Prof. Dr. R. Loogen, M. Dieterle
Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße
D-35032 Marburg

1. November 2011

Philipps Universität Marburg

Übungen zu „Grundlagen des Compilerbau“, Winter 2011/12

Nr. 3, Abgabe der Aufgaben: 8. November 2011 vor der Vorlesung

---

Aufgaben

3.1 **Nichtdeterministischer Backtrackautomat** — 6 Punkte

In der Vorlesung haben Sie kennengelernt, wie man ausgehend von mehreren DFAs das LA-Problem lösen kann. Der Produktautomat erkennt die Vereinigung der Sprachen, die durch die zugrundeliegenden DFAs erkannt werden. Erweitert man diesen mit einem Backtrackkopf, kann er eine flm-Analyse durchführen.

Alternativ kann man auch NFAs als Ausgangspunkt benutzen und die Vereinigung der erkannten Sprachen durch einen zusätzlichen Startzustand erreichen, der durch ε-Übergänge mit den ursprünglichen Startzuständen verbunden ist. Die Transformation ist simpel und das Resultat ist wieder ein NFA. Interessanter ist die Implementierung der Backtrackfunktionalität.

Aufgabe: Implementieren Sie die Backtrackfunktionalität für NFAs.

Die Funktion `checkwordBT` soll die flm-Analyse als Liste von akzeptierenden Zuständen liefern. Es ist davon auszugehen, dass jede Symbolklasse von genau einem akzeptierenden Zustand erkannt wird und die Zustandstypen nach der Priorität der Symbolklassen geordnet sind.
Die zwei Modi des Backtrackautomaten können in wechselseitiger Rekursion mit den Funktionen `normalmode` und `lookaheadmode` simuliert werden.

```
data Analysis st = Recognize [st] | LexErr [st] deriving Show

checkWordBT :: Ord st ⇒ (NFA st) → String → Analysis st

normalmode :: Ord st ⇒
              (NFA st)
              → [st]      -- current states
              → String    -- remaining input
              → [st]      -- current analysis given by accepting states
              → Analysis st -- final analysis given by accepting states

lookaheadmode :: Ord st ⇒
              (NFA st)
              → st        -- backtrack state
              → String    -- speculative read string
              → [st]      -- current states
              → String    -- remaining input
              → [st]      -- current analysis given by accepting states
              → Analysis st -- final analysis given by accepting states
```

Hinweis: Eine Lösung zu Aufgabe 2.2 finden Sie (ab Mittwoch) auf der VL-Seite. Sie können diese oder Ihre eigene Lösung für die neue Aufgabe verwenden.

Bitte wenden!

3.2 **Scanner für eine imperative Sprache**

6 Punkte

Für eine kleine WHILE-Programmiersprache seien die Symbolklassen und Präzedenzen wie folgt definiert:

1. Schlüsselworte **`program, var, int, bool, if, then, else, while, do, print`** bilden **jeweils** eine Symbolklasse.
2. Arithmetische Operatoren (ArithOp): + – * /
3. Relationale Operatoren (RelOp): = != < <= > >=
4. Trennzeichen und Klammern `; , : := ( { ) }` bilden **jeweils** eine Symbolklasse.
5. Ein Wahrheitswert (BoolVal) ist entweder **`True`** oder **`False`**.
6. Eine Zahl (NumVal) besteht entweder aus der Ziffer 0, oder aus einer nicht-leeren Ziffernfolge, die nicht mit einer 0 beginnt. Eine Zahl kann ein Vorzeichen + oder - haben.
7. Eine Variable besteht aus einem Buchstaben, gefolgt von beliebig vielen Buchstaben und Ziffern. Schlüsselwörter dürfen nicht als Variablen verwendet werden.

Abbildung 1: Mikro-Syntax einer While-Sprache

(a) Geben Sie einen Haskell-Datentyp `Token` analog zu den Symbolklassen in Abbildung 1 an. / 2

(b) Benutzen Sie den Scanner-Generator ALEX, um einen Scanner zu erzeugen, der ein While-Programm in eine Liste von Symbolen umwandelt. / 4
Ein Testprogramm finden Sie auf der Vorlesungsseite.