Bonus zu TinyC-Compiler in Java (9 Bonuspunkte)

In diesem Projekt gibt es die Möglichkeit Bonuspunkte zu erwerben. Die Bonusaufgaben sind in zwei Kategorien unterteilt: automatisiert getestete und mündlich abgenommene.

Grammtik

Folgende Grammatik beschreibt die Syntax von TinyC inklusive aller Bonusteile:

```
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
                       | BreakStatement | ContinueStatement
Block                 := '{' (Declaration | Statement)* '}'
BreakStatement        := 'break' ';'
ContinueStatement     := 'continue' ';'
Declaration           := Type Identifier ('=' Expression)? ';'
EmptyStatement        := ';'
ExpressionStatement   := Expression ';'
IfStatement           := 'if' '(' Expression ')' Statement ('else' Statement)?
ReturnStatement       := 'return' Expression? ';'
WhileStatement        := 'while' '(' Expression ')' Statement
Expression            := BinaryExpression | PrimaryExpression | UnaryExpression
                       | ConditionalExpression | PostfixExpression
BinaryExpression      := Expression BinaryOperator Expression
BinaryOperator        := '=' | '==' | '!='  | '<' |  '+' | '-' | '*' | '/'
                       |'||' | '&&'
ConditionalExpression := Expression '?' Expression ':' Expression
PostfixExpression     := Expression PostfixOperator
PostfixOperator       := '(' ExpressionList? ')' | '++' | '--'
ExpressionList        := Expression (',' Expression)*
PrimaryExpression     := CharacterConstant | Identifier | IntegerConstant
                       | StringLiteral | '(' Expression ')'
UnaryExpression       := UnaryOperator Expression
UnaryOperator         := '*' | '&' | 'sizeof' | '++' | '--' | '!'
Type                  := BaseType '*'*
BaseType              := 'char' | 'int' | 'void'
```

Eine Beschreibung der Syntax der Grammatik finden Sie im Hauptprojektdokument.

1 Automatisch Getestete (4 Punkte)

1.1 Break und Continue außerhalb von Schleifen (1 Punkt)

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;
}
```

1.2 Kurzauswertung (1 Punkte)

Integrieren Sie die Operatoren `!`, `&&` und ||. Im Gegensatz zu den anderen Operatoren werden `&&` und || faul ausgewertet. Das heißt der rechte Operand wird nicht ausgewertet, wenn das Ergebnis der Operation schon nach Auswerten des linken Operanden feststeht. Alle drei Operatoren erwarten skalare Operanden. Der Wert eines Operanden ungleich Null (`0`) wird als *wahr* gewertet. Ihr Ergebnis ist jeweils `int`. Wobei *wahr* mit `1` repräsentiert wird und *falsch* mit `0`.

1.3 Bedingter Ausdruck (1 Punkt)

Implementieren Sie den bedingten (ternären) Operator `c ? x : y`. In der semantischen Analyse muss sichergestellt werden, dass die Bedingung skalar ist. Die Typen des zweiten und dritten Operanden müssen vollständige Objekttypen sein und kompatibel sein. Im generierten Code für den bedingten Ausdruck soll je nachdem ob die Bedingung `c` zu wahr oder falsch auswertet, `x` oder `y` ausgewertet werden. Der bedingte Ausdruck ist nicht L-auswertbar.

1.4 Prä- und Postinkrement/-dekrement Operatoren (1 Punkt)

Implementieren Sie die Prä- und Postinkrement/-dekrement Operatoren `++a`, `a++`, `--a` und `a--`. Diese sollen auf Zeigern sowie auf Variablen mit Ganzzahltyp funktionieren. Beachten Sie, dass die Prä-Formen der Operatoren jeweils den *neuen* Wert ihres Operanden zurückliefern. Die Post-Formen hingegen jedoch liefern den *vorherigen* Wert ihres Operanden.

2 Mündlich abgenommene (bis zu 5 Punkte)

Im Folgenden findet sich eine einfache Optimierungen, welche Sie zum Beispiel implementieren können. Generell gibt es hierbei allerdings keine Einschränkungen. Sie können beliebige Optimierungen integrieren, welche bei einem Aufruf von `performOptimizations()` ausgeführt werden. Anhand der mündlichen Abnahme bestimmt sich dann Ihre Punktzahl. Jedoch können Sie nicht mehr als 5 Bonuspunkte hierfür erhalten.

2.1 Konstantenfaltung

Operatoren, bei denen beide Operanden Konstanten sind, können bereits zur Übersetzungszeit ausgewertet werden. In dieser Aufgabe soll eine Konstantenfaltungsoptimierung implementiert werden, welche durch Aufruf von `performOptimizations()` durchgeführt wird. Es sollen hierbei alle faltbaren Konstanten im gesamten Programm gefaltet werden. Betrachten wir beispielsweise den Ausdruck `x + (42 * 24 + 17 - 1)`. Dieser soll nach der Optimierung zu `x + 1024` umgewandelt worden sein. Entsprechend ist es möglich, Anweisungen mit konstanten Bedingungen zu vereinfachen und den bedingten Sprung zu entfernen, z.B. wird `if (23 < 42) f(); else g();` reduziert zu `f();`.