Compilerprojekt
Compilerbau WS 2021/2022

05.11. 2021

1 Termine

Ende Freischuss - Parser und Typechecker :	12. Dezember 2021
Ende Frontend - Parser und Typechecker:	19. Dezember 2021
Ende Freischuss - Codegen und Liveness:	23. Januar 2022
Ende Backend - Codegen und Liveness:	30. Januar 2022
Klausur:	10. Februar 2022 8:30 in 6C
Nachklausur:	18. März 2022 8:30 in 5F

2 Organisation und Aufgabenstellung

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

Die Abgabe des Projektes erfolgt in zwei Phasen. Die erste Phase beginnt mit Ausgabe dieses Dokuments und endet am **19. Dezember 2021**. 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 **30. Januar 2022**) müssen Sie einen Codegenerator schreiben, der aus dem Ergebnis Ihres Parsers Assemblercode für eine Stackmaschine generiert.

Ihr Assemblercode muss sich anschliessend mit dem Jasmin-Assembler in Java Bytecode übersetzen lassen. Außerdem ist eine 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 empfehlen den Einsatz von Java und SableCC.

Andere Tools, Programmiersprachen und Bibliotheken sind **nur nach Rücksprache** gestattet. **Verboten** ist ausdrücklich die Übersetzung des C#-Programms in eine andere Hochsprache und/oder Verwendung eines fremden Compilers zur

Assembler-Erzeugung. Es ist nicht erlaubt das Quellprogramm (unsere Testcases) auszugeben. Der von Ihrem Backend generierte Assemblercode wird mit dem Jasmin Assembler in Java Bytecode übersetzt.

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 SableCC Quelldateien und evtl. zusätzliche Sourcecodes. Eine Information, wie das Programm compiliert wird (Aufruf von SableCC mit allen benötigten Parametern reicht). Bitte keine generierten Java-Dateien abgeben.

2. Phase 2: Alle Quelldateien (inkl. Ihrer SableCC Grammatik), die nötig sind um Ihr Programm zu erzeugen. Eine Kurz-Anleitung, wie Ihr Programm compiliert wird, sowie eine ausführbare Version Ihres Compilers.

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.cs>

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 SableCC 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.cs>

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>

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:

```
using System;

namespace FibonacciRecursive
{
    class Program
    {
        static void Main(string[] args)
        {
                Console.WriteLine(Fib(5));
                Console.WriteLine(Fib(10));
        }

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

Hierbei haben using, der Namespace und Klassen in unserer Sprache keine Semantik, müssen jedoch vorhanden sein, damit ein Programm Compiliert. Klassenname und NameSpacename sind hier bel. Bezeichner. Die main Methode muss ebenfalls diese Signatur haben, obwohl Arrays in unserer Teilmenge nicht enthalten sind. Dies alles hat den Sinn, dass Sie Programme einerseits mit dem C# Compiler testen können, andererseits diese Features aber selbst nicht implementieren müssen.
Achtung: Wir testen mit einer grossen Menge unterschiedlicher Testcases. Es ist keinesfalls ausreichend, wenn Ihr Compiler nur diesen Testcase korrekt übersetzt.

3 Sprachbeschreibung

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

3.1 Programmaufbau

Jedes Programm hat die Form:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
                // more code .
        }

         // more methods
    }
}
```

Beachten Sie, dass alle unsere Testcases diesen Aufbau haben. Ein Compiler welcher dieses Minimalbeispiel nicht parst, wird durch alle Testcases fallen.

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, Doublezahl, Integerzahlen und Stringkonstante.

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

Es müssen nur Doublewerte der Form x.y unterstützt werden (optionales Minus).

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: <, >, <=, >=
- Gleichheit !=, ==
- Logisches Und &&
- Logisches Oder ||

Die Operatoren sind linksassoziativ, z.B. x-y-z = (x-y)-z.

3.5 Ausdrücke

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

3.6 Variablen

Variablen werden in dieser Teilmenge von C# am Anfang einer Methode durch die Anweisung:

```
Typ Bezeichner = Ausdruck;
```

deklariert. Es gibt die Typen: int, double, 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 ist sowohl mit als auch ohne Initialisierung möglich. Die zulässigen Variablennamen sind die gleichen wie in Java (eingeschrängt auf ASCII).

Beispiele:

```
int i= 5-3;
double d = 1.0+2.14;
bool b = true || !false;
string s = "Hallo";
```

3.7 Zuweisungen

Zuweisungen an Variablen erfolgen durch:

```
variable = Ausdruck ;
```

Variable und Ausdruck müssen den gleichen Typ haben. Ausnahme: Umwandlung von int in double. 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.
Achtung: Handelt es sich um einzelne Anweisung, sind keine geschweiften Klammern nötig.

3.9.2 While Schleifen

```
while(BoolscherAusdruck) { Anweisungen }
```

Achtung: Handelt es sich um einzelne Anweisung, sind keine geschweiften Klammern nötig.

3.10 Statische Methoden

Statische Methoden haben in dieser Teilmenge von C# die Form:

```
static Type Bezeichner(Parameter)
{
        Deklarationen
        Anweisungen
        return Ausdruck
}
```

Alle unsere Methoden sind statisch. Parameter sind optional. Ein return ohne Ausdruck ist möglich. Es gibt in unserer Teilmenge die Rückgabetypen void, int, double, bool und string. Lokale Variablen einer statischen Methode können nur an deren Anfang deklariert werden.

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

```
static int MyAdd(int a, int b)
{
        int c;
        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
Als Vereinfachung testen wir in unserer Teilmenge keine überladened Funktionen (gleicher Name, andere Parameter). Jeder Funktionsname kommt also maximal einmal vor.

3.11 Ausgabe

```
Console.WriteLine(Ausdruck)
```

Mit Console.WriteLine 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 C# 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) Console.WriteLine(0); vorkommen.

Es gibt die Typen: int, double, string und bool. Als Vereinfachung werden Variablen vor ihrer Nutzung am Anfang einer Methode deklariert. Ein impliziter Cast zwischen double und int ist bei Zuweisungen oder bei Methodenaufrufen möglich.

5 Backend

5.1 Assembler-Sprache

Bei Operationen zwischen Integer und Double Operanden wird eine implizite erweiternde Typkonvertierung durchgeführt. Der Jasmin Bytecode hierfür ist i2d. 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 Graphen 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 der Microsoft (VS 2019) C# Implementierung getestet. Alle Testcases erzeugen mit dieser Implementierung die selbe Ausgabe wie der hier zu entwickelnde Compiler.