Praktikum Compiler IB 610 Wintersemester 2025/26

Prof. Dr. Markus U. Mock

Praktikum 1: *Der Scanner*

Ausgabe:
Dienstag, 7. Oktober 2025

Abgabe:
siehe Moodle

Inhaltsübersicht:

Überblick

In dieser Praktikumsaufgabe lernen Sie, die erste Phase eines Compiler-Frontends zu erstellen. Sie verwenden JLex, um einen Scanner für eine kleine Sprache (eine Teilmenge von Java) namens *Simple* zu schreiben . Eigenschaften von Simple, die für dieses Praktikum relevant sind, werden im Folgenden beschrieben. Sie werden auch ein Hauptprogramm und eine Eingabedatei schreiben, um den Scanner zu testen. Sie werden sowohl nach der Korrektheit Ihres Scanners als auch nach Ihrer Test-Eingabedatei (Ihren Testfällen) bewertet.

Mit dieser Beschreibung bekommen Hilfsdateien, die die Aufgabenlösung einfacher für Sie gestalten sollen:[1]

Die Dateien sind:

- `simple.jlex` : Eine JLex-Beispielspezifikation.

[1] Alle Dateien sind nur via Docker Container verfügbar, siehe Abschnitt über Docker

- `sym.java` : Token-Definitionen (diese Datei wird später vom Parser-Generator generiert).
- `Errors.java` : Die Errors-Klasse wird zum Drucken von Fehler- und Warnmeldungen verwendet.
- `P2.java` : Enthält das Hauptprogramm, das den Scanner testet.
- `Makefile` : Ein Makefile, das JLex zum Erstellen eines Scanners verwendet und auch `P2.class` erstellt .
- `test.sim` : Eingabe für die aktuelle Version von P2.
- `test2.sim`: Weitere `Testeingabe`. Die Token in dieser Eingabe werden von der an Sie verteilten Version des Scanners nicht erkannt.
- `eof.sim` : Eine Testeingabe mit einer nicht abgeschlossenen Zeichenfolge.

Da das Scannergerüst, das Sie erhalten unvollständig ist (Sie müssen ja den Scanner für Simple erst implementieren), werden Sie beim Aufruf natürlich zunächst Fehler bekommen.

Sie müssen zusätzlich testX.sim Dateien erstellen um die Sprachspezifikation vollständig zu testen. **Die mitgelieferten Tests sind nicht ausreichend für die erfolgreiche Abgabe der Aufgabe.**

JLex

Als Scanner Generator verwenden wir JLex. Das Online JLex Reference Manual erklärt, wie Sie Ihre Token in JLex spezifizieren können. Auf den meisten Linux Systemen kann JLex ganz einfach mittels

```
sudo apt-get install jlex
```

installiert werden.

Alternativ dürfen Sie auch jflex verwenden. Es ist im Wesentlichen identisch mit Jlex, aber manchmal ist es etwas einfacher, mit Jflex reguläre Ausdrücke zu spezifizieren. Für Ihren eigenen Laptop sind sie leicht unter Linux installierbar und auch auf Windows sind sie online

verfügbar. ***In dem Ubuntu Linux Docker Container, der bereit gestellt wird, ist jflex für Sie bereits installiert.***

Die Quellsprache Simple

Hier definieren wir die lexikalischen Aspekte der Programmiersprache, für die Sie einen Compiler entwickeln werden. Die Sprache hat folgende lexikalischen Elemente:[2]

1. *Reservierte Wörter* (das sind Schlüsselwörter der Sprache die nicht als Bezeichner verwendet werden dürfen):

   ```
   String   System.out.println boolean   class do else  false
   if   int   public return static  true     void     while case
   switch default
   ```

2. Bezeichner (Identifier)

   = eine Abfolge von Buchstaben Ziffern und _ (Underscore), wobei der Identifier mit einem Buchstaben beginnen muss und der Bezeichner nicht einem der reservierten Wörter gleich sein darf.

3. Integerliterale (= Folge von einer oder mehr Ziffern)

4. Stringliterale (Zeichenketten)
   Definiert als eine Folge von Null oder mehr “String” Zeichen, die in doppelte Anführungszeichen gestellt sind. Ein String Zeichen ist ist entweder ein escape Zeichen (ein \n \t \’ \” \\ oder einem anderen Zeichen (außer dem Newline \ oder “).
   Beispiele für gültige Stringliterale:
   ```
   “”
   "&!#"
   "use \n to denote a newline character”
   "include a quote like this \" and a backslash like this \\”
   ```

   Beispiele für ungültige Stringliterale:

   ```
    "unterminated
   "also unterminated \"
   "backslash followed by space: \ is not allowed"
   "bad escaped character: \a AND not terminated
   ```

5. Jedes der folgenden Symbole (die aus einem oder zwei Zeichen bestehen):
   ```
    {      }     (      )      ,      =      ;
   +   -   *   /   !   &&   ||
   ```

---

[2] Später im Praktikum werden wir zusätzliche lexikalische Elemente hinzufügen.

== != < > <= >=

Die Token “Namen” (d.h. die Werte, die der Scanner liefern soll) sind in sym.java definiert. Zum Beispiel muss Ihr Scanner `INTLITERAL` liefern, wenn ein Integerliteral erkannt wurde.

6. Einzeilige Kommentare
   Sind wie Java Kommentare: sie beginnen mit // und enden mit dem Ende der Zeile. Ihr Scanner muss Kommentare erkennen, aber COMMENT ist kein Token.

7. Mehrzeilige Kommentare sind ebenso wie in Java, starten mit /* und enden mit */. Auch dafür wird kein Token erzeugt. Kommentare können nicht geschachtelt werden.[3]

8. Whitespace (Leerzeichen)
   Lehrzeichen, Tabulatoren und Newlines sind Whitespace. Whitespace separiert Tokens, werden aber ansonsten (außer in Stringliteralen) ignoriert.

9. Unerlaubte Zeichen
   Jedes Zeichen, das nicht Teil eines Stringliterals oder Kommentar ist illegal.

10. Längenbeschränkungen
    Sie dürfen keine Längenbegrenzung für Bezeichner, Stringliterale, Integerliteral, Kommentare usw. voraussetzen. Funktioniert Ihr Scanner nicht für beliebig lange Bezeichner usw. ist er unkorrekt!

[3] Das heisst: /* ein Kommentar /* geschachtelt */ ende */ funktioniert nicht, da, der Kommentar mit dem ersten */ endet und das **ende */** kein Kommentar mehr ist.

Praktikum via Docker Container

1. Überblick

Die Praktikumsdateien und die dafür notwendigen Tools werden als **Docker-Container** bereitgestellt. Sie können:

- Direkt im Container arbeiten und Ihre Arbeit dort sichern
- Einzelne Dateien und Werkzeuge aus dem Container auf Ihr eigenes (Ubuntu-) Linux-System kopieren
- Ihr Projekt zusätzlich in einem **Git Repository** verwalten (Git ist im Container vorinstalliert)

2. Container starten

Einfacher Start:

```
docker run -it markusmock/ib610:scanner /bin/bash
```

Mit Verzeichnisfreigabe:

Wenn Sie ein Host-Verzeichnis im Container verfügbar machen möchten, um Ihr Praktikum z.B. mit **Eclipse**, **IntelliJ** oder **VSCode** zu bearbeiten:

```
docker run -it -v /pfad/zu/hostverzeichnis:/home/zielpfad markusmock/ib610:scanner /bin/bash
```

Beispiel:

```
docker run -it -v /Users/markusmock/Compiler/solutions:/home/solutions markusmock/ib610:scanner bash
```

Diese Version macht auf meinem Mac das Verzeichnis /Users/markusmock/Compiler/solutions unter dem Pfad /home/solutions im Container verfügbar.

3. Scanner bauen und testen

Nach dem Start erhalten Sie ein Linux-Prompt (`#`). Wechseln Sie ins Scanner-Verzeichnis und führen Sie `make` aus:

```
# cd ib610/scanner
# make
```

Das Makefile enthält eine **Testregel**:

```
root@...:/ib610/scanner# make test
```

Wenn Ihr Scanner noch nicht korrekt funktioniert, sehen Sie eine Fehlermeldung wie:

```
1:1 INTLITERAL (10)
Exception in thread "main" java.lang.Error: Error: could not match input
   at Yylex.zzScanError(simple.jlex.java:463)
   at Yylex.next_token(simple.jlex.java:659)
   at P2.main(P2.java:48)
```

4. Aufgabe

- **Ziel:** Erweitern Sie den Scanner und die Testdateien so, dass er die vollständige Sprachspezifikation erfüllt.
- **Hinweis:** Bearbeiten Sie die Datei `simple.jlex` und ergänzen Sie die fehlenden Regeln zur Spracherkennung.

5. Best Practices

- Speichern Sie regelmäßig Ihre Arbeit im Container.
- Verwenden Sie zusätzlich **Git** zur Versionierung Ihrer Änderungen.
- Arbeiten Sie mit einem gemounteten Verzeichnis, wenn Sie externe IDEs oder Editoren nutzen möchten.