Compilers and computer architecture

11. Code generation for stack machines

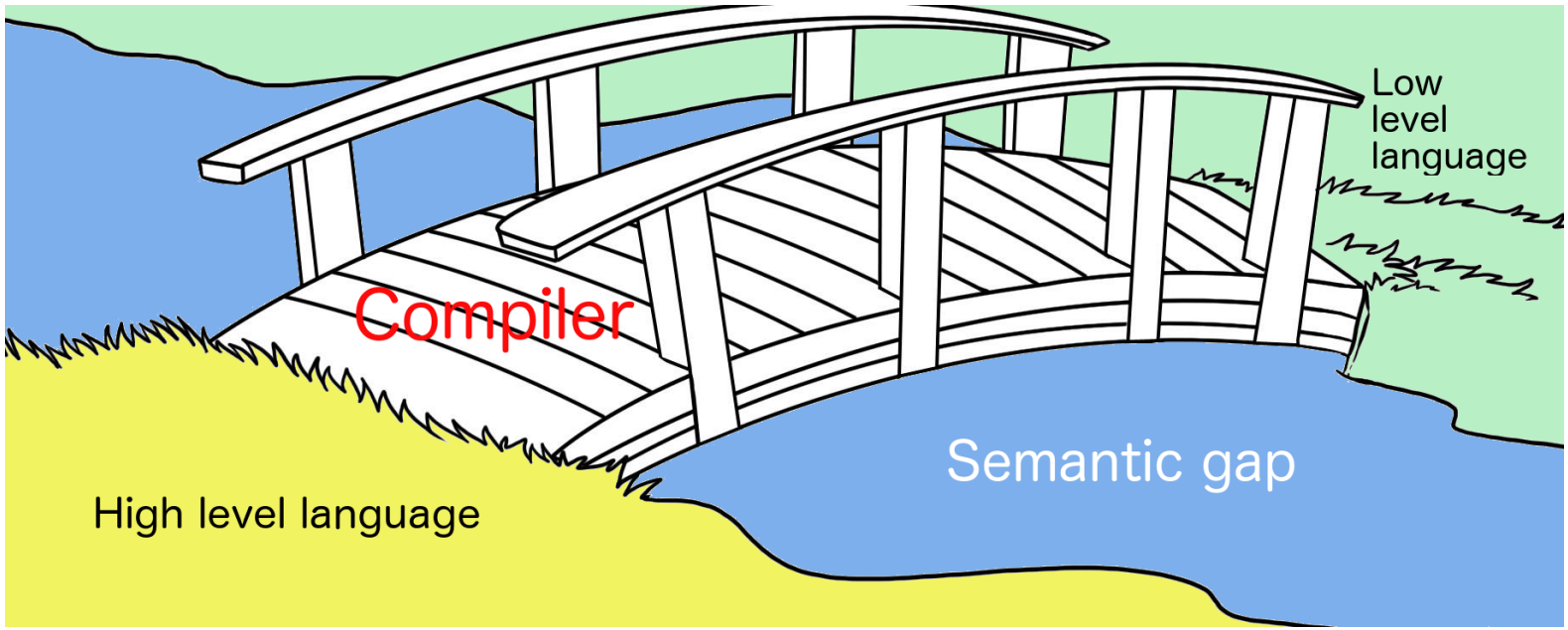
Hsi-Ming Ho

(slides by Niel de Beaudrap)

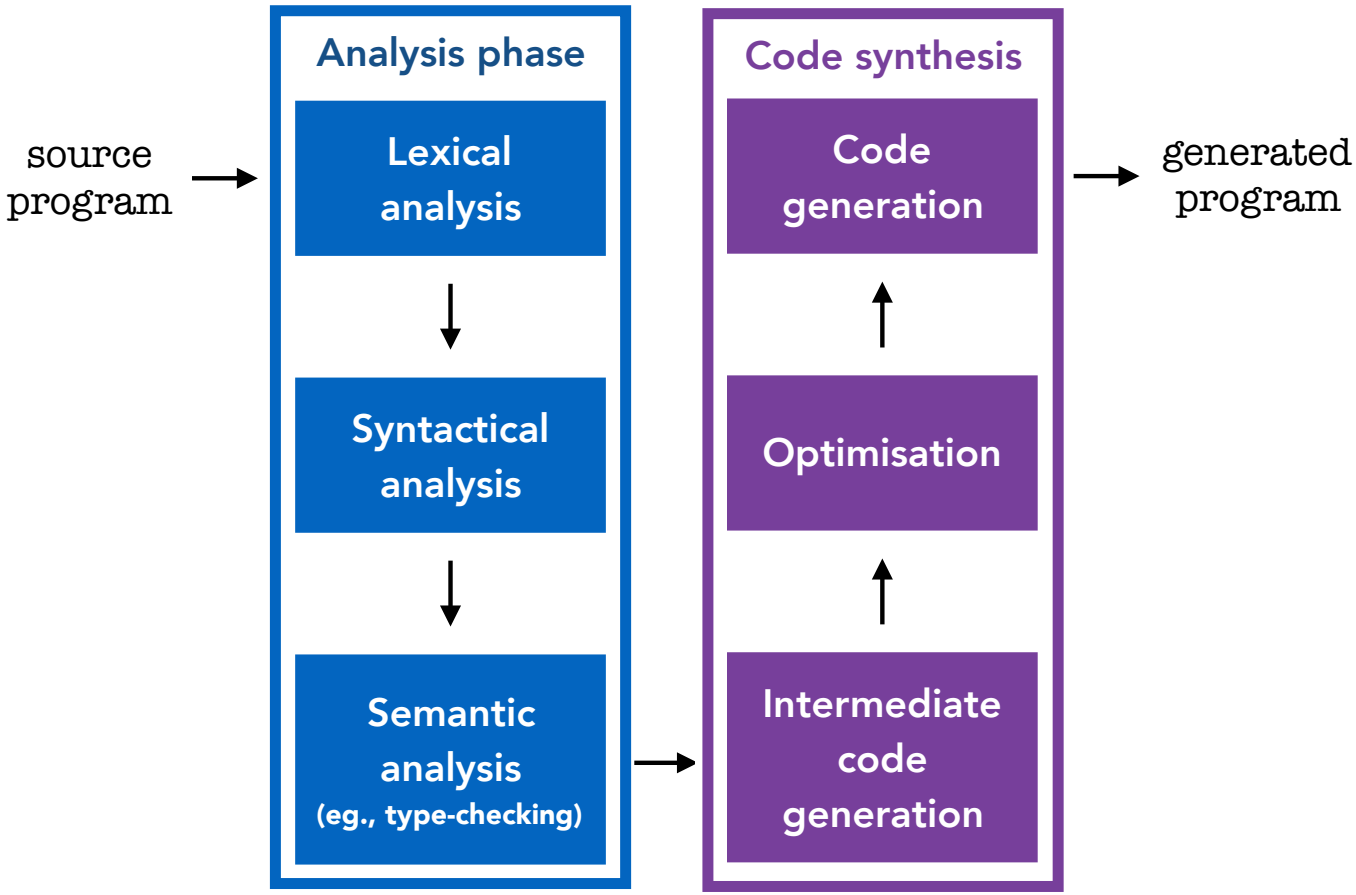
Email: **hsi-ming.ho@sussex.ac.uk**
Office hours: Online, by email appointment

Autumn 2022

Recall the function of compilers

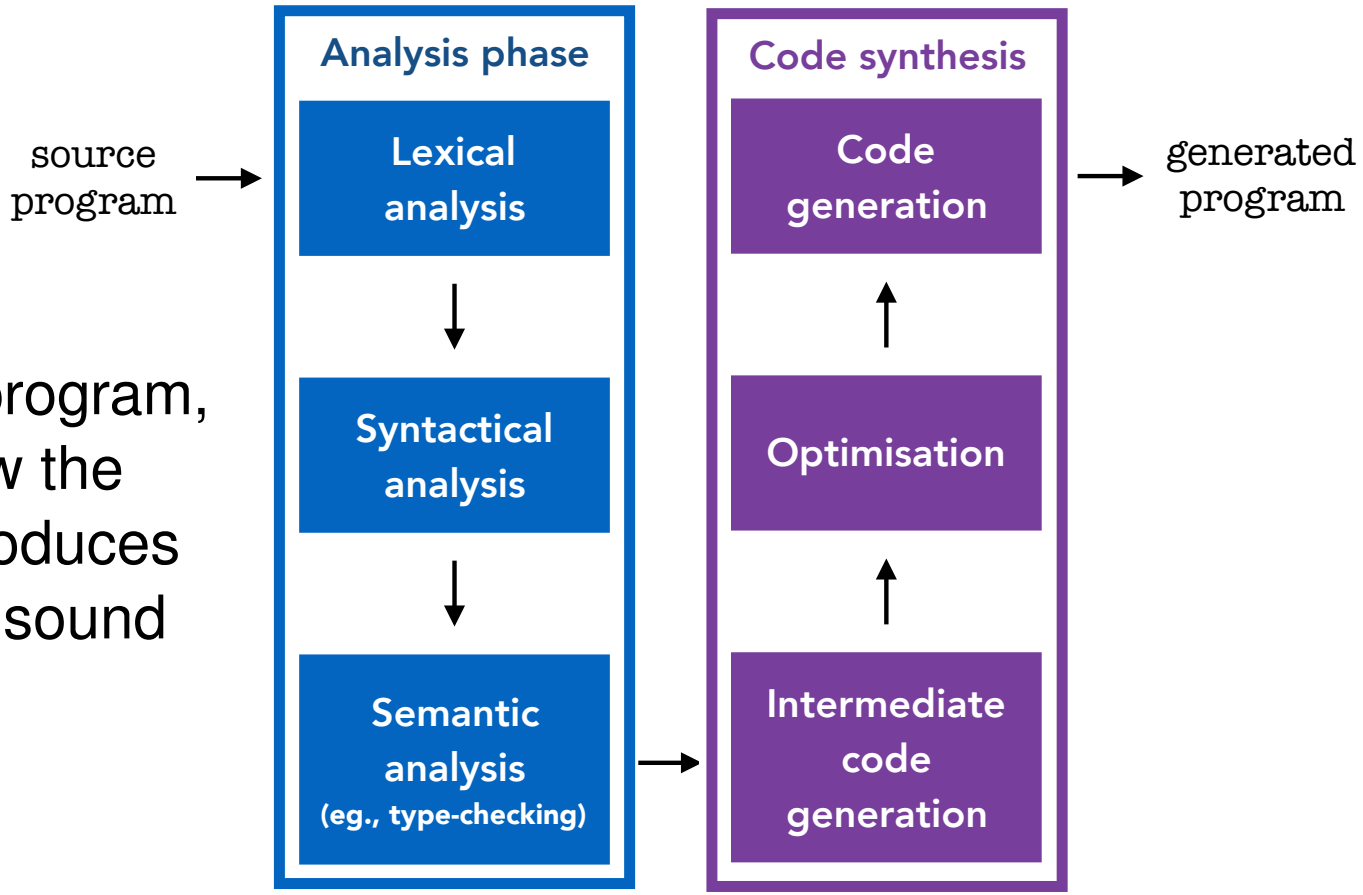


Where we are

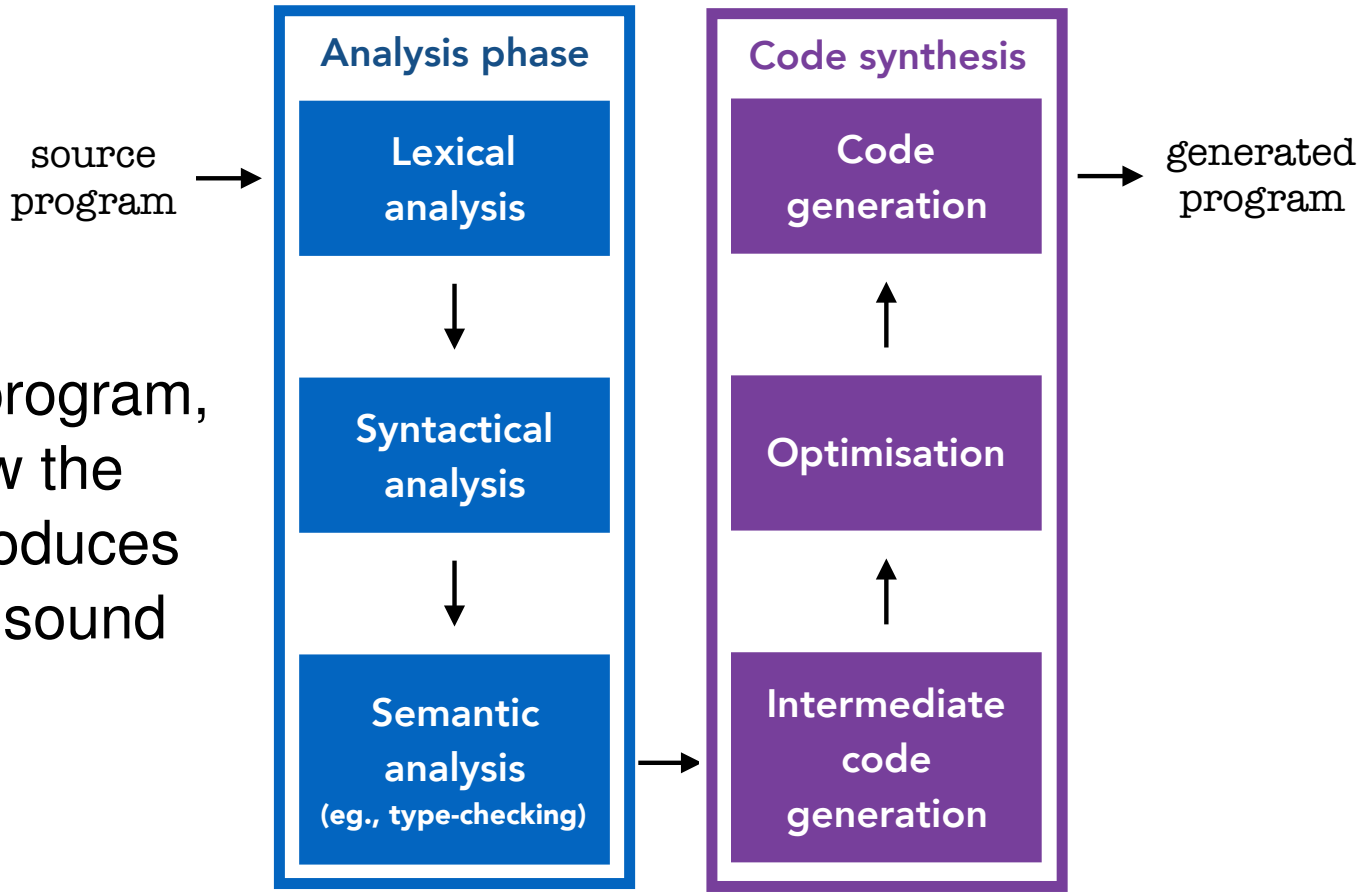

source program
Analysis phase
Lexical analysis
Syntactical analysis
Semantic analysis
(eg., type-checking)
Code synthesis
Code generation
Optimisation
Intermediate code generation
generated program

Where we are

From the source program, we've outlined how the analysis phase produces a syntax tree with sound semantics.



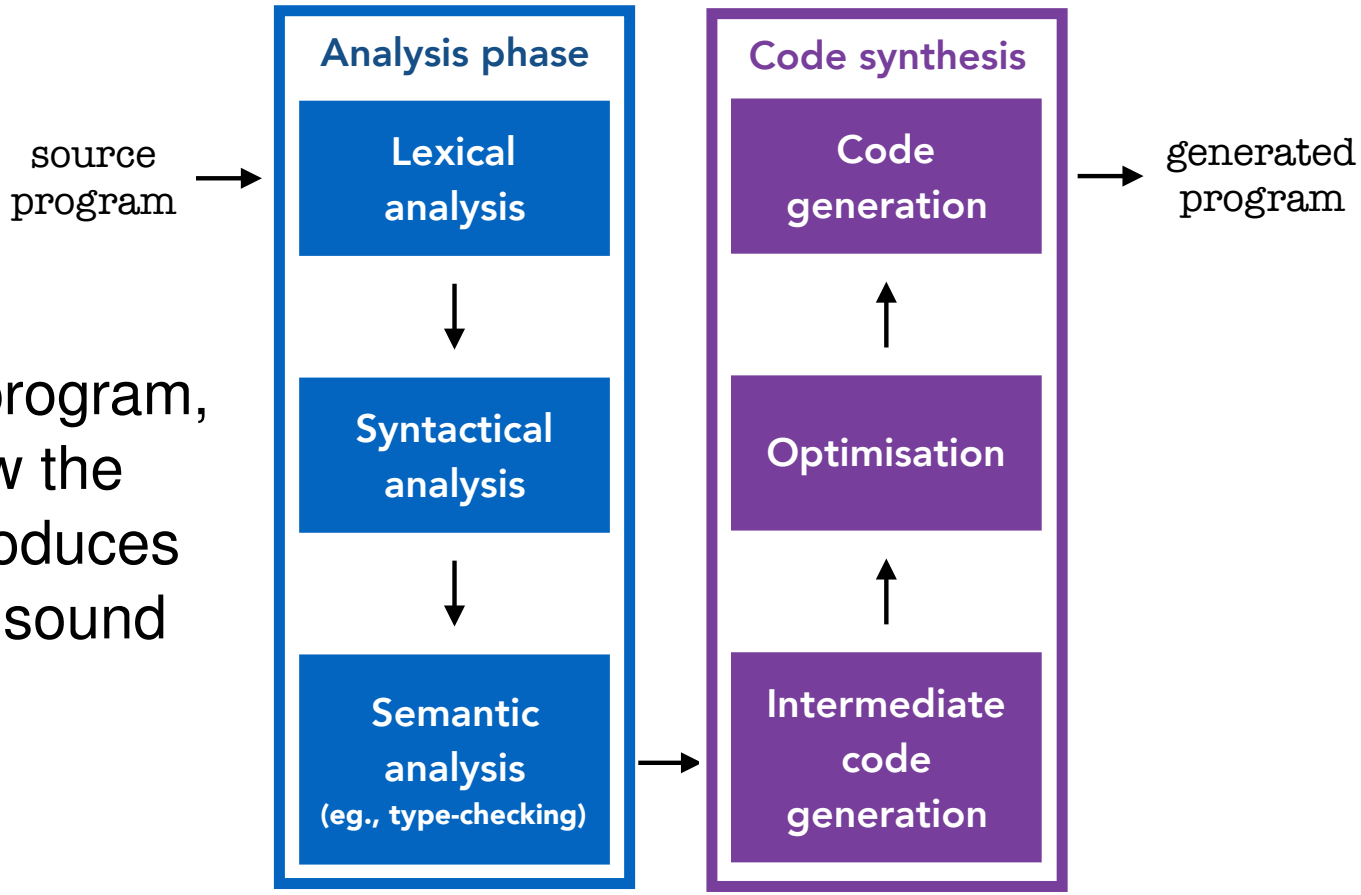
Where we are



From the source program, we've outlined how the analysis phase produces a syntax tree with sound semantics.

On the other end, we've seen (briefly) how to make use of memory and realise things like procedure calls in low-level languages.
In particular, activation records enable the possibility of *recursive* procedure calls (more details to follow later).

Where we are



From the source program, we've outlined how the analysis phase produces a syntax tree with sound semantics.

On the other end, we've seen (briefly) how to make use of memory and realise things like procedure calls in low-level languages. In particular, activation records enable the possibility of *recursive* procedure calls (more details to follow later).

Now: how do we actually go from one to the other?

Code generator input / output

A simple code generator:

- Takes as input: a syntax tree, and possibly a symbol table
- Produces as output: assembly code
  (*e.g.*, represented as a list of instructions)

Code generator input / output

A simple code generator:

- Takes as input: a syntax tree, and possibly a symbol table
- Produces as output: assembly code
  (*e.g.*, represented as a list of instructions)

```
class Tree { ... }

class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

Code generator input / output

A simple code generator:

- Takes as input: a syntax tree, and possibly a symbol table
- Produces as output: assembly code
  (*e.g.*, represented as a list of instructions)

```
class Tree { ... }

class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

The recursive structure of trees is as helpful for code generation just as for semantic analysis.

Code generator input / output

A simple code generator:

- Takes as input: a syntax tree, and possibly a symbol table
- Produces as output: assembly code
  (*e.g.*, represented as a list of instructions)

```
class Tree { ... }

class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

The recursive structure of trees is as helpful for code generation just as for semantic analysis.

If the source and target languages fit into the same paradigm (*e.g.*, both imperative), then code generation can actually be quite straightforward! (*cf.* syntax-directed translation).

Code generator input / output

A simple code generator:

- Takes as input: a syntax tree, and possibly a symbol table
- Produces as output: assembly code
  (*e.g.*, represented as a list of instructions)

```
class Tree { ... }

class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

The recursive structure of trees is as helpful for code generation just as for semantic analysis.

If the source and target languages fit into the same paradigm (*e.g.*, both imperative), then code generation can actually be quite straightforward! (*cf.* syntax-directed translation).

It is helpful to consider code generation for different types of **abstract** (hypothetical) low-level machines!

Compilation target #1: a simple stack machine

The stack machine has the following resources:

Compilation target #1: a simple stack machine

The stack machine has the following resources:

- Main memory, with integer addresses

Compilation target #1: a simple stack machine

The stack machine has the following resources:

- Main memory, with integer addresses
- A **program counter** (PC)
  — pointing to the command to be executed *next*.

Compilation target #1: a simple stack machine

The stack machine has the following resources:

- Main memory, with integer addresses
- A **program counter** (PC)
  — pointing to the command to be executed *next*.
- A **current instruction register** (IR/CI)
  — holding the *actual instruction* currently being executed.
  — this is used internally and usually not visible.

Compilation target #1: a simple stack machine

The stack machine has the following resources:

- Main memory, with integer addresses
- A **program counter** (PC)
  — pointing to the command to be executed *next*.
- A **current instruction register** (IR/CI)
  — holding the *actual instruction* currently being executed.
  — this is used internally and usually not visible.
- A **stack-pointer** (SP)
  — pointing into memory, helps manage data storage
  for *e.g.* evaluation of expressions

Compilation target #1: a simple stack machine

The stack machine has the following resources:

- Main memory, with integer addresses
- A **program counter** (PC)
  — pointing to the command to be executed *next*.
- A **current instruction register** (IR/CI)
  — holding the *actual instruction* currently being executed.
  — this is used internally and usually not visible.
- A **stack-pointer** (SP)
  — pointing into memory, helps manage data storage
  for *e.g.* evaluation of expressions
- A **frame-pointer** (FP)
  — pointing into memory, helps manage data storage
  for *e.g.* procedure calls / recursive algorithms

Advantages of stack machines

- **Simple computer model**
  - Physical CPUs are easier to manufacture (and cheaper)
  - Virtual machines (*e.g.*, the JVM and WebAssembly) are easier to realise as software

Advantages of stack machines

- **Simple computer model**
  - Physical CPUs are easier to manufacture (and cheaper)
  - Virtual machines (*e.g.*, the JVM and WebAssembly) are easier to realise as software
- **Simple compilers**: code generation for stack machines is much simpler than for register machines
  - Specifically, register allocation is not an issue.

Advantages of stack machines

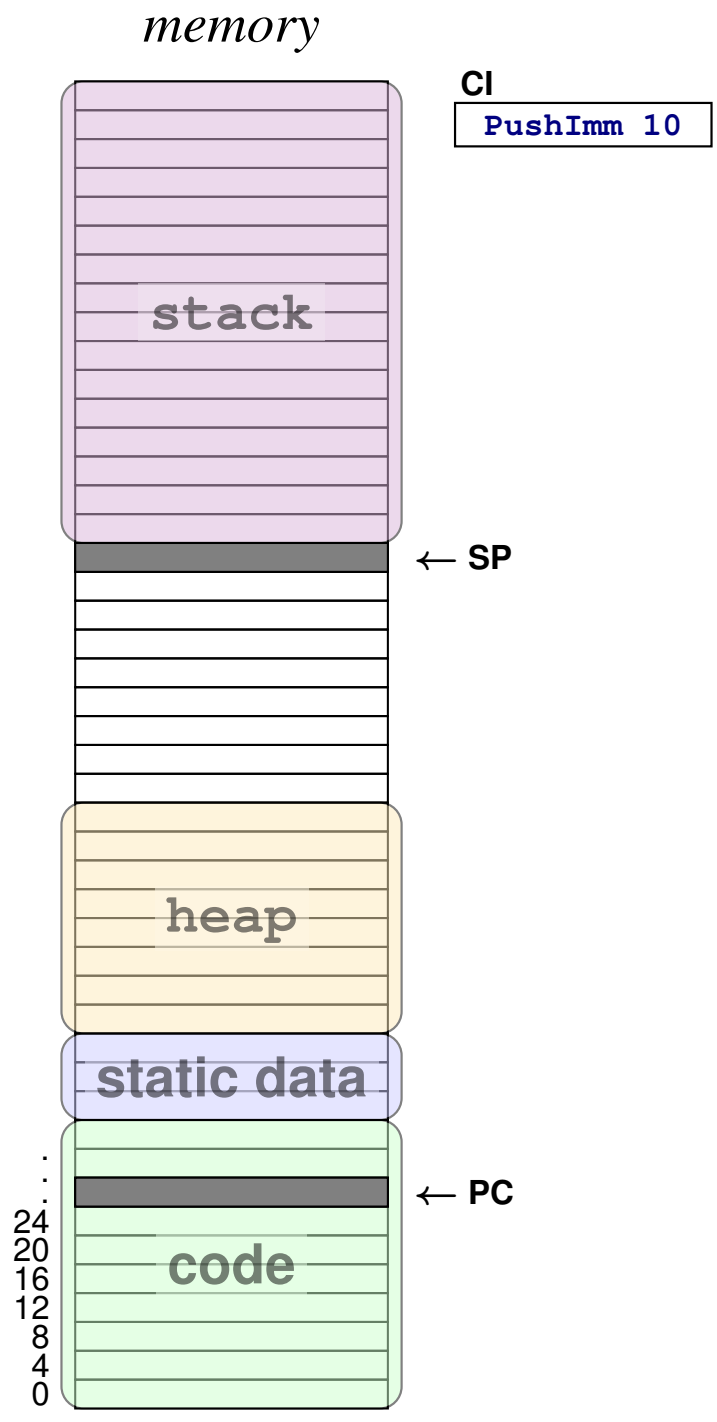
- **Simple computer model**
  - Physical CPUs are easier to manufacture (and cheaper)
  - Virtual machines (*e.g.*, the JVM and WebAssembly) are easier to realise as software
- **Simple compilers**: code generation for stack machines is much simpler than for register machines
  - Specifically, register allocation is not an issue.
- **Compact object code**, which saves memory.
  - Stack machine assembly commands have at most one argument — so take less memory to describe

Advantages of stack machines

- **Simple computer model**
  - Physical CPUs are easier to manufacture (and cheaper)
  - Virtual machines (*e.g.*, the JVM and WebAssembly) are easier to realise as software
- **Simple compilers**: code generation for stack machines is much simpler than for register machines
  - Specifically, register allocation is not an issue.
- **Compact object code**, which saves memory.
  - Stack machine assembly commands have at most one argument — so take less memory to describe

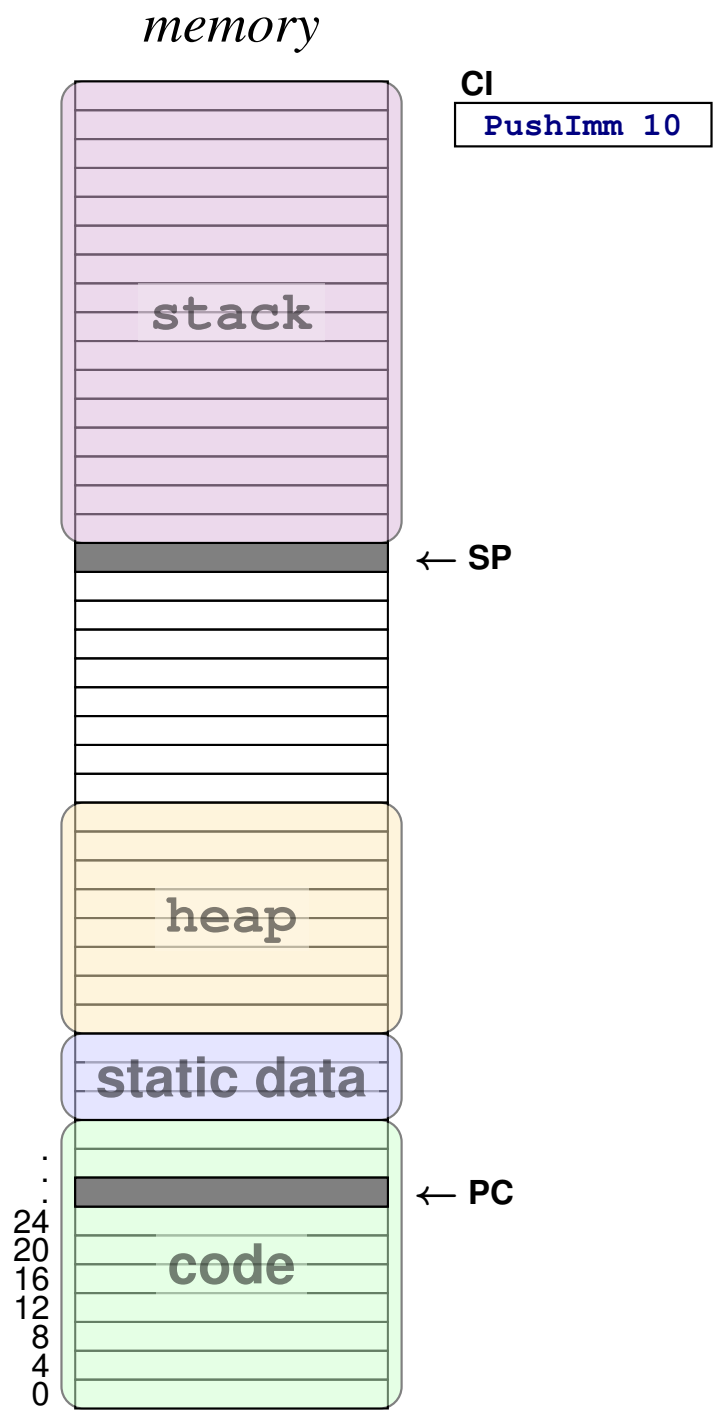
Physical stack machines have disadvantages, mainly that they are **slow** — memory access has much higher latency.

Sketch: the stack machine



Sketch: the stack machine

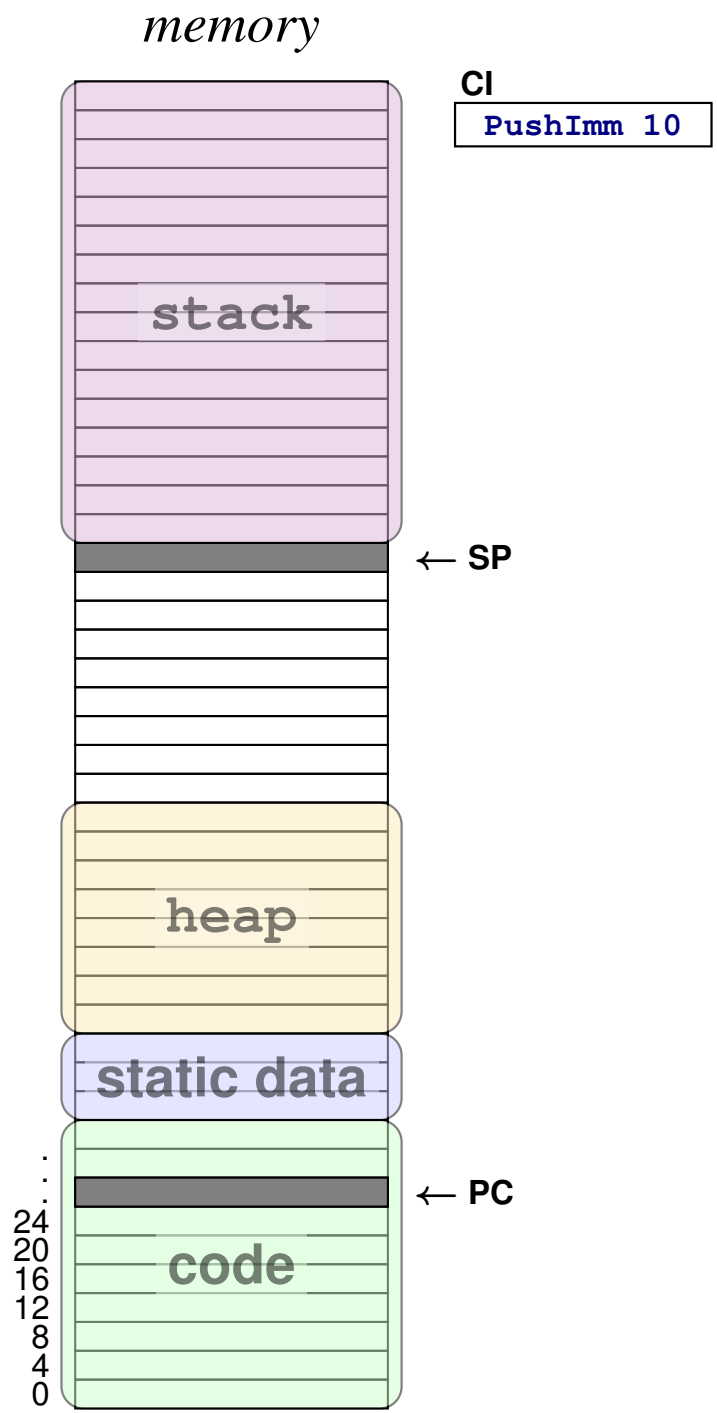
- **Code:** fixed size.
- **Static data:** fixed size, contains static variables.
- **Heap:** dynamic, useful to store persistent data structures
- **Stack:** dynamic, grows and shrinks with movement of **SP** —
  - points to space in front of stack



Sketch: the stack machine

- **Code:** fixed size.
- **Static data:** fixed size, contains static variables.
- **Heap:** dynamic, useful to store persistent data structures
- **Stack:** dynamic, grows and shrinks with movement of **SP** —
  - points to space in front of stack

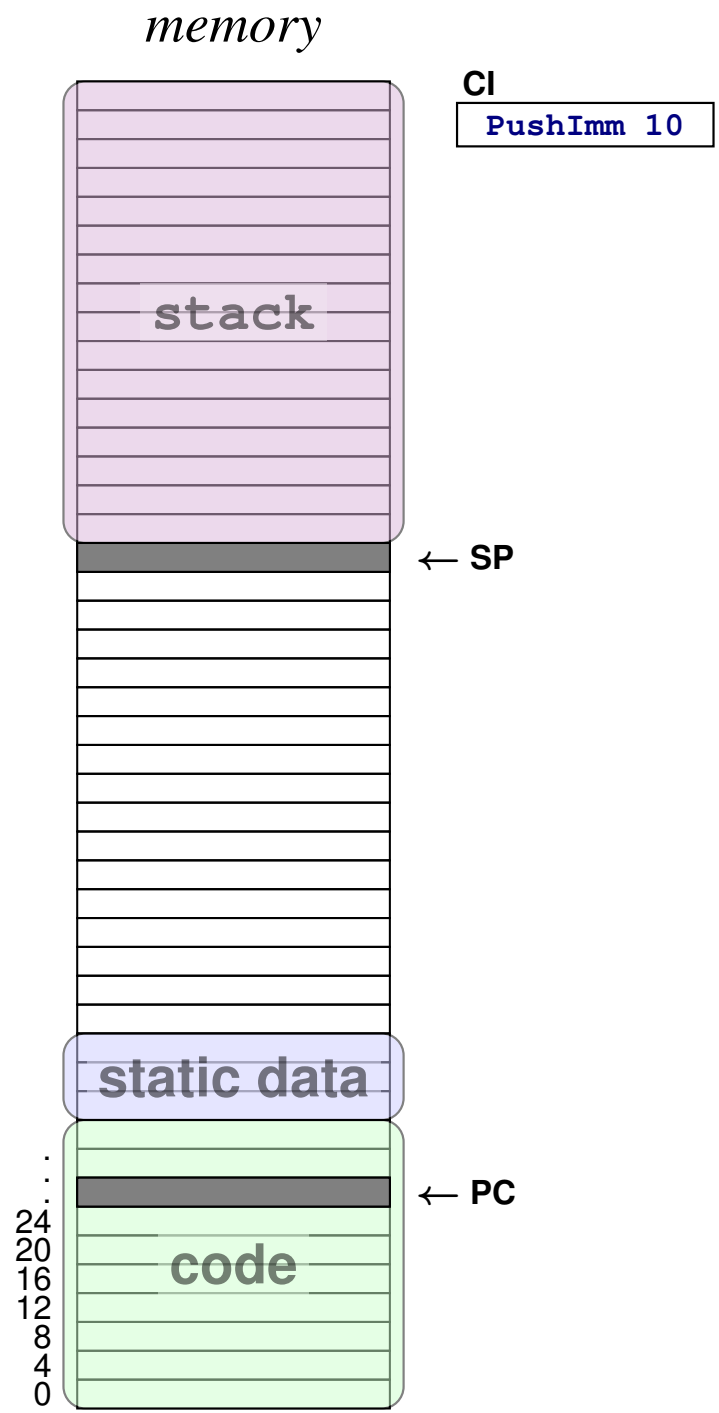
N.B. Like many other architectures, stack machines do not take pre-cautions to prevent “stack overflows”.



Sketch: the stack machine

- **Code:** fixed size.
- **Static data:** fixed size, contains static variables.
- **Heap:** dynamic, useful to store persistent data structures
  — may be absent in some cases
- **Stack:** dynamic, grows and shrinks with movement of **SP** —
  - points to space in front of stack

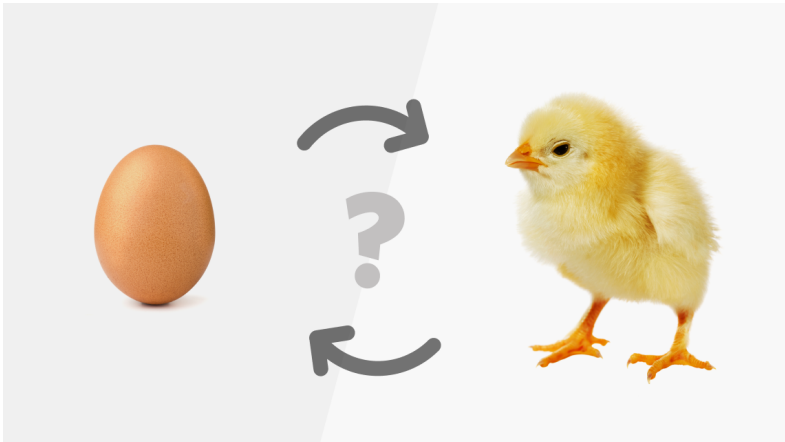
  N.B. Like many other architectures, stack machines do not take pre-cautions to prevent “stack overflows”.



Data access in stack machines

Recall: the computer can only work with

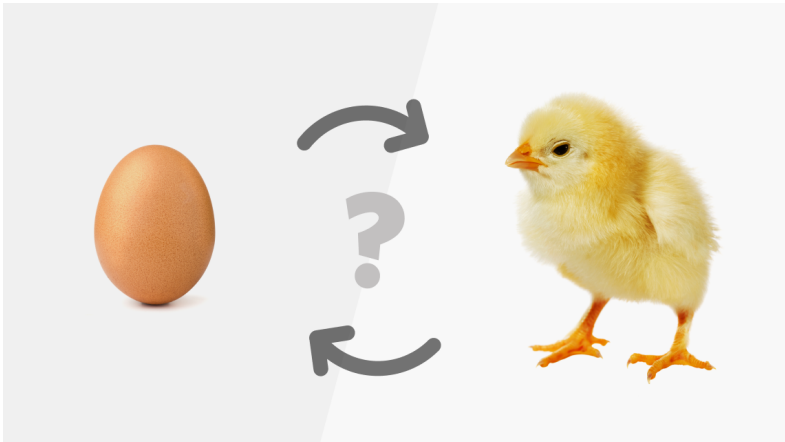
- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers



Data access in stack machines

Recall: the computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers



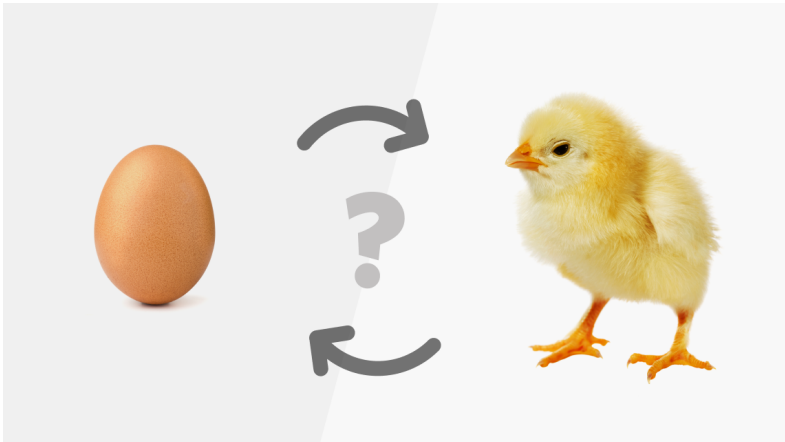
So — static addresses, and locations stored in registers.

(and in memory *pointed at* by registers, and . . .)

Data access in stack machines

Recall: the computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers



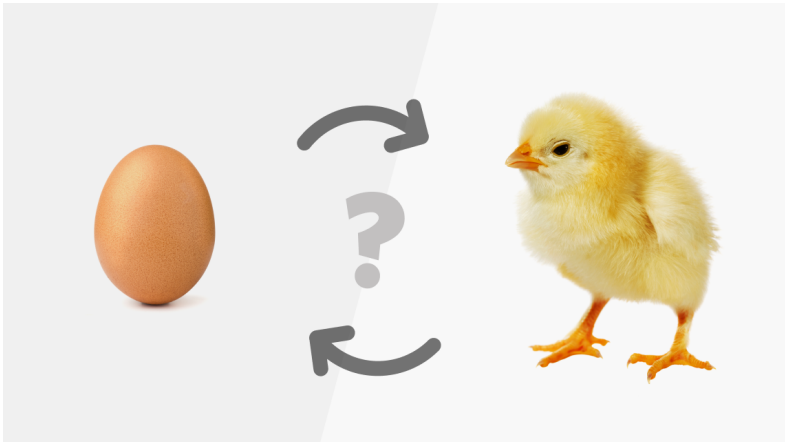
So — static addresses, and locations stored in registers.
(and in memory *pointed at* by registers, and . . .)

- **CI**, **PC**: not for storing pointers to data
  (usually not even directly accessible by the programmer)

Data access in stack machines

Recall: the computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers



So — static addresses, and locations stored in registers.
(and in memory *pointed at* by registers, and . . .)

- **CI**, **PC**: not for storing pointers to data
  (usually not even directly accessible by the programmer)
- **SP**, **FP**: obviously useful for data access.

Approaching assembly on a stack machine

The only way is stack!

Approaching assembly on a stack machine

The only way is stack!

$\Longrightarrow$ the assembly language, if it's going to do something useful, will make use of **SP** (and possibly **FP**).

Approaching assembly on a stack machine

The only way is stack!

$\Longrightarrow$ the assembly language, if it's going to do something useful, will make use of **SP** (and possibly **FP**).

A stack machine is one in which **nearly every** operation involves the stack

Approaching assembly on a stack machine

The only way is stack!

$\Longrightarrow$ the assembly language, if it's going to do something useful, will make use of **SP** (and possibly **FP**).

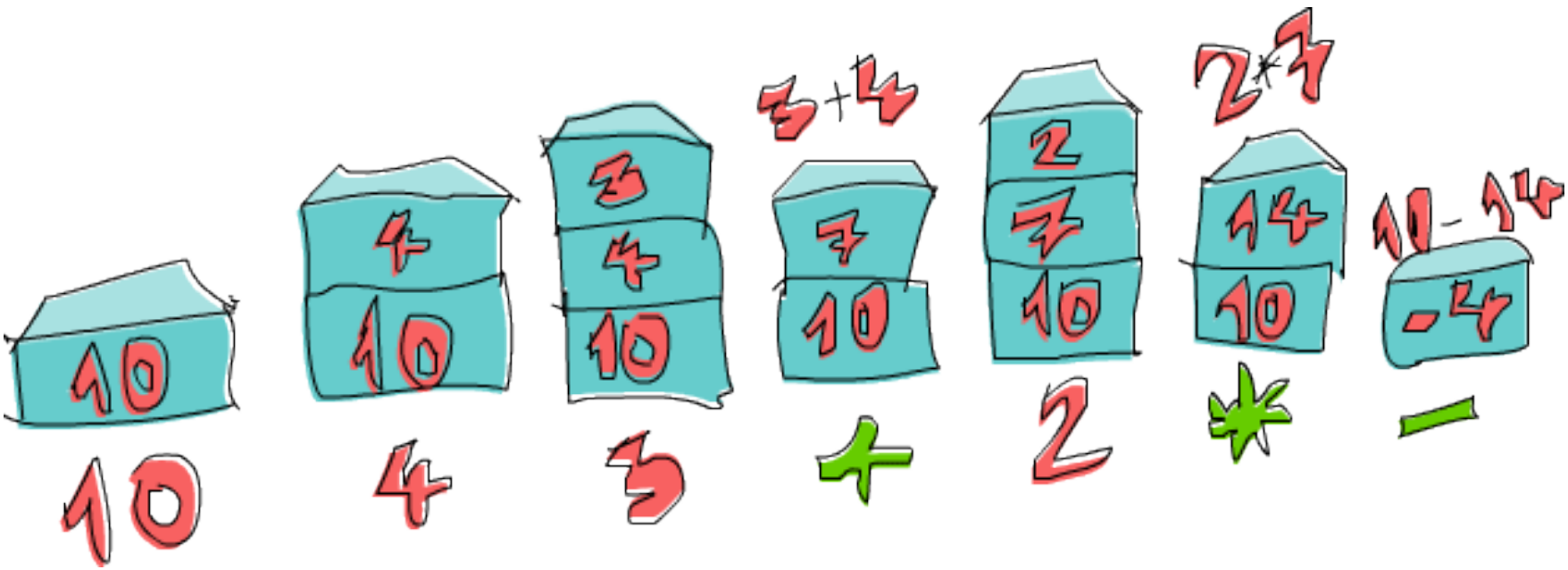
A stack machine is one in which **nearly every** operation involves the stack — because it is the one place that it can reliably access or store data to compute with.

Approaching assembly on a stack machine

The only way is stack!

$\Longrightarrow$ the assembly language, if it's going to do something useful, will make use of **SP** (and possibly **FP**).

A stack machine is one in which **nearly every** operation involves the stack — because it is the one place that it can reliably access or store data to compute with.



Commands of the stack machine

`PushImm n` Pushes the number **`n`** onto stack

Commands of the stack machine

`PushImm n`	Pushes the number `n` onto stack
`PushRel n`	Pushes the content of memory at address **FP** $+$ `n` onto stack

Commands of the stack machine

`PushImm n`	Pushes the number **`n`** onto stack
`PushRel n`	Pushes the content of memory at address **FP** $+$ **`n`** onto stack
`PopRel n`	Pops the top element off the stack and stores it at address **FP** $+$ **`n`**

Commands of the stack machine

`PushImm n`	Pushes the number `n` onto stack
`PushRel n`	Pushes the content of memory at address **FP** $+$ `n` onto stack
`PopRel n`	Pops the top element off the stack and stores it at address **FP** $+$ `n`
`Reserve n`	Decreases **SP** by `n`

Commands of the stack machine

`PushImm n`	Pushes the number `n` onto stack
`PushRel n`	Pushes the content of memory at address **FP** $+$ `n` onto stack
`PopRel n`	Pops the top element off the stack and stores it at address **FP** $+$ `n`
`Reserve n`	Decreases **SP** by `n`
`Discard n`	Increases **SP** by `n`

Commands of the stack machine

`PushImm n`	Pushes the number `n` onto stack
`PushRel n`	Pushes the content of memory at address **FP** $+$ `n` onto stack
`PopRel n`	Pops the top element off the stack and stores it at address **FP** $+$ `n`
`Reserve n`	Decreases **SP** by `n`
`Discard n`	Increases **SP** by `n`
`SetFP`	Set **FP** to the value of **SP**

Commands of the stack machine

`PushImm n`	Pushes the number `n` onto stack
`PushRel n`	Pushes the content of memory at address **FP** $+$ `n` onto stack
`PopRel n`	Pops the top element off the stack and stores it at address **FP** $+$ `n`
`Reserve n`	Decreases **SP** by `n`
`Discard n`	Increases **SP** by `n`
`SetFP`	Set **FP** to the value of **SP**
`SaveFP`	Pushes the value of **FP** onto stack

Commands of the stack machine

`PushImm n`	Pushes the number `n` onto stack
`PushRel n`	Pushes the content of memory at address **FP** $+$ `n` onto stack
`PopRel n`	Pops the top element off the stack and stores it at address **FP** $+$ `n`
`Reserve n`	Decreases **SP** by `n`
`Discard n`	Increases **SP** by `n`
`SetFP`	Set **FP** to the value of **SP**
`SaveFP`	Pushes the value of **FP** onto stack
`RestoreFP`	Pops the top element off the stack and stores it in **FP**

Commands of the stack machine

`CompGT` — Pops top two elements off stack
— Pushes a 1 onto stack if first element was larger, and a 0 if not

Commands of the stack machine

`CompGT`	— Pops top two elements off stack — Pushes a 1 onto stack if first element was larger, and a 0 if not
`...`	(similar ones)

Commands of the stack machine

`CompGT`	— Pops top two elements off stack — Pushes a 1 onto stack if first element was larger, and a 0 if not
`...`	(similar ones)
`Invert`	— Pops the top elements off the stack — Pushes a 1 onto stack if it was zero, and a 0 if it was non-zero

Commands of the stack machine

`CompGT`	— Pops top two elements off stack — Pushes a 1 onto stack if first element was larger, and a 0 if not
`...`	(similar ones)
`Invert`	— Pops the top elements off the stack — Pushes a 1 onto stack if it was zero, and a 0 if it was non-zero
`LogicalAnd`	— Pops top two elements off the stack — Pushes a 1 onto stack if both elements were non-zero, and a 0 otherwise

Commands of the stack machine

`CompGT`	— Pops top two elements off stack — Pushes a 1 onto stack if first element was larger, and a 0 if not
`...`	(similar ones)
`Invert`	— Pops the top elements off the stack — Pushes a 1 onto stack if it was zero, and a 0 if it was non-zero
`LogicalAnd`	— Pops top two elements off the stack — Pushes a 1 onto stack if both elements were non-zero, and a 0 otherwise
`...`	(similar ones)
`Plus`	— Pops top two elements off stack — Pushes their sum onto the stack

Commands of the stack machine

`CompGT`	— Pops top two elements off stack — Pushes a 1 onto stack if first element was larger, and a 0 if not
`...`	(similar ones)
`Invert`	— Pops the top elements off the stack — Pushes a 1 onto stack if it was zero, and a 0 if it was non-zero
`LogicalAnd`	— Pops top two elements off the stack — Pushes a 1 onto stack if both elements were non-zero, and a 0 otherwise
`...`	(similar ones)
`Plus`	— Pops top two elements off stack — Pushes their sum onto the stack
`...`	(similar ones)

Commands of the stack machine

`Jump a`	Jumps to address `a` — *i.e.*, write to **PC**

Commands of the stack machine

`Jump a`	Jumps to address `a` — *i.e.*, write to **PC**
`JumpTrue a`	— Pops top element off stack — If it is non-zero, jumps to address `a`

Commands of the stack machine

`Jump a`	Jumps to address `a` — *i.e.*, write to **PC**
`JumpTrue a`	— Pops top element off stack — If it is non-zero, jumps to address `a`
`Invoke a`	— Pushes the return address onto the stack — Jumps to address `a`

Commands of the stack machine

`Jump a`	Jumps to address `a` — *i.e.*, write to **PC**
`JumpTrue a`	— Pops top element off stack — If it is non-zero, jumps to address `a`
`Invoke a`	— Pushes the return address onto the stack — Jumps to address `a`
`Return`	— Pops top element (an address) off the stack — Jumps to that address

Commands of the stack machine

`Jump a`	Jumps to address `a` — *i.e.*, write to **PC**
`JumpTrue a`	— Pops top element off stack — If it is non-zero, jumps to address `a`
`Invoke a`	— Pushes the return address onto the stack — Jumps to address `a`
`Return`	— Pops top element (an address) off the stack — Jumps to that address
`Nop`	Does nothing

Commands of the stack machine

`Jump a`	Jumps to address `a` — *i.e.*, write to **PC**
`JumpTrue a`	— Pops top element off stack — If it is non-zero, jumps to address `a`
`Invoke a`	— Pushes the return address onto the stack — Jumps to address `a`
`Return`	— Pops top element (an address) off the stack — Jumps to that address
`Nop`	Does nothing
`Print`	Pops the top element off the stack and prints it (as an integer) on the screen

Commands of the stack machine

`Jump a`	Jumps to address `a` — *i.e.*, write to **PC**
`JumpTrue a`	— Pops top element off stack — If it is non-zero, jumps to address `a`
`Invoke a`	— Pushes the return address onto the stack — Jumps to address `a`
`Return`	— Pops top element (an address) off the stack — Jumps to that address
`Nop`	Does nothing
`Print`	Pops the top element off the stack and prints it (as an integer) on the screen
`...`	(similar ones)

A typical assembly language program

```
start:
    PushRel 8
    PushRel 12
    Add
    PopRel -4

    PushRel -8
    PushImm 1
    Minus
    PopRel -8

    PushRel -8
    PushImm 0
    CompEq
    Invert

    JumpTrue start
```

A typical assembly language program

```
start:
    PushRel 8
    PushRel 12
    Add
    PopRel -4

    PushRel -8
    PushImm 1
    Minus
    PopRel -8

    PushRel -8
    PushImm 0
    CompEq
    Invert

    JumpTrue start
```

What does it do?

A typical assembly language program

```
start:  <------ symbolic address label
    PushRel 8
    PushRel 12
    Add
    PopRel -4

    PushRel -8
    PushImm 1
    Minus
    PopRel -8

    PushRel -8
    PushImm 0
    CompEq
    Invert

    JumpTrue start
```

What does it do?

A typical assembly language program

```
start:  <------ symbolic address label
    PushRel 8
    PushRel 12
    Add
    PopRel -4

    PushRel -8
    PushImm 1
    Minus
    PopRel -8

    PushRel -8
    PushImm 0
    CompEq
    Invert

    JumpTrue start
```

What does it do?

Symbolic labels like `start` appear in the compiler's output stream, but are not instructions. They are removed later — *e.g.*, at link time.

Stack machine assembly in memory

Consider a 32-bit machine (4-byte words).

Stack machine assembly in memory

Consider a 32-bit machine (4-byte words).

- There are 28 instructions
  (encodable in 5 bits; we allocate 8).

Stack machine assembly in memory

Consider a 32-bit machine (4-byte words).

- There are 28 instructions
  (encodable in 5 bits; we allocate 8).
- Use 24-bit “immediate” integer values
  — can be stored as part of the instruction.

Stack machine assembly in memory

Consider a 32-bit machine (4-byte words).

- There are 28 instructions
  (encodable in 5 bits; we allocate 8).
- Use 24-bit “immediate” integer values
  — can be stored as part of the instruction.

```
start:
    PushRel 8
    PushRel 12
    Add
    PopRel -4
    PushRel -8
    PushImm 1
    Minus
    PopRel -8
    PushRel -8
    PushImm 0
    CompEq
    Invert
    JumpTrue start
```

Stack machine assembly in memory

Consider a 32-bit machine (4-byte words).

- There are 28 instructions
  (encodable in 5 bits; we allocate 8).
- Use 24-bit “immediate” integer values
  — can be stored as part of the instruction.

```
start:
    PushRel 8
    PushRel 12
    Add
    PopRel -4
    PushRel -8
    PushImm 1
    Minus
    PopRel -8
    PushRel -8
    PushImm 0
    CompEq
    Invert
    JumpTrue start
```

If this program is loaded so that `start` is mapped to address 1024, the memory layout may resemble that on the right:

(Each command would itself be represented as a number.)

Stack machine assembly in memory

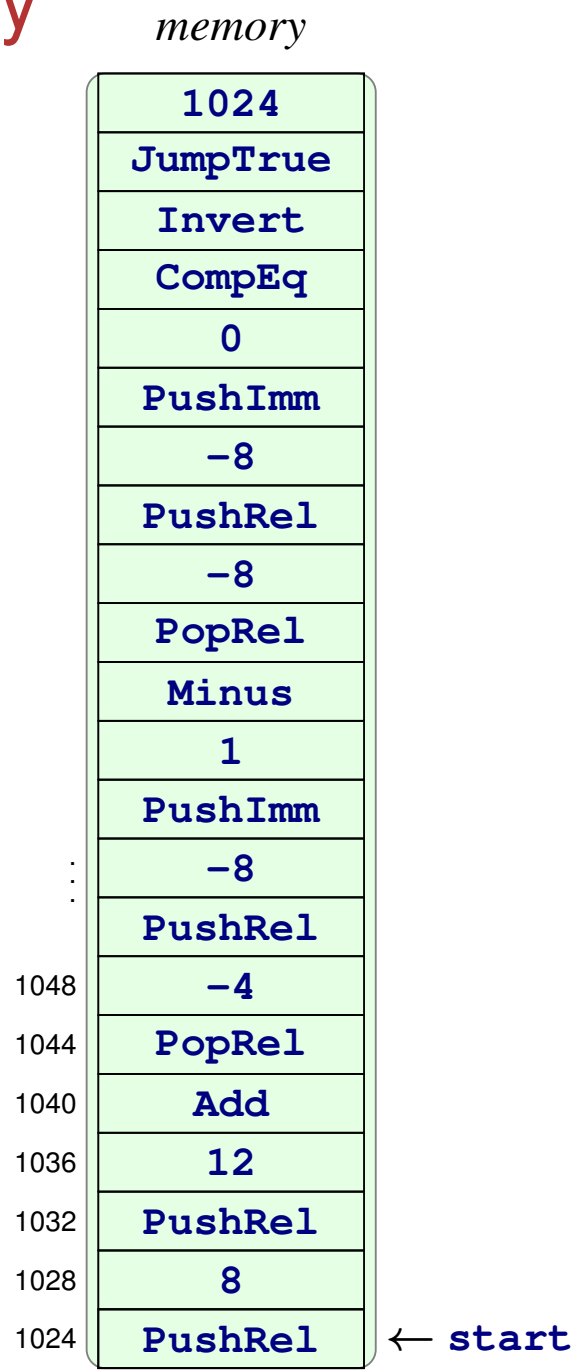
Consider a 32-bit machine (4-byte words).

- There are 28 instructions
  (encodable in 5 bits; we allocate 8).
- Use 24-bit “immediate” integer values
  — can be stored as part of the instruction.

```
start:
    PushRel 8
    PushRel 12
    Add
    PopRel -4
    PushRel -8
    PushImm 1
    Minus
    PopRel -8
    PushRel -8
    PushImm 0
    CompEq
    Invert
    JumpTrue start
```

If this program is loaded so that `start` is mapped to address 1024, the memory layout may resemble that on the right:

(Each command would itself be represented as a number.)



A moment of contemplation

You actually have essentially all the information you need to write a program to simulate a stack machine.

A moment of contemplation

You actually have essentially all the information you need to write a program to simulate a stack machine.

- Each register is some member variable of an object of class `StackMachine` (which you would define yourself). The stack machine's "memory" can be implemented as one long array of integers.

A moment of contemplation

You actually have essentially all the information you need to write a program to simulate a stack machine.

- Each register is some member variable of an object of class `StackMachine` (which you would define yourself). The stack machine's "memory" can be implemented as one long array of integers.
- Each machine instruction would be a method which affects the program counter, possibly the stack pointer, *etc.*

A moment of contemplation

You actually have essentially all the information you need to write a program to simulate a stack machine.

- Each register is some member variable of an object of class `StackMachine` (which you would define yourself). The stack machine's "memory" can be implemented as one long array of integers.
- Each machine instruction would be a method which affects the program counter, possibly the stack pointer, *etc.* You'd have to set some conventions:
  - Whether / how to map instructions to integers (opcodes)

A moment of contemplation

You actually have essentially all the information you need to write a program to simulate a stack machine.

- Each register is some member variable of an object of class `StackMachine` (which you would define yourself). The stack machine's "memory" can be implemented as one long array of integers.
- Each machine instruction would be a method which affects the program counter, possibly the stack pointer, *etc.* You'd have to set some conventions:
  - Whether / how to map instructions to integers (opcodes)
  - decide whether to include "immediates" in the encoding of the instructions, as suggested above (and as in RISC-V)

A moment of contemplation

You actually have essentially all the information you need to write a program to simulate a stack machine.

- Each register is some member variable of an object of class `StackMachine` (which you would define yourself). The stack machine's "memory" can be implemented as one long array of integers.
- Each machine instruction would be a method which affects the program counter, possibly the stack pointer, *etc.* You'd have to set some conventions:
  - Whether / how to map instructions to integers (opcodes)
  - decide whether to include "immediates" in the encoding of the instructions, as suggested above (and as in RISC-V)

  but those are choices you could make yourself.

A moment of contemplation

You actually have essentially all the information you need to write a program to simulate a stack machine.

- Each register is some member variable of an object of class **`StackMachine`** (which you would define yourself). The stack machine's "memory" can be implemented as one long array of integers.
- Each machine instruction would be a method which affects the program counter, possibly the stack pointer, *etc.* You'd have to set some conventions:
  - Whether / how to map instructions to integers (opcodes)
  - decide whether to include "immediates" in the encoding of the instructions, as suggested above (and as in RISC-V)

  but those are choices you could make yourself.

(See **`StackInterpreter.java`** for an example implementation).

A naive code generator for the stack machine

We consider a syntax-directed code generator
for a simple source language:

A naive code generator for the stack machine

We consider a syntax-directed code generator
for a simple source language:

$$S \;::=\; S \texttt{ ; } S \;\mid\; x \texttt{ := } E \;\mid\; \texttt{for ID = } E \texttt{ to } E \texttt{ \{ } S \texttt{ \}}$$

$$E \;::=\; \text{NUM} \;\mid\; \text{ID} \;\mid\; E \texttt{ + } E \;\mid\; E \texttt{ - } E \;\mid\; E \texttt{ * } E \;\mid\; E \texttt{ / } E \;\mid\; \texttt{-} E$$

A naive code generator for the stack machine

We consider a syntax-directed code generator
for a simple source language:

$$S \;::=\; S\ \texttt{;}\ S \mid x\ \texttt{:=}\ E \mid \texttt{for}\ \mathrm{ID}\ \texttt{=}\ E\ \texttt{to}\ E\ \texttt{\{}\ S\ \texttt{\}}$$

$$E \;::=\; \mathrm{NUM} \mid \mathrm{ID} \mid E\ \texttt{+}\ E \mid E\ \texttt{-}\ E \mid E\ \texttt{*}\ E \mid E\ \texttt{/}\ E \mid \texttt{-}\ E$$

Example program: `x := 0; for i = 1 to 100 { x := x + i }`

We use the following definitions to help us to realise trees

```
class Statement extends Tree { ... }
class Expression extends Tree { ... }
```

A naive code generator for the stack machine

We consider a syntax-directed code generator
for a simple source language:

$$S \;::=\; S \texttt{ ; } S \;\mid\; x \texttt{ := } E \;\mid\; \texttt{for ID = } E \texttt{ to } E \texttt{ \{ } S \texttt{ \}}$$

$$E \;::=\; \text{NUM} \;\mid\; \text{ID} \;\mid\; E \texttt{ + } E \;\mid\; E \texttt{ - } E \;\mid\; E \texttt{ * } E \;\mid\; E \texttt{ / } E \;\mid\; \texttt{ - } E$$

Example program: `x := 0; for i = 1 to 100 { x := x + i }`

We use the following definitions to help us to realise trees

```
class Statement extends Tree { ... }
class Expression extends Tree { ... }

class Sequence extends Statement { ... }
class Assign extends Statement { ... }
class ForLoop extends Statement { ... }
```

A naive code generator for the stack machine

We consider a syntax-directed code generator
for a simple source language:

$$S \;::=\; S \texttt{ ; } S \mid x \texttt{ := } E \mid \texttt{for ID = } E \texttt{ to } E \texttt{ \{ } S \texttt{ \}}$$

$$E \;::=\; \text{NUM} \mid \text{ID} \mid E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{- } E$$

Example program: `x := 0; for i = 1 to 100 { x := x + i }`

We use the following definitions to help us to realise trees

```
class Statement extends Tree { ... }
class Expression extends Tree { ... }

class Sequence extends Statement { ... }
class Assign extends Statement { ... }
class ForLoop extends Statement { ... }

class NumLiteral extends Expression { ... }
class Identifier extends Expression { ... }
class BinaryOpExpr extends Expression { ... }
class UnaryOpExpr extends Expression { ... }
```

A naive code generator for the stack machine

Simple source language:

$$
\begin{array}{lcl}
S & ::= & S \texttt{ ; } S \mid x \texttt{ := } E \mid \texttt{for ID = } E \texttt{ to } E \texttt{ \{ } S \texttt{ \}} \\
E & ::= & \text{NUM} \mid \text{ID} \mid E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{ - } E
\end{array}
$$

A naive code generator for the stack machine

Simple source language:

$$
\begin{array}{lcl}
S & ::= & S \texttt{ ; } S \mid x \texttt{ := } E \mid \texttt{for ID = } E \texttt{ to } E \texttt{ \{ } S \texttt{ \}} \\
E & ::= & \text{NUM} \mid \text{ID} \mid E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{ - } E
\end{array}
$$

The way in which we generate assembly code is directed by the Tree data type: we deal with each of the alternatives using a separate rule.

A naive code generator for the stack machine

Simple source language:

$$S \;::=\; S\ \texttt{;}\ S \mid x\ \texttt{:=}\ E \mid \texttt{for}\ \mathrm{ID}\ \texttt{=}\ E\ \texttt{to}\ E\ \texttt{\{}\ S\ \texttt{\}}$$
$$E \;::=\; \mathrm{NUM} \mid \mathrm{ID} \mid E\ \texttt{+}\ E \mid E\ \texttt{-}\ E \mid E\ \texttt{*}\ E \mid E\ \texttt{/}\ E \mid \texttt{-}\ E$$

The way in which we generate assembly code is directed by the Tree data type: we deal with each of the alternatives using a separate rule.

- A common approach in Java is to use `instanceof`, or **reflection** (*e.g.*, `getClass().getName()`).
- New in Java 17 (2021): pattern matching for switch.
- Alternatively, you can use a **visitor pattern** for Tree traversal (*e.g.*, in visitors generated by ANTLR).

A naive code generator for the stack machine

Recall that the signature of our code generator was as follows.

```
class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

A naive code generator for the stack machine

Recall that the signature of our code generator was as follows.

```
class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

We realise a particular "`Representation`" of programs:

- specialised for assembly programs in our stack machine;
- implements `addRepresentation` with overloaded methods.

```
class StackMachineAssembly implements Representation {
  List<String> instrs;
```

A naive code generator for the stack machine

Recall that the signature of our code generator was as follows.

```
class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

We realise a particular "`Representation`" of programs:

- specialised for assembly programs in our stack machine;
- implements `addRepresentation` with overloaded methods.

```
class StackMachineAssembly implements Representation {
  List<String> instrs;

  String emitCode() { // concatenate instrs, etc. }
```

A naive code generator for the stack machine

Recall that the signature of our code generator was as follows.

```
class Representation {
  public void addRepresentation ( Tree s );
  /* ... */ }
```

We realise a particular "**Representation**" of programs:

- specialised for assembly programs in our stack machine;
- implements **addRepresentation** with overloaded methods.

```
class StackMachineAssembly implements Representation {
  List<String> instrs;

  String emitCode() { // concatenate instrs, etc. }


  void addRepresentation ( Tree s ) {
    if (s instanceof ForLoop) {
      addRepresentation((ForLoop) s) }
    else if (s instanceof Assign) {
      addRepresentation((Assign) s) }
    else /* ... */ }}
```

Translation of Sequences

```
class Sequence extends Statement {
  public Statement lhs;  public Statement rhs;
  /* ... */ }
```

Translation of Sequences

```
class Sequence extends Statement {
  public Statement lhs;  public Statement rhs;
  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( Sequence s ) {
  addToRepresentation(s.lhs);
  addToRepresentation(s.rhs);
}
```

Translation of Sequences

```
class Sequence extends Statement {
  public Statement lhs;  public Statement rhs;
  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( Sequence s ) {
  addToRepresentation(s.lhs);
  addToRepresentation(s.rhs);
}
```

Here, **recursion** does all of the work:

- **`lhs`** adds to the representation of left-hand sub-program;
- **`rhs`** adds to the representation of right-hand sub-program;

Translation of Sequences

```
class Sequence extends Statement {
  public Statement lhs;  public Statement rhs;
  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( Sequence s ) {
  addToRepresentation(s.lhs);
  addToRepresentation(s.rhs);
}
```

Here, **recursion** does all of the work:

- **`lhs`** adds to the representation of left-hand sub-program;
- **`rhs`** adds to the representation of right-hand sub-program;

Do you see any problem?

Translation of Assignments

```
class Assign extends Statement {
  public Identifier id;  public Expression exp;
  /* ... */ }
```

Translation of Assignments

```
class Assign extends Statement {
  public Identifier id;  public Expression exp;
  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
String idenSymbAddress(Identifier id) {
 /* ··· */ }

void addToRepresentation ( Assign a ) {
  addToRepresentation(a.exp);
  instrs.add("PopRel " + idenSymbAddress(id));
}
```

Note that the result of **`a.exp`** was left on the top of stack!

Translation of `for` loops

Translation of `for` loops

- Involves branching (jumping around)

Translation of `for` loops

- Involves branching (jumping around)
- Needs a way to track (and allocate!) locations of identifiers

```
class ForLoop extends Statement {
  public Identifier iter;
  public Expression fromExp;  public Expression toExp;
  public Statement body;
  /* ... */ }
```

Translation of `for` loops

- Involves branching (jumping around)
- Needs a way to track (and allocate!) locations of identifiers

```
class ForLoop extends Statement {
  public Identifier iter;
  public Expression fromExp;  public Expression toExp;
  public Statement body;
  /* ... */ }
```

In **`StackMachineAssembly`**:

```
int loopCtr;

String newLabel() {
  loopCtr++;
  return "label_" +
    Int.toString(loopCtr); }
```

Translation of `for` loops

- Involves branching (jumping around)
- Needs a way to track (and allocate!) locations of identifiers

```
class ForLoop extends Statement {
  public Identifier iter;
  public Expression fromExp;  public Expression toExp;
  public Statement body;
  /* ... */ }
```

In **`StackMachineAssembly`**:

```
int loopCtr;

String newLabel() {
  loopCtr++;
  return "label_" +
    Int.toString(loopCtr); }
```

- Each **`for`** loop needs two symbolic addresses to manage the loop

Translation of `for` loops

- Involves branching (jumping around)
- Needs a way to track (and allocate!) locations of identifiers

```
class ForLoop extends Statement {
  public Identifier iter;
  public Expression fromExp;  public Expression toExp;
  public Statement body;
  /* ... */ }
```

In **`StackMachineAssembly`**:

```
int loopCtr;

String newLabel() {
  loopCtr++;
  return "label_" +
    Int.toString(loopCtr); }
```

- Each **`for`** loop needs two symbolic addresses to manage the loop
  - We use a method to produce these labels
  - We define a counter to help produce unique labels

Translation of `for` loops

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
```

Translation of `for` loops

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
```

Translation of `for` loops

In `StackMachineAssembly` class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
```

Translation of `for` loops

In `StackMachineAssembly` class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
  instrs.add(startLabel + ":");
```

Translation of `for` loops

In `StackMachineAssembly` class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
  instrs.add(startLabel + ":");
  addToRepresentation(f.toExp);
  instrs.add("PushRel " + iterAddr);
  instrs.add("CompGT");
  instrs.add("JumpTrue " + endLabel);
```

Translation of `for` loops

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
  instrs.add(startLabel + ":");
  addToRepresentation(f.toExp);
  instrs.add("PushRel " + iterAddr);
  instrs.add("CompGT");
  instrs.add("JumpTrue " + endLabel);
  addToRepresentation(f.body);
```

Translation of `for` loops

In `StackMachineAssembly` class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
  instrs.add(startLabel + ":");
  addToRepresentation(f.toExp);
  instrs.add("PushRel " + iterAddr);
  instrs.add("CompGT");
  instrs.add("JumpTrue " + endLabel);
  addToRepresentation(f.body);
  instrs.add("PushRel " + iterAddr);
  instrs.add("PushImm 1");
  instrs.add("Plus");
```

Translation of `for` loops

In `StackMachineAssembly` class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
  instrs.add(startLabel + ":");
  addToRepresentation(f.toExp);
  instrs.add("PushRel " + iterAddr);
  instrs.add("CompGT");
  instrs.add("JumpTrue " + endLabel);
  addToRepresentation(f.body);
  instrs.add("PushRel " + iterAddr);
  instrs.add("PushImm 1");
  instrs.add("Plus");
  instrs.add("PopRel " + iterAddr);
```

Translation of `for` loops

In `StackMachineAssembly` class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
  instrs.add(startLabel + ":");
  addToRepresentation(f.toExp);
  instrs.add("PushRel " + iterAddr);
  instrs.add("CompGT");
  instrs.add("JumpTrue " + endLabel);
  addToRepresentation(f.body);
  instrs.add("PushRel " + iterAddr);
  instrs.add("PushImm 1");
  instrs.add("Plus");
  instrs.add("PopRel " + iterAddr);
  instrs.add("Jump " + startLabel);
```

Translation of `for` loops

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( ForLoop f ) {
  String startLabel = newLabel();
  String endLabel = newLabel();
  String iterAddr = idenSymbAddress(f.iter);
  addToRepresentation(f.fromExp);
  instrs.add("PopRel " + iterAddr);
  instrs.add(startLabel + ":");
  addToRepresentation(f.toExp);
  instrs.add("PushRel " + iterAddr);
  instrs.add("CompGT");
  instrs.add("JumpTrue " + endLabel);
  addToRepresentation(f.body);
  instrs.add("PushRel " + iterAddr);
  instrs.add("PushImm 1");
  instrs.add("Plus");
  instrs.add("PopRel " + iterAddr);
  instrs.add("Jump " + startLabel);
  instrs.add(endLabel + ":");
}
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;

  /* ... */ }
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;

  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( BinaryOpExpr e ) {
   addToRepresentation(e.lhs);
   addToRepresentation(e.rhs);
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;

  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( BinaryOpExpr e ) {
   addToRepresentation(e.lhs);
   addToRepresentation(e.rhs);
   switch (e.optor) {
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;

  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( BinaryOpExpr e ) {
   addToRepresentation(e.lhs);
   addToRepresentation(e.rhs);
   switch (e.optor) {
     case Plus: instrs.add("Plus"); break;
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;

  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( BinaryOpExpr e ) {
   addToRepresentation(e.lhs);
   addToRepresentation(e.rhs);
   switch (e.optor) {
     case Plus: instrs.add("Plus"); break;
     case Minus: instrs.add("Minus"); break;
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;

  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( BinaryOpExpr e ) {
   addToRepresentation(e.lhs);
   addToRepresentation(e.rhs);
   switch (e.optor) {
     case Plus: instrs.add("Plus"); break;
     case Minus: instrs.add("Minus"); break;
     case Times: instrs.add("Times"); break;
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;

  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( BinaryOpExpr e ) {
   addToRepresentation(e.lhs);
   addToRepresentation(e.rhs);
   switch (e.optor) {
     case Plus: instrs.add("Plus"); break;
     case Minus: instrs.add("Minus"); break;
     case Times: instrs.add("Times"); break;
     case Slash: instrs.add("Slash"); break;
```

Translation of binary operations

```
class BinaryOpExpr extends Expression {
  public enum Op { Plus, Minus, Times, Divide };

  public Expression lhs;  public Expression rhs;
  public Op optor;
  /* ... */ }
```

In **`StackMachineAssembly`** class:

```
void addToRepresentation ( BinaryOpExpr e ) {
   addToRepresentation(e.lhs);
   addToRepresentation(e.rhs);
   switch (e.optor) {
     case Plus: instrs.add("Plus"); break;
     case Minus: instrs.add("Minus"); break;
     case Times: instrs.add("Times"); break;
     case Slash: instrs.add("Slash"); break;
  }}
```

Stack machine code generation: summary

Stack machines have very few registers

- Manages computation directly using the stack
- Nearly every operation involves the stack pointer

Stack machine code generation: summary

Stack machines have very few registers

- Manages computation directly using the stack
- Nearly every operation involves the stack pointer
- Computation involves frequent loading:
  - Of constants, to the top of the stack
  - From memory, to the top of the stack
  - From the top of the stack, to memory

Stack machine code generation: summary

Stack machines have very few registers

- Manages computation directly using the stack
- Nearly every operation involves the stack pointer
- Computation involves frequent loading:
  - Of constants, to the top of the stack
  - From memory, to the top of the stack
  - From the top of the stack, to memory

A code generator would use the recursive structure of the tree (and counter variables / a symbolic address table) to generate assembly code for the stack machine.

The material in the textbooks

- Dragon Book: Chapter 2, introduction to code generation, Chapter 8, especially 8.1 and 8.6.

The material in the textbooks

- Dragon Book: Chapter 2, introduction to code generation, Chapter 8, especially 8.1 and 8.6.
- Appel, Palsberg: Chapter 7, Chapter 9 (although Appel, Palsberg skip simple code generation and concentrate on finding the best instruction to match the context).

The material in the textbooks

- Dragon Book: Chapter 2, introduction to code generation, Chapter 8, especially 8.1 and 8.6.
- Appel, Palsberg: Chapter 7, Chapter 9 (although Appel, Palsberg skip simple code generation and concentrate on finding the best instruction to match the context).
- "Engineering a compiler": Section 4.4: ad-hoc syntax-directed translation, especially Figure 4.14.