TinyC-Compiler in Java (25 Punkte)

Ihre Aufgabe bei diesem Projekt ist es, einen Compiler zu vervollständigen. Der Compiler übersetzt die Sprache TINYC nach MIPS-Assembler.
Das Projekt ist in drei Teilaufgaben aufgeteilt:

- Konstruktion eines abstrakten Syntaxbaumes (7 Punkte),
- semantische Überprüfung des Programms (Namens- und Typanalyse, 9 Punkte),
- die Erzeugung von Maschinencode für MIPS (9 Punkte).

Beachten Sie, dass alle public Tests der jeweiligen Teilaufgabe bestanden werden müssen, um Punkte für die jeweilige Teilaufgabe bekommen zu können.

1 TinyC

TinyC ist eine eingeschränkte Version von C. Die wichtigsten Einschränkungen bzw. Abweichungen zu C sind:

- Es gibt nur drei Basistypen: `char`, `int` und `void` sowie die Typkonstruktoren für Zeiger (`*`) und Funktionen. Funktionen treten nie als Argumente von Typkonstruktoren auf.
- Es gibt keine Verbunde und Varianten (`struct`, `union`).
- Es sind nicht alle unären/binären Operatoren vorhanden.
- Eine Funktion kann nur maximal vier Parameter haben.
- Es gibt keine globalen Variablen und Funktionszeiger.

1.1 TinyC Beispiel

TinyC-Programme bestehen aus mehreren globalen Deklarationen. Diese umfassen globale Funktionsdeklarationen und Funktionsdefinitionen. Jedes Programm startet mit einer Funktion `main`, die immer einen Wert vom Typ `int` zurückgibt.

```
int foo(int);

int main() {
    return foo(1);
}

int foo(int x) {
    return x + 1;
}
```

Abbildung 1: TinyC Programm

Abbildung 1 zeigt ein gültiges TinyC Programm.

1.2 Grammatik

Folgende Grammatik beschreibt die Syntax von TinyC:

```
TranslationUnit       := ExternalDeclaration*
ExternalDeclaration   := Function | FunctionDeclaration
FunctionDeclaration   := Type Identifier '(' ParameterList? ')' ';'
ParameterList         := Parameter (',' Parameter)*
Parameter             := Type Identifier?
Function              := Type Identifier '(' NamedParameterList? ')' Block
NamedParameterList    := NamedParameter (',' NamedParameter)*
NamedParameter        := Type Identifier
Statement             := Block | EmptyStatement | ExpressionStatement
                       | IfStatement | ReturnStatement | WhileStatement
Block                 := '{' (Declaration | Statement)* '}'
Declaration           := Type Identifier ('=' Expression)? ';'
EmptyStatement        := ';'
ExpressionStatement   := Expression ';'
IfStatement           := 'if' '(' Expression ')' Statement ('else' Statement)?
ReturnStatement       := 'return' Expression? ';'
WhileStatement        := 'while' '(' Expression ')' Statement
Expression            := BinaryExpression | PrimaryExpression | UnaryExpression
                       | FunctionCall
BinaryExpression      := Expression BinaryOperator Expression
BinaryOperator        := '=' | '==' | '!=' | '<' | '+' | '-' | '*' | '/'
FunctionCall          := Expression '(' ExpressionList? ')'
ExpressionList        := Expression (',' Expression)*
PrimaryExpression     := CharacterConstant | Identifier | IntegerConstant
                       | StringLiteral | '(' Expression ')'
UnaryExpression       := UnaryOperator Expression
UnaryOperator         := '*' | '&' | 'sizeof'
Type                  := BaseType '*'*
BaseType              := 'char' | 'int' | 'void'
```

Obige Grammatik verwendet folgende Syntax:

- `'x'`: Das Symbol `x` muss so wörtlich in der Eingabe auftreten. (Terminal)
- `Bla`: `Bla` ist der Name einer anderen Regel. (Nichtterminal)
- `(a b)`: Klammern dienen zum Gruppieren, z.B. für einen nachfolgenden `*` (Gruppierung)
- `x?`: `x` ist optional (0 oder 1 mal). (Option)
- `x*`: `x` darf beliebig oft auftreten (auch 0 mal). (Wiederholung)
- `a b`: `a` gefolgt von `b`. (Sequenz)
- `a | b`: Entweder `a` oder `b`. (Alternative)

Die verschiedenen Operatoren sind in absteigender Bindungsstärke aufgeführt. Ein Beispiel: `a | b* c` bedeutet entweder genau ein `a` (und kein `c`) oder beliebig viele `b` gefolgt von einem `c`.

2 Phasen des Compilers und Implementierung

Die Klasse `Compiler` ist die Hauptklasse für Ihre Implementierung. Diese wird genutzt, um alle Phasen des Übersetzers nacheinander auszuführen:

getAstFactory Gibt eine Instanz der `ASTFactory`-Schnittstelle zurück, die intern von der Instanz Ihrer `Compiler` Klasse genutzt wird. Es gibt also pro Instanz Ihrer Klasse genau eine Instanz Ihrer Implementierung der `ASTFactory`.

parseTranslationUnit Parst die durch den übergebenen Lexer definierte Eingabe und erzeugt einen Baum.

checkSemantics Führt die statische semantische Analyse durch, welches die Namens- und Typanalyse umfasst.

performOptimizations Führt Optimierungen auf dem aktuellen Code durch (siehe Bonusaufgabenblatt).

generateCode Generiert Code für das aktuelle Programm.

Zudem enthält es drei weitere Pakete, die jeweils eine Klasse als Basis für Ihre Klassenhierarchie enthalten:

Type Ihre Typklasse, welche einen Typ repräsentiert.

Expression Ihre Ausdrucksklasse, welche beliebige Ausdrücke in TinyC darstellt.

Statement Ihre Anweisungsklasse, welche beliebige Anweisungen in TinyC darstellt.

Diese Klassen dürfen beliebig erweitert werden. Lediglich der Name der Klassen darf nicht geändert werden.

3 Abstrakter Syntaxbaum und Ausgabe (7 Punkte)

Im ersten Teil des Projektes sollen Sie einen abstrakten Syntaxbaum aufbauen. Damit dieser getestet werden kann, sollen Sie zusätzlich die `toString` Methoden der zum AST gehörigen Klassen implementieren.
Wir haben Ihnen bereits einen Lexer und Parser vorgegeben. Ihre Aufgabe ist es, aus den vom Parser gegebenen Token einen AST aufzubauen. Um dies zu tun, implementieren Sie das Interface `ASTFactory` und `getASTFactory` in `Compiler.java`.

3.1 AST Aufbau

Der Aufzählungstyp `TokenKind` beschreibt die verschiedenen Arten von Tokens. Tokens werden durch die Klasse `Token` dargestellt. Tokens verfügen über Informationen ihrer Position im Programmtext (`Location`), eine Tokenart (`TokenKind` und der Getter `getKind()`) und einen Text (`getText()`), der dem Ursprungstext des Tokens im Programmtext entspricht. Der Parser bekommt, zusätzlich zum Lexer, eine Fabrik-Klasse zur Erzeugung der AST-Knoten bei der Initialisierung übergeben (`ASTFactory`).
Für den jeweiligen AST-Knoten wird dann die entsprechende Methode Ihrer Implementierung aufgerufen und Sie können die entsprechenden Klassen Ihrer Hierarchie erzeugen und zurückgeben. Diese werden vom Parser bei der syntaktischen Analyse des Eingabeprogramms aufgerufen. Betrachten wir die Anweisung `return y + 3;`, die in einer Datei `test.c` in Zeile `13` und beginnend an Spalte `19` steht. Es werden zunächst die folgenden Tokens erzeugt:

```
RETURN     (Location("test.c", 13, 19))
IDENTIFIER (Location("test.c", 13, 26), "y")
PLUS       (Location("test.c", 13, 28))
NUMBER     (Location("test.c", 13, 30), "3")
```

Die von `ASTFactory` erzeugten Knoten müssen entsprechend Instanzen der Klassen `Type`, `Expression` oder `Statement` sein. Für das obige Beispiel werden konzeptionell die folgenden Methoden der `ASTFactory` aufgerufen:

```
Expression y     = factory.createPrimaryExpression(IDENTIFIER(..., "y"));
Expression three = factory.createPrimaryExpression(NUMBER(..., "3"));
Expression plus  = factory.createBinaryExpression(PLUS(...), y, three);
Statement  ret   = factory.createReturnStatement(RETURN(...), plus);
```

Neben den Statements die in der Syntax von TinyC beschrieben sind gibt es in der `ASTFactory` noch ein Error-Statement. Dieses Statement ist kein Element der Sprache, sondern wird erzeugt wenn ein Fehler beim Parsen auftritt.
Tipp: Die Methoden `createExternalDeclaration` und `createFunctionDefinition` geben `void` zurück. Erzeugen Sie in Ihrer `ASTFactory` Implementierung eine Liste von `ExternalDeclarations`. Diese beiden Methoden können erzeugte Deklarationen direkt in diese Liste hängen.

3.2 Ausgabe des Compilers

Jede Ihrer Klassen, die von einer der Klassen `Type`, `Expression` oder `Statement` abgeleitet wurden, müssen die Methode `toString` überschreiben. Diese wird zum Testen Ihres eigenen Rahmenwerkes genutzt.
Die genaue Zeichenkettendarstellung ist im folgenden genau spezifiziert:

- Die Ausdrücke sind maximal geklammert. Alle Teilausdrücke, die aus mehr als einem Token bestehen, werden von Klammern umgeben. Zum Beispiel wird `&x + 23 * z` so ausgegeben:

```
((&x) + (23 * z))
```

- In der Ausgabe von Anweisungen müssen sich alle syntaktischen Elemente, wie Klammern oder Schlüsselwörter wie `while`, befinden:

```
while ((i > 0)) { (i = (i - 1)); }
```

- Basistypen werden entsprechend ihrer Namen ausgegeben:

```
char
int
void
```

- Bei Zeigertypen wird zuerst der Zieltyp gefolgt von einem Stern ausgegeben:

```
void*
```

- Bei Funktionstypen wird zunächst der Rückgabetyp ausgegeben. Nun folgt eine öffnende Klammer und danach die Parametertypen (mit Kommata getrennt). Zum Schluss wird der Typ mit einer schließenden Klammer beendet:

```
void(int, int)
```

- Sämtlicher Leerraum (whitespace) wird beim Überprüfen der textuellen Darstellung verworfen.

4 Semantische Analyse (9 Punkte)

Die folgenden Abschnitte beschreiben die Regeln zur Typisierung und den Gültigkeitsbereichen von Variablen in TinyC, die Sie in Ihrer semantischen Analyse überprüfen sollen. Ihr semantische Analyse soll bei dem Aufruf von `checkSemantics` in `Compiler.java` ausgeführt werden.

4.1 Typisierung

TinyC enthält einen Ausschnitt der Typen von C. Diese sind in einer Typhierarchie angeordnet. Es gibt Objekttypen und Funktionstypen. Die Typen `char` und `int` sind Ganzzahltypen. Alle Ganzzahltypen sind vorzeichenbehaftet. Zusammen mit den Zeigertypen bilden sie die skalaren Typen. Alle skalaren Typen und `void` sind Objekttypen. Es gibt keine Funktionszeiger. Der Typ `void` und Funktionstypen sind unvollständige Typen. Insbesondere ist `void*` kein unvollständiger Typ. Abbildung 2 verdeutlicht die Typhierachie.

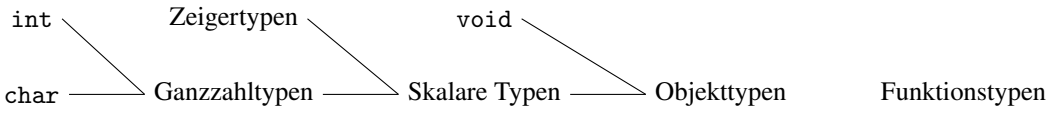

Abbildung 2: Typhierarchie von TinyC

Bei der Typüberprüfung sollen folgende Eigenschaften beachtet werden:

- **Bedingungen:** Der Typ von Bedingungen muss skalar sein.
- **Zuweisungen:** Bei Zuweisungen muss einer der folgenden Punkte gelten:

  - Die Typen beider Operanden sind identisch.
  - Beide Operanden haben Ganzzahltyp.
  - Beide Operanden haben Zeigertyp und mindestens einer der beiden hat Typ `void*`.
  - Der linke Operand hat Zeigertyp und der rechte Operand ist eine Nullzeigerkonstante.

  In jedem Fall müssen beide Operanden vollständige Objekttypen sein.

- **Funktionsaufrufe:** Die Parameterübergabe bei Funktionsaufrufen unterliegt denselben Regeln wie Zuweisungen. Hierbei ist der Parameter der Funktion als "linke Seite" und der übergebene Wert als "rechte Seite" zu interpretieren.
- **Rückgabewerte:** Der Typ von `return` ("rechte Seite") wird mit dem Rückgabetyp der umgebenden Funktion ("linke Seite") wie eine Zuweisung behandelt. Es muss genau dann ein Ausdruck vorhanden sein, wenn der Rückgabetyp nicht `void` ist. Es ist nicht notwendig zu prüfen, ob ein Pfad zum Ende einer Nicht-`void`-Funktion keine `return`-Anweisung enthält. In einem solchen Fall ist der Rückgabewert undefiniert.
- **Nullzeiger:** Eine Nullzeigerkonstante ist eine Zahlkonstante mit dem Wert `0`. In Zeigerkontexten (z.B. Vergleich mit Zeiger, Zuweisung an Zeiger) wird sie als Nullzeiger angesehen.
- **Fehler:** Das `ErrorStatement` hat keinen Typ und unterliegt daher keinen Typbeschränkungen.

Tabelle 1 zeigt alle Operatoren, die Sie implementieren müssen, und beschreibt deren Signaturen.

Operator	Linker Operand	Rechter Operand	Ergebnis	Hinweis
		Binäre Operatoren		
`*`	`Ganzzahl`	`Ganzzahl`	`int`	
`/`	`Ganzzahl`	`Ganzzahl`	`int`	
`+`	`Ganzzahl`	`Ganzzahl`	`int`	
`+`	`Zeiger`	`Ganzzahl`	`Zeiger`	`Zeiger auf vollständigen Typen`
`-`	`Ganzzahl`	`Ganzzahl`	`int`	
`-`	`Zeiger`	`Ganzzahl`	`Zeiger`	`Zeiger auf vollständigen Typen`
`-`	`Zeiger`	`Zeiger`	`int`	`Identischer Zeigertyp auf vollständigen Typen`
`==`	`Ganzzahl`	`Ganzzahl`	`int`	
`==`	`Zeiger`	`Zeiger`	`int`	`Identischer Zeigertyp/void*/Nullzeigerkonstante`
`!=`	`Ganzzahl`	`Ganzzahl`	`int`	
`!=`	`Zeiger`	`Zeiger`	`int`	`Identischer Zeigertyp/void*/Nullzeigerkonstante`
`<`	`Ganzzahl`	`Ganzzahl`	`int`	
`<`	`Zeiger`	`Zeiger`	`int`	`Identischer Zeigertyp`
`=`	`Objekt`	`Objekt`	`Objekt`	`Linke Seite muss zuweisbar sein (LValue)`
		Unäre Operatoren		
`*`		`Zeiger`	`Skalar`	`Zeiger auf vollständigen Typen`
`&`		`Objekt`	`Zeiger`	`Vollständiger Typ, Operand muss zuweisbar sein`
`sizeof`		`Objekt`	`int`	`Vollständiger Typ`

Tabelle 1: Operatoren in TinyC

4.2 Gültigkeitsbereiche (Scopes)

Jeder Block öffnet einen neuen Gültigkeitsbereich. Funktionen eröffnen implizit einen neuen Gültigkeitsbereich zu welchem auch die Parameter gehören. Innere Blöcke können Variablen gleichen Namens in äußeren Blöcken verdecken:

```
int foo(int x, int y) { // Vereinbarung 1
    int y;          // kein gültiges C, aber gültiges TinyC
    {
        int x = 1; // Vereinbarung 2
        x = x + 1; // x bezieht sich auf Vereinbarung 2
    }
    return x; // x bezieht sich auf Vereinbarung 1
}
```

Funktionen und Variablen dürfen textuell jeweils nur nach Ihrer Deklaration verwendet werden. Funktionen dürfen beliebig oft deklariert, jedoch höchstens einmal definiert werden. Falls eine Funktion mehrfach mit unterschiedlicher Signatur definiert oder deklariert wird, soll ein Fehler gemeldet werden [1].

4.3 Diagnostic

Falls ein Problem in dem Programm festgestellt wird (beispielsweise eine unbekannte Variable) wird eine Benachrichtigung an den Benutzer bzw. Programmierer ausgegeben. Dies wird durch eine Instanz der `Diagnostic`-Klasse, die dem Konstruktor der `Compiler`-Klasse übergeben wird, realisiert. Jede Benachrichtigung entspricht einem Fehler, der von Ihrer Analyse erzeugt wird. Der Text der Nachricht an sich wird von uns nicht getestet. Allerdings erwartet jede Nachricht eine genaue Position im Quelltexte des Eingabeprogrammes. Wenn ein Fehler während der Prüfung der statischen Semantik auftritt, ist ein Fehler (mittels `Diagnostic.printError`) und der exakten Stelle des Fehlers im Quellprogramm auszugeben. Sie müssen den ersten Fehler in einem Programm als erstes ausgeben. Es steht Ihnen frei, danach weitere Fehler auszugeben. Nach der Ausgabe des Fehlers darf sich Ihr Programm beliebig verhalten, da es von den Tests sofort abgebrochen wird. Im Falle einer undefinierten Variable sieht dies z.B. so aus:

```
int foo() {
    return 42 + v;
              /* ^ */
}
```

Hier ist `v` unbekannt und es muss ein Fehler mit der exakten Stelle von `v` erzeugt werden (hier also `test.c:2:17`). Ist ein Argument eines Operators ungültig, so ist die Position des Operators auszugeben:

```
int* foo(int* ptr) {
    return ptr * 42;
             /* ^ */
}
```

Da es ungültig ist einen Zeiger mit `42` zu multiplizieren, muss hier also eine Fehlermeldung an der Stelle des Multiplikationsoperators ausgegeben werden (hier also `test.c:2:16`). Im Falle einer Anweisung ist die Stelle der Anweisung entscheidend:

```
void foo(char c) {
    return c;
 /* ^ */
}
```

In diesem Beispiel erwartet `return` keine `Expression`, da der Rückgabetyp der Funktion `void` ist, und daher muss die Position des `return`-Tokens ausgegeben werden.

5 Codegenerierung (9 Punkte)

Die letzte Phase des Compilers ist die Codegenerierung. Sie soll beim Aufruf von `generateCode` in der Klasse `Compiler` ausgeführt werden.

5.1 Einschränkungen von TinyC

Funktionen in TinyC können maximal vier Parameter besitzen. Dies hat zur Folge, dass auch nur maximal vier Argumente bei einem Funktionsaufruf übergeben werden können. Somit passen alle Argumente an eine Funktion in die Register `$a0` bis `$a3`. Sie können davon ausgehen, dass niemals Programme übersetzt werden, bei denen mehr Argumente bei Funktionen vorkommen. Beachten Sie allerdings, dass Sie die Aufrufkonventionen einhalten müssen.
Damit auch für Ausdrücke einfach Code erzeugt werden kann ist sichergestellt, dass die Anzahl der temporär benötigten Register niemals 10 überschreitet. Somit können alle temporären Ergebnisse in den Registern `$t0` bis `$t9` abgelegt werden.
Für das `ErrorStatement` kann kein Code erzeugt werden, und falls dies versucht wird, soll eine `IllegalStateException` geworfen werden.

[1] Anders als in C ist die Semantik von `int f();` dass `f` eine Funktion ist die keine Argumente nimmt.

5.2 Implementierung

Das Paket `mipsasmgen` stellt Hilfsfunktionen und Hilfsklassen für die Codeerzeugung bereit. Die Klasse `MipsAsmGen` stellt hierbei die Hauptklasse für die Codeerzeugung dar. Sie verfügt über die Möglichkeit, einzelne Befehle für das Textsegment sowie Daten und andere Deklarationen im Datensegment zu erzeugen. Dabei wird automatisch (durch überladene Methoden sichergestellt) das entsprechende Segment ausgewählt. Zudem kann man komfortabel Marken (für Text- und Datensegment) erzeugen, welche garantiert eindeutig sind (`makeTextLabel` etc.). Die erzeugten Marken können mittels `emitLabel` explizit im Code platziert werden. Erzeugt man allerdings Daten für das Datensegment, werden die Marken sofort automatisch gesetzt.
Einzelne Befehlsklassen werden mittels *Enum*s repräsentiert. Die Methoden des Assembler-Generators für die Ausgabe sind entsprechend überladen. Die einzelnen Aufzählungseinträge heißen wie die entsprechenden MIPS-Befehle (abgesehen von Großschreibung).
Betrachten wir zum besseren Verständnis ein kleines Beispiel:

```
int main() {
    return 1;
}
```

In dem gezeigten Beispiel besteht das kleine TinyC-Programm aus einer `main` Funktion, welche den Wert 1 zurück gibt. Übersetzen wir nun dieses kleine Programm mittels der Assembler-Generator-Klasse:

```
MipsAsmGen gen  = new MipsAsmGen(System.out);
TextLabel  main = gen.makeTextLabel("main");
gen.emitLabel(main);
gen.emitInstruction(ImmediateInstruction.ADDIU, GPRegister.V0, GPRegister.ZERO, 1);
gen.emitInstruction(JumpRegisterInstruction.JR, GPRegister.RA);
```

Zuerst wird eine neue Instanz mittels des Konstruktors und eines `PrintStreams` erzeugt (welcher in Ihrer Implementierung schon an die entsprechende Methode übergeben wird – siehe *JavaDoc*). Danach folgt die Definition unserer Methode `main`. Mit der Hilfe eines neuen Textlabels haben wir eine Marke auf die Methode zur Verfügung. Wir geben das Label aus und erzeugen Code für den Körper der Methode. Das Speichern der Konstante 1 im Register `$V0` wird durch eine Addition mit dem `$zero` Register umgesetzt. Danach wird noch ein Rücksprung zum Register `$RA` erzeugt um die Funktion zu beenden. Der erzeugte Code ist im Folgenden gezeigt:

```
    .text
main:
    addiu   $v0, $zero, 1
    jr $ra
```

5.3 Regeln zur Codeerzeugung

- **Typgrößen:** Bei MIPS wird der Typ `int` auf `word` und `char` auf `byte` abgebildet.
- **Typumwandlung:** Operanden von Typ `char` werden außer als Operand von `&` und `sizeof` vor der Ausführung der Operation nach `int` umgewandelt.
- **Zuweisungen:** Ist der linke Operand der Zuweisung vom Typ `char`, so wird der neue Wert vor der Zuweisung auf die Zielgröße abgeschnitten. Das Ergebnis einer Zuweisung ist der neue Wert des Objekts ihrer linken Seite. Das selbe gilt für **Rückgabewerte** und **Funktionsaufrufe** (vgl. 4.1).
- **Vergleiche:** Die Vergleichsoperatoren liefern als Ergebnis die Werte `0` (falsch) und `1` (wahr).
- **Bedingungen:** In Bedingungen werden Werte ungleich `0` oder einem Nullzeiger als wahr gewertet.
- **sizeof:** Der `sizeof`-Operator gibt die Größe des Operanden in Byte zurück. Bei den meisten Operanden ist dies die Größe des Typen des Operanden in Byte. Bei einem Zeichenkettenliteral ist es die *Länge* der Zeichenkette einschließlich des abschließenden NUL-Zeichens.[2]

[2]Eigentlich ist der Typ eines Zeichenkettenliterals `char[N]` (mit `N` Anzahl der Zeichen der Zeichenkette), aber zur Vereinfachung können Sie, abgesehen von der Behandlung in sizeof, den Typ `char*` verwenden.

Nutzung des Compilers

Nachdem Sie alle Teile des Projektes implementiert haben, können Sie Ihren Compiler nutzen um TinyC Programme zu übersetzen. Der Compiler erzeugt eine Ausgabedatei mit dem Namen *FileName.s*, wobei *FileName* für den Eingabedateinamen steht (z.B. `test` ohne Dateiendung). Sie können auch eine Ausgabedatei mittels `-o FileName` angeben, wobei *FileName* für den Ausgabedateinamen steht. Um den Übersetzer von der Befehlszeile aus zu starten, ist ein Shellskript mit dem Namen `tinycc` beigelegt. Sie können Ihren Code anschließend mit dem Simulator MARS selbst ausführen. Hierfür ist das Shellskript `runmars` beigelegt.

Projekt aufsetzen

Um das Projekt in Eclipse bearbeiten zu können, müssen Sie erst das Depot auschecken und das Projekt importieren.

1. Klonen Sie das Projekt in einen beliebigen Ordner:
   `git clone https://prog2scm.cdl.uni-saarland.de/git/project6/$NAME /home/prog2/project6`
   wobei Sie $NAME durch ihren CMS-Benutzernamen ersetzen müssen.
2. Benutzen Sie *Import*, ein Unterpunkt des *File* Menüeintrags in Eclipse, um den Importierdialog zu öffnen.
3. Wählen Sie „Existing project into workspace“ aus und benutzen Sie den neuen Dialog um den Ordner des Projekts (siehe Punkt 1) auszuwählen.

Hinweise

- Legen Sie neue Dateien immer in das Paket `src/tinycc/implementation` (oder Subpakete). Von uns vorgegebene Interfaces dürfen nicht verändert werden. Sie dürfen in den abstrakten Klassen Methoden hinzufügen, allerdings nichts darin vorgegebenes verändern.

Viel Erfolg!