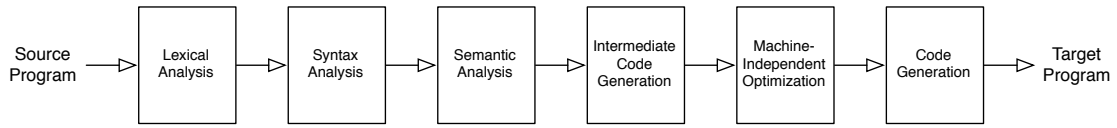
CS 301 Spring 2021 T4
Class Assignment
18 May

Tiny Compiler: Back End

1 Plan



It is time to build the TINY compiler back end. Our front end already parses and produces an AST. We first build a translation from AST to intermediate representation (IR): instructions for an abstract machine with infinitely many named storage locations. Next, we review a code generator to translate IR to x86 assembly code. Finally, we insert basic machine-independent optimizations on the IR.

1.1 Goals

☐ Preview course concepts by building a working compiler at accessible scale, with guidance. Concept exposure (not concept mastery) is the goal of this activity.

☐ Start learning Scala and associated tools. (`https://cs.wellesley.edu/~cs301/s21/tools/`)

☐ Try working with potential project teammates or tutorial group members.

1.2 Instructions

- **Complete this activity in groups of 3.** Choose teammates with whom you have not worked before. Include at least one CS 251 alum per group if possible.
- **Aim for a big-picture view of each compiler stage.** Do not worry if details are fuzzy: guess, follow intuition, experiment to see what works. Check in with me if you are lost or stumped.

2 Setup

Use a terminal in Linux, macOS, or WSL (Windows Subsystem for Linux).
WSL info: `https://cs.wellesley.edu/~cs301/s21/tools/#wsl`.

Start from your `tiny-front` code if it is done (`https://cs.wellesley.edu/~cs301/s21/project/tiny/`). Or, use these steps to get a completed front end:

1. If needed, install and configure IntelliJ, the Scala Plugin, and a Java 11 JDK:
   `https://cs.wellesley.edu/~cs301/s21/tools/#intellij-idea`
2. Clone the starter project:
   (a) Open IntelliJ and choose *New > Project from Version Control* from the menu.
   (b) Enter the URL `https://github.com/wellesleycs301s21t4/tiny-back.git`
   (c) Click *Clone*.
   (d) When prompted about whether to trust and open the BSP project, click *Trust Project*.
   (e) IntelliJ should now clone and automatically build the project. Wait for this to finish: watch progress bars at the bottom of the IntelliJ window.
3. Configure IntelliJ and a terminal session when IntelliJ has finished importing and building:

(a) Open the IntelliJ preferences/settings from the menu (*IntelliJ* > *Preferences* or *File* > *Settings*, varies by OS).

i. In *Build, Execution, Deployment* > *Compiler*, enable *Build project automatically.*

ii. Close the preferences/settings.

(b) Open a terminal, `cd` to the `tiny-back` project directory if needed, and run `source env.sh`. This produces wrapper scripts for the TINY compiler and interpreter and put them on the PATH.

4. Find files in the upper left *Project* pane:

(a) Find source code files in `tinyc/src/tiny`.

(b) Find test files in `test`.

3 Intermediate Representation

The first step in the back end is to translate our TINY AST to intermediate code for a simple abstract machine that is closer to the model of a real computer. Unlike ASTs, but similar to assembly code, intermediate code programs have flat, sequential structure.

Our abstract machine has infinitely many storage cells, each with a unique name: $\mathtt{x}_1$, $\mathtt{x}_2$, etc. The abstract machine supports a small set of instructions that read operands, perform simple computations on them, and store the result in a destination cell. Instruction operands, o, are either:

- the name of a storage cell (thereby referring to the cell's contents); or
- a literal number value.

A program is a linear sequence of instructions to be executed in order. The abstract machine supports the following instructions:

- $\mathtt{x} := o$
  A **copy** instruction reads a source operand, o, and stores it in the destination cell, $\mathtt{x}$.
- $\mathtt{x} := o_1 + o_2$
  An **add** instruction reads two source operands, o_1 and o_2, and stores the sum in the destination cell, $\mathtt{x}$.
- $\mathtt{x} := \mathsf{input}$
  An **input** instruction reads an integer input and stores it in the destination cell, $\mathtt{x}$.
- $\mathsf{print}\ o$
  A **print** instruction reads a source operand and prints it as output.

This style of abstract machine language is called *three-address code* (TAC), since each instruction addresses at most three operands.

Exercise 1. What does the following TAC program print if the user provides input 7?

$\mathtt{x}_1 := 3$
$\mathtt{x}_2 := \mathsf{input}$
$\mathtt{x}_3 := \mathtt{x}_1 + \mathtt{x}_2$
$\mathtt{x}_4 := \mathtt{x}_3 + \mathtt{x}_3$
$\mathsf{print}\ \mathtt{x}_4$
$\mathtt{x}_5 := \mathtt{x}_4 + 5$
$\mathtt{x}_6 := \mathtt{x}_5 + \mathtt{x}_2$
$\mathsf{print}\ \mathtt{x}_6$

Look at that TAC again. It looks similar to a TINY source program. In fact, putting aside superficial syntax differences, it appears that our TAC language is basically the TINY language with expression nesting restricted to one level. Nonetheless, we use an explicitly separate TAC representation in our TINY compiler (and in our later compiler, when there will be wider differences in the languages).

Exercise 2. Write a TAC program to print $4i_1 + 2i_2 + 5$, given any inputs i_1 and i_2, in that order.

As with the original TINY source syntax, this syntax is just a convenient way to write down a concrete representation of an abstract program structure. Unlike the original TINY syntax, we will never use this TAC syntax within our compiler. It is just a convenience for pencil and paper.

Exercise 3. Read the `IR.scala` file. Rewrite your answer from Exercise 2 as a Scala expression of type `Seq[TacInstruction]`. Use a `Vector`, which is also a `Seq`, and acts similarly to an array while also supporting insert, prepend, and append operations. For example, A `Vector[Int]` holding the numbers $1, 2, 3$ in order is constructed using the Scala syntax `Vector(1,2,3)`.

4 Translating ASTs to TAC

Next, we translate a TINY program AST to an equivalent TAC program. These are rather different structures. The program AST is a hierarchical, non-linear tree. The TAC program is a flat, linear sequence of instructions. ASTs have nested expression nodes. TAC instructions accept only storage cells or literals as operands. We must be sure that our translation is *semantics-preserving*. That is, the observable behavior (input and output for us) of the translated program must be identical to that of the source program. Like the *recursive* descent parser used to build the AST from a linear source code string, we will use a recursive algorithm to traverse the AST and emit a new linear structure.

The key is to traverse through the AST recursively, emitting one or more TAC instructions for each node. The result of each expression (*i.e.*, each node) will be stored in its own TAC storage cell. To translate an

expression node that needs this result, we emit an instruction that reads the contents of the result cell. Let us derive the algorithm for this translation. For now, omit variables and assignments.

Exercise 4. Draw the AST for the following TINY program:

```
print ( 7 + ( input + 5 ) ) ;
```

The TAC translation of this program is shown below. Match each node in the original AST to the corresponding instruction in the translated TAC. What tree traversal order (one of *in*, *pre*, *post*, *level*) could produce these instructions in this order? Try to reverse-engineer the algorithm that would generate this code given the AST you extracted. (Use pseudocode on paper or whiteboard. Stick to high-level description. Ignore Scala for now.) The algorithm is described below if you get stuck.

$\mathtt{x}_1 := 7$
$\mathtt{x}_2 :=$ input
$\mathtt{x}_3 := 5$
$\mathtt{x}_4 := \mathtt{x}_2 + \mathtt{x}_3$
$\mathtt{x}_5 := \mathtt{x}_1 + \mathtt{x}_4$
print $\mathtt{x}_5$

It is tempting to translate an AST for an expression `( 3 + 5 )` to a single TAC instruction, $\mathtt{x}_2 := 3 + 5$, or even simpler, $\mathtt{x}_2 := 8$. Resist the urge! Stay away from special cases; they will complicate the translation algorithm. (Spoiler alert: our optimizer will fix this for us later. Let's keep the concerns of translation and optimization separate.)

Exercise 5. Try your proposed AST-to-TAC translation algorithm on the following program:

```
print ( ( 2 + 4 ) + ( 6 + 8 ) ) ;
```

Write the translated TAC program here, then check your answer by evaluating it manually.

4.1 TAC Generation

The algorithm for generating TAC from an AST is explained generally here. The commented partial Scala implementation in `IRGen.scala` makes this more concrete.

For **expressions**, follow this basic plan:

- Create instructions that store the result (if any) of each AST node in a unique TAC cell. (Even an AST node as simple as an integer literal should be copied into a TAC cell!)
- Generate code by a *post-order* traversal of the AST:
  - Emit code for all subexpression children of this node, remembering the destination cell of the last instruction emitted by translating each child node.
  - Emit an instruction to complete the operation at this node, using the destination cells from the translated child expressions as the corresponding source operands of the instruction.

For **statements**, emit instructions to compute the expression (as above), then an instruction tho implement this statement using the last destination cell from the translated expression code as the source operand.

For **programs**, emit instructions for each statement in order. All three of these get slightly more involved when we introduce variables later.

Exercise 6. In `Compiler.scala`, uncomment the call to `backend` in `main`. In `IRGen.scala`, read the Scala code for `translateProgram` and the `Print` case in `translateStmt`. Ignore the `symtab` argument and the `Assign` case for now. Ask questions about anything that does not make sense. The following Scala operations are used:

- Pattern-matching: `match` with some `cases`. Quiz your CS 251 teammates about this, check the relevant section in the Scala documentation (`https://docs.scala-lang.org/tour/pattern-matching.html`), or ask me.
- Sequence (`Seq`) operations. We use immutable `Vectors` as sequences, applying these operations:
  - `s1 ++ s2` produces a new sequence with the elements of `s1` followed by the elements of `s2`.
  - `seq :+ elem` produces a new sequence with all of the elements of `seq` in order followed by the element `elem`.
  - `seq.last` returns the last element of non-empty sequence `seq`.
- The provided `fresh` function returns a new `TacCell` with a unique ID every time it is called.

Exercise 7. In `IRGen.scala`, implement `translateExpr`. The entire body is a single pattern-matching expression with one case for each type of `Expr`. Each case should return a `Seq[TacInstruction]`: a sequence of TAC instructions. We provide the `Input` case. Add the `Num` case, then test it on a simple program. Repeat for `Add`. Ignore `Var` for now. When you need to call `translateExpr`, pass the existing `symtab` argument along. (We use it later for variables.)

4.2 Variables and Symbol Tables

The final AST-to-TAC translation concern is variables. A natural representation of TINY variables exists: represent each unique TINY variable by a unique TAC cell. An assignment statement is translated to a copy instruction that copies the source into the variable's corresponding cell. Later variable references are

translated to instructions that copy from the same cell. This requires a durable mapping from variable name to TAC cell, since variable references may appear arbitrarily later in the program. For this, we use a *symbol table*. (A symbol table has more jobs in a compiler for a larger language.)

Translating an assignment statement requires the usual translation of the source expression and an instruction copying the expression result to the assigned variable's TAC cell. If the variable is already mapped in the symbol table, reuse its mapped TAC cell.[1] Otherwise, create a fresh TAC cell, map it in the symbol table, and use the fresh cell.

Exercise 8. Draw the AST for the following TINY program (or use the front end to generate it), then translate the AST to TAC by hand. Build a symbol table as you go to track which variable maps to which TAC cell. Double check your translated code: run it by hand and check the output.

```
x = ( 4 + input ) ;
y = input ;
print ( x + ( y + 297 ) ) ;
```

Exercise 9. Implement translation of assignment statements (the `Assign` case in `translateStmt`) and variable reference expressions (the `Var` case in `translateExpr`). Test on a few programs.

Our Scala code represents the symbol table with a mutable[2] `Map[String,TacCell]`: a map where keys are source-code variable names (as `String`) and values are the TAC cells that represent them (as `TacCell`). This single symbol table is shared by all levels of the translation algorithm since TINY has no scoping or control-flow constructs.

Entries for key `k` in map `symtab` are accessed with `symtab(k)` and updated with `symtab(k) = v`. Recall that, since we already implemented variable scope checking in the frontend of the TINY compiler, it is guaranteed that any variable use we see here follows its definition.

[1] CS 251 note: This implements a mutable variable semantics for TINY. Nonetheless, since TINY lacks any scoping or control-flow, mutable variables will act identically to immutable bindings with shadowing. In other words, we could just as well map a variable to a fresh cell at every assignment with no discernible difference. This implementation ambivalence is *not* present in more interesting languages.

5 x86 Code Generation

The final stage in a compiler is Code Generation: translation from the intermediate representation to code in the target language. Our TINY-to-x86 compiler targets x86 assembly language. There are actually many interesting machine-specific optimizations to do at this stage, such as mapping TAC cells to registers in a way the minimizes copying and use of the stack, choosing which variables could be mapped to registers vs. stack locations, selecting the best set of instructions, and more.

For the sake of building the TINY compiler quickly, our code generator is straightforward and applies no optimization at all. It does two basic tasks:

- Map TAC cells to stack locations in memory, which it does in truly dull fashion by using the TAC cell's ID to assign an offset into the stack frame.
- Translate each TAC instruction to one or more x86 instructions following a simple template.

After generating x86 assembly code, the compiler invokes the assembler and linker to build a complete binary executable. Take a look through `CodeGen.scala` if you are curious.

Exercise 10. Uncomment the section containing the `CodeGen` and `Link` stages in `Compiler.scala`, per the `FIXME` comments for this exercise. Run your TINY compiler on a program of your choosing. It should produce an executable file `path/to/x.tiny.bin` when given an input file called `path/to/x.tiny`. Now, for the moment of truth, run the command:

```
./path/to/x.tiny.bin
```

Does it work??? Debug until it does, then celebrate accordingly!

6 Optimization

Congrats, you have a compiler that reads TINY source code and produces an x86 executable! Let's make it even better. Instead of improving the simplistic code generator (feel free to think about that on your own), but our goal is to implement a few *machine-independent* optimization on the intermediate representation. Just as with the AST-to-TAC translation and the code generation stage, optimizations must be *semantics-preserving*.

Exercise 11. Consider briefly: what are some benefits of applying optimizations on intermediate code instead of on source code or machine code?

6.1 Design

Later in the course, we spend signficant time developing a general framework for expressing optimizations. For now, we implement a few standard optimizations with ad hoc techniques (though later in the course you will be able to look back and see the unifying thread). Just as with AST-to-TAC translation, a key rule for our optimizations is to keep them general. Consider this TAC:

$\mathtt{x}_1 := 4$

$\mathtt{x}_2 := 5$

$\mathtt{x}_3 := \mathtt{x}_1 + \mathtt{x}_2$

print $\mathtt{x}_3$

It is tempting to try to build an optimizer that automatically recognizes this pattern all at once and produces the equivalent program print 9. A pattern-based approach is actually useful in efficient machine-code generation, but at this stage, we want to avoid a large, unruly, error-prone collection of special cases.

An optimization is just a function that takes a TAC program as input and returns another (equivalent) TAC program as output. This means that we can make a larger optimization by composing two smaller optimization. We therefore develop a small suite of minimal, independent optimizations that do little on their own, but have big impact when composed.

6.2 Copy Propagation

You have likely been irritated at all of the wasteful copy instructions our AST-to-TAC translation emitted. They are everywhere! The *copy propagation* optimization can help eliminate them. Here is the basic idea: if a TAC instruction i has a storage cell as a source operand and this cell was last updated by a copy instruction, it is equivalent for i to use the source operand of that copy instruction instead.

Table 1: Copy propagation example.

Original		After Copy Propagation
$\mathtt{x}_1 := 300$		$\mathtt{x}_1 := 300$
$\mathtt{x}_2 :=$ input		$\mathtt{x}_2 :=$ input
$\mathtt{x}_3 := \mathtt{x}_2$		$\mathtt{x}_3 := \mathtt{x}_2$
$\mathtt{x}_4 := \mathtt{x}_1 + \mathtt{x}_3$	$\rightarrow$	$\mathtt{x}_4 := 300 + \mathtt{x}_2$
print $\mathtt{x}_4$		print $\mathtt{x}_4$

Table 1 shows a transformation of original TAC instructions on the left by applying two copy propagations to produce the new TAC instructions on the right. Both of the copy propagations target operands of the original instruction, $\mathtt{x}_4 := \mathtt{x}_1 + \mathtt{x}_3$:

1. Replace the source operand $\mathtt{x}_1$ by 300, since $\mathtt{x}_1$ was last updated by copying 300.
2. Replace the source operand $\mathtt{x}_3$ by $\mathtt{x}_2$, since $\mathtt{x}_3$ was last updated by copying the contents of $\mathtt{x}_2$ and $\mathtt{x}_2$ has not been changed since then.

Copy propagation stops here: we cannot predict the input that will be stored into $\mathtt{x}_2$ when the program is run, so we cannot propagate it as an operand to the add instruction. The print instruction uses $\mathtt{x}_4$, but it was produced by an add instruction, not a copy instruction. Again, it may be tempting to go further here, but the key is to keep it simple.

Exercise 12. Uncomment lines and remove a line as indicated by the `FIXME` note for this exercise to introduce the `Opt` stage in `Compiler.scala`. Rerun the compiler to make sure it still works.

Exercise 13. Read the code for `copyPropagate` in `Opt.scala` and try to understand how it performs copy propagation algorithmically. Make notes with inline comments.

6.3 Dead Code Elimination

Dead code is code that does nothing useful: code that computes a result that never affects any observable behavior of the program. (In TINY, that means it does not affect input/print.) For example, the second line of this TINY program is dead code:

```
x = input ;
y = ( x + 4 ) ;
print ( x + x ) ;
```

A result is computed and stored in `y`, but the value of `y` never affects any printed output. We could remove the second line without changing the observable behavior of the program. The *dead code elimination* optimization performs this removal automatically. Feel free to take a look at the implementation in `Opt.scala`.

Is it useful? Our AST-to-TAC translation never introduces dead code. TINY programmers may write dead code, but even if they do not, and more importantly, the copy propagation optimization often does cause code to become dead. Table 2 shows the continuation of the example from Table 1 by applying dead code elimination after copy propagation.

Table 2: Dead code elimination example.

Original		After Copy Propagation		After Dead Code Elimination
$x_1 := 300$		$x_1 := 300$	$\rightarrow$	
$x_2 :=$ input		$x_2 :=$ input		$x_2 :=$ input
$x_3 := x_2$		$x_3 := x_2$	$\rightarrow$	
$x_4 := x_1 + x_3$	$\rightarrow$	$x_4 := 300 + x_2$		$x_4 := 300 + x_2$
print x_4		print x_4		print x_4

6.4 Constant Folding

Our last classic optimization is *constant folding*. If a computation instruction (*e.g.*, add) has literals (constants) for all source operands, constant-folding pre-computes the result at compile time and replaces the instruction with a simple copy instruction, where the source is the literal pre-computed answer. In our case, this means replacing an instruction like $x_1 := 300 + 1$ with $x_1 := 301$.

Exercise 14. Implement `constantFold` in `Opt.scala`. Use the `map` method of sequences to produce a new sequence of TAC instructions where addition instructions meeting the constant-folding criteria are replaced by simply copies of constants and all other instructions are unchanged. Consult examples in `copyPropagate` or `deadCodeElim`, your CS 251 teammates, or ask for help with `map`. Scala's anonymous function syntax is `(param => body)`. The function `constantFold` is *much* smaller than the other two optimizations. A simple implementation is 4-5 sparse lines.

6.5 Composing Optimizations

Our AST-to-TAC translation never produces instructions that can be constant-folded, but just as with dead code elimination, constant folding becomes useful when copy propagation is used.

Exercise 15. Apply copy propagation to this TAC sequence manually. Then apply dead code elimination to the result. Then apply constant folding to that result. What do you get? Can you reapply optimizations to make the program even smaller?

$x_1 := 7$

$x_2 := 8$

$x_3 := x_1 + x_2$

print x_3

It turns out that these three optimizations can feed off each other if composed repeatedly. To get the best result, we run them until a *fixed point*, a TAC program for which a pass through all three optimizations produces the exact same program. At this point, there is nothing more for them to do.

Exercise 16. In the `apply` method in `Opt.scala`, replace `once` with `fixpoint`. Run the compiler on a couple Tiny programs to see how much the fixpoint improves optimization.

7 Reflections

At this point we have an optimizing compiler for Tiny. The language is, well, tiny, and the compiler could use smarter code generation and a few other improvements, but we have managed to consider interesting problems and solutions at most stages of a typical compiler architecture. Compilers for larger languages will complicate these problems significantly, so as we move on to start our deeper consideration of each compiler stage over the course of the semester, keep the following in mind:

The combined jobs of a compiler are complicated overall if we consider a compiler as a black box. A key to a clean, approachable implementation is to decompose large complicated problems into smaller simpler problems, solve those individual problems in a simple, elegant way, and then compose the solutions. This is true at a small scale, in the way we design recursive case-by-case translations or many simple individual optimizations. It is just as true at a large scale, where we break the compiler into several independent stages.

Hopefully you enjoyed building the Tiny compiler and gained some initial perspective for the rest of the semester. Do not worry if the details are fuzzy – it will take the whole semester to revisit all these items in detail! If you have any thoughts on how helpful these activities were or how to improve them, let me know. Happy compiling!

A Extending the Language (Optional)

Exercise 17. (Optional) Port your implementation of multiplication from the Tiny frontend to this version of the compiler and extend the backend to support multiplication as well. If you made good design decisions initially, this will be fairly straightforward. Feel free to experiment with other language improvements as well. Think ahead about how each feature impacts each stage of your compiler.