Compiler Overview

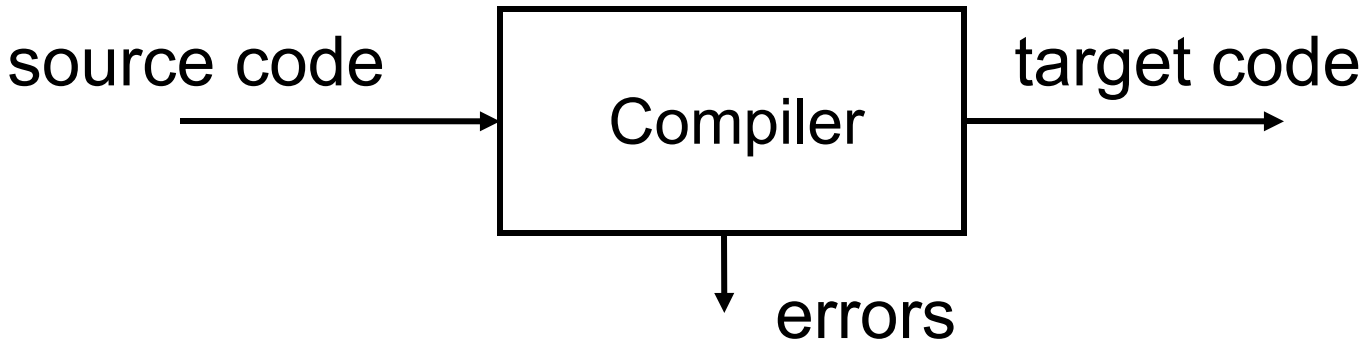
ICS312

Machine-Level and Systems Programming

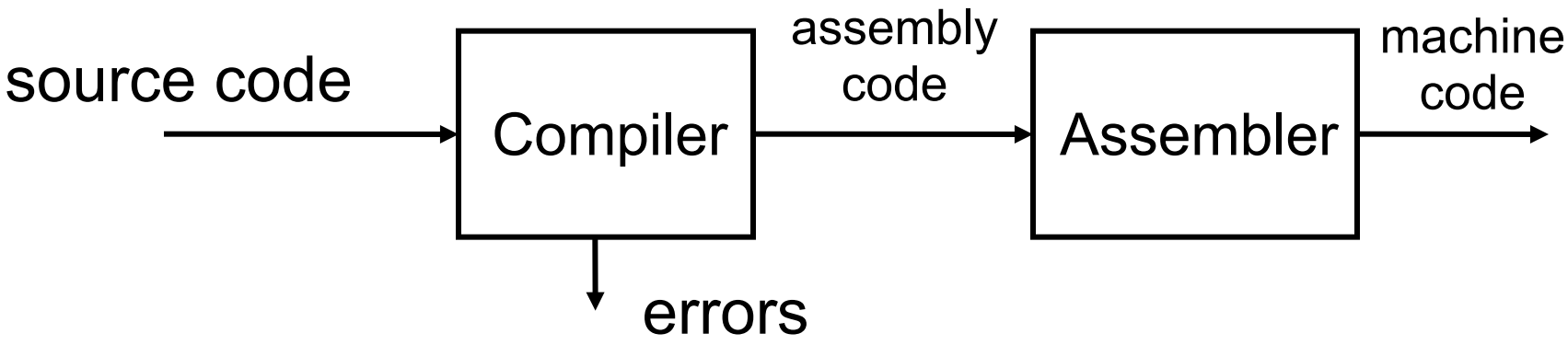
Henri Casanova (henric@hawaii.edu)

What's a compiler

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



- The target code is often assembly



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

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```

ASSEMBLER

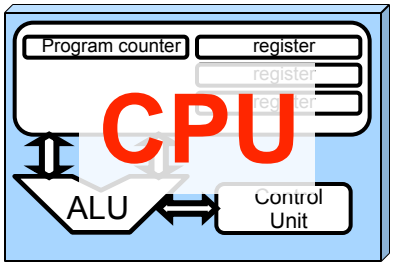
Machine code

```
01000010101011010
10101010111101010
10100101010100010
10101010101001010
11110000101010010
00010101011101011
01000000001000010
00001000100010011
10100101001010101
00010101001001010
01010101010101010
10101010111101010
10101010101001010
11110000101010010
```

COMPILER

Assembly code

```
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
add $t0, $t1, $zero
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```

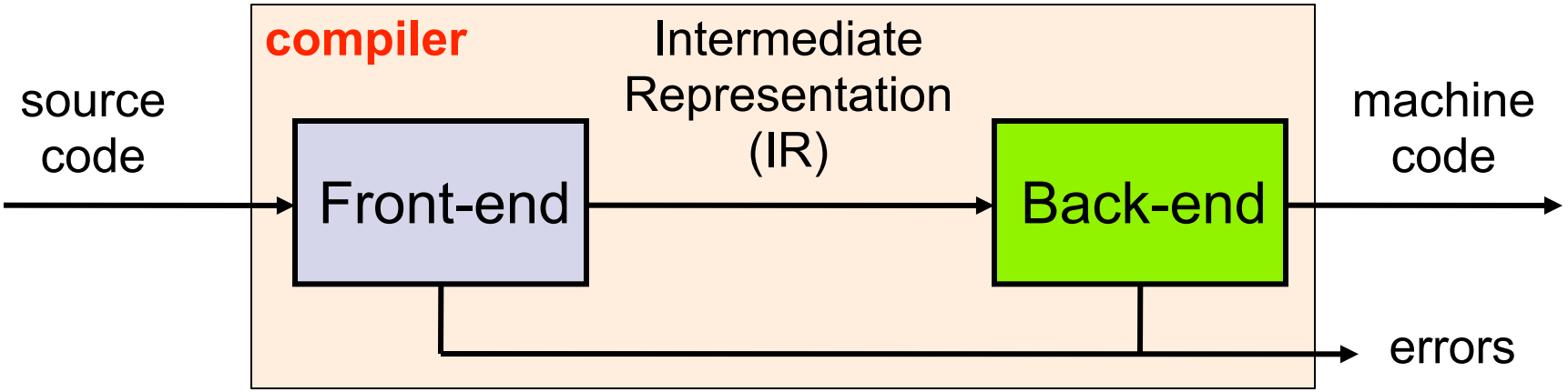


What Should a Compiler Do?

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

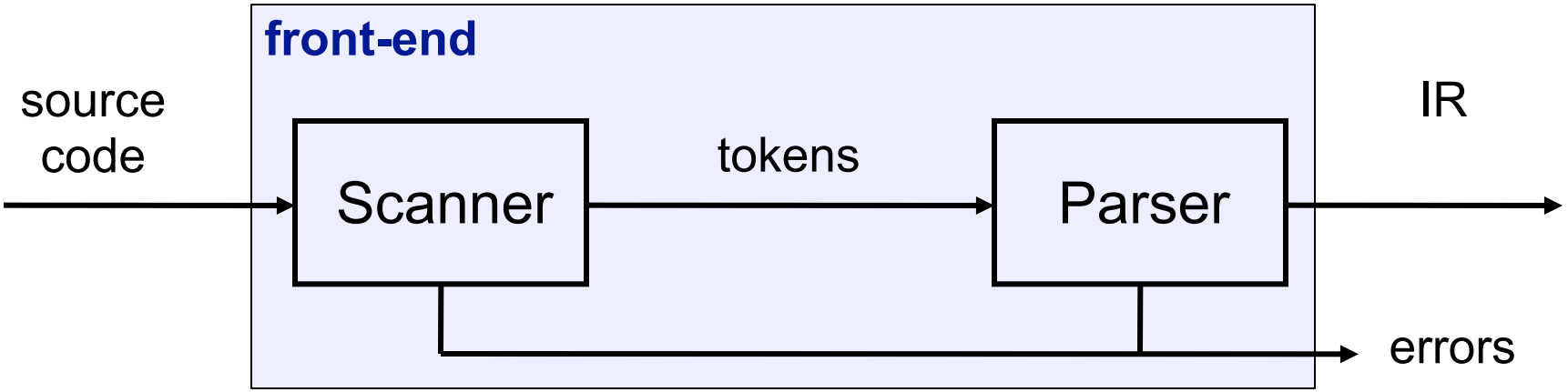
- Although these seem obvious, it wasn't always easy and the first compilers were known to have bugs and limitations
  - Good luck then trying to figure out if a bug is in your code or in the compiler!
  - Still to this day you'll hear people say "I think it's a bug in the compiler" (it's 0.000….001%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

Traditional 2-Pass Compiler



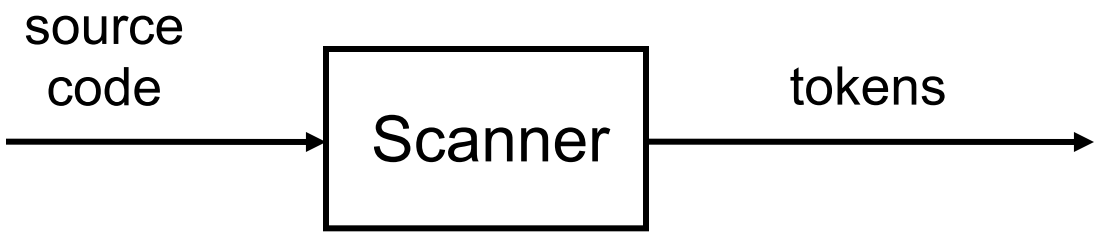
- Compilers use an Intermediate Representation (IR) for the program being compiled
  - Some abstract way to encode the program, essentially a set of tree-like data structures with bells and whistles
- Makes it possible to have multiple front-end versions
  - You could have a front-end that takes in C++, and a front-end that takes in Python, and have 2 compilers for the price of 1.5
  - Doesn't generalize to us having a single back-end in the world since, for instance, there are different architectures

What does the Front-End do?



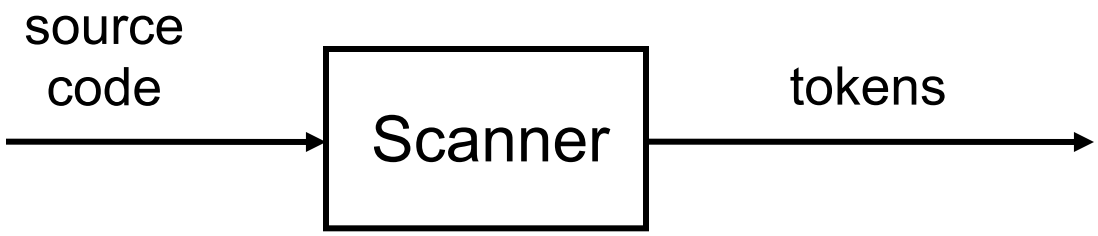
- The front-end is the “easy” (i.e., well-understood for many decades) part of the compiler
- It’s where all the error messages are generated
- Much of the front-end can be automated, and we have well-known tools to generate it
- In practice, some compilers use “implemented by-hand” Scanners or (more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

What does the Scanner Do?



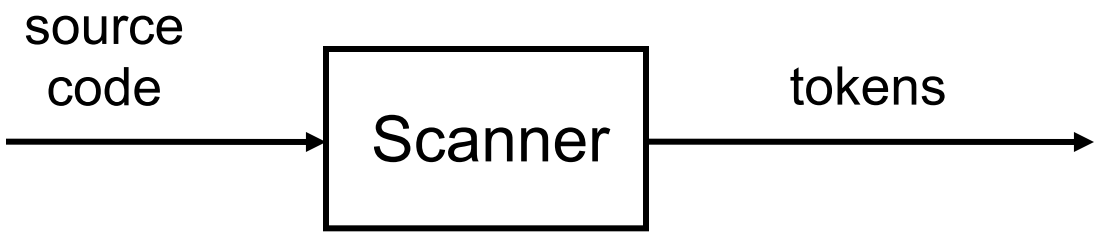
- The Scanner maps a stream of characters (ASCII codes of the characters in the text file that contains your program) into words
- A "word" is called a token, which is really a pair of two things
  - A lexeme: the actual string in the source code
  - A type: what does this word mean in the programming language?
    - Different from the types in the language like "int", "char"

What does the Scanner Do?



- Example:
- Source code: “x = x + stuff - 2”
- Will generate 7 tokens, which could look like this
  - (“x”, IDENTIFIER)
  - (“=“, ASSIGNMENT_OPERATOR)
  - (“x”, IDENTIFIER)
  - (“+”, ADD_OPERATOR)
  - (“stuff”, IDENTIFIER)
  - (“-”, SUB_OPERATOR)
  - (“2”, NUMBER)

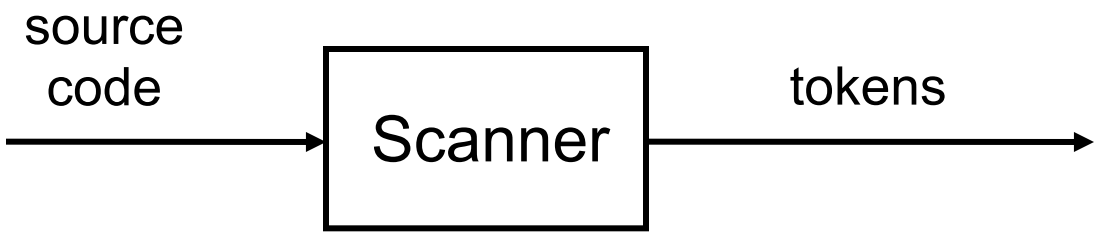
What does the Scanner Do?



- Example:
- Source code: “x = x + stuff - 2”
- Will generate 7 tokens, which could look like this
  - (“x”, IDENTIFIER)
  - (“=“, ASSIGNMENT_OPERATOR)
  - (“x”, IDENTIFIER)
  - (“+”, ADD_OPERATOR)
  - (“stuff”, IDENTIFIER)
  - (“-”, SUB_OPERATOR)
  - (“2”, NUMBER)

Each of these tokens has (probably) only one valid lexeme

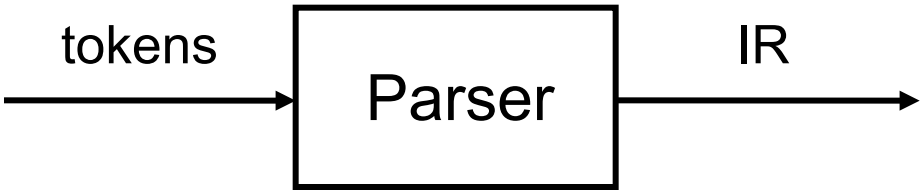
What does the Scanner Do?



- Example:
- Source code: “x = x + stuff - 2”
- Will generate 7 tokens, which could look like this
  - (“x”, IDENTIFIER)
  - (“=“, ASSIGNMENT_OPERATOR)
  - (“x”, IDENTIFIER)
  - (“+”, ADD_OPERATOR)
  - (“stuff”, IDENTIFIER)
  - (“-”, SUB_OPERATOR)
  - (“2”, NUMBER)

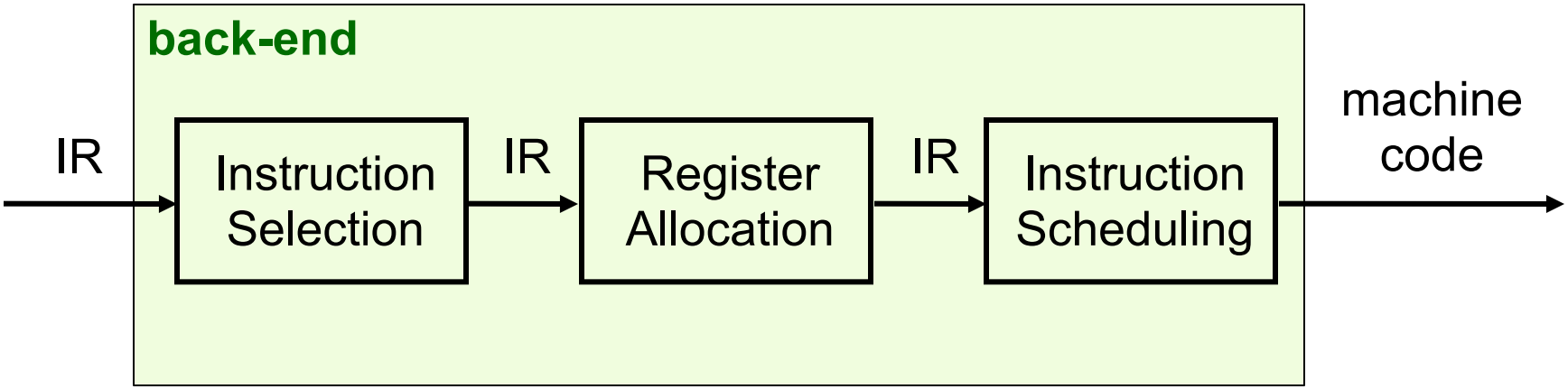
Each of these tokens has many valid lexemes

What does the Parser do?



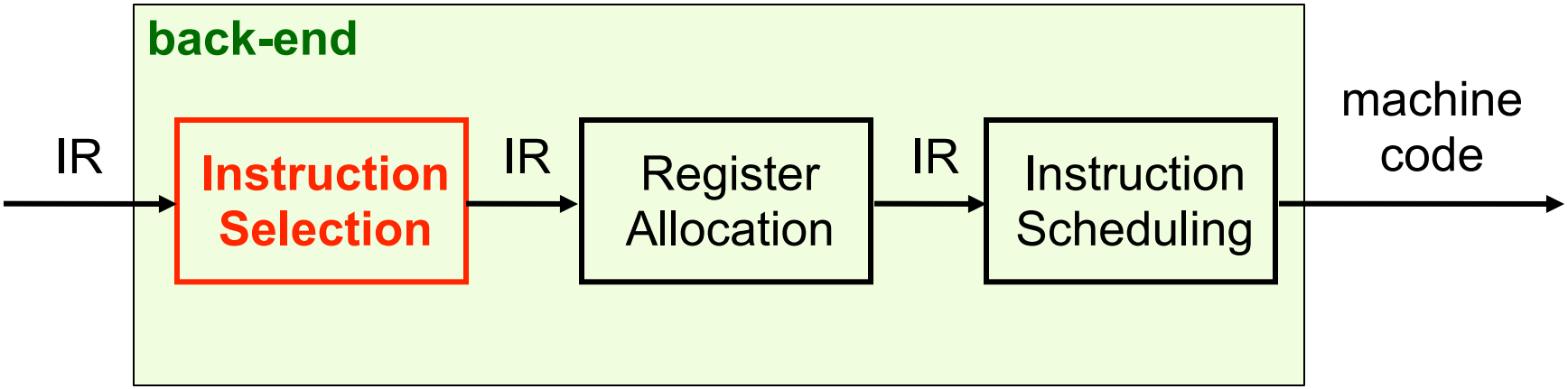
- Recognizes whether the stream of tokens matches the grammar of the language
- In the end, builds an Intermediate Representation (IR), which in this course we can view as an annotated hierarchical view of the programs (known as an abstract syntax tree)
  - We'll look at this in another set of lecture notes
- It's not as simple as the lexer

What does the Back-End do?



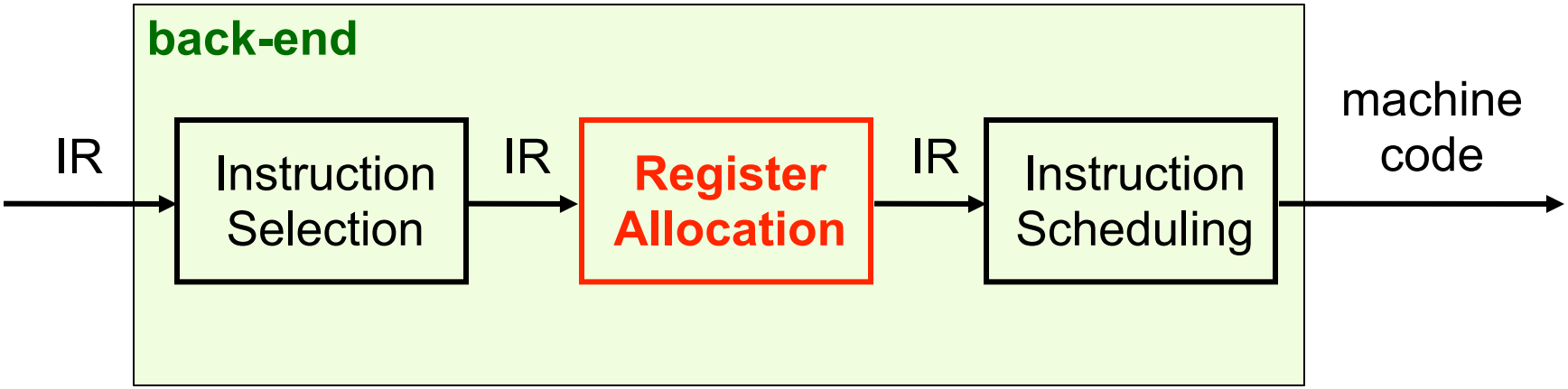
- The back-end translates the IR into machine code
  - It chooses which machine instructions to use to translate the IR
  - It chooses which values should be kept in registers
  - It decides of the order in which instructions should be executed in which order

Instruction Selection



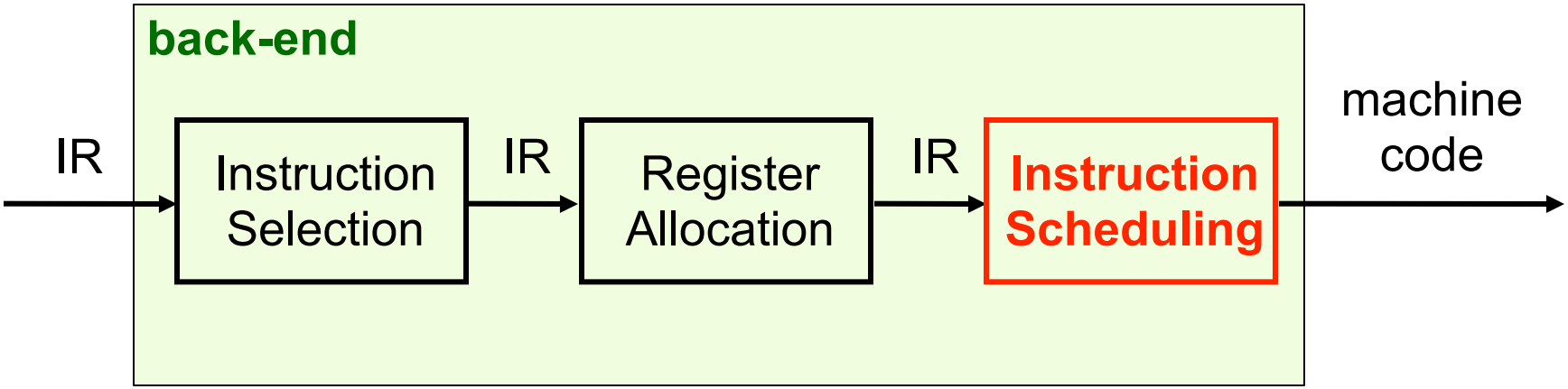
- The goal is to produce fast, compact code
  - E.g., use an “xor eax, eax” rather than “mov eax, 0”
- This used to be a huge issue when ISAs were very complicated with many, many options
  - E.g., VAX
- No longer as big an issue now

Register Allocation



- Registers allow for fast variable access
- But there is a limited number of them
- Optimal allocation is known to be 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

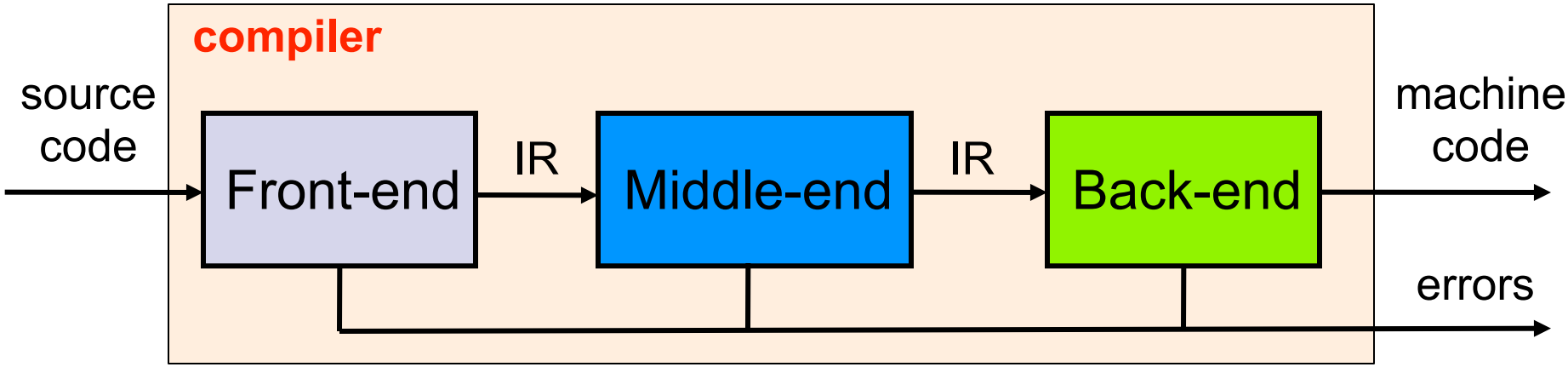


- Problem of deciding how to (re)order the instructions
- Very difficult problem that is the object of decades of research and development
- Compilers use complicated heuristics
- Some scheduling happens in hardware!
- The similarity between “source code in C” and what actually happens in the hardware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

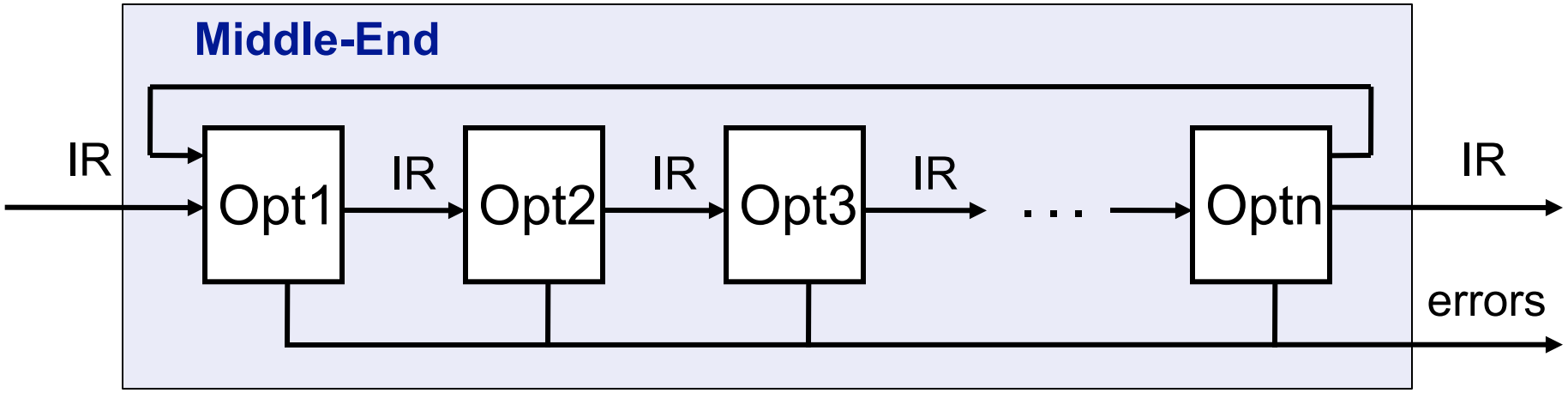
Traditional 3-Pass Compiler



- 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

Typical Middle-End



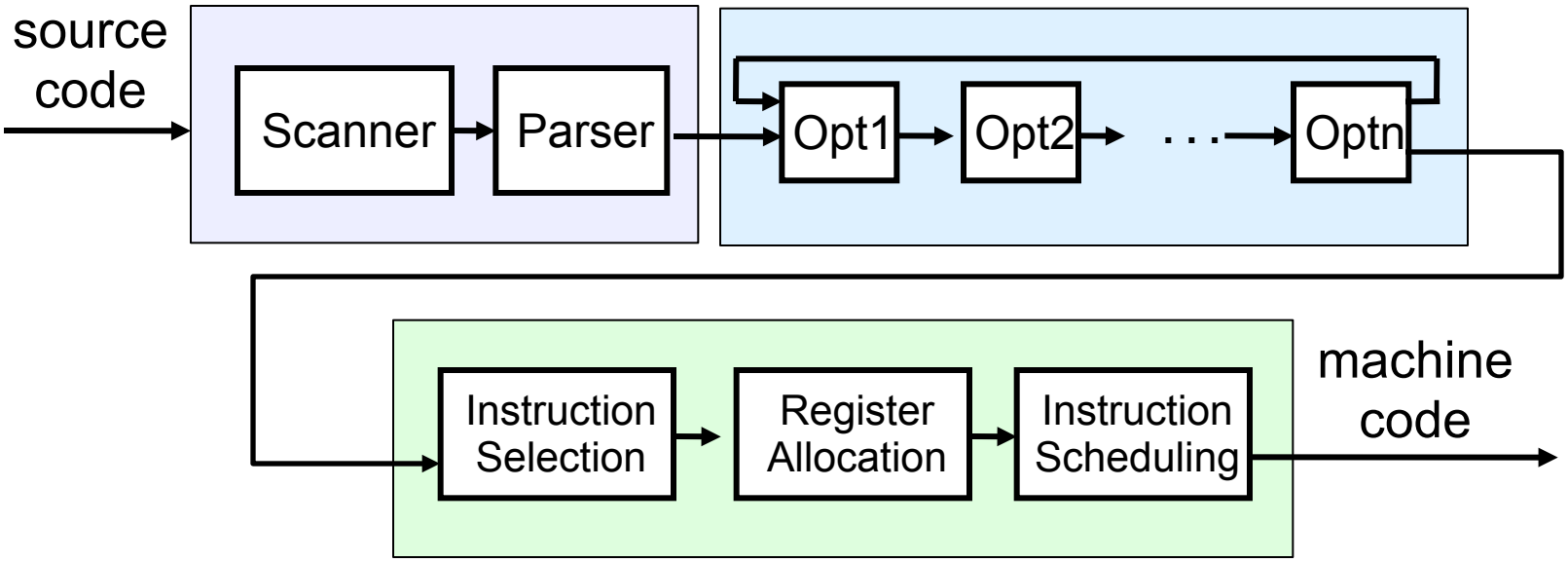
- The Middle-end is a series of optimizations
- Typical transformations
  - Discover a redundant computation and remove it

```
        mov     eax, 12
        mov     eax, 8
```

  - Discover “dead code” and remove it

```
        jmp     foo
        mov     eax, 12
foo:    . . .
```

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