

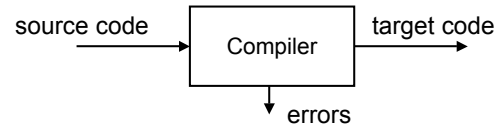
# Compiler Overview

## ICS312 - Spring 2009 Machine-Level and Systems Programming

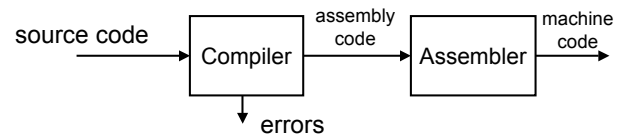
Henri Casanova (henric@hawaii.edu)

## What's a compiler

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



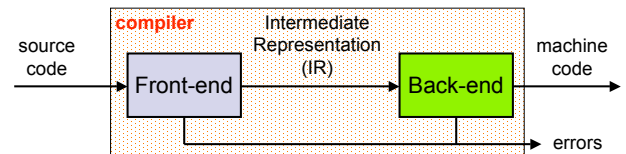
- The target code is typically assembly and then machine code



## What Should a Compiler Do?

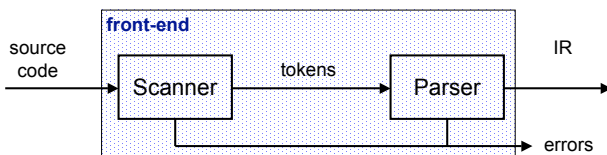
- It should **translate legal code** and **reject illegal code** with (hopefully helpful) error messages
- It should generate correct code
  - Correct text segments
- It should manage storage for all variables
  - Correct data segments
- Must comply with whatever object file format the system expects (see previous lecture)
- Although these seem obvious, it wasn't always easy and the first days of compilers were fraught with peril

## Traditional 2-Pass Compiler



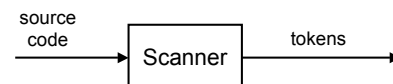
- Compilers use an **Intermediate Representation (IR)** for the program being compiled
- Makes it possible to have multiple front-end versions
  - You could have a front-end that takes in C++, and a front-end that takes in Python, and have 2 compilers for the price of 1.5
  - Limited to how general the IR is
  - Doesn't generalize to us having a single back-end in the world!
- Makes it possible to have multiple back-end passes to try to generate better and better code

## What does the Front-End do?



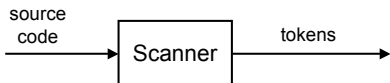
- The front-end is the "easy" part of the compiler
- It's where all the error messages are generated
- Much of the front-end can be automated, and we have well-known tools to generate it
- In practice, some compilers use "implemented by-hand" Scanners or (mor rarely) Parsers, for speed

## What does the Scanner Do?



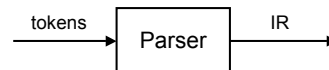
- The Scanner maps a **stream of characters** (ASCII codes of the characters in the text file that contains your program) into **words**
- These "words" are called **tokens**, which are really a pair of two things
  - A **lexeme**: the actual string
  - A **type**: what does this mean in the programming language? (also called a "token")
    - Different from the types in the languages like "int", "double", etc.

## What does the Scanner Do?



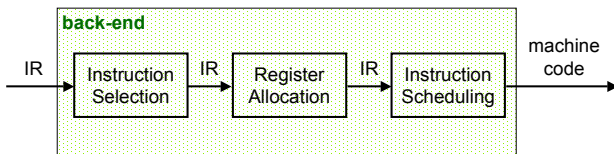
- Example:
- Source code: "x = x + y - 2"
- Will generate 7 tokens, which could look like this
  - ("x", IDENTIFIER)
  - ("=", ASSIGNMENT\_OP)
  - ("x", IDENTIFIER)
  - ("+", ADD\_OP)
  - ("y", IDENTIFIER)
  - ("-", SUB\_OP)
  - ("2", NUMBER)

## What does the Parser do?



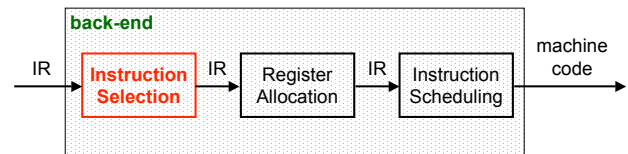
- Recognizes whether the stream of tokens match the **grammar** of the language
- In the end, builds an annotated hierarchical view of the programs called an abstract syntax tree
  - We'll look at this in another lecture

## What does the Back-End do?



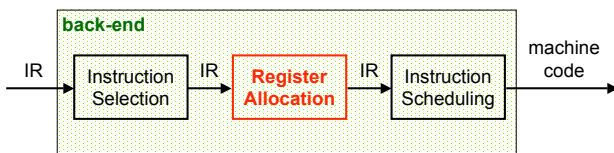
- The back-end translates the IR into machine code
  - It chooses which machine instructions to use to translate the IR
  - It chooses which values should be kept in registers
  - It decides of the order in which instructions should be executed in which order
- Back-end automation has been much less successful than for the front-end

## Instruction Selection



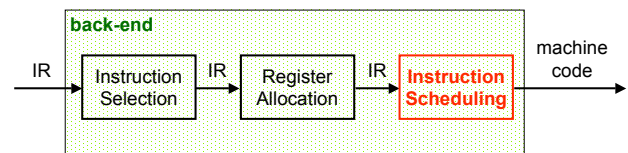
- The goals is to produce fast, compact code
  - E.g., use an "xor eax, eax" rather than "mov eax, 0"
- This used to be a huge issues when ISAs were very complicated with many, may options
  - E.g., VAX

## 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 algorithm to try to get close to the optimal

## Instruction Scheduling

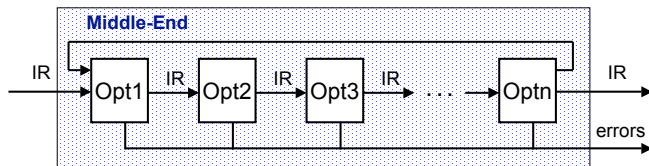


- Avoid hardware stalls and interlock
  - See ICS 431
- Use all functional units productively
  - Parallelism of ALUs
- Optimal scheduling is NP-hard
- Compilers use heuristics
  - Some scheduling happens in hardware!

## Code Optimization

- What we've talked about so far has been known for decades
  - Some parts can be automated/generated using standard tools
  - Some parts have to be done by hand by many well-known techniques and algorithms can be used
- So most people who "work in compilers" today do not really work on these components
- What most people spend their time on these days is **code optimization**
- Done in what people sometimes call the "middle-end"

## Typical Middle-End



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

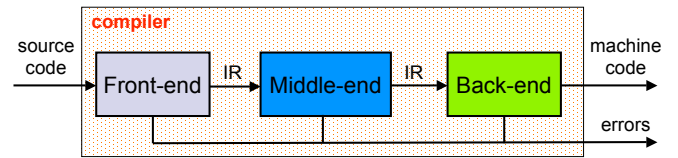
```

mov    eax, 12
mov    eax, 8
                    
```
  - Discover "dead code" and remove it
 

```

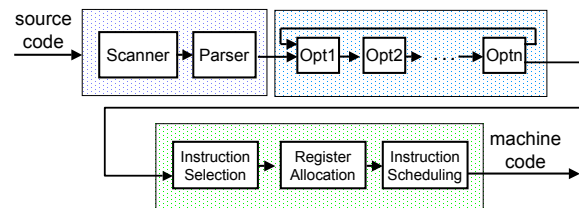
jmp    foo
mov    eax, 12
foo:  ...
                    
```
  - etc.

## 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 meaning of the code
- There are entire graduate courses on just the Middle-end component

## Conclusion



- Compilers are complex pieces of software, which we often take for granted