CSE 302: Compilers | Lab 4

Procedures and Control Flow Graphs

Out: `2021-10-07`
Checkpoint 1: `2021-10-14 23:59:59`
Checkpoint 2: `2021-10-21 23:59:59`
Due: **`2021-11-02 23:59:59`**

---

1 INTRODUCTION

In this lab you will extend your source language, BX, with support for multiple procedures and procedure calls. This will require significant changes to the front end, and modest changes to the intermediate representations. You will also move from linear intermediate langauges to *control flow graphs* (CFG), and use it to implement some *control flow optimizations*.

This lab will be assessed. It is worth 20% of your final grade.

It is recommended that you work in groups of size **2**. Your submission *must* contain a file called `GROUP.txt` that contains the names of the group members.

2 STRUCTURE OF THE LAB

> **Note**
>
> This lab is significantly more complex than the first three labs. Do not put it off to the end. Steady effort is always more likely to succeed than a mad scramble at the last minute.

This lab has two checkpoints, one each at the end of the first two weeks. Each checkpoint by itself can be seen as worth $\frac{100}{3}\%$ of your grade, but earlier checkpoints will not be individually graded if later checkpoints or the final submission is achieved. In other words, you checkpoint 2 submission will override checkpoint 1, and your final submission will likewise override checkpoints 1 and 2.

Keep in mind that partial credit is given for incomplete attempts. Make sure to submit something on every due date regardless of how far you get.

This assignment handout is written in the same order as the checkpoint progression. The deliverables for the checkpoints are enumerated in their own sections, specifically 3.3, 4.4, and 5.3.

[Continued…]

3 CONTROL FLOW GRAPHS (WEEK 1)

In the first week of the lab you will be enriching the intermediate representation in terms of TAC into a control flow graph (CFG), and then perform some simple control flow optimizations. For the updates to the TAC language, see section 5.1.

3.1 *Basic Block Representation*

To transform a linear sequence of TAC instructions for a given procedure, first break it up into *basic blocks*. A basic block is a sequence of TAC instructions that has the following properties:

- The block begins with one or more local labels.
- The block ends with one of the following:
  - a return (`ret`) instruction; or
  - a sequence (zero or more) conditional jump (`jz`, `jnz`, `jl`, `jle`, `jnl`, `jnle`) instructions followed by an unconditional jump (`jmp`) instruction.
- Between the initial labels and the final jumps is the *body*, which consists of ordinary instructions (i.e., $\notin \{$`jmp`, `ret`, `jz`, `jnz`, `jl`, `jle`, `jnl`, `jnle`$\}$) and no other labels.

Basic Block inference As mentioned in lecture 5, to construct the basic blocks from the instructions of a given procedure, it suffices to perform the following in order.

1. Add an entry label before first instruction if needed.
2. For jumps, add a label after the instruction if one doesn't already exist.
3. Start a new block at each label; accumulate instructions in the block until encountering a jump (inclusive), a `ret` (inclusive), or another label (exclusive).
4. Add explicit `jmp`s for fall-throughs. All blocks must end with a `ret` or a `jmp`.

The very first block so inferred is special: we call it the *initial block*.

Control Flow Graph Given the basic blocks, it is a simple matter to construct the *control flow graph* (CFG) of the procedure.

- The nodes of the graph are the basic blocks
- The directed edges of the graph are given by the jumps at the end of the block. Each conditional or unconditional jump adds an edge from the current block to the block beginning with the jump destination label. Note that the CFG is a *simple graph*, so there is at most one edge for a given ordered pair (source, destination) of nodes.

The *successors* of a block are all the blocks that are the destinations of edges leaving from that block. Likewise, the *predecessors* of a block are all the blocks that have that block as a successor. For a block B, we write $\mathsf{next}(B)$ for the set of successor blocks of B; likewise, $\mathsf{prev}(B)$ for the set of predecessor blocks. Observe that all blocks but those that that end in `ret` have at least one successor; similarly, all but the the initial block have at least one predecessor.

Figure 1 contains an example of a CFG for an iterative Fibonacci procedure.

```
proc @fib(%n):
  %.Lentry:
    %0 = const 0;
    %1 = const 1;
    %2 = const 1;
  %.L1:
    jz %n, %.L3;
  %.L2:
    %n = sub %n, %2;
    %3 = add %0, %1;
    %0 = copy %1;
    %1 = copy %3;
    jmp %.L1;
  %.L3:
    ret %0;
```

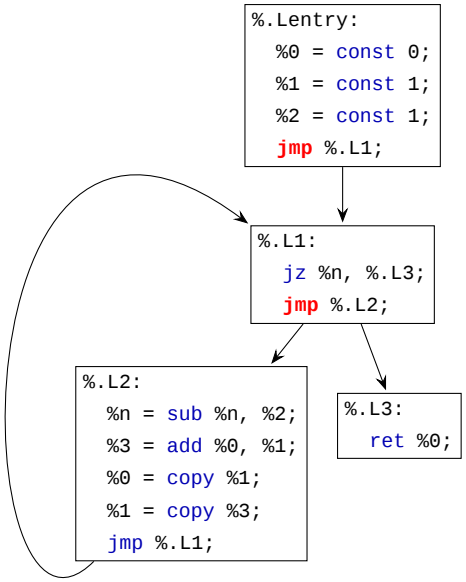


Figure 1: Example: from TAC to CFG. The `jmp`s in red were added in step 4 to prevent fall-through.

SERIALIZATION Since the CFG is just an internal abstraction of the procedure, it needs to be turned back into an ordinary TAC sequence before it can be compiled to x64 in the backend passes you will write in week 3. This process is known as *serialization* (aka. *scheduling*). Any ordering of the blocks of a CFG into a sequence that begins with the entry block is a valid serialization of the CFG.

During serialization, it is a good practice to try to place the block beginning with the destination of a `jmp` instruction right after the current block. This allows for the following simplification of the final TAC sequence, where `%.L0` is just used as an example.

```
1      // ...
2      jmp %.L0;
3  %.L0:
4      // ...
```

$\Longrightarrow$

```
1      // ...
2      // -- deleted: jmp %.L0;
3  %.L0:
4      // ...
```

Observe that the label `%.L0:` on line 3 is not removed because there could be other blocks that have this block as a successor block.

3.2 *Control Flow Simplification*

Once you have built the CFG, you can improve it in a number of ways. In this section are the three improvements that you should implement.

COALESCING Given two blocks B_1 and B_2 that are in a linear sequence—i.e., when $\mathsf{next}(B_1) = \{B_2\}$ and $\mathsf{prev}(B_2) = \{B_1\}$—the blocks can be merged into a single block as follows:

- The new block will have the labels of B_1.
- The body of the new block will be the concatenation of the bodies of B_1 and B_2. Note that the `jmp` at the end of B_1 is deleted.
- The new block will end with the jump or `ret` instructions of B_2.

This is commonly known as *coalescing*. It is useful to perform one round of coalescing after every other CFG simplification phase.

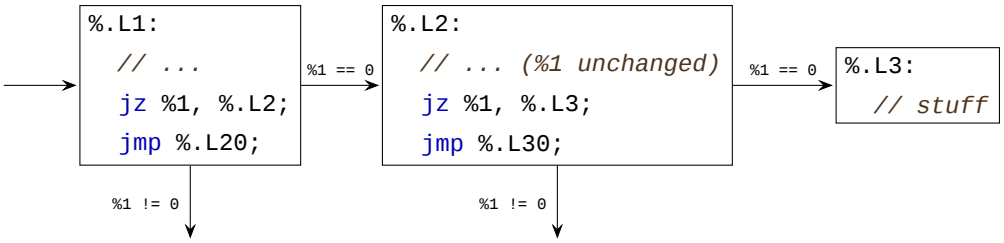
Unreachable Code Elimination (UCE) In a depth-first traversal of the CFG from the entry block by following the next() sets, all blocks that are not encountered are *unreachable* and hence will never be executed. They can safely be removed from the CFG without any change in meaning. This *unreachable code elimination* should also be run after every other simplification, particularly jump threading.

Jump Threading: Sequencing Unconditional Jumps Given a linear sequence of blocks $B_1, \ldots, B_n$—i.e., for each $i \in \{1, \ldots, n-1\}$ it is the case that $\text{next}(B_i) = \{B_{i+1}\}$ and $\text{prev}(B_{i+1}) = \{B_i\}$—where each of $B_2, \ldots, B_{n-2}$ have empty bodies and a single unconditional `jmp` at the end:

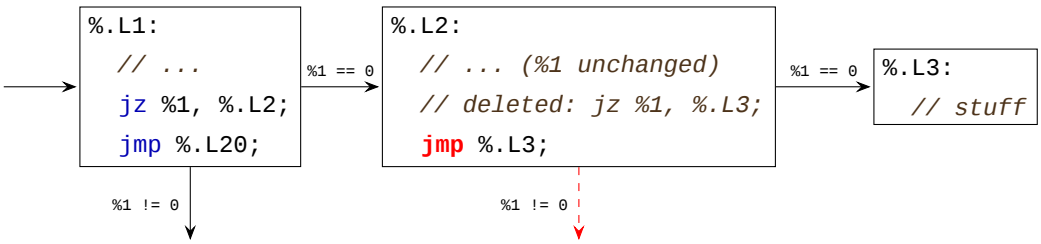
- Change the `jmp` instruction in B_1 to point to B_n instead, so that $\text{next}(B_1) = \{B_n\}$.
- This will make $B_2, \ldots, B_{n-2}$ unreachable, which can then be cleaned up with UCE.

With a little more work, this can also be adapted to the case where $\text{next}(B_1) \supsetneq \{B_2\}$, but here you will need to alter the jump destinations of conditional instructions.

Jump Threading: Turning Conditional into Unconditional Jumps For blocks B_1 and B_2 with $B_2 \in \text{next}(B_1)$, if it is the case that the condition that led to the jump from B_1 to B_2 is not altered in B_2, then testing for the same condition in B_2 with a conditional jump can be replaced with an unconditional jump. As an example, consider the following situation:



In this case, as long as there are no writes to `%1` in the `%.L2` block, the `jz` instruction in this block has a known result: it will always jump to `%.L3`. Therefore, we can simplify the CFG as follows:



Here, the edge between `%.L2` and `%.L30` (shown in red with dashes) is removed. This potentially causes the block with `%.L30` to become disconnected. A UCE pass should therefore be run to potentially remove it and everything it connects to that is not reachable by other paths. Moreover, observe that `%.L2` and `%.L3` are now a candidate for coalescing if `%.L3` has no other predecessors.

Work out the cases for other pairs of conditional jumps by yourself. Note in particular the `jl`/`jz` pair.

3.3 *Checkpoint 1: Deliverables*

The first deliverable of the lab will by a compiler pass called `tac_cfopt.py` that performs the control flow optimizations using the CFG representation that were described in section 3. Specifically, this pass should perform the following:

- CFG inference from linearized TAC
- Coalescing of linear chains of blocks
- Unreachable code elimination (UCE)
- Jump threading for unconditional jump sequences
- Jump threading to turn conditional jumps into unconditional jumps
- Serialization of the CFG back to ordinary TAC

Needless to say, your optimizations should not alter the observable behavior of your compiled code.

Your pass should read TAC (in JSON from) from a file specified in the command line and output TAC (also in JSON form) to standard output, or to a file specified using the `-o` option.

```
$  python3 tac_cfopt.py prog.tac.json
-- prints the optimized TAC(JSON) to standard output --

$  python3 tac_cfopt.py -o prog.cfopt.tac.json prog.tac.json
-- saves the optimized TAC(JSON) to prog.cfopt.tac.json --
```

To aid in your own debugging, you may find it useful to add a few command line flags to enable or disable certain optimizations. You can either add `--enable-opt`/`--disable-opt` flags for each optimization `opt`, or add a flag that takes a list of enabled optimizations, `--opts=opt1,opt2,...`.

To test your compiler pass, run it on the results of `bx2tac.py` from lab 3. You should see dramatic improvements in the case of compiled boolean expressions; e.g., `examples/bigcondition1.bx`.

[Continued…]

4 PROCEDURES IN BX (WEEK 2)

In the next update to the BX language, we have a language of *compilation units*, with each compilation unit consisting of a collection of *global variables* and *procedures*.

- *Global Variable Declarations*: variables that are placed in the `.data` section of the assembly file. These variables may be accessed by every procedure.
- *Procedure Declarations*: a procedure may take zero or more *arguments* (aka. *parameters*) and may optionally *return* a value. In BX any procedure may call any procedure, even itself or procedures that occur lexically later in the source file.

We will give a technical name to the two disjoint classes of procedures:

- *Functions* are procedures that return a computed value.
- *Subroutines* are procedures that do not return any values.

4.1 *Grammar and Structure*

Every compilation unit corresponds to a single BX source file, which continue to use the `.bx` suffix. To be a valid BX program, the compilation unit must define a `main()` subroutine that has no arguments or return value. When a BX program is executed, it is this `main()` subroutine that is called by the BX runtime. When the `main()` subroutine ends, control passes back to the runtime which terminates the program.

The full grammar of BX programs is given in figure 2.

Global Variable Declarations A global variable declaration is a variable declaration (⟨vardecl⟩) outside the body of any procedure. Like all variable declarations, global variable declarations must also have a declared type and initial value. However, the initial value for global variables is required to be a constant of the correct type: a number for `int`, and either `true` or `false` for `bool`.

The grammar rules for ⟨vardecl⟩ allow for multiple variables of the same type to be declared at once. Each (variable, initializer) pair is represented by ⟨varinit⟩, and a sequence of *one or more* ⟨varinit⟩s separated by the `","` token is represented by ⟨varinits1⟩.

Like all declarations, all global variables are accessible by all procedures, regardless of where they are declared. In particular, a procedure can use a global variable that is declared after it in the source.

Procedure Declarations A procedure must begin with the `"def"` token, followed by the name of the procedure, the argument list, the (optional) return type, and the body of the procedure. This is explained in the ⟨proc⟩ production in the grammar. The parameters of the procedure are defined by ⟨param⟩s, which could be empty or a sequence of one or more groups of parameter variables followed by their type (separated by a `":"`).

BX Expressions All the expression forms (⟨expr⟩) of BX continue to be valid expressions in BX. In addition, BX adds procedure calls which consist of a procedure name followed the arguments in parentheses. The argument list can be empty, or it can be *one or more* expressions separated by commas.

The `NUMBER` token is also given a slightly expanded syntax: negative numerals are now also parsed as `NUMBER` tokens instead of as applications of the unary `"-"` operator to a non-negative numeral.

⟨program⟩ ::= ⟨decl⟩* (must contain a `main()` procedure)

⟨decl⟩ ::= ⟨vardecl⟩ | ⟨procdecl⟩

⟨ty⟩ ::= `"int"` | `"bool"` (*the token `"void"` is also reserved, but unused*)

⟨procdecl⟩ ::= `"def"` `IDENT` `"("` (⟨param⟩ (`","` ⟨param⟩)*)? `")"` (`":"` ⟨ty⟩)? ⟨block⟩
⟨param⟩ ::= `IDENT` (`","` `IDENT`)* `":"` ⟨ty⟩

⟨stmt⟩ ::= ⟨vardecl⟩ | ⟨block⟩ | ⟨assign⟩ | ⟨eval⟩ | ⟨ifelse⟩ | ⟨while⟩ | ⟨jump⟩ | ⟨return⟩

⟨vardecl⟩ ::= `"var"` ⟨varinits⟩ `":"` ⟨ty⟩ `";"`
⟨varinits⟩ ::= `IDENT` `"="` ⟨expr⟩ (`","` `IDENT` `"="` ⟨expr⟩)*
(*initializers must be value literals for global vars*)

⟨assign⟩ ::= `IDENT` `"="` ⟨expr⟩ `";"`

⟨eval⟩ ::= ⟨expr⟩ `";"`

⟨ifelse⟩ ::= `"if"` `"("` ⟨expr⟩ `")"` ⟨block⟩ ⟨ifrest⟩
⟨ifrest⟩ ::= ϵ | `"else"` ⟨ifelse⟩ | `"else"` ⟨block⟩

⟨while⟩ ::= `"while"` `"("` ⟨expr⟩ `")"` ⟨block⟩

⟨jump⟩ ::= `"break"` `";"` | `"continue"` `";"`

⟨return⟩ ::= `"return"` ⟨expr⟩? `";"`

⟨block⟩ ::= `"{"` ⟨stmts⟩* `"}"`

⟨expr⟩ ::= `IDENT` | `NUMBER` | `"true"` | `"false"` | `"("` ⟨expr⟩ `")"`
| ⟨expr⟩ ⟨binop⟩ ⟨expr⟩ | ⟨unop⟩ ⟨expr⟩
| `IDENT` `"("` (⟨expr⟩ (`","` ⟨expr⟩)*)? `")"` (procedure calls)

⟨binop⟩ ::= `"+"` | `"-"` | `"*"` | `"/"` | `"%"` | `"&"` | `"|"` | `"^"` | `"<<"` | `">>"`
| `"=="` | `"!="` | `"<"` | `"<="` | `">"` | `">="` | `"&&"` | `"||"`

⟨unop⟩ ::= `"-"` | `"~"` | `"!"`

`IDENT` ::≈ `/[A-Za-z][A-Za-a0-9_]*/` (except reserved words)
`NUMBER` ::≈ `/0|-?[1-9][0-9]*/` (value must fit in 63 bits)

Figure 2: The lexical structure and grammar of the current fragment of BX.

BX Statements Most of the BX statement (⟨stmt⟩) forms continue unchanged, but there are some removals and additions. The ⟨print⟩ statement form is removed, and instead it is generalized to the ⟨eval⟩ form. In an ⟨eval⟩, the expression is evaluated to compute its value (if any), but the values are not stored anywhere. The `"print"` keyword is now demoted to an ordinary subroutine name, so the statement `print(42);` is now just an ⟨eval⟩ of the expression `print(42)`.

Finally, a ⟨return⟩ statement can either return nothing (for subroutines), or return a value (for functions). Every ⟨return⟩ statement represents an immediate exit from the procedure in which it occurs, so it can be seen as a kind of structured jump. Note that the argument of `"return"` is evaluated before the exit from the procedure.

4.2 *Scopes and Scope Management*

Info

This bit is largely the same as in lab 3, but since a lot of you were confused by it I'm adding a few words of explanation.

Scopes and Variables Like in lab 3, all variables are declared in some *scope*. A scope is a mapping from (variable) names to types, and you can think of it (not to mention implement it) as just a Python `dict`. Everywhere a variable declaration can occur, there is always a canonical *current scope*. For global variable declarations, this current scope is called the *global scope*.

A variable declaration is legal if the same variable has not already been defined in the current scope. Thus, the following two declarations are legal in the `main()` subroutine, where we have replaced the initializers with ellipses for now.

```
def main() {
  var x = ... : int;
  var b = ... : bool;
}
```

On the other hand, the following would be illegal since the variable `x` is redeclared in the same scope.

```
def main() {
  var x = ... : int;
  var x = ... : int;
}
```

Scope Stack and Shadowing Whenever a ⟨block⟩ is entered, the current scope changes to the scope of the block. The earlier scope does not disappear – it is merely *shadowed*. It is natural to think of this as a *stack of scopes*: whenever you enter a ⟨block⟩ by means of the `'{'` token, a new scope gets pushed to the end of the scope stack, and at the corresponding exit of the block at the `'}'` the scope that was pushed at the start is popped and discarded.

To look up the type of a variable that occurs in an expression, the scope stack is examined from last (most recently pushed) to first (least recently pushed); as soon as the variable is found in a scope, its corresponding type is used as the type of the variable occurrence. If the variable is not found in any of the scopes, then the variable is undeclared and there is therefore an error in the source program.

Note that while a variable cannot be redeclared in the same scope, it is allowed for an inner scope to have a variable with the same name as a variable in an outer scope. This is known as *shadowing*. Here is an example, where the variable x has been shadowed.

```
def main() {
  var x = 20 : int;
  {
    var x = 40 : int;    // shadows outer x
    print(x);            // outputs 40
  }
  print(x);              // outputs 20; the inner scope has been pop'd
}
```

4.3 *Type-Checking*

SEMANTIC TYPES Even though BX has only two types, `int` and `bool`, that are allowed in programs, during type checking it makes more sense to work with a larger collection of types. These *semantic types* exist only within the compiler and will be used to explain how type-checking behaves for BX.

- *Procedure Types*: these are types of the form: $(\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \tau_o$ where each of the τ_i and τ_o are types. Such a type represents the type of a procedure that takes n arguments of types τ_1, …, τ_n and returns a result of type τ_o.
- *Void Type*: the pseudo-type `void` stands for the result type of subroutines, i.e., procedures that do not return a value. There are no actual values possible of `void` type, and hence this type cannot be used for any of the argument types of a procedure.

In the rest of this section, whenever we mention "type", we will mean this enlarged space of types that includes procedure and void types.

EXPRESSIONS For the most part, the logic of BX from lab 3 continues to hold for type-checking expressions. To type-check an application expression — either a function call or an operator application — requires checking that the arguments to the function or operator have the expected argument types, and if so the result of the application is the result type of the function or operator.

STATEMENTS The statemens of BX continue to have identical type-checking rules. The new statement forms in BX are type-checked as follows.

- *Local Variable Declarations*: first, the initializer is type-checked against the declared type of the variable: if they match, then the variable is added to the current scope, assuming that it is not already present in the current scope.
- *Blocks*: upon entering the block, a new empty scope is pushed onto the scope stack. Each statement in the body of the block is then type-checked in sequence in this extended scope stack. Finally, on exiting the block, the scope that was added at the start of the block is then popped from the stack and discarded.
- *Evaluations*: the expression to be evaluated is type-checked. The type of the result is allowed to be any semantic type, including the type `void`, because the "result" of the evaluation is not stored.

- *Return*: here, there are cases for the kind of procedure the `return` statement is in:
  - *Functions*: an argument is mandatory to the `return` statement, which must be type-checked and its type must match the result type of the function. In your type-checker you may need to do something special to ensure that the full type of the procedure is known at the point where you are type-checking a `return` statement.
  - *Subroutines*: here, an argument to `return` is optional. If one is provided, it must type-check against the expected type `void`.

PROCEDURES Before type-checking the body of a procedure, it is important to know the types of all the other procedures and global variables. Therefore, type-checking of the entire compilation unit must proceed in two phases:

1. First, the types of all the global variables and procedures must be computed and stored in the global scope. This is easy to do, since the type of a global variable is part of the declaration and the procedure type of a procedure is easy to reconstruct from the declared argument and (optional) return types. For example, consider the following short BX program where the bodies and initializers have been replaced by ellipses.

   ```
   var x = ... : int;
   def main() { ... }
   def fib(n : int) : int { ... }
   def print_range(lo, hi : int) { ... }
   ```

   The global scope that is computed in phase 1 is: $\{$`x : int`, `main` : $() \to$ `void`, `fib` : $($`int`$) \to$ `int`, `print_range` : $($`int`, `int`$) \to$ `void`$\}$.

2. Second, every procedure body is type-checked with respect to this computed global scope. Thus, for example, in the body of the `fib` procedure, the type of `fib` is known and can be used to type-check recursive calls.

Note on Function Syntax

You should check that every function (i.e., a procedure with a return type) has a `return` statement on every possible code path, even those paths that may never be taken at runtime. Subroutines, unlike functions, don't require `return` statements.

SPECIALIZING `print()` The `print` procedure in BX is special: it can be used to print both `int`s and `bool`s. This means that it has both types $($`int`$) \to$ `void` and $($`bool`$) \to$ `void`. When type-checking `print` calls, which of the two types is picked depends on the type of the argument to `print`.

As mentioned in the lecture, during the process of type-checking `print` it makes sense to replace the generic `print()` call with calls to one of its specialized forms: `__bx_print_int()` or `__bx_print_bool()`. These functions are actually defined in the BX runtime (shown in figure 3).

4.4 *Checkpoint 2: Deliverables*

For the second week, you should update your BX parser and type-checker to accommodate the extensions to the BX language. Here are the expected deliverables for the first checkpoint.

```c
#include <stdio.h>
#include <stdint.h>
void __bx_print_int(int64_t x)  { printf("%ld\n", x);                    }
void __bx_print_bool(int64_t b) { printf(b == 0 ? "false\n" : "true\n"); }
```

Figure 3: The BX runtime, in file `bx_runtime.c`

- *Frontend Driver*: write an overall program called `bx2front.py` that runs the parser and type-checker alone, printing any error messages for incorrect input.

```
$ python3 bx2front.py ok.bx
– need not produce any output if everything is OK –

$ python3 bx2front.py incorrect.bx
TypeError: At file "incorrect.bx", line 2, character 9:
>   print(print(42));
          ^
Type mismatch: expected int or bool, got void
```

  The above is just an example of an error message. It is sufficient for your to simply say that a given source file fails to be well-formed or type-correct. A detailed diagnostic message is optional.
- *Test Cases*: Like in lab 3, you will be given a regression test suite for lab 4 containing a number of examples of BX source files that fail to parse, fail to type-check, or have other problems. These are in the `regression/` subdirectory.

Note that `bx2front.py` does not need to produce TAC! That will be the topic of week 3.

[Continued…]

5.1 TAC *extensions for procedures*

The TAC language needs some updates to support procedures.

- TAC programs are now *compilation units*, consisting of an unordered collection of *global variable declarations* and *procedures*.
- The temporaries of TAC remain largely unchanged: they are still 64-bit signed integers.
- In TAC, we now make a distinction between a *global name*, written with a prefix `@`, and a local name that has a prefix `%`. *In this lab, all global variables and procedures will be global names.* In fact, local names will only be used in one of the proposed projects concerning *namespace management*.
- Three new instruction opcodes have been added to TAC: `param`, `call` and `ret`. All the other instructions have now been modified to allow global variables for argument and destinations, in addition to temporaries as before.
- The `print()` statement is no longer a proper statement in BX, so the corresponding `print` opcode has now been removed from TAC. Instead, TAC will make ordinary procedure calls to one of the specialzed forms of `print()` function, explained in section 5.

TAC Global Variables A TAC global variable declaration reserves a global label for a global variable name and gives it the initial value, which must be an integer. If the value stored in the variable is intended to stand for a boolean, then the value must be 0/1-encoded. Here are some examples:

```
var @x = 42;
var @y = -42;
var @f = 0;     // value could stand for false
var @t = 1;     // value could stand for true
```

TAC Procedures A TAC procedure declares a procedure name together with its argument list, followed by the instructions that constitute its body. The argument list is either empty or a sequence of *named temporaries* separated by commas. Recall that a named temporary is any temporary whose name matches the regular expression `%[A-Za-z][A-Za-z0-9_]*`. Here are a few examples of TAC procedures:

```
def @main:
  %0 = const 42;
  param 1, %0;
  %_ = call @__bx_print_int, 1;
  ret %_;
```

```
def @fact(%x):
  %0 = const 1;
  jz %x, %.Lend;
  %1 = const 1;
  %2 = sub %x, %1;
  param 1, %2;
  %3 = call @fact, 1;
  %0 = mul %x, %3;
%.Lend:
  ret %0;
```

Note

A valid TAC program requires a `@main` procedure. For checkpoint 1, assume the TAC program is valid.

JSON Representation of TAC You have already seen that a TAC compilation unit is represented in JSON as a list of JSON objects. Each of these JSON objects representing a procedure has a `"proc"` key containing the name of the procedure, and a `"body"` key containing the list of TAC instructions in the body of the procedure. The only change for now will be when the procedure has arguments, which will be listed in an optional `"args"` key. Likewise, global variables are represented with JSON objects that contain the `"var"` key; each such object also lists the required initial value under the `"init"` key. You may ignore all other entries of the list, and any keys you don't recognize. To illustrate, the above global variables and procedures above would be represented in JSON as follows:

```
[ { "var": "@x", "init": 42 },
  { "var": "@y", "init": -42 },
  { "var": "@f", "init": 0 },
  { "var": "@t", "init": 1 },
  { "proc": "@main",
    "args": [],        // may also be omitted
    "body": [ ··· ] },
  { "proc": "@fact",
    "args": ["%x"],
    "body": [ ··· ] } ]
```

Keep in mind that, just like in BX, the order of the `"var"` and `"proc"` entries does not matter.

TAC Operands Unlike in earlier labs, TAC instructions can now take global variables and named temporaries as operands. The following are some examples in JSON.

```
{ "opcode": "add", "args": ["@x", "%y"], result: "%42" }   // %42 = add %x, %y;
{ "opcode": "neg", "args": ["%42"], result: "@y" }         // @y = neg %42;
```

Arguments and Calling Procedure calls in TAC are distributed over several instructions.

1. Each of the parameters of a procedure call are computed, left-to-right. The `param` instruction is then used to "lock in" that parameter. This instruction takes two arguments: the first is the position of the argument in the procedure call. Arguments in a call are numbered incrementally starting with position 1 for the first (leftmost) argument and ending with N for the last (rightmost) argument for a call with $N \geq 1$ arguments. The `param` call then reads and stores the value in the second argument, which is a temporary or a global variable.

   After each `param` call, the temporary or global variable in the second argument may be modified *without* affecting the value of the argument.

2. Once all the parameters have been locked in with `param` instructions, the `call` instruction is used to invoke the procedure call. It takes the name of the procedure as its first parameter, and the number of arguments as its second parameter. This latter number must match the number of `param` instructions before this `call` up to a previous `call` or the start of the procedure.

   The result of the `call` instruction, if any, stores the value returned by the procedure. If the procedure is not expected to return a value—i.e., if it is a subroutine—then the result is omitted.

Note

It is illegal for a call to a subroutine to have a temporary or a global variable as its destination. Type-correct BX2 programs should never compile to such TAC programs.

Returning a Result The `ret` instruction is used to exit a procedure. The optional argument to this instruction can be a temporary or a global variable. If the argument is missing, then the instruction is interpreted as a return from a subroutine call: the corresponding `call` instruction must not have been expecting any output.

Note

A single procedure may have many `ret` instructions, but every code path *must* terminate with a `ret`. That is, it is illegal for the execution to "fall through" the entire body of the procedure without encountering a `ret`.

Provided TAC Interpreter: `tacrun.py` As a debugging aid, you are provided with an implementation of a TAC interpreter, `tacrun.py`. It depends on the PLY library to parse ordinary TAC, and can also accept TAC in JSON form. Given a TAC program, `prog.tac`, you can use `tacrun.py` to interpret it symbolically to see its behavior:

```
$ python3 tacrun.py [-v] prog.tac          # use -v to increase verbosity of output
– several lines of output –

$ python3 tacrun.py [-v] prog.tac.json     # use -v to increase verbosity of output
– likewise –
```

5.2 *Compiling* TAC *to* x64 *(in the Unix/C ABI)*

Assembly language is already a language of compilation units. Compiling TAC to x64 is therefore a one-to-one map of global variables to *data* section and procedures to *text* sections. In the rest of this section, remember to use the canonical schematic image of the stack, figure 4.

Notes

- Remember to remove the `@` prefix from global labels when writing them to assembly.
- Pay attention to 16-byte alignment. All required alignment points are indicated with a "16" to the left in figure 4.

Compiling Global Variables Every global variable will be placed in the `.data` section. Here is an example for a global variable `@x` with initial value 42:

```
1       .globl x
2       .data
3   x:  .quad 42
```

Line 1 declares the symbol `x` to be a global symbol, meaning that it is accessible not only from within the compilation unit but also from external libraries. (This is not strictly necessary since—for now—TAC

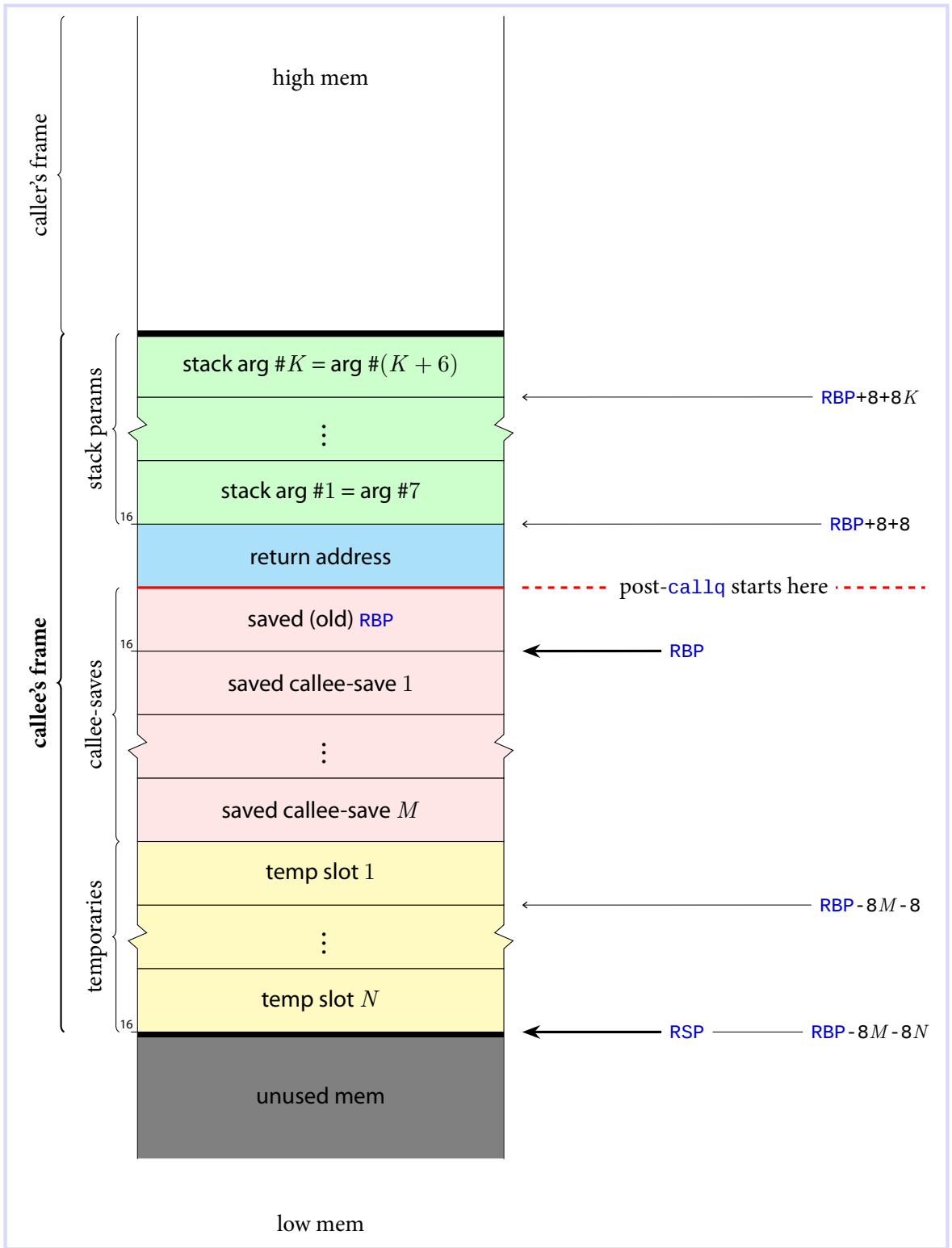


Figure 4: Schematic View of x64 Stack Frames

compilation units are in single files.) Line 2 places the symbol in the data section. Finally, line 3 declares a new label, `x`, and at that location places a *quad word*, which in AT&T/GNU Assembler terminology denotes four 16-bit chunks, i.e., 64 bits. The actual bit pattern that is stored in these 64 bits is given by the number following the `.quad` directive. Like all numbers that appear in assembly, this can also be given in hexadecimal with the `0x` prefix.

Compiling Procedures In labs 2 and 3 you were already writing assembly for the `main()` procedure. In this lab we merely generalize this to other procedures. Like in earlier labs, you will need to construct a map from TAC temporaries to stack slots. There are a few considerations.

- Temporaries that are arguments to the procedure—which, if you are following the instructions carefully, are named temporaries—are already pre-allocated in the stack from argument 7 onwards. The $(7+i)$th argument will be at *positive* offset $16+8i$ from `RBP`.
- The first six argument temporaries, on the other hand, are present in registers `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`, not stack slots.
- The remaining temporaries that occur in your procedure should be assigned to stack slots below `RBP`, just like you did in lab 3.

Let's go through these cases one by one.

First Six Arguments Here is an example of a procedure that takes 3 arguments and its corresponding x64 that shows how to save these temporaries to stack slots.

TAC

```
1  proc @add3(%x, %y, %z):
2    %0 = add %x, %y;
3    %1 = add %0, %z;
4    ret %1;
```

All five temporaries in `@add3()` are assigned to stack slots with the slot map shown in lines 7–8 on the right. Since 5 is not even, we allocate an extra temporary slot to ensure 16-byte alignment. Hence, we allocate space for 6 stack slots, i.e., 48 bytes. Then, the initial values of the three arguments, which come to the function in registers, are then stored in their stack slots to be reused. The rest of the code is exactly like in lab 3: reads of the input temporaries are converted into loads from their stack slots (e.g., lines 15–16).

x64

```
1      .globl add3
2      .text
3  add3:
4      # enable use of RBP
5      pushq %rbp
6      movq %rsp, %rbp
7      # slot map: {%x: 1, %y: 2, %z: 3,
8      #            %0: 4, %1: 5}
9      subq 48, %rsp # 8 bytes * 6 slots
10     # save the inputs in their slots:
11     movq %rdi, -8(%rbp)
12     movq %rsi, -16(%rbp)
13     movq %rdx, -24(%rbp)
14     # %0 = add %x, %y;
15     movq -8(%rbp), %r11
16     addq -16(%rbp), %r11
17     movq %r11, -32(%rbp)
18     # %1 = add %0, %z;
19     movq -32(%rbp), %r11
20     addq -24(%rbp), %r11
21     movq %r11, -40(%rbp)
22     # ret %1;
23     movq -40(%rbp), %rax
24     movq %rbp, %rsp  # restore RSP
25     popq %rbp        # restore RBP
26     retq
```

Argument #Seven and Up

Per figure 4, the arguments that are passed through the stack will have a *positive* offset from `RBP`. Unlike the first six arguments, there is no need to begin by storing these values anywhere, since they are already in the stack.

Be very careful when accessing memory locations above `RBP`. It is very easy to obliterate the old `RBP`, the return address (very bad!), or data in the stack frames higher up in the call chain. If you do any of that, your program will not only crash, but also be un-debuggable using gdb. In your compiler, when generating stack dereferences to these slots, it is a good idea to add extra assertions to make sure that the computed offset from `RBP` is never equal to $+0$, $+8$, or greater than $+8(K+1)$ where $K = \#\text{args} - 6$.

Loading and Storing from Global Variables

Reading and writing to global variables is different from accessing stack slots. In the code below, we show an extract of TAC and its corresponding assembly that reads from and writes to the global variable `@x`.

TAC

```
var @x = 42;

proc @main() {
  @x = neg @x;
}
```

The label `x` is a pointer into the data section of the loaded executable. To dereference such labels, use the notation `x(%rip)` instead of `(x)`. This is known as PC-relative or `RIP`-relative addressing.

x64

```
    .globl x
    .data
x:  .quad 42

    .globl main
    .text
main:
    # load @x
    movq x(%rip), %r11
    negq %r11
    # store @x
    movq %r11, x(%rip)
```

PC-relative addressing is the standard addressing mode for x64. If you don't use this, you will have to compile the assembly with position-independence turned off, using the `--no-pie` option to gcc. This will generate binaries that cannot be loaded as is anywhere in memory; rather, the executable loaded in your kernel must first rewrite all absolute addresses that occur in your executable based on where in memory it actually loads the executable. This is considerably slower, because the loader has to decode the entirety of the machine code of the executable, regardless of how much of it uses absolute addressing in reality. There are no good arguments for using `--no-pie` in modern code.

Compiling `param`

The `param` instruction is turned into `movq`s into the argument registers or `pushq`s to the stack.

- The first six `params` are simply turned into `movq`s to `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`.
- From argument 7 onwards, use `pushq` to save the arguments to the stack. Remember that the stack arguments have to be `pushq`ed in *reverse order*. If the TAC source already has these arguments in reverse order, there is no problem in compiling it. If not, you may have to pre-allocate a number of "scratch" temporaries (i.e., stack slots) that are used to reorder the arguments.

Note

Every `param` *must* be converted to a `movq` or a `pushq`. It is not valid to reorder the `params` directly in Python, because it will either violate the evaluation order of BX or it may accidentally switch the order of reads and writes to the same memory location, which is almost certainly a bug.

Compiling `call` and `ret` The `call` instruction is trivially mapped to `callq`. In x64, the return value from a procedure is delivered in the `RAX` register. After the `callq` instruction in x64, you should remove the values that were `pushq`ed to the stack by `param` earlier. This is easy: subtract 6 from the second argument to `call`, and, if > 0, *add* that many units of 8 bytes to `RSP`.[1]

To compile `ret`, there are two cases. If the argument to `ret` is an ordinary (non-void) temporary or global variable, then it can simply be `movq`ed to `RAX`. However, if the argument is the void temporary, `%_`, then it should be compiled as: `xorq %rax, %rax`, which zeroes out `RAX`. Doing otherwise risks propagating bugs in one function into other functions.

Notes

- It is better to have a *single* `retq` in your procedure and compile all the `ret` instructions into `jmp`s to an "exit" label that leads into this unique `retq`.
- At the exit label, remember to restore any callee-save registers to their original values, including `RBP` and `RSP`. Pay attention to the order of `pushq`s and `popq`s!

5.3 *Final Deliverables*

For the final week of the lab, you will update your `bx2tac.py`, `tac2x64.py`, and `bxcc.py` from lab 3 The final program, `bxcc.py`, should compile a BX2 program all the way to x64 assembly. It is recommended—but not required!—that you proceed as follows in writing `bxcc.py`:

1. If the typed AST generator (`bx2front.py`) that you wrote in checkpoint 2 finds syntactic, semantic, or type errors, then make `bxcc.py` output the corresponding error messages and stop. Otherwise, the generated typed AST will serve as the input for the next stage below.
2. Update the IR generator, `bx2tac.py`, that goes from the typed AST using (a suitably updated) maximal munch algorithm, as described in lecture 7. You can reuse most of your `bx2tac.py` code from lab 3.
3. Write a TAC to x64 instruction generator, called `tac2x64.py`, which is just an update to the similarly named program from lab 3. See the slides of lecture 7 for details.
4. Use the `tac_cfopt.py` from checkpoint 1 to optimize your TAC control flow.
5. Finally, make sure that you can compile and link your x64 programs to the updated BX runtime shown in figure 3.

As in lab 3, you compiler `bxcc.py` has only one requirement: to deliver a `.s` file from a `.bx` file. However, you may choose to have it also generate `.exe` and `.tac.json` files.

Final Submission Checklist

- ☐ `GROUP.txt`
- ☐ `tac_cfopt.py` (Checkpoint 1)
- ☐ `bx2front.py` (Checkpoint 2)
- ☐ `bx2tac.py` (Final week)
- ☐ `tac2x64.py` (Final week)
- ☐ `bxcc.py` (Final week)

[1] Recall that `pushq` first subtracts 8 from `RSP` and then saves the pushed value at the address it points to.