Compiler Construction

Important facts:

Name:	Prof. Dr. Peter Thiemann
Email:	`thiemann@informatik.uni-freiburg.de`
Office:	079-00-015

Exercises:

Name:	Leonardo Mieschendahl, MSc
Email:	`mieschel@informatik.uni-freiburg.de`
Office:	079-00-014
Name:	Marius Weidner, BSc
Email:	`weidner@informatik.uni-freiburg.de`
Office:	079-00-014

Studienleistung	homeworks
Grades	final exam

Things to do

- read Siek chapter 1
- make sure you have a working GitHub account
- brush up on Python
- review development tools
- find the official course webpage

  `https://proglang.github.io/teaching/25ss/cc.html`

Compilers

What is an interpreter?

- a program that reads a program and its input and produces the results of running that program

What is a compiler?

- a program that reads a *source* program in a language and translates it into a *target* program in another language
- we expect the source program and the target program to behave in the same way
- usually, we expect the target program to be better (e.g., faster), in some way, than the source program

This course deals mainly with *compilers*

Many of the same issues arise in *interpreters* and we will use interpreters to specify the (source) language

Motivation

Why study compiler construction?

Why build compilers?

Why attend class?

Interest

Compiler construction is a microcosm of computer science

algorithms

graph algorithms, union-find, dynamic programming

theory

DFAs for scanning, parser generators, lattice theory

systems

allocation and naming, locality, synchronization

architecture

pipeline management, hierarchy management, instruction set use

Inside a compiler, all these things come together

Isn’t it a solved problem?

Machines are constantly changing

Changes in architecture $\Rightarrow$ changes in compilers

- new features pose new problems (e.g., vector instructions, ML specific instructions, peculiarities of architectures)
- changing costs lead to different concerns
- old solutions need re-engineering

Intrinsic Merit

Compiler construction is challenging and fun

- interesting problems
- primary responsibility for performance (*blame*)
- new architectures ⇒ new challenges
- *real* results
- extremely complex interactions

Compilers have an impact on how computers are used

Some of the most interesting problems in computing

Experience

You have used several compilers

What qualities are important in a compiler?

Experience

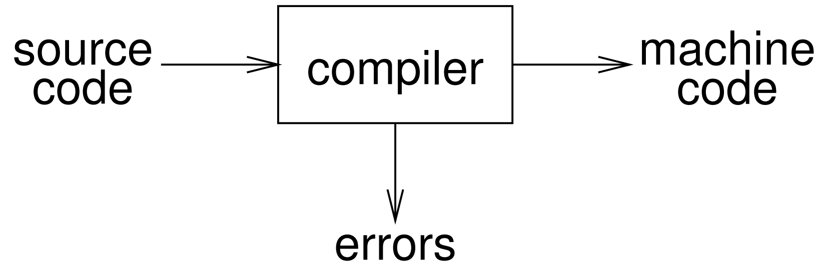
You have used several compilers

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Each of these shapes your expectations about this course

Abstract view

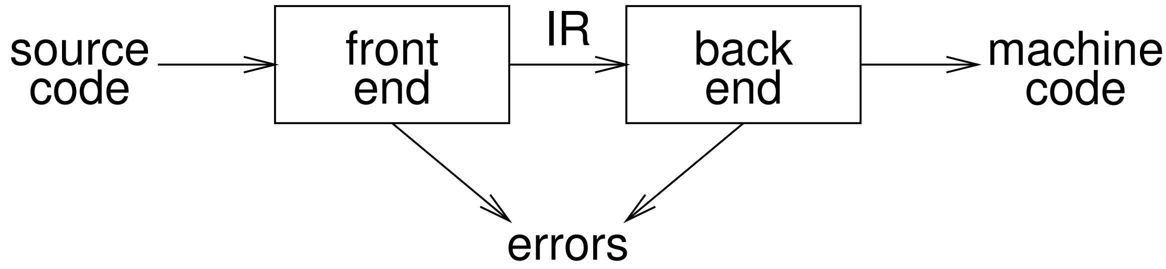


Implications:

- recognize legal programs / reject illegal programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

Big step up from assembler — higher level notations

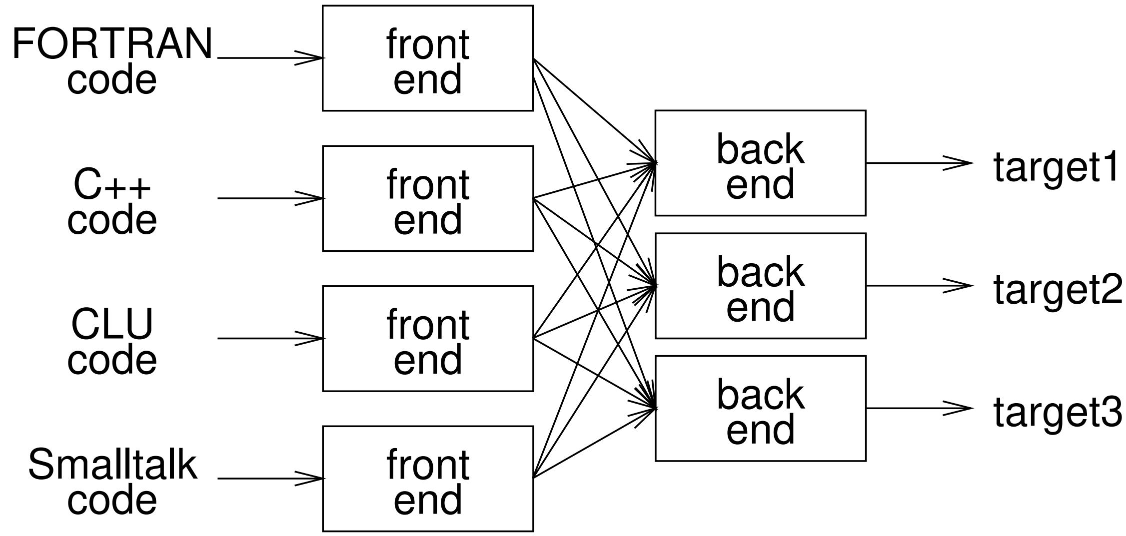
Traditional two pass compiler



Implications:

- intermediate representation (IR)
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes $\Rightarrow$ better code

A fallacy

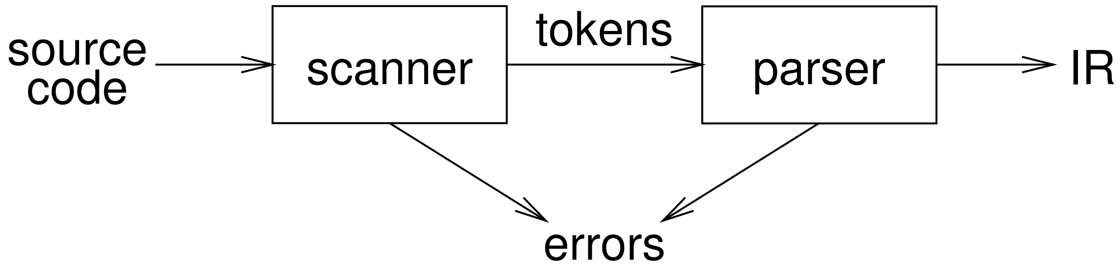


Can we build $n \times m$ compilers with $n + m$ components?

- must encode *all* the knowledge in each front end
- must represent *all* the features in one IR
- must handle *all* the features in each back end

Limited success with low-level IRs — but LLVM

Front end

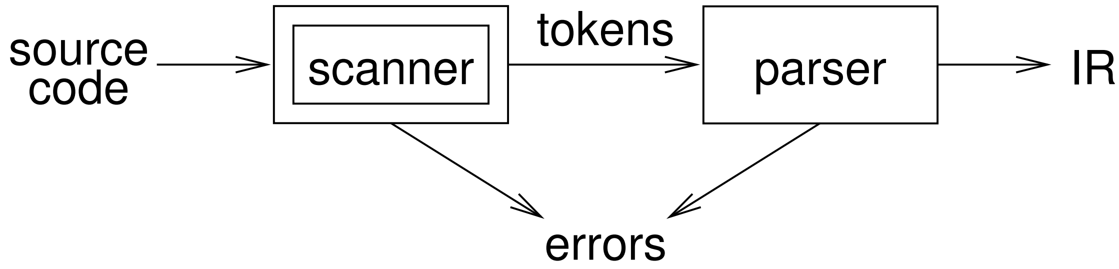


Responsibilities:

- recognize legal programs
- report errors (lexical and syntactic)
- produce IR
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Front end



Scanner:

- maps characters into *tokens* – the basic unit of syntax

  `x = y + 31;`

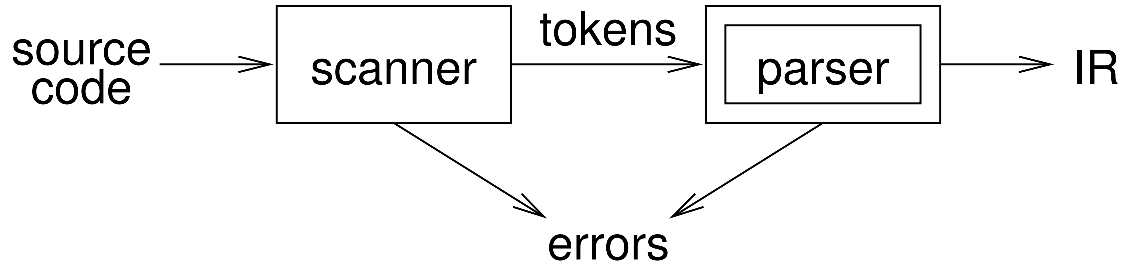
  becomes

  <id, `x`> = <id, `y`> + <number, 31> `;`

- character string value for a *token* is a *lexeme*: `'x'`, `'='`, `'+'`, `'y'`, `'31'`
- typical tokens: *number*, *id*, `+`, `-`, `*`, `/`, `do`, `end`
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed

  $\Rightarrow$ use specialized recognizer (as opposed to `lex`)

Front end



Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct IR(s)
- produce meaningful error messages
- attempt error correction

Parser generators mechanize much of the work

Front end

Context-free syntax is specified with a *grammar*

<sheep noise>	::=	`baa`
	$\mid$	`baa` <sheep noise>

The noises sheep make under normal circumstances

This format for a grammar is called *Backus-Naur form* (BNF)

Formally, a grammar $G = (N, T, P, S)$ where

N is a set of *non-terminal symbols*
T is a set of *terminal symbols*
P is a set of *productions* or *rewrite rules*
    $(P : N \rightarrow (N \cup T)^*)$
S is the *start symbol*

Front end

Context free syntax can be put to better use

1	<goal>	::=	<expr>
2	<expr>	::=	<expr> <op> <term>
3		|	<term>
4	<term>	::=	`number`
5		|	`id`
6	<op>	::=	`+`
7		|	`-`

Simple expressions with addition and subtraction over tokens `id`*,* `number`*,* `+`*, and* `-`

S = *<goal>*
T = `number`, `id`, `+`, `-`
N = *<goal>*, *<expr>*, *<term>*, *<op>*
P = 1, 2, 3, 4, 5, 6, 7

Front end

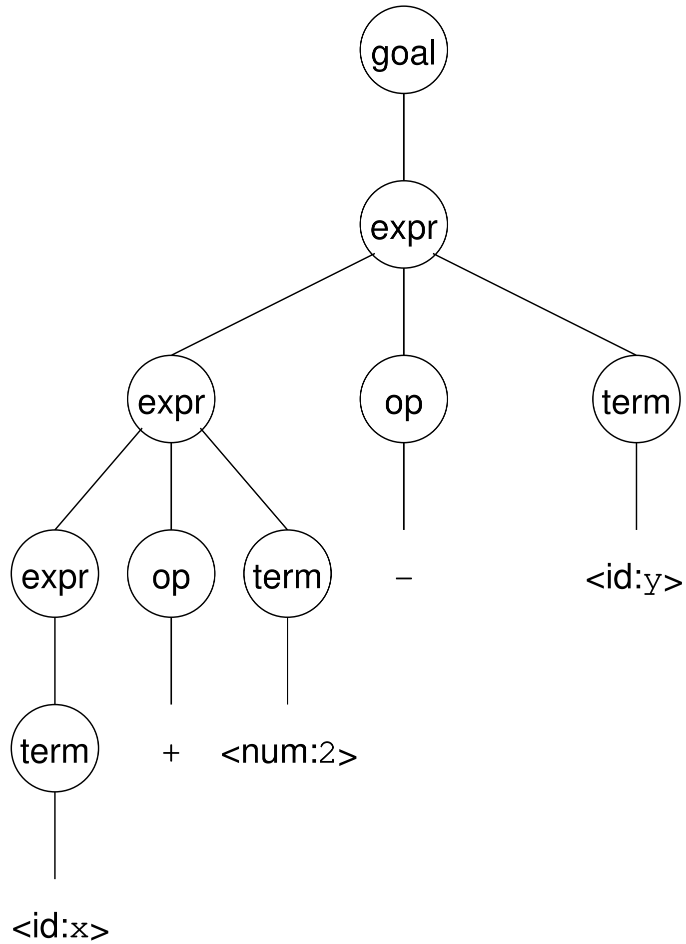
Given a grammar, valid sentences can be derived by repeated substitution.

Prod'n.	Result
	<goal>
1	<expr>
2	<expr> <op> <term>
5	<expr> <op> `y`
7	<expr> `- y`
2	<expr> <op> <term> `- y`
4	<expr> <op> `2 - y`
6	<expr> `+ 2 - y`
3	<term> `+ 2 - y`
5	`x + 2 - y`

To recognize a valid sentence in some CFG, we reverse this process and build up a *parse*

Front end

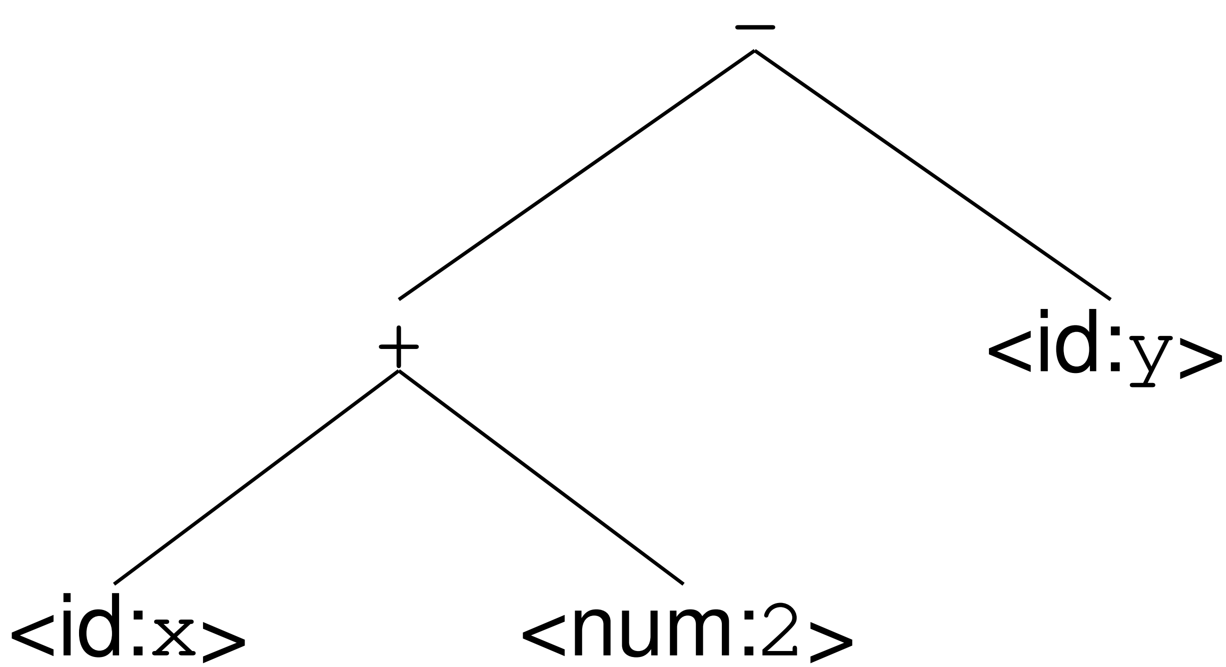
A parse can be represented by a *parse*, or *syntax*, tree



A parse tree contains a lot of unnecessary information

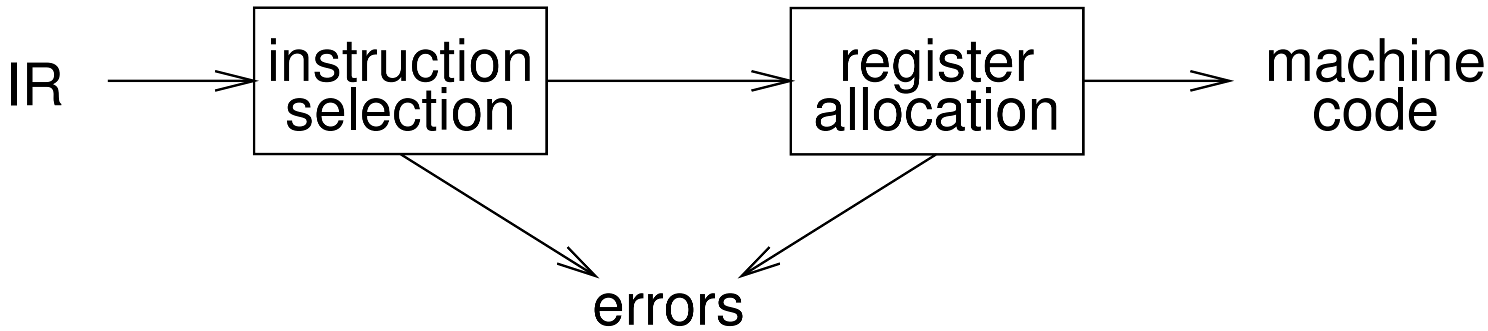
Front end

Internally compilers use a more concise *abstract syntax tree*



Abstract syntax trees (ASTs) are often used as an IR between front end and back end

Back end

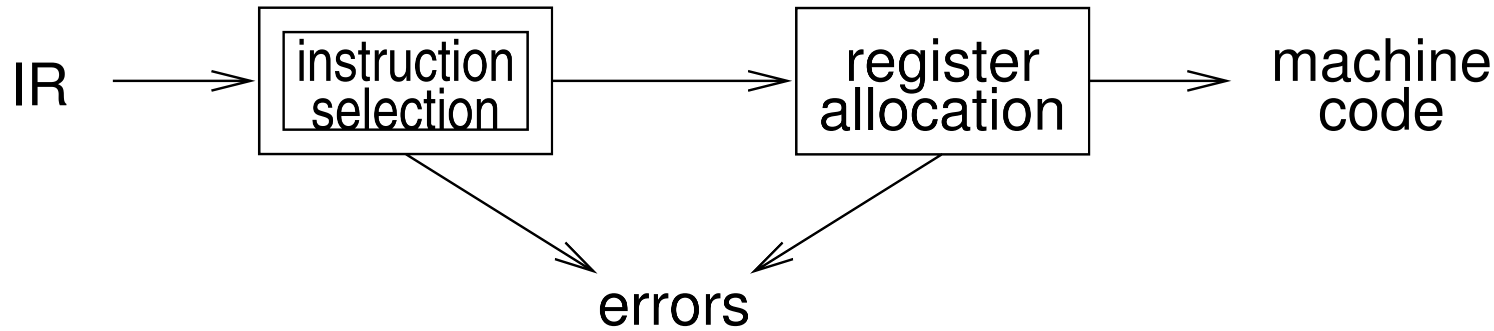


Responsibilities

- translate IR into target machine code
- choose instructions for each IR operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

Automation has been less successful here

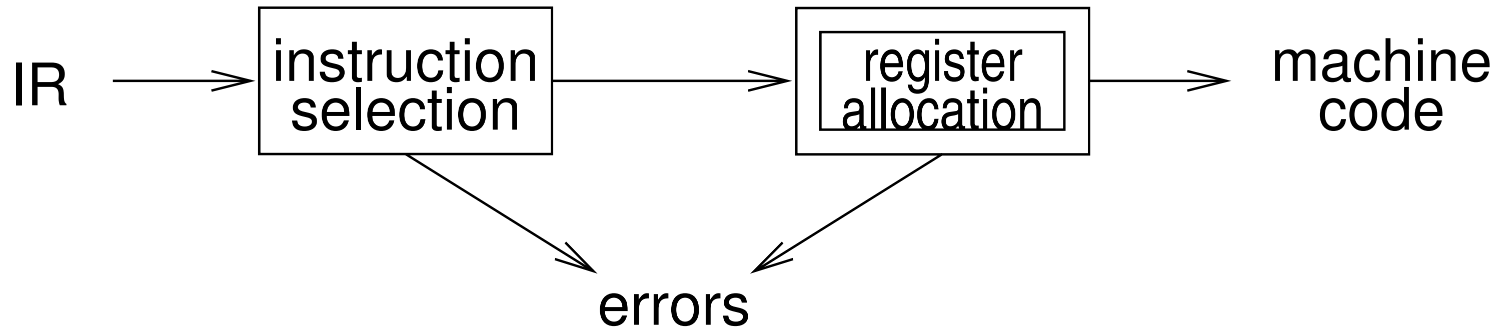
Back end



Instruction selection:

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - *ad hoc* techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming
- ... but much simpler for modern RISC architectures

Back end

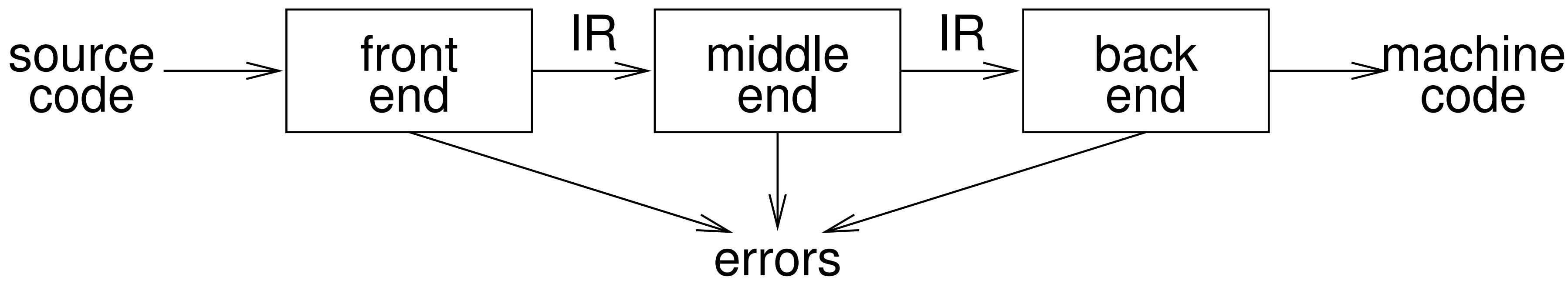


Register Allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult

Modern allocators often rely on graph coloring

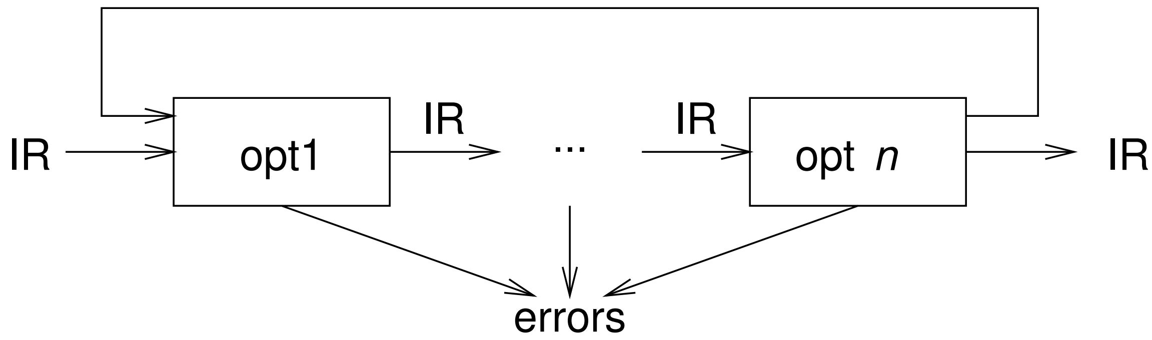
Traditional three pass compiler



Code Improvement

- analyzes and changes IR
- goal is to reduce runtime
- must preserve values

Optimizer (middle end)



Modern optimizers are usually built as a set of passes

Typical passes

- constant propagation and folding
- code motion
- reduction of operator strength
- common subexpression elimination
- redundant store elimination
- dead code elimination

The MiniPython compiler—a sequence of languages

- $\mathcal{L}_{var}$ — integer arithmetic expressions with variables
- $\mathcal{L}_{if}$ — add booleans, conditional statements and expressions
- $\mathcal{L}_{while}$ — add loops
- $\mathcal{L}_{tup}$ — add tuples (dynamically allocated data)
- $\mathcal{L}_{Fun}$ — add top-level functions
- $\mathcal{L}_{\lambda}$ — add lambda expressions
- add generics (deviating from the book)
- $\mathcal{L}_{exc}$ — add exceptions
- a sneak peek at optimization

The MiniPython compiler—multi pass

- We start with four passes
- Each pass has a dedicated function
- Each pair of passes communicate via a dedicated, typed IR
- As we extend the language, more passes will be added