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 3.4, 4.4, and 5.1.

Warning

This assignment handout is incomplete past week 2. Please check back later for the full handout.

[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.

3.1 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. 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 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 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.

Most of these changes to TAC will be explained in section 5. For the first checkpoint, treat the `param` and `call` instructions as ordinary (non-jump) instructions. The only thing you need to worry about right now is the representation of TAC procedures.

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:

```
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. 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. To illustrate, the two procedures `@main` and `@fact` above would be represented as follows:

```
[ { "proc": "@main",
    // no "args"
    "body": [ ··· ] },
  { "proc": "@fact",
    "args": ["%x"],
    "body": [ ··· ] } ]
```

3.2 *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.

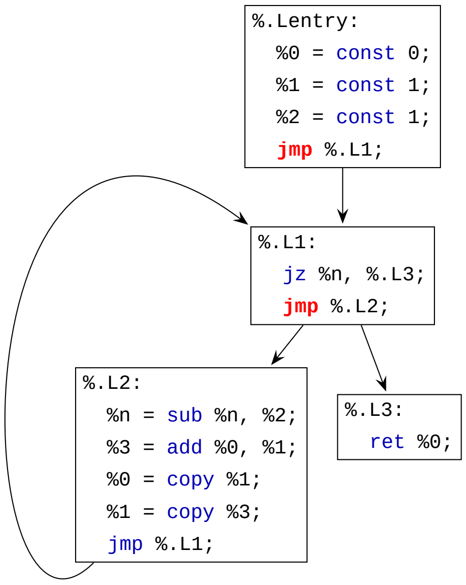
1. Add an entry label before first instruction if needed.
2. For jumps, add a label after the instruction if one doesn't already exist.
3. Start a new block at each label; accumulate instructions in the block until encountering a jump (inclusive), a `ret` (inclusive), or another label (exclusive).
4. 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.

```
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;
```



```
%.Lentry:
  %0 = const 0;
  %1 = const 1;
  %2 = const 1;
  jmp %.L1;
```

```
%.L1:
  jz %n, %.L3;
  jmp %.L2;
```

```
%.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.

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 next(B) for the set of successor blocks of B; likewise, 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.

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.

3.3 *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 $\text{next}(B_1) = \{B_2\}$ and $\text{prev}(B_2) = \{B_1\}$—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.

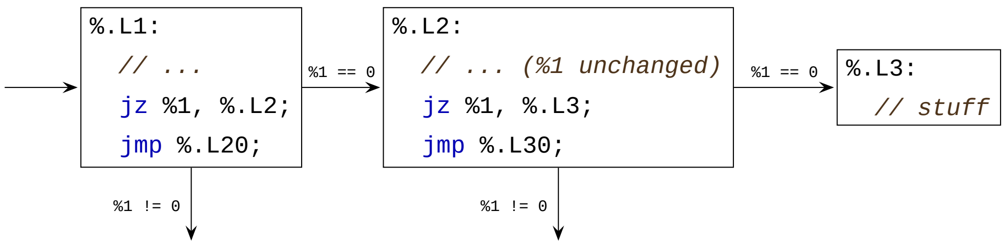
Unreachable Code Elimination (UCE) In a depth-first traversal of the `CFG` from the entry block by following the next() sets, all blocks that are not encountered are *unreachable* and hence will never be executed. They can safely be removed from the `CFG` without any change in meaning. This *unreachable 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:

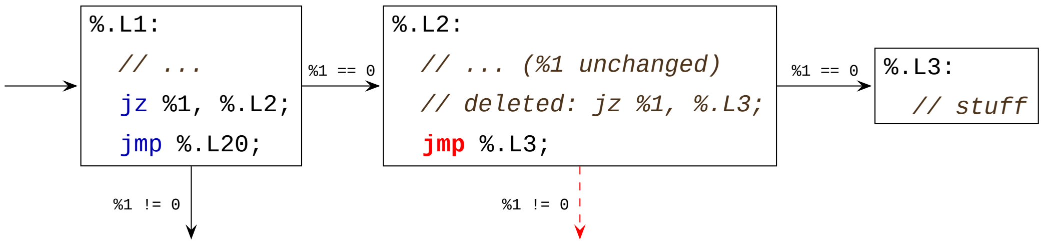
- Change the `jmp` instruction in B_1 to point to B_n instead, so that $\text{next}(B_1) = \{B_n\}$.
- This will make $B_2, \ldots, B_{n-2}$ unreachable, which can then be cleaned up with UCE.

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.

3.4 *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 <u>for global vars</u>*)

⟨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: $(\tau_1, \tau_2, \ldots, \tau_n) \to \tau_o$ 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 continues to hold in BX for type-checking expressions. The operators of BX 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. One way to achieve this, shown in lecture 4, is to make the scope stack a global variable that is manipulated by all the type-checking functions.

Statements The statemens that are common to BX and 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.

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 CODE GENERATION (WEEK 3)

Warning

This section is currently in flux.

5.1 *Final Deliverables*

Warning

This section is currently in flux.