

---

## Was bisher geschah

- Details: Parser, Lexer, AST
- Zwischenrepräsentation: Annotierte Abstrakte Syntax
  - Knoten aus AST repräsentiert durch OCaml Records
  - Lexikalische Bindung aufgelöst durch Verweise
- Modul Annot
  - Erzeugung von AAST aus AST
  - Nachladen von Klassen aus Dateien

---

# Heute

- Programmieren mit der AAST
- Die übrigen Module des Compilers
- Einstieg: Objekte & Klassen
  - Konzeptsammlung
  - Anfang Übersetzung

---

## Erinnerung: 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 in 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;
  ...
}
```

---

## Beispiel: Methoden in AAST

```
type meth = {  
  meth_unique   : string;  
  meth_name     : identifier;  
  meth_mods     : modifier list;  
  meth_ret      : ty;  
  meth_formals  : formal list;  
  meth_throws   : ty list;  
  mutable meth_body : sm option;  
  ...  
}
```

---

# Programmieren mit der AAST

- AAST ist Zwischenrepräsentation im gesamten Compiler

⇒ Wir müssen mir ihr vertraut werden

- Beispielaufgabe: Zähle alle Methoden und ihre Parameteranzahl auf
- Beispielausgabe:

```
> ./all_methods -I ~/oopl-0506/compiler/  
    ~/oopl-0506/compiler/tests/Point.java  
Found moveBy<11> (2 params) in class Point<06>  
Found getX<12> (0 params) in class Point<06>  
Found getY<13> (0 params) in class Point<06>  
Found main<08> (1 params) in class Main<05>
```

---

## Schritt 1: Die Abstrakte Syntax

```
let treat_file f =  
  try  
    let ast = Parsewrap.read_compilation_unit f  
    ...  
  with e -> Annot.print_error e  
let opts = [  
  ("-print", Arg.Set do_print, "print AAST to stdout");  
  ("-I", Arg.String Annot.append_to_class_path,  
    "append directory to class path")  
]  
let main = Arg.parse opts treat_file "usage";
```

- Modul Parsewrap definiert Hilfsfunktionen für Lexer & Parser
- Funktion `read_compilation_unit` liest vollständige Java-Datei ein
- Modul `Arg` aus Standard-Bibliothek verarbeitet Kommandozeile
- `Arg.parse` ruft `treat_file` für jedes Argument auf

---

## Schritt 2: Die AAST erzeugen

```
let treat_file f =  
  ...  
  let ast = Parsewrap.read_compilation_unit f in  
  let symtab = Annot.empty_symtab () in  
  let decls = Annot.add_file symtab ast  
  ...
```

- Leere Symboltabelle anlegen
  - Die neu gelesenen Deklarationen in AST hinzufügen
- ⇒ Referenzen auf andere Klassen werden nachgeladen

---

## Schritt 3: Auf der AAST arbeiten

```
let decls = Annot.add_file symtab ast
...
Annot.print_elems Format.std_formatter
                    Annot.empty_print_hooks decls;
iter
  (fun (cls,meth) ->
    printf "Found %s (%i params) in class %s\n"
      meth.meth_unique
      (length meth.meth_formals)
      cls.cls_unique)
  (meths_of_tydecls decls)
```

- `meths_of_tydecls` liefert Paare (Klasse, Methode)
- Referenzen in die AAST
- Zugriff auf weitere Informationen möglich

---

## Schritt 4: Rekursion über die AAST

```
let meths_of_cls cls =  
  mapfilter (* behält alle Werte in Some, verwirft None *)  
    (fun m ->  
      if is_abstract_meth m  
      then None  
      else Some (cls,m)) (* Paar erzeugen *)  
  cls.cls_meths
```

```
let meths_of_tydecl = function  
  Interface _ -> []  
| Cls cls -> meths_of_cls cls
```

```
let meths_of_tydecls decls =  
  fold_left (* Sammeln von Deklarationen *)  
    (fun ms d -> meths_of_tydecl d @ ms)  
    [] decls
```

→ Üblicherweise eine Funktion pro Datentyp (Namen analog wählen)

---

## Module des Compilers

- Wir müssen jedes Sprachkonstrukt in alle Module des Compilers einbauen
- ⇒ Wir brauchen eine Übersicht über Module des Compilers

## Module des Compilers

Syntax	Konkrete → Abstrakte Syntax ✓
Annot	AST → AAST ✓
Rewrite	Umschreiben des AAST um spätere Phasen zu vereinfachen.
Tc	Typcheck: Werden Daten richtig verwendet?
Layout	Entscheidung über C-Datenstrukturen
Cg	Erzeugung von C-Code aus vorher generierten Informationen

---

## Modul Rewrite

- Java-Compiler muss Code generieren
    - `super()` Aufrufe in Konstruktoren einfügen
    - Default-Konstruktor anlegen wenn kein Konstruktor gegeben ist
    - Default-Initialisierung von Feldern
    - Feld-Initialisierung → Zuweisung in Konstruktoren
  - Änderungen in AST oder AAST?
    - AST ist unübersichtlich, weil abhängig von Eingabereihenfolge
    - AAST ist schwer zu erzeugen wegen Querverbindungen
- ⇒ Änderungen in AAST, aber Annot übernimmt die Verbindungen
- Erzeuge AST für gewünschten Code
  - Rufe Funktionen aus der Schnittstelle von Annot auf

---

## Modul Props

- Spätere Phasen fügen per Seiteneffekt Information zu AAST hinzu
- Diese Informationen beziehen sich auf AAST zurück

⇒ Wechselseitige Rekursion

- OCaml erlaubt keine wechselseitig rekursiven Datenstrukturen über Modulgrenzen hinweg
- Alternativen
  - ✗ Alle Daten in Modul Annot definieren, auch wenn sie erst viel später verwendet werden
  - ✓ Informationen zu AAST-Records in Tabellen speichern

---

## Modul Props

- Props kapselt Verwaltung von `Properties` zu AAST-Records ein
  - ✓ Dynamisch verwaltet
  - ✓ Typsicher
  - ✓ Wechselseitige Rekursion aufgelöst
  - ✗ Ineffizient (für unsere Anwendung irrelevant)
- Zugriffsfunktionen `set_p` und `p` pro Property `p`
- Beispiele: Ausgewählte Methode, interner Name einer lokalen Variable
  - `val set_exp_acc_meth : Annot.exp -> Annot.meth -> unit`  
`val exp_acc_meth : Annot.exp -> Annot.meth`
  - `val local_nm : Annot.local -> string`  
`val set_local_nm : Annot.local -> string -> unit`

---

## Modul Tc

- Das Typsystem von Java verhindert ungültige Berechnungen
  - Arithmetik funktioniert nur mit primitiven Zahltypen
  - Felder von Objekten müssen für Zugriff existieren
  - Methoden müssen für Aufruf existieren und die Argumente müssen zu den Methodenparametern passen
  - Überladenen Methoden: Welche Methode soll aufgerufen werden?
  - Konversionen
    - Upcast zu Super-Klasse und Interfaces
    - Downcast zu abgeleiteter Klasse
- Modul Tc berechnet Typinformation
  - Rückgabetypen von Ausdrücken
  - Benötigte Konversionen
  - Ausgewählte Methoden und Konstruktoren bei Überladung
  - Wird per Props zu AAST hinzugefügt

---

## Module Layout

- Entscheidet über den Aufbau von Datenstrukturen in C
- Repräsentiert C-Datenstrukturen als OCaml Daten
- Hauptarbeit für die objekt-orientierten Aspekte
  - Überschreiben von Methoden
  - Polymorphie
  - Interfaces
- Wir werden immer Ausschnitte in der Vorlesung behandeln

---

## Modul Cg

- Die Vorarbeit ist getan
  - Typchecker hat Typen, Konversionen, Methoden berechnet
  - Layout hat die Datenstrukturen auf der C-Seite definiert
  - AAST enthält den Code von Methoden, Anweisungen, Ausdrücken
- Codegenerator Cg erzeugt C-Code mit rekursivem Durchlauf durch AAST
- Ergebnis: Abstrakte Syntax von C
  - Modul C stellt Abstrakte Syntax und Ausgabe bereit
  - Modul Cg ist Funktion von Bäumen zu Bäumen
  - Vergleichbar Compiler für de-Bruijn-Interpreter von Mini

---

## Modul Backend

- Herausschreiben von C-AST ins Dateisystem
- Auswahl von Pfaden für die einzelnen Dateien
- Ansteuerung von C-Compiler gcc
- Verfolgen von Querreferenzen für Linking
- Entscheidung über Recompilierung
- Insgesamt: Technisch, nicht komplex

---

## Treiber Mainjc

- Auswertung der Kommandozeile
- Gesamtsteuerung der einzelnen Phasen
- Debugging-Ausgaben