# Compilerprojekt
# Compilerbau WS 2022/2023

08.10. 2022

## 1 Termine

| | |
|---|---|
| Ende Freischuss - Parser und Typechecker : | 11. Dezember 2022 |
| Ende Frontend - Parser und Typechecker: | 18. Dezember 2022 |
| Ende Freischuss - Codegen und Liveness: | 22. Januar 2023 |
| Ende Backend - Codegen und Liveness: | 29. Januar 2023 |
| Klausur: | 16. Februar 2023 11:30 in 5E |
| Nachklausur: | 30. März 2023 11:30 in 5F |

## 2 Organisation und Aufgabenstellung

Aufgabe ist es einen Compiler (einer Teilmenge) von Go nach Jasmin Assembler Code zu schreiben. Die Semantik der Teilmenge wird ebenfalls **einige starke Vereinfachungen** zu Go aufweisen.

Die Abgabe des Projektes erfolgt in zwei Phasen. Die erste Phase beginnt mit Ausgabe dieses Dokuments und endet am **18. Dezember 2022**. In dieser Zeit müssen Sie einen Parser und Typechecker schreiben, der die in diesem Dokument beschriebene Sprache erkennt und die korrekte Typisierung prüft.

In der zweiten Phase (bis zum **29. Januar 2023**) müssen Sie einen Codegenerator schreiben, der aus dem Ergebnis Ihres Parsers Assemblercode für eine Stackmaschine generiert (Jasmin).

Ihr Assemblercode muss sich anschliessend mit dem Jasmin-Assembler in Java Bytecode übersetzen lassen. Außerdem ist eine (optionale) Liveness-Analyse Bestandteil der zweiten Phase. Die erfolgreiche Bearbeitung des Projektes ist **Voraussetzung für die Teilnahme an der Klausur**. Das Ergebnis des Compiler-Projektes geht **zu 40%** in die **Gesamtnote** ein.

### 2.1 Tools

Wir unterstützen den Einsatz von Java und ANTLR 4.

Andere Tools, Programmiersprachen und Bibliotheken sind **nur nach Rücksprache** gestattet, erhalten jedoch keinen Support von unserer Seite. **Verboten** ist ausdrücklich die Übersetzung des Go-Programms in eine andere Hochsprache und/oder Verwendung eines fremden Compilers zur Assembler-Erzeugung.

2 组织和任务
任务是将Go的编译器（一个子集）编写成Jasmine的汇编代码。子集的语义也会对围棋有一些强烈的简化。
项目的提交将分两个阶段进行。第一阶段从本文件发布时开始，到2022年12月18日结束，在此期间，
你需要编写一个解析器和类型检查器，以识别本文件中描述的语言并检查正确的类型。

在第二阶段（至2023年1月29日），你必须编写一个代码生成器，从你的解析器的结果中生成堆栈机（Jasmin）的汇编代码。
然后你的汇编代码必须能够使用Jasmin汇编器翻译成Ja-va字节码。此外，第二阶段还包括（可选）有效性分析。成功完成该项目是参加考试的先决条件。编译器项目的结果占总成绩的40%。

2.1 工具
我们支持使用Java和ANTLR 4。
其他工具、编程语言和库只有在协商后才允许使用，但不会得到我们方面的任何支持。明确禁止将围棋程序翻译成另一种高级语言和/或使用外国编译器生成汇编程序的行为。

Es ist nicht erlaubt das Quellprogramm (unsere Testcases) auszugeben. Überprüfen Sie hier bitte Ihren Compiler auf Debug-Ausgaben vor Ihrer Abgabe.

Der von Ihrem Backend generierte Assemblercode wird in Phase 2 mit dem Jasmin Assembler in Java Bytecode übersetzt und von uns ausgeführt.

## 2.2 Abgabe

Sie müssen Ihr Programm fristgerecht per Mail einreichen an:

*Lukas.Lang@uni-duesseldorf.de CC: John.Witulski@uni-duesseldorf.de Betreff: Abgabe CoBa Projekt Phase X - Name - MNR*

Abgegeben werden müssen (als zip oder tar.gz):

1. Phase 1: Die ANTLR Quelldateien und evtl. zusätzliche Sourcecodes. Eine Information, wie das Programm compiliert wird (Aufruf von ANTLR mit allen benötigten Parametern reicht). Bitte keine generierten Java-Dateien abgeben.
2. Phase 2: Alle Quelldateien (inkl. Ihrer ANTLR Grammatik), die nötig sind um Ihr Programm zu erzeugen. Eine Kurz-Anleitung, wie Ihr Programm compiliert wird.

Geben Sie in allen Abgaben die Version Ihrer Programmiersprache, von Jasmin und ANTLR mit an. In beiden Phasen gibt es eine Freischuss-Regel. Wer sein Programm vor dem Freischusstermin abgibt, bekommt die Ergebnisse unserer Testfälle und kann seine Abgabe bis zum endgültigen Abgabetermin überarbeiten.

## 2.3 Benutzerschnittstelle

Ihr Compiler soll von der Kommandozeile aufrufbar sein, die Syntax für den Aufruf des Compilers ist:

*java StupsCompiler -compile <Filename.go>*

Wenn das Programm keine Syntaxfehler enthält, wird eine Datei Filename.j erzeugt. Diese Datei muss dann mit Jasmin in Java Bytecode übersetzbar sein. Sollten Lexikalische-, Syntax- oder Typfehler auftreten, geben Sie auf der Standardausgabe eine aussagekräftige Nachricht aus. Im Falle von Fehlern, die von ANTLR generiert werden, muss die Nachricht **mindestens die Exception-Nachricht** beinhalten. Die Liveness-Analyse (Bonusaufgabe für 5 Punkte - maximal jedoch 40 im Projekt) soll durch das Kommando

*java StupsCompiler -liveness <Filename.go>*

gestartet werden. Als Ergebnis soll der Compiler die Mindestanzahl der benötigten Register ausgeben, wenn alle Variablen in Registern gehalten werden. Ihre Ausgabe muss in jedem Fall die Zeile:

*Registers: <Zahl>*

它不允许输出源程序（我们的测试案例）。在提交之前，请检查你的编译器是否有调试输出。
由你的后端生成的汇编代码将在第二阶段用Jasmin汇编器编译成Java字节码，并由我们执行。

2.2 提交
你必须按时将你的方案通过电子邮件提交给。
Lukas.Lang@uni-duesseldorf.de CC:
John.Witulski@uni-duesseldorf.de Betre˙: Submission
CoBa Project Phase X - Name - MNR

第一阶段：ANTLR源文件和任何额外的源代码。关于如何编译程序的信息（调用ANTLR并输入所有需要的参数即可）。请不要提交生成的Java文件。
第二阶段：生成程序所需的所有源文件（包括你的ANTLR语法）。关于如何编制你的方案的简要说明。

在所有提交的文件中包括你的编程语言、Jasmine和ANTLR的版本。这两个阶段都有一个自由得分规则。如果你在自由拍摄的截止日期前提交方案，你会收到我们的测试案例的结果，并可以修改你的提交，直到最后的截止日期。

2.3 用户界面
你的编译器应该可以从命令行中调用，调用编译器的语法是。
java StupsCompiler -compile <Filename.go>.
如果该程序不包含任何语法错误，就会创建一个文件Filename.j。
然后
这个文件必须可以用Jasmin翻译成Java字节码。如果发生词法、语法或类型错误，在标准输出上打印一条有意义的信息。
如果是由ANTLR产生的错误，该信息必须至少包含异常信息。
灵活性分析（5分的奖励任务--但在项目中最多为40分）要由命令来执行
java StupsCompiler -liveness <文件名.go>
应该启动。
因此，如果所有的变量都保存在寄存器中，编译器应该输出所需的最小数量的寄存器。它的输出在任何情况下都必须包含这一行。

enthalten. Die Angabe muss in einer eigenen Zeile erfolgen.

Falls der Kommandozeilenaufruf falsch eingegeben wurde (fehlende Parameter, etc.) schreiben Sie eine Nachricht mit der korrekten Aufrufsyntax auf die Standardausgabe. **Es darf nichts an dieser Aufrufsyntax modifiziert werden**, insbesondere darf der Compiler keine Fragen an den Benutzer stellen oder weitere Eingaben verlangen. Bei einem falschen Aufrufsyntax werden evtl. keine Testergebnisse an Sie versendet.

### 2.4 Bewertung

Wir testen Ihre Abgaben automatisiert. Wenn Sie nicht die korrekten Aufrufkonventionen verwenden, wird ihre Abgabe nicht als korrekt gewertet. Wir werden nicht Ihre Programme für Sie debuggen und modifizieren.

Folgenden Testcase können Sie beispielhaft verwenden:

```
package main

import "fmt"

func main() {
        fmt.Println(Fib(5))
        fmt.Println(Fib(10))
}

func Fib(n int) int {
        if n > 0 {
                if n <= 2 {
                        return 1
                } else {
                        return Fib(n-1) + Fib(n-2)
                }
        } else {
                return 0
        }
}
```

Hierbei haben die ersten Drei Zeilen in unserer Sprache keine Semantik, müssen jedoch vorhanden sein, damit ein Programm compiliert. Sinn dieser Regel ist es, dass Ihr Programm zum offiziellen Go Compiler kompatibel ist.

Achtung: Wir testen mit einer grossen Menge unterschiedlicher Testcases. Es ist keinesfalls ausreichend, wenn Ihr Compiler nur diesen Testfall korrekt übersetzt.

## 3 Sprachbeschreibung

Wir verwenden eine Untermenge von Go. Sie können den Go Compiler verwenden um die Syntax auszuprobieren. Ein Programm welches der Go Compiler nicht parst, ist auch hier ein Fehler. **Dies gilt nicht umgekehrt**. Wir verwenden nur aus diesem Grund die Syntax von Go. Die Semantik wurde für dieses Teilmenge etwas vereinfacht.

## 3.1 Programmaufbau

Jedes Programm hat die Form:

```
package main

import "fmt"

func main() {
        // more code ...
}

 // more methods ...
```

Beachten Sie, dass alle unsere Testfälle diesen Aufbau haben. Ein Compiler welcher dieses Minimalbeispiel nicht parst, wird keinen Testfall erfolgreich durchlaufen.

## 3.2 Kommentare

Folgende Kommentare sind möglich:

```
x = 42  // Kommentar bis zum Zeilenende
/* das ist ein
mehrzeiliger
Kommentar */
```

Geschachtelte Kommentare müssen nicht unterstützt werden.

## 3.3 Literale

Wir benötigen die Literale true, false, Flaotzahl, Integerzahlen und Stringkonstante.

| Name / Typ | Beispiel |
|---|---|
| int | 1, -1, 42 |
| float64 | 1.0, 3.12, -0.45 |
| string | "Hallo Welt" |
| bool | true, false |

Es müssen nur Floatwerte der Form x.y unterstützt werden (optionales Minus).
Ein Integer-Literal kann im Falle von Zuweisungen und Funktionsaufrufen auch ein Float-Literal sein. Beispiel **var f float = 42**

## 3.4 Operatoren

Es gibt die folgenden Operatoren:
Arithmetisch:

- + : Addition und unäres Plus (Bsp. 3+9 und +5)
- - : Subtraktion und unäres Minus (Bsp. 3-9 und -5)

- * : Multiplikation (Bsp. 5*2)
- / : Division (Bsp. 5 / 2)
- % : Modulo (Bsp. 5 % 2)

Vergleiche:

- == : Gleich
- < : Kleiner
- > : Grösser
- <= : Kleiner oder gleich
- >= : Grösser oder gleich
- != : Ungleich

Boolsche:

- && : logisches und
- || : logisches oder
- ! : logisches nicht

Es gelten die folgenden Operatorpräzedenzen (nach absteigender Präzedenz sortiert):

- Unäre Operatoren: !,+,-
- Multiplikative Operatoren: *, /, %
- Additive Operatoren: +,-
- Vergleichsoperatoren: !=, == <, >, <=, >=
- Logisches Und &&
- Logisches Oder ||

Die Operatoren sind linksassoziativ, z.B. x-y-z = (x-y)-z. Geklammerte Ausdrücke haben die höchste Präzedenz.

## 3.5 Ausdrücke

Ausdrücke können aus beliebig vielen Literalen, Operatoren und Funktionsaufrufen bestehen.

## 3.6 Variablen

Variablen werden in dieser Teilmenge von Go am Anfang einer Methode oder Klasse durch die Anweisung:

```
var Bezeichner Typ = Ausdruck
```

deklariert. Es gibt die Typen: int, float64, string, bool. Im Backend sollen Sie diese Typen mit den entsprechenden **primitiven** Java-Typen int, double und boolean sowie dem Objekt String identifizieren. Eine Deklaration ohne Definition ist nicht möglich (Vereinfachung). Die zulässigen Variablennamen sind die gleichen wie in Java (eingeschrängt auf ASCII). Die Testfälle testen nur solche Fälle wo float64 und Java-double identisch sind.

Beispiele:

```
var i int = 5-3
var f float64 = 1.0+2.14
var b bool    = true || !false
var s string = "Hallo"
```

## 3.7 Zuweisungen

Zuweisungen an Variablen erfolgen durch:

```
variable = Ausdruck
```

Variable und Ausdruck müssen den gleichen Typ haben. **Hinweis:** Auch Methodenaufrufe können Ausdrücke sein.

## 3.8 Block-Strukturen

Anweisungen können zu einem Block zusammengefasst werden:

```
{
    Anweisung1
    // ...
    AnweisungN
}
```

Ein solcher Block kann wie eine einzelne Anweisung betrachtet werden.

## 3.9 Kontrollstrukturen

### 3.9.1 If Else Blöcke

```
if BoolscherAusdruck { Anweisung1 } else { Anweisung2 }
```

Der else Teil kann weggelassen werden. Er ist optional.
**Hinweis:** Go kennt also kein Dangling-Else Problem.

### 3.9.2 For Schleifen

```
for BoolscherAusdruck { Anweisungen }
```

Hinweis: Wie unterstützen nur diese Variante der Go For-Schleife, welche sich wie eine While-Schleife verhält.

## 3.10 Funktionen

Funktionen haben in dieser Teilmenge von Go die Form:

```
func Bezeichner(Parameter0 Typ, ... , ParameterN Typ) Rueckgabetyp
{
        Deklarationen
        Anweisungen
        return Ausdruck
}
```

Parameter und Rückgabetyp sind optional. Ein return ist auch ohne Ausdruck möglich. Am Ende aller Kontrollflusspfade muss return stehen, wenn die Funktion einen Rückgabewert hat. Es gibt in unserer Teilmenge die Rückgabetypen int, float64, bool und string. Lokale Variablen einer Funktion können nur an deren Anfang deklariert werden (Vereinfachung).

Parameter sind hierbei eine Liste von Bezeichnern und Typen. Beispiel:

```
func MyAdd(a int, b int) int
{
        var c int = 0
        c = a+b
        return c
}
```

Beachten Sie hier die lokale Sichtbarkeit von Variablen.
Funktionsaufrufe enthalten immer Klammern. z.B MyAdd(5,7) oder f(). Der Rückgabewert kann verworfen werden
In Go gibt es keine überladenen Funktionen (gleicher Name, andere Parameter). Jeder Funktionsname kommt also maximal einmal vor.

## 3.11 Ausgabe

```
fmt.Println(Ausdruck)
```

Mit fmt.Println wird der Wert des Ausdrucks auf die Standardausgabe geschrieben, gefolgt von einem newline. Es ist nicht erlaubt den Inhalt von Testcases auszugeben (egal ob Quellcode oder ASTPrinter).

# 4 Typechecking

Diese Teilmenge von Go ist streng typsicher, jede Variable muss im Deklarationsteil einen Typ zugewiesen bekommen, der nicht mehr geändert werden kann.

Ihr Typchecker soll für alle Ausdrücke und Zuweisungen prüfen, ob die Typen korrekt sind. Es darf z.B. kein boolscher Wert in eine Integervariable geschrieben werden oder eine Anweisung wie if true > 9 { fmt.Println(0) } vorkommen.

Es gibt die Typen: int, float64, string und bool. Als Vereinfachung werden Variablen vor ihrer Nutzung am Anfang einer Methode deklariert.

# 5 Backend

## 5.1 Assembler-Sprache

Für den Typ String wird als Operation nur die Gleichheit und Ungleichheit implementiert. Dies ist kein Referenzvergleich, sondern ein Vergleich auf Stringgleichheit
Die Sprachsyntax der Jasmin Assemblersprache finden sie auf der Homepage des Tools: `http://jasmin.sourceforge.net/`

## 5.2 Bonusaufgabe: Liveness-Analyse

Sie können als Bonus eine Liveness-Analyse implementieren. Hiermit ist der Ausgleich von nicht bestandenen Testfällen möglich.

Dabei sollen die minimal nötigen Register ermittelt werden, wenn alle Variablen in Registern gehalten werden sollen. Sie müssen mindestens die in 2.3 geforderte Angabe machen, Wenn Sie den Register-Interferenz-Graph ausgeben, schreiben Sie eine kurze Erläuterung, wie die Ausgabe zu interpretieren ist in Ihre Dokumentation. Wir testen die Analyse für nur eine einzelne Funktion pro Programm. Es ist also keine Interprozedurale Analyse notwendig.

# 6 Referenzimplementierung

Die Syntax aller unserer Testcases wurde mit go version go1.18.1 linux/amd64 getestet. Alle Testcases erzeugen mit dieser Implementierung die selbe Ausgabe wie der hier zu entwickelnde Compiler. Siehe https://go.dev/ref/spec