Technische Universität München
Fakultät für Informatik
Prof. Dr. Helmut Seidl
Michael Petter

WS 2003/04

Übersetzung von Programmiersprachen
Merkblatt 4

Verwendung von ANT

1 Vorbereitungen

Das Werkzeug *Ant* entstand durch Anstrengungen des Apache Jakarta - Projekts. *Ant* ist ein Java-basiertes Build-Werkzeug. Es übernimmt die Aufgabe, aus einer Sammmlung von Quellcodes das gewünschte Programm zu erzeugen. Der Benutzer kann das Verhalten von *Ant* dabei über *XML*-Konfigurationsdateien steuern. *Ant* ähnelt daher sehr dem unter C++ Programmmierern verbreiteten `Make`, nur dass es speziell auf die Bedürfnisse von Java angepasst ist, und einige Unsauberkeiten von `Make` vermeidet.

Ant ist wie gesagt ein Java-basiertes Werkzeug, ist also Betriebssystemunabhängig und setzt nur ein installiertes Java 2 voraus. In der Rechnerhalle ist *Ant* bereits vorinstalliert und liegt in `/usr/proj/uebbau/apache-ant-VERSION-bin` vor. *Ant* für den eigenen Rechner bekommt im Internet unter der Adresse `http://ant.apache.org/bindownload.cgi`, und speichert es am Besten in einem temporären Verzeichnis auf seiner Festplatte ab. Die so erhaltene Datei `apache-ant-{VERSION}-bin.zip` entpackt man nun in ein Verzeichnis freier Wahl, zum Beispiel unter `C:\Programme\Ant` oder `~/apache-ant-1.5.3`. Damit *Ant* nun verwendet werden kann, müssen nur noch einige Umgebungsvariablen (`ANT_HOME`, `JAVA_HOME` und `PATH`) richtig gesetzt oder modifiziert werden. Hier unterscheidet man nach dem jeweiligen Betriebssystem:

Unix/Linux/Solaris: Um die Umgebungsvariablen permanent zu setzen müssen gewisse Eintragungen in den Startdateien vorgenommen werden. Dies ist zum Beispiel in der Datei `~/.xsession` möglich, falls diese bei Ihnen existiert, ansonsten nehmen Sie zum Beispiel die Datei `~/.bashrc` beziehungsweise die `~/tcshrc` oder eine ähnliche, je nach dem, welchen Kommandointerpreter (Shell) man verwendet. Den Namen der verwendeten Shell findet man durch die Eingabe von `echo $SHELL` heraus. Auch die Art, wie Sie Umgebungsvariablen setzen können, ist von Shell zu Shell verschieden:

bash Umgebungsvariablen können hier mit `export VARIABLENNAME=INHALT` gesetzt werden

csh Umgebungsvariablen werden hier mit `setenv VARIABLENNAME INHALT` gesetzt

andere für die Bedienung Ihrer Shell konsultieren Sie bitte `man SHELLNAME`

Für einen Benutzer der `bash` seien beispielhaft die zu setzenden Einträge in die `~/.bashrc` in Abbildung 1 auf Seite 2 erläutert:

```
...
export JAVA_HOME=/usr/lib/java2
export ANT_HOME=~/apache-ant-1.5.3
export PATH=$PATH:$ANT_HOME/bin
```

Abbildung 1: Einrichtung der .bashrc

Windows XP: Hier kann man Umgebungsvariablen setzen, indem man auf dem Arbeitsplatz das Symbol “Arbeitsplatz” rechts anklickt und im Kontextmenü dann “Eigenschaften” anwählt. Im daraufhin erscheinenden Dialog “Systemeigenschaften” wählt man nun den Reiter “Erweitert” und dort den Schalter “Umgebungsvariablen”.

In dem nun erschienen Dialog kann man durch klicken auf “Neu” die entsprechenden Variablen, wie in Abbildung 2 auf Seite 3 gezeigt, anlegen.

Nun können Sie *Ant* über die Eingabe von `ant` in die Kommandozeile starten. Falls Sie bei der Installation von *Ant* auf Probleme gestoßen sind, finden Sie unter `http://ant.apache.org/manual/install.html` genauere Installationsinstruktionen.

2 Verwendung

In diesem Teil des Merkblatts wird geschildert, wie man mit *Ant* zu einem Ergebnis kommt. Wir setzen für das einführende Beispiel voraus, dass Sie in der Lage sind, eine einfache HelloWorld.java-Klasse zu erstellen :-)

2.1 Anlegen der Projektumgebung

Für die Arbeit mit *Ant* empfiehlt sich ein projektorientiertes Denken. Am besten erstellen Sie daher für jedes neue Projekt, oder hier jede neue Aufgabe, eine eigene Verzeichnisstruktur. Bewährt haben sich Verzeichnisstrukturen wie folgt:

bin enthält alle Werzeuge, die zur Codegenerierung notwendig sind (wie z.B. JFlex, o.ä.)

classes enthält die fertig kompilierten Klassen

dist enthält das fertige java-Archiv (die ausführbare .jar-Datei)

java für die .java-Quelldateien

lib für Bibliotheken, die für das Übersetzen und Ausführen der Software wichtig sind

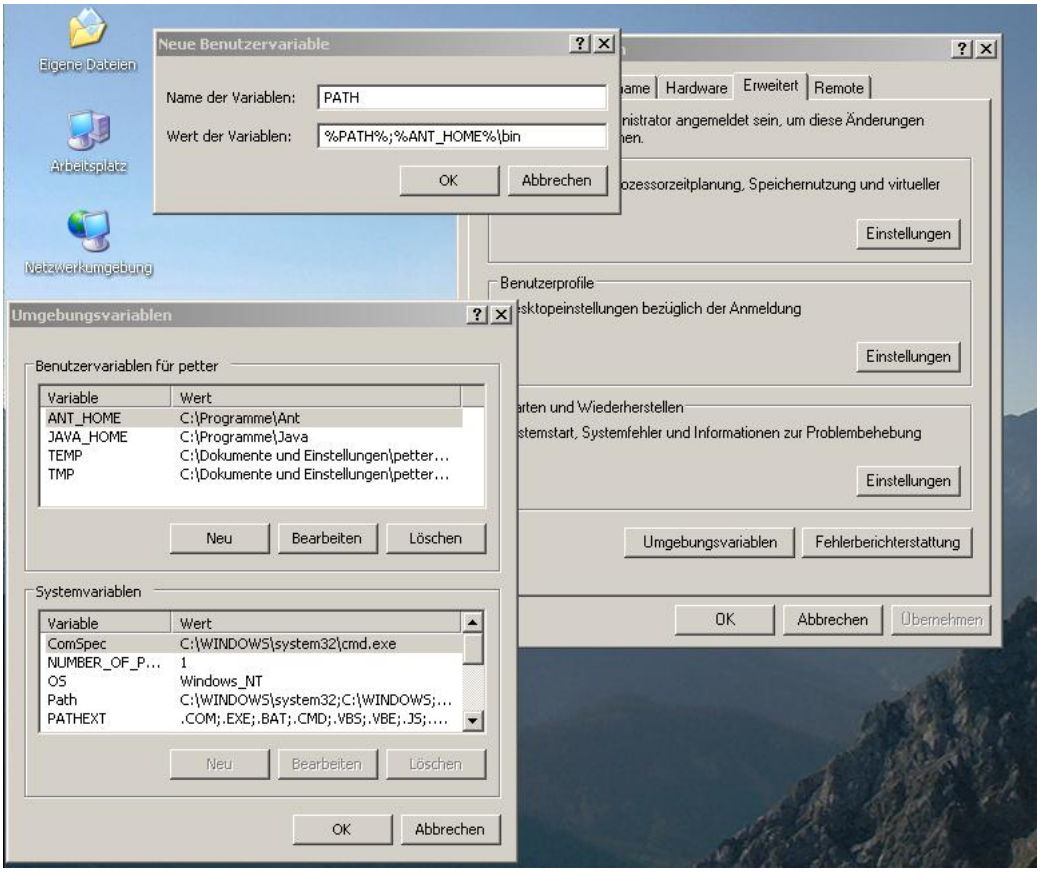

Abbildung 2: Einrichtung unter Windows

Auch wenn es am Anfang schwer fällt, sich konsequent an diese Struktur zu halten, so werden Sie schnell sehen, wie gut Sie Ihnen dabei hilft, bei wachsenden Projekten den Überblick zu behalten.

2.2 Hello World

Speichern Sie Ihr fertiges HelloWorld.java in das Verzeichnis für die Quelldateien (also `java`) ab.

2.3 build.xml

Bevor wir nun anfangen können, den Quellcode zu übersetzen, müssen wir eine Datei für *Ant* erstellen, die beschreibt, was *Ant* tun soll! Diese Datei heißt `build.xml` und wird in das Projektverzeichnis auf unterster Ebene abgelegt. Für unser HelloWorld-Beispiel würde sie aussehen, wie in Abbildung 3 auf Seite 4. Sobald Sie nun `ant dist` in die Kommandozeile eingeben, star-

```
<project name="HelloWorld" default="compile" basedir=".">
  <property name="java"    location="java"/>
  <property name="classes" location="classes"/>
  <property name="dist"    location="dist" />
  <property name="lib"     location="lib"/>

  <path id="libraries">
    <pathelement location="${lib}" />
    <fileset dir="${lib}">
        <include name="**/*.jar" />
        <include name="**/*.zip" />
    </fileset>
    <pathelement path="${java.class.path}" />
  </path>

  <target name="init">
    <mkdir dir="${classes}"/>
    <mkdir dir="${dist}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${java}" destdir="${classes}">
      <classpath refid="libraries" />
    </javac>
  </target>

  <target name="dist" depends="compile">
    <jar jarfile="${dist}/HelloWorld.jar" basedir="${classes}">
     <manifest>
       <attribute name="Main-Class" value="HelloWorld" />
     </manifest>
    </jar>
  </target>

  <target name="run" depends="dist">
    <java jar="${dist}/HelloWorld.jar" fork="true" />
  </target>

  <target name="clean">
    <delete dir="${dist}" />
    <delete dir="${classes}" />
  </target>
</project>
```

Abbildung 3: Einfache build.xml

tet *Ant* und sucht nach der Datei `build.xml`, die dann entsprechend ausgewertet wird. Das Resultat des Erzeugungsprozess aus Abbildung 3 finden Sie im nunmehr neuen Verzeichnis `dist`. *Ant* hat auf unsere Anweisungen hin die fertige `HelloWorld.jar` Datei dort abgelegt. Diese ist gebrauchsfertig, Sie können Sie mit dem Aufruf `java -jar dist/HelloWorld.jar` starten. Sie sehen, der Umwandlungsvorgang ist durch die Verwendung von *Ant* sehr unkompliziert

geworden - Sie müssen nur noch einen Aufruf absetzen.

Am Anfang erscheint der Aufbau der `build.xml` noch etwas schwerfällig, das relativiert sich jedoch sehr schnell, wenn der Erzeugungsvorgang komplexer wird. Nebenbei: Auch zu diesem frühen Zeitpunkt kann sich der Einsatz von *Ant* rentieren: *Ant* führt nämlich immer nur die Schritte durch, die zur Erstellung des Ziels unbedingt notwendig sind. Ist zum Beispiel Ihr Quellcode `HelloWorld.java` unverändert geblieben, so wird die bereits gültige `.class` Datei nicht erneut erzeugt, sondern dieser Schritt übersprungen. Dies kann sich fortsetzen, so dass im Extremfall (zwei unmittelbar aufeinander folgende Eingaben von `ant dist`) beim zweiten Erzeugungsvorgang gar nichts mehr erzeugt (und damit wertvolle Zeit gespart) wird.

2.4 Schrittweise durch die Abhängigkeiten

Wie Sie vielleicht schon bemerkt haben, ist der Erzeugungsvorgang mit *Ant* noch um einige Spezialitäten reicher als ein "gewöhnliches" Shellskript. So ist der ganze Prozess in kleine Teilschritte, sogenannte "Targets" aufgeteilt. Jedes Target beschreibt einen Übersetzungsbestandteil, der in einer Kette von aufeinanderfolgenden Schritten ausgeführt werden muß. Die Reihenfolge der einzelnen Targets aufeinander wird dabei durch die Angabe der "depends" Attribute festgelegt. So kann man zum Beispiel modellieren, dass das Target "compile" immer vor dem Target "dist" ausgeführt werden muß: Klar, denn erst wenn die `.class`-Dateien erzeugt worden sind, können sie mittels `jar cf` in ein Java-Archiv verpackt werden. Diese Abhängigkeiten lassen sich sehr schön in einem Diagrammm wie in Abbildung 4 auf Seite 5 darstellen.

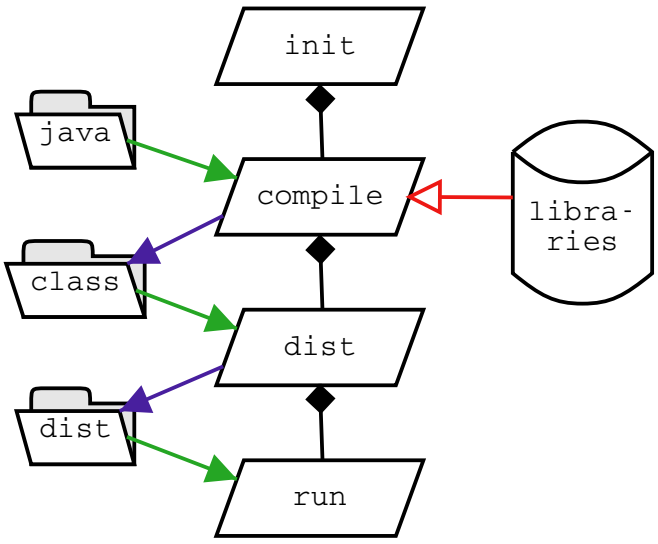

Abbildung 4: HelloWorld Abhängigkeiten

Der Umwandlungsvorgang mit *Ant* lässt sich um beliebige weitere Targets erweitern. Die genaue Herangehensweise beim Erstellen von `build.xml` Dateien lässt sich im Internet unter `http://ant.apache.org/manual/using.html` nachlesen, eine Übersicht über verwendbare

Tasks findet man unter `http://ant.apache.org/manual/anttaskslist.html`.

2.5 Fortgeschrittenes

Wenn Sie bereits mit `Makefiles` gearbeitet haben, werden Sie nun sagen: OK, das kenn ich ja alles schon. Das kann `Make` auch, warum soll ich jetzt *Ant* hernehmen?

Die Gründe für den Umstieg auf *Ant* sind schnell genannt:

1. die Entwicklung mit *Ant* ist plattformunabhängig; Projekte, die auf einem Windows-Rechner kompiliert werden, können ohne Umstellungsaufwand auf eine Sun-Maschine kopiert werden, und kompilieren dort genauso.

2. alle neueren Entwicklungsumgebungen unterstützen *Ant* auf die eine oder andere Weise. Projekte können dadurch mit beliebigen Oberflächen bearbeitet werden, oder sogar von Hand editiert werden.

3. *Ant* gleicht einige Nachteile von Makefiles aus: Die Neuumwandlung veränderter Klassen wird durch *Ant* geschickter gehandhabt als über die althergebrachte Makefile-Methode! Die Einbindung von Zweit-Bibliotheken wird durch die flexible Handhabung von Klassenpfaden zum Kinderspiel. Die Formulierung von Abhängigkeitsbedingungen ist bei *Ant* ausgereifter als bei Make.

4. *Ant* bringt zu den “Standard”-Tasks eine Menge weiterer praktischer Tasks mit, die die Programmentwicklung stark vereinfachen. CVS-Einbindung, Datenbank-Aufrufe, WEB-APP Deployment und automatisches signieren der `.jar`s sind nur ein kleiner Ausschnitt aus dem reichen Fundus an *Ant* Tasks. Und warum sollte man das Rad nochmal erfinden und diese Dienste von Hand über ein Makefile implementieren, wenn sie schon existieren? Außerdem ist die Bedienung dieser Tasks dank *XML* einheitlicher und einfacher.

3 Erweiterungen

Im Verlauf des Praktikums werden wir weitere Werkzeuge zur Quellcodeerzeugung verwenden, die auch in *Ant* eingebunden werden sollten. Wie das im Detail abläuft, können Sie, sobald Sie an einem Punkt angelangt sind, an dem Sie das jeweilige Werkzeug brauchen, hier nachlesen:

3.1 Allgemein

Bevor Sie eines der folgenden Codegenerierungswerkzeuge verwenden, sollten Sie sich klarmachen, dass eine Vermischung eigenen Quellcodes mit generiertem Quellcode keine gute Idee ist. Codegenerierungswerkzeuge haben die Eigenschaft, schnell eine große Menge an Quellcode zu erzeugen, so dass Sie bald den Überblick verlieren. Außerdem könnten Sie auf die Idee kommen, generierten Code abzuändern, was fatal wäre, da bei der nächsten Re-Generierung Ihre Veränderungen verloren gehen.

Die Lösung zu diesem Problem besteht in der Trennung von “eigenem” und generiertem Quellcode. Dazu führen wir einfach ein weiteres Verzeichnis ein, mit Namen `src`. Sie speichern in Zukunft alle Ihre Quellcodes in dieses Verzeichnis, und lassen *Ant* über den `<copy/>`-Task vor jedem Compiliervorgang Ihre aktuellen Quellen in das Umwandlungsverzeichnis `java` übertragen, wo auch die generierten Quellen hineingeschrieben werden. Auf diese Weise bleibt Ihr Quellcode “sauber”.

Weiterhin sollten Sie ein Verzeichnis zur Aufbewahrung der von Ihnen benutzten Codegenerierungswerkzeuge einführen. Wir nennen dieses Verzeichnis `bin`. Es empfiehlt sich, um diese Werkzeuge später komfortabel nutzen zu können, einen eigenen Klassenpfad darauf zu definieren:

```
<path id="binaries">
  <pathelement location="${bin}" />
  <fileset dir="${bin}">
    <include name="**/*.jar" />
    <include name="**/*.zip" />
  </fileset>
  <pathelement path="${java.class.path}" />
</path>
```

3.2 Emugen

Um Emugen nutzen zu können, müssen Sie zuerst die beiden Dateien `VisualEmugen.jar` und `emu_runtime.jar` besorgen. Ersteres speichern Sie im `bin`-Verzeichnis ab, zweiteres legen Sie ins `lib`-Verzeichnis.

Emugen unterstützt seit einiger Zeit die Einbindung in *Ant* über einen eigenen Task. Sie können ihn benutzen, wenn sie ihn zuerst in ihrer `build.xml` durch folgenden Schritt angemeldet haben:

```
<taskdef name="emugen"
         classname="emugen.anttask.EmugenTask"
         classpathref="binaries"
/>
```

Sie können den neuen Task `<emugen/>` jetzt wie gewohnt in Ihrer `build.xml` verwenden:

```
<target name="emugen" depends="init">
 <emugen file="${emu}/InputFile.emu"
         destdir="${java}"
         overwrite="true"
         formsonly="true"
  />
</target>
```

3.3 JFlex

Um JFlex nutzen zu können, müssen Sie die Datei `JFlex.jar` von `http://jflex.sf.net/download.html` herunterladen und in das `bin` Verzeichnis ablegen.

JFlex unterstützt aktiv die Einbindung in *Ant* durch einen eigenen Anttask. Bevor den JFlex-eigenen Task benutzen können, müssen Sie Ihn zuerst noch durch die folgende Zeile in der `build.xml` bekannt machen:

```
<taskdef name="jflex"
         classname="JFlex.anttask.JFlexTask"
         classpathref="binaries"
/>
```

Dadurch wird der neue Task `<jflex/>` verfügbar. Sie können die Generierung des Scanners nun über ein eigenes Ant-Target realisieren:

```
<target name="jflex" depends="init">
 <jflex file="${jflex}/Scanner.jflex" destdir="${java}" />
</target>
```

3.4 CUP

Die Benutzung von CUP mit *Ant* gestaltet sich etwas schwieriger. Zuerst müssen Sie CUP von `http://www.cs.princeton.edu/~appel/modern/java/CUP/` herunterladen, und die Programmklassen in das `bin`-Verzeichnis speichern. Die zur Ausführung eines CUP-Parsers nötigen Laufzeitbibliotheken müssen Sie in das `lib`-Verzeichnis kopieren. Für CUP existiert leider kein eigener Anttask, so dass man sich manuell mit dem bestehenden "Allzwecktask" java behelfen muss:

```
<target name="cup" depends="init">
  <java classname="java_cup.Main"
        dir="${java}"
        failonerror="true"
        fork="true">
    <arg value="-interface" />
    <arg value="-parser" />
    <arg value="Parser" />
    <arg value="${cup}/Parser.cup" />
    <classpath refid="binaries" />
  </java>
</target>
```

Ab Version 0.10l wird CUP vermutlich auch einen ANT-Task bekommen. In der Zwischenzeit kann die an der TU gepatchte Variante java-cup-10k-b2-TUM verwendet werden. Die Benutzung erfolgt dann folgendermaßen:

```
<taskdef name="cup"
         classname="java_cup.anttask.CUPTask"
         classpathref="binaries"
/>
```

Die Benutzung des neuen Task `<cup/>` wird analog zum Kommandozeileninterface, nur mit einigen Verbesserungen, ablaufen:

```
<target name="cup" depends="init">
 <cup srcfile="${cup}/Parser.cup"
      destdir="${java}"
      interface="true"
  />
</target>
```

3.5 classgen

Classgen bekommen Sie unter `http://sf.net/projects/classgen`, Sie müssen die entsprechende `.jar`-Datei wieder in das `bin`-Verzeichnis ablegen.

Auch Classgen besitzt leider keinen eigenen Anttask. Analog zu CUP muss also auch hier ein entsprechender java-Task verwendet werden, damit Classgen von *Ant* benutzt werden kann.

```
<target name="classgen" depends="init">
  <java classname="classgen.Main"
        dir="${java}"
        failonerror="true"
        fork="true">
    <arg value="-visitor" />
    <arg value="-overwrite" />
    <arg value="${cl}/ast.cl" />
    <classpath refid="binaries" />
  </java>
</target>
```

Ab Classgen 1.5pre ist auch ein neuer ANT-Task eingeführt worden. Diese Variante von Classgen kann allerdings momentan nur über das Sourceforge CVS bezogen werden. Die Benutzung erfolgt dann folgendermaßen:

```
<taskdef name="classgen"
    classname="classgen.AntTask"
    classpathref="binaries"
/>
```

Die Benutzung des neuen Task `<classgen />` wird analog zum Kommandozeileninterface, nur mit einigen Verbesserungen, ablaufen:

```
<target name="classgen" depends="init">
 <classgen file="${cl}/ast.cl"
           destdir="${java}"
           visitor="true"
  />
</target>
```

3.6 Alles zusammen

Bei der Verwendung aller Werkzeuge gemeinsam empfiehlt es sich, sehr genau über die Abhängigkeiten der Komponenten untereinander nachzudenken. Im folgenden Listing (Internet)wird eine Mögliche Realisierung gezeigt, in Abbildung 5 auf Seite 12 finden Sie das zugehörige Diagramm.

```
<project name="Compiler" default="compile" basedir=".">
  <property name="cl"      location="cl"      />
  <property name="cup"     location="cup"     />
  <property name="flex"    location="flex"    />
  <property name="src"     location="src"     />
  <property name="java"    location="java"    />
  <property name="classes" location="classes" />
  <property name="dist"    location="dist"    />
  <property name="lib"     location="lib"     />
  <property name="bin"     location="bin"     />
  <property environment="env" />

  <!-- We rely on CUP-10k-TUM, JFlex 1.3.5 and Classgen 1.5pre -->
  <!-- residing in our directory ``bin'' -->
  <path id="binaries">
    <pathelement location="${bin}" />
    <fileset dir="${bin}">
      <include name="**/*.jar" />
      <include name="**/*.zip" />
    </fileset>
    <!-- When the user installed CCK, we use these tool directories -->
    <fileset dir="${env.CCK_HOME}/lib" includes="**/*.jar" />
    <pathelement path="${java.class.path}" />
  </path>

  <path id="libraries">
    <pathelement location="${lib}" />
    <fileset dir="${lib}">
      <include name="**/*.jar" />
      <include name="**/*.zip" />
    </fileset>
    <!-- When the user installed CCK, we use these lib directories -->
    <fileset dir="${env.CCK_HOME}/lib" includes="**/*.jar" />
    <pathelement path="${java.class.path}" />
  </path>

  <taskdef name="jflex"
           classname="JFlex.anttask.JFlexTask"
           classpathref="binaries"
  />
  <!-- We have Classgen 1.5pre -->
  <taskdef name="classgen"
           classname="classgen.AntTask"
           classpathref="binaries"
  />
  <!-- We also use CUP-TUM -->
  <taskdef name="cup"
           classname="java_cup.anttask.CUPTask"
           classpathref="binaries"
  />

  <target name="init">
    <mkdir dir="${classes}"/>
```

```
    <mkdir dir="${java}"/>
    <mkdir dir="${dist}"/>
  </target>

  <target name="classgen" depends="init">
    <classgen file="${cl}/ast.cl"
              destdir="${java}"
              visitor="true"
     />
  </target>

  <target name="cup" depends="classgen">
    <cup srcfile="${cup}/Parser.cup"
         destdir="${java}"
         interface="true"
    />
  </target>

  <target name="jflex" depends="cup">
    <jflex file="${flex}/Scanner.jflex" destdir="${java}" />
  </target>

  <target name="copy_src" depends="jflex">
   <copy todir="${java}">
     <fileset dir="${src}" includes="**/**.java" />
   </copy>
  </target>

  <target name="compile" depends="copy_src">
    <javac srcdir="${java}" destdir="${classes}">
      <classpath refid="libraries" />
    </javac>
  </target>

  <target name="dist" depends="compile">
    <jar jarfile="${dist}/Compiler.jar" basedir="${classes}">
     <manifest>
       <attribute name="Main-Class" value="Compiler" />
     </manifest>
    </jar>
  </target>

  <target name="run" depends="dist">
    <java jar="${dist}/Compiler.jar" fork="true" />
  </target>

  <target name="clean">
    <delete dir="${java}" />
    <delete dir="${classes}" />
    <delete dir="${dist}" />
  </target>
</project>
```

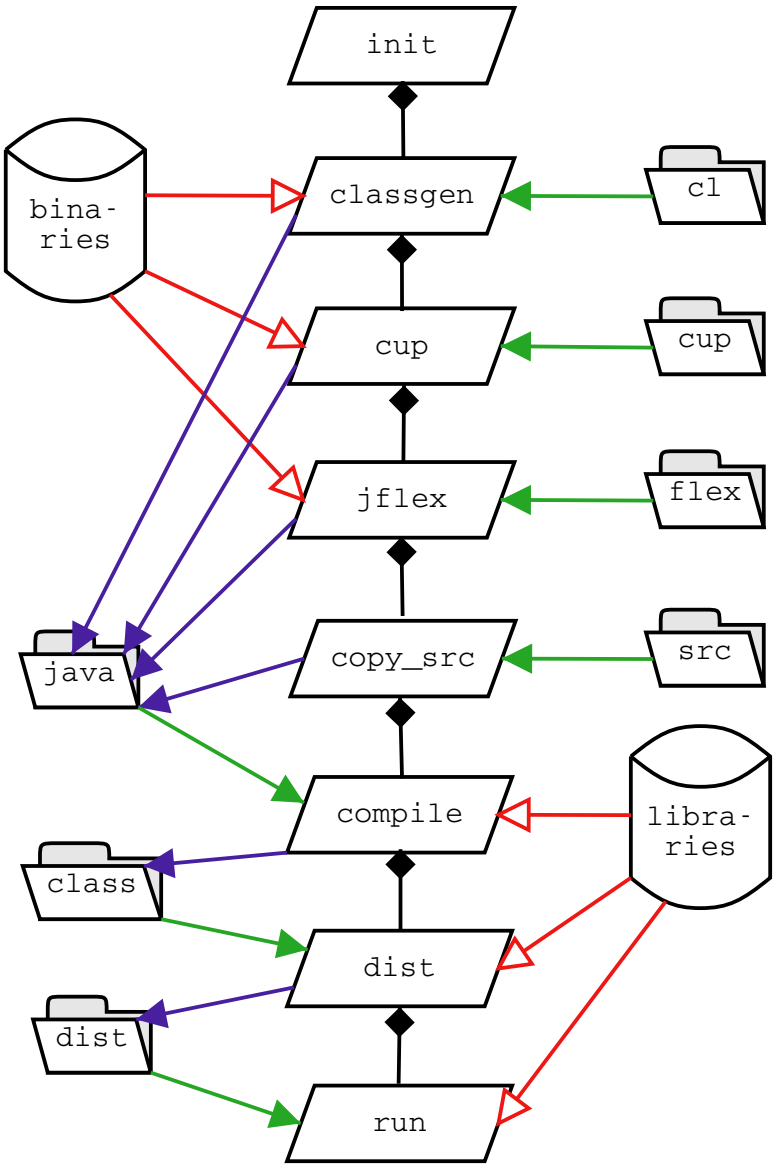

Abbildung 5: Abhängigkeiten in einem kompletten Übersetzer