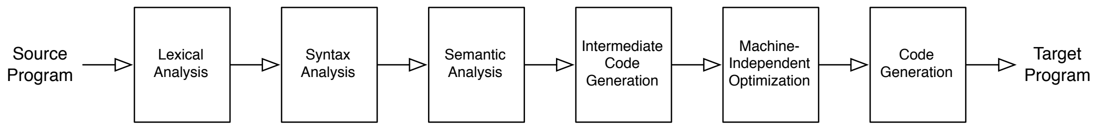
CS 301 Spring 2016
January 29 Lab

Tiny Compiler: Back End

Plan



After wrapping up loose ends in the front end of our TINY compiler, we turn to the back end. Here we make three main steps. Our front end already parses and produces an AST. We first build a translation to from AST to intermediate representation: instructions for an abstract machine with infinitely many named storage locations. Next, we take a quick look at a code generator I have written to translate this intermediate representation to x86 assembly code. Finally, backing up to the intermediate representation again, we add some basic machine-independent optimizations.

Directions:

- Work through this activity in groups of 2-3. **Choose different partners than the ones you had on Tuesday.**
- As you work, check in with me when you have questions.
- I will not collect anything from you, but do write your thoughts down for reference. Naturally, you will want to write down the programming parts too.
- **I do not expect to finish today**, but hopefully we will be able to run some compiled code. There is no requirement to continue work on this. Feel free to continue exploring if you are curious, but please prioritize your preparation for the first tutorial meeting. We will take a couple minutes in meetings or lab next week to hit the highlights.
- Refer to Scala documentation on the tools page as needed:
  `https://cs.wellesley.edu/~cs301/tools.html#scala`

Setup

With Mercurial, run:

```
hg commit; hg pull && hg merge && hg commit
```

This will commit your working version, pull my updates, merge them into your code, and commit the result.

I have provided a separate project (tiny2) that contains front end code for this part. If you want to keep yours, get the new code and then copy over all the files except `Parse.scala`.

Semantic Analysis

We do only minor *semantic analysis* in our TINY compiler. Typically that stage involves tasks like type-checking (we have only one type) and checking for proper use of variables. We will check to ensure variables are defined before they are used, but we will do so during the translation from ASTs to intermediate code, since it is so straightforward for TINY.

Intermediate Representation

The next step is to translate our TINY AST to intermediate code for a simple abstract machine that is closer to the model of a real computer exposed by assembly language, our eventual target. Unlike ASTs, 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 the source operand, o, and writes it in the destination cell, x.
- $\mathtt{x} := o_1 + o_2$
  An **add** instruction reads the two source operands, o_1 and o_2, and stores their sum in the destination cell, x.
- $\mathtt{x} :=$ input
  An **input** instruction reads an integer input from the user and stores it in the destination cell, x.
- print o
  A **print** instruction reads the 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?

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

Tangent. Look at that TAC again. It looks quite similar to a TINY source program with mildly different syntax. 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. A `Vector[Int]` holding the numbers $1, 2, 3$ in order is constructed using the Scala syntax `Vector(1,2,3)`.

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 with a recursive algorithm, 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 simply emit an instruction that will read the contents of the cell where the result is to be stored. Let us derive the algorithm for this translation. For now, omit variables and assignments.

Exercise 4. Draw the AST for the TINY program `print ( 7 + ( input + 5 ) ) ;`. Its TAC translation 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. (Feel free to go straight to that if short on time.)

$x_1 := 7$
$x_2 :=$ input
$x_3 := 5$
$x_4 := x_2 + x_3$
$x_5 := x_1 + x_4$
print x_5

Tangent. It is tempting to translate an AST for an expression `( 3 + 5 )` to a single TAC instruction, $x_2 := 3 + 5$, or even simpler, $x_2 := 8$. Resist the urge. Stay away from special cases; they will complicate our 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 program:

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

Check your answer by running your TAC program manually.

TAC Generation. The algorithm for generating TAC from an AST is explained generally here. The commented Scala implementation in `IRGen.scala` makes this more concrete. We provide part as an example; you implement the rest.

For expressions, follow this basic plan:

- Store the result (if any) of each AST node in a unique TAC cell. (Even 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 for this node, using the child destination cells as the corresponding source operands of the instruction.

For statements, emit instructions to compute the expression (using the plan above) and use the last destination cell in the resulting instructions as the source operand for an instruction to implement the statement.

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

Exercise 6. 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 (and possibly unfamiliar):

- Pattern-matching: `match` with some `case`s. Quiz your CS 251 teammates about this, check the relevant section in the *Scala for Java Programmers* tutorial linked on the web page, or ask me.
- Sequence (`Seq`) operations. We use immutable `Vector`s as sequences, applying the following operations:
  - `a ++ b` produces a new sequence with the elements of `a` followed by the elements `b`.
  - `seq :+ elem` produces a new sequence with the elements of `seq` followed by `elem`.
  - `seq.last` returns the last element of non-empty sequence `seq`.
- `fresh()`, which is provided, returns a new `TacCell` with a fresh, 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`s for now. When you need to call `translateExpr`, pass the existing `symtab` argument along. (We use it later for variables.)

Variables and Symbol Tables. The last 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 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`. `symtab.contains(k)` checks if `symtab` contains an entry for key `k`.

Exercise 10. If you did not happen to do this in Exercise **??**, add checks to ensure all variable references occur *after* definitions of their variables. This is simple: if the variable is not in the symbol table when trying to translate a reference to it, it has not been defined yet. This adds a line or two to `translateExpr`.

[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.

[2]Immutable would be fine if we also passed its state along in the return values.

x86 Code Generation

The last stage in a compiler is called Code Generation: it translates the intermediate code to code in the target language. For our TINY-to-x86 compiler, that's 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:

1. 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.
2. Translate each TAC instruction to one or more x86 instructions following a simple template.

Feel free to read through this later.

After generating x86 assembly code, we run the assembler and linker to build an executable.

Exercise 11. (The `Link` stage will not work on Windows.) Uncomment the `CodeGen` and `Link` stages in `Compiler.scala`. Run your TINY compiler on a program of your choosing. It should produce an executable file `a.out`. And, for the moment of truth: run the command `./a.out`

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

Optimization

Congrats! You have a compiler that reads TINY source code and produces an x86 executable! Let's make it even better. We will not improve the simplistic code generator (feel free to think about that on your own), but we will do some *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 12. Consider briefly (no more than a minute or two): what are some benefits of doing optimization on intermediate code instead of on source code or machine code?

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. It is tempting to see TAC like:

$x_1 := 4$
$x_2 := 5$
$x_3 := x_1 + x_2$
print x_3

and 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.

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 will 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. For example, we may safely replace this code:

$x_1 := 300$
$x_2 := \mathsf{input}$
$x_3 := x_2$
$x_4 := x_1 + x_3$
$\mathsf{print}\ x_4$

with this code:

$x_1 := 300$
$x_2 := \mathsf{input}$
$x_3 := x_2$
$x_4 := 300 + x_2$
$\mathsf{print}\ x_4$

via two copy propagations in $x_4 := x_1 + x_3$:

1. Replace the source operand x_1 by 300, since x_1 was last updated by copying 300.
2. Replace the source operand x_3 by x_2, since x_3 was last updated by copying the contents of x_2.

Copy propagation stops here: we cannot predict the input that will be stored into x_2 when the program is run, so we cannot propagate it as an input to the add instruction. The print instruction uses x_4, but it was produced by an add instruction, not a copy instruction.

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

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 `y` never affects any printed output. We could remove the second line without changing the observable behavior of the program. This is the purpose of the *dead code elimination* optimization. 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, this optimization is still useful because of copy propagation. Looking back at the copy propagation example, note that the unoptimized code contained no dead code, but the optimized code offers dead code elimination a moment to shine: a dead copy instruction $x_3 := x_2$.

Constant Folding. Our last classic optimization is *constant folding.* If a computation instruction (*e.g.*, add) has literals (constants) for all source operands, pre-compute the result at compile time and replace the instruction with a simple copy instruction, where the source is the literal pre-computed answer. In our case, this means replace instructions 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 instructions meeting the constant-folding criteria are replaced and the rest stay as is. Consult examples in `copyPropagate` or `deadCodeElim`, your CS 251 teammates, or Ben for help with `map`. Scala's anonymous function syntax is `(param => body)`. `constantFold` is *much* smaller than the other two optimization. My implementation is 4-5 sparse lines.

Composing Optimizations. Interestingly, 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 code. 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.

Reflections

At this point we have an optimizing compiler for Tiny. It could use smarter code generation and a few other improvements, but we have managed to consider interesting problems and solutions at just about every stage in 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 down into several independent stages.

I hope you enjoyed this exercise and find it useful to gain an initial perspective for the rest of the semester. If you have any thoughts on how helpful it was or how to improve it, please let me know. Happy compiling!