Compilerprojekt
Compilerbau WS 2023/2024

24.10.2023

1 Termine

Ende Freischuss - Parser und Typechecker :	10. Dezember 2023
Ende Frontend - Parser und Typechecker:	17. Dezember 2023
Ende Freischuss - Codegen und Liveness:	21. Januar 2024
Ende Backend - Codegen und Liveness:	28. Januar 2024
Klausur:	15. Februar 2024 11:30 in 5A
Nachklausur:	27. März 2024 11:30 in 5A

2 Organisation und Aufgabenstellung

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

Die Abgabe des Projektes erfolgt in zwei Phasen. Die erste Phase beginnt mit Ausgabe dieses Dokuments und endet am **17. Dezember 2023**. In dieser Zeit müssen Sie einen Parser und Typechecker schreiben, der die in diesem Dokument beschriebene Sprache erkennt und die korrekte Typisierung prüft. Bei der Verwendung von Tools, ist es Ihre Aufgabe selbstständig eine Grammatik zu entwickeln und abzugeben.

In der zweiten Phase (bis zum **28. Januar 2024**) 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 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 sowie von SableCC.

Andere Tools, Programmiersprachen und Bibliotheken sind **nur nach Rücksprache** gestattet, erhalten jedoch keinen Support von unserer Seite. **Verboten**

ist ausdrücklich die Übersetzung des Julia-Programms in eine andere Hochsprache und/oder Verwendung eines fremden Compilers zur Assembler-Erzeugung. Auch die Nutzung von Grammatiken anderer Personen und der Einsatz von KI-Tools ist untersagt. 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: Ihre ANTLR 4 / SableCC Grammatik und zusätzliche Sourcecodes. Eine Information, wie das Programm compiliert wird. Bitte keine generierten Java-Dateien oder JARs abgeben.
2. Phase 2: Alle Quelldateien (inkl. Ihrer ANTLR 4 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 einmalig 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.jl>

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 soll durch das Kommando

java StupsCompiler -liveness <Filename.jl>

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:

```
function main()
    a::Integer = 1
        b::Integer = 1
        temp::Integer = 0
        while a < 144
            temp = b
                b = a + b
                a = temp
                println(a)
        end
end

main()
```

Hierbei hat die letzte Zeile in unserer Sprache keine Semantik, muss jedoch vorhanden sein, damit ein Programm compiliert. Sinn dieser Regel ist es, dass Ihr Programm zur offiziellen Julia Implementierung kompatibel ist. Es ist der einzige Top-Level Aufruf.

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 Julia. Sie können die offizielle Julia Implementierung verwenden um die Syntax auszuprobieren. Ein Programm welches Julia nicht parst, ist auch hier ein Fehler. **Dies gilt nicht umgekehrt**. Wir verwenden nur aus diesem Grund die Syntax von Julia. Die Semantik wurde für dieses Teilmenge etwas vereinfacht. Insbesondere beim Thema Typisierung.

3.1 Programmaufbau

Jedes Programm hat die Form:

```
function main()
    # more code
end

# more functions ...
main()
```

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, Floatzahl, Integerzahl und Stringkonstante.

Name / Typ	Beispiel
Integer	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 auch ein Float-Literal sein. Beispiel **celsius::Float64 = 20**

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 ||

Logische Operatoren und unäre Operatoren sind rechtsassoziativ. Alle anderen Operatoren sind linksassoziativ, z.B. x-y-z = (x-y)-z. Geklammerte Ausdrücke haben die höchste Präzedenz.

Für den Typ String wird als Operation nur die Zuweisung, Gleichheit und Ungleichheit implementiert. Dies ist kein Referenzvergleich, sondern ein Vergleich auf Stringgleichheit

3.5 Ausdrücke

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

3.6 Variablen

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

```
Bezeichner::Typ = Ausdruck
```

deklariert. Es gibt die Typen: Integer, Float64, String, Bool. Im Backend sollen Sie diese Typen mit den entsprechenden **primitiven** Java-Typen int, double und boolean sowie dem Objekt String identifizieren. Dies ist eine Vereinfachung: Es gibt keine Testscases wo diese Abbildung nicht funktioniert. Insbesondere bilden wir den Julia Typ Integer auf den 32-Bit Java int Typ ab.

Eine Deklaration ohne Definition ist nicht möglich (Vereinfachung). Die zulässigen Variablennamen sind die gleichen wie in Java (eingeschrängt auf ASCII).

Beispiele:

```
i::Integer = 5-3
f::Float64 = 1.0+2.14
b::Bool = true || !false
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:

```
begin
    # Anweisung1
    # ...
    # AnweisungN
end
```

3.9 Kontrollstrukturen

3.9.1 If Else Blöcke

```
if BoolscherAusdruck Anweisung1 else Anweisung2 end
```

Der else Teil kann weggelassen werden. Er ist optional.

3.9.2 While Schleifen

```
while BoolscherAusdruck Anweisungen end
```

3.10 Funktionen

Funktionen haben in dieser Teilmenge von Julia die Form:

```
function Bezeichner(Parameter0::Typ, ... , ParameterN::Typ)::Rueckgabetyp
        Deklarationen
        Anweisungen
        return Ausdruck
end
```

Parameter 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 Integer, 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:

```
function MyAdd(a::Integer, b::Integer)::Integer
        c::Integer = 0
        c = a+b
        return c
end
```

Beachten Sie hier die lokale Sichtbarkeit von Variablen.
Funktionsaufrufe enthalten immer Klammern. z.B MyAdd(5,7) oder f(). Der Rückgabewert kann verworfen werden

3.11 Ausgabe

```
println(Ausdruck)
```

Mit 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 Julia 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+false > "Halloprintln(0) end vorkommen.

Es gibt die Typen: Integer, Float64, String und Bool. Als Vereinfachung werden Variablen vor ihrer Nutzung am Anfang einer Methode deklariert.

5 Backend

5.1 Assembler-Sprache

Die Sprachsyntax der Jasmin Assemblersprache finden sie auf der Homepage des Tools: `http://jasmin.sourceforge.net/`

5.2 Liveness-Analyse

Für die Liveness-Analyse soll 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 Abgabe-E-Mail. 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 julia version 1.5.2 getestet. Alle Testcases erzeugen mit dieser Implementierung die selbe Ausgabe wie der hier zu entwickelnde Compiler. Siehe https://docs.julialang.org/en/v1/