UNIVERSITÄT
DES
SAARLANDES

Programming 2 - SS 2023

Project 6 - Compiler

Haoyi Zeng, Nicolas Edelmann

July 5th 2023

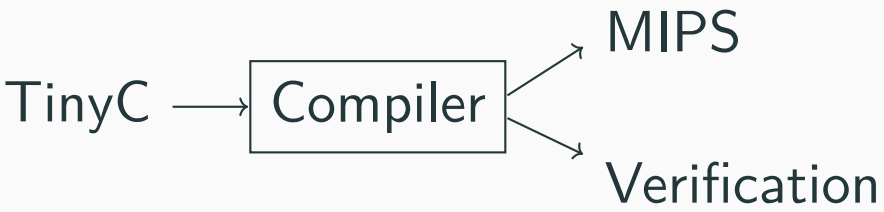
Saarland University

Overview

1. Compiler Structure Overview
2. Repository Overview
3. Abstract Syntax Tree
4. Semantic Analysis
5. Code Generation
6. Verification
7. Tests and Debugging

Compiler Structure Overview

Project Overview



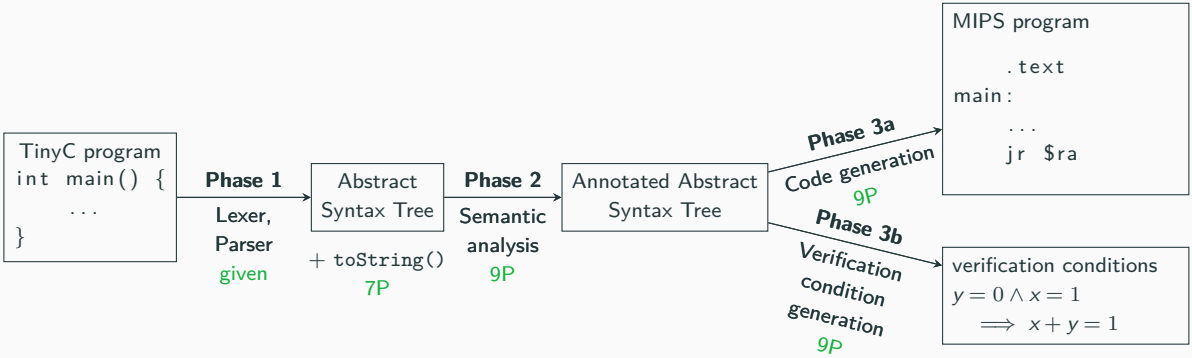
TinyC

TinyC is like C, but (for example)

- `float pi = 3.1415;`
- `struct Date { /* content */ };`
- `int mask = 0xff << 8;`
- `void foo(int a, int b, int c, int d, int e);`
  (more than four arguments)

are **not** supported.

Compiler Structure



Project Points

Regular:

	7	ASTFactory implementation + `toString()` methods
+	9	semantic analysis (types & scopes)
+	9	syntax-guided code generation **or** verification
$\sum$	25	

Bonus:

	9	syntax-guided code generation **or** verification (the other one)
+	5	`break`, `continue`
+	5	`performOptimizations()`
$\sum$	19	

5 bonus points - tested automatically

`break` and `continue`

- (1 points) semantic analysis
- (2 points) code generation
- (2 points) verification

5 bonus points - oral evaluation

- constant folding
- and many other optimizations ...
- If you want these points, send an E-Mail to Marcel to get an oral evaluation

Any questions (so far)?

Repository Overview

Git Repository

We can clone the project using `git clone` and the following URL:

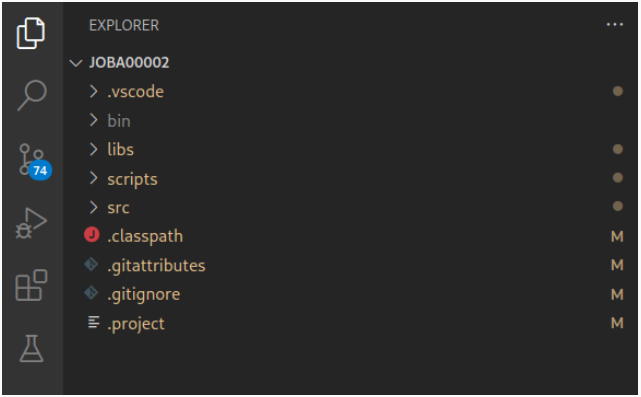
```
ssh://git.prog2.de:2222/project6/$NAME.git
```

$NAME: Your CMS username

Opening in VS Code

Make sure to open the correct folder!

If done correctly, it should look like this:



Package Structure

The compiler is implemented in the package `tinycc` and split into several subpackages:

driver	driver to control the compilation process (parses command line arguments etc.)
parser	lexer and parser for `TinyC` (already implemented)
diagnostic	functionality for handling and printing errors and warnings generated by the compiler
mipsasmgen	functionality for generating and printing MIPS assembly
logic	functionality for building logical formulas and invoking an SMT solver
implementation	**your implementation goes here**
tests	**your tests go here**

All new classes have to be in `tinycc.implementation` or subpackages thereof.

Common mistake:
Not doing the project at all!

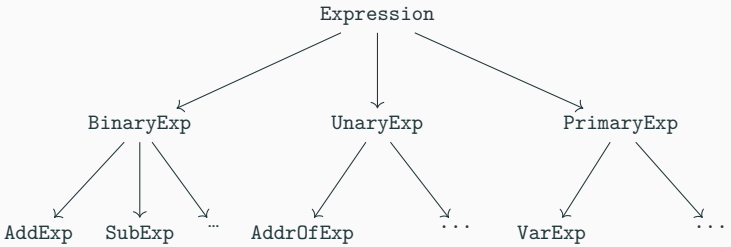
Abstract Syntax Tree

`ASTFactory`

- methods for all statements, expressions and types
- `createExternalDeclaration` and `createFunctionDefinition` return `void`, you have to store them!
- **Hint:** List of `ExternalDeclarations` in `ASTFactory` implementation
- **Hint:** You may want more classes than there are methods in the ASTFactory!
  - e.g. one class per operator

Class Structure (AST)

`Type` `Expression` `Statement`



For more inspiration, take another look at the expressions project from the exercise sheet!

You will need **a lot** of classes (the tutor implementation has 81)

`toString()`

- Format using abstract syntax!
- `int*` $\rightsquigarrow$ `Pointer[Type_int]`
- `x + 3` $\rightsquigarrow$ `Binary_+[Var_x, Const_3]`
- `{int x; x = 5;}` $\rightsquigarrow$
  `Block[Declaration_x[Type_int], Binary_=[Var_x, Const_5]]`
- Some parts may be `null` (e.g. `if` without `else`), in which case they should not be printed!
- Whitespace is ignored

Any questions (so far)?

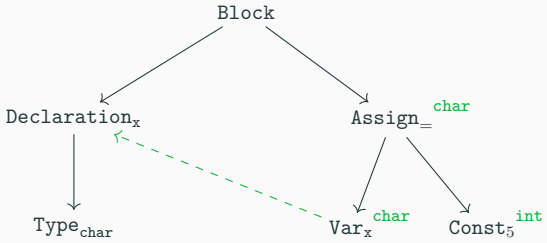
Semantic Analysis

Semantic Analysis

The semantic analysis serves two purposes:

- Reject invalid programs before the next step
  - ill-typed expressions
  - variables of type void
  - etc.
- Annotate the AST with information needed for further steps
  - types of expressions
  - etc.

What is an annotated AST?



Semantic Analysis - Diagnostic

When a program is rejected, errors must be printed using the `Diagnostic` class:

- `printError` prints error messages. The actual message is ignored, but the location must match exactly!
- `printNote` can optionally be used to print additional information. You will not need this.

Your Program will be stopped after the first error message, so it is not important whether you print more.

Semantic Analysis - Types

- incomplete types
  (only `void` und Function types)
- complete types
  (everything else, including `void*`)

Annotated AST in Code

- Add more fields to your classes!
- For example:
  - Save the type of an expression, since it is needed afterwards
  - Whether `Binary_+` does `Int+Ptr`, `Ptr+Int`, `Int+Int` (similarly for `Binary_-`)
  - The type of arguments during a function call
  - The `While` loop that `Break` and `Continue` belong to (and inversely)
  - A `Block` might want to know its contained `Declarations`
- Factor common functionality into proper classes
  - `Type::isAssignableFrom(Expression)`
  - `Type::isComplete`, `Type::isObject`, `Type::isScalar`

Annotated AST in Code – Declarations

- Feel free to add more classes / interfaces
- Very useful: `public interface Declaration`
  - Should give you name and type
  - Uses of variables should remember their declaration
  - Redeclarations require special care
    - Hint: store first declaration and delegate to it
- Already mentioned: `ExternalDeclaration`
  - For storing in the ASTFactory
  - Should likely `implements Declaration`

Common Mistakes

- Not writing the type checker at all
- Not reading the script, in particular Chapter 9.2
- „Being a null pointer“ is a property of expressions, not types
  - `char* c = 0;` is valid, since 0 is a null pointer.
  - `char* c = 1;` is not.
- Assignment is an expression
  - `a = b = c` is valid
  - Expression statements in general: `if(1) 3;` is also valid.
- Character constants `'a'` have type `int` (and size 4)
- Implicit conversions: Operations on chars almost always type to `int`
- Well-known C compilers do not actually implement C
  - Compile with `-std=c11 -pedantic` (which only makes it better)

Any questions (so far)?

Code Generation

Code Generation

Given an annotated AST, generate MIPS code.

- Hardest part of the project
- Implement algorithm from the lecture notes
- Uses results from Semantic Analysis
  - becomes easier when Semantic Analysis is well-designed

MIPSAsmGen

- Helper class to generate assembly easily and correctly
- Two kinds of labels
  - `makeUniqueTextLabel` for internal labels (`If` and `While`)
  - `makeTextLabel` for global labels (functions and global variables)
  - `makeTextLabel` can only be created once
    - Store label in declaration
    - Be careful with re-declarations
- Note that you can arbitrarily often switch between text and data segment:

```
.text
  addiu $t0 $zero 42
.data
str1: .asciiz "Hello, World!"
.text
  la $t1 str1
```

Common mistakes

- Functions must follow the calling convention
  - Your code will call and be called by hand-written assembly functions
- Working with pointers
  - `int* p = ..; p = p + 1` increases `p` by 4 bytes, as `sizeof(*p)` is 4
  - `(p + 1) - p` must evaluate to 1, even as pointers are 4 bytes apart
- `sizeof` does not evaluate its argument, but only looks at the type
  - Special case: `sizeof("Hello, World!")` evaluates to 14
- Evaluation order is often nondeterministic
  - Evaluate expressions that need more registers first

Common mistakes 2 – Electric Boogaloo

- Handling *conversion as if by assignment*
  - `a = b = c;` equivalent to `b = c; a = b;`, as `c` is converted
  - During function calls
  - During function returns
  - During stores
- Booleans do not exist
  - Comparisons evaluate to 0 or 1
  - A value is „true“ if it is not 0
  - Pointers have truthiness, the null pointer is „false“

Functions you might want

- Create a `class CompilationScope`
- `Expression::codeL`, `Expression::codeR`, `Expression::conditionalJump(TextLabel, TextLabel, ..)`
- `Type::getSize`, `Type::getStoreInsn`, `Type::getLoadInsn`
- `Type::generateMove(GPRegister, GPRegister)`
- `Declaration::codeL`
- And much more...

Any questions (so far)?

Verification

Verification

- Use vc/pc to generate a formula describing when a program is correct.
- Works on a heavily restricted version of TinyC
  - No function calls
  - No nested assignments
  - No division
  - No pointers

  $\Rightarrow$ Less corner cases than Code Generation

  $\Rightarrow$ Debugging is much harder, though
- Building logical formulas made easy, see `tinycc.logic`
  - In particular, `Formula::subst` implements substitution
- When you are done, you *will* understand pc/vc

SMT Solvers

- Can you find values for x, y such that
  - $x > y$ Yes!
  - $x < y \wedge y < x \vee 3 = 4$ No!
  - $x = 3 \wedge x = 4 \rightarrow y = 2$ Yes!
- A SMT solver can solve such problems automatically
- Idea: Generate verification formula φ.
  If $\neg\varphi$ has no solution, then φ holds in general
- We use the Z3 SMT solver to check whether vc/pc formula holds in general
- If the formula does not, you will be given a counterexample
  - This will make debugging possible

A simple example

```
int min(int a, int b) {
  int res;
  if (a < b) res = a;
  else res = b;
  _Assert (res <= a && res <= b);
  return res;
}
```

- Formula φ: $(a < b \rightarrow a \leq a \wedge a \leq b) \wedge (a \geq b \rightarrow b \leq a \wedge b \leq b)$
- Negation $\neg\varphi$: $\neg((a < b \rightarrow a \leq a \wedge a \leq b) \wedge (a \geq b \rightarrow b \leq a \wedge b \leq b))$
  - Can you find an assignment that makes $\neg\varphi$ true? No!
  - Thus, the program is correct

A simple bugged example

```
int min(int a, int b) {
  int res;
  if (a < b) res = a;
  else res = b;
  _Assert (res <= a && res <= b);
  return res;
}
```

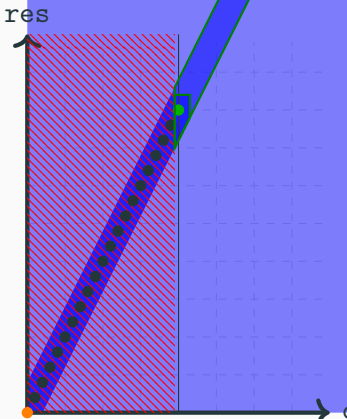
- Let's say our compiler confuses consequence and alternative
- Formula φ: $(a < b \rightarrow b \leq a \wedge b \leq b) \wedge (a \geq b \rightarrow a \leq a \wedge a \leq b)$
- Negation $\neg\varphi$: $\neg((a < b \rightarrow b \leq a \wedge b \leq b) \wedge (a \geq b \rightarrow a \leq a \wedge a \leq b))$
  - Can you find an assignment that makes $\neg\varphi$ true? Yes!
  - For example $a = 0, b = 1$
- Problem: Spotting the bug is hard

Loop invariants

- Verifying loops is hard
  - In general, verifier would have to solve the halting problem
- Idea: Let's help the verifier
- Programmer must find and specify loop invariant that validates the program

How to find the loop invariant

Consider the state space of the loop: Find mathematical formula describing state space: Ensure it fits the postcondition:



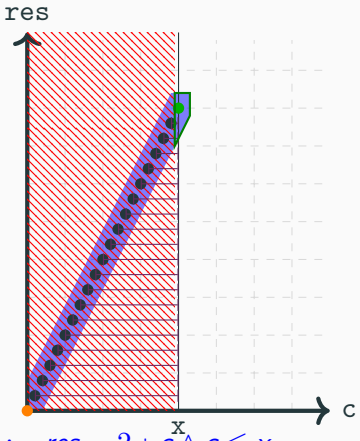
$$[\text{While}]\ \frac{\vdash \{I \wedge e\}\ s\ \{I\}}{\vdash \{I\}\ \texttt{while}\,(e)\ s\ \{I \wedge \neg e\}}$$

```
int mul2(int x) {
  _Assume (x >= 0);
  int res = 0;
  int c = 0;
  while (c < x)
  _Invariant (I) {
    c = c + 1;
    res = res + 2;
  }
  _Assert (res == 2 * x);
  return res;
}
```

$I := ?$ $I := \texttt{true}$ $I := res = 2 * c$ $I := res = 2 * c \wedge c \leq x$

Loop termination

Does the loop always terminate? Yes! Why?



$I := res = 2 * c \wedge c \leq x$

```
int mul2(int x) {
  _Assume (x >= 0);
  int res = 0;
  int c = 0;
  while (c < x)
  _Invariant (I)
  _Term (t) {
    c = c + 1;
    res = res + 2;
  }
  _Assert (res == 2 * x);
  return res;
}
```

$t :=$ something ≥ 0 that always gets smaller $t :=$ how many loop

Loop termination and k

- $I := res = 2 * c \wedge c \leq x$

  $t := x - c$

- Remember total correctness rule for `while`:

$$\text{[While]}\ \frac{\vdash [\overbrace{I \wedge e \wedge 0 \leq t \leq k+1}^{I'}]\ s\ [I \wedge 0 \leq t \leq k]}{\vdash [I]\ \texttt{while}\ (e)\ s\ [I \wedge \neg e]}$$

- Need to preserve $x - c \leq k + 1$ during the inner loop:

  $I' := I \wedge e \wedge 0 \leq t \leq k + 1$

  $I' := res = 2 * c \wedge c < x \wedge x - c \leq k + 1$

- Problem: k is ghost state, how to use in code?

```
int mul2(int x) {
  _Assume (x >= 0);
  int res = 0;
  int c = 0;
  while (c < x)
  _Invariant (I)
  _Term (t) {
    while (0)
    _Invariant(I')
    _Term (0) {}
    c = c + 1;
    res = res + 2;
  }
  _Assert (res == 2 * x);
  return res;
}
```

```
int mul2(int x) {
  _Assume (x >= 0);
  int res = 0;
  int c = 0;
  while (c < x)
```

Common mistakes

- Thinking that MIPS CodeGen is easier
  - Verification is about 30% of the code required for CodeGen
- Implementing *def*
  - Not needed, since there is no undefined behavior
  - We ignore overflow
- Forgetting about pointers during type check
  - Your type checker needs to work for full TinyC!
- Name variables apart!
  - x can occur several times, referring to different variables (shadowing)
- `bool` exists in logical formulas (only)
  - Sometimes, integer conditions must be converted: $e \mapsto e \neq 0$

Useful hints

- Notice that *pc* and *vc* are always called with the same argument
  - A clever implementation could merge them
  - And create a class `VerificationResult` storing both
- Give every declaration a unique id (to handle shadowing)
- Construct logical formulas step by step to keep things readable
- You might want methods like
  - `Statement::toPC(bool, ..)`, `Statement::toVC(bool, ..)`
    - bool indicates total correctness mode
    - Or do `Pair Statement::toFormula(bool, ..)`
  - `Declaration::getFormulaVar`
  - `Declaration::isGhostVariable`, `Scope::isInGhostExpression` for handling k
  - `Expression::toTerm`
  - `Expression::toFormulaAsStatement` (mostly for assignments)

Any questions (so far)?

Additional Hints

- Lecture notes and project description are very helpful
- Strictly adhere to grammar and typing rules
- Unless noted otherwise, the C11 rules apply

Tests and Debugging

Tests and Debugging

- Write your own tests!
  - Will be tested against the reference
  - Highly recommended to resolve ambiguities
  - Hint: Test examples can be found in these slides, the script, exercise solutions, ...
- Use minimal examples first
- `DEFAULT_TIMEOUT` in `prog2.tests.CompilerTests.java` can be raised locally
- `PRINT_ASM_CODE = true` prints the generated code so you can try it in MARS
- `PRINT_VC = true` prints the generated verification condition

Questions?

Thank you and good luck!