Programmierpraktikum

Einfacher Crosscompiler von Python nach C++

Felix Helsch
Julian Buchhorn

19. Februar 2015

Betreuer
Dr. Holger Arndt

1 Einleitung

Im Rahmen eines Programmierpraktikums an der Bergischen Universität Wuppertal haben wir einen Compiler von Python zu C++ geschrieben. Dabei ging es in erster Linie darum, zu verstehen wie ein Compiler funktioniert und nicht darum einen vollständigen Compiler zu schreiben.

Als vorläufiges Ziel hatten wir uns gesetzt ein einfaches Testprogramm mit grundlegenden Syntaxelementen zu übersetzen (vgl. Lst. **??**).

Listing 1: *Python Testprogramm für den Compiler*

```
1  from __future__ import division
2
3  """
4  Test program  for codegeneration
5  """
6
7  def f(x):
8      """calculate"""
9      return x**2 + x*4 - (x-3) /x -1.3e2+2\
10             +5 #comment
11
12
13
14 if __name__=='__main__':
15
16     L= [0]*10
17
18     for i in range(len(L)):
19         if i>3:
20             L[i]= f(i+1)
21         else:
22             L[i]+=1
23
24     for x in L:
25         print x
26
27     print 'program', 'end'
```

Das Ziel haben wir erreicht. Unser Compiler kann in seinem momentanen Zustand das Python Programm übersetzen und liefert das C++ Programm in Listing **??**. Dabei mussten wir, wie erwartet, einige Annahmen über das Programm machen, um die Übersetzung zu vereinfachen.

Listing 2: *Vom Compiler erzeugtes C++ Programm*

```
1  #include <array>
2  #include <cmath>
3  #include <iostream>
4
5  using namespace std;
6
```

```
7  /*
8  Test program  for codegeneration
9  */
10
11 double f(double x) {
12     /*calculate*/
13     return pow(x, 2)+ x*4 - (x-3) /(double) x-1.3e2+2
14            +5; //comment
15 }
16
17
18
19 int main() {
20
21     array<double,10> L={};
22
23     for (int i= 0; i<L.size(); i++) {
24         if (i>3) {
25             L[i]= f(i+1);
26         }
27         else {
28             L[i]++;
29         }
30     }
31
32     for (auto x : L) {
33         cout<< x << endl;
34     }
35
36     cout<< "program" <<' '<< "end" << endl;
37
38     return 0;
39
40 }
```

In den folgenden Kapiteln werden wir erklären wie man den Compiler verwenden kann, welche Programme benötigt werden und welche Features der Compiler im Moment unterstützt. Zum Schluss werden wir noch kurz auf die Struktur des Codes eingehen, den wir geschrieben haben.

Für eine detailliertere Erklärung verweisen wir auf die Bachelorarbeit von Felix Helsch, in welcher die Erstellung des Compilers fortgeführt wird.

2 Verwendung

Um ein Python Programm nach C++ zu compilieren, muss man nur das Main-Programm `py2cpp.py` aufrufen und dabei den Namen des zu übersetzenden Programms als Kommandozeilenparameter angeben. Das heißt, dass zum Beispiel unser Testprogramm `program.py` mit dem folgenden Konsolenbefehl übersetzt werden kann:

```
python py2cpp.py program.py
```

Dafür muss man allerdings die benötigten Programme installiert haben (s. Kapitel **??**) und beachten, dass das zu übersetzende Programm nur die momentan unterstützten Features enthält (s. Kapitel **??**).

3 Installation

Zuerst muss man das Archiv mit den Compilerdateien herunterladen [1] und an einer beliebigen Stelle auspacken. Außerdem müssen Python und C++ installiert sein (siehe nächster Abschnitt). Innerhalb dieses Verzeichnisses kann der Compiler dann wie beschrieben aufgerufen werden, wenn sich die zu übersetzende Datei ebenfalls im selben Verzeichnis befindet. Sonst sollte man entsprechend den absoluten Pfad angeben.

3.1 Benötigte Programme

Folgende Programme werden zur Verwendung des Compilers in jedem Fall benötigt, jeweils in der angegebenen Version oder einer neueren:

- `Python` 2.7
  - `pandas` 0.15.2
- `C++` 11
- `Python-Runtime für Antlr` 4.4

Die `Python-Runtime` ist in dem Compiler Archiv schon enthalten, braucht also nicht extra installiert werden. Falls man sie doch runterladen möchte findet man sie in der entsprechenden Sektion der `Antlr` Website [2].

`pandas` ist ein Package für Python welches der Compiler ebenfalls benötigt. Da `pandas` wiederum von den Packages `numpy`, `python-dateutil` und `pytz` abhängt, müssen diese ebenfalls installiert werden. Falls man diese Installationen nicht alle manuell durchführen will, bietet es sich hier an einen Package Manager zu benutzten (für Python unter Windows z.B. `Anaconda`). Für eine detaillierte Installationanleitung verweisen wir auf die Website von `pandas` [3].

3.2 Optionale Programme

Die nachfolgenden Programme werden prinzipiell nur benötigt, wenn man die Grammatik `py.g4` neukompillieren möchte:

- `Antlr` 4.4
  - `Java` 1.6

`Antlr` bietet allerdings auch noch zusätzliche Funktionen für die sich eine Installation lohnen kann. Zum Beispiel kann `Antlr` auch grafische Ansichten des Parsetrees erzeugen (s. Abbildung **??**), welche sehr nützlich sind zur Fehlersuche, oder einfach nur um die Struktur des Parsetrees besser zu verstehen. Außerdem kann die Grammatik dann nicht nur für `Python` kompilliert werden, sondern auch für die anderen Sprachen welche `Antlr` unterstützt.

Da `Antlr` in `Java` geschrieben ist, benötigt es auch eine `Java` Installation. Insgesamt ist die Installation von `Antlr` etwas aufwändiger (insbesondere unter Windows), weswegen wir dazu noch eine ausführlichere Anleitung verfasst haben, welche sich in Abschnitt **??** des Anhangs befindet. Insbesondere für die Installation unter Linux wollen wir aber auch auf die Anleitung auf der `Antlr` Website verweisen [4].

4 Features

4.1 Unterstützte Syntaxelemente

Sowohl Lexer als auch Parser unterstützen prinzipiell alle Syntaxelemente von Python und funktionieren – soweit getestet – mit allen Python Programmen.

Bei der Codegenerierung werden bis jetzt nur die grundlegenden Syntaxelemente unterstützt: Schleifen, `if-elif-else` und `print` Statement. Von den Schleifen wird insbesondere auch die Python `for` Schleife je nachdem in eine Zähl- oder eine `foreach` Schleife übersetzt.

Für unser Testprogramm mussten wir, wie bereits erwähnt, einige Annahmen zur Vereinfachung treffen:

Alle Typdeklarationen haben wir als `double` angenommen. Dadurch ist das Übersetzen von einfachen Funktionen, wie in Zeile **??** des Programms, möglich.

Insbesondere werden Variablendeklarationen noch nicht unterstützt. In Zeile **??** mussten wir deshalb einfach annehmen, dass ein `array` deklariert wird.

Außerdem benötigt das Python Programm noch einen `if __name__=="__main__"`-Block um beim Übersetzen zu erkennen, an welcher Stelle die `main` Funktion in C++ platziert werden soll.

4.2 Fehlererkennung

Der Compiler erkennt auch Fehler im zu kompilierenden Programm und gibt entsprechende Fehlermeldungen aus, die möglichst genau anzeigen, an welcher Stelle der Fehler aufgetreten ist und um welche Art von Fehler es sich handelt:

Ändert man z.B. Zeile **??** des Testprogramms in

```
for i in range(len(L):
```

erkennt der Lexer die fehlende Klammer und gibt folgende Meldung aus:

```
CompilerError:
-------------------------------------------------------------
 Lexer, File "program.py", line 18, col 19:
-------------------------------------------------------------
    L= [0]*10

    for i in range(len(L):
                      ^
  ERR: Missing Closer for  (
```

Ebenso werden falsche Operatoren erkannt, z.B. bei Änderung von Zeile **??**:

```
CompilerError:
------------------------------------------------------------------
 Lexer, File "program.py", line 22, col 20:
------------------------------------------------------------------
            L[i]= f(i+1)
        else:
            L[i]+==1
                  ^
  ERR: Unknown operator +==
```

Und auch inkorrekte Einrückungen (Änderung in Zeile **??**):

```
CompilerError:
------------------------------------------------------------------
 Lexer, File "program.py", line 21, col 9:
------------------------------------------------------------------
        if i>3:
            L[i]= f(i+1)
         else:
        ^
  ERR: Dedentation to incorrect level
```

Außerdem werden auch die Fehler automatisch weitergeleitet, welche der mit Antlr erstellte Parser findet. Antlr gibt typischerweise die betroffene Zeilen- und Spaltennummer an, sowie das nicht passende Zeichen und die Alternativen welche an dieser Stelle möglich wären. Zum Beispiel erzeugt eine Änderung von Zeile **??** in

```
if i=7>3:
```

die Fehlermeldung

```
line 20:12 mismatched input '=' expecting {'!=', '**', '>>', '^',
    '(', '>=', '<', '//', '<>', '+', '/', '<<', 'if', '<=', '&',
   'is', '*', '.', ':', '[', '==', '|', '>', 'or', '%', 'in', '
   and', 'not', '-', ':$blockbegin'}
```

da eine Zuweisung in Python immer ein eigenes Statement ist und keine Expression.

Natürlich ist die Fehlererkennung noch nicht vollständig. Das ist für diesen Crosscompiler aber auch nicht unbedingt nötig, da man normalerweise davon ausgehen kann, dass das zu übersetzende Python Programm korrekt ist.

Aber man kann erkennen, dass die Fehlererkennung mit unserem Compiler auch gut möglich ist und bei Bedarf könnte sie auch noch entsprechend erweitert werden.

5 Codestruktur

Schließlich wollen wir in diesem Abschnitt noch kurz die Struktur unseres Codes erklären, damit man die Funktion des Compilers bei Interesse besser nachvollziehen kann.

In der `main` Datei `py2cpp.py` werden die verschiedenen Programmteile des Compilers aufgerufen:

`LexerLoop.lex(prog)` Ruft den schleifenbasierten Lexer auf. Dieser Lexer hat den Vorteil, dass es einfacher ist an bestimmten Stellen Fehlermeldungen einzufügen, da das Programm Buchstabe für Buchstabe durchlaufen wird.

Der Lexer welchen wir eigentlich verwendet haben, basiert auf regulären Ausdrücken und wird als Teil von `space2braces(prog)` aufgerufen.

`space2braces(prog)` Lext das Programm mit regulären Audrücken, findet aus der Einrückung Blockanfang und -ende herraus und fügt entsprechende Tokens ein (ebenso für Statement Ende). Gibt den Namen der so generierten Programmdatei zurück.

`compile_grammar()` Damit kann das Neukompilieren der Antlr Grammatik `py.g4` optional auch direkt aus dem Wrapper gestartet werden. Beim kompilieren zu Python erzeugt Antlr die Dateien: `pyLexer.py`, `pyListener.py`, `pyParser.py`, `py.tokens` und `pyLexer.tokens`.

`pytree(prog_)` Parst das umgewandelte Programm `prog_` mit dem aus der Antlr Grammatik generierten Parser. Anschließend wird der Parsetree mit dem Listener von Antlr durchlaufen und in unserem eigenen Parsetree gespeichert, welcher dann zurückgegeben wird.

`AST.tofile(prog.replace('.py', '.cpp'))` Durchläuft den Parsetree, wandelt die Python Syntax in C++ Syntax um und speichert das Ergebnis unter demselben Namen wie das Ausgangsprogramm, nur mit Endung `cpp` statt `py`.

`space2braces(...)` ist definiert in `space2braces.py` und befindet sich im Ordner `pylex` zusammen mit den restlichen Dateien, welche für den Lexer benötigt werden

`pytree(...)` ist definiert in `antlr2py.py` und ruft `parsetree.py` auf, wo die Datenstruktur für den Parsetree definiert ist, welche die Methode `tofile` besitzt. In `parsetree.py` befinden sich im Moment außerdem auch die Visitor Methoden für den Parsetree, das heißt dort findet die eigentliche Codegenerierung statt.

A Installation und Verwendung von Antlr

`Antlr` ist prinzipiell ein Parser-Generator mit ähnlicher Syntax wie `Yacc` oder `Bison`. Der hauptsächliche Unterschied liegt darin, dass im Gegensatz zu `Yacc` ein recursiv-descent Parser erzeugt wird. Daraus ergibt sich der Vorteil, dass `Antlr` bessere Fehlermeldungen erzeugen kann. Ein Nachteil ist, dass keine indirekte Linksrekursion möglich ist.

Für eine Erklärung der Syntax verweisen wir auf die `Antlr` Homepage [5], in diesem Kapitel werden nur die Installation beschrieben und die benötigten Kommandozeilenaufrufe für `Antlr` bzw. den erzeugten Parser.

A.1 Installation

Im folgenden geben wir die einzelnen Installationsschritte für Antlr an:

1. Installation des Java Development Kit (SDK/JDK) in der Version 1.6 oder höher
2. Download `Complete ANTLR 4.4 Java binaries jar` (oder neuer) [6]
3. Kopieren der heruntergeladenen Java-Bibliothek in den `jre/lib/ext` Ordner des JDK (zum Beispiel: `C:\Program Files (x86)\Java\jdk1.8.0_25\jre\lib\ext` bzw. unter Linux: `/usr/lib64/jdk1.8.0_25/jre/lib/ext/`)
4. Hinzufügen der `.jar` Datei zum Klassenpfad

   Optional: Alias Einträge für Antlr und das Ausführen des Parsers anlegen

jre vs jdk? => geht beides!

Den Path- und die Aliaseinträge kann man unter Windows zum Beispiel vornehmen in dem man die folgenden Zeilen zum Powershell Profil hinzufügt:

execution policy

Listing 3: *Path- und Aliaseinträge für Antlr am Beispiel von Powershell unter Windows*

```
1 $env:Path+= ";C:\Program Files\Java\jre8\lib\ext\antlr-4.5-
      complete.jar"
2
3 function antlr {java org.antlr.v4.Tool $args}
4 function grun {java org.antlr.v4.runtime.misc.TestRig $args}
```

A.2 Rekompilieren und grafische Syntaxtree-Ansicht

Nachdem man Antlr wie im vorherigen Kapitel beschrieben installiert hat, kann man eine Grammatik kompilieren indem man Antlr mit dem jeweiligen Grammatiknamen aufruft, dass heißt für unsere Pythongrammatik `py.g4`:

```
antlr py.g4
```

Standardmäßig wird aus der Grammatik nun ein Parser in Java generiert. Möchte man den Parser in einer anderen Sprache verwenden, kann man die Grammatik auch in eine andere Zielsprache kompilieren, sofern sie von Antlr unterstützt wird. Für Python 2 zum Beispiel ändert man den Aufruf dafür folgendermaßen ab:

```
antlr py.g4 -Dlanguage=Python2
```

Mit dem Java Parser kann man auch noch weitere Funktionen von Antlr nutzen, so ist es dann z.B. möglich beim Parsen eine grafische Darstellung des Parsetrees anzuzeigen, wie in Abbildung **??**. Der dargestellte Parsetree zeigt eine Funktion aus unserem Beispielprogramm.

Zur Erzeugung des Parsetrees kompiliert man zuerst die Grammatik nach Java, übersetzt dann die Java Dateien in ausführbare Dateien und wendet schließlich den Parser auf das gewünschte Programm an. Das heißt für unsere Grammatik, angewendet auf unser Beispielprogramm, verwendet man die folgenden Aufrufe:

```
antlr py.g4
javac py*.java
grun py prog -gui "program.py"
```

Das zweite Argument beim Ausführen des Parsers ist die Startregel der Grammatik (hier `prog`).

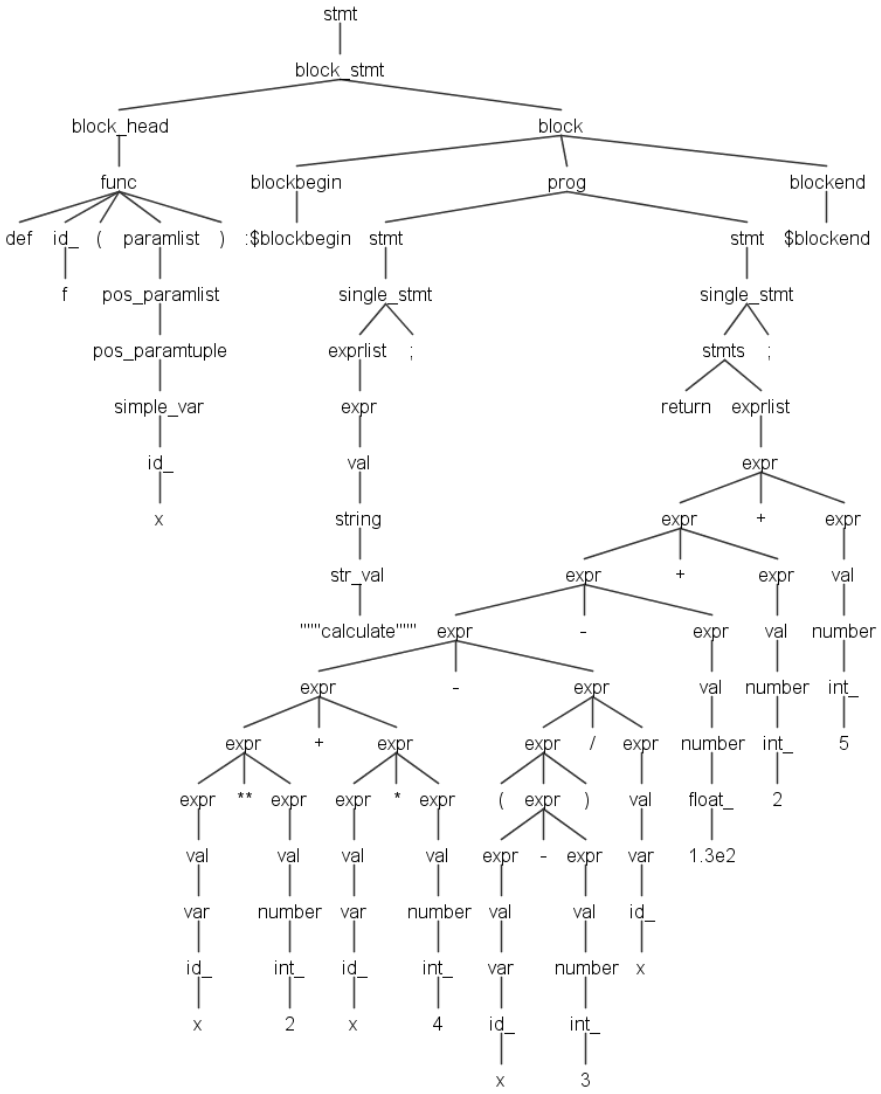

Abbildung 1: *Beispiel Parsetree*

Literatur

[1] *Compiler*. URL: ??.

[2] *antlr, python-runtime*. URL: https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Python+Target (besucht am 08. 02. 2015).

[3] *pandas, Installation*. URL: http://pandas.pydata.org/pandas-docs/stable/install.html (besucht am 08. 02. 2015).

[4] *antlr, Installation*. URL: https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Getting+Started+with+ANTLR+v4 (besucht am 08. 02. 2015).

[5] *antlr, Homepage*. URL: http://www.antlr.org/ (besucht am 08. 02. 2015).

[6] *antlr, Download*. URL: http://www.antlr.org/download.html (besucht am 08. 02. 2015).