

---

# Ein Compiler für Java

- Ein größeres Projekt ( $\approx$  4 Wochen)
- Übersetze (eine große Teilmenge von) Java nach C
  - Klassen, Objekte
  - Methoden
  - Interfaces
  - Anweisungen und Ausdrücke
- Nicht dabei
  - Innere Klassen
  - Garbage Collection
  - Kleinigkeiten (Felder in Interfaces u.ä.)
  - Reflection (Übungsaufgabe?)
  - Serialisierung
  - VM, Dynamisches Laden von Klassen

---

## Strategischer Aspekt

- Idee: Bedeutung von Programmen durch Übersetzung festlegen
  - Ein Java-Sprachkonstrukt  $K$  wird übersetzt in ein C-Konstrukt  $K'$
  - Wir wissen, was  $K'$  bedeutet (wenn wir C kennen)
  - Wir wissen dann, was  $K$  in Java bedeutet — dasselbe wie  $K'$  in C
- Übliche Technik zur Erklärung von Programmiersprachen
- Metapher: “Unbekanntes auf Bekanntes zurückführen”.

---

## Einwand: C als Voraussetzung?

- C ist eine kleine Sprache
  - C muß man als Informatiker sowie (gut) können
  - Ausdrücke und Anweisungen sehr ähnlich zu Java
  - Neu: Zeigerdatenstrukturen und Speicherverwaltung
- ⇒ Schwierig ist nur die maschinennahe Programmierung
- Wir besprechen die C Konstrukte, wenn sie zum ersten Mal vorkommen

---

## Ausgangspunkt: Java

- 1994/95
- Objekt-orientiert
- Keine Maschinendetails
- Abstraktionen:
  - Klasse
  - Objekt
  - Methodenaufruf
  - Interfaces
  - Speicherverwaltung
  - Datenrepräsentation
  - Speicheraufbau
  - Rechnerarchitektur

---

## Endpunkt: C

- 1970er
- Prozedural (imperativ)
- Maschinen-orientiert
- Abstraktionen:
  - Funktionsaufruf
  - Datenrepräsentation
- Ähnlichkeiten zu Java
  - Primitive Datentypen
  - Arithmetische Ausdrücke
  - Call-by-value Funktionsaufruf
  - Anweisungen

---

## Java → C

- C hat keine Unterstützung für Objekte — wir müssen alles selbst machen
  - C hat fast keine Einschränkungen — wir dürfen alles selbst machen
  - Grober Plan
    - Anweisungen → Anweisungen
    - Arithmetische Ausdrücke → Arithmetische Ausdrücke
    - Methoden → Funktionen
    - Methodenaufruf → Funktionsaufruf
    - Objekte → Structs (Records)
    - Klassen → Struct-Definitionen & Konstruktoren
- ⇒ Ähnlichkeiten konsequent ausnutzen
- ⇒ Gerade die OO-Konzepte in C ausdrücken
- ⇒ Java einschränken (und umdefinieren), um Übersetzung zu erleichtern

---

## Motivation

- Bei der Übersetzung müssen wir alle Entscheidungen und Berechnungen, die javac durchführt, im Detail nachvollziehen — wir können uns nicht um die Details drücken.
  - Wie funktioniert ein Methodenaufruf?
  - Was bedeutet `extends`?
  - Was bedeutet dagegen `implements`?
  - Wann wird Überladung, wann Überschreiben behandelt?
  - Wie implementiert man Interfaces?
- Da C keine OO-Unterstützung bietet, können wir diese vollständig selbst erkunden und implementieren.

---

# Motivation

- Im Compiler erkennen wir den Sinn von Einschränkungen der Sprache — die allgemeineren Fälle wären einfach nicht implementierbar.
  - Warum darf sich der Rückgabotyp einer überschriebenen Methode nicht ändern?
  - Warum kann ich eine Methode nicht über den Rückgabotyp überladen?
- Effizienzüberlegungen
  - Sind abstrakte Methoden teurer als normale Methoden?
  - Ist ein Aufruf einer Interface-Methode teurer als ein Aufruf einer Klassen-Methode?
  - Was kann der Compiler schon entscheiden, was kann erst zur Laufzeit passieren?
  - Was kostet ein Downcast?

---

## Einwand: Warum nicht für die JVM kompilieren?

- Die JVM (Java Virtual Machine) ist selbst objekt-orientiert
  - Klassen und Objekte
  - Beinhaltet einen Garbage Collector
  - Definiert sehr viele Einschränkungen auf dem erlaubten Code
  - Der ClassLoader lädt automatisch Klassen nach
  - Ist riesig und würde zur Erklärung allein 2-3 Doppelstunden brauchen  
→ “. . . auf Bekanntes zurückführen”!
- Der Behehlssatz beinhaltet beispielsweise
  - Methodenaufruf
  - Instanziierung
- Die JVM ist gerade so gemacht, daß die Übersetzung von Java leicht ist  
→ Man sieht nicht, wie die objekt-orientierten Konstrukte implementiert werden

---

# Aufbau des Compilers

- Die Übersetzung läuft in Phasen ab:
  1. Lexer & Parser
  2. Lexikalische Bindung
  3. Code-Rewriting: Syntaktischen Zucker auflösen
  4. Typcheck: Überprüfung von Korrektheitsbedingungen
  5. Layout: Festlegung der C Datenstrukturen
  6. Codegenerator: Erzeugung von C Funktionen und Definitionen
  7. gcc Aufrufe übersetzen die einzelnen Klassen
  8. gcc Aufruf erzeugt lauffähiges Programm (Linker)
- Folgerungen
  - Jedes Java-Sprachkonstrukt muss in allen Phasen behandelt werden
  - Die Phasen leisten einen inkrementellen Beitrag zum Gesamtergebnis
  - Der Code in jeder Phase für ein gegebenes Konstrukt ist übersichtlich

---

# Zwischenrepräsentation

- Softwarearchitektur folgt den Phasen (1 Phase  $\approx$  1 Modul)
  - Die meisten Module arbeiten auf derselben **Zwischenrepräsentation**:
    - **Annotierte Abstrakte Syntax (AAST)**
    - Lexikalischen Bindungen sind durch Zeiger repräsentiert
    - Elemente der AST in Gruppen eingeordnet
  - AAST zentral für das Verständnis
- ⇒ Hauptaufgabe für heute
- ⇒ Vertiefung auf dem Übungsblatt am Donnerstag

---

## Module des Compilers

- Lexer, Parser, Parsewrap erzeugen AST
- Annot Erzeugt AAST aus AST
- Rewrite schreibt die Eingabe um, damit spätere Phasen einfacher werden
- Props verwaltet Zusatzinformationen (Properties) zu AAST
- Tc (Typcheck) erweitert per Seiteneffekt AAST um Typinformationen
- Layout hält Laufzeit-Datenstrukturen in OCaml fest
- Cg (Codegenerator) erzeugt C-Programm durch Rekursion über AAST
- Backend erzeugt Aufrufe für gcc (compile, link)

---

## Modul: Parser, Lexer

- Erzeuge AST analog zu Mini (→ KvP-Teil)
- Unterschied: Anzahl der Nonterminale und der möglichen Fälle
- Trägermengen für Nonterminale = OCaml-Typen
  - Typen → `ty`
  - Literale → `literal`
  - Ausdrücke → `exp`, `exp_desc`
  - Anweisungen → `sm`, `sm_desc`
  - Deklarationen in Klassen → `class_decl`
  - Deklarationen in Interfaces → `interface_decl`
  - Dateiebene → `tydecl_desc`
- Ausgabefunktionen `print_...`
- Option `-ast` von Programm `jc` gibt AST aus

---

## Ausschnitt AST

```
type literal =
  Literal_int of int
  | Literal_float of float
  ...
type exp = {
  exp_desc : exp_desc;
  exp_pos  : Lexing.position }
and exp_desc =
  Exp_id of identifier
  | Exp_literal of literal
  | Exp_call of exp * identifier * exp list
type class_decl =
  Class_field of vardecl
  | Class_method of identifier * ... * sm option
and tydecl_desc =
  Type_class of identifier * ... * class_decl list
```

---

## Parser in OCaml

```
# #load "parseaux.cmo";;  
# #load "parser.cmo";;  
# #load "lexer.cmo";;  
# #load "parsewrap.cmo";;  
# Parsewrap.read_compilation_unit "Hello.java";;
```

Eingabe:

```
class Hello {  
    public static void main(String argv[]) {  
        System.out.println("Hello, world!");  
    }  
}
```

---

## Ergebnis

```
- : Ast.file =
Ast.File ("Hello.java", None, [],
  [{Ast.tydecl_desc =
    Ast.Type_class (Ast.Id "Hello", [], [], [],
      [Ast.Class_method (Ast.Id "main", [Ast.Mod_static; Ast.Mod_public],
        Ast.Ty_void,
        [Ast.Formal (Ast.Id "argv",
          Ast.Ty_array_ref (Ast.Qid (Ast.Id "String"), 1))],
        [],
        Some
          {Ast.sm_desc =
            Ast.Sm_block
              [Ast.Smb_sm
                {Ast.sm_desc =
```

---

```

Ast.Sm_exp
  {Ast.exp_desc =
    Ast.Exp_call
      ({Ast.exp_desc =
        Ast.Exp_access
          ({Ast.exp_desc = Ast.Exp_id (Ast.Id "System");
            Ast.exp_pos =
              {Lexing.pos_fname = "Hello.java";
                Lexing.pos_lnum = 3; Lexing.pos_bol = 59;
                Lexing.pos_cnum = 60}},
            Ast.Id "out");
          Ast.exp_pos = {Lexing.pos_fname ... }},
        Ast.Id "println",
        [{Ast.exp_desc =
          Ast.Exp_literal (Ast.Literal_string "Hello, world!");
          Ast.exp_pos = { ... }
        }]);
      Ast.exp_pos = {...}};
    Ast.sm_pos = {... }]);
  Ast.sm_pos = { ... }]);
Ast.tydecl_pos = { ... }])

```

---

## Ausgabe jc -ast Hello.java

```
===== Hello.java =====  
<<no package>>  
class (from Hello.java:1:0) Hello  
  extends ()  
  implements ()  
  {  
    static public void main(argv:String[])  
    {  
      ((System).out).println(<Hello, world!>);  
    }  
  }
```

- Genaue Ausgabe des AST als vollgeklammerter Text (→ Trägermenge i.S. der Abstrakten Syntax)
- Nicht vorhandene Elemente als leere Einträge

---

## Zwischenrepräsentation: AAST

- Motivation: Lexikalische Bindung
  - In Mini: Umgebung in Compiler mitführen
  - Für JC: Umgebung in jeder Compiler-Phase mitführen
- ⇒ Spätere Phasen müssen lexikalische Bindung immer wieder neu berechnen
- ⇒ Modul `Annot` berechnet lexikalische Bindung
- ⇒ Zwischenrepräsentation: Annotierte Abstrakte Syntax (AAST)
- Repräsentiert benötigte Typen (Klassen/Interfaces)
  - Lädt automatisch referenzierte Typen aus Dateisystem
  - Ersetzt Baumstruktur durch Records → leichter Zugriff
  - Ersetzt Variablen durch Referenzen auf ihre Bindung
  - Ersetzt Typnamen durch Referenzen in Typ-Liste

---

## Beispiel: Klassen im AAST

```
type cls = {
  cls_unique      : string;
  cls_name        : qidentifier;
  cls_mods        : modifier list;
  mutable cls_extends : cls; (** Object: self-loop *)
  mutable cls_implements : interface list;
  mutable cls_meths  : meth list;
  mutable cls_fields : field list;
  mutable cls_inits  : init list;
  mutable cls_ctors  : ctor list;
  mutable cls_refs   : ref_ty list;
  cls_pos           : Lexing.position;
  cls_ast           : Ast.tydecl;
  cls_file          : string; (* the source file *)
}
```

---

## Beispiel: Variablen und Bindung im AAST

```
type exp = {
  exp_desc : exp_desc;
  exp_pos  : Lexing.position }
type exp_desc =
  Exp_id of exp_id
  | ...
type exp_id =
  | Eid_field of field
  | Eid_local of local
  | ...
type local = {
  local_unique : string;
  local_mods  : modifier list;
  local_id    : identifier;
  local_ty    : ty;
  local_init  : exp option; }
```

---

## Variablen und Bindung im AAST

- Records repräsentieren Variablen (Feld, Parameter, lokale Variable)
- Referenzen auf Variablen werden Referenzen auf Record
- Gleiches Vorgehen für
  - Klassen (können für statischen Aufrufe referenziert werden)

```
type exp_id =
```

```
...
```

```
| Eid_cls of cls
```

- Referenztypen

```
type ref_ty =
```

```
| Rty_cls of cls
```

```
| Rty_interface of interface
```

---

## Ausdrücke und Anweisungen im AAST

- Struktur von Ausdrücken und Anweisungen in AAST analog zu AST

```
type exp_desc =  
    Exp_infix of operator * exp * exp  
  | Exp_access of exp * identifier  
  | Exp_call of exp * identifier * exp list  
  | ...  
type sm_desc =  
    Sm_exp of exp  
  | Sm_block of sm_block list  
  | Sm_if of exp * sm  
  | ...
```

- Änderung gegenüber AST nur in
  - Exp\_id of exp\_id für gebundene Variablen
  - Smb\_local of local list für Variablendeclaration

---

## AAST-Ausgabe

```
Class (Hello<03> from Hello.java:1:0): Hello()  
  extends <C:java.lang.Object(Object<01>)>  
  implements  
  Fields:  
  Methods:  
    static public <<void>> main(main<05>  
      (argv(argv<04>) : <C:java.lang.String(String<02>)>[]{{}})  
      {{{}}({{{}}({{{}}System<06>).out).println({{}}<Hello, world!>);  
    }
```

Constructors:

Info:

- Record-Felder von `c1s` werden ausgegeben
- Eindeutiger Namen `Hello<03>` (Records werden durchnummeriert)
- Referenzen auf Records → Ausgabe ihrer eindeutigen Namen
- `{{. . . }}` werden von Layout & Typcheck ausgefüllt

---

## Zwischenstand: AAST

- ✓ AST Knoten werden durch OCaml-Records ersetzt
  - Zugriff über Feldnamen statt `match`
  - Eingabereihenfolge irrelevant: Listen von ähnlichen Einträgen
  - Logische Struktur statt Struktur der abstrakten / konkreten Syntax
  
- ✓ AAST erfasst Bindungen durch Referenzen in OCaml
  - Leichtes Verfolgen von Referenzen / Bindungen
  - Spätere Phasen brauchen keine lexikalische Umgebung
  - Eindeutige Referenzen: Genau Record pro Klasse und Interface
  
- ✓ Ausdrücke und Anweisungen i.w. unbehandelt übernommen
  - Werden in späteren Phasen behandelt

---

## Berechnung der AAST

- Wie lexikalische Bindung in Mini
  - Primitive (strukturelle) Rekursion über AST
  - Mitführen einer Compilezeitumgebung mit
    - lokalen Variablen und Parametern
    - Feldern der aktuellen Klasse
    - Globale Umgebung (Symboltabelle) mit Klassen und Interfaces
- $\Delta$  Mini: `Annot` erweitert (durch Seiteneffekt) die globale Umgebung um neue Klassen erweitert, sobald diese referenziert werden
  - Für die Rekursion scheinen die Klassen schon immer in der Umgebung gewesen zu sein
  - Jede Klasse wird nur einmal geladen  $\rightarrow$  Alle Referenzen zeigen auf denselben Record
  - Änderungen am Record für alle Referenzen sichtbar

---

## Compilezeitumgebung für Annot

- Annot führt Rekursion über AST durch
- Funktionen für jeden Typ  $t$  in AST

$$\text{connect}_t : \text{Annot.env} \rightarrow \text{Ast.t} \rightarrow \text{Annot.t}$$

- Compilezeitumgebung env ist

```
type env = {  
  env_syntab    : syntab;  
  env_package  : qidentifier option;  
  env_imports  : qidentifier list; (* expanded *)  
  env_exp_ids  : (identifier * exp_id) list;  
  env_ret_ty   : ty;  
}
```

---

## Symboltabelle für Annot

- Symboltabelle symtab ist

```
type elem =  
  | Cls of cls  
  | Interface of interface
```

```
type symtab = {  
  mutable symtab_elems : elem list; (* processed entities *)  
  mutable symtab_files : string list; (* files already read *)  
}
```

- Neue Klassen werden durch Seiteneffekte eingefügt
- Bereits geladene Dateien werden mitgeführt

---

## Rekursionsbasis: Klasse / Interface in Typ

```
let rec connect_ty env = function
  | Ast.Ty_name qid -> Ty_ref(find_ty env qid)
  | Ast.Ty_array_ref(qid,dims) ->
    Ty_array_ref(find_ty env qid, dims)
  | ...

let find_ty env qid =
  try
    match qid with
      | Ast.Qid id -> find_unqualified_ty env id
      | Ast.Qid_suffix _ -> find_qualified_ty env qid
  with Not_found ->
    if not(load_file_containing_ty env qid)
    then raise (Undefined_ty qid)
```

---

## Rekursionsbasis: Klasse / Interface in Typ

- `find_unqualified_ty env ty` ergänzt `ty` um imports aus `env`
- `find_qualified_ty env` extrahiert nur `env.env_syntab`
- Eigentliche Arbeit in folgender Funktion:

```
let find_qualified_ty_in_syntab syntab qid =  
  match  
    find (function  
          Cls { cls_name = qid' } -> qid = qid'  
          | Interface { interface_name = qid' } -> qid = qid')  
      syntab.syntab_elems  
  with  
    Cls cls -> Rty_cls cls  
    | Interface i -> Rty_interface i
```

---

## Rekursionsbasis: Variable in Ausdruck

- connect\_exp erzeugt AAST aus AST Ausdruck

```
let rec connect_exp env =  
  fun exp ->  
    let desc = match exp.Ast.exp_desc with  
      Ast.Exp_id id -> Exp_id (find_exp_id env id)  
      | ...
```

- Die Funktion find\_exp\_id

```
let find_exp_id env id =  
  try assoc id env.env_exp_ids  
  with Not_found -> try  
    match find_ty env (Ast.Qid id) with  
      Rty_cls cls -> Eid_cls cls  
      | _ -> raise (Undefined_id(id,env))  
  with Undefined_ty _ -> raise (Undefined_id(id,env))
```

---

## Zwischenstand

- ✓ Übersicht über Compileraufbau
  - Zwischenrepräsentation AAST
  - Phasen und Module
  
- ✓ Tiefere Einsicht in AAST
  - Motivation
  - Struktur
  - Erzeugung
  
- ✗ Überblick Rewrite, Properties, Typcheck, Codegenerator, Backend