



## SWP Übersetzerbau im SS 12

Informationen und Organisatorisches

Maximilian Konzack   Tim Rakowski  
Freie Universität Berlin

Einführungsveranstaltung am 12. April 2012

## Projektidee

### Vorstellung der Projekte

- Händisch erstellter Frontend
- Lexergenerator
- Parsergenerator
- Semantische Analyse
- Optimierungstechniken
- Maschinencodgenerierung

### Infrastruktur des Projekts

- Low Level Virtual Machine
- Repository

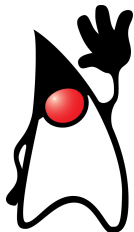
### Organisatorisches

- Treffen
- Zuteilung der Projekte

## Ende



- ▶ Arbeit an einem Übersetzer
- ▶ viele kleine Teilprojekte für Compilerkette
- ▶ Quellsprache ist imperativ und statisch typisiert
- ▶ Zwischencode kann von anderen Compilern genutzt werden



1. Kommunikation zwischen den Teilprojekten!
2. Modulare Komponenten für die Teilprojekte
3. LLVM als Zwischencode
4. Absprache von Schnittstellen
5. Implementierungssprache ist Java

## Aufgaben

1. Softwaredesign und -architektur für das Projekt
2. Implementierung eines Lexers, Parsers mit Codegenerierung mittels einfacher Übersetzerbautechniken
3. Verwaltung der Interfaces aller Phasen und ihrer Kontrolle (z.B. Token, Symboltabelle, abstrakter Syntaxbaum, ...)

## Gruppengröße

6 bis 8 Personen

## Aufgaben

1. Erzeugung eines funktionsfähigen Lexers
2. Eingabe für den Generator sind reguläre Definitionen
3. Tokenerkennung soll über endliche Automaten erfolgen

## Gruppengröße

2 bis 4 Personen

## Aufgaben

1. Erzeugung eines tabellengetriebenen LL(1)-Parsers
2. Eingabe für den Generator ist eine Grammatikdefinition in BNF
3. Erstellung eines konkreten Syntaxbaums (Parsebaums) zur weiteren Verarbeitung in Folgephasen

## Gruppengröße

2 bis 4 Personen

## Aufgaben

1. Alle Analysen dekorieren oder modifizieren den abstrakten Syntaxbaum
  - ▶ Typüberprüfung und -inferenz
  - ▶ Erzeugung aussagekräftiger Fehlermeldungen
  - ▶ Zwischengenerierung
  - ▶ Instrumentierung des Zwischencodes
2. Ausgabe ist LLVM-Zwischencode

## Gruppengröße

2 bis 4 Personen



## Aufgaben

1. Eingabe ist LLVM-Zwischencode
2. Ausgabe erfolgt als optimierter LLVM-Zwischencode
3. Teilaufgaben sind
  - ▶ Gemeinsame Teilausdrücke
  - ▶ Eliminierung von totem (unerreichbarem) Code
  - ▶ def-use-Ketten: Warning bei deklarierten, aber nicht verwendeten Variablen
  - ▶ weitere Datenflussanalysen

## Gruppengröße

2 bis 4 Personen

## Aufgaben

1. Eingabe ist LLVM-Zwischencode
2. Ausgabe erfolgt als ausführbarer Maschinencode
3. z.B. Java Byte Code, Intel Assembler, GNU AS, ...

## Gruppengröße

2 bis 4 Personen

## Ziele

- ▶ Unterstützung vielfältiger Compilertechniken
- ▶ stabiler Intermediate Code
- ▶ anpassbar für verschiedene Aufgaben

## Aufgaben

- ▶ verschiedenen Quellsprachen
- ▶ Zwischencodeerzeugung
- ▶ Optimierungen
- ▶ Debugging
- ▶ Statische Analysen
- ▶ ...



## Beispiel in C

```
int adder(int a, int b)
{
    int c = a + b;
    return c;
}

int main(void) {
    int sum = adder(5,7);
    printf("sum = %i\n",sum);
    printf("hello world\n");
    return 0;
}
```

```
@.str = private constant [10 x i8] c"sum = %i\0A\00"
@.str1 = private constant [13 x i8] c"hello world\0A\00"

define i32 @adder(i32 %a, i32 %b) nounwind {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %c = alloca i32, align 4
    store i32 %a, i32* %1, align 4
    store i32 %b, i32* %2, align 4
    %3 = load i32* %1, align 4
    %4 = load i32* %2, align 4
    %5 = add nsw i32 %3, %4
    store i32 %5, i32* %c, align 4
    %6 = load i32* %c, align 4
    ret i32 %6
}
```

## Umsetzung der main()-Funktion

```
define i32 @main() nounwind {
    %1 = alloca i32, align 4
    %sum = alloca i32, align 4
    store i32 0, i32* %1
    %2 = call i32 @adder(i32 5, i32 7)
    store i32 %2, i32* %sum, align 4
    %3 = load i32* %sum, align 4
    %4 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
    %5 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
    ret i32 0
}

declare i32 @printf(i8*, ...)
```

# Subversion vs. Git

## Treffen aller Teilnehmer

- ▶ wöchentlich
- ▶ donnerstags von 10 bis 12 Uhr c.t.
- ▶ jedes Team berichtet ca. 5 Min. über Status, Probleme, ...
- ▶ anschließend kurze Diskussion möglich (5 bis 10 Min.)
- ▶ bei zu vielen Fehlterminen wird Anwesenheitspflicht eingeführt!

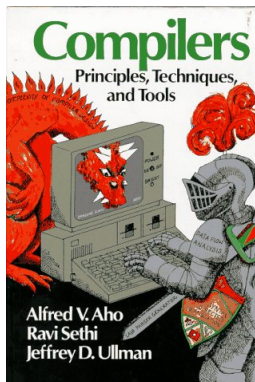
## Projekttreffen

1. interne Treffen eigenständig organisieren
2. regelmäßige Treffen mit Tim, Max und bei Bedarf andere Teams
3. Ansprechpartner für SWP Betreuer festlegen!
4. Agenda für Treffen vorbereiten oder besser an uns schicken



1. Verteilen der Listen
2. Maximal 2x Eintragen
3. Wünsche bitte über Anmerkung notieren
4. Zuteilung erfolgt heute!

# Danke für die Aufmerksamkeit.



## 1. Gruppeneinfindung






- ▶ interne Treffen organisieren
- ▶ Ansprechpartner für Tim und Max
- ▶ Erste Konzepte entwerfen: Interfaces, Beispielsyntax, ...
- ▶ Probleme festhalten
- ▶ Absprache mit anderen Teams?

## 2. Repository einrichten

## 3. Quellsprache verstehen

## 4. LLVM näher kennen lernen

## 5. Literatur konsultieren

-  Alfred V. Aho, Jeffrey Ullman, and Ravi Sethi.  
*Compiler: Prinzipien, Techniken und Werkzeuge.*  
Pearson Studium, 2. edition, 2008.
-  FindBugs – Find Bugs in Java Programs.  
<http://findbugs.sourceforge.net/>.
-  Chris Lattner and Vikram S. Adve.  
LLVM: A compilation framework for lifelong program analysis & transformation.  
In *CGO*, pages 75–88. IEEE Computer Society, 2004.
-  Chris Arthur Lattner.  
LLVM: An infrastructure for multi-stage optimization, 2002.
-  Michael Lee Scott.  
*Programming language pragmatics.*  
Morgan Kaufmann Publishers, 3. edition, 2009.