Grundlagen der Informatik III

Wintersemester 2010/2011

Wolfgang Heenes, Patrik Schmittat

TECHNISCHE UNIVERSITÄT DARMSTADT

10. Aufgabenblatt

24.01.2011

Hinweis: Der Schnelltest und die Aufgaben sollen in den Übungsgruppen bearbeitet werden. Die Hausaufgaben sind in der Kalenderwoche 5 (31.01. bis 04.02.) bei den Tutoren in **physikalischer Form** (handschriftlich oder gedruckt) abzugeben. Bei allen Abgaben ist der Name des Tutors und die Übungsgruppe deutlich anzugeben. Bei Teamabgaben wird nur eine Lösung eingereicht, die alle Namen der Teammitglieder enthält.

Schicken Sie Ihre Lösungen von Programmieraufgaben zusätzlich zur schriftlichen Abgabe per E-Mail an Ihren Tutor. Kommentieren Sie Ihren Quellcode.

Aufgabe 1: Schnelltest

Fragen	Antworten
1. Die Syntaxdarstellung innerhalb einer Analysephase basiert meistens auf folgenden Sprachen:	☐ reguläre Sprache ☐ kontextfreie Grammatik ☐ Backus-Naur-Form ☐ Wörterbuch ☐ Mengenaufzählung
2. Welche Aussagen über Parse-Bäume sind korrekt?	☐ Der Knoten **A** besitzt soviele Nachfolger wie Regeln in der Form `A -> ...` existieren. ☐ Wenn ein String von Terminalen nicht abgeleitet werden können nimmt der Parser diese Folge in die Regeln mit auf. ☐ Die Wurzel enthält das Startsymbol der gegebenen Grammatik. ☐ Ein Knoten ohne Nachfolger kann das ϵ-Symbol enthalten. ☐ Terminal können auch in inneren Knoten auftreten.
3. Optimierende Compiler können Programmtransformationen vornehmen und logische Ausdrücke auswerten?	☐ Ja ☐ Nein ☐ Vielleicht
4. Bei der Benutzung von Gleitkommazahlen wird bei modernen Prozessoren und Compilern automatisch die SSE-Einheit verwendet.	☐ Ja ☐ Nein ☐ Vielleicht

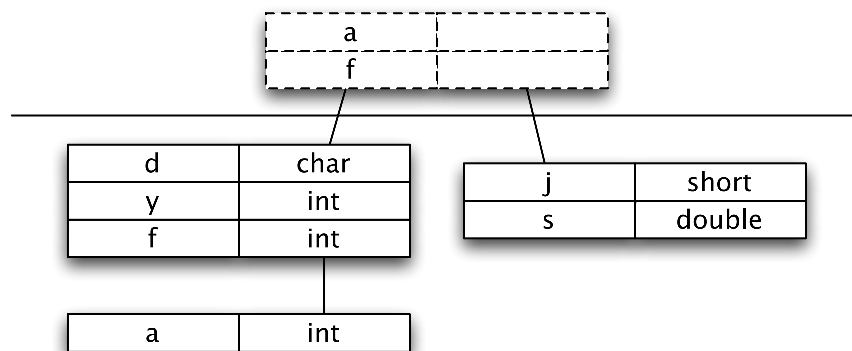
Aufgabe 2: Präzedenz und Assoziativität

Gegeben ist folgende Liste von Operatoren auf ganze Zahlen. Bilden Sie eine kontextfreie Grammatik, die diese Operatoren, deren Präzedenz und Assoziativität abbildet. Die Grammatik soll die Menge aller ganzzahligen arithmetischen Ausdrücke darstellen. Nehmen Sie für $\Sigma := \{++, --, *, /, \%, \hat{}, +, -, 0, ..., 9\}$ an.

`++, --` (postfix)	linksassoziativ
`++, --` (prefix)	rechtsassoziativ
`^`	rechtsassoziativ
`*, /, %`	linksassoziativ
`+, -`	linksassoziativ

Aufgabe 3: Symboltabellen und Definitionen

a) Geben Sie zur der aufgeführten Hierarchie von Symboltabellen einen möglichen Codeblock an, aus dem diese Hierarchie resultiert. Halten Sie sich dafür an die in Vorlesung 22 Folie 12 eingeführte Konvention.



Die oberste Tabelle ist dabei ausserhalb des anzugebenen Blocks. Achten Sie jedoch bei ihrem Codeblock darauf, dass die nicht typisierten Einträge durch den Codeblock resultieren.

b) Geben Sie zu dem aufgeführten Codeblock die resultierende Hierarchie von Symboltabellen an.

```
{int i; int j;
  {char m;
    ...m...; ...o...;
  }
  ...i...; ...j...; ...g...;
}
{short w; double k;
  {char m;
    ...m...; ...g...; ...o...;
  }
  {char n;
    ...n...; ...k...;
  }
  ...w...; ...k...;
}
```

Aufgabe 4: Mehrdeutigkeit

Gegeben sei folgende Grammatik:

$\Sigma := \{if, expr, then, else, begin, end, a := 1\}$

Startsymbol: STMT

$$
\begin{aligned}
STMT &\rightarrow ASSIGN \mid IFTHEN \mid IFTHENELSE \mid BEGINEND \\
IFTHEN &\rightarrow \texttt{if expr then}\ STMT \\
IFTHENELSE &\rightarrow \texttt{if expr then}\ STMT\ \texttt{else}\ STMT \\
BEGINEND &\rightarrow \texttt{begin}\ STMTLIST\ \texttt{end} \\
STMTLIST &\rightarrow STMTLIST\ STMT \mid STMT \\
ASSIGN &\rightarrow \texttt{a:=1}
\end{aligned}
$$

Zeigen Sie, dass diese Grammatik mehrdeutig ist.

Aufgabe 5: Lex Einführung

In der Vorlesung wurde kurz das Thema Lex und Yacc behandelt. An dieser Stelle soll tiefer auf Lex eingegangen werden, damit Sie anschließend in der Lage sind kleine Lexerprogamme zu schreiben.

Als erstes muss dafür die Struktur eines Lexprogramms näher erläutert werden.

```
Deklaration
%%
Uebersetzungsregeln
%%
Hilfsfunktionen
```

Deklaration :

- **C Code** : Jeglicher Code innerhalb von %{ und %} wird direkt in den Lexer übernommen. Meistens werden an dieser Stelle Variablen definiert. Bsp: `int wordcount = 0;`
- **Definitionen** : Hier können einfache reguläre Ausdrücke mit einem Namen versehen werden, die später in den Regeln aufgegriffen werden können. Bsp: `number [0-9]+`
- **Statusdefinitionen** : Wenn eine Regel abhängig von einem Kontext ist kann dies mit vorher definierten Stati gelöst werden. Bsp: `%s OPEN`

Übersetzungsregeln : Dieser Bereich enthält eine Menge von Pattern-Aktion Paaren. Bei den Patterns können Definitionen genutzt werden und die Aktionen sind regulärer C Code, der zwischen geschweiften Klammern gesetzt ist. Bsp: `{letter}+ {wordcount++;}`. Der übereinstimmende Text kann durch die Variable `char *yytext` eingesehen werden, sowie dessen Länge mit `int yyleng`. Weiterhin kann in der Aktion der aktuelle Status gewechselt werden, sowie ein Status gesetzt werden unter dem das Pattern anwendbar ist. Bsp: `<SOMESTATE>pattern {BEGIN OTHERSTATE;}`. Eine andere Methode um einen Kontext vorauszusetzen ist es ein Slash `/` zu nutzen. Der Teil vor dem Slash wird in `yytext` übernommen und der Teil danach nur für den Abgleich genutzt (dieser Teil wird auch wieder in den Stream zurückgeschrieben). Bsp: `number/\.number { /* Vorkommastelle */; }`. Falls mehrere Regeln zutreffen sollten wird diejenige genommen, bei der `yytext` am längsten wäre. Falls auch dies nicht eindeutig ist wird die weiter oben definierte Regel bevorzugt.

Hilfsfunktionen : Alleinstehende Lexprogramme definieren hier ihre `main`-Funktion, ebenso finden hier weitere Hilfsfunktionen ihren Platz.

Nun sollten Sie folgendes einfaches Lexprogramm `count.l` verstehen können.

```
%option noyywrap
%{
  int charcount = 0, linecount = 0;
%}
%%
. charcount++;
\n {linecount++; charcount++;}
%%
int main() {
```

```
  yylex(); // Ruft den eigentlichen Lexer auf
  printf("%d Buchstaben in %d Zeilen gefunden.\n", charcount, linecount);
  return 0;
}
```

In der ersten Zeile steht eine notwendige Einstellung, damit die Programme kompilierbar werden, daher diese für Ihre Programme immer übernehmen (dies gilt auch für die `main(...)`). Anschließend kann das Lexprogramm mit "`make count`" kompiliert werden. Dadurch steht Ihnen der fertige Lexer "`count`" zur Verfügung. Einen Test können Sie mit Kommando "`echo Ein Test fuer den neuen Lexer | ./count`" durchführen.

a) Erweitern Sie nun obiges Lexerprogramm, sodass es möglich wird Wörter zu zählen. Achten Sie dabei darauf, dass die Anzahl der Buchstaben weiterhin korrekt gezählt wird.

b) Schreiben Sie ein Lexerprogramm, dass alle ganzzahligen vorzeichenbehafteten Zahlen in dem gegebenen Text findet und aufaddiert. Als Beispiel sei folgende Eingabe gegeben, bei dem Ihr Lexerprogramm 5 zurückgeben sollte.

```
Wir haben 2 Aepfel, 4 Birnen und -1 Banane gekauft.
Wieviele Fruechte sind es insgesamt?
```

Hausaufgabe 1: Lex für Fortgeschrittene (6 Punkte)

Aufbauend auf der Präsenzübung sollen nun kompliziertere Lexerprogamme geschrieben werden, daher sollten Sie für diese Aufgabe die Präsenzübung und speziell „Lex Einführung" erfolgreich absolviert haben.

a) Fügen Sie am Anfang jeder Zeile in einer Datei die entsprechende Zeilennummer davor. Starten Sie dabei von Zeile 1 und sorgen Sie dafür das bei einer Menge von bis zu 99,999 Zeilen die Zeilen trotzdem korrekt eingerückt bleiben.

```
Zeile eins           1: Zeile eins
Zeile zwei           2: Zeile zwei
...               ...
Zeile tausend     1000: Zeile tausend
```

b) Extrahieren Sie aus einem gegebenen C Quelltext alle Vorkommnisse von `TODO:`, `FIXME:` oder `CHANGED:` inklusive dem Text ab dieser Folge. Achten Sie dabei jedoch darauf nur Vorkommnisse in Kommentaren auszugeben und nicht etwa aus Strings.

```
// TODO: Funktion verbessern
void main() {
  int i = 0;
  // CHANGED: Eingabe anders!      TODO: Funktion verbessern
  i = argc;                        CHANGED: Eingabe anders!
  /* TODO: Weitere Funktionen      TODO: Weitere Funktionen
  FIXME: int main()                FIXME: int main()
  */
  char *str = "TODO: Ausgabe";
}
```

c) Bauen Sie einen Lexer, der wie in Vorlesung 20 Folie 16 vorgestellt einfache Rechenbefehle in eine Folge von Tokens umschreibt. Nutzen Sie dabei die vorgestellte `<...>`-Schreibweise und stellen Sie sicher, dass Variablen vom gleichen Namen die gleiche Id erhalten. Als Operationen reichen `+, -, *, /, =` auf Ganzzahlen aus. Für die Vergabe von Ids eigenet sich eine einfache Liste, die von einem gegebenen Listenelement aus die Liste nach dem Variablennamen sucht und dessen Position zurückgibt. Halten Sie sich hierfür an die gegebene `line.h` und das Beispiel `lineExample.c`.

```
val = i + val * -60;     <id,1> <=> <id,2> <+> <id,1> <*> <-60>
res = val * i;           <id,3> <=> <id,1> <*> <id,2>
```

Hinweis: Eingaben wie `i*-60` müssen Sie nicht erwarten, daher jedes eigentliche Token ist schon getrennt aufgeführt.

Hausaufgabe 2: Codeanalyse optimierender Compiler (4 Punkte)

Gegegeben ist folgendes Code-Fragment:

```
#include <stdio.h>

int main (void)
{
  int n = 20;
  int erg;

  if(n == 20) {
    erg = n * 2;
  }

  printf("Ergebnis␣%d\n",erg);

  return 0;
}
```

Im Folgenden sollen ein paar Analysen vorgenommen werden.

a) Wie kann der Ausdruck `erg = n * 2;` des C-Programms effizient berechnet werden?

b) Analysieren Sie den vom GCC Compiler generierten Code. Das Programm soll mit gcc test.c (also ohne Optionen) übersetzt werden. Disassemblieren Sie den vom GCC erzeugten Code (objdump -S ./a.out). Geben Sie das Programmfragment in Assembler an, welches den Ausdruck `erg = n * 2;` berechnet.

c) Der Ausdruck im obigen C-Programm hat nun die Form `erg = n * 8;`. Disassemblieren Sie den vom GCC erzeugten Code (objdump -S ./a.out). Geben Sie das Programmfragment in Assembler an, welches den Ausdruck `erg = n * 8;` berechnet.

d) Das Programm soll nun mit gcc -O2 test.c übersetzt werden. Disassemblieren Sie den vom GCC erzeugten Code (objdump -S ./a.out). Geben Sie das Programmfragment in Assembler an, welches den Ausdruck `erg = n * 8;` berechnet.

Bei der Einführung der Gleitkommazahlen wurden die Gleitkommarechenwerke x87 und SSE der IA32-Architektur kurz vorgestellt. Gegeben ist nun folgender Ausschnitt aus einem C-Programm.

```
#include <stdio.h>

void add(float *a, float *b, float *c)
{
  int i;
  for (i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
  }
}

int main()
{
  ...
  return 0;
}
```

e) Was berechnet die Funktion add?

f) Übersetzen Sie nun das Programm mit dem GCC Compiler und analysieren Sie die Funktion add. Welches der Gleitkommarechenwerke wird benutzt? Geben Sie das Programmfragment in Assembler an.

g) Nutzen Sie jetzt die Möglichkeiten des GCC Compilers, Code für die SSE-Einheiten zu erzeugen. Geben Sie das Programmfragment in Assembler an.