

Language Processor Construction Main Project

CS142b is a project course – so building an actual compiler is the main point of the class. Hopefully, along the way you will also be learning a lot more than you would learn simply by attending a lecture or reading a textbook. Obviously, compilers for typical programming languages are very large and complex, so in this course we are going to be using a relatively simple language. However, you will be learning and implementing the same optimization techniques that are at the core of all modern compilers and hopefully gain some skills that will be seriously helpful in your future careers (including for landing a job at companies such as Apple, Google, Meta, or Tesla).

In particular, your task will be to construct an optimizing compiler for the small and simple programming language `tiny`. The syntax of `tiny` is given at the end of this handout; the semantics of the various syntactical constructs are hopefully more or less obvious.

While it won't win any prizes for elegance, `tiny` was designed to be parsed by a recursive-descent parser with a single symbol lookahead. For example, for this reason assignments start with the keyword `let` and function calls start with the keyword `call` – you may consider this is a little bit ugly, but it removes the grammatical ambiguity that exists in many programming languages in which both assignments and function calls start with the same grammatical construct. For example, in C++ you may need to parse an unbounded number of symbols before you can decide whether you are seeing an assignment or a function call:

```
a[ p->x + 17 ] -> f = 7 // assign 7 to a member pointed to by array element a[p->x+17]
a[ p->x + 17 ] -> f ( 7) // call a routine stored in a function pointer a[p->x+17]->f
```

Only when you reach the equal sign in the first example or the opening parenthesis in the second can you decide whether this is an assignment or a function call, and there is no upper limit to the number of symbols / characters that you may need to parse before you can make that decision. The `tiny` language doesn't have this problem, nor does it have the “dangling else” issue that affects many programming languages.

All variables in `tiny` are of type integer and there are two different kinds of function: functions are either explicitly declared as `void` and don't return anything, or they return an integer. There are three predefined functions *InputNum*, *OutputNum*, and *OutputNewLine*. Note that, as is

common in many programming languages, the names of these built-in functions are not keywords of the language grammar but regular identifiers that just happen to be defined already when parsing starts. All arguments to functions are passed by value. To simplify matters, we limit functions to accept at most 3 parameters, but you can ignore this limit if you like (you may need to add additional instructions to the IR if you choose to go this route).

We will not be using any lexer or parser generator tools in this course but will build everything from scratch. It is much easier to understand what is going on, and you will have more control over the project. And because the language grammar is in the class LL(1), you should be able to build a recursive descent parser in no more than 2 hours. In fact, you should be able to build directly on the tokenizer and parser that you already constructed during the second warm-up assignment. The objective of this course is not to teach you about parsing, but about program optimization, which is a far more interesting problem that far fewer people understand.

The one tool that we *will* be using in this class is a graph visualization tool. It is almost impossible to debug the kind of complex dynamic data structures that are used in optimizing compilers without such tools. I would suggest that you use the simplest graph description language around, which is called “Dot” and which you can learn in about 5 minutes. The easiest way to translate a Dot graph description into a graph image is pasting the Dot program into the <http://www.webgraphviz.com> website, but there are many open-source graph viewers available as well that you can install on your machine. There are even plug-ins for various IDEs.

This is a complex project and there is a danger that you will be overwhelmed by it. I strongly recommend that you start off by implementing a subset of the language first, leaving out user-defined functions altogether – and then add these only after you have thoroughly debugged the rest of your compiler. The built-in input and output functions of the language can be mapped directly onto the corresponding operations available in the intermediate representation. You should use this shortcut in your compiler because it is much simpler than handling user-defined functions.

Project Step 1

Build a simple recursive-descent parser for a subset of the `tiny` language that leaves out user-defined functions. Your front-end should generate an SSA-based intermediate representation (IR) appropriate for subsequent optimizations. Your intermediate representation will be a dynamic data structure in memory that models both individual instructions and super-imposed basic blocks. Perform copy propagation and common subexpression elimination.

The operations encoded in instruction nodes consist of an operator and up to two operands. The following operators are available (the meaning of *Phi* will be explained in the lecture):

const #x define an SSA value for a constant

add x y addition
 sub x y subtraction
 mul x y multiplication
 div x y division
 cmp x y comparison

phi x1 x2 ***compute Phi(x1, x2)***

end end of program
 bra y branch to y
 bne x y branch to y on x not equal
 beq x y branch to y on x equal
 ble x y branch to y on x less or equal
 blt x y branch to y on x less
 bge x y branch to y on x greater or equal
 bgt x y branch to y on x greater

Operations for user-defined functions:

jsr x jump to subroutine x, the return value is the result of the operation
 ret x return result x from subroutine (argument ignored for void functions)
getpar1 define an SSA value for formal parameter 1
getpar2 define an SSA value for formal parameter 2
getpar3 define an SSA value for formal parameter 3
 ...
setpar1 x pass x as actual parameter 1
setpar2 x pass x as actual parameter 2
setpar3 x pass x as actual parameter 3

In order to directly model the built-in input and output routines without having to deal with the complexity of parameter passing for user-defined functions, we add three more operations:

read read
 write x write
 writeNL writeNewLine

The language `tiny` discourages but does not prohibit the use of uninitialized variables. When you detect such a case, your compiler should emit a warning, but should continue compiling while assigning an initial value of zero.

In order to simplify your compiler, you may assume that all input programs are syntactically correct, that constant arithmetic doesn't generate overflow or divide-by-zero conditions, that variables are declared before they are used, and that no variable is declared more than once (not even in different lexical scopes).

Project Step 2

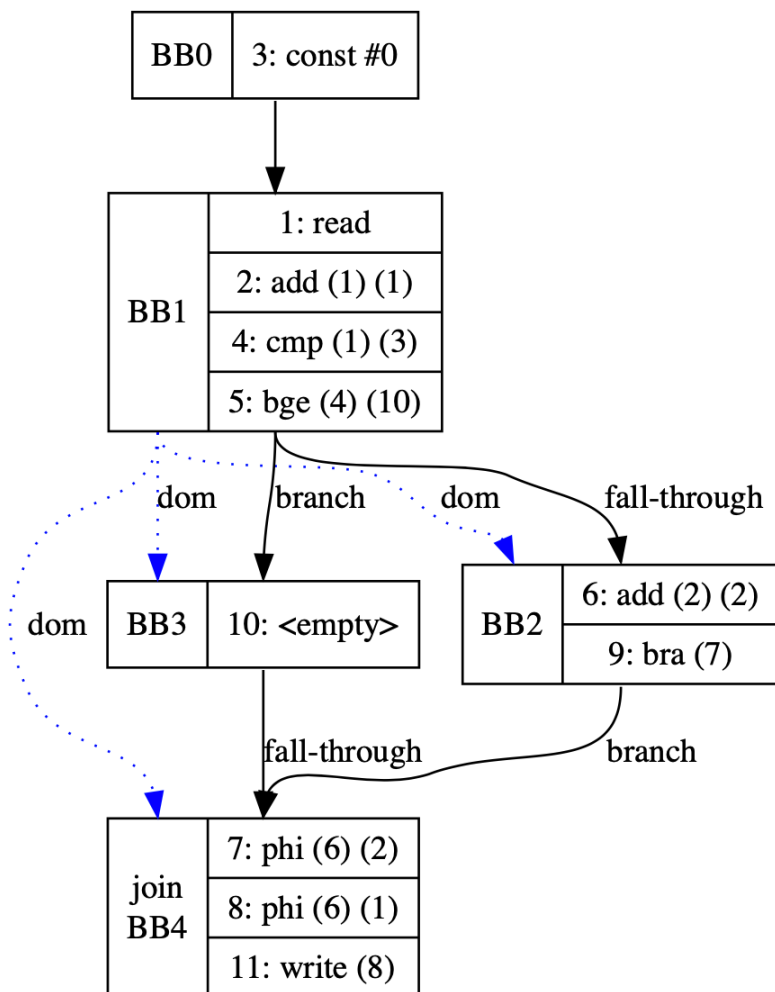
Build an IR visualization tool for your compiler that can walk the IR generated by your front-end and that generates a graph program appropriate for a graph visualizer. For example, the source program

```
main
var a, b, c, d, e; {
  let a <- call InputNum();
  let b <- a;
  let c <- b;
  let d <- b + c;
  let e <- a + b;
  if a < 0 then let d <- d + e; let a <- d else let d <- e fi;
  call OutputNum(a)
}.
```

might be translated into the following graph program in the *Dot* graph language:

```
digraph G {
  bb0 [shape=record, label="<b>BB0 | {3: const #0}"];
  bb1 [shape=record, label="<b>BB1 |
                                {1: read|2: add (1) (1)|4: cmp (1) (3)|5: bge (4) (10)}"];
  bb2 [shape=record, label="<b>BB2| {6: add (2) (2)|9: bra (7)}"];
  bb3 [shape=record, label="<b>BB3| {10: \<empty\>}"];
  bb4 [shape=record, label="<b>join\nBB4|
                                {7: phi (6) (2)|8: phi (6) (1)|11: write (8)}"];
  bb0:s -> bb1:n ;
  bb1:s -> bb2:n [label="fall-through"];
  bb1:s -> bb3:n [label="branch"];
  bb2:s -> bb4:n [label="branch"];
  bb3:s -> bb4:n [label="fall-through"];
  bb1:b -> bb2:b [color=blue, style=dotted, label="dom"]
  bb1:b -> bb3:b [color=blue, style=dotted, label="dom"]
  bb1:b -> bb4:b [color=blue, style=dotted, label="dom"]
}
```

Using a graph drawing engine such as GraphViz, this is then automatically translated into the following:



Note that the automatic graph layout engine in this case put the *else* block on the left and the *then* block on the right.

Project Step 3 (required for everyone)

When you are confident that arrays have been implemented correctly, add user-defined functions to your language. In the compiler's IR, each function is modeled as a separate control flow graph.

For each formal parameter that a function accepts, we have a matching **getpar** instruction that serves as the SSA definition of that parameter value. Similarly, at the call site we use the **setpar** instruction to associate the actual parameter value with the correct parameter "slot". It is up to the register allocator to eventually fulfil these constraints.

To make things much easier, you may interpret the `tiny` language definition in such a way that global variables are accessible only from inside the `main()` function.

Project Step 4 (required only for groups of two, three, or four)

Implement a global register allocator for your compiler. For this purpose, track the live ranges of all the individual values generated by the program being compiled, and build an interference graph. Color the resulting graph, assuming that the target machine has 5 general-purpose data registers. If more registers are required, map the values that cannot be accommodated onto virtual registers in memory. Eliminate all Phi-Instructions, inserting move-instructions wherever necessary. Display the final result using graph visualization, and perform experiments to test your implementation.

Project Step 5a (required only for groups of three)

Write a code generator that emits optimized (CSE, copy propagation, register allocation) *native* programs in the *native load format of a simulated or real platform*. **You may use the DLX processor simulator** or choose your target platform among x86/Windows, x86/Linux, or WebAssembly. **No target specific help will be given, so please do this only if at least one of you is a total machine code guru for your chosen platform.**

Project Step 5b (required only for groups of four)

Write a code generator that emits optimized (CSE, copy propagation, register allocation) *native* programs in the *native load format of a real platform*. You may choose your target platform among x86/Windows, x86/Linux, or WebAssembly. **No target specific help will be given, so please join a group of four only if at least one of you is a total machine code guru for your chosen platform.**

EBNF for the **tiny** Programming Language

letter = “a” | “b” | ... | “z”.

digit = “0” | “1” | ... | “9”.

relOp = “==” | “!=” | “<” | “<=” | “>” | “>=”.

ident = letter {letter | digit}.

number = digit {digit}.

factor = ident | number | “(“ expression “)” | funcCall¹.

term = factor { (“*” | “/”) factor }.

expression = term { (“+” | “-”) term }.

relation = expression relOp expression .

assignment = “let” ident “<-” expression.

funcCall = “call” ident [² “(“ [expression { “,” expression }] “)”].

ifStatement = “if” relation “then” statSequence [“else” statSequence] “fi”.

whileStatement = “while” relation “do” StatSequence “od”.

returnStatement = “return” [expression] .

statement = assignment | funcCall³ | ifStatement | whileStatement | returnStatement.

statSequence = statement { “,” statement } [“,”]⁴ .

varDecl = “var” ident { “,” ident } “;” .

funcDecl = [“void”] “function” ident formalParam “,” funcBody “;” .

formalParam = “(“ [ident { “,” ident }] “)” .

funcBody = [varDecl] “{” [statSequence] “}”.

computation = “main” [varDecl] { funcDecl } “{” statSequence “}” “.” .

Predefined Functions

InputNum() read a number from the standard input

OutputNum(x) write a number to the standard output

OutputNewLine() write a carriage return to the standard output

¹ only non-void functions can be used in expressions, for example `y <- call f(x) + 1;`

² functions without parameters can be called with or without parentheses

³ only void functions can be used in statements, e.g. `call do(); call this(x); call do;`

⁴ the semicolon is a statement separator; non-strictly necessary terminating semicolons are optional