PVL-Projekt zur Vorlesung “Compilerbau“

– *PVL-Projekt*

Vorbemerkung

Im Rahmen der Prüfungsvorleistung für Compilerbau sollen Sie dieses Projekt bearbeiten. Das Projekt kann in Gruppen mit bis zu drei Studierenden zusammen gelöst und gemeinsam abgegeben werden. Vergessen Sie dann nicht, alle Gruppenmitglieder (einschließlich Matrikelnummer) zu benennen. Die Bearbeitung soll dann als pdf-Datei von einem Gruppenmitglied in Moodle hochgeladen werden.

Die Bearbeitungszeit ist noch offen.

Bei den Aufgaben erwarte ich eine "sinnvolle Bearbeitung" – das ist nicht dasselbe wie "richtiges Ergebnis". Wichtig ist, dass geeignete Methoden fachgerecht eingesetzt werden.

- Strukturieren Sie Ihre Lösung analog zur Aufgabenstellung
- Erläutern Sie Ihre Vorgehensweise in einer Einleitung jeweils kurz: Was war die Idee, was die Lösungsschritte, usw.
- Code/Codeteile immer *formatiert* in das Dokument kopieren.
- Protokollieren Sie die Ergebnisse Ihres Algorithmus/Programms: formatierte, überschaubare Ausschnitte der Ausgaben, evtl. mit Erläuterungen.
- Reflektieren Sie die Ergebnisse am Ende kurz in Textform: Auffälligkeiten, erwartete Ergebnisse, usw.

Offensichtlich abgeschriebene Lösungen werden nicht gewertet.

1 Einführung

In Moodle finden Sie das Projekt `CB – PVL-Projekt.zip` – das ist ein vollständiges Java-Projekt.

Das Programm erlaubt es, endliche Automaten in Text- oder Tabellenform (auch als csv-Datei) zu definieren, einzulesen und auszuführen. Im Folgenden sehen Sie ein (mehr oder weniger sinnvolles) Beispiel für eine solche Definitionsdatei:

Die Syntax ist Zeilenbasiert, Trennzeichen sind alle Arten von White-Spaces, die einzigen Schlüsselwörter sind **`def`** zur Definition eines konstanten Ausdrucks, der zur Automatendefinition genutzt werden kann und **`fsm`** zur Definition eines endlichen Automaten in Form einer Tabelle. Kommentare werden mit einem **`#`** eingeleitet und gehen bis zum Ende der Zeile. Die Syntax zur Definition von Bereichen ist an regex angelehnt.

```
# Das ist ein Kommentar, der bis zum Zeilenende geht.
def  char [a-zA-Z] # Defintion eines Bereichs names "char"
                   # der Name einer Definition muss mindestens
                   # zwei Zeichen lang sein
def  digit[0-9]
def  ex   ~[abc]   # Definition eines exklusiven Bereichs (druch ~)
def  all  ~[]      # Alle Zeichen
def  plus +        # Definition der Konstanten plus mit dem Wert „+“
def  space\        # Definition von Zeichen mit Excape-Sequenzen (hier das
                   # Leerzeichen – grundsätzlich können alle Zeichen
                   # maskiert werden – Zeichen, die anderweitig genutzt werden
                   # müssen maskiert werden (z.B. \, \n, -, [, ], ~ und insbes.
                   # Whitespaces, da sie als Trennzeichen dienen.
def  tab  \t
def  lf   \n

def empty \e       # \e stellt Epsilon dar

fsm bsp            # eine Automaten-Definition als Tabelle

     a    b    c    # der Kopf, hier können die Definitionen genutzt werden
0s   1    1    2    # Zustandsübergänge: s Startzustand, e Endzustand
1    1    2    3
3e   -    -    3    # - kein Übergang

fsm integer        # definiert ein integer

     digit         # verwemdet die digit-Definition von oben
0s        1
1e        1

fsm nondet         # ein nicht-deterministischer Automat

     a    b    \e   # kann einen Epsilon-Übergang haben
0s   1    1|2  1|2|3# kann mehrere mögliche Folgezustände haben
1    1    1    1
2    1    2    3
3e   -    -    -
```

Die Definition wird von einem Recursive-Decent-Parser (siehe Folgevorlesung) eingelesen und geparst.

Der wesentliche Befehl zum Einlesen und Parsen ist

```
Parser p = Parser.fromFile("samples/sample1.txt");
p.entries();
```

War der Parse-Vorgang erfolgreich, stehen die Definitionen in der Symboltabelle zur Verfügung und können über die Ids aus der Symboltabelle ausgelesen und verwendet werden:

```
Symbols s = p.getSymbols();
System.out.println(s);

StateMachine sm = new StateMachine(s.getTable("integer"));
System.out.println(sm.toDetailedString());

sm.init();
int pos = 0;
String input = "1234";
while(sm.isRunning()) {
     char c = (pos < input.length() ? input.charAt(pos++) : 0);
     sm.consume(c);
}
System.out.println("Consuption of '" + input + "' succeeded" +
                                                  " : m.succeeded());
```

Hinweis: Das Projekt ist noch in einem Experimentierstadium – Fehler sind deshalb durchaus zu erwarten. Sie können alle Klassen ändern oder ergänzen.

2 Aufgaben

Falls Sie alleine sind, genügt es eine der beiden Teilaufgaben zu lösen. Falls Sie in einer Gruppe sind, sollten Sie beide Teilaufgaben lösen.

Sie können meine bestehenden Klassen für die Übung natürlich ergänzen oder abändern.

2.1 „Konstruktion NEA / DEA“

Realisieren Sie die Umwandlung eines NEA nach DEA gem. dem Algorithmus der Vorlesung. D.h. ein eingelesener nicht-deterministischer Automat soll in einen deterministischen umgewandelt werden.

Sie können vom Parser die Symboltabelle mit getSymbols() erfragen. Von der Symboltabelle können Sie einen expliziten Automaten anfordern (`getTable(String id)`) oder alle (`getTables()`).

Eine Tabelle ist im Wesentlichen folgendermaßen aufgebaut:

- Ein Header: eine Liste von Expressions, die jeweils ein Zeichen oder eine Gruppe von Zeichen beschreiben; dabei ist auch Epsilon (\e) möglich;
- Eine Liste von Übergängen („Transition“).
- Jede Transition stellt eine Zeile im Automaten dar. Eine Transition hat einen Ausgangszustand und eine Liste von Folgezuständen in der Reihenfolge des Headers

Die Zustände sind in den Klassen des Pakets `tables.semantics.states` definiert für deterministische Automaten sollten nur die `SingleStates` genutzte werden; für nicht-deterministische Automaten kann ein Folgezustand auch ein `StateSet` sein. Ein `StateSet` enthält alle möglichen Folgezustände.

Der Startpunkt der Aufgabe wäre eine (nichtdeterministische) Tabelle einzulesen, zu parsen und diese Tabelle dann durch Ihr Programm umzuwandeln;

```
Symbols s = p.getSymbols();
System.out.println(s);

Table t = s.getTable("nondet");
System.out.println(t);

Table t2 = Tables.toDea(t); // Das ist z.B. Ihre neue Klasse
System.out.println(t2);
```

2.2 Realisierung eines Maximum-Munch-Scanners

Realisieren Sie einen Scanner, der mit Hilfe von (deterministischen) Automaten eine Eingabe zeichenweise erhält und Tokens (die durch die Table-Id gegeben sind) erkennt.

Hinweis: die Aufgabe kann unabhängig von der anderen Aufgabe gelöst werden – Voraussetzung ist natürlich, dass Sie mit deterministischen Automaten arbeiten.

Eine gute Beschreibung mit Beispielen für den Algorithmus ist hier zu finden: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/01/Slides01.pdf

Startpunkt wäre hier, dass Sie zunächst einige endliche Automaten für einige Ausdrücke (z.B. die aus dem pdf) definieren, einlesen und parsen. Damit können Sie nun auf die Tabellen zugreifen und (wie in dem Beispiel zu Beginn) die zugehörigen endlichen Automaten definieren.

Um eine zu scannende Eingabe zeichenweise zu lesen, können Sie die vorgegebene Klasse `scanner.reading.CharacterReader` nutzen. Diese Klasse erlaubt einen `InputStream` zu lesen; z.B.:

```
public class RunCharacterReaderFromString {

     public static void main(String[] args) {
          String input = """
                    Das ist eine Eingabe,
                    die über mehrere Zeilen geht.
                    Und hier zu Ende ist.""";

          CharacterReader s = new CharacterReader(new StringInputStream(input));
          char c;
          do {
               c = s.nextChar();
               System.out.println((int)c + " " + c);
               // Hier konsumiert Ihr Algorithmus das c
          } while(c != 0); // End of Input
     }

}
```

(`StringInputStream` ist eine Unterklasse des `InputStream`, die Klasse ist im Beispielcode enthalten.)

Auf diese gelesenen Zeichen würden Sie den Maximum-Munch-Algorithmus anwenden. Als Besonderheit kann der CharacterReader mit der Methode `acceptLexem(int length)` ein Lexem der Länge `length` liefern – der Startpunkt des nächsten Lesevorgangs wird auf das Ende des Lexems gesetzt. D.h. wenn ein Automat zu weit gelesen hat (was er in der Regel tut, da er ja bis zum „fail“ liest), wird dies hier wieder rückgängig gemacht.