bezeichnet eine Null-Referenz. |
| Namen von Variablen und Funktionen | Namen sind beliebig lange Folgen<br>von Buchstaben (`A-Z`, `a-z`), Ziffern (`0-9`) und „`_`“ (Unterstrich).<br>Das erste Zeichen muß ein Buchstabe oder ein Unterstrich sein.<br><br>Die Sprache unterscheidet Groß- und Kleinschreibung:<br>`x` und `X` sind verschiedene Namen. |

# Datentypen der Programmiersprache Jack

- **Basisdatentypen** (primitive types)
  - `int` 16-Bit-Ganzzahlen im Zweierkomplement (`-15`, `2`, `3`, …).
  - `boolean` Werte `0` und `-1`, die für falsch bzw. wahr stehen.
  - `char` Unicode-Zeichen (‘`a`’, ‘`x`’, ‘+’, ‘%’ …)
- **Abstrakte Datentypen** (vom Betriebssystem oder Benutzer definiert)
  - `String` (definiert durch Betriebssystem)
  - `Fraction` (definiert durch Benutzer)
  - `List` (definiert durch Benutzer)
- **Anwendungsspezifische Datentypen** (vom Benutzer definiert)
  - `BankAccount`
  - `Bat` / `Ball`
  - …

# Jack-Datentypen: Speicheranforderung

```
// This code assumes the existence of Car and Employee classes.
// Car objects have model and licensePlate fields.
// Employee objects have name and car fields.
var Employee e, f;          // creates variables e, f that contain null references
var Car c;                  // creates a variable c that contains a null reference
…
let c = Car.new("Jaguar","007");            // constructs a new Car object
let e = Employee.new("Bond",c);             // constructs a new Employee object
// At this point c and e hold the base addresses of the new memory segments
// allocated to the two objects.
let f = e;                  // Only the reference is copied -- no new object is constructed.
```

- Objekttypen werden durch einen Klassennamen dargestellt und durch eine Referenz implementiert, d.h., eine Speicheradresse.

- Speicheranforderung:
  - Variablen der Basisdatentypen wird bei ihrer Deklaration Speicher zugeordnet.
  - Objektvariablen wird Speicher zugeordnet, wenn sie durch einen Konstruktor erzeugt werden.

# Arten und Sichtbarkeit von Jack-Variablen

| Variablenart | Definition / Beschreibung | Wo deklariert? | Sichtbarkeit |
|---|---|---|---|
| statische Variablen | `static type name1, name2, …;`<br>Es gibt nur eine Kopie jeder statischen Variable und diese Kopie wird von allen Objektinstanzen der Klasse geteilt (wie `private static` in Java). | Klassen-Deklaration | Klasse, in der sie deklariert sind |
| Instanzvariablen | `field type name1, name2, …;`<br>Jede Objektinstanz der Klasse hat eine private Kopie der Instanzvariablen (wie `private` in Java). | Klassen-Deklaration | Klasse, in der sie deklariert sind, außer Funktionen |
| lokale Variablen | `var type name1, name2, …;`<br>Lokale Variablen werden auf dem Stapel angelegt, wenn das Unterprogramm aufgerufen wird, und freigegeben, wenn es zurückkehrt (wie lokale Variablen in Java). | Unterprogramm-Deklaration | Unterprogramm, in dem sie deklariert sind |
| Parametervariablen | `type name1, name2, …;`<br>Bezeichnen Eingaben für Unterprogramme:<br>`function void drive`<br>`(Car c, int miles);` | Unterprogramm-deklarationen (Parameterlisten) | Unterprogramm, in dem sie deklariert sind |

# Jack-Ausdrücke

Ein **Ausdruck** der Programmiersprache Jack kann sein:

- eine **Konstante**,
- ein sichtbarer **Variablenname**
  (die Variable kann `static`, `field`, `local`, oder `parameter` sein),
- das Schlüsselwort `this`, das das aktuelle Objekt bezeichnet
  (kann nicht in Funktionen verwendet werden).
- ein **Arrayelement** in der Schreibweise `name[expression]`,
  wobei `name` ein sichtbarer Variablenname vom Typ `Array` sein muß.
- ein **Unterprogrammaufruf** der keinen `void`-Typ zurückliefert.
- ein Ausdruck dem einer der unären Operatoren `-` oder `~` vorangestellt ist:
  - `-expression`: arithmetische Negation;
  - `~expression`: Boolesche Negation (bitweise für Ganzzahlen).
- eine Verknüpfung von zwei Ausdrücken mit einem binären Operator, d.h.,
  `expression operator expression`, wobei `operator` einer der folgenden ist:
  - `+ - * /`: arithmetische Ganzzahloperationen;
  - `& |` : Boolesches Und, Oder;
  - `< > =` : Vergleichsoperatoren
- ein Ausdruck in Klammern, d.h., `( expression )`.

# Jack-Anweisungen

```
let variable = expression;
```
oder
```
let variable[expression] = expression;
```

- Wertzuweisung

```
if (expression) {
   statements
} else {
   statements
}
```

- bedingte Anweisung

```
while (expression) {
   statements
}
```

- Schleife

```
do function-or-method-call;
```

- Unterprogrammaufruf

```
return expression;
```
oder
```
return;
```

- Wertrückgabe bzw.
  Rückkehr aus Unterprogramm

# Jack-Unterprogrammaufrufe 1

- Allgemeine Syntax: `subroutineName(arg1, arg2, …)`
- Jedes Argument muß ein gültiger Jack-Ausdruck sein.
- Argumente werden als Wert übergeben (call by value, für Basisdatentypen) oder als Referenz (call by reference, für Objekttypen).

**Beispiele:** Angenommen, wir haben `function int sqrt(int n)`.
Diese Funktion kann wie folgt aufgerufen werden (call by value):

- `sqrt(17)` Argument ist eine Konstante
- `sqrt(x)` Argument ist eine Variable
- `sqrt(a *c -17)` Argument ist ein zusammengesetzter Ausdruck
- `sqrt(a *sqrt(c -17) +3)` Argument ist ein zusammengesetzter Ausdruck

Angenommen, wir haben `method Matrix plus (Matrix other)`.

- Sind `u` und `v` vom Typ Matrix, kann dies so aufgerufen werden: `u.plus(v)`.
- Die Variable `v` wird als Referenz übergeben (call by reference).

# Jack-Unterprogrammaufrufe 2

```
class Foo {
   // some subroutine declarations -- code omitted
   …
   method void f() {
      var Bar b;                  // declares a local variable of class type Bar
      var int i;                  // declares a local variable of primitive type int
      …
      do g(5,7);                  // calls method g of class Foo (on this object)
      do Foo.p(2);                // calls function p of class Foo
      do Bar.h(3);                // calls function h of class Bar
      let b = Bar.r(4);           // calls constructor or function r of class Bar
      do b.q();                   // calls method q of class Bar (on object b)
      let i = w(b.s(3), Foo.t());             // calls method w on this object
                                              // method s on object b and function
                                              // or constructor t of class Foo
      …
   }
}
```

# Jack-Programmstruktur

**Klassendeklarationen:**

```
class name {
   field declarations
   static variable declarations
   constructor type name (parameter-list) {
      declarations
      statements
   }
   method type name (parameter-list) {
      declarations
      statements
   }
   function type name (parameter-list) {
      declarations
      statements
   }
```

- Reihenfolge von `field` und `static` ist beliebig.
- Reihenfolge der Unterprogrammdeklarationen ist beliebig.

- Jede Klasse gehört in eine eigene Datei (compilation unit).
- Der Dateiname und der Klassenname müssen übereinstimmen.
- Ein **Jack-Programm** ist eine Sammlung von Klassen.
- Es muß genau eine Klasse `Main` mit genau einer Funktion `main` geben, d.h., genau ein `Main.main()`.

  (Es können auch noch weitere Funktionen vorhanden sein.)
- Wenn ein Jack-Programm gestartet wird, wird die Funktion `Main.main()` ausgeführt.

# Beachtenswerte Eigenschaften der Sprache Jack

- Das (lästige) Schlüsselwort `let`, z.B. in `let x = 0;` und
  das (lästige) Schlüsselwort `do`, z.B. in `do reduce();` (für `void` Funktionen).
- Keine Operatorpräzedenz: `1+2*3` ergibt `9` (Auswertung von links nach rechts);
  um das erwartete Ergebnis zu erzielen, schreibe man: `1+(2*3)`.
- Nur drei Basisdatentypen: `int`, `boolean`, `char`;
  alle drei Datentypen werden als 16-Bit-Zahl dargestellt.
- Keine Typwandlung/Typprüfung: beliebige Zuweisungen sind möglich.
- Array-Deklaration: `Array x;` gefolgt von `let x = Array.new();`
- Statische Methoden werden Funktionen genannt (`function`).
- Konstruktor-Methoden heißen `constructor`;
  ein Konstruktor wird aufgerufen durch `ClassName.new(argsList);`
- Warum hat Jack diese (z.T. lästigen) Eigenschaften?
  **Um das Schreiben eines Übersetzers einfach zu machen!**

# Jack-Standardbibliothek und Betriebssystemfunktionen

```
class Math {
  function void init()
  function int abs (int x)
  function int multiply (int x, int y)
  function int divide (int x, int y)
  function int min (int x, int y)
  function int max (int x, int y)
  function int sqrt (int x, int y)
}
```

```
class Array {
  constructor Array new (int size)
  method void dispose ()
}
```

```
class String {
  constructor String new (int maxLength)
  method void   dispose ()
  method int    length ()
  method char   charAt (int j)
  method void   setCharAt (int j, char c)
  method String appendChar (char c)
  method void   eraseLastChar ()
  method int    intValue ()
  method void   setInt (int j)
  function char backspace ()
  function char doubleQuote ()
  function char newLine ()
}
```

```
class Screen {
  function void init ()
  function void clearScreen ()
  function void setColor (boolean b)
  function void drawPixel (int x, int y)
  function void drawLine (int x1, int y1, int x2, int y2)
  function void drawRectangle (int x1, int y1, int x2, int y2)
  function void drawCircle (int x, int y, int r)
}
```

# Jack-Standardbibliothek und Betriebssystemfunktionen

```
class Output {
  function void init ()
  function void moveCursor (int i, int j)
  function void printChar (char c)
  function void printString (String s)
  function void printInt (int i)
  function void println ()
  function void backspace ()
}
```

```
class Memory {
  function void   init ()
  function int    peek (int address)
  function void   poke (int address)
  function Array  alloc (int size)
  function void   deAlloc (Array o)
}
```

```
class Keyboard {
  function void   init ()
  function char   keyPressed ()
  function char   readChar ()
  function String readLine (String message)
  function int    readInt  (String message)
}
```

```
class Sys {
  function void init ()
  function void halt ()
  function void error (int errorCode)
  function void wait  (int duration)
}
```

- Diese Standardbibliothek sollte in jeder Implementierung der Programmiersprache Jack zur Verfügung stehen.
- Sie kann auch als sehr einfaches Betriebssystem gesehen werden.

# Inhalt

# Warum betrachten wir Übersetzer?

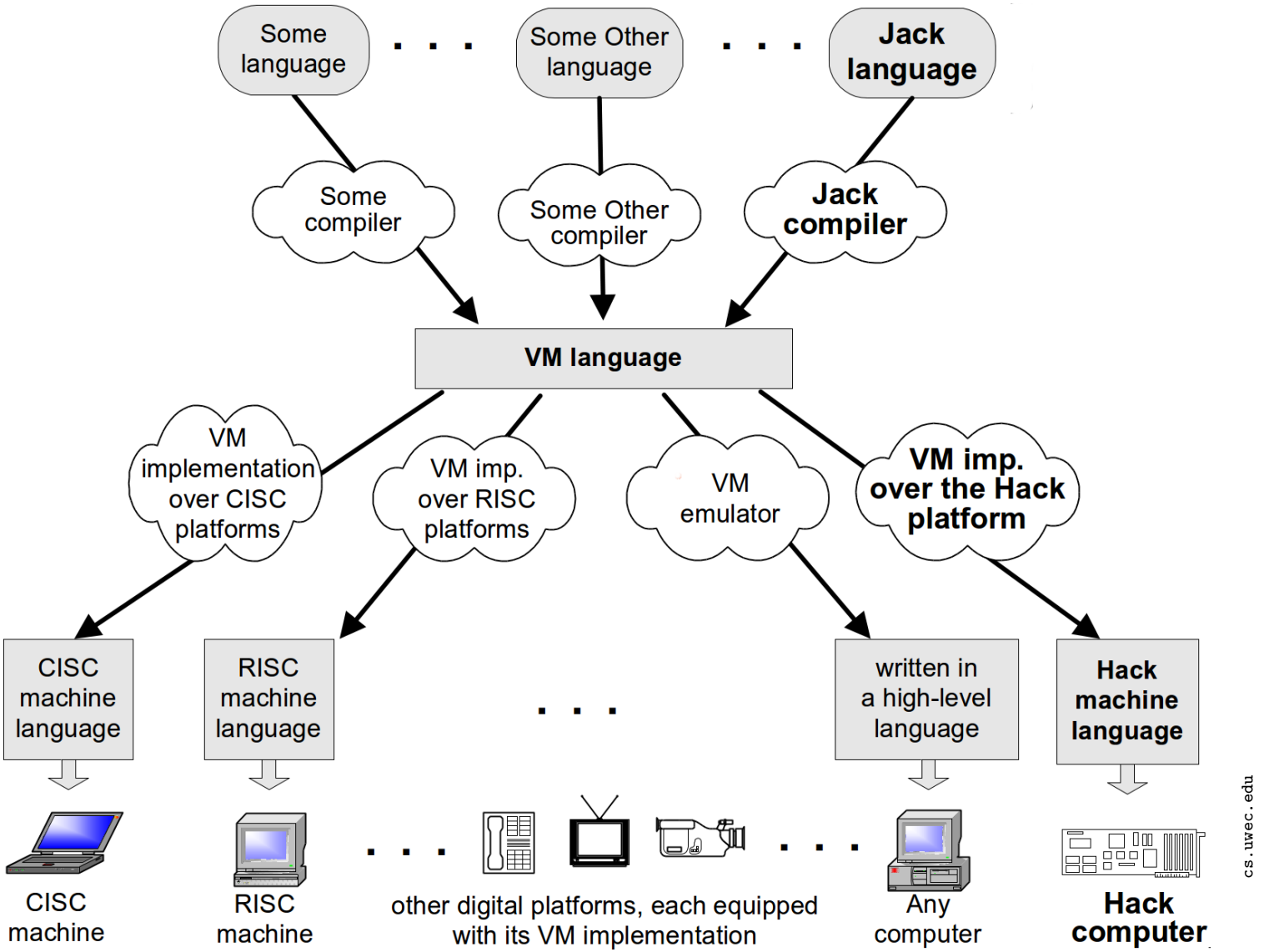
**Weil Übersetzer** (***compiler***) …

- ein wesentlicher Teil der Informatik sind,
- sehr wichtig für die Computerlinguistik sind,
- durch klassische Programmiertechniken implementiert werden,
- wichtige Programmierprinzipien und Entwurfsmethoden benutzen,
- ein gutes Training für die Entwicklung von Programmen sind, die eine Struktur in eine andere umwandeln (Programme, Dateien, Transaktionen, …)
- dazu anleiten, in Begriffen von „Beschreibungssprachen“ zu denken.

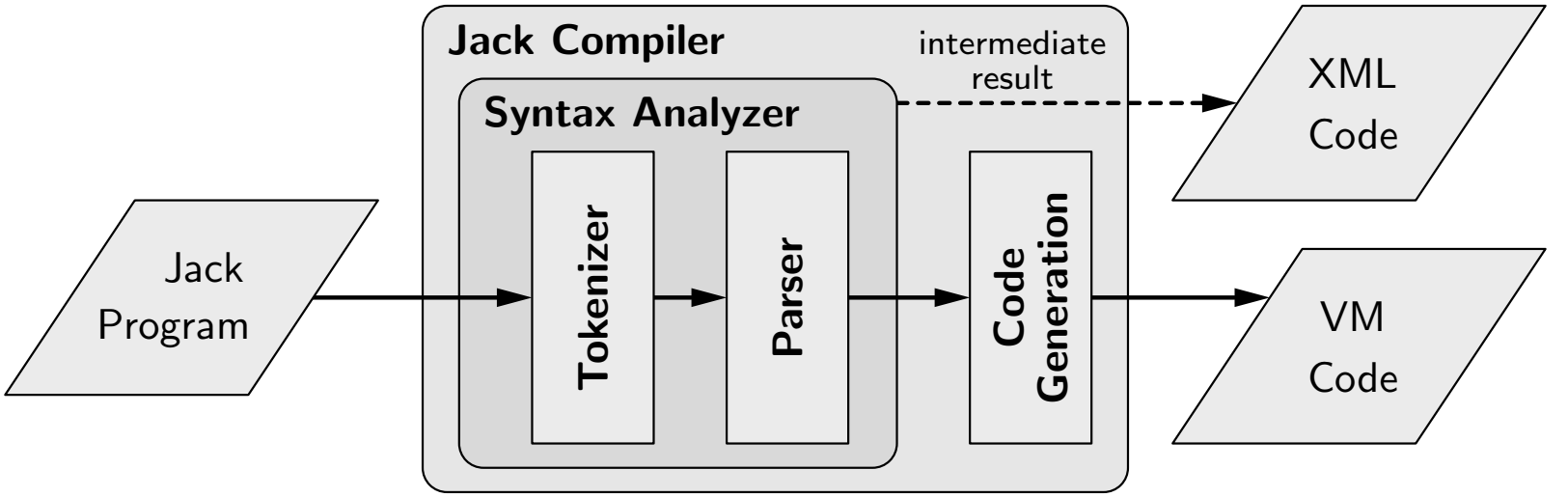
**Moderne Übersetzer sind zweistufig.**

- **1. Stufe** (front-end): von Hochsprache nach Zwischensprache
- **2. Stufe** (back-end): von Zwischensprache in Maschinensprache

# Moderne Übersetzer sind zweistufig



# Übersetzerarchitektur (1. Stufe, Front-End)

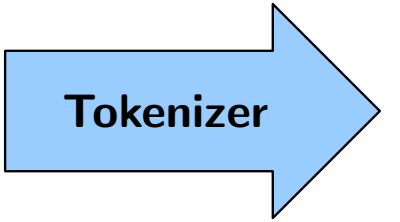



- **Syntaxanalyse:** Verstehen der durch den Quelltext ausgedrückten Semantik.
  - Lexikalische Analyse (tokenizing):
    Erzeugen eines Stroms von Sprachatomen (token stream).
  - Parsen (parsing):
    Zuordnen der Sprachatome (token) zu der Sprachgrammatik.
  - XML-Ausgabe als Nachweis, daß die Syntaxanalyse funktioniert.
- **Codeerzeugung:** Nachbilden der Semantik des Quelltextes in der Zielsprache.

# Lexikalische Analyse (Tokenizing)

**Code-Ausschnitt**

```
while (count <= 100) { /** demonstration */
   count++;
   // body of while continues
   …
```




**Tokens**

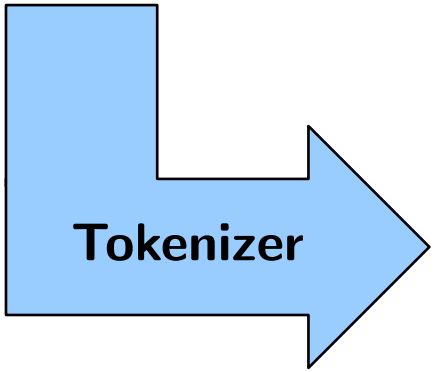
```
while
(
count
<=
100
)
{
count
++
;
…
```

- Entfernen von Leerzeichen und Kommentaren.
- Erzeugen einer Liste von Sprachatomen (tokens).
- Zu beachtende sprachspezifische Eigenschaften:
  - Regeln: wie sind z.B. „++“ oder „<=“ zu behandeln?
  - Klassifikationen: Schlüsselworte, Symbole, Bezeichner, Konstanten etc.
- In der lexikalischen Analyse wird normalerweise nicht nur ein Strom von Sprachatomen erzeugt, sondern den Sprachatomen auch ihre lexikalische Klasse zugeordnet und ggf. ihr Wert (etwa der Zahlenwert bei numerischen Konstanten).

# Lexikalische Analyse (Tokenizing)

## Jack-Quelltext

```
if (x < 153) { let city = "Paris"; }
```




## XML-Ausgabe der lexikalischen Analyse

```
<tokens>
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> < </symbol>
  <integerConstant> 153 </integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> city </identifier>
  <symbol> = </symbol>
  <stringConstant> Paris </stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
</tokens>
```

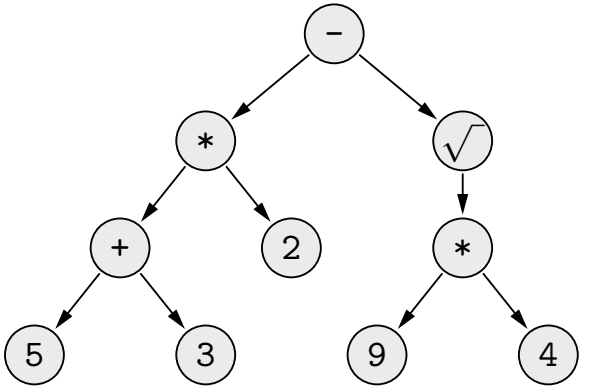
# Kontextfreie Grammatiken und Parser

- Jede (formale) Sprache kann durch eine **Grammatik** beschrieben werden.
- Eine solche Grammatik besteht aus einer Menge von **Produktionsregeln**, die angeben, wie aus Wörtern (Sätzen) neue Wörter (Sätze) produziert werden.
- Je nach den Einschränkungen, denen die Produktionsregeln unterliegen, unterscheidet man verschiedene Arten von (formalen) Grammatiken ($\Rightarrow$ theoretische Informatik, formale Sprachen, Chomsky-Hierarchie)
- Für Programmiersprachen werden sogenannte **kontextfreie Grammatiken** benutzt, da diese die Implementierung von Übersetzern vereinfachen.
- Kontextfreie Grammatiken beschreiben Sprachen, die von **Stapel-/Kellerautomaten** erkannt werden können. (Dies ist ein Grund für Stapel als zentrales Element virtueller Maschinen.)
- Eine Grammatik kann auf zwei Weisen gesehen werden:
  - Deklarativ: Welche Kombinationen von Sprachatomen sind erlaubt?
  - Analytisch: Wie kann man Kombinationen von Sprachatomen zerlegen?

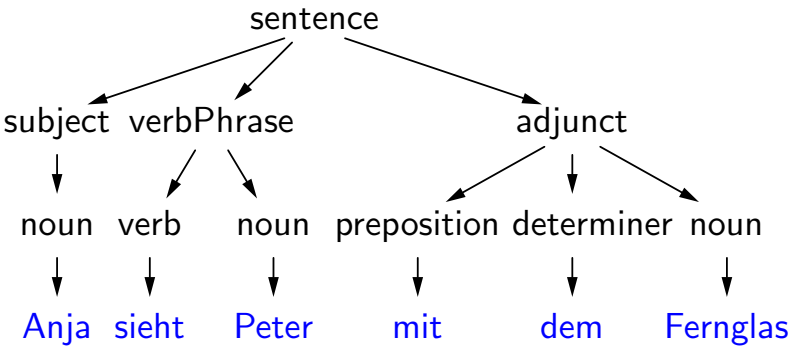
# Beispiele für das Parsen von Ausdrücken/Sätzen

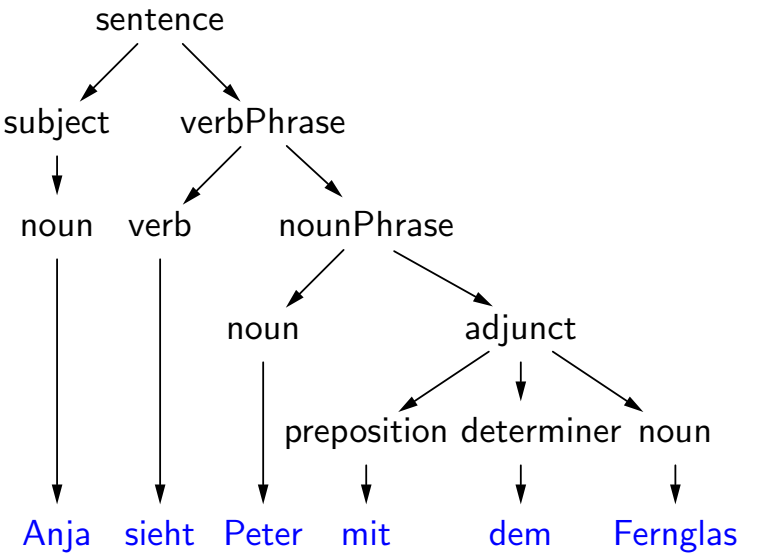
### Jack-Ausdruck

```
(5+3)*2 - sqrt(9*4)
```




### Satz in deutscher Sprache




- Parsen ist ein schwieriger Prozeß, der an Zweideutigkeiten leiden kann.
- Zweideutigkeiten sollten in Programmiersprachen vermieden werden.

# Typische Grammatik einer C-artigen Sprache

## Grammatik

```
program:              statement

statement:            whileStatement
                    | ifStatement
                    | // other statement possibilities
                    | '{' statementSequence '}'

whileStatement:       'while' '(' expression ')' statement

ifStatement:          simpleIf
                    | ifElse

simpleIf:             'if' '(' expression ')' statement

ifElse:               'if' '(' expression ')' statement
                      'else' statement

statementSequence:    '' // null, i.e, the empty sequence
                    | statement ';' statementSequence

expression:         // definition of an expression

// more definitions follow
```

## Beispielcode

```
while (expression) {
  if (expression)
    while (expression) {
      statement;
      if (expression)
        statement;
    }
  while (expression) {
    statement;
    statement;
  }

if (expression) {
  statement;
  while (expression)
    statement;
  if (expression)
    if (expression)
      statement;
    else
      statement;
}
```

- Einfache (terminale) und komplexe (nicht-terminale) Formen.
- Grammatik: Regeln, wie komplexe aus einfacheren Formen erzeugt werden.
- Im allgemeinen sind Grammatiken für Programmiersprachen stark rekursiv.

# Parse-Bäume

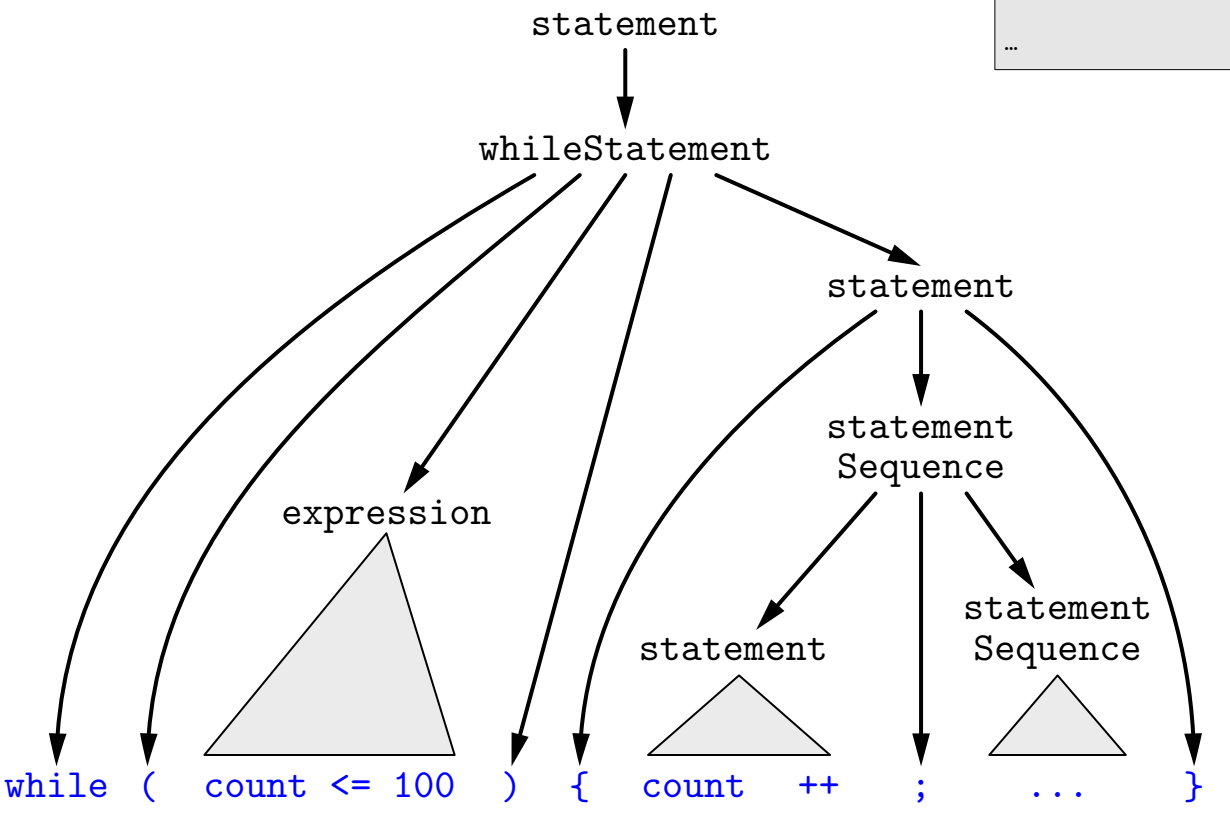
**Source Code:**

```
while (count <= 100) {
   /** demonstration */
   count++;
   …
```

```
program:          statement
statement:        whileStatement
                | ifStatement
                | // other possibilities
whileStatement:   'while'
                  '(' expression ')'
                  statement
…
```

**Tokens**

```
while
(
count
<=
100
)
{
count
++
;
…
```



# Parsen durch rekursiven Abstieg

```
…
statement:           whileStatement
                   | ifStatement
                   | // other statement possibilities
                   | '{' statementSequence '}'
whileStatement:      'while' '(' expression ')' statement
ifStatement:       // definition of an if statement
statementSequence:   '' // null, i.e, the empty sequence
                   | statement ';' statementSequence
expression:        // definition of an expression
// more definitions follow
```

```
while (expression) {
  statement;
  statement;
  while (expression)
    while (expression) {
      statement;
      statement;
    }
}
```

- Stark rekursiv.
- LL(0)-Grammatiken: Das erste Sprachatom bestimmt die Regel.
- In anderen Grammatiken muß man ggf. vorausschauen (look ahead).
- Jack ist fast LL(0).

Implementierung: Eine Funktion je Regel.

- `parseStatement();`
- `parseWhileStatement();`
- `parseIfStatement();`
- `parseStatementSequence();`
- `parseExpression();`

# Die Jack-Grammatik 1

| | |
|---|---|
| **Lexical elements:** | The Jack language includes five types of terminal elements (tokens): |
| keyword: | 'class' \| 'constructor' \| 'function' \| 'method' \| 'field' \| 'static' \| 'var' \| 'int' \| 'char' \| 'boolean' \| 'void' \| 'true' \| 'false' \| 'null' \| 'this' \| 'let' \| 'do' \| 'if' \| 'else' \| 'while' \| 'return' |
| symbol: | '{' \| '}' \| '(' \| ')' \| '[' \| ']' \| '.' \| ',' \| ';' \| '+' \| '-' \| '*' \| '/' \| '&' \| '\|' \| '<' \| '>' \| '=' \| '~' |
| integerConstant: | A decimal number on the range 0 . . . 32767. |
| stringConstant: | '"' a sequence of Unicode characters not including double quotes or newline '"' |
| identifier | A sequence of letters, digits and underscore ('_') not starting with a digit. |
| **Program structure:** | A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax: |
| class: | 'class' className '{' classVarDec∗ subroutineDec∗ '}' |
| classVarDec: | ('static' \| field) type varName (',' varName)∗ ';' |
| type: | 'int' \| 'char' \| 'boolean' \| className |
| subroutineDec: | ('constructor' \| 'function' \| 'method') ('void' \| type) subroutineName '(' parameterList ')' subroutineBody |
| parameterList: | ((type varName) (',' type varName)∗)? |
| subroutineBody: | '{' varDec* statements '}' |
| varDec: | 'var' type varName (',' type varName)∗ ';' |
| className: | identifier |
| subroutineName: | identifier |
| varName: | identifier |

| | |
|---|---|
| 'x': | x appears verbatim |
| x: | x is a language construct |
| x?: | x appears 0 or 1 times |
| x*: | x appears 0 or more times |
| x\|y: | either x or y appears |
| (x,y): | x appears, then y |

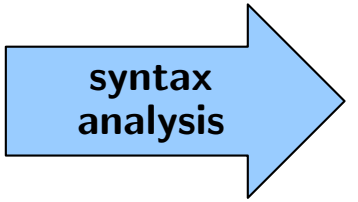
# Die Jack-Grammatik 2

| | |
|---|---|
| **Statements:** | |
| statements: | statement$*$ |
| statement: | letStatement \| ifStatement \| whileStatement \| doStatement \| returnStatement |
| letStatement: | 'let' varName ('[' expression ']')? '=' expression ';' |
| ifStatement: | 'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')? |
| whileStatement: | 'while' '(' expression ')' '{' statements '}' |
| doStatement: | 'do' subroutineCall ';' |
| returnStatement: | 'return' expression? ';' |
| **Expressions:** | |
| expression | term (op term)$*$ |
| term: | integerConstant \| stringConstant \| keywordConstant \| varName \| varName '[' expression ']' \| subroutineCall \| '(' expression ')' \| unaryOp term |
| subroutineCall: | subroutineName '(' expressionList ')' \| (className \| varName) '.' subroutineName '(' expressionList ')' |
| expressionList: | (expression (',' expression)$*$)? |
| op | '+' \| '−' \| '*' \| '/' \| '&' \| '\|' \| '<' \| '>' \| '=' |
| unaryOp | '−' \| '~' |
| keywordConstant | 'true' \| 'false' \| 'null' \| 'this' |

| | |
|---|---|
| 'x': | x appears verbatim |
| x: | x is a language construct |
| x?: | x appears 0 or 1 times |
| x*: | x appears 0 or more times |
| x\|y: | either x or y appears |
| (x,y): | x appears, then y |

# Jack-Syntaxanalyse

```
class Bar {
  method Fraction foo (int y) {
    var int temp; // a variable
    let temp = (xxx+12)*-63;
    …
  …
```




```
<varDec>
  <keyword> var </keyword>
  <keyword> int </keyword>
  <identifier> temp <identifier>
  <symbol> ; </symbol>
</varDec>
<statements>
  <letStatement>
    <keyword> let </keyword>
    <identifier> temp </identifier>
    <symbol> = </symbol>
    <expression>
      <term>
      <symbol> ( </symbol>
      <expression>
        <term>
          <identifier> xxx </identifier>
        </term>
        <symbol> + </symbol>
        <term>
          <integerConstant> 12 </integerConstant>
        </term>
      </expression>
    </expression> ...
```

## Syntax-Analysierer

- Auf der Grammatik aufbauend, kann ein Syntax-Analysierer implementiert werden.
- Er liest einen Quelltext und vergleicht ihn mit der Grammatik.
- Falls dies erfolgreich ist, kann er einen Parse-Baum in einem strukturierten Format erzeugen, z.B. in XML.

# Bemerkungen

- Der Parse-Baum kann während des Lesens des Quelltextes konstruiert werden (on the fly).

- Syntax-Analysierer können leicht mit Hilfe von speziellen Werkzeugen erstellt werden, z.B.:
  - `lex` und `flex` für die lexikalische Analyse
  - `yacc` und `bison` für das Parsen

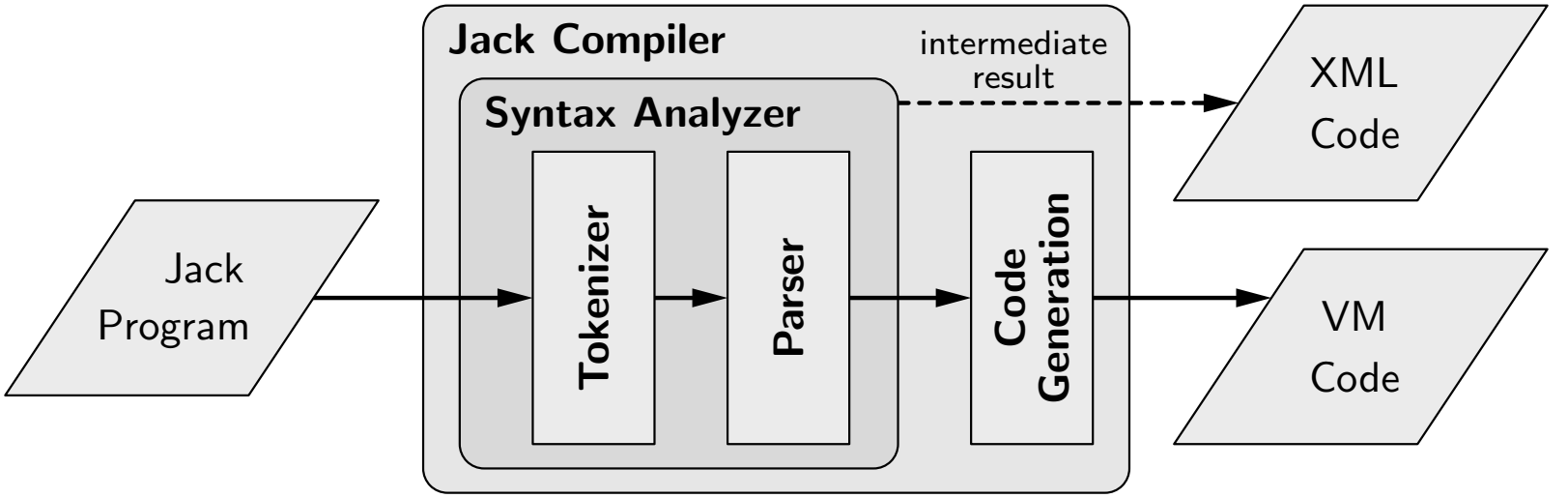
- Die Jack-Sprache ist absichtlich einfach gehalten:
  - Anweisungspräfixe: `let`, `do`, …
  - keine Operatorpräzedenz
  - keine Fehlerprüfung

- Typische Sprachen sind reichhaltiger und erfordern mächtigere Übersetzer.

- Der Jack-Übersetzer wurde entworfen, um die Grundideen zu illustrieren, die modernen Übersetzern zugrundeliegen.

# Codeerzeugung

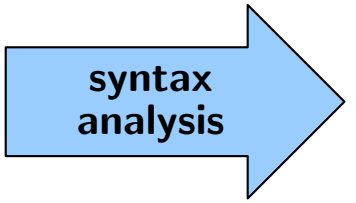
- **Syntaxanalyse:** Verstehen der durch den Quelltext ausgedrückten Semantik.
- **Codeerzeugung:** Nachbilden der Semantik des Quelltextes in der Zielsprache.



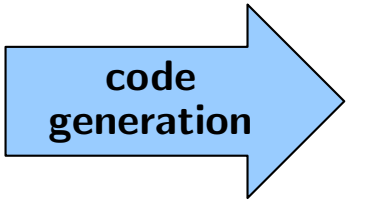

- Gesucht: Erweiterung des Syntax-Analysierers,
  so daß statt passiven XML-Codes ausführbarer VM-Code erzeugt wird.
- Herausforderungen der Codeerzeugung:
  - Datenbehandlung
  - Anweisungsbehandlung

# Codeerzeugung: Beispiel

```
method int foo () {
  var int x;
  let x = x + 1;
  …
```




```
<letStatement>
  <keyword> let </keyword>
  <identifier> x </identifier>
  <symbol> = </symbol>
  <expression>
    <term>
      <identifier> x </identifier>
    </term>
    <symbol> + </symbol>
    <term>
      <constant> 1 </constant>
    </term>
      </expression>
    </expression> ...
```




```
push local 0
push constant 1
add
pop local 0
```

Wenn der Übersetzer auf eine Variable `x` trifft, muß er wissen:

- Was ist der Datentyp von `x`?
  Basisdatentyp oder abstrakter Datentyp? ⇒ Speicheranforderung
- Was für eine Art von Variable ist `x`?
  `local`, `static`, `field`, `argument`? ⇒ Speichersegment

# Variablen: Abbildung auf Speichersegmente

```
class BankAccount {
  // class variables
  static int nAccounts;
  static int bankCommission;
  // account properties
  field int id;
  field String owner;
  field int balance;

  method void transfer (int sum, BankAccount from, Date when) {
    var int i, j;        // some local variables
    var Date due;        // Date is a user-defined type
    let balance = (balance + sum) - commission(sum * 5);
    // more code …
  }
```

- Die Zielsprache verwendet acht Speichersegmente: `argument`, `local`, `static`, `constant`, `this`, `that`, `pointer`, `temp`
- Jedes Speichersegment, z.B. `static`, ist eine indizierbare Folge von 16-Bit-Zahlen, die angesprochen werden z.B. durch `static 0`, `static 1` etc.

Wenn die obige Klasse übersetzt wird, werden die Variablen zugeordnet:

| | | | |
|---|---|---|---|
| Klassenvariablen | `nAccounts`, `bankCommisson` | → | `static 0, 1` |
| Instanzvariablen | `id`, `owner`, `balance` | → | `this 0, 1, 2` |
| Argumentvariablen | `sum`, `from`, `when` | → | `argument 0, 1, 2` |
| lokale Variablen | `i`, `j`, `due` | → | `local 0, 1, 2` |

# Variablen: Symboltabellen

```
class BankAccount {
  // class variables
  static int nAccounts;
  static int bankCommission;
  // account properties
  field int id;
  field String owner;
  field int balance;

  method void transfer (int sum, BankAccount from, Date when) {
    var int i, j;        // some local variables
    var Date due;        // Date is a user-defined type
    let balance = (balance + sum) - commission(sum * 5);
    // more code …
  }
```

- Der Übersetzer erzeugt eine verkettete Liste von Hash-Tabellen, von denen jede eine Sichtbarkeitsebene darstellt.
- Das Aufsuchen einer Variable schreitet von der aktuellen Tabelle in Richtung auf den Listenkopf fort.

## Symboltabelle für Klasse

| Name | Type | Kind | # |
|---|---|---|---|
| nAccounts | int | static | 0 |
| bankCommission | int | static | 1 |
| id | int | field | 0 |
| owner | String | field | 1 |
| balance | int | field | 2 |

## Symboltabelle für Methode

| Name | Type | Kind | # |
|---|---|---|---|
| this | BankAccount | argument | 0 |
| sum | int | argument | 1 |
| from | BankAccount | argument | 2 |
| when | Date | argument | 3 |
| i | int | local | 0 |
| j | int | local | 1 |
| due | Date | local | 2 |

# Variablen: Lebenszyklus

Symboltabelle für Klasse

| Name | Type | Kind | # |
|---|---|---|---|
| nAccounts | int | static | 0 |
| bankCommission | int | static | 1 |
| id | int | field | 0 |
| owner | String | field | 1 |
| balance | int | field | 2 |

Symboltabelle für Methode

| Name | Type | Kind | # |
|---|---|---|---|
| this | BankAccount | argument | 0 |
| sum | int | argument | 1 |
| from | BankAccount | argument | 2 |
| when | Date | argument | 3 |
| i | int | local | 0 |
| j | int | local | 1 |
| due | Date | local | 2 |

**Lebenszyklen von Variablen:**

| | |
|---|---|
| `static` Variablen | Eine Kopie muß erhalten bleiben solange das Programm läuft. |
| `field` Variablen | Für jedes Objekt muß eine eigene Kopie erzeugt werden. |
| `local` Variablen | Werden beim Eintritt in ein Unterprogramm erzeugt, beim Verlassen freigegeben. |
| `argument` Variablen | Ähnlich zu lokalen Variablen. |

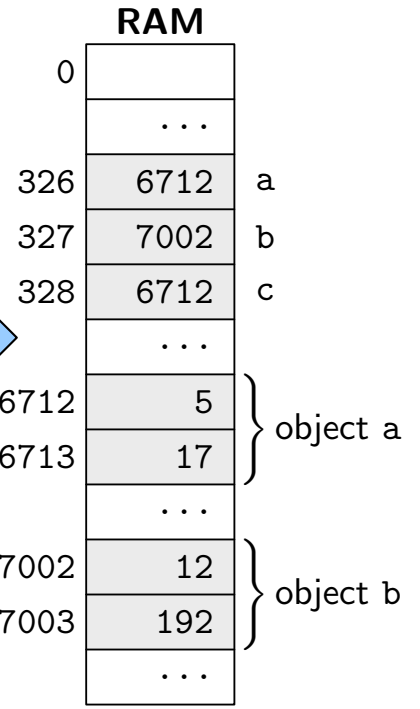
**Die virtuelle Maschine kümmert sich bereits um diese Dinge!**
(Folglich braucht der Übersetzer hier gar nichts zu tun.)

# Objekte: Speicheranforderung

## Java-Quelltext

```
class Complex {
  // fields (properties):
  int re;                 real part
  int im;                 imaginary part
  …
  /** Constructs a new complex number */
  public Complex (int re, int im) {
    this.re = re;
    this.im = im;
  }
  …

class Foo {
  public void bla () {
    Complex a, b, c;
    …
    a = new Complex(5,17);
    b = new Complex(12,192);
    …
    c = a;                // only the reference is copied
    …
  }
}
```

following compilation




`foo = new ClassName(…)`

erzeugt

`foo = Memory.alloc(n);`

wobei `n` die Anzahl Speicherzellen ist, die benötigt werden, um ein solches Objekt zu speichern. An `foo` wird die Basisadresse des Speicherbereichs zugewiesen.

# Objekte: Speicherzugriff

## Java-Quelltext

```
class Complex {
  // fields (properties):
  int re;               real part
  int im;               imaginary part
  …
  /** Constructs a new complex number */
  public Complex (int re, int im) {
    this.re = re;
    this.im = im;
  }
  …
  /** Multiplies this complex number
      by the given scalar */
  public void mult (int c) {
    re = re * c;
    im = im * c;
  }
  …
}
```

Übersetzung der Zuweisung

```
im = im * c;
```

- Aufsuchen der beiden Variablen in der Symboltabelle.
- Erzeugen des Codes:

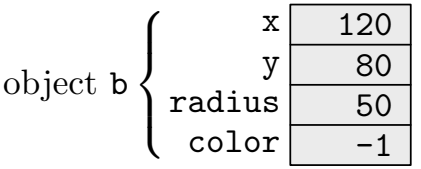
```
*(this + 1) = *(this + 1)
              times
              (argument 0)
```

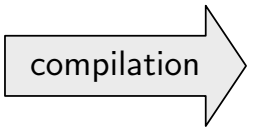
Dieser Pseudo-Code sollte in der Zielsprache ausgedrückt werden.

# Objekte: Zugriff auf Instanzvariablen

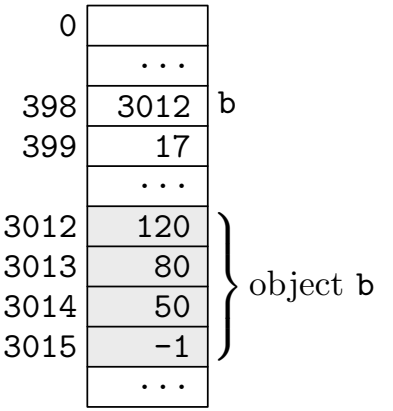
Hintergrund: Wir haben ein Objekt `b` vom Typ `Ball`, das Felder `x`, `y`, `radius` und `color` besitzt.

high-level program view



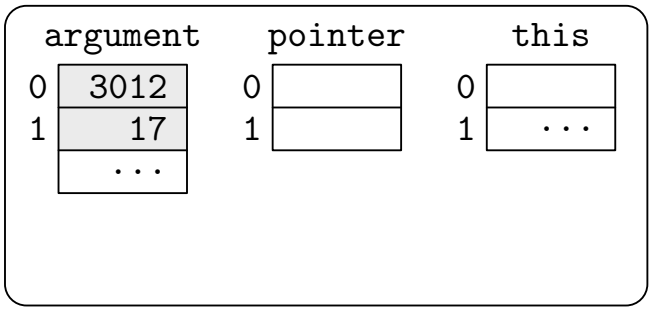

RAM view



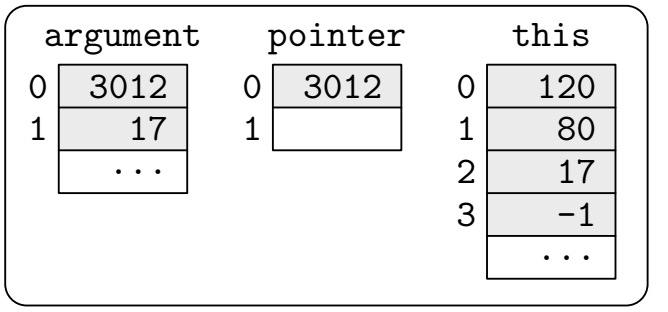

```
/* Assume that b and r
   were passed to the function
   as its first two arguments.
   The following code
   implements the operation
   b.radius = r. */

// get b's base address:
   push     argument 0
// point the this segment to b:
   pop      pointer 0
// get r's value
   push     argument 1
// set b's third field to r:
   pop      this 2
```

Actual RAM locations of program variables are run-time dependent and thus the addresses shown here are arbitrary examples.

Virtual memory segments just before the operation `b.radius = 17`:




Virtual memory segments just after the operation `b.radius = 17`:




`this 0` is now aligned with `RAM[3012]`

# Objekte: Methodenaufrufe

## Java-Quelltext

```
class Complex {
  // fields (properties):
  int re;                 real part
  int im;                 imaginary part
  …
  /** Constructs a new complex number */
  public Complex (int re, int im) {
    this.re = re;
    this.im = im;
  }
  …

class Foo {
  public void bla () {
    Complex x;
    …
    x = new Complex(1,2);
    x.mult(5);
    …
    …
  }
}
```

`x.mult(5)`

kann gesehen werden als

`mult(x,5)`

und wird daher übersetzt in

```
push x
push 5
call mult 2
```

Allgemein wird ein Methodenaufruf

`foo.bar(v1,v2,…)`

übersetzt in

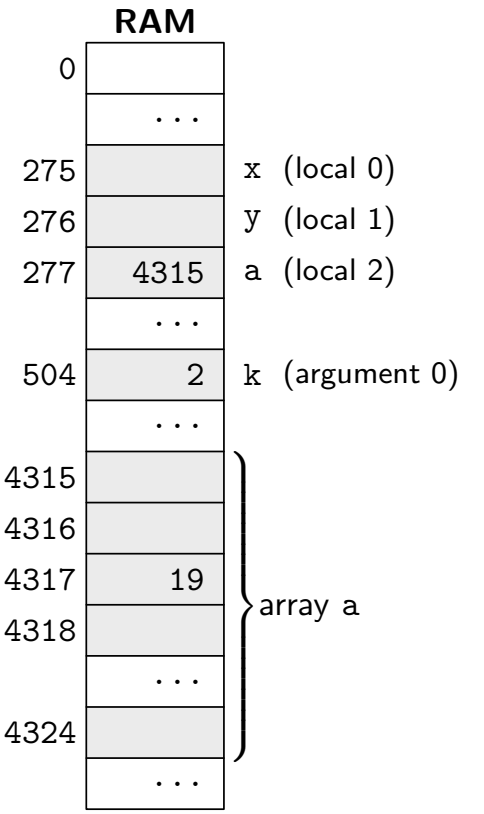
```
push foo
push v1
push v2
…
call bar n + 1
```

# Arrays: Speicheranforderung

**Java-Quelltext**

```
class Bla {
  …
  void foo (int k) {
    int x, y;
    int[] a;              // declare an ar
    …
    // constructs the array:
    a = new int[10];
    …
    a[k] = 19;
    …
  }
}
…
Main.foo(2);              // call the foo method
…
```

following compilation




Speicheranforderung:

`a = new int(n)` erzeugt `a = Memory.alloc(n);`

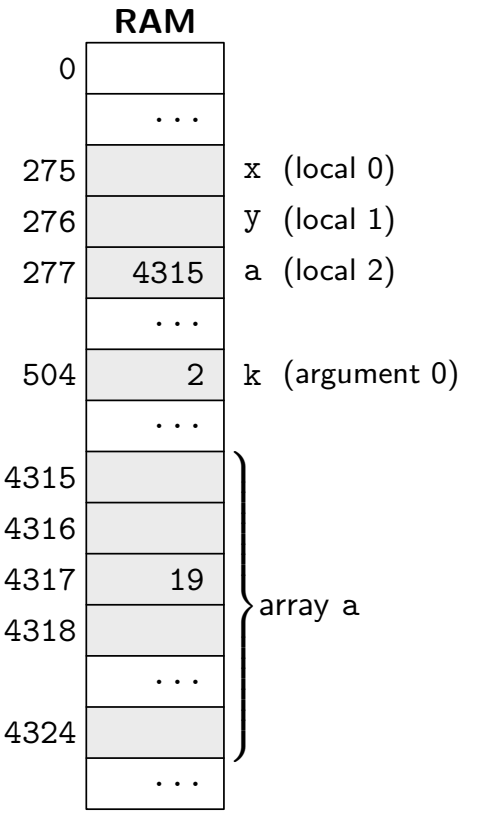
(wenn jede Speicherzelle eine Ganzzahl vom Typ `int` aufnehmen kann).

# Arrays: Speicherzugriff

## Java-Quelltext

```
class Bla {
  …
  void foo (int k) {
    int x, y;
    int[] a;              // declare an ar
    …
    // constructs the array:
    a = new int[10];
    …
    a[k] = 19;
    …
  }
}
…
Main.foo(2);              // call the foo method
…
```

following compilation




VM-Code (pseudo):
a[k] = 19

```
push a
push k
add
pop addr
push 19
pop *addr
```

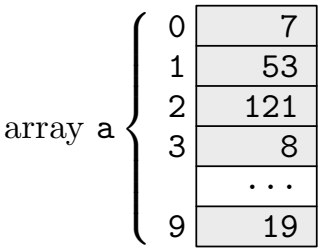
VM-Code (actual):
a[k] = 19

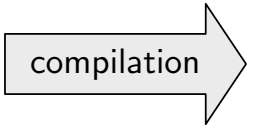
```
push local 2
push argument 0
add
pop pointer 1
push constant 19
pop that 0
```

# Arrays: Speicherzugriff 1

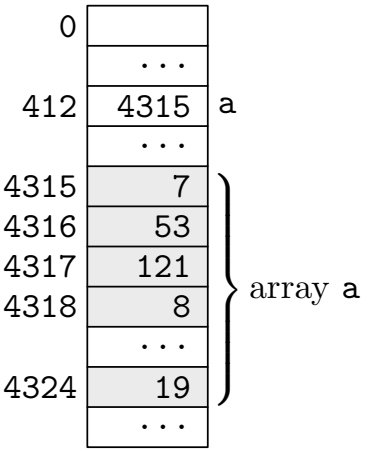
Hintergrund: Wir haben ein Array `a` vom Typ `int` und wollen `a[2]` auf den Wert 27 setzen.

high-level program view



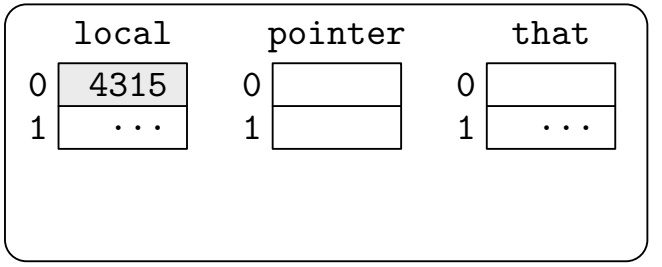

RAM view



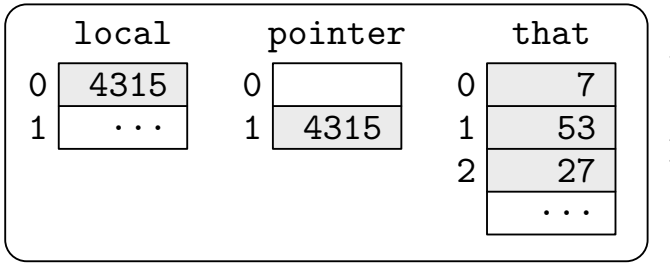

```
/* Assume that a is the
   first local variable
   declared in the program.
   The following code
   implements the operation
   a[2] = 27 or *(a+2) = 27 */

// get a's base address:
   push     local 0
// point the that segment to a:
   pop      pointer 1
// get value to store in a[2]:
   push     constant 27
// set a[2] to 27:
   pop      that 2
```

Actual RAM locations of program variables are run-time dependent and thus the addresses shown here are arbitrary examples.

Virtual memory segments just before the operation `a[2] = 27`:




Virtual memory segments just after the operation `a[2] = 27`:

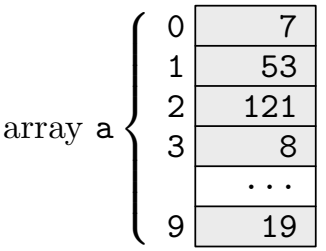


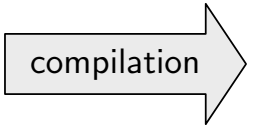

`that 0` is now aligned with `RAM[4315]`

# Array: Speicherzugriff 2

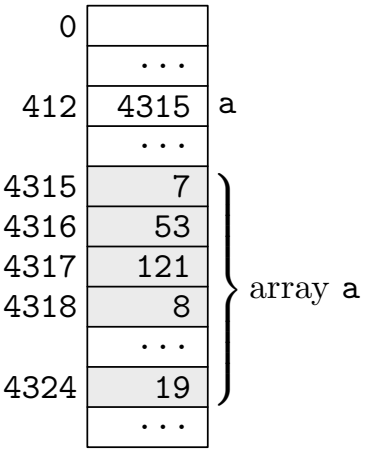
Hintergrund: Wir betrachten nun `a[i] = x`, wobei `i = 2` und `x = 31` die ersten beiden Funktionsargumente sind.

high-level program view



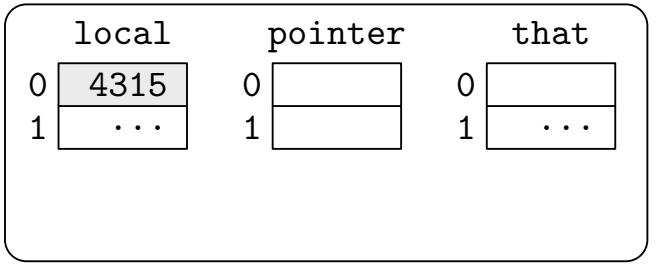

RAM view



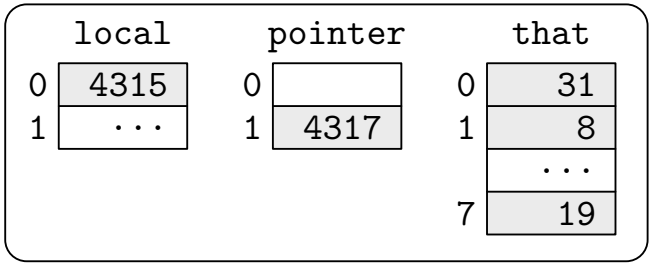

```
/* Assume that a is the
   first local variable and
   i and x are the first
   two function arguments.
   the following implements
   a[i] = x or *(a+i) = x */

// get a's base address:
   push     local 0
// add index i to a's address:
   push     argument 0
   add
// point that segment to a+i:
   pop      pointer 1
// get value to store in a[i]:
   push     argument 1
// set a[i] to x:
   pop      that 0
```

Actual RAM locations of program variables are run-time dependent and thus the addresses shown here are arbitrary examples.

Virtual memory segments just before the operation `a[i] = x`:

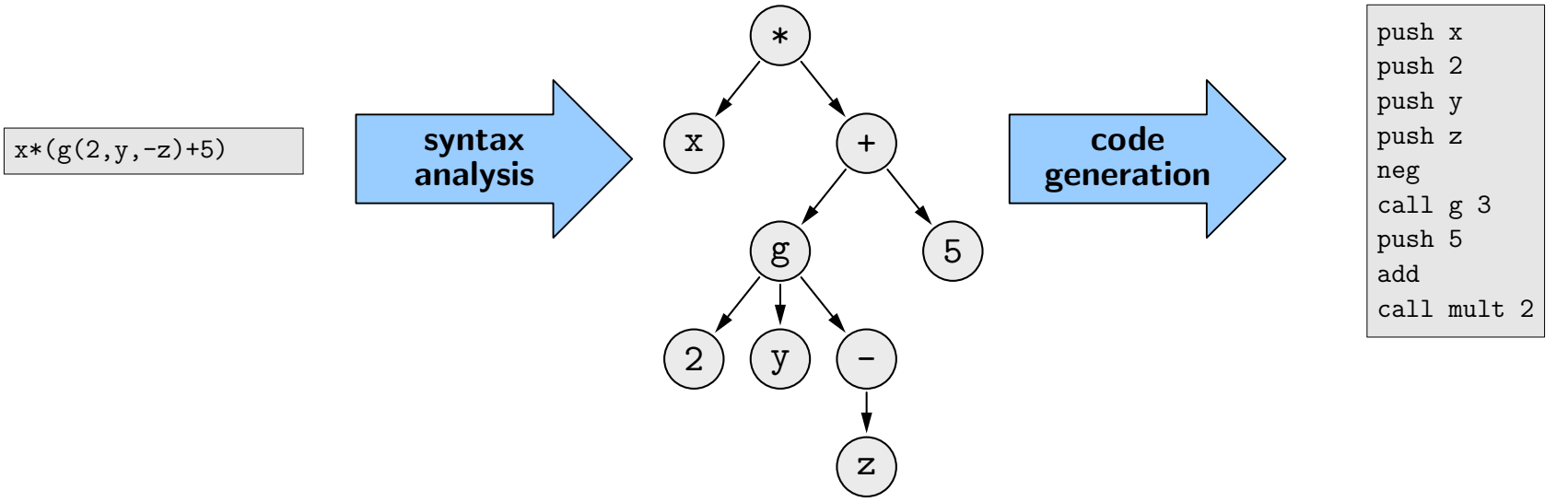



Virtual memory segments just after the operation `a[i] = x`:




`that 0` is now aligned with `RAM[4315+i]` (for `i = 2`)

# Behandlung von Ausdrücken



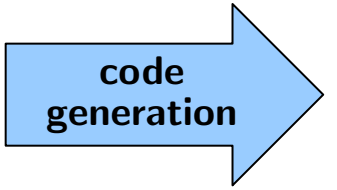

Algorithmus `codeWrite(`*expr*`)` zum Übersetzen von Ausdrücken:

| | | |
|---|---|---|
| if ***expr*** is a number $n$ | then | output "`push` $n$"; |
| if ***expr*** is a variable $v$ | then | output "`push` $v$"; |
| if $expr = (e_1$ op $e_2)$ | then | `codeWrite(`$e_1$`);` `codeWrite(`$e_2$`);` output "op" |
| if $expr =$ op$(e)$ | then | `codeWrite(`$e$`);` output "op" |
| if $expr = f(e_1, \ldots, e_n)$ | then | `codeWrite(`$e_1$`);` ...`codeWrite(`$e_n$`);` output "`call` $f$" |

# Programmablauf

## Ablauf in Java/Jack

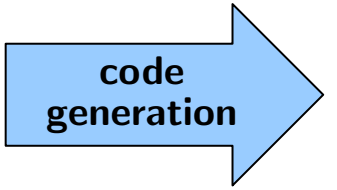
```
if (cond)
    statements 1
else
    statements 2
…
```




## VM Pseudo-Code

```
  VM code for computing ~(cond)
  if-goto L1
  VM code for executing statements 1
  goto L2
label L1
  VM code for executing statements 2
label L2
…
```

```
while (cond)
    statements 1
…
```




```
label L1
  VM code for computing ~(cond)
  if-goto L2
  VM code for executing statements 1
  goto L1
label L2
…
```

# Abschließendes Beispiel

```
class BankAccount {
  // class variables
  static int nAccounts;
  static int bankCommission;
  // account properties
  field int id;
  field String owner;
  field int balance;

  method void transfer (int sum, BankAccount from, Date when) {
    var int i, j;        // some local variables
    var Date due;        // Date is a user-defined type
    let balance = (balance + sum) - commission(sum * 5);
    // more code …
  }
```

```
function BankAccount.transfer
  // code for letting this point
  // to passed object (omitted)
  push balance
  push sum
  add
  push this
  push sum
  push 5
  call mult
  call commission
  sub
  pop balance
  // more code
```

Symboltabelle für Klasse

| Name | Type | Kind | # |
|---|---|---|---|
| nAccounts | int | static | 0 |
| bankCommission | int | static | 1 |
| id | int | field | 0 |
| owner | String | field | 1 |
| balance | int | field | 2 |

Symboltabelle für Methode

| Name | Type | Kind | # |
|---|---|---|---|
| this | BankAccount | argument | 0 |
| sum | int | argument | 1 |
| from | BankAccount | argument | 2 |
| when | Date | argument | 3 |
| i | int | local | 0 |
| j | int | local | 1 |
| due | Date | local | 2 |

```
function BankAccount.transfer
  push argument 0
  pop pointer 0
  push this 2
  push argument 1
  add
  push argument 0
  push argument 1
  push constant 5
  call Math.mult
  call Bankaccount.commission
  sub
  pop this 2
  // more code
```

# Zusammenfassung: Programmiersprachen und Compiler

- **Die Programmiersprache Jack**
  - Allgemeine Syntax
  - Datentypen und Speicheranforderung
  - Anweisungen, Ausdrücke und Funktionsaufrufe

- **Compiler** (speziell für die Jack-Programmiersprache)
  - Architektur, Lexikalische Analyse, Parsing
  - Kontextfreie Grammatiken
  - Parse-Bäume, Parsen durch rekursiven Abstieg
  - Jack-Grammatik
  - Jack-Syntaxanalyse
  - Codeerzeugung
  - Datenbehandlung, Speicherorganisation