Programming 2 (SS 2023)
Saarland University
Faculty MI
Compiler Design Lab

Project 6

Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

TinyC-Compiler in Java (25 Points + 19 Bonus Points)

Your assignment for this project is to complete a compiler. This compiler translates the language TinyC to MIPS-assembly and emits verification conditions for the correctness of the program.
To be able to edit the project in Visual Studio Code, you first have to checkout the repository and import the project:

1. Clone the project in any folder:
   `git clone ssh://git@git.prog2.de:2222/project6/$NAME.git`
   where $NAME has to be replaced with your CMS username.
2. Open the cloned directory in Visual Studio Code and confirm that the contents are trusted.

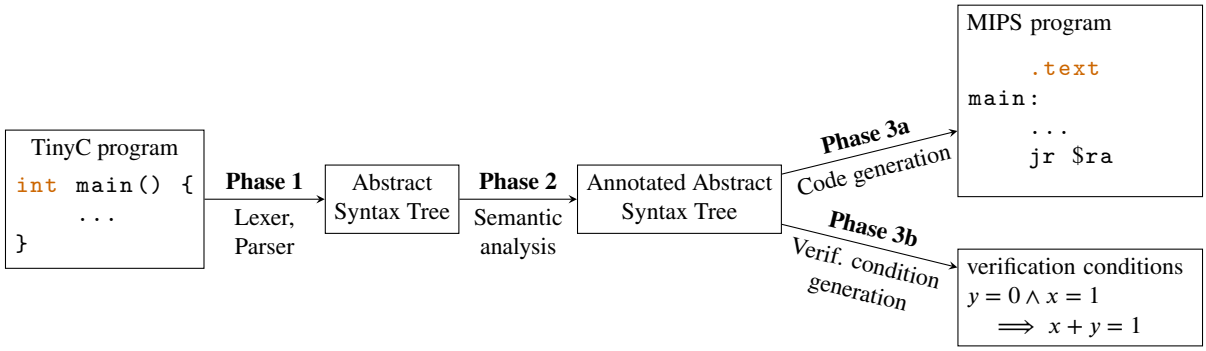
The project is split into three consecutive subtasks (+ bonus tasks):

1. Constructing an abstract syntax tree (AST) **(7 Points)**
2. semantically checking the AST (name and type checking) **(9 Points)**

3a. generating machine code for MIPS **or alternatively** **(9 Points)**

3b. creating formulas for verification conditions **(9 Points)**

You may choose whether you want to implement the code generation or the verification condition generation describing the verification conditions for TinyC programs. It is sufficient to implement only one of the two variants. If you implement both, you may achieve bonus points.
The AST part of the project has an intermediate deadline **at 23:59 on July 11 (CEST)**. Your AST is graded at this date, and will not be graded again afterwards. You are allowed to implement additional parts of the compiler before the intermediate deadline, they will be graded at the final deadline.
The following diagram visualizes the compiler pipeline:



1 TinyC

TinyC is a reduced version of C. The most notable restrictions and deviations with regard to C are:

- There are only three built-in base types: `char`, `int`, and `void`. On top of that, pointers and function types are members of the type system, with the exception of pointers to functions.
- There are no `structs` and `unions`.
- Not all unary/binary operators are available.
- A function may have at most four arguments.

```
int globalVariable;

int foo();

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

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

Figure 1: TinyC example program

1.1 TinyC example

TinyC programs comprise multiple global declarations. These include global function declarations, function definitions and global variables. Every program is started using the `main` function, which always returns a value of the type `int`. Figure 1 shows a valid TinyC example program.

1.2 Grammar

Figure 5 formally defines the syntax of TinyC. The grammar uses the following syntax:

- `'x'`: The symbol `x` has to appear literally in the input. (Terminal)
- `Bla`: `Bla` is the name of another rule. (Non-terminal)
- `(a b)`: Parentheses are used for grouping, e.g. for a following `*`. (Group)
- `x?`: `x` is optional (0 or 1 times). (Option)
- `x*`: `x` may appear an arbitrary number of times (including 0 times) in succession. (Recurrence)
- `a b`: `a` followed by `b`. (Sequence)
- `a | b`: Either `a` or `b`. (Alternative)

The different operators are listed in their *descending* order of precedence. For example: `a | b* c` stands for either exactly one `a` (and no `c`) or arbitrarily many `b` followed by one `c`.

2 Overview over the compiler and the implementation

The compiler resides in the package `tinycc` and is split into a number of sub-packages:

driver Contains the *driver* that steers the translation process. It parses the command line and passes the corresponding arguments to the rest of the compiler. You do not have to make changes here.

parser This package contains the lexer and parser for TinyC. These are provided by us as well.

diagnostic Contains functionality for handling errors (error messages and warnings), which is required for the static semantic analysis (phase 2, see section 4).

mipsasmgen Contains functionality to generate and output MIPS-assembly (phase 3a, see section 5).

logic Contains functionality to create logical formulas (phase 3b, see section 6).

implementation Your implementation shall be done in this package (and sub-packages).

You shall add your implementation in the package `tinycc.implementation`. The `Compiler` class in this package is the main class for your implementation. It is used to execute all phases of the compiler one after the other. Here a short overview of its methods:

getAstFactory Returns an instance of the `ASTFactory` interface that is used internally by the `Compiler` class to build the abstract syntax tree (phase 1, see section 3). The returned object should be the same every time this method is called.

parseTranslationUnit Uses the given lexer to parse the specified input and builds an abstract syntax tree according to the interface given by `getASTFactory`. This method is already implemented.

checkSemantics Performs the semantic analysis including name and type checking. Annotates the syntax tree with type information (phase 2, see section 4).

generateCode Generates code for the current program (phase 3a, see section 5).

genVerificationConditions Creates a propositional logic formula which can be used to check the correctness of the current program (phase 3b, see section 6).

Furthermore, there are three additional packages in the package `tinycc.implementation` that each contain a base class for your class hierarchy:

Type Your type class, which represents a type in the type system of TinyC.

Expression Your expression class, which represents arbitrary expressions in TinyC.

Statement Your statement class, which represents arbitrary statements in TinyC.

These classes may be extended at will. Merely the names of the classes must not be changed. Use these classes as basis for your implementation.

3 Abstract syntax tree and output (7 Points)

In the first part of the project you build an abstract syntax tree. In addition, you implement the `toString` methods of the associated classes such that it can be tested.
We provide a lexer and a parser. Your task is to build an abstract syntax tree from given the tokens. To do so, implement the interface `ASTFactory` (from the package `tinycc.parser`) as well as the method `getASTFactory` in `Compiler.java`.

3.1 AST construction

The enumeration `TokenKind` describes the various kinds of tokens. Tokens are represented by the class `Token`. Tokens contain information about their position in the source code (`Location`), their kind (`TokenKind` and the getter `getKind()`) and the text (`getText()`), which corresponds to the original text of the token in the source code. The parser is initialized with the lexer and a factory class for the creation of AST nodes (`ASTFactory`).

For every AST node the corresponding method of your implementation is called. It should return the respective class from your hierarchy. The methods are called by the parser during the syntactic analysis of the input. Consider the statement `return y + 3;` that is located in a file `test.c` in line `13` starting at column `19`. The following tokens are emitted:

```
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")
```

The nodes created by `ASTFactory` have to be corresponding instances of the classes `Type`, `Expression` or `Statement`. For the example, conceptually the following methods of `ASTFactory` are called:

```
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);
```

The method `createExternalDeclaration` and `createFunctionDefinition` return `void`. Store a list of `ExternalDeclarations` in your `ASTFactory` implementation. Those two functions shall be used to add new declarations directly to this list.

Optional syntactic constructs Some methods correspond to constructs that are optional. The methods `createAssertStatement`, `createAssumeStatement` and `createAnnotatedWhileStatement` are only needed for the verification phase (section 6), while the methods `createBreakStatement` and `createContinueStatement` are only needed for the bonus exercises (section 7). These methods may be left unimplemented prior to doing these exercises, and do not have to be implemented before the first deadline.

3.2 Output of the compiler

All objects that are returned by your AST factory implementation have to override the `toString` method. The `toString` method should always return the abstract syntax of the corresponding syntactic construct returned by a factory method. It will be used to test your implementation.
The abstract syntax of TinyC is similar to the abstract C0 syntax. For example, the expression `&x + 23 * z`, when constructed by the appropriate factory methods, is printed as:

```
Binary_+[Unary_&[Var_x], Binary_*[Const_23, Var_z]]
```

and the statement

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

is printed as

```
While[Binary_>[Var_i, Const_0],
    Block[
        Binary_=[Var_i, Binary_-[Var_i, Const_1]]
    ]
]
```

In detail, the abstract syntax used for the string representation is specified as follows:

- The three built-in base types are output as `Type_` followed by their name, i.e, `Type_char`, `Type_int` and `Type_void` respectively.
- For pointer types, the abstract syntax is `Pointer[type]`, e.g., `Pointer[Type_void]`.
- For function types, first the return type is written, followed by the comma separated parameter types. For example `void(int, int)` is printed as:

  ```
  FunctionType[Type_void, Type_int, Type_int]
  ```

- Variables are printed as `Var_name`. For example, the variable `abc_def` is printed as:

  ```
  Var_abc_def
  ```

- Integer constants, character constants and string literals are printed as `Const_c`. Character constants and string literals are printed including their respective quotes.

  ```
  Const_42
  Const_'c'
  Const_"abc"
  ```

- Unary expressions are printed as `Unary_op[operand]`. For example, `&x` is printed as:

```
Unary_&[Var_x]
```

- Binary operations are printed as `Binary_op[left, right]`. Note that in TinyC, assignments are treated as binary expression statements, so they follow this syntax as well.

```
Binary_+[Const_2, Var_x]
Binary_=[Var_y, Const_0]
```

- Function calls are printed as `Call[name, args...]`.

```
Call[Var_foo]
Call[Var_bar, Const_42, Var_x]
```

- Declarations are printed as `Declaration_name[type, init]`. If the initialization expression `init` is absent, the syntax is `Declaration_name[type]`.

```
Declaration_x[Type_char]
Declaration_y[Type_int, Const_0]
```

- Expression statements have exactly the same abstract syntax as their expression.
- Blocks are printed using a comma-separated list of their containing statements or declarations. For empty blocks not containing any statements or declarations, this list is empty. Examples:

```
Block[Declaration_x[Type_int], Binary_=[Var_x, Const_2]]
Block[Block[]]
```

- The return statement is printed as `Return[expr]`, if the return expression exists. Otherwise, the abstract syntax is `Return[]`.
- The if statement is printed as `If[cond, cons, alt]`, if the alternative statement exists. Otherwise, the abstract syntax is `If[cond, cons]`.
- The while statement is printed as `While[cond, body]`.

All whitespace is ignored while checking the textual representation, so it does not matter whether you insert spaces or even newlines into the returned string. You are encouraged to do so to improve the visual clarity.

4 Semantic analysis (9 Points)

The following sections present the type system and the scoping of variables in TinyC that you check in your semantic analysis. Your semantic analysis is performed upon calling `checkSemantics` in `Compiler.java`.

4.1 Typing

4.1.1 Types

TinyC supports a subset of C's types. These are arranged in a type hierarchy. There are object types and function types. The types `char` and `int` are integer types. All integers in TinyC are *signed*. Combined with the pointer types they form the scalar types. All scalar types and `void` are object types. There are no function pointers. The type `void` and function types are incomplete types. Notably, `void*` is a pointer type and hence complete. Figure 2 visualizes the type hierarchy.
Character and numeral constants are always of type `int`. The type of a string literal is `char*`.[1] The type of parenthesized expressions is the type of the inner expression. The type of non-primary expressions is further refined in the following.

[1] In C, the type of string literals is `char[N]` (where `N` is the number of characters in the string, including the NULL-character), but for the sake of simplicity, we use the type `char*` and define `sizeof` for string literals explicitly.

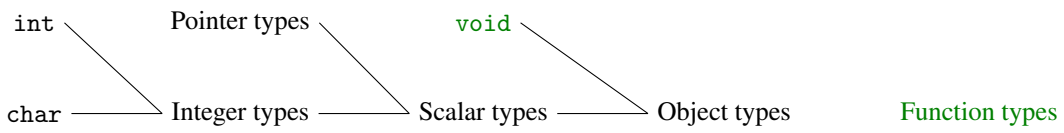


Figure 2: Type hierarchy in TinyC

4.1.2 Value categories

As in C, TinyC has the concept of *L-values*, which classify expressions that reference an object. An expression in TinyC is an L-value, if and only if the expression is an identifier or if the expression is an indirection (`*e`) and the operand is a pointer to a complete type.

4.1.3 Additional remarks

For the type checking, the following properties have to be considered:

- **Automatic conversion:** Operands of type `char` are, except when used as operand for `&` and `sizeof`, automatically converted to `int` before performing the operation.
- **Conditions:** The type of a condition (`if, while, ...`) has to be scalar.
- **Null pointer:** A null pointer constant is an integer constant with the value `0`.
- **Assignments:** For assignments, one of the following has to be true:
  - The types of the operands are identical.
  - Both operands have integer types.
  - Both operands have pointer types and at least one of the two has type `void*`.
  - The left operand has a pointer type and the right operand is a null pointer constant.

  In all cases, the left operand has to be an L-value. The type of the assignment is the type of the left operand.
- **Function calls:** The passing of parameters to function calls has the same rules as assignments. The parameter of the function is considered the "left side" and the passed value the "right side".
- **Return values:** The rules for assignments also apply to `returns`. Hereby, the return type of the function is the "left side", and the returned expression is the "right side". An expression has to be present if and only if the return type is not `void`. It is not necessary to check whether any path to the exit of a non-`void`-function lacks a `return` statement. In that case, the return value is undefined.
- **Declarations:** Variables must not be defined with type `void`.
- **Function parameter:** Function parameters must not be of type `void`.

Table 1 and Table 2 show all operators you have to implement and specify their signatures.

4.2 Scopes

Every block opens a new scope. The outermost scope is called *global scope* from now on and contains *global variables*, as well as *function declarations* and *definitions*. All inner scopes are *local scopes*.

Functions may be declared arbitrarily often inside the global scope, but may be defined only once. Global variables may be declared arbitrarily often. For all following declarations, the type of the variable or the signature of a function respectively, must not change. Note that parameter names are not part of a function's signature. Deviating from C, the semantics of `int f();` is that `f` is a function that does not accept any arguments.

In contrast, variables may be declared only once in a local scope. Function definitions open a new local scope. The function's parameters belong to this local scope. Blocks additionally open new local scopes. Following the

syntactic rules, declaring or defining functions in a local scope is not possible. Furthermore, function parameters and variable declarations in a local scope may hide variables from an outer scope.

Functions and variables may be used, only (textually) after their declaration or definition. If any of the listed rules is not followed, an error shall be emitted. Figure 3 shows an example.

```
int x; // Declaration of x in the global scope
int x; // OK, redeclaration in the global scope
//char x; Error: Redeclaration with a different type

int foo(int x, int y); // Funciton declaration
//void foo(int x, int y); // Error: Redeclaring function with different signature

// Definition of the function foo
int foo(int x, int y) { // 2nd declaration of x, hides global variable
    // int y;  Error: y is already declared in this scope (as parameter)
    {
        int x = 1;        // 3rd declaration of x
        x = x + 1;        // x references the 3rd declaration
    }
    return x;             // x references the 2nd declaration
}
```

Figure 3: Example of scopes in a TinyC program.

4.3 Diagnostic

If a problem in a program is encountered (e.g. an unknown variable), a message to the user / programmer is emitted. This is done via the `Diagnostic` class that the `Compiler` class receives as constructor parameter. Every message corresponds to one error that is emitted by your analysis. The text of the message itself is not tested by us. However, every message expects the precise position in the source code. If an error occurs during the static semantic checking, an error (using `Diagnostic.printError`) and the exact location of the error in the source code is to be printed. You have to first print the first error. Afterwards you may print further errors, if you want. After outputting the first error, your program may behave in any manner, as it is immediately aborted by the tests anyway. Consider the following example, where an undefined variable is used:

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

In this case `v` is unknown and an error containing the exact location of `v` has to be emitted (in this case `test.c:2:17`). If one operand of an operation is invalid, the position of the operator shall be output:

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

As it is invalid to multiply a pointer with 42, an error has to be emitted referencing the location of the multiplication operator (in this case `test.c:2:16`). In case of a statement, the position of the statement is relevant:

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

In this example, `return` does not expect an `Expression`, as the return type of the function is `void`, therefore the position of the `return` token has to be emitted.

5 Code generation (9 Points)

The code generation is the last phase of the compiler and emits executable MIPS machine code corresponding to the input source code. This phase shall be executed upon calling the `generateCode` method in the `Compiler` class.

5.1 Restrictions of TinyC

Functions in TinyC can have at most four parameters. Hence, at most four arguments can be passed to a function call. Therefore, all arguments for a function fit inside the `$a0` to `$a3` registers of a MIPS processor. You may assume that only programs that use functions with at most four arguments are compiled. Remember to comply with the calling convention.

To simplify code generation for expressions, you can assume that the number of temporarily required registers never exceeds ten. Therefore, you can store all temporary results in the registers `$t0` to `$t9`.

5.2 Implementation

The package `mipsasmgen` provides helper functions and classes for the code generation. The class `MipsAsmGen` is the main class for the code generation. It contains the possibility to generate individual instructions in the text segment as well as data and other declarations in the data segment. Thereby, the correct segment is automatically selected (via overloaded methods). Additionally, it is possible to comfortably create labels (for the text and data segment) that are guaranteed to be unique (`makeTextLabel` etc.). The created labels can be placed in the code using `emitLabel`. When creating data inside the data segment, labels are placed automatically.

Individual instruction classes are represented via *Enum*s. The methods of the assembly generator for the output are overloaded correspondingly. The individual enum entries are named like their respective MIPS instructions (up to capitalization).

Consider the following example:

```
int global_var;

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

In the shown example, the TinyCprogram comprises a global variable and a `main` function which loads and returns the value of the variable. The compilation of this example program using the assembly generator class can be thought of as:

```
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);
```

First, a new instance is created by handing the constructor a `PrintStream` (in your implementation this is already done – see the *JavaDoc*). Afterwards, the global variable `global_var` is represented as a new `DataLabel`. To allocate this variable with the created `DataLabel`, we call `emitWord` as the variable is of type `int`. Next, we define the method `main`. By adding a new text label, we have a marker for this method. We emit the label and

generate code for the method. Loading a variable is done using a `MemoryInstruction`, where the target register is `$v0`. Lastly, a returning jump to the register `$RA` exits the method. The generated code is shown in the following:

```
    .data
global_var:
    .word 0
    .text
main:
    lw $v0, global_var
    jr $ra
```

5.3 Code generation rules

- **Type sizes:** For MIPS, the type `int` is mapped to `word` and `char` to `byte`.
- **Automatic conversion:** Operands of type `char` are cast to `int`, except for the operations `&` and `sizeof`.
- **Assignment:** If the left operand is of type `char`, the new value is truncated to the target size. The result of an assignment is the new value of the object on the left side. The same is true for **return values** and **function calls** (cf. subsection 4.1).
- **Comparisons:** The comparison operators return the values `0` (false) and `1` (true).
- **Conditions:** In conditions, all values except `0` or a null pointer are interpreted as true.
- **sizeof:** The `sizeof`-operator returns the size of its operand in bytes. For most operands this is the size of the operand's type in bytes. For string literals it is the length of the string, including the terminating NUL-character.
- **Function call:** You do not have to consider that function calls can have nested function calls as arguments.

6 Verification using verification conditions (9 Points)

The alternative task to code generation is to implement a verification method for annotated TinyC-programs. The method is based on verification conditions (see Chapter 7.4 in the script).
To establish the partial or total correctness of the program `p`, the validity of the formula vc(`p` | *true*) ∧ pc(`p` | *true*) must be established (see Theorem 7.4.11 and Lemma 7.4.13). Your task is to generate this formula for the input program. To this end, you must implement the method `genVerificationConditions` in the `Compiler` class to return this formula.

6.1 Extending the grammar

First, to inform the compiler about the assumed preconditions and the postconditions it should verify, the grammar is extended as described in Figure 5. An `_Assert` statement contains a condition that is to be proven. You may consider the condition inside an `_Assume` statement as true and use it to aid verification. Programs with loops need further information for verification. For partial correctness, only a loop invariant (`_Invariant`) annotation is necessary. Total correctness additionally requires a loop termination function, given via the `_Term` annotation, which may optionally include a an identifier, the purpose of which is explained below. Lastly, the operators *and* (`&&`), *or* (`||`) and *not* (`!`) are added to construct assertion formulas. They represent the logical operators ∧, ∨, and ¬.

To build an AST for the extended syntax, you need to implement the methods `createAssertStatement`, `createAssumeStatement` and `createAnnotatedWhileStatement` in your `ASTFactory` to return a corresponding instance of `Statement`. As before, the `toString` method of these objects should return the abstract syntax of the respective construct:

- Expressions involving the new operators are printed analogously to ordinary binary or unary expressions, e.g., `Unary_![Binary_>[Var_x, Const_0]]`.

- Assume and Assert statements are printed without underscores as `Assume[expr]` and `Assert[expr]`:

```
Assume[Binary_==[Var_x, Const_0]]
Assert[Binary_==[Var_x, Const_0]]
```

- The annotated while statement is printed as `While_identifier[cond, body, inv, term]`, if a termination function with an identifier is given. If only the termination function is given but the identifier is omitted, the abstract syntax is `While[cond, body, inv, term]`. If no termination function is given, the abstract syntax is `While[cond, body, inv]`.

After extending the AST construction, extend the semantic analysis to verify the semantics of expressions with these operators (see Table 1 and Table 2). The extension of the grammar does not affect code generation.

6.2 Total correctness

By specifying termination functions for loops, we can prove the total correctness of loops and programs containing loops. The termination function for a loop (if it exists) is the expression t specified by `_Term`. In a nutshell, we want to prove that the termination function t remains non-negative and strictly decreases in every loop iteration. Since natural numbers cannot decrease forever, the loop must eventually terminate. Formally, we extend vc and pc for loops with termination functions as follows to also prove termination:

$$\text{pc}(\texttt{while (}e\texttt{) \_Invariant(}I\texttt{) \_Term(}t\texttt{) } s \mid Q) = I \land 0 \leq t$$

$$\begin{aligned}\text{vc}(\texttt{while (}e\texttt{) \_Invariant(}I\texttt{) \_Term(}t\texttt{) } s \mid Q) = &\{I \land e \land 0 \leq t \leq k+1 \Rightarrow \text{pc}(s \mid I \land 0 \leq t \leq k)\} \cup \\ &\{\neg e \land I \Rightarrow Q\} \cup \\ &\text{vc}(s \mid I \land 0 \leq t \leq k)\end{aligned}$$

The variable k is used to specify an upper bound for t inside the verification condition and is called the loop bound of the termination function. Intuitively, we show that for an arbitrary value of k, if $t \in [0, k+1]$ before the loop is entered, then $t \in [0, k]$ after the iteration completes. Consequentially, t decreases in each loop iteration, and always remains non-negative. The precondition makes sure that t is initially non-negative.

6.2.1 Nested Loops

Proving the termination of nested loops requires additional syntactical support. Consider the following example:

```
int i = 6;
while (i > 0) _Invariant(1) _Term(i) {
    i = i - 1;
    while (0) _Invariant(I) _Term(0) {}
}
```

When computing the verification conditions for this program, we eventually generate the verification condition $\text{vc}(\texttt{while (0) \_Invariant(}I\texttt{) \_Term(0) \{\}} \mid 0 \leq i \leq k) = I \Rightarrow 0 \leq i \leq k$ (after simplication). However, to prove $I \Rightarrow 0 \leq i \leq k$, the invariant I must make statements about k, yet k does not naturally exist within the program. To resolve this issue, the optional identifier specified by `_Term` introduces a designated variable for the loop bound k of this termination function that can be referred to by nested invariants. Termination of the above program can now be verified:

```
int i = 6;
while (i > 0) _Invariant(1) _Term(i; k) {
    i = i - 1;
    while (0) _Invariant(0 <= i <= k) _Term(0) {}
}
```

The loop bound identifier of `_Term` (if present) is declared in an intermediate scope between the invariant of the loop and the loop body (which itself opens a nested scope within this scope). The identifier is of type `int` and must (as long as it is not hidden by another declaration) only be used inside invariants of nested loops, as it exists only to enable the proof of termination. You must extend your semantic analysis to verify that the identifier is never used in other contexts.

6.3 Formula construction

Your task is to implement the function `genVerificationConditions` in `Compiler.java`. This function shall return a `Formula` that was constructed following the rules from chapter 7.4.2. In addition to the rules from the script, you need to support the `return` statement and while loops with termination functions. The precondition for a `return` statement is *true* and it has no verfication conditions.

$$\text{pc}(\texttt{return}\ expr;\mid Q) = true \qquad \text{vc}(\texttt{return}\ expr;\mid Q) = \{\}$$

We already provide you with the formula classes in the package `tinycc.logic`. The following example demonstrates the usage of the given classes: Figure 4 shows a TinyC program where one may assume that y has the value 5. After setting x to 5 it checks whether $x == y$ holds true.

Example input:

```
int f(int x, int y) {
    // Computed formula: (y == 5) => (5 == y)
    _Assume(y == 5);
    // 5 == y
    x = 5;
    // x == y (or (x == y) /\ true)
    _Assert(x == y);
    // true
    return x;
    // true
}
```

Exemplary construction of the corresponding formula:

```
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);
```

Figure 4: TinyC program, the to be computed formula and usage of the logic package.

The formula that you shall deduce is for example $((y == 5) \Rightarrow (5 == y))$. To create this formula with the given formula classes, you can start by creating a constant and a variable, and then setting up a formula that expresses the equality of those two. The same can be done for the other side of the implication. Afterwards, for the right side you create the conjunction with the boolean constant "true". Upon building both sub-formulas, the implication can be created from them.

6.3.1 Restrictions on TinyC for testing preconditions

The program must have exactly one function definition and may have arbitrarily many function and variable declarations. The defined function marks the entry point for the verifier. You therefore must compute the formula $\text{pc}(\texttt{B} \mid true) \wedge \text{vc}(\texttt{B} \mid true)$ inside the method `genVerificationConditions`, where `B` is the body of the function. In this function...

- ...no function calls are present.
- ...all loops have invariants.
- ...only integer types may be used (no side effects).

- ...assignments shall only be used in `ExpressionStatements` and must be the outermost expression. The left side of the assignment has to be an identifier. Therefore, `x = 8 + z;` is permitted, whereas `y = (z = 5);` is not.
- ...expressions must not contain divisions (`/`).

Ensure that these restrictions are fulfilled *while building the formulas* and throw any exception, if that is not the case. You can find the example from Figure 4 in the public tests.

6.3.2 Translating expressions to formulas

The language TinyC does not have boolean types. The formulas that are constructed, contain both, `int` and `bool`. The operators are interpreted during formula construction as follows:

$+ - *$	The operands are of type `int` and the overall expression has the type `int`.
$> \geq < \leq$	The operands are of type `int` and the overall expression is of type `bool`.
$== \neq$	Both operands are of type `int` or both operands are of type `bool`. They return a `bool`.
$\wedge \vee \Rightarrow \neg$	Their operand(s) are of type `bool` and the overall expression is of type `bool`.

It is sometimes necessary to convert from `int` to `bool` to satisfy the type requirements. Take the expression `1 && 2` for example. You should generate a formula where both integers are explicitly converted from `int` to `bool` by adding $\neq 0$ at the necessary points. The result is the formula $(1 \neq 0) \wedge (2 \neq 0)$.

An implicit conversion from `bool` to `int` is not required. We will not test expressions such as `(1 && 2) < 3`, where the operator $\wedge$ results in a boolean formula and the operator $<$ expects a formula of type `int`.

6.3.3 SMT solver

SMT solvers can detect whether a formula has a variable assignment, such that the formula evaluates to true. The result is *satisfiable*, if such an assignment exists and *unsatisfiable*, if such an assignment does not exist. The solver does not directly check whether a formula evaluates to true for all variable assignments. As we want to prove this property for our programs, we use a common trick: We negate the formula and check whether the result is unsatisfiable. Therefore, if the formula you constructed is universally true, the test result is *unsatisfiable*. Otherwise, you get the result *satisfiable* and a counter example (`model`).

In TinyC variables can be hidden, i.e. there may be multiple variables with the same name. The solver however assumes that variables with the same name are identical. Therefore, you have to ensure in your implementation that different variables with the same name are disambiguated. This can be achieved by appending a unique number to the name of a variable while constructing the formula.

For this project, the freely available theorem prover Z3 is used.

Remarks:

- The classes and interfaces inside the package `tinycc.logic.solver` are helper classes for translating the TinyC-formulas to Z3-formulas. You do not have to use these for your implementation and do not have to take a look at them.
- The script defines the function `def e`, which builds a formula describing all states in which `e` is defined. You may assume that expressions are always defined (due to preceding name checking) and can therefore ignore the function or set it to *true*.
- Whether you shall show partial or total correctness is decided by whether all loops have a termination condition in addition to the invariant. If this is the case, determine total correctness, otherwise partial. To show partial correctness, you may use the formula for total correctness and omit the termination function or replace it with *true*.
- We do not check whether your formula is syntactically identical to a given formula. Merely the satisfiability has to be the same.

7 Bonus for TinyC-Compiler in Java (19 Bonus points)

For this project, it is possible to achieve bonus points. To be eligible for bonus points, you have to pass all public tests for the non-bonus assignments. The bonus assignments are split in two categories: automatically tested and orally evaluated.

7.1 Automatically evaluated (14 Bonus Points)

Verification or Code Generation (9 Bonus Points)

If you implement both phase 3a and 3b of the compiler, you will get points for both phases. If you obtain more than 9 points in both phases, you achieve the rest of the points as bonus points.

Break and Continue (5 Bonus Points)

In this bonus assignment TinyC is **extended** with `break` and `continue` (see Figure 5). The statements can appear at any point of the source code. However, they are only valid inside loops.

Semantic analysis (1 Bonus Point) Extend your semantic analysis to report an error in case the statements are at an invalid location. For example, the following program shall be rejected, and the error message shall point to the position of the `break`-statement:

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

It suffices to pass the *public* tests for the AST construction and semantic analysis to be eligible for this bonus.

Code generation (2 Bonus Points) While `break` immediately exists a loop and jumps to the instruction after the surrounding loop, `continue` leads to directly jumping back to the loop condition. Implement the code generation for `break` and `continue`. It is useful to create labels for every loop, as they can be used as the respective jump targets for `break` and `continue`.

Verification (2 Bonus Points) We first define the verification- and preconditions for `continue`. If a `continue` is encountered, the current postcondition is discarded and the formula $I \wedge 0 \leq t \leq k$ is used instead, where I is the invariant of the surrounding loop, t the termination term and k the upper bound of the termination function. We effectively "forget" the postcondition and act as if we started from the end of the loop. No verification conditions are added.

$$\text{pc}(\texttt{continue;} \mid Q) = I \wedge 0 \leq t \leq k \qquad \text{vc}(\texttt{continue;} \mid Q) = \{\}$$

A `break` statement terminates the enclosing loop immediately when executed, therefore the termination proof can be ignored in this case. Apart from that, `break` behaves analogous to `continue`:

$$\text{pc}(\texttt{break;} \mid Q) = I \qquad \text{vc}(\texttt{break;} \mid Q) = \{\}$$

When we encounter a `break`, we can no longer assume that the loop condition of the enclosing loop is false after the loop is left. Therefore, we have to slightly **change** the verification condition of while loops:

$$\text{vc}(\texttt{while (}e\texttt{) \_Invariant(}I\texttt{) \_Term(}t\texttt{) } s \mid Q) = \begin{cases} \text{vc}(\texttt{s} \mid I \wedge 0 \leq t \leq k) \cup & \\ \{I \Rightarrow Q\} \cup & \text{if s contains} \\ \{e \wedge I \wedge 0 \leq t \leq k+1 & \text{a non-nested } \texttt{break} \\ \quad \Rightarrow \text{pc}(\texttt{s} \mid I \wedge 0 \leq t \leq k)\} & \\ & \\ \text{vc}(\texttt{s} \mid I \wedge 0 \leq t \leq k) \cup & \\ \{\neg e \wedge I \Rightarrow Q\} \cup & \text{otherwise} \\ \{e \wedge I \wedge 0 \leq t \leq k+1 & \text{(regular vc)} \\ \quad \Rightarrow \text{pc}(\texttt{s} \mid I \wedge 0 \leq t \leq k)\} & \end{cases}$$

For loops without termination functions, the verification condition is analogous (without the subformulas involving the termination function).

7.2 Orally evaluated (up to 5 Bonus Points)

In the following a simple optimization is described, that you can implement for example. You are free to integrate any optimizations, which are executed when `performOptimizations()` is called. Your points are determined in the oral evaluation. It is not possible to achieve more than 5 Bonus points for this, though.

Constant folding

Operators, where both operands are constants can be evaluated already at compile-time. For this task you can implement constant folding, that is performed by calling `performOptimizations()`. Thereby all foldable constants in the whole program shall be folded. For example consider the following expression `x + (42 * 24 + 17 - 1)`. This shall be transformed to `x + 1024` after the optimization. Similarly, it is possible to simplify expressions with constant conditions and removing the conditional jump, e.g. `if (23 < 42) f(); else g();` is reduced to `f();`.

8 Guide and tips

This project represents the last and also most challenging project of the lecture. For your orientation we give advice to aid your (time) management of the project:

General Write your own tests and use them to regularly check the individual phases of your implementation for correctness. In the public tests you can find examples that you should take inspiration from. The semantics of C and hence TinyC can sometimes be rather surprising or unintuitive. Therefore, ask questions when they arise. The classes `CompilerTests` and `FatalDiagnostic` have fields for debugging that you can set to `true` to view the translated assembly code or your formulas for the verification conditions.

AST The creation of the abstract syntax tree is the simplest part of the project. The class hierarchy that you build for the abstract syntax tree however can either simplify or complicate the further work on the project. Therefore, think about how to represent the syntactic elements in your class hierarchy in a reasonable way. The syntactic categories of the formal syntax can guide you.

Semantic analysis The semantic analysis is somewhat more difficult as there are many semantic errors to check. Think about how to reasonably represent the type hierarchy of TinyC. Afterwards, consider how to store nested scopes as a data structure. Remember that the type of a variable, if correctly declared, depends on the current scope. As soon as you have set up the basic data structures, you can start with the semantic analysis. Implement

the rules bit by bit for all expressions from Table 1 and Table 2 and afterwards for all statements.

Remember to annotate the AST with semantic information during this phase. The type of an expression is required often and should not have to be recomputed. Furthermore, it is advantageous to store which identifier references which declaration during the name analysis. You later can disambiguate the identifiers via the declarations.

Code generation The code generation is the most challenging part of the implementation. First, bring to your mind which elements of TinyC correspond to which MIPS-elements (e.g. constants, string-literals, global variables, ...).

Do *not* store local variables in registers. Use the stack instead. Associate every declaration in a function with a unique offset on the stack, so that you know from which address the corresponding variable is loaded when used, or stored to in assigning contexts.

Follow the calling convention for function calls. You also have to secure the caller-save registers on the stack before a function call and restore them after returning. Conversely, functions have to take care of callee-save registers. Note that registers that are not used by your implementation don't have to be saved, which can save some work. The temporary registers have to be used for expressions. Memorize during code generation for which expressions which temporary registers are already used, to avoid accidentally overwriting other results in the same expression.

Verification conditions The verifier that you can implement instead of the code generation is also very demanding. Read chapter 7.4 of the script again and ask when questions arise. You do not have to comprehend the proofs for your implementation.

The way the formula is constructed naturally suggests to implement this construction via two methods for the precondition formula and the verification condition formula construction respectively. In the script, many examples as well as the precise definitions of the preconditions and verification conditions can be found. After extending the syntax tree, start with the simple cases. Implement the construction of the precondition and verification formulas for loops last. To get unique names for shadowed identifiers, the association between identifiers and declarations mentioned earlier can be used.

Usage from the command line

After implementing parts 1 – 3a or b of the project, you can use your compiler or verifier to translate or verify TinyC programs.
A shell script with the name `tinycc` is given to execute the program via the command line. The compilation mode is specified with the option `-c`, the verification mode with `-v`. The compilation results in an output file *FileName.s*, where *FileName* references the input file name (e.g. `test` without file extension). You can also specify the output file using `-o FileName`, where *FileName* for the output file name. The verifier returns the formula for the verification condition directly on the console. Example:

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

Alternatively, you can execute the program from Visual Studio Code using the "Run & Debug" panel. You can execute the generated machine code using the MARS simulator. Use the shell script `runmars` for this, which can be executed from VS Code by a task as well (*Terminal* → *Run Task*...). Example:

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

Setting up Z3 outside the VM

To install the Z3 libraries on POSIX based operating systems, you can use the shell script `scripts/install_z3_posix.sh`. This builds and installs the required Z3 libraries to the project direc-

tory `libs` (the build process takes a while). Prerequisite for this is python3. For Ubuntu, the Z3 Java-bindings can also be installed more easily using the package manager (`sudo apt-get install libz3-java`). For Windows, you can use the batch script `scripts/install_z3_windows.bat` that downloads the required library files and copies them to the `libs` folder.

Additional remarks

- Always add new files to the package `src/tinycc/implementation` (or sub-packages). Provided interfaces must not be changed. You may add methods to the abstract classes, but you must not change already present ones.
- If you come across methods in the given interfaces, that are marked with "BONUS", but are not mentioned in the project description, you may work on those parts, but they are neither tested nor do they yield points.

Good Luck!

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

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'
```

Figure 5: The formal syntax of TinyC. Additional constructs from phase 3b are marked in green. Constructs for the bonus assignments are marked in **orange**.

Operator	Operand	Result	Remark
`*`	Pointer	Object	Pointer to complete type
`&`	Object	Pointer	Complete type, operand has to be L-value
`sizeof`	Object	`int`	Not Stringliteral and complete type
`sizeof`	*Stringliteral*	`int`	Value is the length of the string including `'\0'`
`!`	Scalar	`int`	

Table 1: Unary operators in TinyC. Additional operators from phase 3b are marked in green.

Operator	L. Operand	R. Operand	Result	Remark
`+`	Integer	Integer	`int`	
`+`	Pointer	Integer	Pointer	Pointer to complete type
`+`	Integer	Pointer	Pointer	Pointer to complete type
`-`	Integer	Integer	`int`	
`-`	Pointer	Integer	Pointer	Pointer to complete type
`-`	Pointer	Pointer	`int`	Identical pointer type, pointing to a complete type
`*`	Integer	Integer	`int`	
`/`	Integer	Integer	`int`	
`== !=`	Integer	Integer	`int`	
`== !=`	Pointer	Pointer	`int`	Identical pointer type / void* / Null pointer constant
`< > <= >=`	Integer	Integer	`int`	
`< > <= >=`	Pointer	Pointer	`int`	Identical pointer type
`=`	Scalar	Scalar	Scalar	Left side has to be assignable L-value
`&&`	Scalar	Scalar	`int`	
`||`	Scalar	Scalar	`int`	

Table 2: Binary operators in TinyC. Additional operators from phase 3b are marked in green.