Unit 13: Instruction Processing, JIT and Stack Machines

(aka. "Fitting it all Together")

SCC 312 Compilation

John Vidler
j.vidler@lancaster.ac.uk

Split-Stage Compilation

- This exists in two forms
  - Compilers with an intermediate language (IL)
  - Compilers which export code intended for an *interpreter*

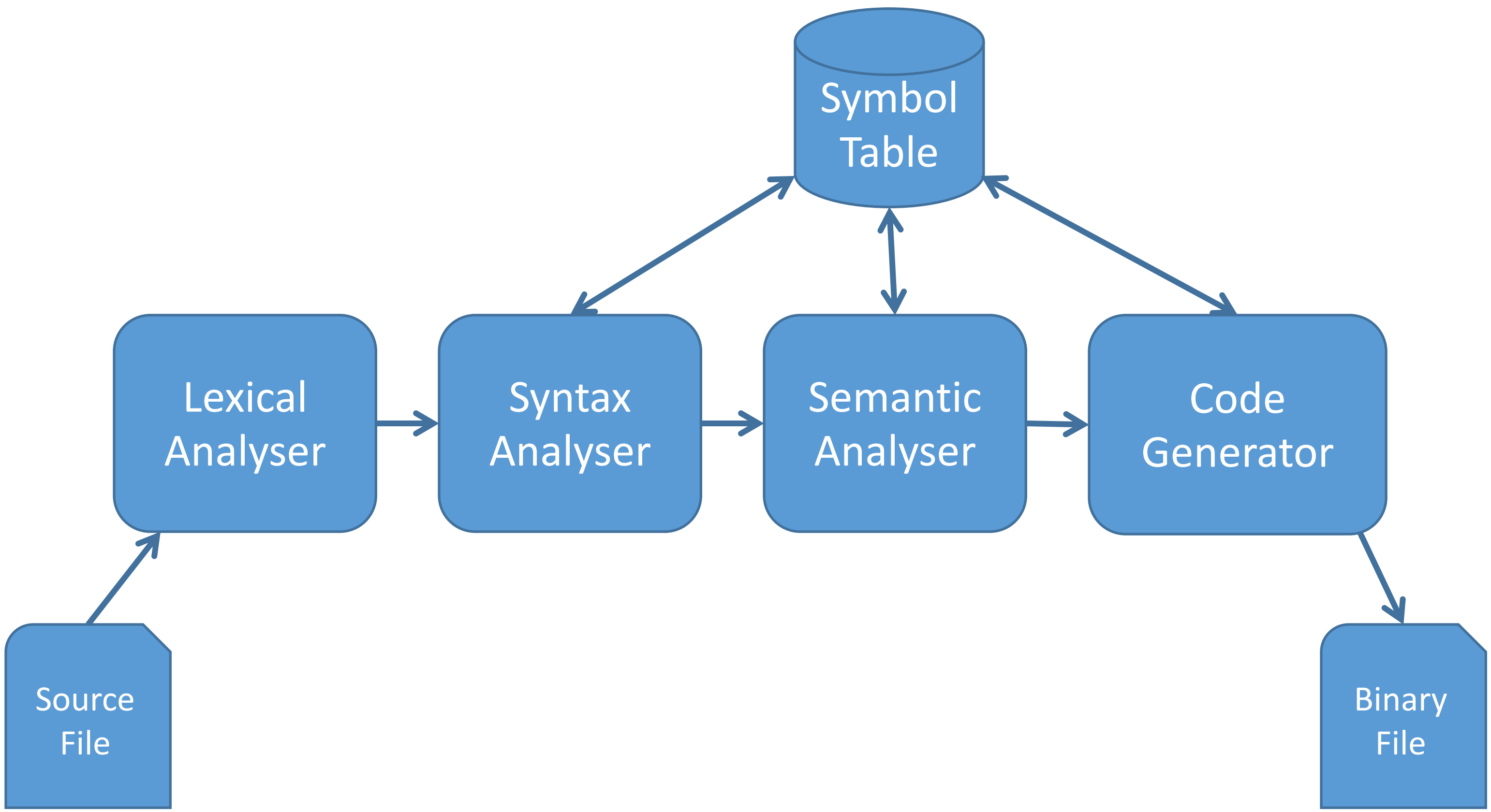
- In both cases, rather than compile to a specific target, a halfway language is used

- This must be high level enough to be flexible ...
  - ... but cannot be too high level, or the final compiler steps will take too long!

Intermediate Language Compilers

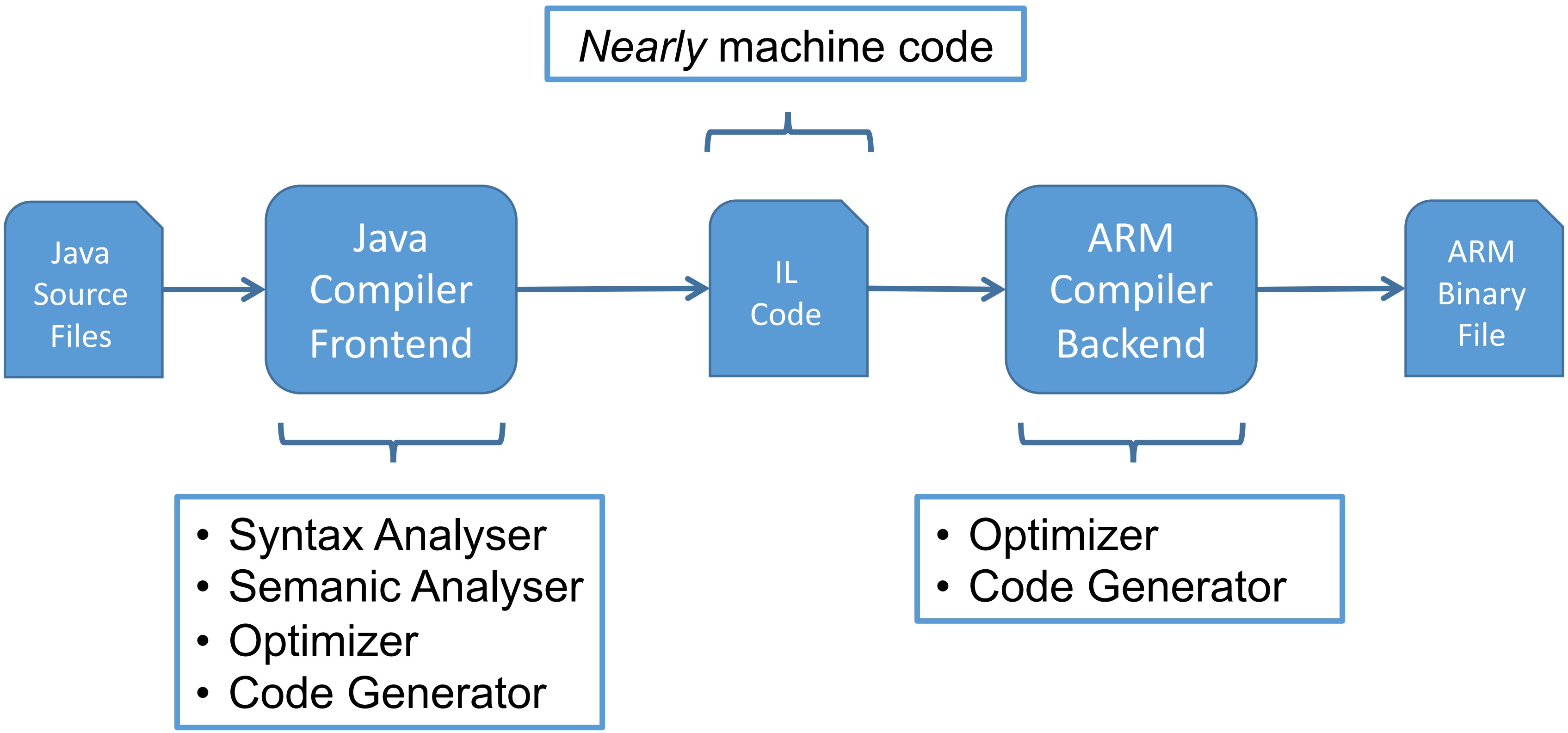
A Structure Reminder

What we have seen so far

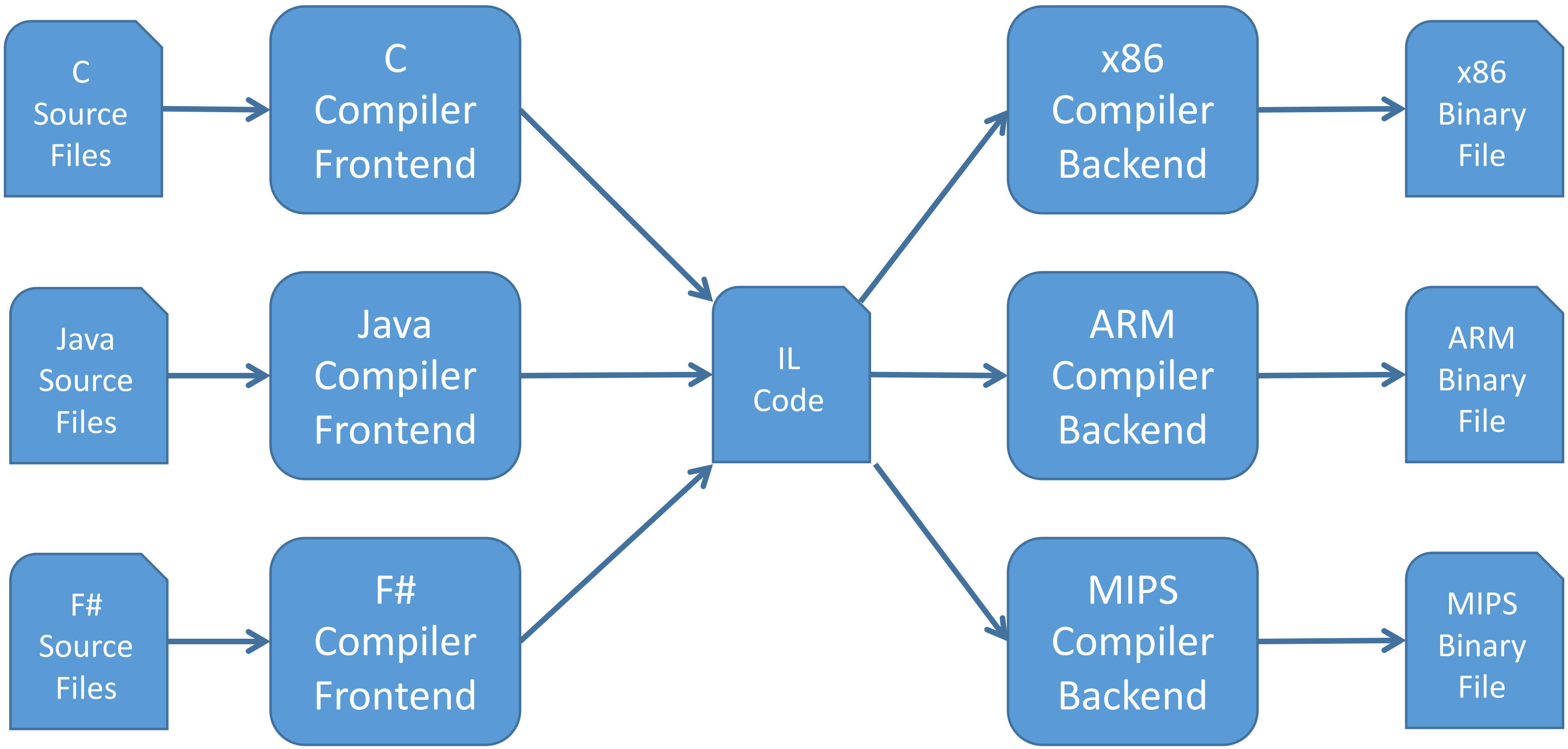


Split Stage Compilation

(Via an Intermediate Language)

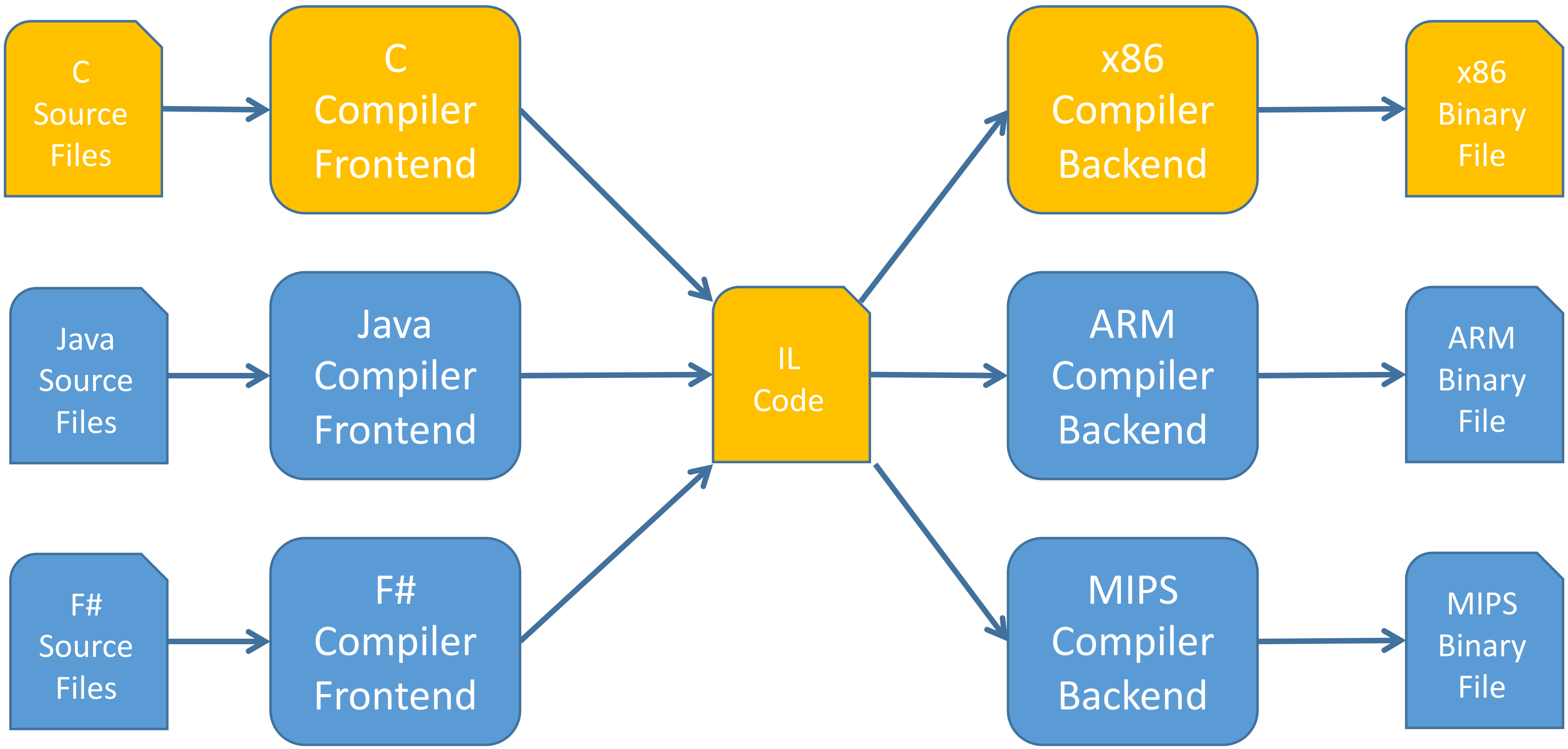


Split Stage Compilation
(Via an Intermediate Language)



Split Stage Compilation

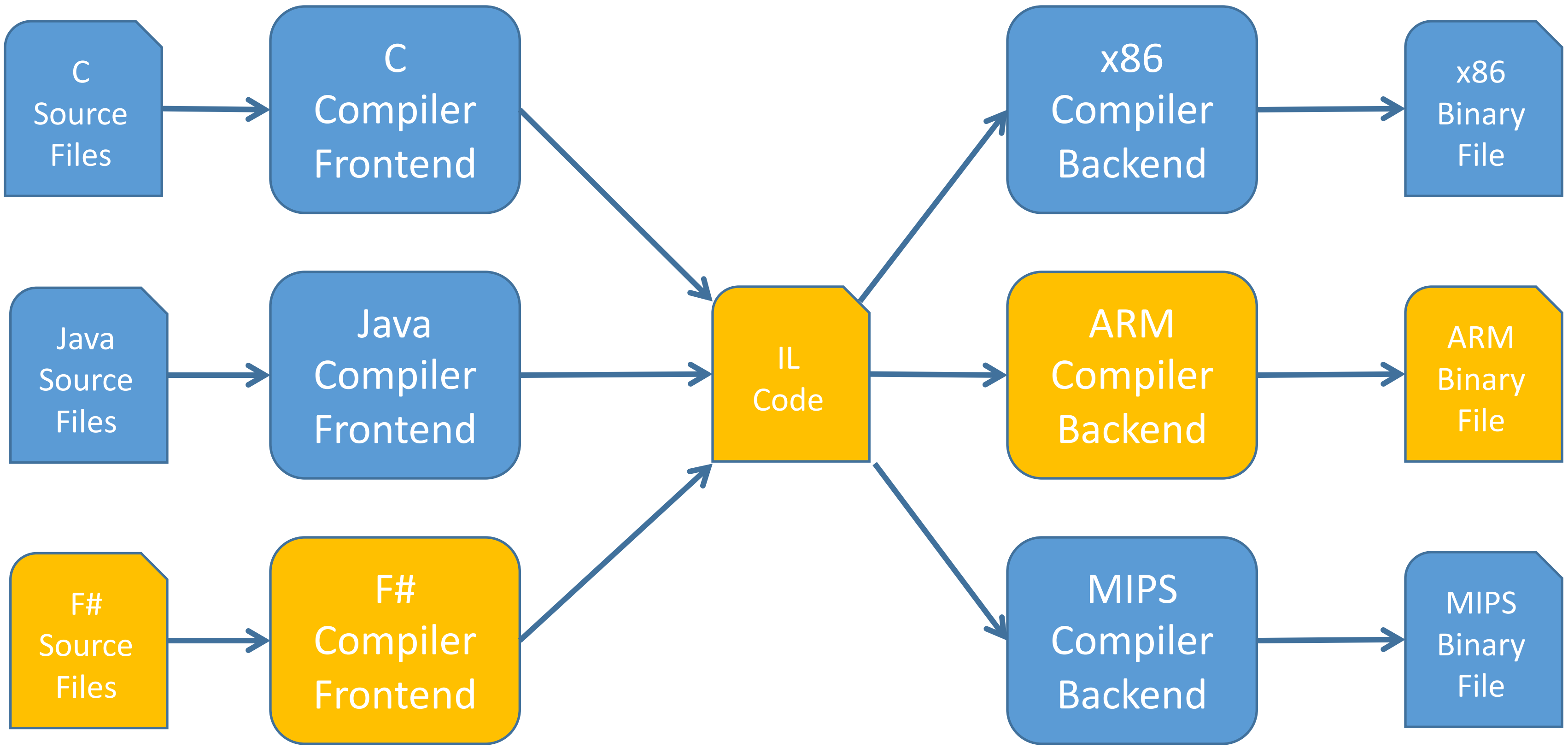
(Via an Intermediate Language)



Building C code for x86 target platforms

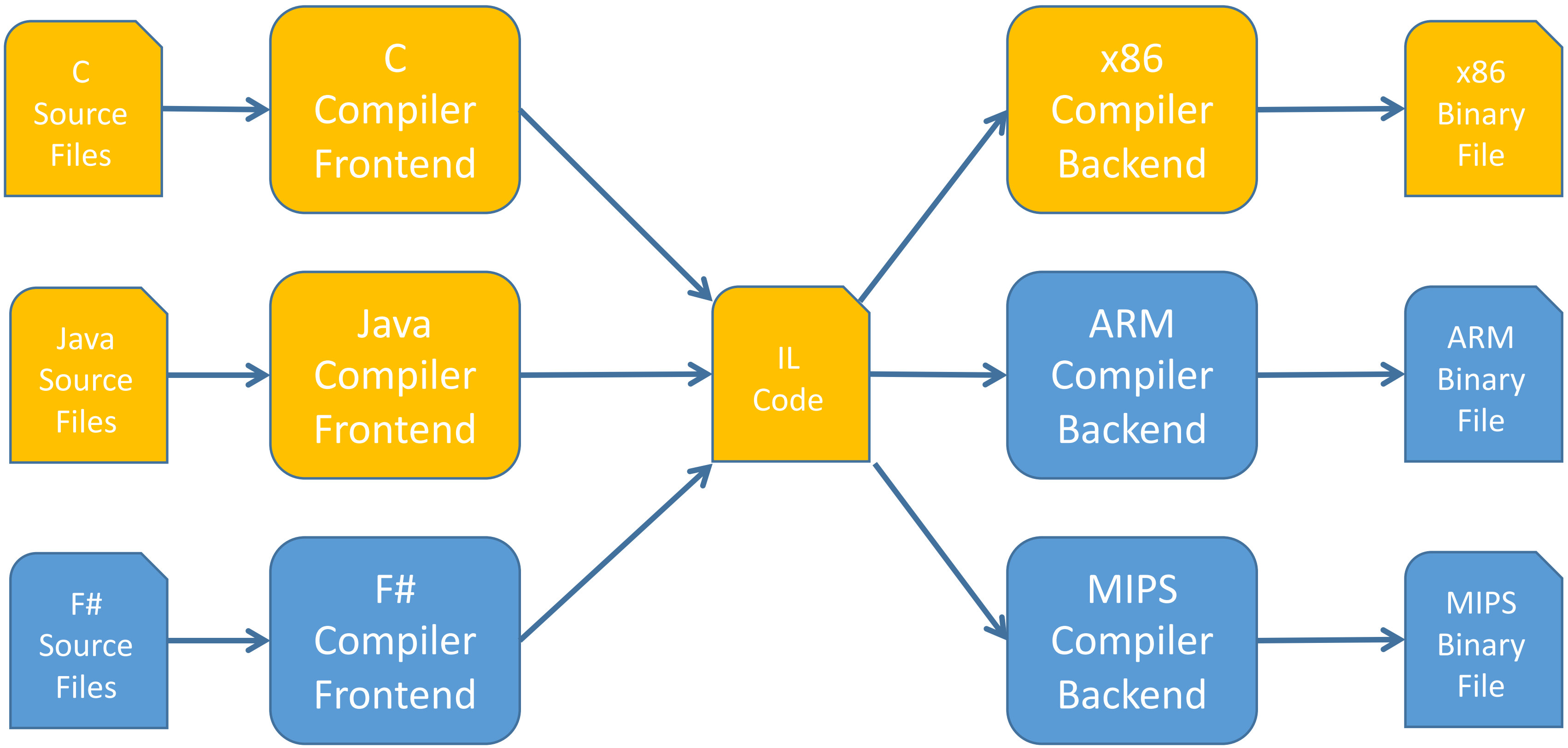
Split Stage Compilation

(Via an Intermediate Language)



Building F# code for ARM target platforms

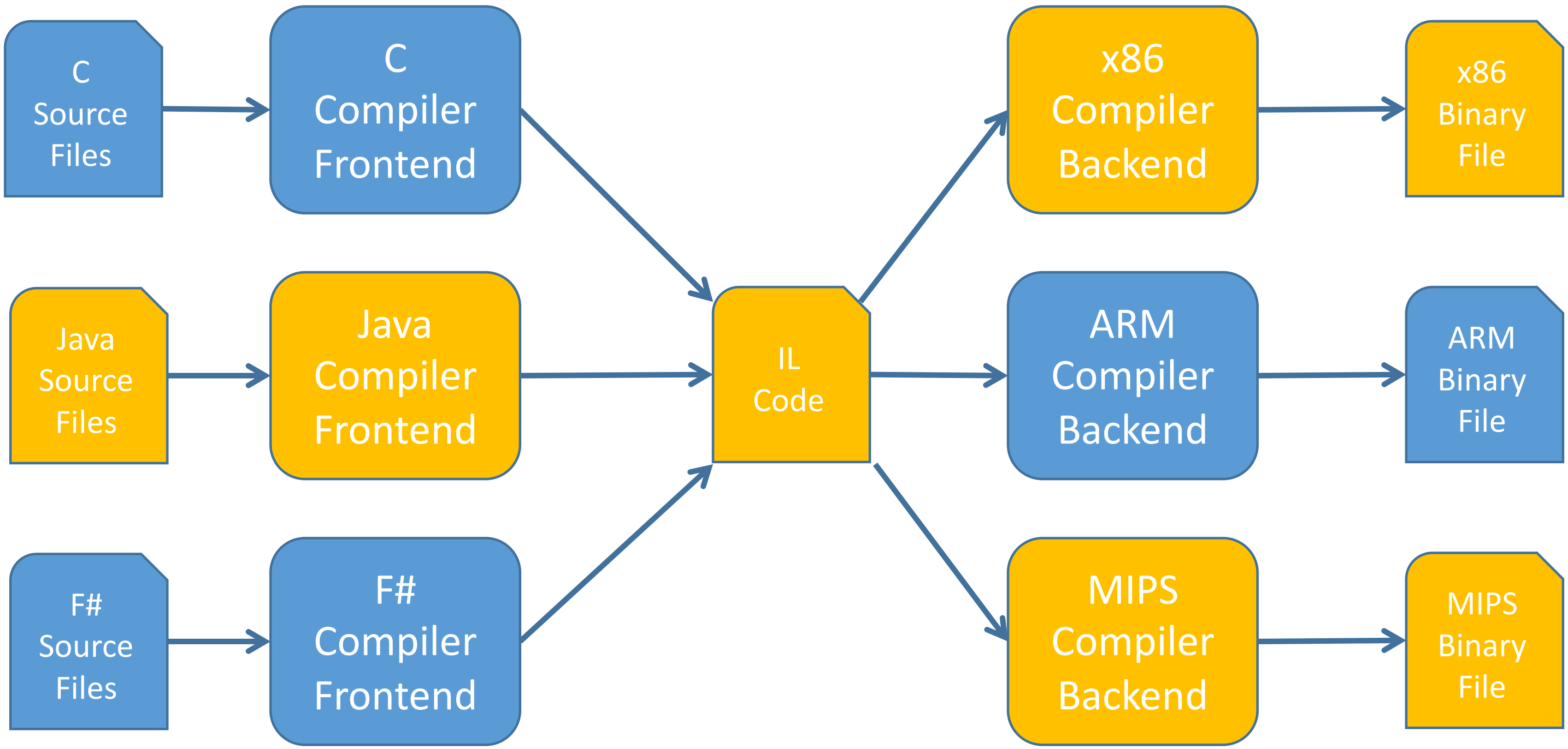
Split Stage Compilation
(Via an Intermediate Language)



Building **mixed** C and Java code for x86 target platforms

Split Stage Compilation

(Via an Intermediate Language)



Building Java code for x86 **and** MIPS target platforms

Split Stage Compilation
(Via an Intermediate Language)

- This compiler design still has problems
  - Constraints in the IL language may lead to over-optimisation in the front-end
  - Constraints in the back-end may lead to overly verbose code for some targets

  - The lack of communication between the layers can lead to lost semantics between the two halves.
    - The parse tree is gone by the time we reach the back-end
    - No token metadata is available

- Linkage in *mixed* configurations may break down due to back-end workings being different
  - Call semantics may be different between MIPS and x86, for example

Split Stage Compilation
(Via an Intermediate Language)

- The Intermediate Language may also constrain the types of targets that are possible (or reasonable) to create
  - Register-less machines are hard to handle with an IL that expresses things in terms of registers
  - Accumulator-based machines can result in overly verbose code, requiring more optimisation at the backend than normal

- But overall, the design provides a good degree of freedom, and prevents the internals of the compiler becoming too convoluted.
  - That said... look at the LLVM compiler code some time - it's not a fun place :|

Just In Time Compilation

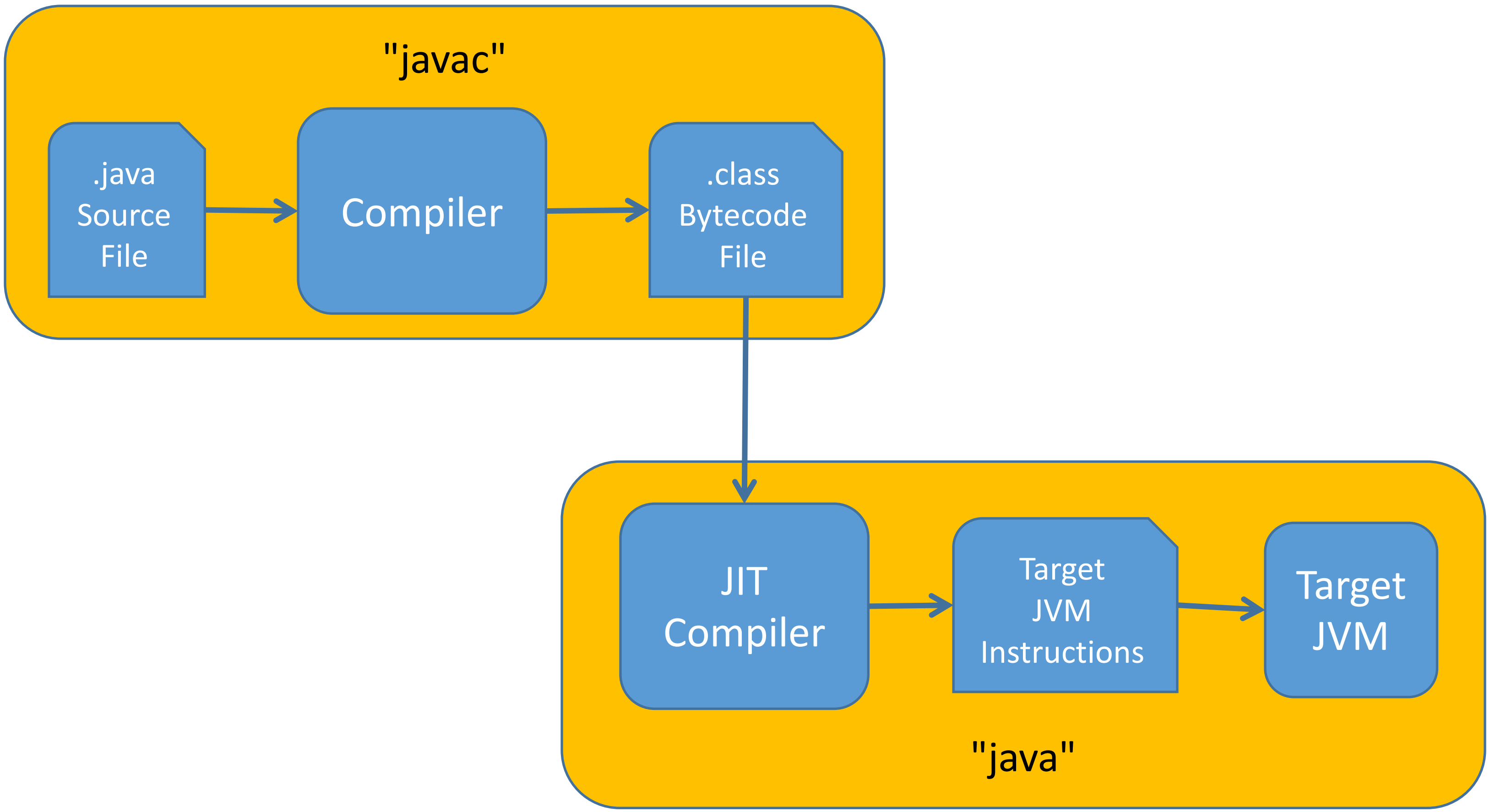
Just in Time Compilation

- Rather than Split-Stage compilers, a JIT compiler does its final stage as part of the execution environment.
  - The intermediate language becomes a merge of high-level and low-level opcodes

- The local high-level opcodes are then transcoded just before the runtime
  - This means we can do this for target-specific reasons
    - Runtime-environment patches
  - This also means we can effectively update the binary *after* the main compilation stage
    - Patches to bad behaviour, extensions, performance increases
    - Runtime flags can literally modify the code that actually runs

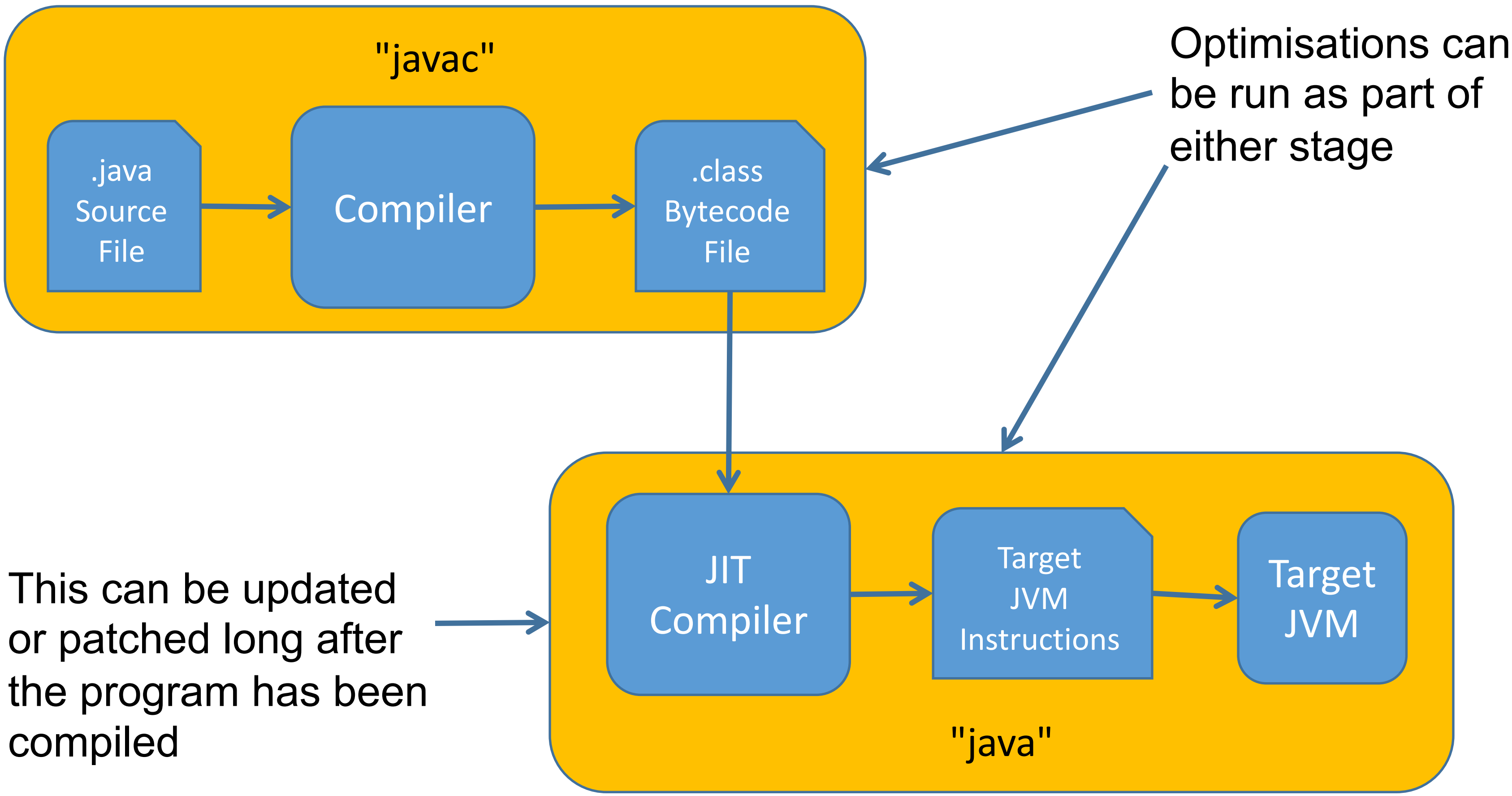
Just In Time (JIT) Compilation + Interpreter

(Roughly how Java works)



Just In Time (JIT) Compilation + Interpreter

(Roughly how Java works)



JIT - Compiler:
Java to Bytecode

- The compiler starts with the java source
  - We assume that this is from a method - otherwise it would fail in the syntax analyser!

```
outer:
for (int i = 2; i < 1000; i++) {
  for (int j = 2; j < i; j++) {
    if (i % j == 0)
      continue outer;
  }
  System.out.println (i);
}
```

- At this stage, all the usual steps are taken:
  - Lexical Analysis, Syntax Analysis, Semantic Analysis
  - But we finish with *Bytecode Generation*

JIT - Compiler:
Java to Bytecode

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush 1000
6:   if_icmpge 44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge 31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne 25
22:  goto 38
25:  iinc 2, 1
28:  goto 11
31:  getstatic #84;
34:  iload_1
35:  invokevirtual #85;
38:  iinc 1, 1
41:  goto 2
44:  return
```

Method java/io/PrintStream.println:(I)V

Field java/lang/System.out:Ljava/io/PrintStream;

Example code from https://en.wikipedia.org/wiki/Java_bytecode, retrieved 2018

JIT - Compiler:
Java to Bytecode

- Most of the instructions correlate well with assembly-level instructions (direct translations)
  - `iload -> load`
  - `istore -> store`
  - `iconst -> literal integer constant value`
  - `goto -> jump`

- Others are completely alien to most architectures!
  - `getstatic -> load static table reference`
  - `invokevirtual -> invoke method from virtual table`

Taking an LR(0) parser to an Interpreter

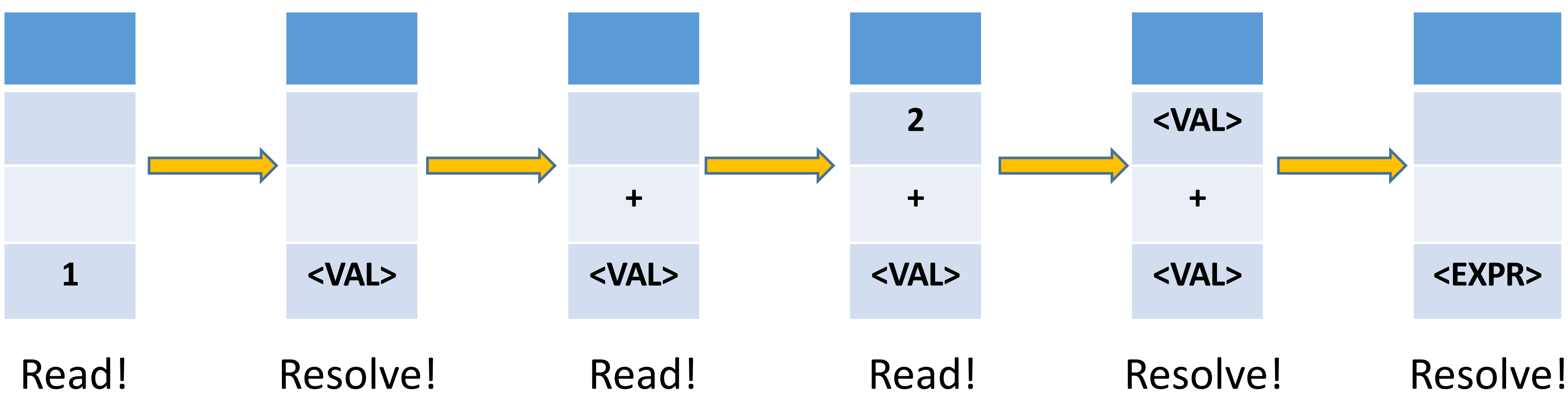
Stack Machines as Interpreters

- Use a bottom-up parse strategy
  - Needs a language with an LR(0) compatable grammar
- Handle each operation as it resolves
- Sanction rules as they are processed
- Hope for no ambiguity or errors!
  - If the input is ambiguous, the interpreter is unable to proceed
    - ... most of the time

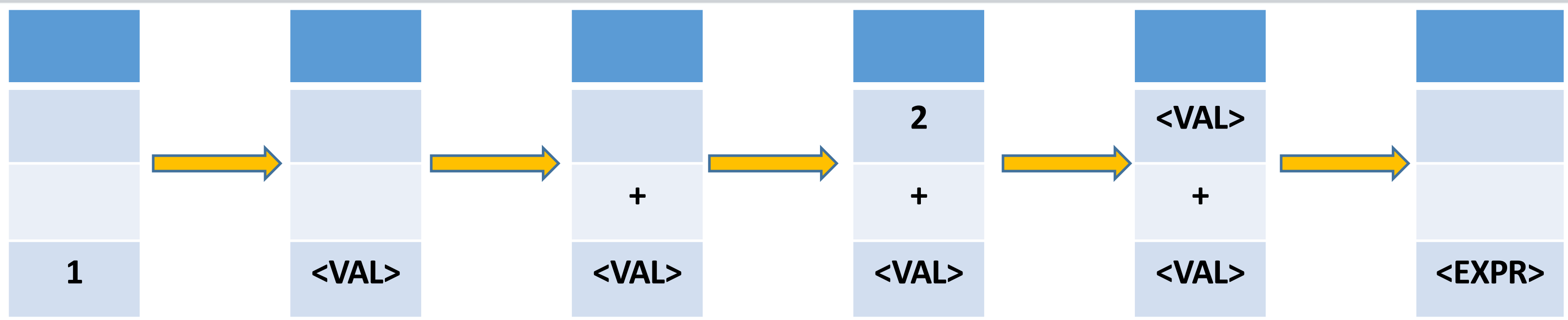
We have already seen how we can design our grammar to avoid this

Stack Machines as Interpreters

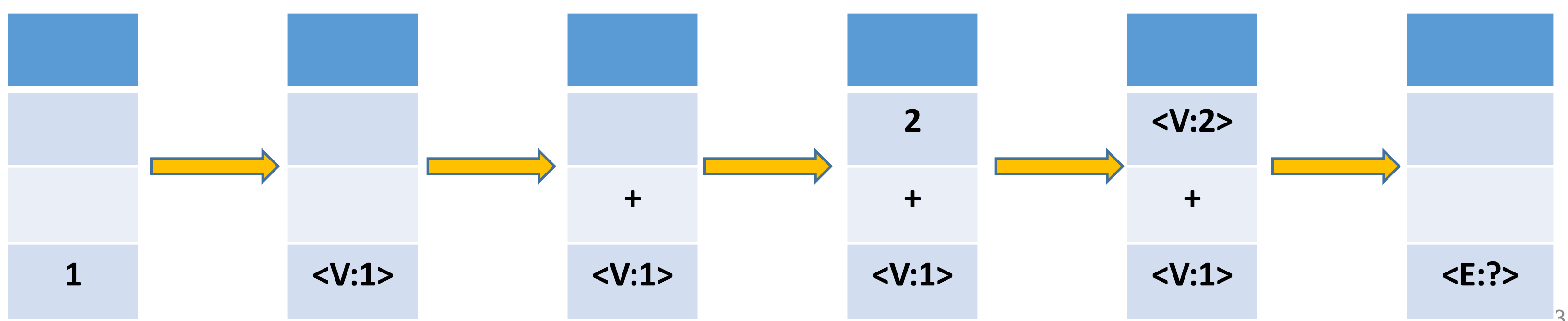
- Recall the way LR(0) parsers generate their tree
  - Push tokens onto a stack
  - Sanction rules by popping the components off the stack and replacing them with the non-terminal



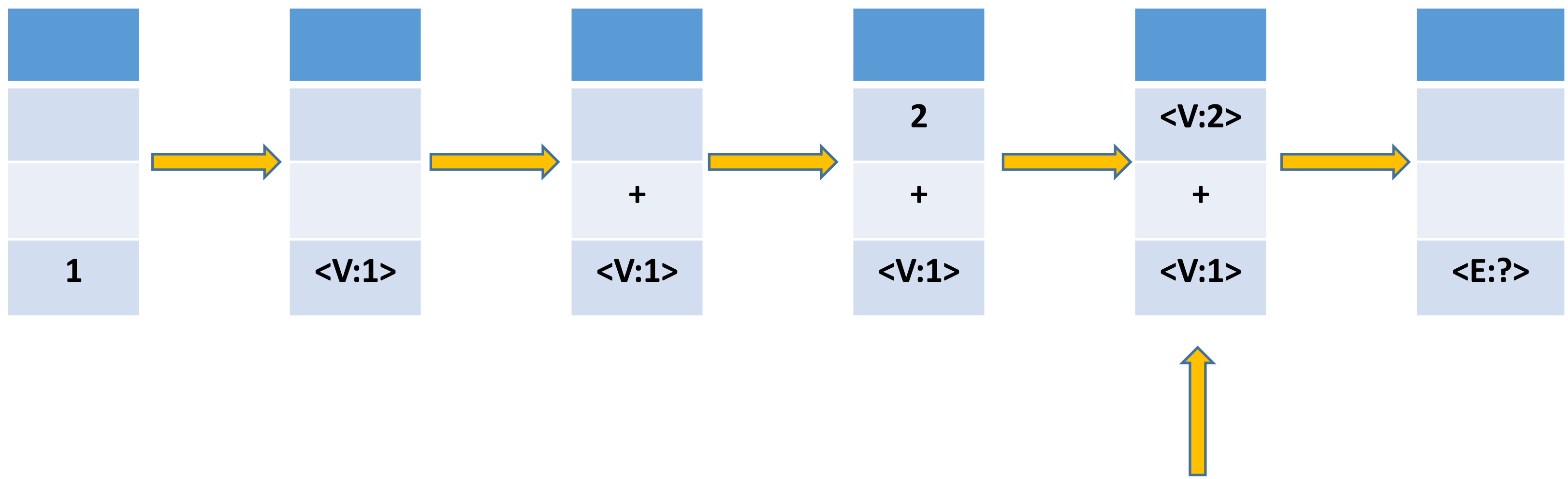
Stack Machines as Interpreters



- While processing the input stream, we can also generate opcodes on the fly.
  - We do need to keep some of the original metadata around on the stack to correctly handle this example - the values are lost in this representation.
  - Instead we keep the result of the rule sanctioning as well as the non-terminal



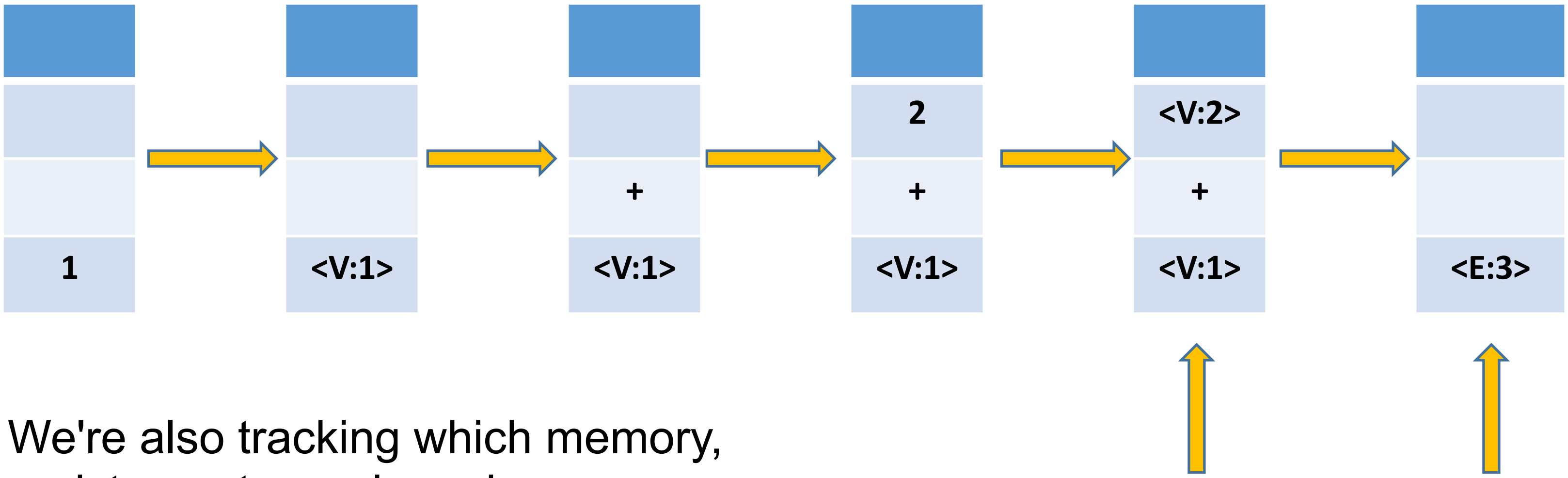
Stack Machines as Interpreters



Note
Our notation here is
`<RULE:VALUE>`

Sanction
`<expr> := <val> + <val>`
AND execute:
`ADD 2, 1`

Stack Machines as Interpreters



- We're also tracking which memory, registers, etc. we have in use
  - Register Table
  - Memory Map

- If interpreting, we do the operations **live**
  - If JIT-ing, we export new binary without the evaluation

Sanction

`<expr> := <val> + <val>`

AND execute:

`ADD 2, 1`

Processing Complete Expressions

x = 1 + y; LR(1) style

- LR(0) has some issues for interpreted languages
  - Without any lookahead, the grammar must have **no ambiguity**
  - Any ambiguous statements end up being processed in-order, which is probably not the order they should be handled.

- LR(1) allows us to fix this problem to a great extent
  - Is there more expression to evaluate?
  - Are there precidence modifiers in the input stream?

- Precidence Modifiers -> "(" and ")", for example.
  - We can use the presence of these to supress the evaulation of the current top stack operation

Processing Complete Expressions

x = 1 + y; LR(1) style

NEXT	Stack State --> Top					
"x"						
"="	SYM(x)					
"1"	SYM(x)	ASSIGN				
"+"	SYM(x)	ASSIGN	LT(1)			
	Here LR(1) is useful to see if we have a completed expression ie. is the next token an EOL ';' ?					
"y"	SYM(x)	ASSIGN	LT(1)	OP(+)		
;	SYM(x)	ASSIGN	LT(1)	OP(+)	SYM(y)	
	Lookahead lets us know we have a complete expression. Now begin evaluating					
	SYM(x)	ASSIGN	LT(3)			
	SYM(x)					

Symbol	Value
x	4
y	2

ADD Op

ASSIGN Op

Stack Machines as Interpreters

- Interpreter Problems
  - Forward declarations are an issue
    - If we haven't seen a function yet, but we're already calling it - how do we jump to it?
    - If we're careful we can build a partially compiled tree for evaulation later
      - Just grab the function blocks, save them for evaluation later
    - OR just enforce a declare-first language
      - Function headers always before calls

  - But! We can combine the both JIT and Interpreter techniques for a better interpreter!
    - Use the Semantic Analyser to track scopes
    - Keep scope code we parse in memory ready for later
    - Build relations between scopes

The END