Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

# Projekt 6

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

## TinyC-Compiler in Java (25 Punkte + 12 Bonuspunkte)

Ihre Aufgabe bei diesem Projekt ist es, einen Compiler zu vervollständigen. Der Compiler übersetzt die Sprache TINYC nach MIPS-Assembler und generiert Vorbedingungen für die Korrektheit des Programms.

Klonen Sie zunächst das Projekt mit

```
git clone https://prog2scm.cdl.uni-saarland.de/git/project6/$NAME
```

Für Teilaufgabe 3b des Projekts muss zusätzlich der SMT-Solver Z3 nachinstalliert werden. Führen Sie dazu in der VM das Installationsskript `scripts/install_z3_posix.sh` aus[1]. Anleitungen für andere Betriebssysteme finden Sie am Ende der Projektbeschreibung.

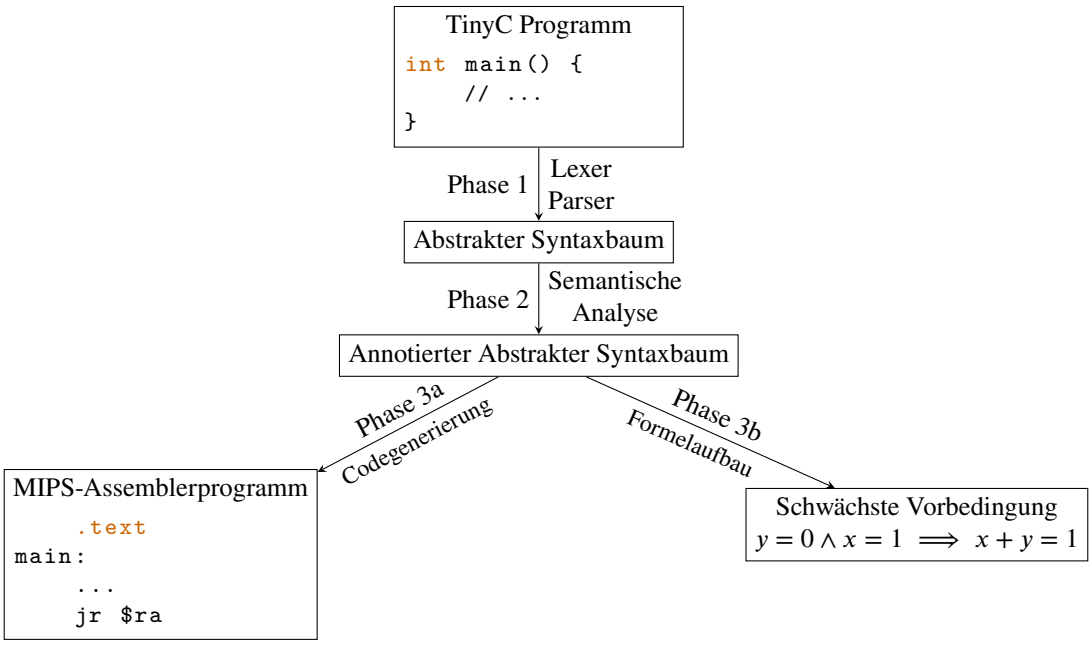
Das Projekt ist in *drei* aufeinanderfolgende Teilaufgaben (+ Bonusaufgaben) aufgeteilt:

1. Konstruktion eines abstrakten Syntaxbaumes **(7 Punkte)**
2. semantische Überprüfung des Programms (Namens- und Typanalyse) **(9 Punkte)**

3a. die Erzeugung von Maschinencode für MIPS **oder alternativ** **(9 Punkte)**

3b. das Aufstellen von Formeln für (schwächste) Vorbedingungen **(9 Punkte)**

Sie dürfen selbst entscheiden ob Sie lieber die Codegenerierung oder die Erzeugung von schwächsten Vorbedingungen für TinyC Programme implementieren wollen. Dieses Projekt ist für 25 Punkte angesetzt. Es genügt also, sich für eine der beiden Varianten zu entscheiden. Implementieren Sie allerdings beides, so können Sie Bonuspunkte erhalten. Folgendes Diagramm verdeutlicht die Pipeline des Compilers:




[1] Die Installation kann, je nach Hardware und VM-Einstellungen, zwischen 2 und 20 Minuten dauern.

# 1 TinyC

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

- Es gibt nur die 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.

```
int globaleVariable;

int foo();

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

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

Abbildung 1: TinyC Beispielprogramm

## 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. Abbildung 1 zeigt ein gültiges TinyC Beispielprogramm.

## 1.2 Grammatik

Abbildung 5 beschreibt die Syntax von TinyC formal. Die 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 Übersicht über den Compiler und die Implementierung

Der Compiler befindet sich in dem Paket `tinycc` und ist dort in mehrere Unterpakete unterteilt:

**driver** Enthält den *Treiber*, der den Übersetzungsprozess steuert. Dieser parst die Befehlszeile und leitet die entsprechenden Argumente an den Rest des Übersetzers weiter. Hier müssen Sie keine Änderungen vornehmen.

**parser** Dieses Paket enthält den Lexer und den Parser für TinyC. Diese werden von uns bereitgestellt.

**diagnostic** Enthält Funktionalität zur Fehlerbehandlung (Fehlermeldungen und Warnungen) welche in der statischen semantischen Analyse des Programms zum Einsatz kommt (siehe Abschnitt 4)

**mipsasmgen** Enthält Funktionalität, um MIPS-Assemblerbefehle zu erzeugen und auszugeben (siehe Abschn. 5)

**logic** Enthält Funktionalität, um Formeln zu erzeugen (siehe Abschnitt 6)

**implementation** Ihre Implementierung soll in diesem Paket (sowie Unterpaketen) implementiert werden.

Ihre Implementierung müssen Sie im Paket `tinycc.implementation` anlegen. Die darin vorgegebene Klasse `Compiler` ist die Hauptklasse für Ihre Implementierung. Diese wird genutzt, um alle Phasen des Übersetzers nacheinander auszuführen. Im Folgenden wird eine kurze Übersicht über die Methoden gegeben:

**getAstFactory** Gibt eine Instanz der `ASTFactory`-Schnittstelle zurück, die intern von der Instanz Ihrer `Compiler` Klasse genutzt wird um den abstrakten Syntaxbaum zu generieren (siehe Abschnitt 3). 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 Syntaxbaum gemäß der durch `getASTFactory` übergebenen Schnittstelle. Diese Methode ist bereits implementiert.

**checkSemantics** Führt die statische semantische Analyse durch, welches die Namens- und Typanalyse umfasst. Annotiert den Syntaxbaum mit Typinformation etc. (siehe Abschnitt 4).

**generateCode** Generiert Code für das aktuelle Programm. (siehe Abschnitt 5)

**genVerificationConditions** Erzeugt eine aussagenlogische Formel anhand der sich die Korrektheit des aktuellen Programms überprüfen lässt. (siehe Abschnitt 6)

Zudem gibt es drei weitere Pakete im Packet `tinycc.implementation`, 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. Nutzen Sie diese Klassen als Basis für ihre Implementierung.

# 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 Tokens einen abstrakten Syntaxbaum aufzubauen. Um dies zu tun, implementieren Sie das Interface `ASTFactory` (aus dem Paket `tinycc.parser`), sowie die Methode `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 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 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 Fabrikmethoden für Statements die in der Syntax von TinyC beschrieben sind gibt es in der `ASTFactory` noch eine Methode `createErrorStatement`. Dieses Statement ist kein Element der Sprache, sondern wird nur erzeugt wenn ein Fehler beim Parsen auftritt.

Die Methoden `createExternalDeclaration` und `createFunctionDefinition` geben `void` zurück. Speichern Sie in Ihrer `ASTFactory` Implementierung eine Liste von `ExternalDeclarations`. Diese beiden Methoden sind dazu da, um erzeugte Deklarationen direkt in diese Liste einzuhängen.

## 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. Ausdrücke die nur aus einem Token stehen sind nicht geklammert. 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 Typanalyse

### 4.1.1 Typen

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 in TinyC 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, sondern ein vollständiger Zeigertyp. Abbildung 2 verdeutlicht die Typhierachie. Unvollständige Typen sind hier grün markiert.

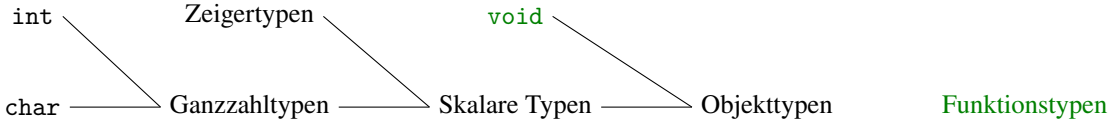



Abbildung 2: Typhierarchie von TinyC

Zeichenkonstanten und Zahlkonstanten haben immer den Typ `int`. Der Typ eines Stringliterals lautet `char*` [2]. Der Typ von geklammerten Ausdrücken ist der Typ des inneren Ausdrucks. Der Typ von nicht-primären Ausdrücken wird im Folgenden noch genauer erläutert.

### 4.1.2 Wertekategorien

Wie in C gibt es in TinyC das Konzept von *L-Values*, welche Ausdrücke klassifizieren, die auf ein Objekt verweisen. Ein Ausdruck in TinyC ist ein L-Value, genau dann wenn der Ausdruck ein Bezeichner ist, oder der Ausdruck eine Indirektion (`*e`), und der Operand ein Zeiger auf einen vollständigen Objekttyp ist.

[2] Eigentlich ist der Typ eines Zeichenkettenliterals `char[N]` (mit `N` Anzahl der Zeichen der Zeichenkette inkl. NUL-Zeichen), aber zur Vereinfachung verwenden wir den Typ `char*` und behandeln Stringkonstanten explizit wenn Operand von `sizeof`.

### 4.1.3 Weitere Hinweise

Bei der Typüberprüfung müssen folgende Eigenschaften beachtet werden:

- **Typanpassung:** Operanden von Typ `char` werden, außer als Operand von `&` und `sizeof`, vor der Ausführung der Operation nach `int` umgewandelt.
- **Bedingungen:** Der Typ von Bedingungen (`if, while ...`) muss skalar sein.
- **Nullzeiger:** Eine Nullzeigerkonstante ist eine Zahlkonstante mit dem Wert `0`.
- **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 muss der linke Operand L-Value sein. Der Typ der Zuweisung ist der Typ des linken Operanden.
- **Funktionsaufrufe:** Die Parameterübergabe bei Funktionsaufrufen unterliegt denselben Regeln wie Zuweisungen. Hierbei ist der Parameter der Funktion die "linke Seite" und der übergebene Wert die "rechte Seite".
- **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.
- **Deklarationen:** Es dürfen keine Variablen mit dem Typen `void` deklariert werden.
- **Funktionsparameter:** Es dürfen keine Parameter mit dem Typen `void` vereinbart werden.
- **Fehler:** Das `ErrorStatement` ist immer wohlgetypt.

Die Tabellen 1 und 2 zeigen alle Operatoren, die Sie implementieren müssen, und beschreibt deren Signaturen.

## 4.2 Gültigkeitsbereiche (Scopes)

Jeder Block öffnet einen neuen Gültigkeitsbereich. Der äußerste Gültigkeitsbereich wird im Folgenden *globaler Gültigkeitsbereich* genannt und beinhaltet *globale Variablen*, sowie *Funktionsdeklarationen* und *-definitionen*. Alle inneren Gültigkeitsbereiche sind *lokale Gültigkeitsbereiche*.

Funktionen dürfen im globalen Gültigkeitsbereich beliebig oft deklariert, jedoch nur einmal definiert werden. Globale Variablen dürfen beliebig oft deklariert werden. Jedoch muss bei jeder erneuten Deklaration der Typ der Variable bzw. die Signatur der Funktion identisch sein [3] . Anders als in C ist die Semantik von `int f();` dass `f` eine Funktion ist, die keine Argumente entgegen nimmt.

In lokalen Gültigkeitsbereichen dürfen Variablen hingegen nur einmal deklariert werden. Funktionsdefinitionen eröffnen einen neuen lokalen Gültigkeitsbereich, zu welchem auch die Parameter gehören. Blöcke eröffnen ebenfalls einen neuen lokalen Gültigkeitsbereich. Das Deklarieren oder Definieren von Funktionen ist hier gemäß syntaktischer Regeln nicht möglich. Des Weiteren können Funktionsparameter und Deklarationen in lokalen Gültigkeitsbereichen eine Variable in einem äußeren Gültigkeitsbereichen verdecken.

Funktionen und Variablen dürfen textuell jeweils nur nach Ihrer Deklaration/Definition verwendet werden. Wird gegen eine der genannten Regeln verstoßen, so soll ein Fehler gemeldet werden. Abbildung 3 zeigt ein Beispiel.

[3] Parameternamen sind nicht Teil der Signatur einer Funktion.

```
int x; // Deklaration von x im globalen Gültigkeitsbereich
int x; // OK, erneute Deklaration im globalen Gültigkeitsbereich
//char x; Fehler: Erneute Deklaration mit unterschiedlichem Typ

int foo(int x, int y); // Funktionsdeklaration
//void foo(int x, int y); // Fehler: Erneute Funktionsdeklaration mit untersch. Signatur

// Funktionsdefinition von foo
int foo(int x, int y) { // Vereinbarung 2 von x, verdeckt globale Variable
    // int y;  Fehler: y bereits in diesem Gültigkeitsbereich deklariert (Parameter)
    {
        int x = 1;        // Vereinbarung 3 von x
        x = x + 1;        // x bezieht sich auf Vereinbarung 3
    }
    return x;             // x bezieht sich auf Vereinbarung 2
}
```

Abbildung 3: Beispiel zu Gültigkeitsbereichen in einem TinyC Programm.

## 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 Codegenerierung ist die letzte Phase des Compilers und gibt ausführbaren MIPS-Maschinencode zum Eingabeprogramm aus. 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` des MIPS-Prozessors. 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.

## 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 global_var;

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

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

```
MipsAsmGen gen  = new MipsAsmGen(System.out);

DataLabel  data = gen.makeDataLabel("global_var");
gen.emitWord(data, 0);

TextLabel  main = gen.makeTextLabel("main");
gen.emitLabel(main);

gen.emitInstruction(MemoryInstruction.LW, GPRegister.V0, data, 0, GPRegister.ZERO);
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 wird die globale Variable `global_var` mittels eines neuen `DataLabels` repräsentiert. Um diese Variable mit dem erzeugten `DataLabel` zu erzeugen, verwenden wir einen Aufruf von `emitWord`, da die Variable vom Typ `int` ist. 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 Laden der Variable wird mittels einer `MemoryInstruction` realisiert, wobei das Zielregister `$v0` ist. Danach wird noch ein Rücksprung zum Register `$RA` erzeugt um die Funktion zu beenden. Der erzeugte Code ist im Folgenden gezeigt:

```
    .data
global_var:
    .word 0
    .text
main:
    lw $v0, global_var
    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.
- **Funktionsaufruf:** Sie müssen den Fall nicht berücksichtigen, dass Funktionsaufrufe weitere Funktionsaufrufe in Ihren Argumenten besitzen.

## 6 Verifizierung durch (schwächste) Vorbedingungen (9 Punkte)

Die Alternative zur Codegenerierung besteht darin, ein Verifikationsverfahren für annotierte TinyC-Programme zu implementieren, welches auf schwächsten Vorbedingungen (Kapitel 6.4 – 6.6 des Skripts) beruht.

### 6.1 Erweiterung der Grammatik

Um dem Compiler die zu überprüfenden Bedingungen zu kommunizieren, wird die Grammatik wie Abbildung 5 beschrieben **erweitert**. 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.

Falls das Programm verschachtelte Schleifen beinhaltet, so kann die Invariante der inneren Schleife syntaktisch nicht ohne Weiteres auf die obere Schranke $k$ der Terminierungsfunktion der umgebenden Schleife Bezug nehmen. Die innere Schleife ”vergisst” also an diesem Punkt notwendigerweise jegliche Aussagen, die sich auf die obere Schranke $k$ beziehen, was den Terminierungsbeweis für die äußere Schleife in der Regel verunmöglicht.

Um dieses Problem zu umgehen beschreiten wir folgenden Weg: Eine Terminierungsbedingung besitzt optional einen Bezeichner, welcher die in der Verifikationsbedingung der Schleife vorkommende obere Schranke $k$ für die Terminierungsfunktion vereinbart. Terminierungsbedingungen öffnen, zwischen Invariante und Schleifenrumpf, einen neuen Gültigkeitsbereich in welchem dieser Bezeichner (falls existent) vereinbart wird. Falls existent besitzt der Bezeichner den Typ `int` und darf (insofern er nicht von einer Deklaration des Programms verdeckt wird) ausschließlich innerhalb von Invarianten verwendet werden, um den Terminierungsbeweis zu ermöglichen.

Ausdrücke werden wie in den Abbildungen 1 und 2 beschrieben um die Operatoren *und* (`&&`), *oder* (`||`) und *nicht* (`!`) **erweitert**. Sie entsprechen den logischen Operatoren $\wedge$, $\vee$ und $\neg$ und operieren auf Ganzzahlen [4]. 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 zunächst so, dass er diese erweiterte Grammatik akzeptiert. Danach passen Sie ihre semantische Analyse so an, dass sie die neuen Statements mit überprüft.

## 6.2 Formelaufbau

Ihre Aufgabe ist es, die Funktion `genVerificationConditions` in `Compiler.java` zu implementieren. Diese Funktion soll eine `Formel` zurückgeben, die nach den Regeln aus Kapitel 6.6 aufgebaut wurde. Hinzu kommen in diesem Projekt `return`-Anweisungen. Die schwächste Vorbedingung für eine `return` Anweisung ist immer erfüllt.

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 4 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) {
  // Berechnete Formel: (y == 5) => (5 == y)
  _Assume(y == 5);
  // 5 == y
  x = 5;
  // x == y (oder (x == y) /\ true)
  _Assert(x == y);
  // true
  return x;
  // true
}
```

Beispielaufbau der berechneten Formel:

```
Variable leftY = new Variable("y", Type.INT);
IntConst left5 = new IntConst(5);
BinaryOpFormula left = new BinaryOpFormula(BinaryOperator.EQ, leftY, left5);


IntConst rightL5 = new IntConst(5);
Variable rightLY = new Variable("y", Type.INT);
BinaryOpFormula rightL = new BinaryOpFormula(BinaryOperator.EQ, rightL5, rightLY);

BoolConst rightR = Boolconst.TRUE;

BinaryOpFormula right = new BinaryOpFormula(BinaryOperator.AND, rightL, rightR);

Formula imp = new BinaryOpFormula(BinaryOperator.IMPLIES, left, right);
```

Abbildung 4: TinyC Programm, die zu berechnende Formel und Nutzung der Formelpakete.

Die Formel die Sie für dieses Programm herleiten sollen ist beispielsweise $((y == 5) \Rightarrow (5 == y))$. Um diese Formel mit den vorgegebenen Formelklassen zu erstellen, können Sie zuerst eine Konstante und eine Variable anlegen, und daraus eine Formel die die Gleichheit ausdrückt erstellen. Das selbe für die andere Seite der Impli-

[4] In C müssen beide Operanden lediglich skalar sein, jedoch entfallen Zeiger in dieser Teilaufgabe.

kation. Danach erstellen Sie für die rechte Seite die Konjunktion mit der booleschen Konstante für "wahr". Wenn Sie beide Subformeln gebaut haben, kann daraus die Implikation erstellt werden.

### 6.2.1 Einschränkungen auf TinyC für das Testen von Vorbedingungen

Das Programm hat genau eine Funktionsdefinition und beliebig viele Funktions- oder Variablendeklarationen. Die definierte Funktion markiert den Eintrittspunkt für den Verifizierer. Sie berechnen also vc(`B` | true) $\wedge$ pc(`B` | true), wobei `B` der Rumpf der Funktion ist. In dieser Funktion…

- …dürfen keine Funktionsaufrufe gemacht werden.
- …besitzen alle Schleifen Invarianten.
- …dürfen nur Ganzzahltypen verwendet werden (keine Nebeneffekte).
- …dürfen Zuweisungen nur in `ExpressionStatements` vorkommen und müssen der äußerste Ausdruck sein. Die linke Seite der Zuweisung muss zudem ein Bezeichner 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 beliebige Ausnahme, wenn sie verletzt sind. Sie können das oben angegebene Beispiel in den öffentlichen Tests finden.

### 6.2.2 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.

### 6.2.3 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, um Formeln als allgemeingültig zu erkennen.

**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 aufgrund vorheriger Namensanalyse immer definiert sind und diese Funktion somit außer Betracht lassen bzw. durch *wahr* ersetzen.
- 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 bzw. durch *wahr* ersetzen.
- Wir überprüfen nicht, ob ihre Formel syntaktisch identisch zu einer von uns vorgegebenen Formel ist. Lediglich die Erfüllbarkeit muss die gleiche sein.

# 7 Bonusaufgabe: Break und Continue

In dieser Bonusaufgabe wird TinyC um `break` und `continue` **erweitert** (siehe Abbildung 5). Die Anweisungen können an beliebigen Stellen im Quelltext auftauchen. Allerdings sind sie nur innerhalb von Schleifen erlaubt.

## 7.1 Semantische Analyse (1 Bonuspunkt)

Erweitern Sie Ihre semantische Analyse so, dass Sie bei Programmen, in denen solche Anweisungen an nicht erlaubten Stellen auftauchen, 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;
}
```

Die Bonuspunkte für diese Teilaufgabe können Sie bereits erhalten, wenn Sie die *öffentlichen* Tests zu AST Aufbau und semantischer Analyse bestehen.

## 7.2 Codegenerierung (2 Bonuspunkte)

Während `break` eine Schleife sofort abbricht und zur Anweisung direkt hinter der umgebenden Schleife übergeht, sorgt `continue` dafür, dass sofort zur Schleifenbedingung zurückgesprungen wird. Implementieren Sie die Codegenerierung für `break` und `continue`. Hierfür ist macht es Sinn, für jede Schleife Sprungmarken zu erzeugen, welche für `break` und `continue` an die jeweils richtige Stelle verweisen.

## 7.3 Verifikation (2 Bonuspunkte)

Um auch Programme mit `continue` verifizieren zu können, müssen die Verifizierungs- und Vorbedingungen für dieses Konstrukt erst formal definiert werden. Wird ein `continue` gesichtet, so wird die aktuelle Nachbedingung verworfen und zur Formel $i \wedge 0 \leq t \leq k$ zurückgesetzt, wobei $i$ die Invariante der umgebenden Schleife ist, $t$ der Terminierungsterm und $k$ die obere Schranke für die Terminierungsfunktion. Wir ”vergessen” also die Nachbedingung und tun so, als ob wir vom Ende der Schleife starten.

$$\text{vc}(\texttt{continue};|N) = \text{true} \qquad \text{pc}(\texttt{continue};|N) = i \wedge 0 \leq t \leq k$$

Um auch `break` zu erlauben, darf die Schleifenbedingung nach der Schleife nicht mehr als falsch angenommen werden, falls die Schleife ein nicht verschachteltes, das heißt nicht auf eine verschachtelte Schleife bezogenes, `break`

beinhaltet. Außerdem terminiert die Schleife bei Ausführung des `break` sofort, weswegen der Teil des Terminierungsbeweises ignoriert werden kann. Ansonsten verhält sich `break` analog zu `continue`:

$$\text{vc}(\texttt{break;} \mid N) = \text{true} \qquad \text{pc}(\texttt{break;} \mid N) = i$$

$$\text{pc}(\texttt{while (e) s;} \mid N) = \begin{cases} \text{vc}(\texttt{s} \mid i \wedge 0 \leq t \leq k) \\ \wedge\, (i \Rightarrow N) \\ \wedge\, (e \wedge i \wedge 0 \leq t \leq k+1 \\ \quad \Rightarrow \text{pc}(\texttt{s} \mid i \wedge 0 \leq t \leq k)) \end{cases} \quad \text{falls s ein nicht verschachteltes } \texttt{break} \text{ beinhaltet.}$$

$$\begin{cases} \text{vc}(\texttt{s} \mid i \wedge 0 \leq t \leq k) \\ \wedge\, ((i \wedge \neg e) \Rightarrow N) \\ \wedge\, (e \wedge i \wedge 0 \leq t \leq k+1 \\ \quad \Rightarrow \text{pc}(\texttt{s} \mid i \wedge 0 \leq t \leq k)) \end{cases} \quad \text{sonst}$$

# 8 Wegweiser und Tipps

Dieses Projekt stellt das letzte und auch anspruchsvollste Projekt der Vorlesung dar. Zur Orientierung geben wir Ihnen einige Tipps die Ihnen das (zeitliche) Management des Projekt erleichtern sollen:

**Allgemein** Schreiben Sie eigene Tests, mit denen Sie die einzelnen Phasen Ihrer Implementierung regelmäßig auf Korrektheit überprüfen können. In den public Tests finden sich Beispiele von denen Sie sich inspirieren lassen sollten. Die Semantik von C bzw. TinyC kann manchmal überraschend oder unintuitiv sein. Stellen Sie also Fragen sollten diese aufkommen. Die Klassen `CompilerTests` und `FatalDiagnostic` besitzen außerdem Debuggingfelder die sie bei Bedarf auf `true` setzen können um sich beispielsweise Ihren übersetzten Assemblercode oder Ihre Formel für die Verifikationsbedingung ausgeben zu lassen.

**AST** Die Generierung des abstrakten Syntaxbaums ist der einfachste Teil des Projekts. Die Klassenstruktur die Sie für den abstrakten Syntaxbaum anlegen kann Ihnen allerdings die weitere Bearbeitung des Projekts erleichtern oder erschweren. Überlegen Sie sich zunächst wie Sie die syntaktischen Elemente sinnvoll in Ihrer Klassenhierarchie repräsentieren. Die syntaktischen Kategorien der formalen Syntax bieten hierfür Orientierung.

*Diese Phase des Projekts sollten Sie in den ersten 3 Tagen bearbeitet haben.*

**Semantische Analyse** Die semantische Analyse gestaltet sich etwas schwieriger, da es eine Menge semantischer Fehler zu überprüfen gibt. Überlegen Sie sich, wie die Typhierarchie von TinyC sinnvoll repräsentiert werden kann. Danach sollten Sie sich überlegen, wie Sie einen bzw. mehrere verschachtelte Gültigkeitsbereiche sinnvoll als Datenstruktur repräsentieren. Beachten Sie auch, dass der Typ einer Variablen, insofern ordnungsgemäß deklariert, vom momentanen Gültigkeitsbereich abhängt. Sobald Sie die grundlegenden Datenstrukturen ausgelegt haben können Sie mit der semantischen Analyse beginnen. Implementieren Sie diese nach und nach für alle Ausdrücke aus Tabelle 1 und 2, und anschließend für alle Anweisungen.

Denken Sie auch daran den AST während dieser Phase mit semantischer Informationen zu annotieren. Der Typ eines Ausdrucks wird beispielsweise an zahlreichen Stellen gebraucht und sollte nicht erneut berechnet, sondern zwischengespeichert werden. Ebenfalls ist es von Vorteil während der Namensanalyse zu speichern, welche Bezeichner in Ausdrücken sich auf welche Deklaration beziehen. Über die Deklarationen können Sie die Bezeichner später disambiguieren.

*Diese Phase des Projekts sollten Sie nach 9 Tagen bearbeitet haben.*

**Codegenerierung** Die Codegenerierung ist der schwierigste Teil der Implementierung. Machen Sie sich zunächst noch einmal klar welche Elemente von TinyC welchen MIPS-Elementen zuzuordnen sind (z.B. Konstanten, String-Literale, globale Variablen etc.).

Speichern Sie lokale Variablen *nicht* in Registern, sondern auf dem Stack. Ordnen Sie dazu jeder Deklaration in einer Funktion einen eindeutigen Offset auf dem Stack zu, damit Sie wissen von welcher Adresse Sie die zugehörige Variable in benutzendem Kontext laden, oder in zuweisendem Kontekt speichern müssen.

Achten Sie darauf bei Funktionsaufrufen die Aufrufkonvention zu beachten. Sie müssen also die caller-save Register vor einem Funktionsaufruf auf dem Stack retten und nach dem Zurückkehren wiederherstellen. Umgekehrt müssen Funktionen die callee-save Register beachten. Allerdings müssen Register, die ihre Implementierung nicht benutzt, auch nicht gerettet werden, was Ihnen Arbeit ersparen kann. Für Ausdrücke müssen die temporären Register verwendet werden. Merken Sie sich während der Codegenerierung für Ausdrücke welche temporären Register bereits belegt sind, damit Sie nicht versehentlich andere Ergebnisse im selben Ausdruck überschreiben.

**Schwächste Vorbedingungen** Der zur Codegenerierung alternative Verifizierer ist ebenfalls anspruchsvoll. Lesen Sie sich hierfür Kapitel 6.4 – 6.6 noch einmal durch und stellen Sie Fragen, falls Ihnen etwas unklar ist. Die Beweise müssen für die Implementierung nicht nachvollzogen werden.

Im Skript finden sich viele Beispiele und genaue Definitionen die beschreiben wie die Formel aufgebaut werden muss. Nachdem Sie Ihren Syntaxbaum erweitert haben, fangen Sie mit den einfachen Fällen an. Erst zum Schluss sollten Sie den Formelaufbau für Schleifen implementieren. Um für verdeckte Bezeichner einen eindeutigen Namen zu erhalten bietet es sich an, die oben genannte Assoziierung von Bezeichnern mit Deklarationen zu verwenden.

# Nutzung auf der Befehlszeile

Nachdem Sie Teil 1 – 3a bzw. 3b des Projektes implementiert haben, können Sie Ihren Compiler bzw. Verifizierer nutzen um TinyC Programme zu übersetzen bzw. verifizieren.

Um das Programm von der Befehlszeile aus zu starten, ist ein Shellskript mit dem Namen `tinycc` beigelegt. Der Compilermodus wird mittels der Option `-c` angegeben, der Verifizierungsmodus mittels `-v`. Die Übersetzung 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. Der Verifizierer gibt die Formel für die schwächste Vorbedingung direkt auf der Konsole aus. Beispiel:

```
./scripts/tinycc -c main.c -o main.s
./scripts/tinycc -v main.c
```

Alternativ können Sie das Programm auch in Visual Studio Code vom Run & Debug Panel aus starten. Den produzierten Maschinencode können Sie mit dem Simulator MARS selbst ausführen. Hierfür ist das Shellskript `runmars` beigelegt, welches auch in Code durch einen Task (*Terminal* → *Run Task...*) aufgerufen werden kann. Beispiel:

```
./scripts/runmars main.s
```

# Z3 außerhalb der VM aufsetzen

Um die Z3 Bibliotheken auf POSIX basierten Betriebssystemen einzurichten, können Sie das Shell-Skript `scripts/install_z3_posix.sh` benutzen. Dies baut und installiert die nötigen Z3 Bibliotheken ins Projektverzeichnis `libs`. Vorraussetzung hierfür ist python3. Auf Ubuntu lassen sich die Java-bindings für Z3 auch einfacher mittels Paket-Manager (`sudo apt-get install libz3-java`) installieren. Für Windows existiert das Batch-Skript `scripts/install_z3_windows.bat` welches die nötigen Bibliotheksdateien herunterlädt und ins `libs` Verzeichnis kopiert.

## Weitere 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!*

**Top-Level Constructs**

```
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
```

**Statements**

```
Statement             := Block | EmptyStatement | ExpressionStatement
                       | IfStatement | ReturnStatement | WhileStatement
                       | AssertStatement | AssumeStatement
                       | BreakStatement | ContinueStatement
BreakStatement        := 'break' ';'
ContinueStatement     := 'continue' ';'
Block                 := '{' (Declaration | Statement)* '}'
Declaration           := Type Identifier ('=' Expression)? ';'
EmptyStatement        := ';'
ExpressionStatement   := Expression ';'
IfStatement           := 'if' '(' Expression ')' Statement ('else' Statement)?
ReturnStatement       := 'return' Expression? ';'
WhileStatement        := 'while' '(' Expression ')' Statement
                       | 'while' '(' Expression ')'
                       '_Invariant' '(' Expression ')' Statement
                       | 'while' '(' Expression ')'
                       '_Invariant' '(' Expression ')'
                       '_Term' '(' Expression (; Identifier)? ')' Statement

AssertStatement       := '_Assert' '(' Expression ')' ';'
AssumeStatement       := '_Assume' '('  Expression ')' ';'
```

**Expressions**

```
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'
```

**Types**

```
Type                  := BaseType '*'*
BaseType              := 'char' | 'int' | 'void'
```

Abbildung 5: Die formale Syntax von TinyC. Zusätzliche Konstrukte aus Phase 3b sind in **grün** dargestellt. Konstrukte aus Bonusaufgaben sind in **orange** dargestellt.

| Operator | Operand | Ergebnis | Hinweis |
|---|---|---|---|
| `*` | Zeiger | Objekt | Zeiger auf vollständigen Typen |
| `&` | Objekt | Zeiger | Vollständiger Typ, Operand muss L-Value sein |
| `sizeof` | Objekt | `int` | Nicht Stringliteral & vollständiger Typ |
| `sizeof` | *Stringliteral* | `int` | Wert ist Länge der Zeichenkette inklusive `'\0'` |
| `!` | Ganzzahl | `int` | |

Tabelle 1: Unäre Operatoren in TinyC. Zusätzliche Operatoren aus Phase 3b sind **grün** markiert.

| Operator | L. Operand | R. Operand | Ergebnis | Hinweis |
|---|---|---|---|---|
| `+` | Ganzzahl | Ganzzahl | `int` | |
| `+` | Zeiger | Ganzzahl | Zeiger | Zeiger auf vollständigen Typen |
| `+` | Ganzzahl | Zeiger | 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` | |
| `/` | Ganzzahl | Ganzzahl | `int` | |
| `== !=` | Ganzzahl | Ganzzahl | `int` | |
| `== !=` | Zeiger | Zeiger | `int` | Identischer Zeigertyp / void* / Nullzeigerkonstante |
| `< > <= >=` | Ganzzahl | Ganzzahl | `int` | |
| `< > <= >=` | Zeiger | Zeiger | `int` | Identischer Zeigertyp |
| `=` | Skalar | Skalar | Skalar | Linke Seite muss zuweisbarer L-Value sein |
| `&&` | Ganzzahl | Ganzzahl | `int` | |
| `\|\|` | Ganzzahl | Ganzzahl | `int` | |

Tabelle 2: Binäre Operatoren in TinyC. Zusätzliche Operatoren aus Phase 3b sind **grün** markiert.