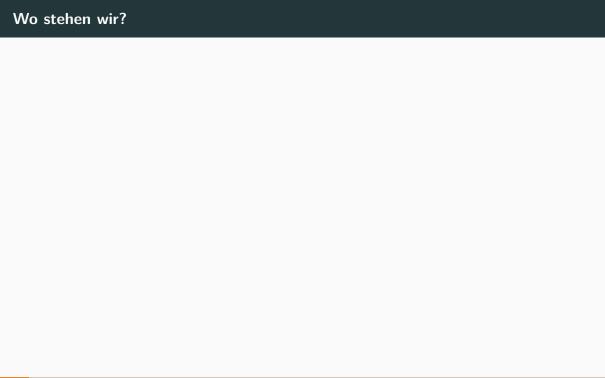
Einstieg Builder für Mini-Python

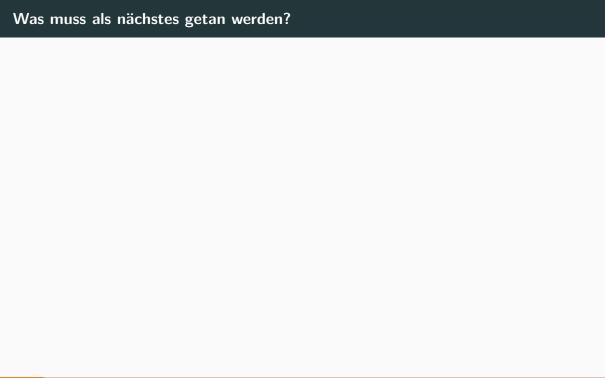
BC George (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

Wiederholung



Motivation



Themen für heute

- Zwischencode
- CBuilder

Zwischencodeerzeugung

Was ist Zwischencode?

- $\bullet \ \ \text{eine quellcode- und maschinencodeunabhängige Darstellung des Programms}$
- Grundlage für einen Interpreter oder Compiler

Warum Zwischencode?

- zentrale Datenstruktur im Übersetzungsprozess
- wird am häufigsten durchlaufen (Anzahl Pässe)
- maschinenunabhängige Analysen und Optimierungen (z. B. dead code elimination)
- bei der Portierung eines Compilers muss nur das Backend neu geschrieben werden
- verschiedene Frontends (Sprachen) können denselben Zwischencode (Backend) benutzen
- wenn der Zielprozessor nicht sehr mächtig ist

Ein Wort zur Optimierung

- Codegeschwindigkeit erhöhen
- möglichst kleine ausführbare Dateien erzeugen
- möglichst energiesparende Programme erzeugen
- Die meisten Optimierungen finden auf dem Zwischencode statt

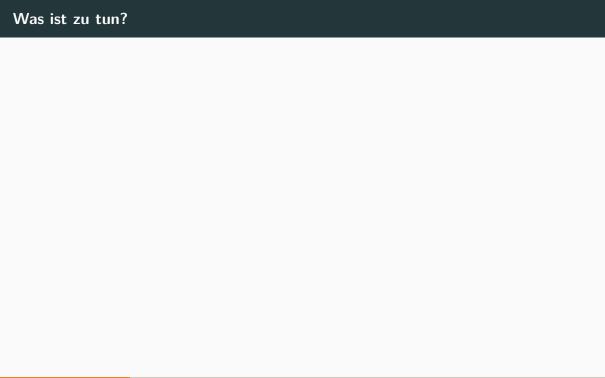
Unser Zwischencode

Und wenn wir C als Zwischencode nehmen?

- wird durchaus so eingesetzt (Haskell, Lush, Eiffel, ...)
- compilierter C-Code ist sehr schnell
- C-Compiler existiert für fast alle Plattformen
- Man bekommt die Optimierung umsonst
- C ist relativ nah an Python

Generierung unseres

Zwischencodes



CBuilder



Den CBuilder gibt es schon!



https://github.com/Compiler-Campus Minden/Mini-Python-Builder

Das Projekt enthält ein Gradle-Buildscript und ein Makefile.

Was wird unter Linux benötigt?

- Make
- gcc oder clang

Und für Windows?

Der generierte Code benötigt Funktionen aus dem POSIX.1-2008-Standard, läuft also nicht direkt unter Windows. Abhilfe:

- mit MSYS2 arbeiten (pacman -S make und pacman -S gcc zur Installation von Make und gcc aufrufen) oder
- 2) WSL (Windows Subsytem für Linux) benutzen oder
- 3) Docker einsetzen oder
- 4) mit einer virtuellen Maschine arbeiten

Und für den Mac

Sie können Make und clang (und einige weitere Kommandozeilen-Tools) mittels xcode-select --install installieren. Alternativen können Sie auch Homebrew oder andere Alternativen nutzen.

Installation des CBuilders (Linux)

- das Repo clonen: ergibt Gradle-Projekt (build.gradle im Hauptverzeichnis vom Git-Repo)
- das ANTLR-Projekt (Ihren bisherigen Quellcode) hineinkopieren
- Main-Klasse festlegen (siehe Doku)

Benutzung des CBuilders

Toolchain

- die Aufrufe des CBuilders in Ihr Projekt integrieren
- den CBuilder ausführen (z. B. über ./gradlew run). Das generierte C-Programm wird z. B. als ./src/program.c geschrieben.
- in diesem Verzeichnis **Make** aufrufen: make all übersetzt das Programm, make run führt es auch aus, make clean entfernt überflüssige erzeugte Dateien.
- Das übersetzte Programm (./bin/program) läuft in einer Konsole (mit input und print-Anweisungen)

Java-Interface des CBuilders

Anlegen des CBuilders und Definieren der Funktionalität

Auszug aus der Doku:

Definition von Literalen

Auszug aus der Doku:

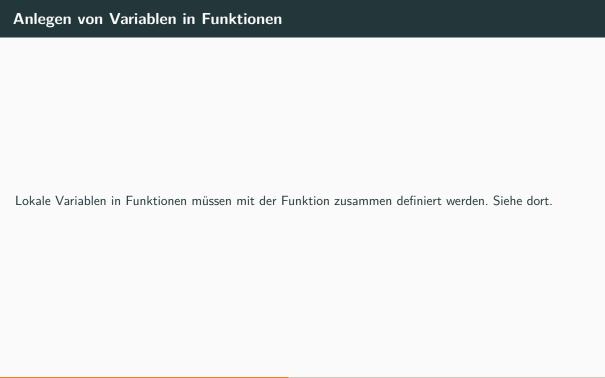
```
new StringLiteral("foo"); // String mit Inhalt "foo" erzeugen
new IntLiteral(5); // Integer mit Wert "5" erzeugen
new BoolLiteral(true); // Boolean mit Wert "True" erzeugen
```

Anlegen von globalen Variablen

 ${\tt builder.addVariable(new\ VariableDeclaration("a"));}$

Danach wird mit Referenzen auf die Variablen gearbeitet:

Reference varA = new Reference("a");



Logische Operatoren

Logische Operatoren können explizit aufgerufen werden:

```
// a = a and b

Expression aAndB = new AndKeyword(varA, varB);
Assignment assignA = new Assignment(varA, aAndB);
builder.addStatement(assignA);
```

Arithmetische Operatoren und Vergleichsoperatoren

Arithmetische Operatoren werden in Form von Methodenaufrufen realisiert. Das geht, weil alle Variablen in Python von object erben, wo die Operatoren als Methoden definiert sind (und man sie so auch überladen kann).

Methodennamen:

```
__add__(), __sub__(), __mul__(), __div__()
__eq__(), __ne__(),__ge__(),__gt__(), __le__(), __lt__()
```

Beispiel mit anschließender Zuweisung:

```
// d = a + b
AttributeReference addA = new AttributeReference("__add__", varA);
Expression add = new Call(addA, List.of(new Expression[] { varB }));
Assignment assignD = new Assignment(varD, add);
```

Kontrollstrukturen

siehe Doku

Built-In-Funktionen

Auszug aus der Doku:

```
Reference printRef = new Reference("print");
List<Expression> parameterRefList = List.of(new Expression[] {varA});
Call printCall = new Call(printRef, parameterRefList);
builder.addStatement(printCall);
```

Eigene Funktionen definieren: Funktionskörper

```
VariableDeclaration localVarYDecl = new VariableDeclaration("y");
Assignment assignYWithX = new Assignment(new Reference("y"), new Reference("x"));
Call printY = new Call(printRef, List.of(new Expression[] {new Reference("y")});
Statement returnY = new ReturnStatement(new Reference("y"));
```

Eigene Funktionen definieren: Argumente für den Konstruktor

Eigene Funktionen definieren: Anlegen der Funktion

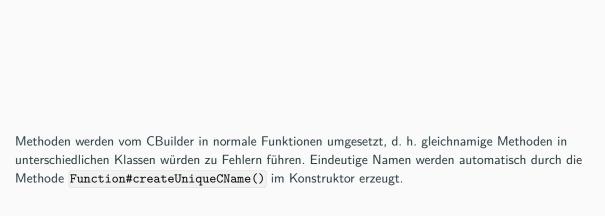
```
Function func1 = new Function("func1", body, parameterArguments, localVariables);
builder.addFunction(func1);
```

eigene Funktionen aufrufen

```
Call callFunc1 = new Call(new Reference("func1"), List.of(new Expression[] {varA}));
Call callPrint = new Call(printRef, List.of(new Expression[] {callFunc1}));
builder.addStatement(callPrint);
```

Klassen definieren

- Typ MPyClass
- Angabe der Referenz auf Elternklasse (__MPytype_Object, wenn keine eigene Oberklasse vorhanden ist)
- __init__() muss immer implementiert werden
- in __init__() muss als erstes super (Klasse SuperCall aufgerufen werden
- Alle Methodendefinitionen müssen als ersten Parameter self enthalten, nicht jedoch die Methodenaufrufe.



Methodennamen

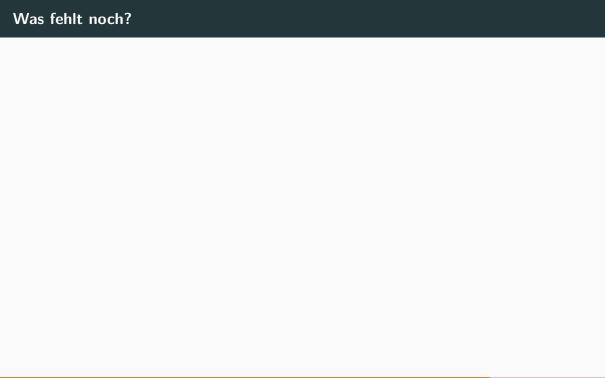
Methoden: ___init___(self) anlegen

```
Statement simpleSuperCall = new SuperCall(List.of());
List<Statement> initBody = List.of(new Statement[] { simpleSuperCall });
List<Argument> initParamList = List.of(new Argument[] { new Argument("self", 0)});
Function methodInit = new Function("__init__", initBody, initParamList, List.of());
```

Methoden: foo(self, x) anlegen

Methoden: Klasse A anlegen

```
List<Function> functionListA = List.of(new Function[] { methodInit, methodFooA });
Reference refToObject = new Reference("__MPyType_Object");
MPyClass classA = new MPyClass("A", refToObject, functionListA, Map.of());
```



Vererbung

eigentlich wie gehabt:

```
// "__init__(self)"
Statement simpleSuperCall = new SuperCall(List.of());
List<Statement> initBody = List.of(new Statement[] { simpleSuperCall });
List<Argument> initParamList = List.of(new Argument[]
    {new Argument("self", 0)});
Function methodInitB = new Function("__init__", initBody, initParamList, List.of());
// "foo(self, x)"
Statement fooPrint = new Call(printRef, List.of(new Expression[] { new Reference("x")}));
List<Argument> fooParamList = List.of(new Argument[] {new Argument("self", 0), new Argument
List<Statement> fooBody = List.of(new Statement[]{ fooPrint });
Function methodFooB = new Function("foo", fooBody, fooParamList, List.of());
// Klasse "B"
List<Function> functionListB = List.of(new Function[]{ methodInitB, methodFooB });
MPyClass classB = new MPyClass("B", new Reference("A"), functionListB, Map.of());
builder.addClass(classB):
```

Die Klassen benutzen

self verwenden: Ausgangscode in MiniPython

```
class C:
    def __init__(self, y):
        self.x = y
    #end

    def getX(self):
        return self.x
    #end
#end
```

```
// Weise "self.x" den Methodenparameter "y" zu
Statement assignSelfX = new AttributeAssignment(new AttributeReference("x", new Reference(
    new Reference("v"));
// Zugriff auf "self.x" in "getX(self)"
Expression getSelfX = new AttributeReference("x", new Reference("self"));
Statement returnX = new ReturnStatement(getSelfX);
// " init (self, y)"
List<Statement> initBodyWithSelfAssign = List.of(new Statement[] { simpleSuperCall,
     assignSelfX });
List<Argument> initParamListWithX = List.of(new Argument[] {new Argument("self", 0),
    new Argument("v", 1)});
Function methodInitWithSelf = new Function("__init__", initBodyWithSelfAssign,
    initParamListWithX, List.of());
```

self verwenden: getX

```
List<Statement> getXBody = List.of(new Statement[] { returnX });
List<Argument> paramListGetX = List.of(new Argument[] {new Argument("self", 0)});
Function getX = new Function("getX", getXBody, paramListGetX, List.of());

// Class "C"
List<Function> functionListC = List.of(new Function[] { methodInitWithSelf, getX });
MPyClass classC = new MPyClass("C", refToObject, functionListC, Map.of());
builder.addClass(classC);
```

Methoden auf Objekten aufrufen

```
// Variable "objectC"
Reference varObjectC = new Reference("objectC");
VariableDeclaration varObjectCDecl = new VariableDeclaration("objectC");
builder.addVariable(varObjectCDecl);
// Erzeugung und Zuweisung eines Objekts der Klasse "C" mit " init (self, 5)"
Call newC = new Call(new Reference("C"), List.of(new IntLiteral(5)));
Assignment assignObjectC = new Assignment(varObjectC, newC);
builder.addStatement(assignObjectC);
// Auf dem Objekt der Klasse "C" die Methode "getX" aufrufen und Rückgabewert ausgeben.
Expression callGetX = new Call(new AttributeReference("getX", varObjectC), List.of());
builder.addStatement(new Call(printRef, List.of(callGetX)));
```

Wrap-Up

Das sollen Sie heute mitnehmen

- Zwischencode ist ein wichtiger Bestandteil eines Compilers, insbesondere für die Optimierung
- Über die Java-API des CBuilders kann aus dem AST heraus Zwischencode in C erzeugt und mit einem vorhandenen C-Compiler in Maschinencode übersetzt werden.

LICENSE



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.