Programming 2 (SS 2021)
Saarland University Faculty
MI
Compiler Design Lab

Project 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 points + 12 bonus points)

Your task in this project is to complete a compiler. The compiler translates the TINYC language into MIPS assembler and generates preconditions for the correctness of the program.

First clone the project with

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

For subtask 3b of the project, the SMT solver Z3 must also be installed. To do this, execute the installation script `scripts/install_z3_posix.sh` in the VM[1]. Instructions for other operating systems can be found at the end of the project description.

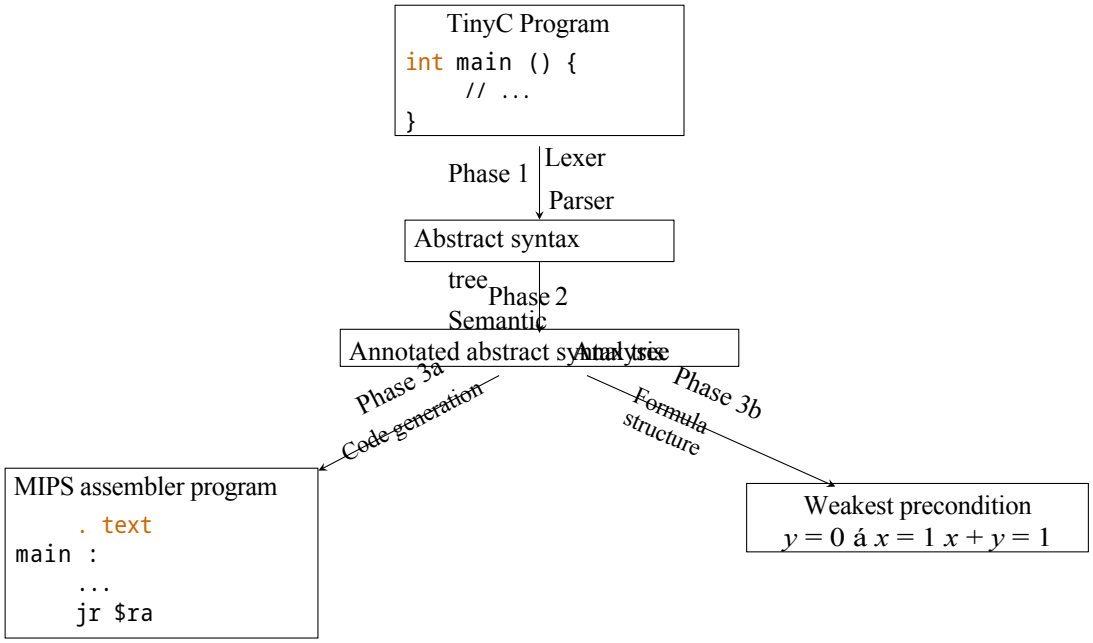
The project is divided into *three* consecutive subtasks (+ bonus tasks):

1. Construction of an abstract syntax tree **(7 points)**
2. Semantic review of the program (name and type analysis) **(9 points)**

3a. the generation of machine code for MIPS **or alternatively** **(9 points)**

3b. setting up formulas for (weakest) preconditions **(9 points)**

You can decide for yourself whether you prefer to implement code generation or the generation of weakest preconditions for TinyC programs. This project is worth 25 points. It is therefore sufficient to choose one of the two variants. However, if you implement both, you can receive bonus points. The following diagram illustrates the compiler pipeline:



[1]Depending on the hardware and VM settings, the installation can take between 2 and 20 minutes.

1 TinyC

TinyC is a restricted version of C. The most important restrictions and differences to C are:

- There are only the three basic types `char`, `int` and `void`, as well as the type constructors for pointers (`*`) and functions. Functions never appear as arguments of type constructors.
- There are no compounds and variants (`struct`, `union`).
- Not all unary/binary operators are available.
- A function can only have a maximum of four parameters.

```
int global variable ;

int foo ();

int main () {
    global variable = 1;
    return foo ();
}

int foo () {
    return global variable + 1;
}
```

Figure 1: TinyC example program

1.1 TinyC example

TinyC programs consist of several global declarations. These include global function declarations, function definitions and global variables. Each program starts with a `main` function, which always returns a value of type `int`. Figure 1 shows a valid TinyC example program.

1.2 Grammar

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

- `'x'`: The symbol `x` must appear literally in the input. (Terminal)
- `Bla`: `Bla` is the name of another rule. (non-terminal)
- `(a b)`: Brackets are used for grouping, e.g. for a subsequent `*` (grouping)
- `x?`: `x` is optional (0 or 1 time). (option)
- `x*`: `x` may occur any number of times (including 0 times). (repetition)
- `a b`: `a` followed by `b`. (sequence)
- `a | b`: Either `a` or `b`. (alternative)

The various operators are listed in *descending order of* binding strength. An example: `a | b* c` means either exactly one `a` (and no `c`) or any number of `b` followed by a `c`.

2 Overview of the compiler and the implementation

The compiler is located in the `tinycc` package and is divided into several sub-packages:

driver Contains the *driver* that controls the translation process. This parses the command line and forwards the corresponding arguments to the rest of the translator. You do not need to make any changes here.

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

diagnostic Contains functionality for error handling (error messages and warnings) which is used in the static semantic analysis of the program (see section 4)

mipsasmgen Contains functionality to generate and output MIPS assembler commands (see section 5)

logic Contains functionality to create formulas (see section 6)

implementation Your implementation should be implemented in this package (and sub-packages).

You must create your implementation in the `tinycc.implementation` package. The `Compiler` class specified in this package is the main class for your implementation. This is used to execute all phases of the compiler one after the other. A brief overview of the methods is given below:

getAstFactory Returns an instance of the `ASTFactory interface`, which is used internally by the instance of your `compiler` class to generate the abstract syntax tree (see section 3). There is therefore exactly one instance of your implementation of the `ASTFactory` per instance of your class.

parseTranslationUnit Parses the input defined by the transferred lexer and generates a syntax tree according to the interface transferred by `getASTFactory`. This method is already implemented.

checkSemantics Performs static semantic analysis, which includes name and type analysis. Annotates the syntax tree with type information etc. (see section 4).

generateCode Generates code for the current program. (see section 5)

genVerificationConditions Generates a propositional logic formula that can be used to check the correctness of the current program. (see section 6)

There are also three other packages in the `tinycc.implementation` package, each of which contains a class as the basis for your class hierarchy:

Type Your type class, which represents a type.

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

Statement Your statement class, which represents any statement in TinyC.

These classes can be extended as required. Only the name of the classes may not be changed. Use these classes as the basis for your implementation.

3 Abstract syntax tree and output (7 points)

In the first part of the project, you will build an abstract syntax tree. So that this can be tested, you should also implement the `toString` methods of the classes belonging to the AST.

We have already given you a lexer and parser. Your task is to build an abstract syntax tree from the tokens given by the parser. To do this, implement the `ASTFactory` interface (from the `tinycc.parser` package) and the `getASTFactory` method in `Compiler.java`.

3.1 AST structure

The `TokenKind` enumeration type describes the different types of tokens. Tokens are represented by the `Token` class. Tokens have information about their position in the program text (`Location`), a token type (`TokenKind` and the getter `getKind()`) and a text (`getText()`), which corresponds to the original text of the token in the program text. In addition to the lexer, the parser is given a factory class to generate the AST nodes during initialization (`ASTFactory`).

The corresponding method of your implementation is then called for the respective AST node and you can return the corresponding classes of your hierarchy. These are called by the parser during the syntactic analysis of the input program. Let's look at the statement `return y + 3;`, which is located in a file `test.c` in line `13` and starting at column `19`. The following tokens are generated first:

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

The nodes created by `ASTFactory` must be instances of the classes `Type`, `Expression` or `statement`. The following `ASTFactory` methods are called conceptually for the example:

```
Expression y     = factory . create Primary Expression ( IDENTIFIER (... , "
Expression three = factory . create Primary Expression ( NUMBER (... , " 3
" )); Expression  plus= factory . create Binary Expression ( PLUS (...) ,
y, three ); Statement            ret= factory . create Return Statement (
RETURN (...) , plus );
```

In addition to the factory methods for statements described in the syntax of TinyC, there is also a `createErrorStatement` method in the `ASTFactory`. This statement is not an element of the language, but is only created if an error occurs during parsing.

The methods `createExternalDeclaration` and `createFunctionDefinition` return `void`. Store a list of `ExternalDeclarations` in your `ASTFactory` implementation. These two methods are used to add created declarations directly to this list.

3.2 Output of the compiler

Each of your classes derived from one of the `Type`, `Expression` or `Statement` classes must override the `toString` method. This is used to test your own framework.
The exact character string representation is specified below:

- The expressions are *enclosed in parentheses at most*. All sub-expressions that consist of more than one token are surrounded by brackets. Expressions consisting of only one token are not enclosed in brackets. For example, `&x + 23 * z` is output like this:

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

- All syntactic elements, such as brackets or keywords like `while`, must be included in the output of statements:

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

- Base types are output according to their names:

```
char
int
void
```

- For pointer types, the target type is displayed first, followed by an asterisk:

```
void *
```

- For function types, the return type is output first. This is followed by an opening bracket and then the parameter types (separated by commas). Finally, the type is ended with a closing bracket:

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

- All whitespace is discarded when checking the textual representation.

4 Semantic analysis (9 points)

The following sections describe the rules for typing and the validity ranges of variables in TinyC that you should check in your semantic analysis. Your semantic analysis should be executed when you call `checkSemantics` in `Compiler.java`.

4.1 Type analysis

4.1.1 Types

TinyC contains a section of the types of C. These are arranged in a type hierarchy. There are object types and function types. The `char` and `int` types are integer types. All integer types in TinyC are *pre-characterized*. Together 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. In particular, `void*` is *not* an incomplete type, but a complete pointer type. Figure 2 illustrates the type hierarchy. Incomplete types are marked in green.

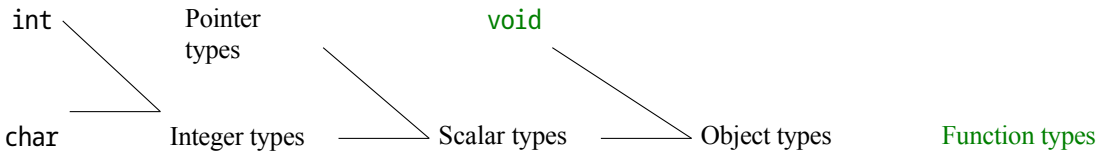


Figure 2: Type hierarchy of TinyC

Character constants and number constants always have the type `int`. The type of a string literal is `char*` [2]. The type of compound expressions is the type of the inner expression. The type of non-primary expressions is explained in more detail below.

4.1.2 Value categories

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

[2] Actually, the type of a string literal is `char[N]` (with `N` number of characters of the string including NUL characters), but for simplification we use the type `char*` and treat string constants explicitly if operand of `sizeof`.

4.1.3 Further information

The following properties must be observed during the type check:

- **Type adaptation:** Operands of type `char` are converted to `int` before the operation is executed, except as an operand of `&` and `sizeof`.
- **Conditions:** The type of conditions (`if, while ...`) must be scalar.
- **Zero** pointer**:** A zero pointer constant is a number constant with the value `0`.
- **Assignments:** For assignments, one of the following must apply:
  - The types of both operands are identical.
  - Both operands have integer type.
  - Both operands have pointer type and at least one of them has type `void*`.
  - The left operand has pointer type and the right operand is a zero pointer constant.

  In any case, the left operand must be L-Value. The type of the assignment is the type of the left operand.
- **Function calls:** The parameter transfer for function calls is subject to the same rules as assignments. Here, the parameter of the function is the "left-hand side" and the value passed is the "right-hand side".
- **Return values:** The type of `return` ("right side") is treated like an assignment with the return type of the surrounding function ("left side"). There must be an expression if the return type is not `void`. It is not necessary to check whether a path to the end of a `non-void` function does not contain a `return statement`. In such a case, the return value is undefined.
- **Declarations:** No variables with the type `void` may be declared.
- **Function parameters:** No parameters with the type `void` may be agreed.
- **Error:** The `ErrorStatement` is always well-typed.

Tables 1 and 2 show all the operators you need to implement and describe their signatures.

4.2 Scopes of validity

Each block opens a new scope. The outermost scope is referred to below as *the global scope* and contains *global variables* as well as *function declarations* and *definitions*. All inner scopes are *local scopes*.

Functions may be declared as often as required in the global scope of validity, but may only be defined once. Global variables can be declared as often as required. However, the type of the variable or the signature of the function must be identical for each new declaration [3] . Unlike in C, the semantics of `int f();` is that `f` is a function that takes no arguments.

In local scopes, however, variables may only be declared once. Function definitions open a new local scope, which also includes the parameters. Blocks also open a new local scope of validity. Declaring or defining functions is not possible here according to syntactic rules. Furthermore, function parameters and declarations in local scopes of validity can cover a variable in an external scope of validity.

Functions and variables may only be used textually after their declaration/definition. If one of the above rules is violated, an error should be reported. Figure 3 shows an example.

[3] Parameter names are not part of the signature of a function.

```
int x; // Declaration of x in the global validity range
int x; // OK , new declaration in the global validity range
// char x; Error : Renewed declaration with different type

int foo ( int x, int y); // Function declaration
// void foo ( int x, int y); // Error : Renewed function declaration with undersigned . signature

// Function definition of foo
int foo ( int x, int y) { // Agreement 2 of x, hides global variable
    // int y;    Error : y already declared in this validity range ( parameter )
    {
        int x = 1;          // Agreement 3 of x
        x = x + 1;          // x refers to agreement 3
    }
    return x;               // x refers to agreement 2
}
```

Figure 3: Example of scopes in a TinyC program.

4.3 Diagnostic

If a problem is detected in the program (e.g. an unknown variable), a message is sent to the user or programmer. This is realized by an instance of the `Diagnostic` class, which is passed to the constructor of the `compiler class`. Each notification corresponds to an error generated by your analysis. The text of the message itself is not tested by us. However, each message expects an exact position in the source code of the input program. If an error occurs during the static semantics check, an error must be output (using `Diagnostic.printError`) and the exact location of the error in the source program. You must output the first error in a program first. You are free to output further errors afterwards. After the error has been output, your program may behave as you wish, as it will be aborted immediately by the tests. In the case of an undefined variable, this looks like this, for example:

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

Here `v` is unknown and an error must be generated with the exact position of `v` (here `test.c:2:17`). If an argument of an operator is invalid, the position of the operator must be output:

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

As it is invalid to multiply a pointer by `42`, an error message must be output at the position of the multiplication operator (in this case `test.c:2:16`). In the case of an instruction, the position of the instruction is decisive:

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

In this example, `return` does not expect `an expression`, as the return type of the function is `void`, and therefore the position of the `return token` must be output.

5 Code generation (9 points)

Code generation is the last phase of the compiler and outputs executable MIPS machine code for the input program. It should be executed when `generateCode` is called in the `Compiler` class.

5.1 Restrictions of TinyC

Functions in TinyC can have a maximum of four parameters. This means that only a maximum of four arguments can be passed when a function is called. This means that all arguments to a function fit into registers `$a0` to `$a3` of the MIPS processor. You can assume that programs will never be compiled if there are more arguments for functions. Please note, however, that you must adhere to the calling conventions.

To ensure that code can also be easily generated for expressions, it is ensured that the number of temporarily required registers never exceeds 10. This means that all temporary results can be stored in the registers `$t0` to `$t9` can be stored.

No code can be generated for the `ErrorStatement`, and if this is attempted, a `IllegalStateException` can be thrown.

5.2 Implementation

The `mipsasmgen` package provides auxiliary functions and auxiliary classes for code generation. The `MipsAsmGen` class is the main class for code generation. It has the ability to generate individual commands for the text segment as well as data and other declarations in the data segment. The corresponding segment is selected automatically (ensured by overloaded methods). You can also conveniently create labels (for text and data segments) which are guaranteed to be unique (`makeTextLabel` etc.). The generated labels can be explicitly placed in the code using `emitLabel`. However, if you generate data for the data segment, the labels are immediately set automatically.

Individual command classes are represented by *enums*. The methods of the assembler generator for the output are overloaded accordingly. The individual enumeration entries are named like the corresponding MIPS commands (apart from capitalization).

Let's look at a small example for a better understanding:

```
int global_var ;

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

In the example shown, the small TinyC program consists of a global variable and a `main` function that loads and returns the value of the variable. Let us now compile this small program using the assembler generator class:

```
Mips Asm Gen     gen= new Mips Asm Gen ( System . out );

Data Label  data = gen . make Data Label (" global_var
"); gen . emit Word ( data , 0);

Text Label  main = gen . make Text Label (" main
"); gen . emit Label ( main );

gen . emit Instruction ( Memory Instruction . LW , GPRegister . V0 , data , 0 , GPRegister .
ZERO ); gen . emit Instruction ( Jump Register Instruction . JR , GPRegister . RA );
```

First, a new instance is created using the constructor and a `PrintStream` (which is already passed to the corresponding method in your implementation - see *JavaDoc*). Then the global variable `global_var` is represented by a new `DataLabel`. To create this variable with the generated `DataLabel`, we use a call to `emitWord`, as the variable is of type `int`. This is followed by the definition of our method `main`. With the help of a new text label, we have a label available for the method. We output the label and generate code for the body of the method. The variable `is` loaded using a `MemoryInstruction`, where the target register is `$v0`. A return to register `$RA` is then generated to end the function. The generated code is shown below:

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

5.3 Rules for code generation

- **Type sizes:** With MIPS, the type `int` is mapped to `word` and `char` to `byte`.
- **Type conversion:** Operands of type `char` are converted to `int` before the operation is executed, except as an operand of `&` and `sizeof`.
- **Assignments:** If the left operand of the assignment is of type `char`, the new value is truncated before the assignment to the target value. The result of an assignment is the new value of the object on its left-hand side. The same applies to **return values** and **function calls** (see 4.1).
- **Comparisons:** The comparison operators return the values `0` (false) and `1` (true).
- **Conditions:** In conditions, values not equal to `0` or a null pointer are evaluated as true.
- **sizeof:** The `sizeof operator` returns the size of the operand in bytes. For most operands, this is the size of the type of the operand in bytes. For a character string literal, it is the *length* of the character string including the terminating NUL character.
- **Function call:** You do not have to consider the case that function calls have other function calls in their arguments.

6 Verification by (weakest) preconditions (9 points)

The alternative to code generation is to implement a verification procedure for annotated TinyC programs, which is based on weakest preconditions (chapters 6.4 - 6.6 of the script).

6.1 Expansion of grammar

In order to communicate the conditions to be checked to the compiler, the grammar is **extended** as described in Figure 5. An `_Assert` statement contains a condition that is to be proven. The condition within an `_Assume` statement may be assumed to be true by you and used for the proof. To prove programs with loops partially correct, a loop `invariant` (`_Invariant`) must be annotated to loops. To prove programs with loops totally correct, a termination condition (`_Term`) must be specified in addition to the loop invariant.

If the program contains nested loops, the invariant of the inner loop cannot syntactically refer to the upper bound k of the termination function of the surrounding loop without further ado. At this point, the inner loop necessarily "forgets" any statements that refer to the upper bound k, which generally makes it impossible to prove the termination of the outer loop.

To circumvent this problem, we take the following approach: A termination condition has an optional identifier that defines the upper bound *k* for the termination function that occurs in the verification condition of the loop. Scheduling conditions open a new scope between the invariant and the loop body in which this identifier (if it exists) is agreed. If it exists, the identifier has the type `int` and may only be used within invariants (as long as it is not hidden by a declaration of the program) in order to enable the termination proof.

Expressions are **extended** by the operators *and* (&&), *or* (||) and *not* (!) as described in Figures 1 and 2. They correspond to the logical operators á, v and ¬ and operate on `integers` [4]. The operators behave like normal binary or unary operators in the output. A read-in
`_Assert(x > 0);` is output as `_Assert((x > 0));`, the other new statements work in the same way.

First extend your AST structure so that it accepts this extended grammar. Then adapt your semantic analysis so that it also checks the new statements.

6.2 Formula structure

Your task is to implement the `genVerificationConditions` function in `Compiler.java`. This function should return a `formula` that has been constructed according to the rules in chapter 6.6. In addition, `return statements` are used in this project. The weakest precondition for a `return` statement is always fulfilled.

We have already given you the formula classes; you can find these classes in the `tinycc.logic` package. To demonstrate the use of the predefined classes, here is a small example: Figure 4 shows a TinyC program in which you can assume that y has the value 5 and which checks whether $x == y$ after setting x to 5.

Example input:

```
int f( int x, int y) {
  // Calculated 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
}
```

Example structure of the calculated formula:

```
Variable left Y = new Variable (" y", Type . INT );
Int Const left5 = new Int Const (5);
Binary Op Formula left = new Binary Op Formula ( Binary Operator . EQ , leftY , left5 );


Int Const right L 5 = new Int Const (5);
Variable right LY = new Variable (" y", Type . INT );
Binary Op Formula right L = new Binary Op Formula ( Binary Operator . EQ , rightL5 , right LY );

Bool Const right R = Boolconst . TRUE ;

Binary Op Formula right = new Binary Op Formula ( Binary Operator . AND , rightL , right R );

Formula imp = new Binary Op Formula ( Binary Operator . IMPLIES , left , right );
```

Figure 4: TinyC program, the formula to be calculated and use of the formula packages.

The formula that you should derive for this program is, for example, $((y == 5) \Rightarrow (5 == y))$. To create this formula with the given formula classes, you can first create a constant and a variable, and then create a formula that expresses the equality. Do the same for the other side of the implicit formula.

[4] In C, both operands only have to be scalar, but pointers are omitted in this subtask.

cation. Then create the conjunction with the Boolean constant for "true" for the right-hand side. Once you have built both subformulas, the implication can be created from them.

6.2.1 Restrictions on TinyC for testing preconditions

The program has exactly one function definition and any number of function or variable declarations. The defined function marks the entry point for the verifier. You therefore calculate vc(`B` | true) á pc(`B` | true), where `B` is the body of the function. In this function . .

- . . . no function calls may be made.
- . . . all loops have invariants.
- . . . only integer types may be used (no side effects).
- . . . assignments may only occur in `ExpressionStatements` and must be the outermost expression. The left-hand side of the assignment must also be an identifier. This means that `x = 8 + z;` is permitted, but `y = (z = 5);` is not.
- . . . expressions must not contain division (`/`).

Ensure these restrictions *when building the formula* and throw any exception if they are violated. You can find the above example in the public tests.

6.2.2 Translate expressions into formulas

The TinyC language does not recognize boolean types. In the formulas that are to be constructed, however, there are both `int` and `bool` types. The operators are therefore interpreted as follows in the formula structure:

+ -	×The operands are of type `int` and the total expression is of type `int`.
> ≥ < ≤	The operands are of type `int` and the overall expression is of type `bool`.
== ≠	The operands are both of type `int` or both of type `bool`. They return a `bool`.
áv Ù ¬	Your operand(s) are of type `bool` and the overall expression is of type `bool`.

In order to do justice to the types, it is necessary to convert from `int` to `bool` in some places. For example, you should build the expression `1 && 2` into a formula that explicitly converts the two integers from `int` to `bool` by inserting a ≠ 0 comparison in the required places. The result is the formula (1 ≠ 0) á (2 ≠ 0).

An implicit conversion from `bool` to `int` is not required. We will therefore not test expressions such as `(1 && 2) < 3` because the operator á returns a boolean formula and the operator < expects a formula of type `int`.

6.2.3 SMT Solver

SMT solvers can determine whether a formula has a variable assignment so that the formula evaluates to true. The result is *ratirfiable* if such an assignment exists and *unratirfiable* if there is no such assignment. The solver does not directly offer a way to say that a formula evaluates to true for all variable assignments. However, as we want to prove this property for our programs, we use a popular trick: we negate the formula and expect it to be unratirfiable. In tests, you therefore get the result *unratirfiable* if the formula you have calculated is universally valid. If you get the result *ratirfiable* together with a counterexample (`model`), the formula is not universally valid.

In TinyC there is variable overlap, i.e. there can be several variables with the same name. However, the solver assumes that variables with the same name are identical. You must therefore ensure in your implementation that you disambiguate different variables with the same name. This can be achieved, for example, by adding a unique number to the name of the variable when you build the formula.

In this project, the freely available theorem prover Z3 is used to recognize formulas as universally valid.

Notes:

- The classes and interfaces in the `tinycc.logic.solver` package are auxiliary classes that translate TinyC formulas into Z3 formulas. You do not need to use or look at these classes for your implementation.
- The script contains the function `def e`, which generates a formula that describes the set of all states on which `e` is defined. You can assume that expressions are always defined based on previous name analysis and therefore disregard this function or replace it with *true*.
- Whether you should show partial or total correctness is defined by whether all loops have a termination condition in addition to the invariant. If this is the case, you determine total correctness, otherwise partial correctness. To show partial correctness, you can 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 formula we have specified. Only the fulfillability must be the same.

7 Bonus task: Break and continue

In this bonus task, TinyC is **extended** by `break` and `continue` (see Figure 5). The statements can appear anywhere in the source code. However, they are only permitted within loops.

7.1 Semantic analysis (1 bonus point)

Extend your semantic analysis so that you generate an error for programs in which such statements appear at non-permitted positions. For example, the following program should be rejected and the error message should point to the position of the `break statement`:

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

You can already receive bonus points for this subtask if you pass the *public* tests on AST structure and semantic analysis.

7.2 Code generation (2 bonus points)

While `break` terminates a loop immediately and switches to the statement directly after the surrounding loop, `continue` ensures that the loop condition is returned to immediately. Implement the code generation for `break` and `continue`. To do this, it makes sense to create jump labels for each loop, which refer to the correct position for `break` and `continue`.

7.3 Verification (2 bonus points)

In order to be able to verify programs with `continue`, the verification and preconditions for this construct must first be formally defined. If a `continue` is sighted, the current postcondition is discarded and reset to the formula i á $0 \leq t \leq k$, where i is the invariant of the surrounding loop, t is the termination term and k is the upper bound for the termination function. We therefore "forget" the post- condition and pretend that we are starting from the end of the loop.

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

To also allow `break`, the loop condition after the loop must no longer be assumed to be false if the loop contains a non-nested `break`, i.e. not related to a nested loop.

is included. In addition, the loop terminates immediately when `break` is executed, which is why the part of the termination proof can be ignored. Otherwise, `break` behaves in the same way as `continue`:

$$\mathrm{vc}(\texttt{break};\,|N) = \mathrm{true} \qquad \mathrm{pc}(\texttt{break};\,|N) = i$$

$$\mathrm{pc}(\texttt{while (e) s};\,|N) = \begin{cases} \mathrm{fvc}(\texttt{s}\;|\,i \mathbin{\acute{a}} 0 \le t \le k) \\ \quad \mathbin{\acute{a}} (i \mathbin{\grave{U}} N) \\ \quad \mathbin{\acute{a}} (e \mathbin{\acute{a}} i \mathbin{\acute{a}} 0 \le t \le k + 1 \\ \quad \mathbin{\grave{U}} \mathrm{pc}(\texttt{s}\;|\,i \mathbin{\acute{a}} 0 \le t \le k)) & \text{if s contains a non-nested break.} \\ \mathrm{vc}(\texttt{s}\;|\,i \mathbin{\acute{a}} 0 \le t \le k) \\ \quad \mathbin{\acute{a}} ((i \mathbin{\acute{a}} \neg e) \mathbin{\grave{U}} N) \\ \quad \mathbin{\acute{a}} (e \mathbin{\acute{a}} i \mathbin{\acute{a}} 0 \le t \le k + 1 \\ \quad \mathbin{\grave{U}} \mathrm{pc}(\texttt{s}\;|\,i \mathbin{\acute{a}} 0 \le t \le k)) & \text{otherwise} \end{cases}$$

8 Signposts and tips

This project is the last and also the most demanding project of the course. We will give you some tips to help you manage the project (in terms of time):

General Write your own tests with which you can regularly check the individual phases of your implementation for correctness. In the public tests you will find examples to inspire you. The semantics of C or TinyC can sometimes be surprising or unintuitive. So ask questions if these arise. The `CompilerTests` and `FatalDiagnostic` classes also have debugging fields that you can `set` to `true` if required, for example to display your compiled assembly code or your formula for the verification condition.

AST Generating the abstract syntax tree is the simplest part of the project. However, the class structure that you create for the abstract syntax tree can make further processing of the project easier or more difficult. First think about how to represent the syntactic elements in your class hierarchy in a meaningful way. The syntactic categories of formal syntax provide orientation for this.

You have worked on this phase of the project in the last 3 days.

Semantic analysis The semantic analysis is somewhat more difficult, as there are a lot of semantic errors to check. Think about how the type hierarchy of TinyC can be meaningfully represented. Then you should consider how to represent one or more nested scopes as a data structure in a meaningful way. Also note that the type of a variable, if properly defined, depends on the current scope. Once you have laid out the basic data structures, you can start with the semantic analysis. Gradually implement this for all expressions from tables 1 and 2, and then for all statements.

Also remember to annotate the AST with semantic information during this phase. For example, the type of an expression is used in numerous places and should not be recalculated, but stored temporarily. It is also advantageous to save which identifiers in expressions refer to which declaration during the name analysis. You can use the declarations to disambiguate the identifiers later.

You will have completed this phase of the project after 9 days.

Code generation Code generation is the most difficult part of the implementation. First of all, make it clear which elements of TinyC are to be assigned to which MIPS elements (e.g. constants, string literals, global variables etc.).

Do not store local variables in registers, but on the stack. To do this, assign a unique offset on the stack to each declaration in a function so that you know from which address you have to load the associated variable in the using context or save it in the assigning context.

Make sure you observe the calling convention when calling functions. You must therefore save the caller-save registers on the stack before a function call and restore them after returning. Conversely, functions must observe the callee-save registers. However, registers that are not used by their implementation do not need to be saved, which can save you work. The temporary registers must be used for expressions. When generating code for expressions, note which temporary registers are already occupied so that you do not inadvertently overwrite other results in the same expression.

Weakest preconditions The alternative verifier for code generation is also demanding. Please read chapters 6.4 - 6.6 again and ask questions if anything is unclear. The proofs do not have to be reproduced for the implementation.

The script contains many examples and precise definitions that describe how the formula must be structured. After you have expanded your syntax tree, start with the simple cases. Only at the end should you implement the formula structure for loops. In order to obtain a unique name for hidden identifiers, it is advisable to use the above-mentioned association of identifiers with declarations.

Use on the command line

After you have implemented part 1 - 3a or 3b of the project, you can use your compiler or verifier to translate or verify TinyC programs.

To start the program from the command line, a shell script with the name `tinycc` is included. The compiler mode is specified using the `-c` option, the verification mode using `-v`. The translation generates an output file with the name *FileName.r*, where *FileName* stands for the input file name (e.g. `test` without file extension). You can also specify an output file using `-o FileName`, where *FileName* stands for the output file name. The verifier outputs the formula for the weakest precondition directly to the console. Example:

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

Alternatively, you can also start the program in Visual Studio Code from the Run & Debug panel. You can execute the generated machine code yourself using the MARS simulator. The shell script `runmars` is included for this purpose, which can also be called in code via a task (*Terminal → Run Tark. . .*). Example:

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

Set up Z3 outside the VM

To set up the Z3 libraries on POSIX-based operating systems, you can use the shell script `scripts/install_z3_posix.sh`. This builds and installs the necessary Z3 libraries into the project directory `libs`. The prerequisite for this is python3. On Ubuntu, the Java bindings for Z3 can also be installed more easily using the package manager (`sudo apt-get install libz3-java`). For Windows there is the batch script `scripts/install_z3_windows.bat` which downloads the necessary library files and copies them into the `libs` directory.

Further information

- Always place new files in the `src/tinycc/implementation` package (or sub-packages). Interfaces specified by us may not be changed. You may add methods to the abstract classes, but you may not change anything specified in them.
- If you find methods in given interfaces that are marked with "BONUS" but do not appear in the task, you are welcome to work on these parts, but they will not be tested and no points will be awarded for them.

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 | 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'
```

Figure 5: The formal syntax of TinyC. Additional constructs from phase 3b are shown in **green**. Constructs from bonus tasks are shown in **orange**.

Operator	Operand	Result	Note
`*`	Pointer	Object	Pointer to complete types
`&`	Object	Pointer	Complete type, operand must be L-Value
`sizeof`	Object	`int`	Non-string literal & complete type
`sizeof`	*Stringliteral*	`int`	Value is the length of the character string including `'\0'`
`!`	Integer	`int`	

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

Operator	L. Operand	R. Operand	Result	Note
`+`	Integer	Integer	`int`	
`+`	Pointer	Integer	Pointe r	Pointer to complete types
`+`	Integer	Pointer	Pointe r	Pointer to complete types
`-`	Integer	Integer	`int`	
`-`	Pointer	Integer	Pointer	Pointer to complete types
`-`	Pointer	Pointer	`int`	Identical pointer type on complete types
`*`	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 must be assignable L-value
`&&`	Integer	Integer	`int`	
`||`	Integer	Integer	`int`	

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