CSE 302: Compilers | Lab 4

Procedures and Control Flow Graphs

Starts:	`2020-10-08`
Checkpoint 1:	`2020-10-15 23:59:59`
Checkpoint 2:	`2020-10-22 23:59:59`
Lab due:	**`2020-11-02 23:59:59`**

---

1 INTRODUCTION

In this lab we will extend our source language from `BX1` to `BX2`, which mainly adds support for procedures. This will require significant changes to the front end, and modest changes to the intermediate representations. We will also move from linear intermediate langauges to *control flow graphs* (`CFG`), and implement some dataflow optimizations with the help of *static single assignment* (SSA) form.

This lab will be assessed. It is worth **20%** of your final grade. You should work in groups of 3.

Covid-19

Observe all health guidelines—particularly the interpersonal distancing guidelines—when working in groups. Use a video-conferencing platform to collaborate. If a physical meeting is unavoidable, try to do it outdoors, keep it short, and wear masks for the entire duration.

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 sheet is written in the same order as the checkpoint progression. The deliverables for the checkpoints are enumerated in their own subsections, specifically 3.4, 4.3 and **??**.

Warning

This document is not complete as of **2020-10-15**. Currently, it describes the tasks up to **checkpoint 2**. Stay tuned for the rest.

3 THE BX2 LANGUAGE

The next generation of the BX language family, BX2, departs from BX1 by becoming a language of *compilation units*, with computations being restricted to occur within *procedures*. A compilation unit is an *unordered* collection of the following:

- *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 BX2 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.

3.1 *Grammar and Structure*

BX2 Programs Every compilation unit corresponds to a single BX2 source file, which continue to use the `.bx` suffix. To be a valid BX2 program, the compilation unit must define a `main()` subroutine that has no arguments. In other words, the subroutine must look like this:

```
proc main() {
   // code
}
```

When a BX2 program is executed, it is this `main()` subroutine that is called by the BX2 runtime. When the `main()` subroutine ends, control passes back to the runtime which terminates the program. Note that while BX2 is not backwards compatible with BX1, a BX1 program can be converted to a BX2 program by surrounding it with `proc main() { ... }`.

The full grammar of BX2 programs is given in figure 1.

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 `'proc'` 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 ⟨params⟩, which could be empty or a sequence of one or more *parameter groups* (⟨paramgroup⟩) separated by commas. This is defined using the auxiliary nonterminal ⟨params1⟩ to stand for *one or more* ⟨paramgroup⟩s. Each ⟨paramgroup⟩ itself consists of *one or more* parameter variables (⟨paramvars1⟩) together with their common type separated by a colon.

⟨program⟩ ::= ϵ | ⟨decl⟩ ⟨program⟩

⟨decl⟩ ::= ⟨vardecl⟩ | ⟨proc⟩

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

⟨vardecl⟩ ::= `'var'` ⟨varinits1⟩ `':'` ⟨ty⟩ `';'` (*initializers must be value literals for global vars*)
⟨varinits1⟩ ::= ⟨varinit⟩ | ⟨varinit⟩ `','` ⟨varinits1⟩
⟨varinit⟩ ::= `IDENT` `'='` ⟨expr⟩

⟨proc⟩ ::= `'proc'` `IDENT` `'('` ⟨params⟩ `')'` ⟨retty⟩ ⟨block⟩
⟨params⟩ ::= ϵ | ⟨params1⟩
⟨params1⟩ ::= ⟨paramgroup⟩ | ⟨paramgroup⟩ `','` ⟨params1⟩
⟨paramgroup⟩ ::= ⟨paramvars1⟩ `':'` ⟨ty⟩
⟨paramvars1⟩ ::= `IDENT` | `IDENT` `','` ⟨paramvars1⟩
⟨retty⟩ ::= ϵ | `':'` ⟨ty⟩

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

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

⟨eval⟩ ::= ⟨expr⟩ `';'`

⟨block⟩ ::= `'{'` ⟨stmts⟩ `'}'`
⟨stmts⟩ ::= ϵ | ⟨stmt⟩ ⟨stmts⟩

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

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

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

⟨return⟩ ::= `'return'` `';'` | `'return'` ⟨expr⟩ `';'`

⟨expr⟩ ::= `IDENT` | `NUMBER` | `'true'` | `'false'` | ⟨call⟩
| ⟨expr⟩ ⟨binop⟩ ⟨expr⟩ | ⟨unop⟩ ⟨expr⟩
| `'('` ⟨expr⟩ `')'`
⟨call⟩ ::= `IDENT` `'('` ⟨args⟩ `')'`
⟨args⟩ ::= ϵ | ⟨args1⟩
⟨args1⟩ ::= ⟨expr⟩ | ⟨expr⟩ `','` ⟨args1⟩

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

⟨unop⟩ ::= `'-'` | `'~'` | `'!'`

`IDENT` ← `[A-Za-z_][A-Za-z0-9_]*` (*excepting keywords*)
`NUMBER` ← `0|-?[1-9][0-9]*` (*numerical value* $\in [-2^{63}, 2^{63})$)

Figure 1: The lexical structure and grammar of the BX2 language. New or altered nonterminals w.r.t. BX1 are highlighted in red. The precedence table for operators remains unchanged from BX1.

BX2 Expressions All the expression forms (⟨expr⟩) of BX1 continue to be valid expressions in BX2. In addition, BX2 adds procedure calls (⟨call⟩) which consist of a procedure name followed the arguments in parentheses. The argument list (⟨args⟩) can be empty, or it can be *one or more* (⟨args1⟩) expression 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.

BX2 Statements Most of the BX1 statement (⟨stmt⟩) forms continue to be valid BX2 statements, but there are some removals and additions. The ⟨print⟩ statement form from BX1 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)`.

Variable declarations (⟨vardecl⟩), blocks (⟨block⟩) and returns (⟨return⟩) are further new statement forms in BX2. A ⟨vardecl⟩ introduces a new variable in the most current *scope*; more on scopes in section 3.2. Like with global variable declarations, a local variable is declared together with its initializer, but in the case of local variables the initializer is allowed to be any arbitrary expression.

A ⟨block⟩ used to occur as the components of conditionals and loops, but now any ⟨block⟩ can be a statement by itself. Moreover, every ⟨block⟩ introduces a new (inner) scope, which allows for a fresh set of local variable declarations.

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.

3.2 *Scopes and Scope Management*

Scopes and Variables In BX2 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.

```
proc 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.

```
proc 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.

```
proc 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 poped
}
```

3.3 *Type-Checking*

Semantic Types Even though BX2 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 BX2.

- *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 BX1 continues to hold in BX2 for type-checking expressions. The operators of BX1 will be given the obvious associated procedure types. 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.

To type-check variables, the name of the variable will have to be looked up in the scope stack. This means that the scope stack should become a parameter to the type-checking procedure. This can be achieved in two ways. One way, shown in the lecture, is to make the scope stack a global variable that is manipulated by all the type-checking functions. This style is not recommended for your compilers since

it is very hard to write unit and regression tests for such things. A better option is to make the scope stack an actual parameter of the various type-checking functions, avoiding the problems of global variables.

Statements The statemens that are common to BX1 and BX2 continue to have identical type-checking rules. The new statement forms in BX2 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 BX2 program where the bodies and initializers have been replaced by ellipses.

   ```
   var x = ... : int;
   proc main() { ... }
   proc fib(n : int) : int { ... }
   proc 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.

SPECIALIZING `print()` The `print` procedure in BX2 is special: it can be used to print both `int`s and `bool`s. This means that it has both types `(int)` → `void` and `(bool)` → `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 4).

3.4 *Checkpoint 1: Deliverables*

For the first week, you should update your BX1 parser and type-checker to accommodate the extensions to the BX2 language. Here are the expected deliverables for the first checkpoint.

- *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 BX2 source files that fail to parse, fail to type-check, or have other problems. These are in the `regression/bad` subdirectory. For the first checkpoint, you should not only test your compiler on these test cases but also submit two other test cases.
  - One test case that fails for a reason different from any of the provided tests. Find a corner case of the language that is missed by the provided regression suite and write a test case for it.
  - One test case that passes and makes use of a feature in BX2 that was not present in BX1.

  Name these test files `bad.bx` and `good.bx`. All the contributed tests will then be released to everyone as additional tests starting in week 2.

4 UPDATES TO TAC AND COMPILING TO X64

4.1 TAC *for Procedures*

From the typed abstract syntax tree, we will now produce TAC output. However, the language TAC itself has been updated to match the changes from BX1 to BX2.

- TAC programs are now compilation units as well, 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. However, we add a new construct that looks like a temporary, called a *void temporary* and writte `%_`, which behaves like a hole: reading it doesn't yield a value, and writes to it are ignored. The void temporary

⟨program⟩ ::= ϵ | ⟨gvardecl⟩ ⟨program⟩ | ⟨procdecl⟩ ⟨program⟩

⟨gvardecl⟩ ::= `'var'` GLABEL `'='` NUM64 `';'`

⟨procdecl⟩ ::= `'proc'` GLABEL `'('` ⟨argtemps⟩ `')'` `':'` ⟨instrs⟩

⟨argtemps⟩ ::= ϵ | ⟨argtemps1⟩
⟨argtemps1⟩ ::= TEMP | TEMP `','` ⟨argtemps1⟩ (*all the TEMPs must be named temporaries*)

⟨instrs⟩ ::= ϵ | ⟨instr⟩ ⟨instrs⟩
⟨instr⟩ ::= `'nop'` `';'`
| LABEL `':'`
| ⟨var⟩ `'='` `'const'` NUM64 `';'`
| ⟨var⟩ `'='` ⟨unop⟩ ⟨var⟩ `';'`
| ⟨var⟩ `'='` ⟨binop⟩ ⟨var⟩ `','` ⟨var⟩ `';'`
| `'param'` NUM64 `','` ⟨var⟩ `';'`
| ⟨var⟩ `'='` `'call'` GLABEL `','` NUM64 `';'`
| `'ret'` ⟨var⟩ `';'`
| `'jmp'` LABEL `';'`
| ⟨jcc⟩ ⟨var⟩ `','` LABEL `';'`

⟨var⟩ ::= TEMP | GLABEL (*cannot be a procedure name*)
⟨unop⟩ ::= `'copy'` | `'neg'` | `'not'`
⟨binop⟩ ::= `'add'` | `'sub'` | `'mul'` | `'div'` | `'mod'` | `'shl'` | `'shr'` | `'and'` | `'or'` | `'xor'`
⟨jcc⟩ ::= `'jz'` | `'jnz'` | `'jl'` | `'jle'`

TEMP ← `%(_|0|[1-9][0-9]*|[A-Za-z][A-Za-z0-9_]*)`
LABEL ← `\.L[A-Za-z0-9_]+`
GLABEL ← `@[A-Za-z_][A-Za-z0-9_]*`
NUM64 ← `0|-?[1-9][0-9]*` (*value must be in* $[-2^{63}, 2^{63})$)

Figure 2: Grammar of TAC. The given interpreter, `tac.py`, accepts any ⟨program⟩.

serves as a placeholder for a required argument or destination position in an instruction but where no value can be supplied or is expected.

- In TAC, we now make a distinction between a *global name*, written with a prefix `@`, and a local name that has no 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.
- Since `print()` is no longer a proper procedure in BX2, 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.

Figure 2 contains the full updated grammar of TAC. We will now visit the above changes one by one.

TAC Compilation Units A TAC program consists of a sequence of global variable declarations (⟨gvardecl⟩) and procedure declarations (⟨procdecl⟩). As with BX2, the order of these declarations does make a difference to the behavior of the program, but each declaration must define a unique global label.

A ⟨gvardecl⟩ simply 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
```

A ⟨procdecl⟩ declares a procedure name together with its argument list, followed by the instructions that constitute its body. The argument list, ⟨argtemps⟩ 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 ⟨procdecl⟩s:

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

```
proc @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.

Void Temporary The temporary `%_` is a new form that has been added to TAC. It is intended to be used in places where no value is expected. The primary uses for it in TAC are:

- As the result temporary for a subroutine call, i.e., a call to a procedure of retrn type `void`.
- As the argument temporary for the return instruction (`ret`) inside subroutines.

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, and stored in a temporary or global variable. 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 destination of the `call` instruction stores the value returned by the procedure. If the procedure is not expected to return a value, the void temporary `%_` may be used as the destination.

Note

It is illegal for a call to a procedure of return type `void` to have a normal (non-void) temporary or a global variable as its destination. Type-correct BX2 programs should never compile to such TAC programs. The TAC interpreter, `x64`-generator, or the BX runtime are not required to detect this problem, since it is the job of the type-checker to guarantee that it doesn't happen.

RETURNING A RESULT The `ret` instruction is used to exit a procedure. The argument to this instruction can be a temporary or a global variable. If the argument is the void temporary, 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 FRAMEWORK: `tac.py` You are provided with an implementation of a TAC parser and interpreter, `tac.py`. It depends on the PLY library and the provided `ply_util.py` supplementary library. Given a TAC program, `prog.tac`, you can use `tac.py` to interpret it symbolically to see its behavior:

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

Internally, TAC is represented using three classes: `Gvar`, `Proc`, and `Instr`. Here are their constructors, where each parameter becomes a data attribute of the same name.

```
class Gvar:
    def __init__(self, name, value):
        """Create a <gvardecl>.

           name: a GLABEL (str)
           value: a NUM64 (int)"""
```

```
class Proc:
    def __init__(self, name, t_args, body):
        """Create a <procdecl>.

           name: a GLABEL (str)
           t_args: sequence of named temporaries (list or tuple)
           body: sequence of Instrs (list)"""
```

```
class Instr:
    def __init__(self, dest, opcode, arg1, arg2):
        """Create an <instr>. The arguments have the usual meanings
           already seen in labs 2 and 3."""
```

Two other functions in the `tac` module may be useful, particularly in debugging.

```
def load_tac(tac_filename):
    """Parse a TAC file with name `tac_filename'.
       Does not perform any checks such as repeated defs, etc.
       Returns a list of Gvar and Proc instances."""
```

```
def execute(gvars, prog, proc, args):
    """Execute a TAC procedure

       gvars: all global variables (dict mapping GLABELs to Gvar instances)
       prog:  all procedures (dict mapping GLABELs to Proc instances)
       proc:  the procedure to run (GLABEL)
       args:  values for all the arguments (tuple of NUM64s)
       Returns: int   (if `proc' is a function)
                None  (if `proc' is a subroutine)."""
```

4.2 *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 3

> **Note**
>
> Remember to remove the `@` prefix from global labels when writing them to assembly.

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 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.

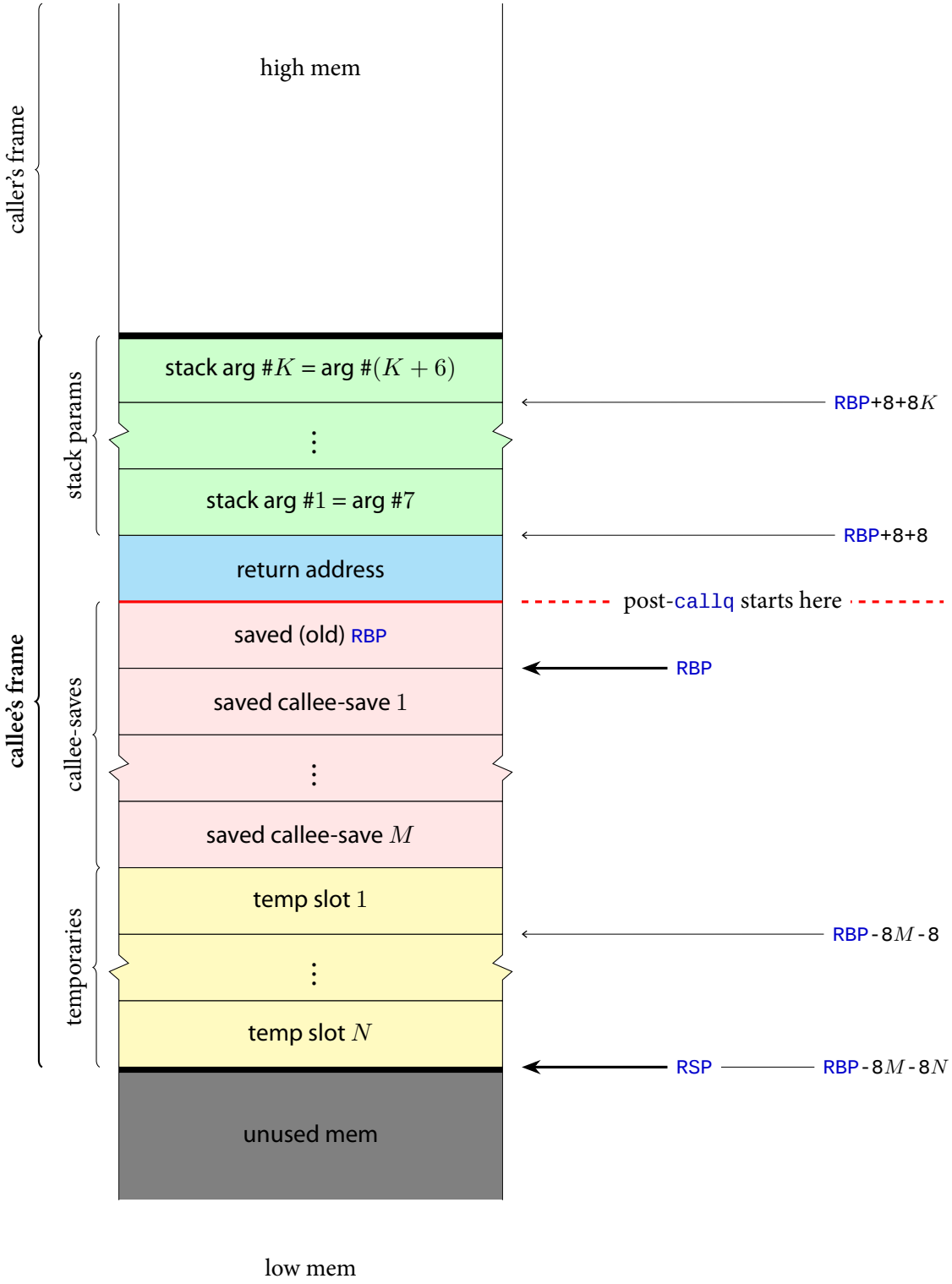

Figure 3: Schematic View of x64 Stack Frames

- 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. If you want to use any of these registers for any other purpose, you should allocate a stack slot to these arguments and `movq` the corresponding register to its stack slot.

  Note that the `imulq` and `idivq` instructions implicitly make use of `RDX`, so if your procedure uses these instructions then you have no choice but to allocate a stack slot for `RDX`.
- 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. Hence, we allocate space for 5 stack slots, i.e., 40 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 40, %rsp # 8 bytes * 5 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 3, 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

```
#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 4: The BX2 runtime, in file `bx_runtime.c`

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!

4.3 *Checkpoint 2 Deliverable: `bx2cc.py`*

For checkpoint 2, you will update your `bx1cc.py` from lab 3 into `bx2cc.py` and submit that. This program should compile a BX2 program all the way to X64 assembly. It is recommended—but not required!—that you proceed as follows:

1. If the typed AST generator (`bx2front.py`) that you wrote in checkpoint 1 finds syntactic, semantic, or type errors, then make `bx2cc.py` output the corresponding error messages and stop.
2. Write a TAC generator from the typed AST using (a suitably updated) maximal munch algorithm. Call this `bx2tac.py` and have it output a `.tac` file that you can test with `tac.py`. You can reuse most of your `bx1tac.py` code from lab 3.
3. Write a TAC to X64 compiler, called `tacx64.py`, which is just an update to the similarly named program from lab 3.
4. Finally, make sure that you can compile and link your X64 programs to the updated BX2 runtime shown in figure 4.

As in lab 3, you compiler `bx2cc.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` files.

Rest of the lab 4 assignment to come in week 3. Please re-download this file **after 2020-10-19**.

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