Compiler Overview

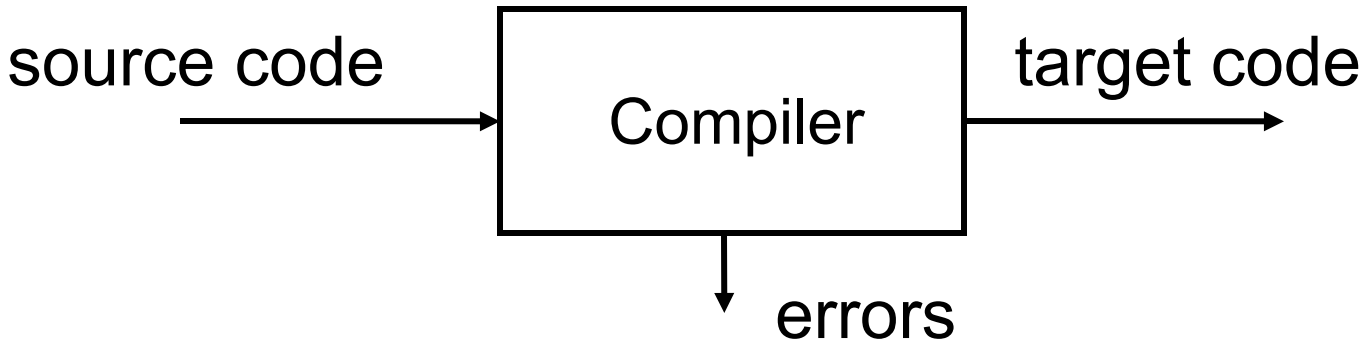
ICS312

Machine-Level and Systems Programming

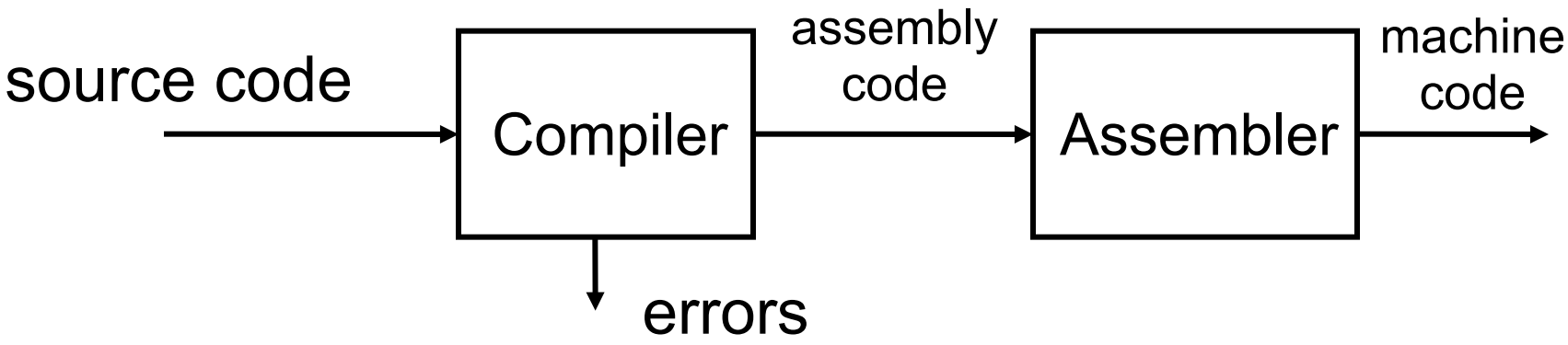
Henri Casanova (henric@hawaii.edu)

What's a compiler

- A compiler is a translator
- It translates from a source language into a target language



- The target code is often assembly



Why Should You Care?

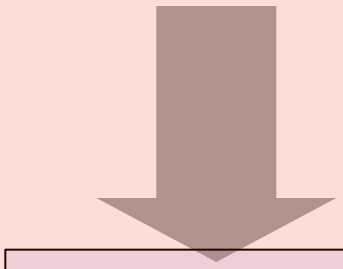
- Granted, chances are that you will not write a compiler
- But knowing how a compiler works is crucial for several reasons
  - To learn fundamental concepts and techniques that are useful beyond compilers
    - Regular expressions and context-free grammars (stay tuned)
  - To be able to write/design tools that use compiler techniques for simpler purposes than programming language translation
    - e.g., to convert files from one format to another
  - To understand why compiled code is sometimes surprisingly fast, and why sometimes it isn't!
  - Because it's a fascinating achievement of mankind :)

The Big Picture (again)

High-level code

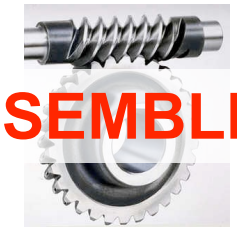
```
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
int i,j;

if (!(f = fopen(filename,"r"))) {
  xbt_assert1(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
  if (!strncmp(buffer,"SCHEDULER",9))
    num_schedulers++;
  if (!strncmp(buffer,"REQUESTSUBMITTER",16))
    num_request_submitters++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_
```



Hand-written Assembly code

```
push    ebp
mov     ebp, esp
push    ebx
mov     eax, 0
mov     ebx, [ebp+8]
shr     ebx, 1
adc     eax, 0
neg     eax
inc     eax
pop     ebx
pop     ebp
```

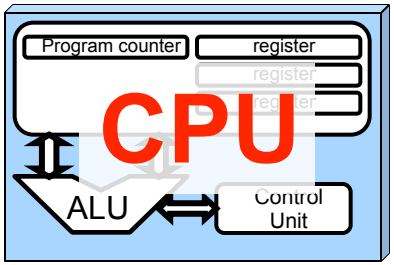


Machine code

```
01000010101011011O
1010101011110101O1
1010010101010001O1
1010101010100101O1
1111000010101001O1
0001010101110101O1
0100000000100001O0
0000100010001000O1
1010010100101010O1
0001010100100101O1
0101010101010101O1
1010101011110101O1
1010101010100101O1
1111000010101001O1
```

Assembly code

```
mov     eax, list_msg
call    print_string
push    dword 10
push    Array
call    printArray
add     esp, 8
push    plus_one
push    dword 10
push    Array
call    map
add     esp, 12
call    print_nl
mov     eax, mapped1_msg
call    print_string
push    dword 10
push    Array
```

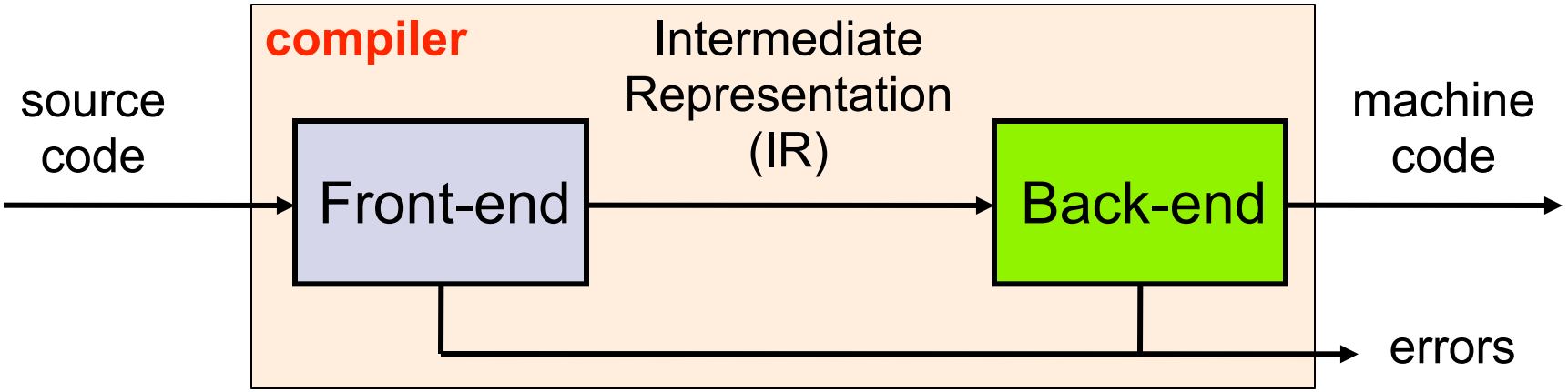


What Should a Compiler Do?

- It should accept legal code and reject illegal code with (hopefully helpful) error messages
- It should generate correct translated code
  - If generating x86 assembly, correct data, bss, and text segments

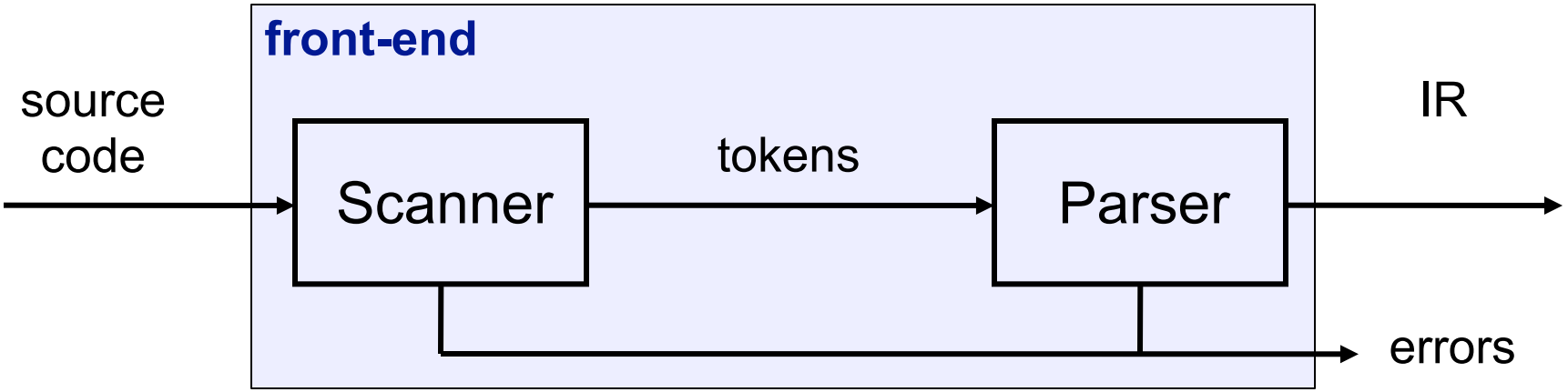
- Although these seem obvious, it wasn't always easy and the first compilers were known to have bugs and limitations
  - Good luck then trying to figure out if a bug is in your code or in the compiler!
  - Still to this day you'll hear people say "I think it's a bug in the compiler" (it's 0.000….001% chance)
  - Poll: have you ever blamed a compiler for a bug or hear anybody do that?
    - If so, was it really a compiler bug????

Traditional 2-Pass Compiler



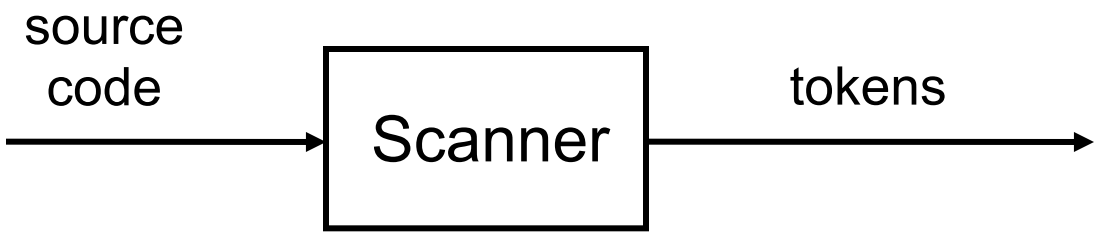
- Compilers use an Intermediate Representation (IR) for the program being compiled
  - Some abstract way to encode the program: tree-like data structures with bells and whistles
- Makes it possible to reuse front-ends/back-ends
  - GCC has front-ends for C, C++, Fortran, Ada, Go; and backends for x86, ARM, RISC-V
  - The LLVM-based Clang C/C++ compiler and Rust rustc compiler share the same backends

What does the Front-End do?



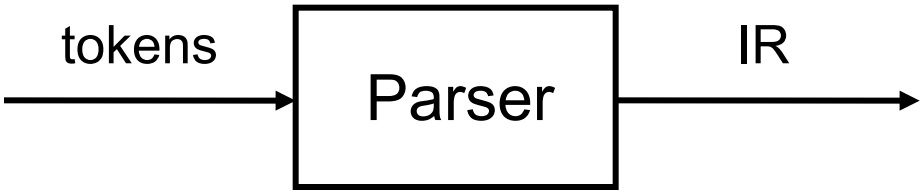
- The front-end is the “easy” (i.e., well-understood for many decades) part of the compiler
- It’s where all the error messages are generated
- Much of the front-end can be automated, and we have well-known tools to generate it
- Some compilers use “implemented by-hand” Scanners or (more rarely) Parsers, for speed

What does the Scanner Do?



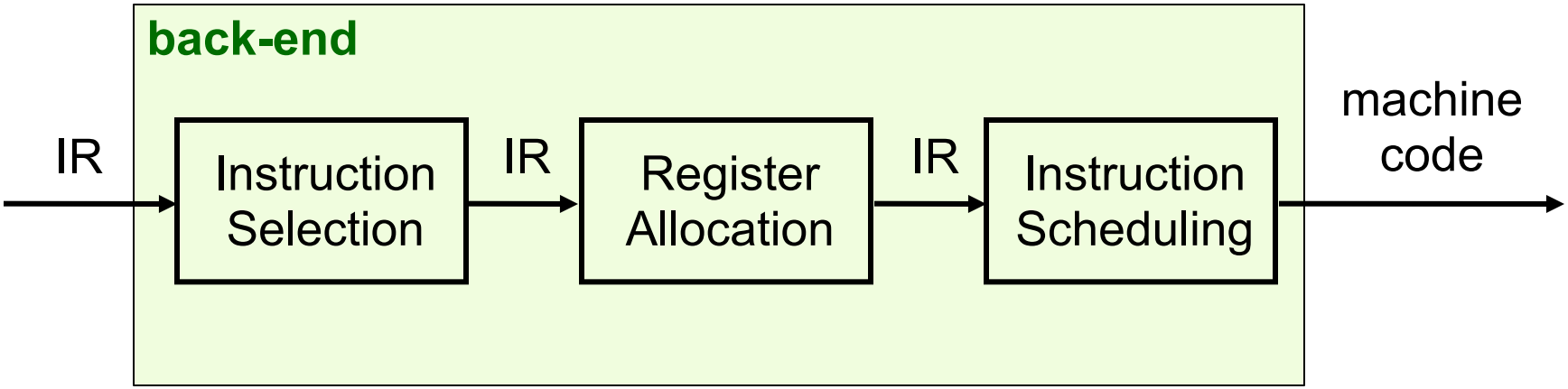
- The Scanner maps a stream of characters (ASCII codes of the characters in the text file that contains your program) into words
- A “word” is called a token, which is really a pair of two things
  - A lexeme: the actual string in the source code
  - A type: what does this word mean in the programming language?
  - Example: (“keyword”, “if”) (“identifier”, “my_var”)

What does the Parser do?



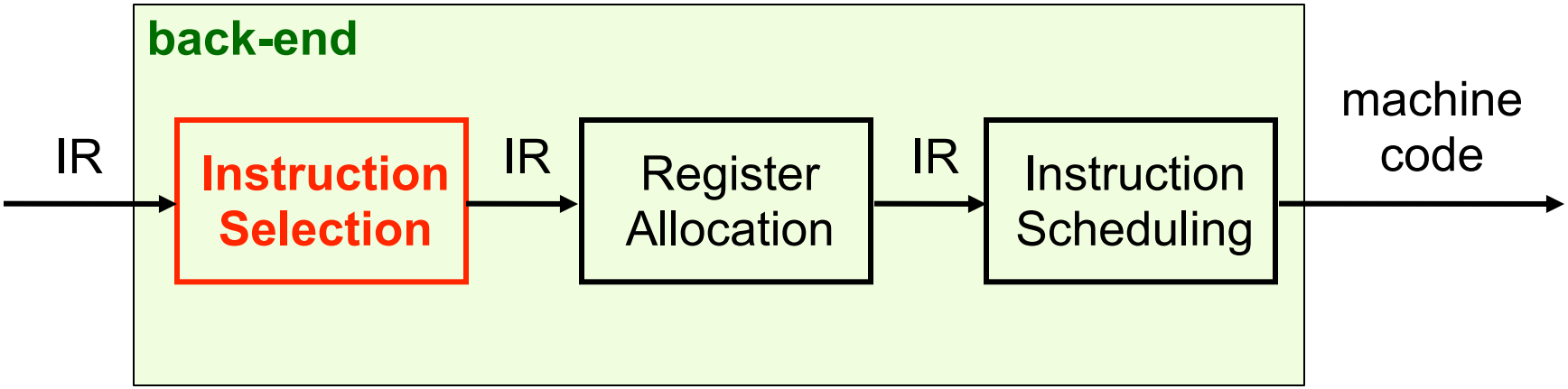
- Recognizes whether the stream of tokens matches the grammar of the language
- Builds an Intermediate Representation (IR): in this course it's an annotated hierarchical view of the programs (known as an abstract syntax tree or AST)
  - We'll look at this in another set of lecture notes
- It's not as simple as the lexer

What does the Back-End do?



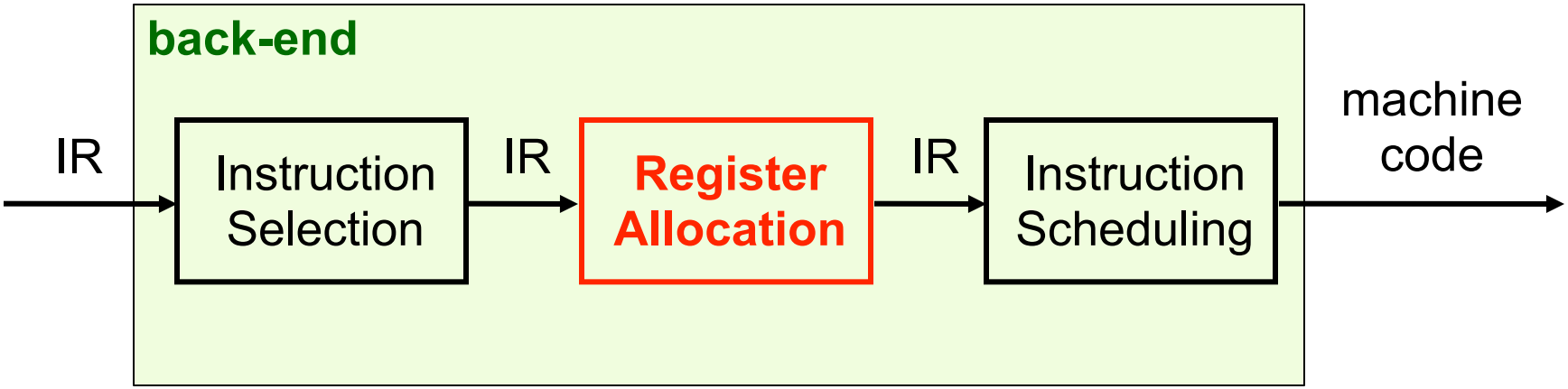
- The back-end translates the IR into machine code
  - It chooses which machine instructions to use to translate the IR
  - It chooses which values should be kept in registers
  - It decides of the order in which instructions should be executed in which order

Instruction Selection



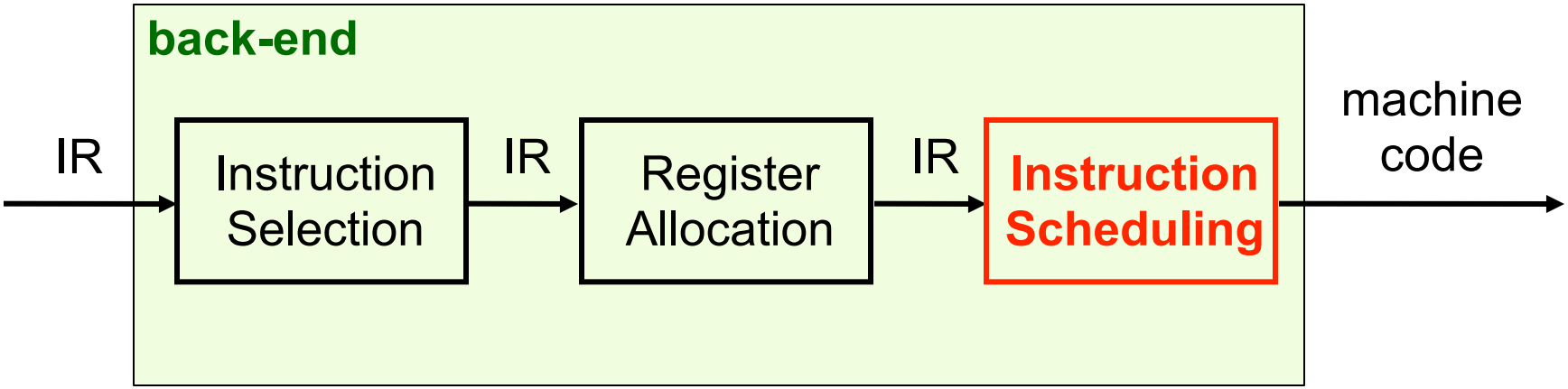
- The goal is to produce fast, compact code
  - E.g., use an “xor eax, eax” rather than “mov eax, 0”
- This used to be a huge issue when ISAs were very complicated with many, many options
  - In the pre-RISC architecture era (CISC)
- Still an issue for vectorized/specialized instructions

Register Allocation



- Registers allow for fast variable access
- But there is a limited number of them
- Optimal allocation is a difficult problem (NP-hard)
- Compilers use approximation algorithms to try to get close to the optimal
  - Think of how you thought about used registers as much as possible when writing assembly…

Instruction Scheduling

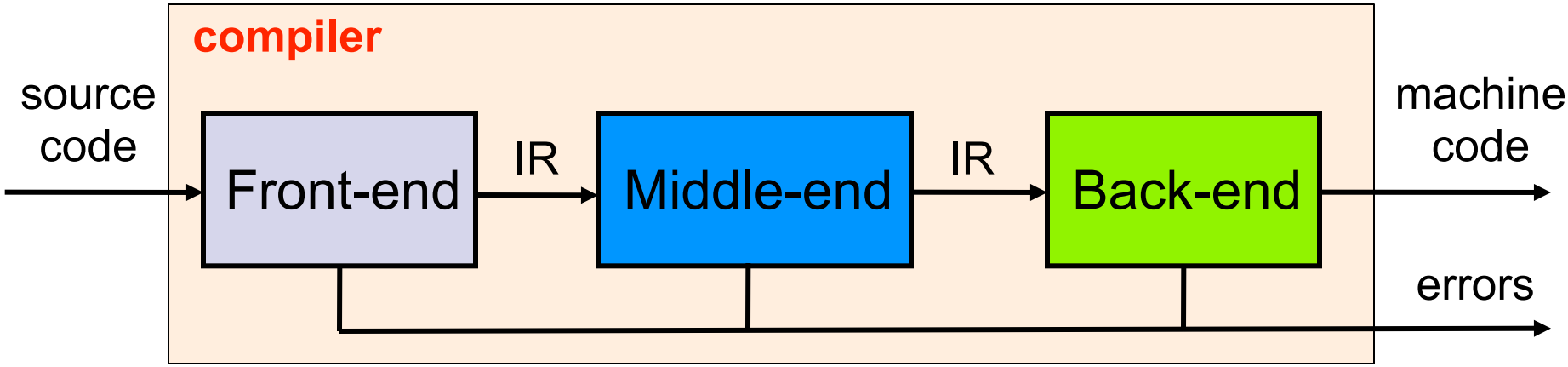


- Problem of deciding how to (re)order the instructions
- Very difficult problem that is the object of decades of research and development
- Compilers use complicated heuristics
- Some scheduling happens in hardware!
- The similarity between “source code in C” and what actually happens in the hardware is sometimes tenuous

Code Optimization

- What we've talked about so far has been known for decades
  - Some parts can be automated/generated using standard tools
    - Especially the front-end, as we'll see!
  - Some parts have to be done by hand and many well-known techniques and algorithms can be reused
- Most people who "work in compilers" today do not really work on these components
- More interesting is code optimization
- What people sometimes call the "middle-end"

Traditional 3-Pass Compiler



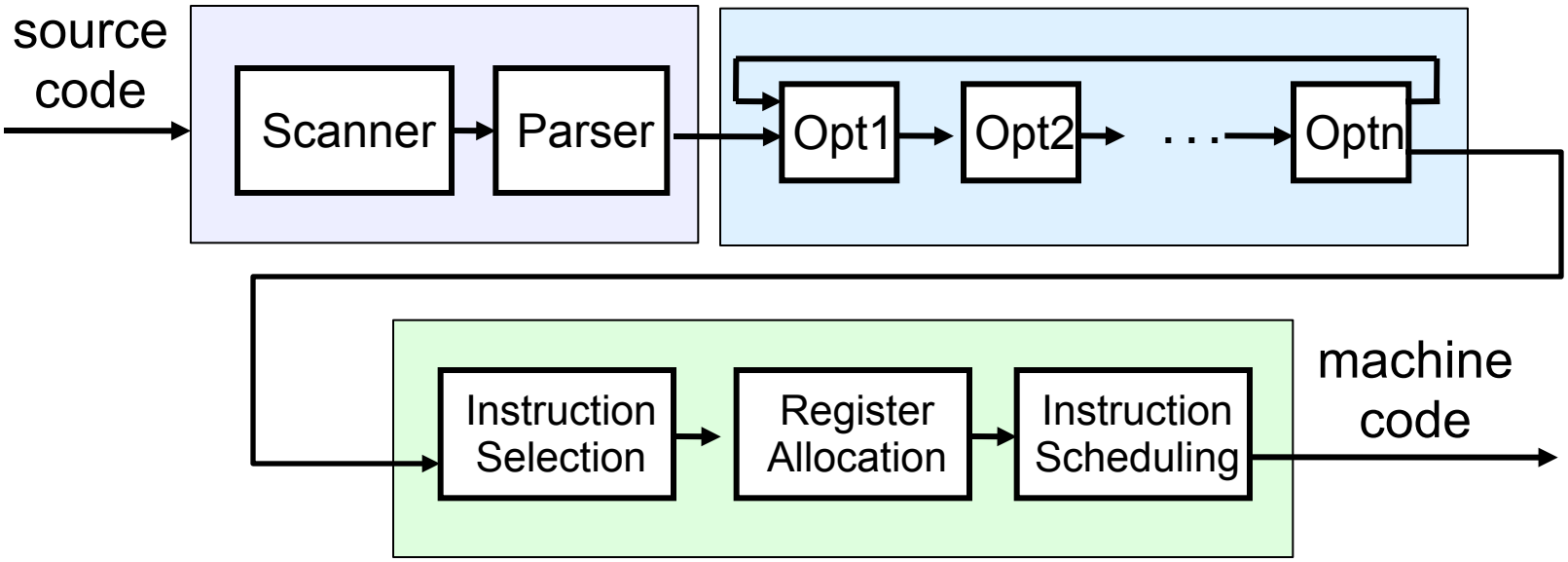
- The Middle-end is all about improving the code
- Iteratively transforms/rewrites the Intermediate Representation
- The goal: reduce the running time of the produced code
- The constraint: must preserve the exact behavior of the code

- There are entire graduate courses on just the Middle-end component

What about Interpreters?

- Not all programming languages are compiled, some are interpreted (Java, Python, JavaScript)
- An interpreter has a front-end very close to what compilers have
  - Code is translated to an intermediate language (Java to .class byte code, Python to byte .pyc byte code)
- The backend is very different: instead of generating machine code, a “virtual machine” executes the code as software
  - e.g., the JVM is a C++ program that knows how to execute byte code, same story for the CPython implementation of Python
- Modern interpreters actually do compilation on the fly, called Just In Time (JIT) compilation, so that the code can run fast
- Overall: interpreters ease development/debugging/portability, while compilers are used for absolute code performance

Conclusion



- Compilers are very complex (and interesting!) pieces of software, which we all take for granted
  - Unless you were writing code a long time ago