Design and Implementation of the SuperCollider Interpreter

Brian Heim

IEM Graz

2018-06-16

Overview

- Compilers & interpreters crash course
- SuperCollider interpreter design
  - Compilation pipeline
  - Primitives
  - Stack
  - Byte code

Part I
Compilers & Interpreters
Crash Course

Compilers & Interpreters

- A *compiler* is a *translator*, typically from text to machine instructions
  - Typically, to assembly code, which is then translated by an *assembler* to machine code
  - Machine code = actual instructions for CPU
  - Assembly code = very light abstraction over machine code
- Compiled languages: C, C++, Swift, Rust

Compilers & Interpreters

- An *interpreter* is a compiler whose output language is virtual machine code
  - This is not the only definition of interpreter, but it's the one I like
- A *virtual machine* (VM) is a program that emulates a CPU
  - Think of it as a layer between the programming language and the underlying OS/chip architecture
  - The VM knows how to execute the compiler's virtual instructions as machine code instructions
  - May be a separate process or a function call
- Interpreted languages: Python, SuperCollider, JavaScript

Compilers & Interpreters

- What about Java?
- Java can be both compiled and interpreted (as Java byte code), depending on the implementation
- Using the JIT compiler, Java byte code in a program may be translated to machine code, at run-time!
- This makes frequently-used pieces of code run faster

Compilers & Interpreters

- Compared to compiled languages, interpreted languages:
  - Are easier to write
  - Run slower
  - Compile faster
  - May be more appropriate for a highly specialized domain

Language Design: Types

- Strong vs weak typing:
  - *Strong typing* means you will get a compiler (or runtime) error when trying to use one type as another
  - *Weak typing* means you will only sometimes, or never
  - Automatic conversions from integer to floating point in C are an example of weak typing
- Static vs dynamic ("duck") typing
  - With a *static type* system, the types of all expressions are known at compile time
  - With a *dynamic type* system, the runtime will need to make checks on the the types of objects and variables
    - This is called Run-Time Type Information (RTTI)
  - C and Haskell are statically typed; C++ is dynamically typed (using *virtual* classes)
- Compiled vs interpreted
- Paradigm
  - Imperative
  - Object-oriented
  - Functional
  - Logic-based
- Garbage collected?
- Safety vs speed

Language Design: Types

- Strong vs weak typing:
  - *Strong typing* means you will get a compiler (or runtime) error when trying to use one type as another
  - *Weak typing* means you will only sometimes, or never
  - Automatic conversions from integer to floating point in C are an example of weak typing
- Static vs dynamic (“duck”) typing
  - With a *static type* system, the types of all expressions are known at compile time
  - With a *dynamic type* system, the runtime will need to make checks on the the types of objects and variables
    - This is called Run-Time Type Information (RTTI)
  - C and Haskell are statically typed; C++ is dynamically typed (using *virtual* classes)

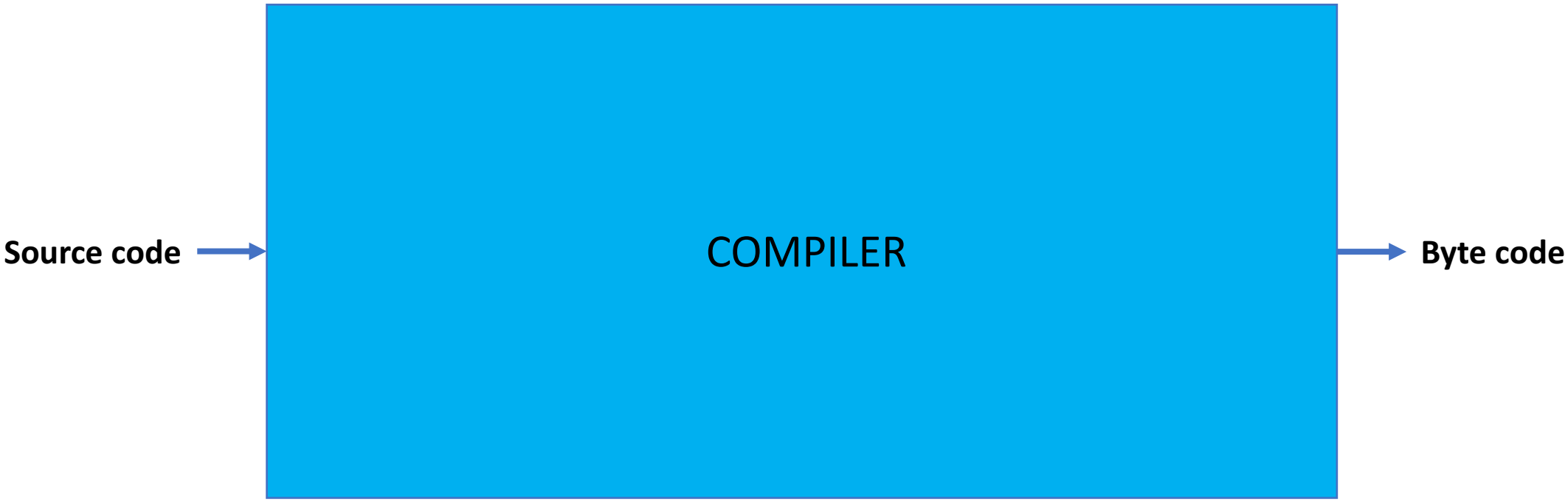
Language Design: Paradigms

- Languages can fit one or more of several design paradigms
  - Languages that support more than one category are called *multi-paradigm*
- *Imperative*: most closely models how a CPU runs
  - Statements are written in order from top to bottom
  - Ex.: C, Fortran
- *Object-oriented*: state is bundled with the functions that operate on it
  - Objects “own” and “encapsulate” how their internal state should behave
  - Ex.: SuperCollider, Java, C++
- *Functional*: no state or side-effects
  - All functions are pure, meaning the same inputs always produce the same outputs
  - Typically heavy use of recursion
  - Ex.: Haskell, ML, C++ (sometimes)
- *Logic-based*: programs are written as declarative logical statements
  - The compiler figures out the rest, like magic!
  - Ex.: Prolog

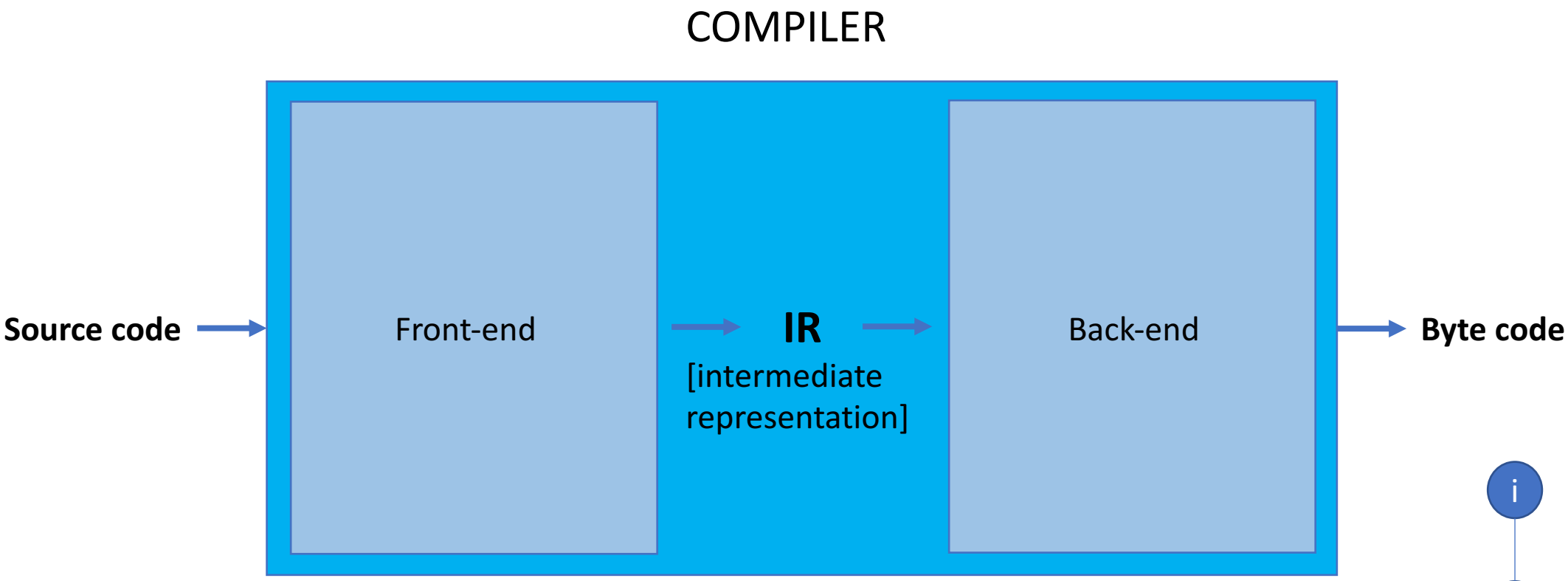
Language Design: Other considerations

- Garbage collection
  - In a garbage collected language, the runtime tracks all allocated memory and automatically disposes of it once it can no longer be referenced by the running program
  - Ex.: SuperCollider, Java, Python
  - This is typically slower than memory managed languages, but is less prone to errors and easier to use
- Speed vs. safety and simplicity vs. robustness are the main tradeoffs in language design
  - Above all, consider the intended purpose and expected users of the language

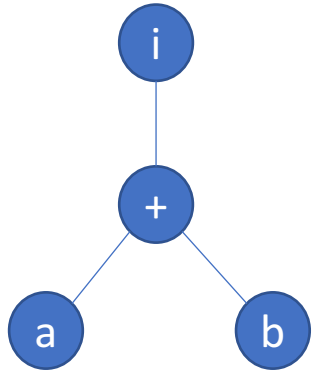
Compiler design



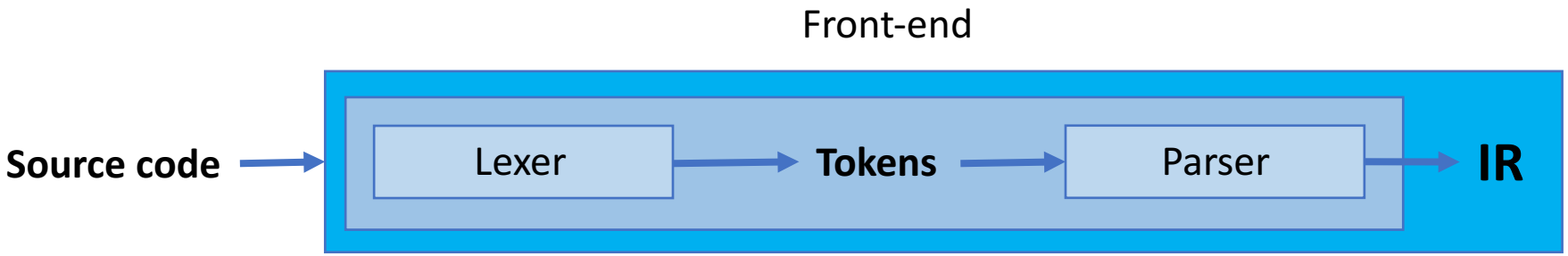
Compiler design



The *intermediate representation* is the compiler's refined representation of the source code. Typically, it is in a tree-based format. For example, a statement like `int i = a + b;` contains a lot of unimportant information. The compiler whittles it down to something like shown on the right. The IR will also contain some annotations about types, memory requirements, and other details depending on the language.

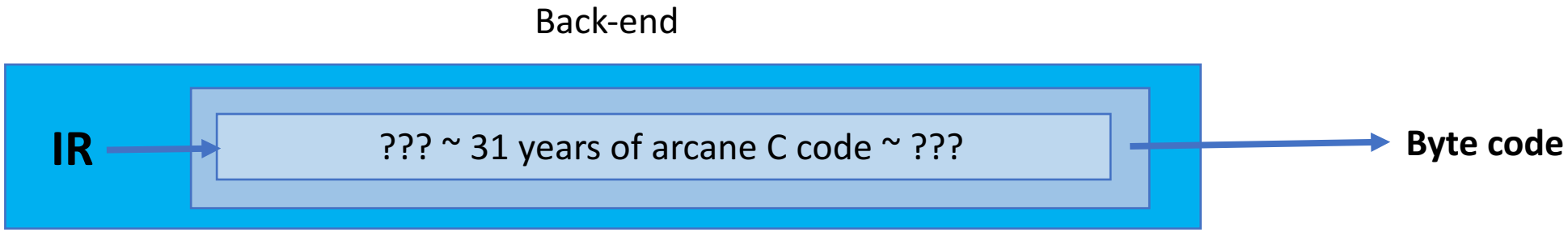


Compiler design – front-end



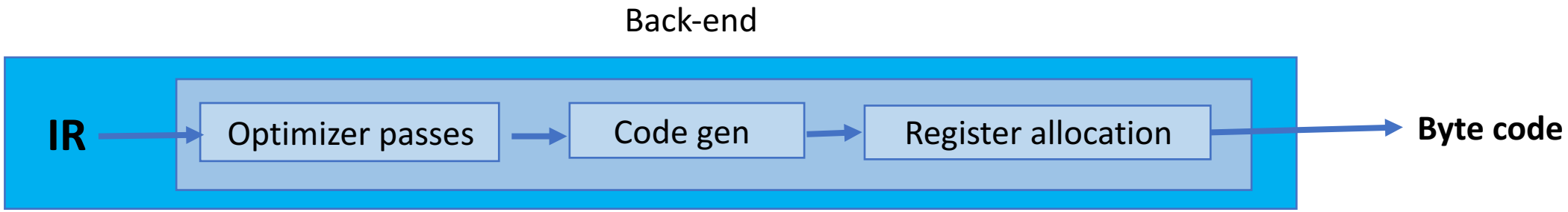
- A *lexer* converts raw text into *tokens*
- A *parser* uses the lexer to convert tokens into *IR*
- Typically, compiler authors use lexer- and parser-generators
  - *lex/flex* for lexers
  - *yacc/bison* for parsers
- Optional steps:
  - Semantic analysis: variable and function declaration order, function argument counts, etc.
    - The parser on its own usually can't tell if a statement like `a = b;` is correct; that requires knowing where 'a' and 'b' are declared, if at all.
  - Type checking, for statically typed languages

Compiler design – back-end



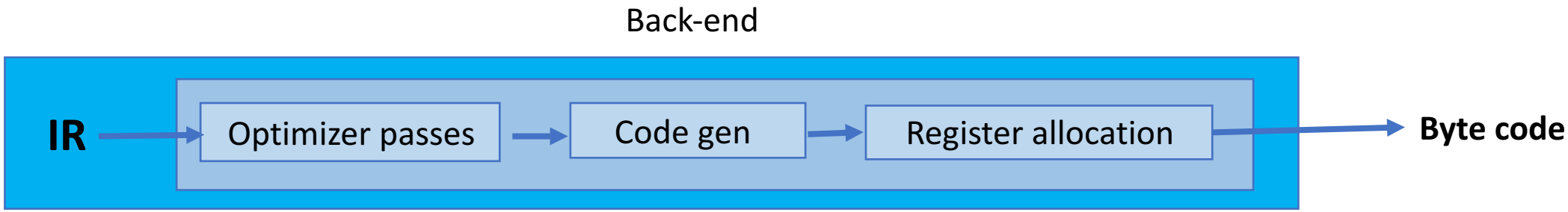
This is what usually happens in practice.

Compiler design – back-end



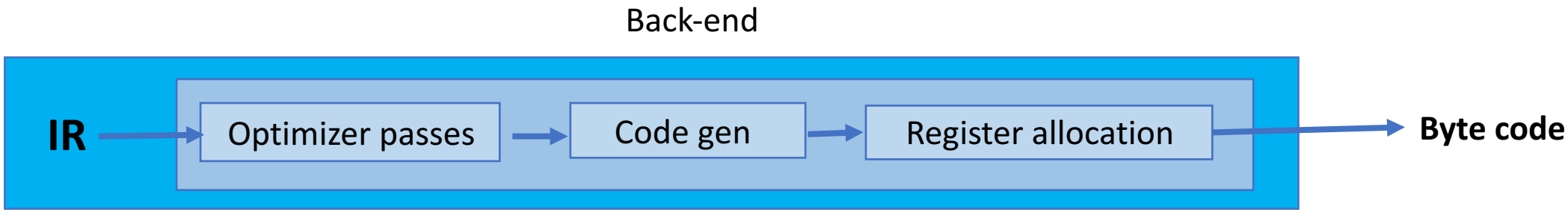
- *Optimization* phases help the code run faster by performing inference (and sometimes guesswork)
- Optimization examples:
  - Constant folding
  - Dead code elimination
  - Dead branch elimination
  - Function inlining
  - Loop invariants

Compiler design – back-end: optimization



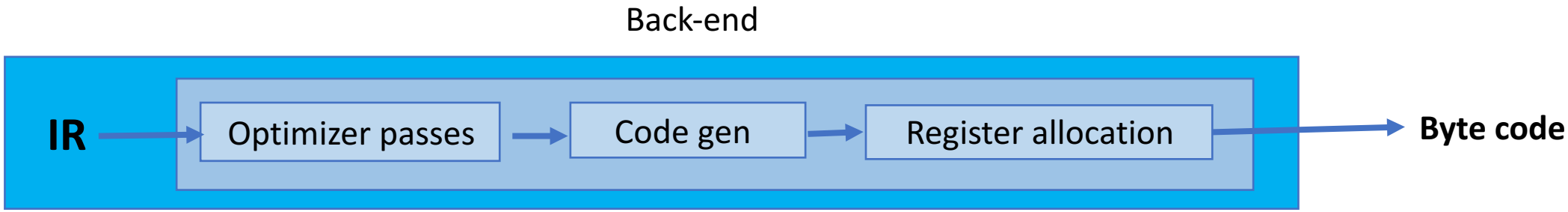
- *Optimization* phases help the code run faster using inference (and sometimes guesswork) on safe transformations
  - The same program can be represented by an infinite number of combinations of instructions. The optimizer's job is to find a really good representation.

Compiler design – back-end: optimization



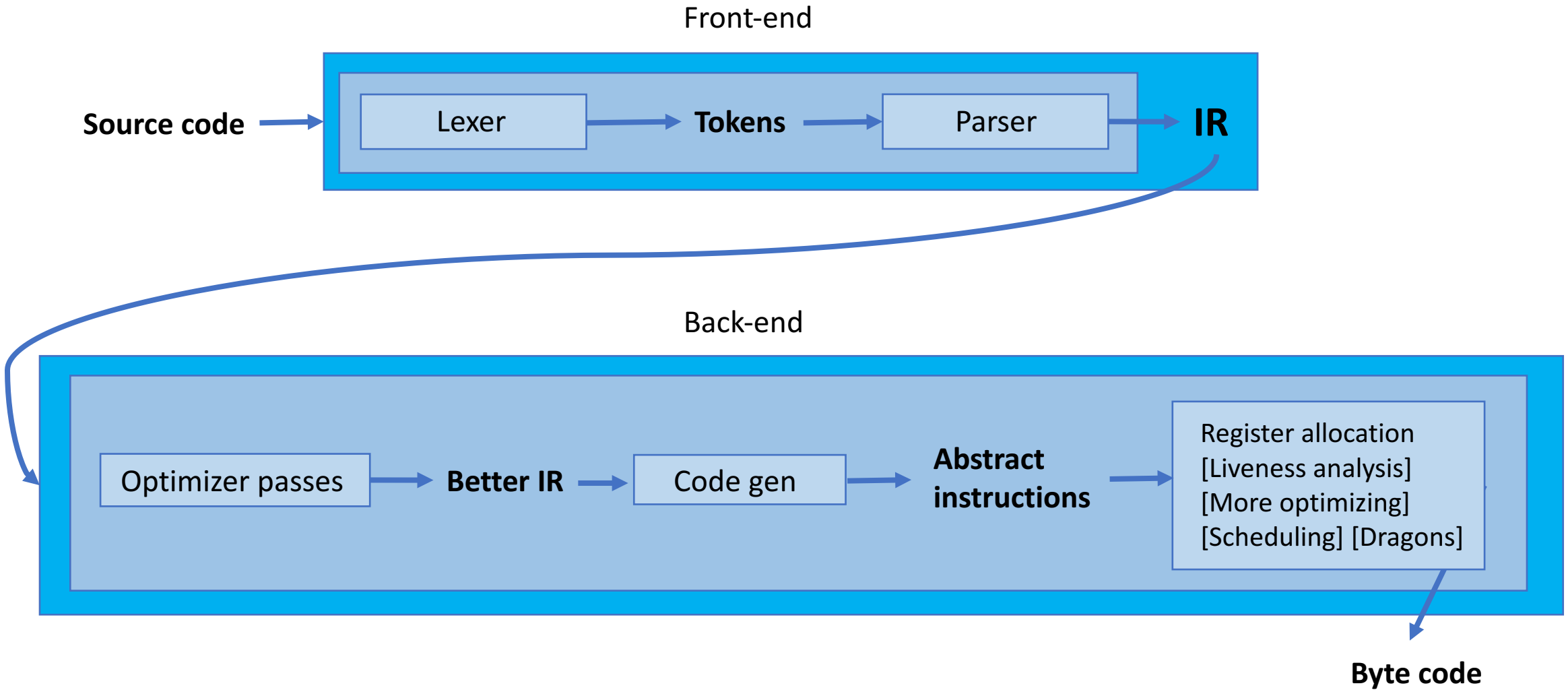
- Optimization examples:
  - Constant folding (`5 + 6 => 11`)
  - Dead code elimination
    - If a variable is assigned-to but never used, it can be eliminated
  - Dead branch elimination (`if(false)`)
  - Function inlining
    - As if the body of a function were written directly in the body of the calling function
    - Can eliminate lots of instructions for moving arguments around
  - Loop invariants
    - `for (int i = 0; i < 5; ++i) { x = 3; /* … */ }`
    - ”x = 3” can be safely moved outside the loop

Compiler design – back-end: optimization



- *Code gen[eration]* turns IR into "abstract" assembly (no registers assigned yet)
  - This makes it easier to generalize over various instruction sets
  - Optimization may also be performed after this
- *Register allocation* assigns specific symbols to specific CPU registers by analyzing the *liveness* of variables
  - *Registers* are the 32 or so pieces of memory the CPU can act on directly, for example with + or * or &&
  - If two variables never need to be used at the same time, their contents can safely be stored in the same CPU register
- *Instruction scheduling* – arranging instructions in the best possible ordering – is also done here
  - For example, loading a value from memory can take a long time. If the instructions after a load depend on the memory value, there will be a stall.
  - A good compiler interleaves instructions so that consecutive instructions have as few dependencies as possible (a good CPU will do this too!)

Compiler design – putting it all together



Part II
The SuperCollider Interpreter

Overview

- Lexer: handwritten [PyrLexer.cpp]
  - (possibly modified from machine-generated?)
- Parser: Bison-generated [lang11d / make_parser.sh]
  - Separate modes for class library parsing vs. JIT compilation
- IR: tree of `struct PyrParseNode` [PyrParseNode.h]
- Compiled into virtual machine byte code [PyrParseNode.cpp]
- Virtual machine [PyrInterpreter3.cpp]
  - `void Interpret(VMGlobals *g)`
- Runtime: garbage collector [GC.cpp], primitives calls [PyrPrimitive.cpp], objects as slots [PyrSlot.h]

Parser

```
classextension : '+' classname '{' methods '}'
        {
        $$ = (intptr_t)newPyrClassExtNode((PyrSlotNode*)$2, (PyrMethodNode*)$4);
        }
        ;

methods : { $$ = 0; }
        | methods methoddef
                { $$ = (intptr_t)linkNextNode((PyrParseNode*)$1, (PyrParseNode*)$2); }
        ;
```

- EBNF (Extended Backus-Naur form) notation – a way of expressing any context free grammar
- Token types are given as a sequence of formulas + code snippets
- Several formulas can be used to specify one token
- Bison converts this to C or C++ code

Slot

- A *slot* in SuperCollider is the basic representation of an object [PyrSlot.h]
- Essentially, a tagged union (value + RTTI)
  - The runtime must look at the value of the tag to know what the memory next to it represents; it could be a char, a double, a symbol, or anything else.
  - The meaning of each tag value is defined in another location.

```
/* in PyrSlot64.h */
typedef struct pyrslot {
          long tag;
          union {
                    int64 c; /* char */
                    int64 i;
                    double f;
                    void *ptr;
                    struct PyrObject *o;
                    PyrSymbol *s;
                    struct PyrMethod *om;
                    struct PyrBlock *oblk;
                    struct PyrClass *oc;
                    struct PyrFrame *of;
                    struct PyrList *ol;
                    struct PyrString *os;
                    struct PyrInt8Array *ob;
                    struct PyrDoubleArray *od;
                    struct PyrSymbolArray *osym;
                    struct PyrProcess *op;
                    struct PyrThread *ot;
                    struct PyrInterpreter *oi;
          } u;
} PyrSlot;
```

Primitives

- A *primitive* in SuperCollider is a foreign function interface (FFI) to C++ code
  - Primitive names are written with a leading underscore; e.g., “_SerialPort_Open”
- During startup, primitives are stored in a map :: name -> function
- At runtime, the primitive name is looked up in the map, and the corresponding function gets called with full access to the virtual machine state
- Primitive is responsible for checking types, unpacking slots, maintaining the virtual machine stack

Byte code

- *Byte codes* are the instructions given to the SuperCollider interpreter
- Range from simple [add 1] to complex [for-loop]
- Many chosen to optimize for common operations, especially looping
  - Integer.do, Integer.reverseDo
  - Many unary and binary operations (log, tan, <=, %)
  - Many array operations (collect, select, every, choose)
- Can get the bytecode listing for any function by calling

```
{ /* ... */ }.def.dumpByteCodes
```

VM Stack

- The SuperCollider VM uses a virtual *stack* to maintain local variables and pass arguments to function calls
- The context inside a function or method body is called a *frame*

Byte code example 1

```
(
{
  var x = 4, y = 3, z;
  x = x + y;
  z = x.squared;
  z;
}.def.dumpByteCodes
)
```

BYTECODES: (11)

0	30	PushTempZeroVar 'x’
1	31	PushTempZeroVar 'y’
2	E0	SendSpecialBinaryArithMsg '+’
3	80 00	StoreTempVar 'x’
5	30	PushTempZeroVar 'x’
6	DC	SendSpecialUnaryArithMsg 'squared’
7	80 02	StoreTempVar 'z’
9	32	PushTempZeroVar 'z’
10	F2	BlockReturn

The next couple slides walk through the compiled instructions for a simple block of code, line by line. Each line is color coded to show its corresponding instructions. The bytecodes are given exactly as ’dumpByteCodes’ shows them. The numbers in the first column are the indices of the instructions; the instructions are given in the second column in hexidecimal notation. The right-hand column is a brief description of the instruction.

Note: The variables x, y, and z are already present and initialized by the time the first byte code is executed. Thus, no byte codes to list for the first line.

Byte code example 1

```
(
{
  var x = 4, y = 3, z;
  x = x + y;
  z = x.squared;
  z;
}.def.dumpByteCodes
)
```

BYTECODES: (11)

```
 0  30      PushTempZeroVar 'x'
 1  31      PushTempZeroVar 'y'
 2  E0      SendSpecialBinaryArithMsg '+'
 3  80 00   StoreTempVar 'x'
 5  30      PushTempZeroVar 'x'
 6  DC      SendSpecialUnaryArithMsg 'squared'
 7  80 02   StoreTempVar 'z'
 9  32      PushTempZeroVar 'z'
10  F2      BlockReturn
```

Stack (0)

x (4)

y (3)

Stack (1)

x (4)

y (3)

Stack (2)

temp (7)

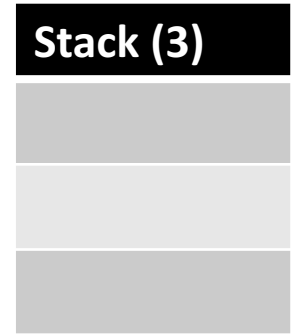
First, the interpreter loads the variables x and y onto the stack so they can be added together. The addition operation consumes the top two items on the stack and replaces them with the result (3 + 4 = 7). The diagrams on the right show the state of the stack after each byte code. In keeping with tradition, I have shown the stack as growing downward.

Byte code example 1

```
(
{
  var x = 4, y = 3, z;
  x = x + y;
  z = x.squared;
  z;
}.def.dumpByteCodes
)
```

BYTECODES: (11)

```
 0  30      PushTempZeroVar 'x'
 1  31      PushTempZeroVar 'y'
 2  E0      SendSpecialBinaryArithMsg '+'
 3  80 00   StoreTempVar 'x'
 5  30      PushTempZeroVar 'x'
 6  DC      SendSpecialUnaryArithMsg 'squared'
 7  80 02   StoreTempVar 'z'
 9  32      PushTempZeroVar 'z'
10  F2      BlockReturn
```



Next, the ”store” operation takes the top of the stack and stores it in the variable x. This leaves the stack empty.

Byte code example 1

```
(
{
  var x = 4, y = 3, z;
  x = x + y;
  z = x.squared;
  z;
}.def.dumpByteCodes
)
```

BYTECODES: (11)

```
 0  30      PushTempZeroVar 'x'
 1  31      PushTempZeroVar 'y'
 2  E0      SendSpecialBinaryArithMsg '+'
 3  80 00   StoreTempVar 'x'
 5  30      PushTempZeroVar 'x'
 6  DC      SendSpecialUnaryArithMsg 'squared'
 7  80 02   StoreTempVar 'z'
 9  32      PushTempZeroVar 'z'
10  F2      BlockReturn
```

Stack (5)

x (7)

Stack (6)

temp (49)

Stack (7)

Now, x is pushed back onto the top of the stack. The top of the stack is squared, which leaves the result value 49. Finally, 49 is popped off the stack and stored in z.

Byte code example 1

```
(
{
  var x = 4, y = 3, z;
  x = x + y;
  z = x.squared;
  z;
}.def.dumpByteCodes
)
```

BYTECODES: (11)

```
 0  30      PushTempZeroVar 'x'
 1  31      PushTempZeroVar 'y'
 2  E0      SendSpecialBinaryArithMsg '+'
 3  80 00   StoreTempVar 'x'
 5  30      PushTempZeroVar 'x'
 6  DC      SendSpecialUnaryArithMsg 'squared'
 7  80 02   StoreTempVar 'z'
 9  32      PushTempZeroVar 'z'
10  F2      BlockReturn
```

Stack (9)
z (49)

At the end of the block, z is pushed back onto the top of the stack, which effectively makes it the result value of the entire block. The instruction “BlockReturn” indicates to the interpreter that this is the end of the function, and program execution returns to whoever called the function. Note that we could have also simply written “`x.squared`” instead of the last two lines; 49 would have been left on the top of the stack and then used as the result value for the block.

Byte code example 1

```
BYTECODES: (11)
 0  30        PushTempZeroVar 'x'
 1  31        PushTempZeroVar 'y'
 2  E0        SendSpecialBinaryArithMsg '+'
 3  80 00     StoreTempVar 'x'
 5  30        PushTempZeroVar 'x'
 6  DC        SendSpecialUnaryArithMsg 'squared'
 7  80 02     StoreTempVar 'z'
 9  32        PushTempZeroVar 'z'
10  F2        BlockReturn
```

Internally, the compiler does not care that the names of these variables are ‘x’, ‘y’, and ‘z’. It only cares that they are the first, second, and third declared variables in the local scope respectively. The bytecodes that load and store variables encode this information – see how the bytecode for ”load x” is 30, while that for “load y” is 31. Similarly, the second byte of the “store” byte code indicates which local variable is to be stored to.

This is an optimization in the instruction set design to make accessing local variables faster. For code using more than 16 local variables, other byte codes are used.

Byte code example 2

```
(
{
  var x = 4, y = 5;
  if(x < y) {
    x = y;
  } {
    x = y.neg;
  };
  x;
}.def.dumpByteCodes
)
```

BYTECODES: (21)

0	30	PushTempZeroVar 'x‘
1	31	PushTempZeroVar 'y‘
2	E8	SendSpecialBinaryArithMsg '<‘
3	F8 00 07	JumpIfFalse 7 (13)
6	31	PushTempZeroVar 'y‘
7	08 00 00	StoreTempVarX 'x‘
10	FC 00 05	JumpFwd 5 (18)
13	31	PushTempZeroVar 'y'
14	D0	SendSpecialUnaryArithMsg 'neg'
15	08 00 00	StoreTempVarX 'x'
18	F0	Drop
19	30	PushTempZeroVar 'x'
20	F2	BlockReturn

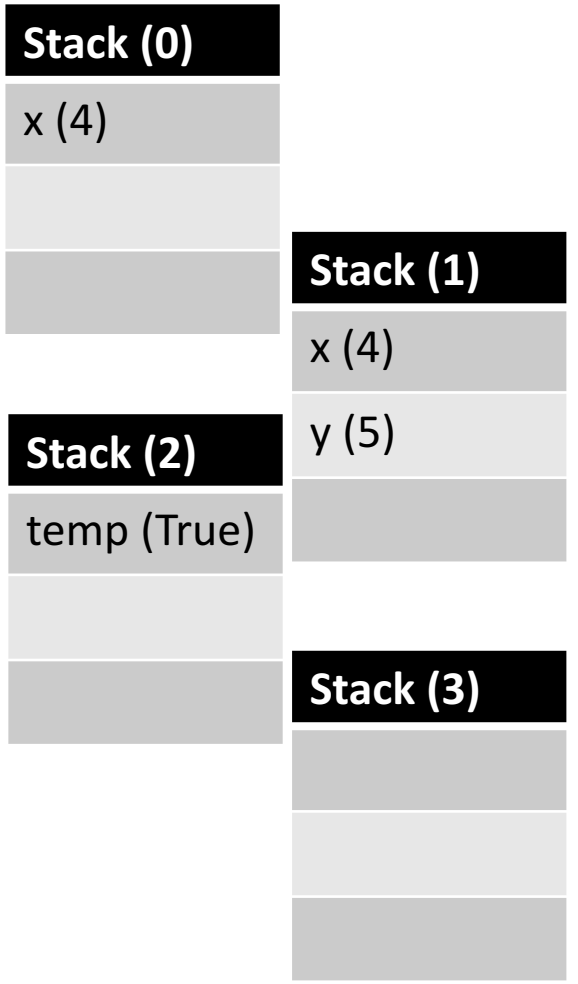
Here is a more complicated example to illustrate *branching* instructions – any operation (such as *if*, *while*, *for*, or *switch*) that may cause execution to take more than one path.

Byte code example 2

```
(
{
  var x = 4, y = 5;
  if(x < y) {
    x = y;
  } {
    x = y.neg;
  };
  x;
}.def.dumpByteCodes
)
```

BYTECODES: (21)

```
 0  30          PushTempZeroVar 'x'
 1  31          PushTempZeroVar 'y'
 2  E8          SendSpecialBinaryArithMsg '<'
 3  F8 00 07    JumpIfFalse 7  (13)
 6  31          PushTempZeroVar 'y'
 7  08 00 00    StoreTempVarX 'x'
10  FC 00 05    JumpFwd 5  (18)
13  31          PushTempZeroVar 'y'
14  D0          SendSpecialUnaryArithMsg 'neg'
15  08 00 00    StoreTempVarX 'x'
18  F0          Drop
19  30          PushTempZeroVar 'x'
20  F2          BlockReturn
```



Again, the two variables x and y are pushed onto the stack; then the binary comparison instruction '<' consumes them and replaces them with the result (4 < 5 => true).

JumpIfFalse is a special instruction that will cause execution to jump somewhere other than the next instruction if the top of the stack is the value False. In this case, it will jump 7 *bytes* (not instructions!) ahead from the index of the next instruction; the target is in parentheses (13). Note that JumpIfFalse consumes the top of the stack.

Byte code example 2

```
(
{
  var x = 4, y = 5;
  if(x < y) {
    x = y;
  } {
    x = y.neg;
  };
  x;
}.def.dumpByteCodes
)
```

BYTECODES: (21)

```
 0  30          PushTempZeroVar 'x'
 1  31          PushTempZeroVar 'y'
 2  E8          SendSpecialBinaryArithMsg '<'
 3  F8 00 07    JumpIfFalse 7  (13)
 6  31          PushTempZeroVar 'y'
 7  08 00 00    StoreTempVarX 'x'
10  FC 00 05    JumpFwd 5  (18)
13  31          PushTempZeroVar 'y'
14  D0          SendSpecialUnaryArithMsg 'neg'
15  08 00 00    StoreTempVarX 'x'
18  F0          Drop
19  30          PushTempZeroVar 'x'
20  F2          BlockReturn
```

Stack (6)
y (5)

Stack (7)
y (5)

If the top of the stack was True in the last step, execution simply “falls through” to the next instruction. In the true branch of the *if*, the value in y is stored to x. However, note the different instruction name – “StoreTempVarX”, not ”StoreTempVar”. This is a different, “extended” instruction that stores the value but also leaves it on the top of the stack. We would expect this *if* branch to return the value of ‘x’ – but since ‘y’ has just been assigned to it, both variables hold exactly the same value; therefore, the interpreter avoids unnecessarily modifying the stack and returns ‘y’ instead; a very nice optimization!

Byte code example 2

```
(
{
  var x = 4, y = 5;
  if(x < y) {
    x = y;
  } {
    x = y.neg;
  };
  x;
}.def.dumpByteCodes
)
```

BYTECODES: (21)

```
 0  30          PushTempZeroVar 'x'
 1  31          PushTempZeroVar 'y'
 2  E8          SendSpecialBinaryArithMsg '<'
 3  F8 00 07    JumpIfFalse 7  (13)
 6  31          PushTempZeroVar 'y'
 7  08 00 00    StoreTempVarX 'x'
10  FC 00 05    JumpFwd 5  (18)
13  31          PushTempZeroVar 'y'
14  D0          SendSpecialUnaryArithMsg 'neg'
15  08 00 00    StoreTempVarX 'x'
18  F0          Drop
19  30          PushTempZeroVar 'x'
20  F2          BlockReturn
```

Stack (10)
y (5)

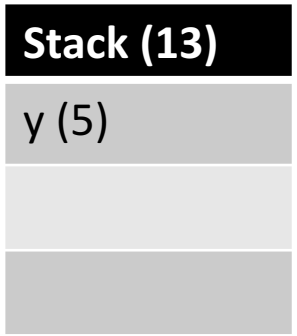
At the end of this branch, the instruction JumpFwd causes execution to immediately jump 5 *bytes* (not instructions) ahead of the next instruction's index. In this case, the target is index 18, the end of the entire *if* statement. The value of 'y' is still on the stack here.

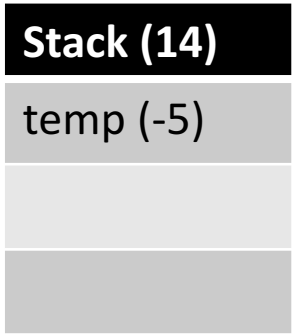
Byte code example 2

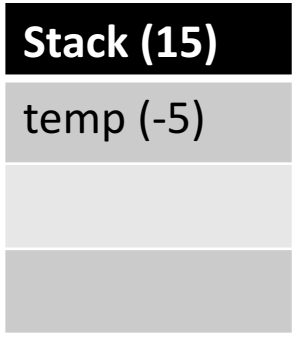
```
(
{
  var x = 4, y = 5;
  if(x < y) {
    x = y;
  } {
    x = y.neg;
  };
  x;
}.def.dumpByteCodes
)
```

BYTECODES: (21)

```
 0  30          PushTempZeroVar 'x'
 1  31          PushTempZeroVar 'y'
 2  E8          SendSpecialBinaryArithMsg '<'
 3  F8 00 07    JumpIfFalse 7  (13)
 6  31          PushTempZeroVar 'y'
 7  08 00 00    StoreTempVarX 'x'
10  FC 00 05    JumpFwd 5  (18)
13  31          PushTempZeroVar 'y'
14  D0          SendSpecialUnaryArithMsg 'neg'
15  08 00 00    StoreTempVarX 'x'
18  F0          Drop
19  30          PushTempZeroVar 'x'
20  F2          BlockReturn
```







In the false branch of the *if*, the value of y is pushed to the stack, negated via the “neg” message, and stored in x. Since the next instruction is the rendezvous point for both branches, execution is allowed to fall through to the next instruction.

The previously mentioned “StoreTempVarX” is also used here to store into x, so the temporary result of ’neg’ is still on the stack at the end of this branch.

Byte code example 2

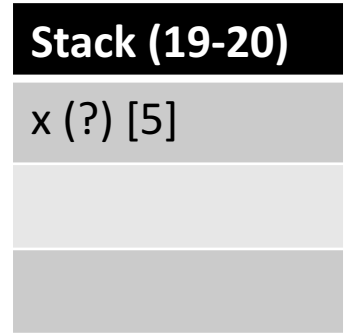
```
(
{
  var x = 4, y = 5;
  if(x < y) {
    x = y;
  } {
    x = y.neg;
  };
  x;
}.def.dumpByteCodes
)
```

BYTECODES: (21)

```
 0  30          PushTempZeroVar 'x'
 1  31          PushTempZeroVar 'y'
 2  E8          SendSpecialBinaryArithMsg '<'
 3  F8 00 07    JumpIfFalse 7  (13)
 6  31          PushTempZeroVar 'y'
 7  08 00 00    StoreTempVarX 'x'
10  FC 00 05    JumpFwd 5  (18)
13  31          PushTempZeroVar 'y'
14  D0          SendSpecialUnaryArithMsg 'neg'
15  08 00 00    StoreTempVarX 'x'
18  F0          Drop
19  30          PushTempZeroVar 'x'
20  F2          BlockReturn
```

Stack (18)

Stack (19-20)



At the end of the *if* statement, the result value is unused. We're not at the end of the function yet, so we need to discard this value. This is what Drop does: it simply pops whatever is on the top of the stack and does nothing with it.

Finally, the value of x is pushed back onto the stack; depending on which branch was taken it may have one of two values. (Although in fact, we can use our powers of human deduction to see that it must be 5!).

Byte code example 2

```
(
{
  var x = 4, y = 5;
  if(x < y) {
    x = y;
  } {
    x = y.neg;
  };
  x;
}.def.dumpByteCodes
)
```

BYTECODES: (21)

0	30	PushTempZeroVar 'x‘
1	31	PushTempZeroVar 'y‘
2	E8	SendSpecialBinaryArithMsg '<‘
3	F8 00 07	JumpIfFalse 7 (13)
6	31	PushTempZeroVar 'y‘
7	08 00 00	StoreTempVarX 'x‘
10	FC 00 05	JumpFwd 5 (18)
13	31	PushTempZeroVar 'y'
14	D0	SendSpecialUnaryArithMsg 'neg'
15	08 00 00	StoreTempVarX 'x'
18	F0	Drop
19	30	PushTempZeroVar 'x'
20	F2	BlockReturn

Note: if this was C code, it would be optimized to: `return 5;`

Thank you!