TinyC-Compiler in Java (25 Punkte + 1 Bonuspunkt)

Ihre Aufgabe bei diesem Projekt ist es, einen Compiler zu vervollständigen. Er akzeptiert die Sprache TinyC und generiert Vorbedingungen für die Korrektheit des Programms.
Das Projekt ist in Teilaufgaben aufgeteilt:

- Konstruktion eines abstrakten Syntaxbaumes (7 Punkte)
- semantische Überprüfung des Programms (Namens- und Typanalyse, 9 Punkte)
- das Aufstellen von Formeln für (schwächste) Vorbedingungen (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.

1.1 TinyC Beispiel

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

```
int globaleVariable;

int foo();

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

int foo() {
    return globaleVariable + 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 | GlobalVariable
GlobalVariable        := Type Identifier ';'
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.

genVerificationConditions Erzeugt eine Formel für die Korrektheit des aktuellen Programms.

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.

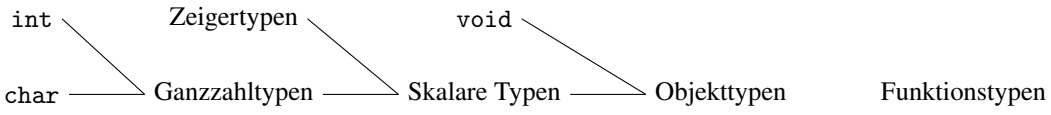

Abbildung 2: Typhierarchie von TinyC

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`.
- **Deklarationen:** Es dürfen keine Variablen mit dem Typen `void` deklariert werden.
- **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 (Schwächste) Vorbedingung (9 Punkte)

Die Aufgabe besteht darin, eine Formel zu generieren, die genau dann allgemeingültig ist, wenn alle (schwächsten) Vorbedingungen eingehalten werden.

5.1 Erweiterung der Grammatik

Um dem Compiler die zu überprüfenden Bedingungen zu kommunizieren, wird die Grammatik wie folgt **erweitert**:

```
Statement         := Block | EmptyStatement | ExpressionStatement
                   | IfStatement | ReturnStatement | WhileStatement
                   | AssertStamtement | AssumeStatement
AssertStatement   := '_Assert' '(' Expression ')' ';'
AssumeStatement   := '_Assume' '('  Expression ')' ';'
WhileStatement    := 'while' '(' Expression ')' Statement
                   | 'while' '(' Expression ')'
                              '_Invariant' '(' Expression ')' Statement
```

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

```
                  | 'while' '(' Expression ')'
                              '_Invariant' '(' Expression ')'
                              '_Term' '(' Expression ')' Statement
BinaryOperator        := '=' | '==' | '!=' | '<' | '>' | '<=' | '>=' | '+' | '-' | '*' | '/'
                         |'||' | '&&'
UnaryOperator         := '*' | '&' | 'sizeof' | '!'
```

Ein `_Assert` Statement beinhaltet eine Bedingung, die bewiesen werden soll. Die Bedingung innerhalb eines `_Assume` Statements darf als wahr von Ihnen angenommen werden und für den Beweis genutzt werden. Um Programme mit Schleifen partiell korrekt zu beweisen, muss eine Schleifeninvariante (`_Invariant`) an Schleifen annotiert sein. Um Programme mit Schleifen total korrekt zu beweisen muss neben der Schleifeninvariante auch eine Terminierungsbedingung (`_Term`) angegeben sein.
Ausdrücke sind um die Operatoren *und* (`&&`), *oder* (`||`) und *nicht* (`!`) bereichert. Sie entsprechen den logischen Operatoren $\wedge$, $\vee$ und $\neg$.
Die Operatoren verhalten sich bei der Ausgabe wie normale binäre bzw. unäre Operatoren. Ein eingelesenes `_Assert(x > 0);` wird als `_Assert((x > 0));` ausgegeben, die anderen neuen Statements funktionieren analog.
Erweitern Sie Ihren AST-Aufbau so, dass er diese erweiterte Grammatik akzeptiert.

5.1.1 Erweiterung der semantischen Analyse

Passen Sie ihre semantische Analyse so an, dass sie die neuen Statements mit überprüft. Für die neuen Operatoren müssen Sie nur Ganzzahlen als Operanden unterstützen, alle drei Operatoren haben den Ergebnistypen `int`.

5.2 Formelaufbau

Ihre Aufgabe ist es, die Funktion `public Formula genVerificationConditions()` in `Compiler.java` zu implementieren. Diese Funktion soll eine `Formel` zurückgeben, die genau dann allgemeingültig ist, wenn alle `_Asserts` im Programm bewiesen werden können. Die Formelklassen haben wir Ihnen bereits vorgegeben, sie finden diese Klassen im Paket `tinycc.logic`. Um die Nutzung der vorgegebenen Klassen zu demonstrieren, ein kleines Beispiel: Abbildung 3 zeigt ein TinyC Programm bei dem man annehmen darf, dass y den Wert 5 hat und das nachdem es x auf 5 setzt überprüft, ob $x == y$ gilt.

Beispieleingabe:

```
int f(int x, int y) {
  _Assume(y == 5);
  x = 5;
  _Assert(x == y);
  return x;
}
```

Zu berechnende Formel: $5 == 5$
Beispielaufbau:

```
IntConst left = new IntConst(5);
IntConst right = new IntConst(5);
BinaryOpFormula Eq = new BinaryOpFormula(BinaryOperator.EQ, left, right);
```

Abbildung 3: TinyC Programm, die zu berechnende Formel und Nutzung der Formelpakete.

Die Formel die Sie für dieses Programm herleiten sollen ist $5 == 5$. Im Skript können Sie in Kapitel 6.6 eine Vorgehensweise finden, Formeln aufzubauen. Um diese Formel mit den vorgegebenen Formelklassen zu erstellen, können Sie zuerst die Konstanten anlegen, und darauf aufbauend den Vergleich.

5.2.1 Einschränkungen auf TinyC für das Testen von Vorbedingungen

Im ganzen Programm …

- … dürfen keine Funktionsaufrufe gemacht werden.

- ...besitzen alle Schleifen Invarianten.
- ...dürfen nur Ganzzahltypen verwendet werden (nur der Rückgabetyp einer Funktion darf `void` sein).
- ...dürfen Zuweisungen nur in `ExpressionStatements` vorkommen und müssen der äußerste Ausdruck sein. Die linke Seite der Zuweisung muss zudem eine Variable sein. Damit ist `x = 8 + z;` erlaubt, `y = (z = 5);` jedoch nicht.
- ...dürfen Ausdrücke keine Division (`/`) enthalten.

Stellen Sie diese Einschränkungen *beim Formelaufbau* sicher und werfen Sie eine Ausnahme, wenn sie verletzt sind. Sie können das oben angegebene Beispiel in den öffentlichen Tests finden.

5.2.2 Ausdrücke zu Formeln übersetzen

Die Sprache TinyC kennt keine boolschen Typen. In den Formeln die aufgebaut werden sollen gibt es jedoch sowohl den Typen `int` als auch `bool`. Die Operatoren werden daher im Formelaufbau wie folgt interpretiert:

$+ - *$	Die Operanden sind vom Typ `int` sind und der Gesamtausdruck hat den Typ `int`.
$> \geq < \leq$	Die Operanden sind vom Typ `int` sind und der Gesamtausdruck hat den Typ `bool`.
$== \neq$	Die Operanden sind beide vom Typ `int` oder beide vom Typen `bool`. Sie geben einen `bool` zurück.
$\wedge \vee \Rightarrow \neg$	Ihr(e) Operand(en) sind vom Typ `bool` und der Gesamtausdruck hat den Typ `bool`.

Um den Typen gerecht zu werden, ist es notwendig, an manchen Stellen von `int` zu `bool` zu konvertieren. Der Ausdruck `1 && 2` sollte zum Beispiel von Ihnen zu einer Formel aufgebaut werden, die die beiden Integer explizit von `int` nach `bool` konvertiert, indem Sie einen $\neq 0$ Vergleich an den erforderlichen Stellen einbauen. Das Ergebnis ist die Formel $(1 \neq 0) \wedge (2 \neq 0)$.
Eine Implizierte Konvertierung von `bool` zu `int` wird nicht verlangt. Wir werden daher keine Ausdrücke wie z.B. `(1 && 2) < 3` testen, weil der Operator $\wedge$ eine boolsche Formel zurückgibt und der Operator $<$ eine Formel vom Typ `int` erwartet.

5.2.3 SMT Solver

SMT Solver können bestimmen, ob eine Formel eine Variablenbelegung hat, sodass die Formel zu wahr auswertet. Das Ergebnis ist *satisfiable*, wenn eine solche Belegung existiert und *unsatisfiable* wenn es keine solche Belegung gibt. Der Solver bietet nicht direkt eine Möglichkeit zu sagen, dass eine Formel für alle Variablenbelegungen zu wahr auswertet. Da wir diese Eigenschaft für unsere Programme aber beweisen möchten, verwenden wir einen beliebten Trick: Wir negieren die Formel und erwarten, dass sie dann unerfüllbar ist. In Tests bekommen Sie daher das Ergebnis *unsatisfiable* wenn die von Ihnen berechnete Formel allgemeingültig ist. Bekommen Sie das Ergebnis *satisfiable* zusammen mit einem Gegenbeispiel (`model`), so ist die Formel nicht allgemeingültig.
In TinyC gibt es Variablenüberdeckung, d.h. es kann mehrere Variablen mit dem selben Namen geben. Der Solver geht allerdings davon aus, dass Variablen mit dem selben Namen identisch sind. Sie müssen daher in Ihrer Implementierung sicherstellen, dass Sie unterschiedliche Variablen mit dem selben Namen disambiguieren. Das kann z.B. erreicht werden, indem Sie eine eindeutige Zahl an den Namen der Variable anfügen wenn Sie die Formel aufbauen.
In diesem Projekt wird der frei verfügbare Theorembeweiser Z3 benutzt.
Hinweise:

- Die Klassen und Interfaces im `tinycc.logic.solver` Paket sind Hilfsklassen, die TinyC-Formeln zu Z3-Formeln übersetzen. Diese Klassen müssen Sie für Ihre Implementierung nicht nutzen oder sich anschauen.
- Im Skript gibt es die Funktion `def e`, die eine Formel erzeugt, die die Menge aller Zustände beschreibt, auf denen `e` definiert ist. Sie dürfen davon ausgehen, dass Ausdrücke immer definiert sind und diese Funktion somit außer Betracht lassen.
- Ob Sie partielle oder totale Korrektheit zeigen sollen, definiert sich dadurch, ob an allen Schleifen neben der Invariante auch eine Terminierungsbedingung steht. Ist das der Fall, bestimmen Sie totale Korrektheit, ansonsten partielle. Um partielle Korrektheit zu zeigen, können Sie die Formel für totale Korrektheit nehmen, und die Terminierungsfunktion weglassen.

- Wir überprüfen nicht, ob ihre Formel wenn sie ausgegeben wird, syntaktisch identisch zu einer von uns vorgegebenen Formel ist. Lediglich die Erfüllbarkeit muss die gleiche sein.
- Sie dürfen für das Testen dieses Aufgabenteils davon ausgehen, dass nur eine definierte Funktion in der Eingabe vorkommt (mehrere Deklarationen sind jedoch möglich).

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/project5/$NAME /home/prog2/project5`
   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.

Z3 außerhalb der VM aufsetzen

Um das Projekt auch außerhalb der VM in Eclipse ausführen zu können, müssen Sie für den Teil zur (schwächsten) Vorbedingung die benötigten Z3 Bibliotheken herunterladen und einrichten (in der VM ist Z3 bereits korrekt eingerichtet). Die Version von Z3 die auch in der VM installiert ist finden Sie hier: `https://github.com/Z3Prover/z3/releases/tag/z3-4.8.6` Wählen Sie das korrekte zip-Archiv für Ihr Betriebssystem aus und extrahieren Sie die heruntergeladene Datei. Gehen Sie in den erstellten `bin` Ordner und kopieren Sie von dort die `libz3` und `libz3java` Bibliotheken (die Dateiendung ist je nach Betriebssystem entweder `.dll`, `.so`, oder `.dylib`) in den `libs` Unterordner in Ihrem geklonten Projekt.
Damit ihr Betriebssystem die Z3 Bibliotheken bei der Ausführung des Programms findet, müssen Sie den `libs` Unterordner von Ihrem Projekt noch in einer Umgebungsvariable eintragen. In Eclipse können Umgebungsvariablen in einer *Run Configuration* angepasst werden. Wenn Sie das Projekt (bzw. die Tests) zum ersten Mal ausführen, wird automatisch eine entsprechende Run Configuration angelegt. Nachdem Sie das getan haben, gehen Sie in der oberen Leiste von Eclipse auf “Run” und dann auf “Run Configurations...”. Wählen Sie die Run Configuration für das Compiler Projekt aus. Gehen Sie in das Tab “Environment”, gehen Sie dann auf “Add...”, und fügen Sie die Variable wie folgt hinzu:

- **Name:** abhängig von Ihrem Betriebssystem:
  - Windows: `PATH`
  - Linux: `LD_LIBRARY_PATH`
  - Mac OS: `DYLD_LIBRARY_PATH`
- **Value:** `libs`

Falls Sie für das Projekt mehrere Run Configurations anlegen (zum Beispiel wenn Sie verschiedene Teile der Tests laufen lassen möchten), müssen Sie die Einstellung für die neuen Konfigurationen wiederholen.

6 Bonus: Break und Continue außerhalb von Schleifen (1 Punkt)

Die folgende Grammatik erweitert TinyC um `break` und `continue`:

```
Statement             := Block | EmptyStatement | ExpressionStatement
                       | IfStatement | ReturnStatement | WhileStatement
                       | BreakStatement | ContinueStatement
BreakStatement        := 'break' ';'
ContinueStatement     := 'continue' ';'
```

Die Anweisungen `break` und `continue` können an beliebigen Stellen im Quelltext auftauchen. Allerdings sind sie nur innerhalb von Schleifen erlaubt. Erweitern Sie Ihre semantische Analyse so, dass Sie bei Programmen, in denen solche Anweisungen an nicht erlaubten Stellen auftauchen, jeweils einen Fehler generieren. Zum Beispiel soll das folgende Programm abgelehnt werden und die Fehlermeldung soll auf die Position der `break`-Anweisung zeigen:

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

Beim Generieren von Vorbedingungen sollen Sie Programme ablehnen, in denen `break` und `continue` vorkommen. Die Bonuspunkte für diese Aufgabe können Sie bereits erhalten, wenn Sie die öffentlichen Tests zu AST Aufbau und semantischer Analyse bestehen.

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.
- Falls Sie in vorgegebenen Interfaces Methoden finden, die mit "BONUS"markiert sind, aber nicht in der Aufgabenstellung vorkommen, dürfen Sie diese Teile gerne bearbeiten, sie werden allerdings nicht getestet und es gibt dafür keine Punkte.

Viel Erfolg!