Grundlagen der praktischen Informatik
Dr. Stefan Siemer
M.Sc. Timo Specht

Sommersemester 2025
Georg-August-Universität Göttingen
Institut für Informatik
Lehrstuhl für Theoretische Informatik

Übungsblatt 6 (100 Punkte)

Nicht rekursive Parser

Abgabe bis Di, 3. Juni, 14 Uhr.

Für das erfolgreiche absolvieren der Übungen, müssen Sie sich einmalig in FlexNow zum Übungsmodul **B.Inf.1102.Ue: Grundlagen der Praktischen Informatik - Übung** anmelden. Melden Sie sich **rechtzeitig** in **FlexNow** an.

Rechnerübung: Hilfe zu den praktischen Übungen können Sie in den Rechnerübungen bekommen. Details dazu werden noch bekannt gegeben.

Für die Prüfungszulassung benötigen Sie auf 8 verschiedenen Übungsblättern im theoretischen und praktischen Teil jeweils 40%, also jeweils 20 Punkte.

Theoretischer Teil

Hinweise zur Abgabe der Lösungen:

Die Lösungen der Aufgaben werden in geeigneter Form in der Stud.IP-Veranstaltung der Vorlesung über das Vips-Modul hochgeladen. Sie können Ihre Bearbeitungen gerne mit LaTeX formatieren, es ist aber auch der Upload von Text und Bilddateien in gängigen Formaten möglich.

Sollten Sie LaTeX nutzen wollen, gibt es hierfür Hinweise in Form eines Videos in Stud.IP und Sie können unser LaTeX-Template für Ihre Abgaben nutzen.

Die Aufgaben können in Gruppen von 2 bis 3 Personen bearbeitet und abgegeben werden. Wenn Sie in einer Gruppe abgeben möchten, achten Sie darauf, dass Sie sich vor der Abgabe im Vips-Modul der Vorlesung in einer Gruppe zusammenschließen.

Aufgabe 1 (24 Punkte)

Grammatik $G = (N, T, S, P\}$.

Nichtterminale $N = \{$*program, stmtList, stmt, expr, termTail, term, factorTail, factor, addOp, multOp*$\}$

Terminale $T = \{$`id, (, ), :=, +, -, *, /`$\}$

Startsymbol $S =$ *program*

Produktionen

P = {
program → *stmtList*
stmtList → *stmt stmtList* | ε
stmt → `id :=` *expr*
expr → *term termTail*
termTail → *addOp term termTail* | ε
term → *factor factorTail*
factorTail → *multOp factor factorTail* | ε
factor → `(` *expr* `)` | `id`
addOp → `+` | `-`
multOp → `*` | `/`
}

1. Folgende FIRST-Mengen sind gegeben.

   FIRST(multOp factor factorTail) = {`*`, `/`}

   FIRST(ϵ) = {ϵ}

   FIRST(`(` expr `)`) = FIRST(`(`) = {`(`}

   FIRST(`id`) = {`id`}

   FIRST(`+`) = {`+`}

   FIRST(`-`) = {`-`}

   FIRST(`*`) = {`*`}

   FIRST(`/`) = {`/`}

   Berechnen Sie für jede noch fehlende rechte Seite α der Produktion aus P jeweils die Menge FIRST(α).
   (6 Punkte)
2. Folgende FOLLOW-Mengen sind gegeben.

   FOLLOW(program) = {$$}

   FOLLOW(stmtList) = {$$}

   FOLLOW(stmt) = {`id`, $$}

FOLLOW(expr) = {`id`, `)`,$$}

FOLLOW(multOp) = {`(`, `id`}

Berechnen Sie für die übrigen Nichtterminale A die Menge FOLLOW(A).
(10 Punkte)

3. Konstruieren Sie eine Parse-Tabelle mit Hilfe von FIRST und FOLLOW.
(8 Punkte)

   Hinweis. Es reicht aus in die Zellen die rechten Seiten der Produktionen einzutragen.

Aufgabe 2 (16 Punkte)

Betrachten Sie folgende Parse-Tabelle, `<1>` ist das Startsymbol der zugehörigen Grammatik.

	0	1	¬	∨	∧	$$
<1>	<2>	<2>	<2>			
<2>	<3><4>	<3><4>	<3><4>			
<3>	0	1	¬<2>			
<4>				∨<3><4>	∧<3><4>	ε

Hinweis

In den Zellen sind nur die rechten Seiten der Produktionen notiert. Z.B. Zelle `[<3>, ¬ ]` enthält ¬<2>, d.h. der Inhalt dieser Zelle repäsentiert die Produktion <3>→ ¬<2>.

1. Geben Sie die zugehörige Grammatik $G = (N, T, S, P\}$ an.
(3 Punkte)
2. Stellen Sie dar wie ein nicht-rekursiver Parser unter Benutzung der Parse-Tabelle folgende Eingabe abarbeitet ($$ markiert das Ende der Eingabe).

   ¬0 ∨ 0 ∧ 1 ∨ ¬1 $$

   Vervollständigen Sie dazu nachfolgende Tabelle.
(13 Punkte)

   Hinweis

   In einer Tabellenzeile ist der Zustand des Stapels und die noch nicht verarbeitete Eingaben, inklusive des aktuellen Eingabenzeichen, dargestellt. Weiterhin die Aktion, die aus Stapel und Eingabe folgt. Mögliche Aktionen sind z.B. Folgende.

   - Anwendung einer Produktion, dann wird diese notiert.
   - Entfernen des obersten Stapelsymbols und des aktuellen Eingabensymbols, notiere *match*.

- Stapel γ, Eingabe \$\$, notiere *stop*.
- Ansonsten *error*.

Aus der Aktion ergeben sich Stapel und Eingabe der nächsten Zeile.

Stapel	Eingabe	Aktion
`<1>`γ	$\neg 0 \vee 0 \wedge 1 \vee \neg 1\$\$$	`<1>` $\rightarrow$ `<2>`
`<2>`γ	$\neg 0 \vee 0 \wedge 1 \vee \neg 1\$\$$	`<2>` $\rightarrow$ `<3><4>`
`<3><4>`γ	$\neg 0 \vee 0 \wedge 1 \vee \neg 1\$\$$	`<3>` $\rightarrow \neg$`<2>`
$\neg$`<2><4>`γ	$\neg 0 \vee 0 \wedge 1 \vee \neg 1\$\$$	match
`<2><4>`γ	$0 \vee 0 \wedge 1 \vee \neg 1\$\$$	

Aufgabe 3 (10 Punkte)

$P' = \{$

product	$\rightarrow$	*factor*
factor	$\rightarrow$	*factor* `*` `id` | `id`
sum	$\rightarrow$	`id` *sumList*
sumList	$\rightarrow$	`+` *sum* | `+` `(` *product* `)`

$\}$

Schreiben Sie die Produktionen in P' so um, dass keine Linkrekursionen und keine gemeinsamen Präfixe mehr vorkommen.
(10 Punkte)

Praktischer Teil

Hinweise zur Abgabe der Lösungen:

Die Prüfsumme wird in geeigneter Form in der Stud.IP-Veranstaltung der Vorlesung über das Vips-Modul hochgeladen. Jedes Aufgabenblatt hat die Aufgabe *Abgabe-Prüfsumme*. Dort hinterlegen Sie Ihre generierte Prüfsumme. Sie können zudem auch Ihren Quellcode dort hinterlegen.

Die Aufgaben können in Gruppen von 2 bis 3 Personen bearbeitet und abgegeben werden. Wenn Sie in einer Gruppe abgeben möchten, achten Sie darauf, dass Sie sich vor der Abgabe im Vips-Modul der Vorlesung in einer Gruppe zusammenschließen. Dabei ist zu beachten, dass die Gruppen für den theoretischen und praktischen Teil identisch sein müssen.

Aufgabe 4 (50 Punkte)

Haskell - First Mengen

In dieser Aufgabe geht es darum, das Berechnen von First Mengen in Haskell nachzubilden. Machen Sie sich dafür zunächst ein bisschen mit dem Datentyp `Map` vertraut. Informationen zu Maps in Haskell finden sie hier: https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html .

Betrachten Sie folgenden vorgebenen Code.

```haskell
import Data.Map qualified as Map

data Symbol = N String | T String | Epsilon deriving (Ord)

type Production = [Symbol]

data Rule = Rule {lhs :: Symbol, rhs :: [Production]}

data Grammar = Grammar [Rule]

instance Eq Symbol where
  (==) (N n1) (N n2) = n1 == n2
  (==) (T t1) (T t2) = t1 == t2
  (==) Epsilon Epsilon = True
  (==) _ _ = False
```

Eine Grammatik wird in diesem Kontext als eine Liste von Ableitungsregeln gesehen. Sie wird durch den Datentyp `Grammar` repräsentiert.

Ableitungsregeln werden durch eine linke und rechte Seite dargestellt. Dabei ist die linke Seite ein einzelnes Nichtterminal und die rechte Seite eine Liste von Produktionen. Ableitungsregeln werden durch den Datentyp `Rule` dargestellt.

Eine Produktion ist eine Liste aus Symbolen. So wird bspw. die Produktion

$$E \to TE',$$

durch die Liste $[T, E']$ repräsentiert. Produkionen werden mit dem Typ `Production` realsiert.

Zuletzt gibt es den Datentyp `Symbol`. Dieser unterscheidet zwischen den Fällen **N**ichterminal, **T**erminal und **Epsilon**. Ebenfalls exsitiert bereits eine `Eq` instanz für diesen Datentyp.

Betrachten Sie für die Aufgabe die folgende Grammatik:

Nichtterminale $N = \{E,\ E',\ F,\ T,\ T'\}$

Terminale $T = \{$`id, +, *, (, )`$\}$

Startsymbol $S = E$

Produktionen

$$
\begin{array}{lll}
\text{P} = \{ & E & \rightarrow TE' \\
 & E' & \rightarrow + \ TE' \mid \varepsilon \\
 & T & \rightarrow FT' \\
 & T' & \rightarrow * \ FT' \mid \varepsilon \\
 & F & \rightarrow (E) \mid id
\end{array}
$$

Um die First Mengen von einer gegebenen Grammatik zu berechnen, sollen Sie zuerst `show` für die oben angeführten Datentypen implementieren. Im Anschluss geht es darum, die Berechnung von First Menge mittels verschiedener Hilfsmethoden zu realsieren.

1. Implementieren Sie `show` für `Symbol`.

```
instance Show Symbol where
    ...
```

   Eine Beispiel für eine übersichtliche Ausgabe könnte wie folgt aussehen.

```
let e1 = N "E1"; pl = T "+"; e = Epsilon
print e1
print pl
print e
```

```
<E1>
+
epsilon
```

   (5 Punkte)

2. Implementieren Sie `show` für `Rule`.

```
instance Show Rule where
    ...
```

   Eine Beispiel für eine übersichtliche Ausgabe könnte wie folgt aussehen.

```
let r = Rule {lhs = e2, rhs = [[pl, t1, e2], [e]]}
print r
```

```
<E2> -> [+,<T1>,<E2>] | [epsilon]
```

Hinweis: Es könnte nützlich sein, eine Methode zu schreiben die eine Liste von Produktionen bekommt und diese in einen String umwandelt. (5 Punkte)

3. Implementieren Sie `show` für `Grammar`.

```
instance Show Grammar where
    ...
```

Eine Beispiel für eine übersichtliche Ausgabe könnte wie folgt aussehen. Dabei wurde die oben definierte Grammatik genutzt.

```
let e1 = N "E1"; e2 = N "E2"; f = N "F"; t1 = N "T1"; t2 = N "T2"
let id = T "id"; pl = T "+"; mul = T "*"; po = T "("; pc = T ")"
let e = Epsilon

let r1 = Rule {lhs = e1, rhs = [[t1, e2]]}
let r2 = Rule {lhs = e2, rhs = [[pl, t1, e2], [e]]}
let r3 = Rule {lhs = t1, rhs = [[f, t2]]}
let r4 = Rule {lhs = t2, rhs = [[mul, f, t2], [e]]}
let r5 = Rule {lhs = f, rhs = [[po, e1, pc], [id]]}
let g = Grammar [r1, r2, r3, r4, r5]
print g
```

```
<E1> -> [<T1>,<E2>]
<E2> -> [+,<T1>,<E2>] | [epsilon]
<T1> -> [<F>,<T2>]
<T2> -> [*,<F>,<T2>] | [epsilon]
<F> -> [(,<E1>,)] | [id]
```

(5 Punkte)

4. Implementieren Sie eine Funktion

```
removeEpsilon :: [Symbol] -> [Symbol]
```

die eine Liste von Symbolen übergeben bekommt und, sofern vorhanden, ε aus dieser Liste entfernt und anschließend das Ergebnis zurückliefert. (5 Punkte)

5. Implementieren Sie eine Funktion

```
containsEpsilon :: [Symbol] -> Bool
```

die eine Liste von Symbolen übergeben bekommt und überprüft, ob ε in dieser Liste enthalten ist. Ist ε in der Liste, wird `True` zurückgegeben, ansonsten `False`. (5 Punkte)

6. Implementieren Sie eine Funktion

```
findRules :: Symbol -> [Rule] -> [Production]
```

die ein Nichtterminal und eine Liste von Regeln übergeben bekommt und nach allen Regeln sucht, sodass das übergebene Nichtterminal auf der linken Seite steht. D.h., die Funktion sucht alle Produkionen heraus, die zu einem gegebenen Nichtterminal gehören. (5 Punkte)

7. Implementieren Sie eine Funktion

```
firstSetProd :: Grammar -> Production -> [Symbol]
```

   die eine Grammatik und eine Production übergeben bekommt und anschließend die First Menge, die zu dieser Produktion gehört berechnet. Nutzen Sie dafür die aus der Vorlesung bekannten Regeln. Dabei wird die First Menge als Liste von Symbolen zurückgeliefert. (10 Punkte)

8. Implementieren Sie eine Funktion

```
firstSetHelper :: Grammar -> [Production] -> [Symbol]
```

   die eine Grammatik und eine Liste von Produktionen übergeben bekommt und für jede Produktion aus dieser Liste die First Menge berechnet. Alle First Mengen werden in einer Liste zusammengefasst und zurückgeliefert. (5 Punkte)

9. Implementieren Sie eine Funktion

```
firstSet :: Grammar -> Symbol -> [Symbol]
```

   die eine Grammatik und ein Symbol übergeben bekommt und die First Menge für das gegebene Symbol berechnet. Dabei ist die First Menge für das Symbol die Liste aller First Mengen der einzelnen Produktionen die zu diesem Symbol gehören. Testen Sie Ihre Funktion mit der oben angeführten Grammatik und einem beliebigen Nichtterminal aus dieser. (5 Punkte)