

Compilerprojekt

Softwaretechnik und Programmiersprachen

15.6.2011

1 Termine

Ende Freischuss Parser:	1. Juli 2011
Ende Parser:	15. Juli 2011
Ende Freischuss Backend:	1. August 2011
Ende Backend:	15. August 2011
Letzter Anmeldetermin Prüfung:	26. August 2011
Klausur:	23. September 2011

2 Organisation und Aufgabenstellung

Aufgabe ist es einen Compiler (einer Teilmenge) von Pascal nach Jasmin Assembler Code zu schreiben.

Die Abgabe des Projektes erfolgt in zwei Phasen. Die erste Phase beginnt mit Ausgabe dieses Dokuments und endet am 15. Juli 2011. 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 15. August 2011) 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 Bytecode übersetzen lassen. Außerdem ist eine Liveness-Analyse Bestandteil der zweiten Phase. Die erfolgreiche Bearbeitung des Projektes ist **Voraussetzung für die Klausur**, das Ergebnis des Compiler-Projektes geht auch in die **Gesamtnote** ein.

2.1 Tools

Wir bieten Support für JavaCC (+jjTree), SableCC und Prolog.

Andere Tools und Bibliotheken sind nur nach Rücksprache gestattet. **Verboten** ist ausdrücklich die Übersetzung des Pascal-Programms in eine andere Hochsprache und Verwendung eines fremden Compilers zur Assembler-Erzeugung. Der von Ihrem Backend generierte Assemblercode wird mit Jasmin in Bytecode übersetzt.

2.2 Abgabe

Sie müssen Ihr Programm fristgerecht per Mail einreichen an:

John.Witulski@uni-duesseldorf.de

Die Abgaben (als zip oder tar.gz) müssen folgendes enthalten:

1. Phase 1: Die JavaCC/SableCC/Prolog Quelldateien und evtl. zusätzliche Sourcecodes. Eine Information, wie das Programm mit compiliert wird (Aufruf von JavaCC/SableCC/Prolog mit allen benötigten Parametern reicht). Bitte keine generierten Java-Dateien abgeben.
2. Phase 2: Alle Sourcecodes (inkl. Ihrer JavaCC/SableCC Grammatik bzw Prolog Quellcode), die nötig sind um Ihr Programm zu erzeugen. Eine Kurz-Anleitung, wie Ihr Programm compiliert wird, sowie eine Binärversion 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.pas>  
prolog -s StupsCompiler.pl —compille <Filename.pas>
```

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 Standard-Fehlerausgabe eine aussagekräftige Nachricht aus. Im Falle von Fehlern, die von JavaCC/SableCC generiert werden, muss die Nachricht mindestens die Exception-Message beinhalten. Die Liveness-Analyse soll durch das Kommando

```
java StupsCompiler -liveness <Filename.pas>  
prolog -s StupsCompiler.pl —liveness <Filename.pas>
```

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.

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:

```
program fibonacci;  
var a: integer;  
var b: integer;  
var temp: integer;  
begin  
a := 1;  
b := 1;  
while True do begin  
  writeln(a);  
  temp := b;  
  b := a + b;  
  a := temp;  
end  
end.
```

Achtung: Wir testen mit ca. 80-100 unterschiedlichen Testcases. Es ist keinesfalls ausreichend, wenn Ihr Compiler diesen einen Testcase korrekt übersetzt.

3 Sprachbeschreibung

Wir verwenden eine Untermenge von Pascal. Sie können den GNU Pascal Compiler verwenden um die Syntax auszuprobieren.

3.1 Literale

Wir benötigen die Literale `True`, `False` und Integerzahlen.

3.2 Das Semicolon

Das Semikolon wird nicht wie in Java als Befehls-Abschluss interpretiert, sondern als Trennzeichen zwischen Anweisungen. Vor einem `end` kann es somit weggelassen werden. Vor einem `else` darf es gar nicht stehen, da sonst der `if`-Zweig als abgeschlossen angesehen werden würde.

3.3 Kommentare

In Pascal wird folgendermassen kommentiert:

```
x:=77; // Kommentar bis zum Zeilenende  
{ das ist ein  
  mehrzeiliger  
  Kommentar }
```

Geschachtelte Kommentare sind in Pascal nicht erlaubt und müssen daher nicht gesondert behandelt werden.

3.4 Programmaufbau

Jedes Programm hat die Form:

```
program Name;  
Deklarationen  
begin  
    Anweisungen  
end.
```

3.5 Variablen

Variablen werden in Pascal im Deklarationsteil durch die Anweisung:

```
var Bezeichnerliste :Typ
```

deklariert. Die Bezeichner in der Bezeichnerliste werden durch Kommata voneinander getrennt. Es gibt zwei Typen: **integer** und **boolean**. Im Backend sollen Sie diese beiden Typen mit den entsprechenden Java-Typen `int` und `boolean` identifizieren. Gross- und Kleinschreibung werden nicht unterschieden, d.h. `VAR1`, `vaR1` und `var1` bezeichnen die gleiche Variable. Die zulässigen Variablennamen sind die gleichen wie in Java.

3.6 Zuweisung

Zuweisungen an Variablen erfolgen durch

```
variable := Ausdruck
```

Variable und Ausdruck müssen den gleichen Typ haben.

3.7 Block-Strukturen

Anweisungen können zu einem Block zusammengefasst werden:

```
begin  
    Anweisung1 ;  
    Anweisung2  
end
```

Ein solcher Block kann wie eine einzelne Anweisung betrachtet werden.

3.8 Kontrollstrukturen

3.8.1 if then else

```
if BoolscherAusdruck then Anweisung1 else Anweisung2
```

Der else Teil kann weggelassen werden.

3.8.2 while do

```
while BoolscherAusdruck do Anweisung
```

Die Schleife kann auch durch `break` verlassen werden.

3.9 Ausgabe

writeln (Ausdruck)

Mit `writeln` wird der Wert des Ausdrucks auf die Standardausgabe geschrieben, gefolgt von einem `newline`.

3.10 Operatoren

Arithmetisch:

- `+`: Addition und unäres Plus (Bsp. `3+9` und `x:= +5`)
- `-`: Subtraktion und unäres Minus (Bsp. `3-9` und `x:= -5`)
- `*`: Multiplikation (Bsp. `x:= 5*2`)
- `div`: Integerdivision (Bsp. `x:= 5 div 2`)
- `mod`: Modulo (Bsp. `x:= 5 mod 2`)

Vergleiche:

- `=`: Gleich
- `<`: Kleiner
- `>`: Größer
- `<=`: Kleiner oder gleich
- `>=`: Größer oder gleich
- `<>`: Ungleich

Boolsche:

- `and`: logisches und
- `or`: logisches oder
- `xor`: logisches entweder oder
- `not`: logisches nicht

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

- unäre Operatoren `+, -, not`
- Multiplikative Operatoren: `*, div, mod, and`
- Additive Operatoren: `+, -, or`
- Vergleichsoperatoren

Die Operatoren sind linksassoziativ, d.h. $x-y-z = (x-y)-z$. Als Folge der Präzedenz wird z.B. die Anweisung

```
if 2=2 and not 2<>2 then writeln(1);
```

als

```
if (2=(2 and (not 2)))<>2 then writeln(1);
```

geparsed und führt u.a. zu einem Typfehler im Ausdruck (not 2). Es muss also folgendermaßen geklammert werden:

```
if (2=2) and not(2<>2) then writeln(1);
```

4 Typechecking

Pascal 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 boolescher Wert in eine Integervariable geschrieben werden oder eine Anweisung wie `if True > 9 then writeln(0);` vorkommen.

5 Backend

5.1 Liveness-Analyse

Sie müssen eine Liveness-Analyse implementieren. Dabei sollen die minimal nötigen Register ermittelt werden, wenn alle Variablen in Registern gehalten werden sollen. Sie müssen mindestens die in 3.2 geforderte Angabe machen, Für die Darstellung des Abhängigkeitsgraphen in einer Text-Form (Adjazenzmatrix, Adjazenzliste, o.ä.) gibt es Bonuspunkte. Wenn sie den Graphen ausgeben, schreiben Sie eine kurze Erläuterung, wie die Ausgabe zu interpretieren ist in Ihre Dokumentation.

5.2 Assembler-Sprache

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