

# Compiler Construction

---

Important facts:

*Name:* Prof. Dr. Peter Thiemann

*Email:* thiemann@informatik.uni-freiburg.de

*Office:* 079-00-015

Exercises:

*Name:* Dipl.-Inform. Manuel Geffken

*Email:* geffken@informatik.uni-freiburg.de

*Office:* 079-00-014

Basis for grades:

50% homeworks

50% final exam

Copyright ©2012 by Antony L. Hosking. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from [hosking@cs.purdue.edu](mailto:hosking@cs.purdue.edu).*

# Things to do

---

- read Appel chapter 1
- make sure you have a working account
- start brushing up on Java
- review Java development tools
- find the official course webpage  
<http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2012ws/>

# Compilers

---

What is a compiler?

- a program that reads an *executable* program in one language and translates it into an *executable* program in another language
- we expect the program produced by the compiler to be better, in some way, than the original

What is an interpreter?

- a program that reads an *executable* program and produces the results of running that program
- usually, this involves executing the source program in some fashion

This course deals mainly with *compilers*

Many of the same issues arise in *interpreters*

Why study compiler construction?

Why build compilers?

Why attend class?

# Interest

---

Compiler construction is a microcosm of computer science

## **artificial intelligence**

greedy algorithms, learning algorithms

## **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
- changing costs lead to different concerns
- old solutions need re-engineering

*Changes in compilers should prompt changes in architecture*

- New languages and features

# Intrinsic Merit

---

*Compiler construction is challenging and fun*

- interesting problems
- primary responsibility for performance *(blame)*
- new architectures  $\Rightarrow$  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*

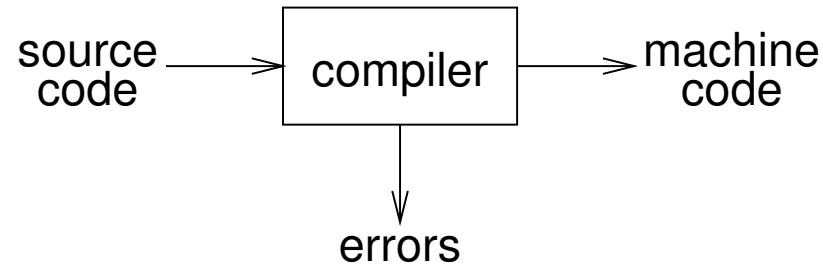
*What qualities are important in a compiler?*

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

---



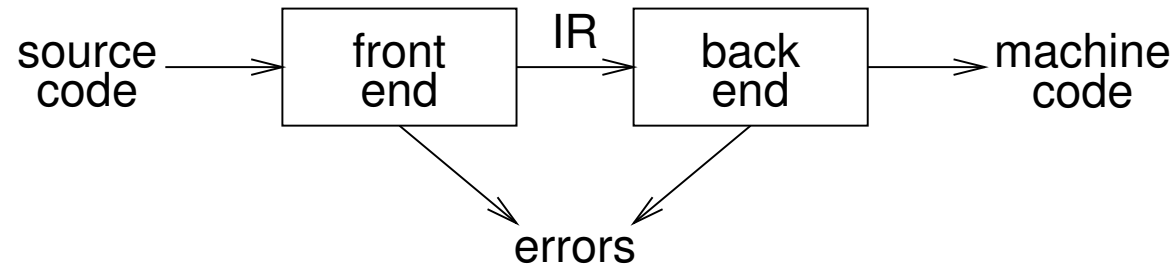
Implications:

- recognize legal (and 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

---

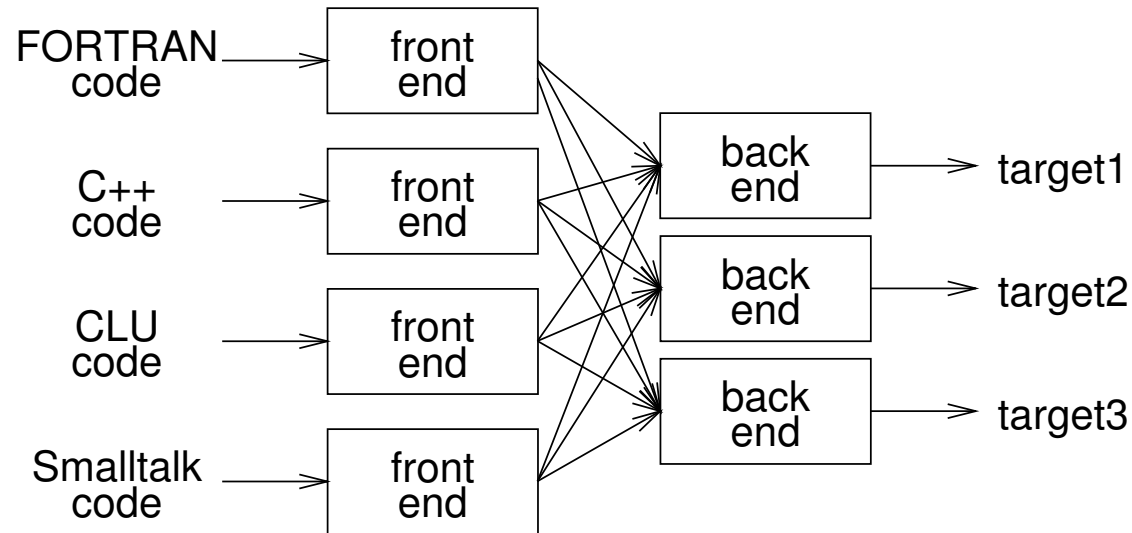


Implications:

- 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

---



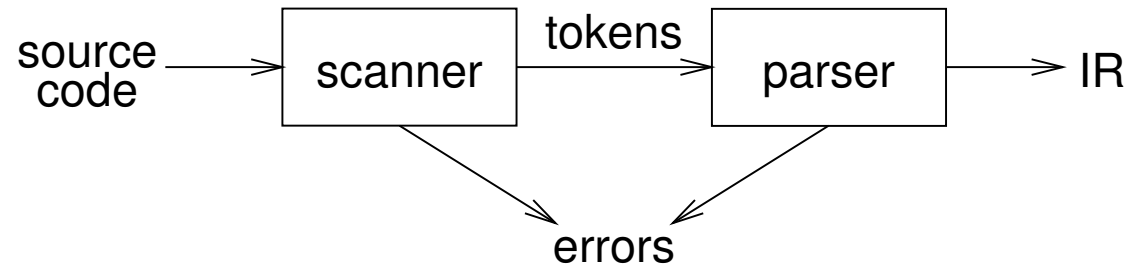
Can we build  $n \times m$  compilers with  $n + m$  components?

- 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

---



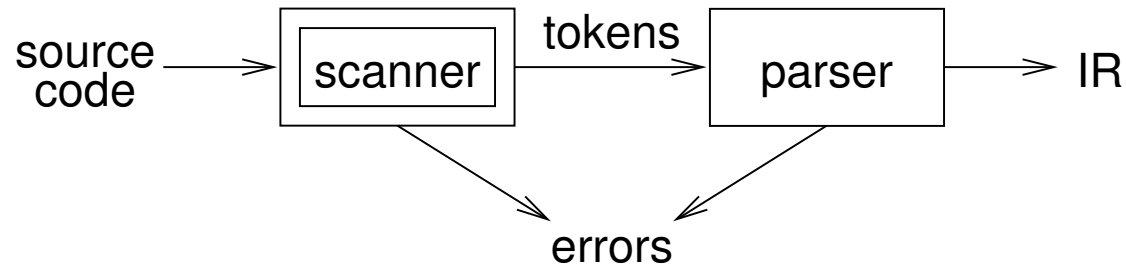
## Responsibilities:

- recognize legal procedure
- report errors
- 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 = x + y;`

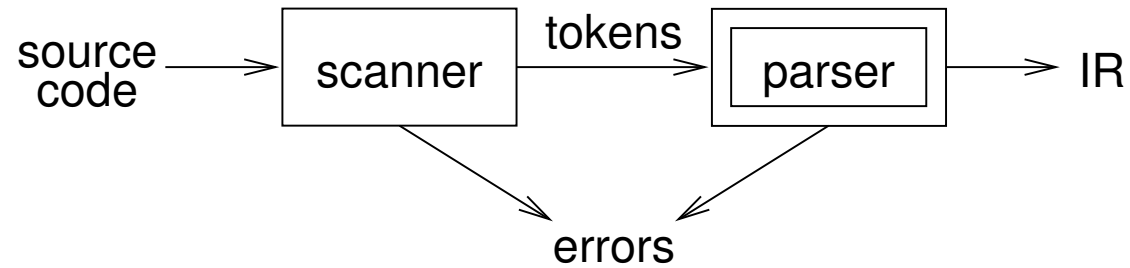
becomes

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

- character string value for a *token* is a *lexeme*
- typical tokens: *number, id, +, -, \*, /, do, end*
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed  
⇒ 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*

$$\begin{aligned} \langle \text{sheep noise} \rangle & ::= \text{baa} \\ & \quad | \text{baa} \langle \text{sheep noise} \rangle \end{aligned}$$

*The noises sheep make under normal circumstances*

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

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

$S$  is the *start symbol*

$N$  is a set of *non-terminal symbols*

$T$  is a set of *terminal symbols*

$P$  is a set of *productions or rewrite rules*

$(P : N \rightarrow N \cup T)$



# 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 and number*

$S = \langle goal \rangle$

$T = \text{number, id, +, -}$

$N = \langle goal \rangle, \langle expr \rangle, \langle term \rangle, \langle op \rangle$

$P = 1, 2, 3, 4, 5, 6, 7$

# Front end

---

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

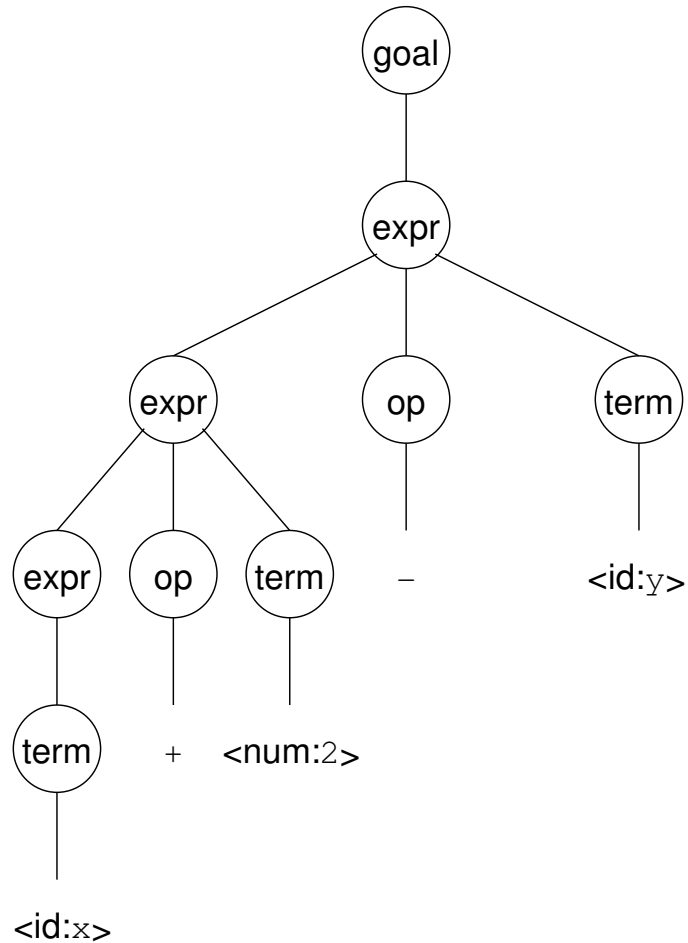
Prod'n.	Result
	<goal>
1	<expr>
2	<expr> <op> <term>
5	<expr> <op> y
7	<expr> - y
2	<expr> <op> <term> - y
4	<expr> <op> 2 - y
6	<expr> + 2 - y
3	<term> + 2 - y
5	x + 2 - y

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

# Front end

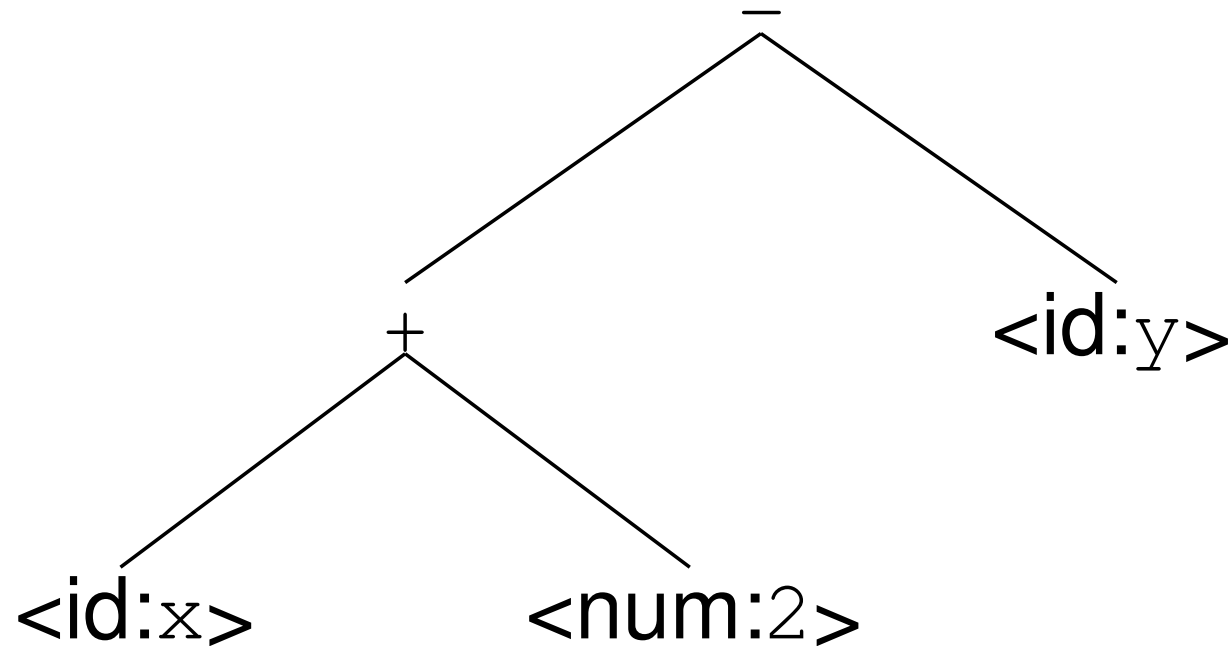
---

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



A parse tree contains a lot of unnecessary information

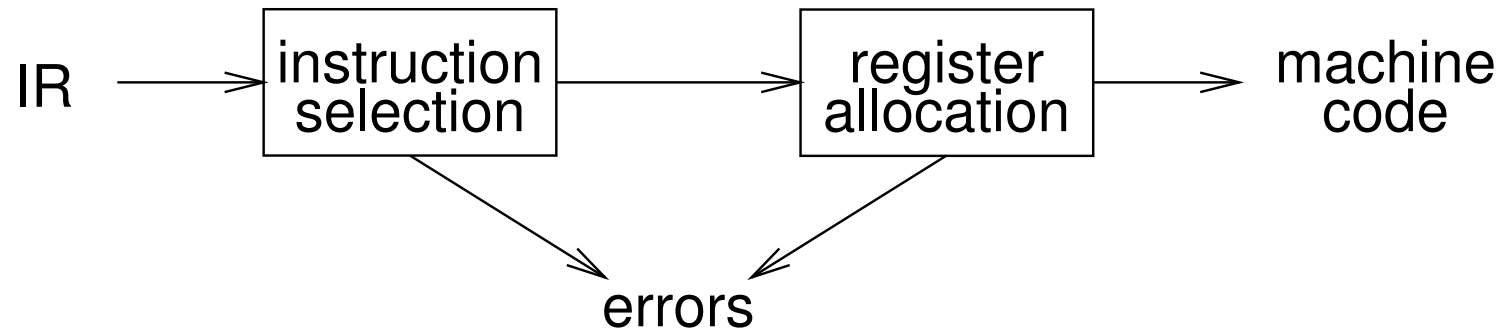
So, compilers often use a more concise *abstract syntax tree*



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

# Back end

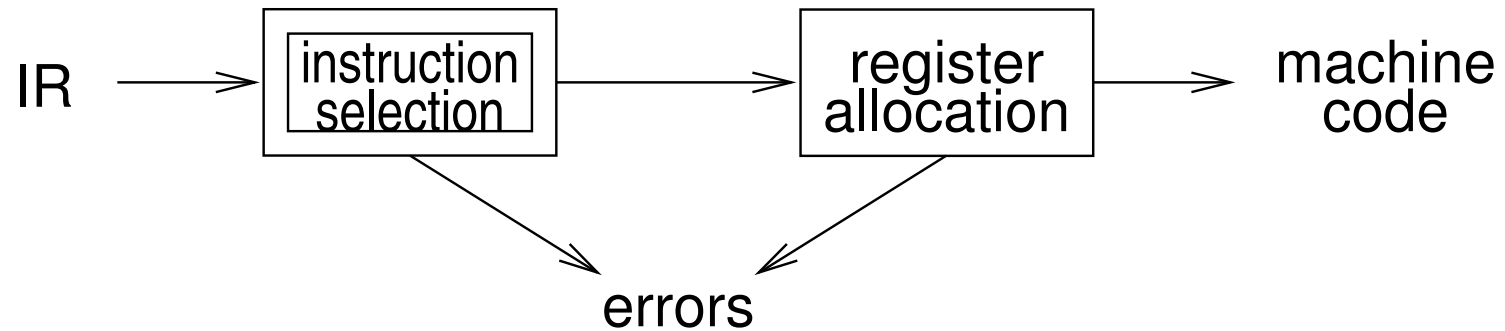
---



## Responsibilities

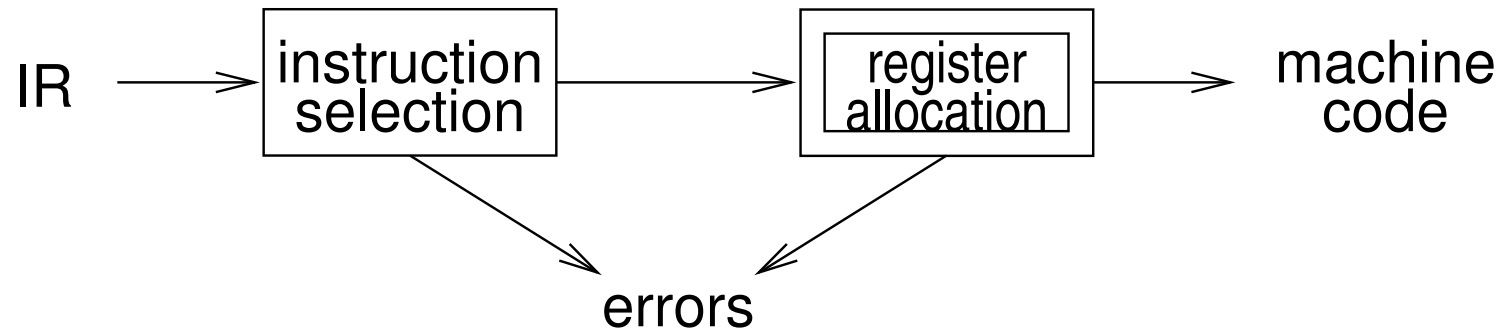
- translate IR into target machine code
- choose instructions for each IR operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

*Automation has been less successful here*



Instruction selection:

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - *ad hoc* techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming



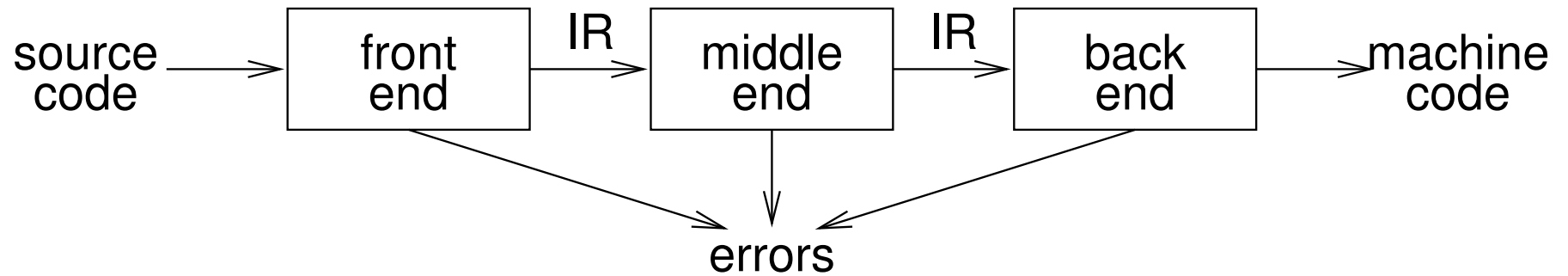
## Register Allocation:

- 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

---



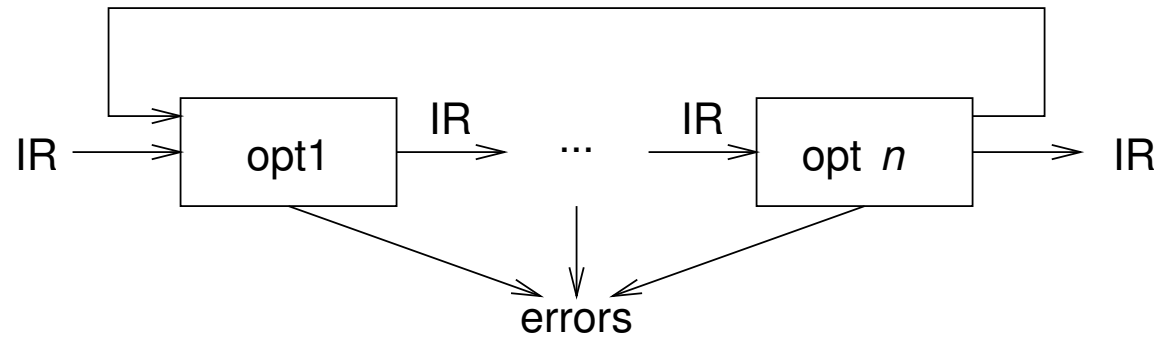
## Code Improvement

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



# Optimizer (middle end)

---

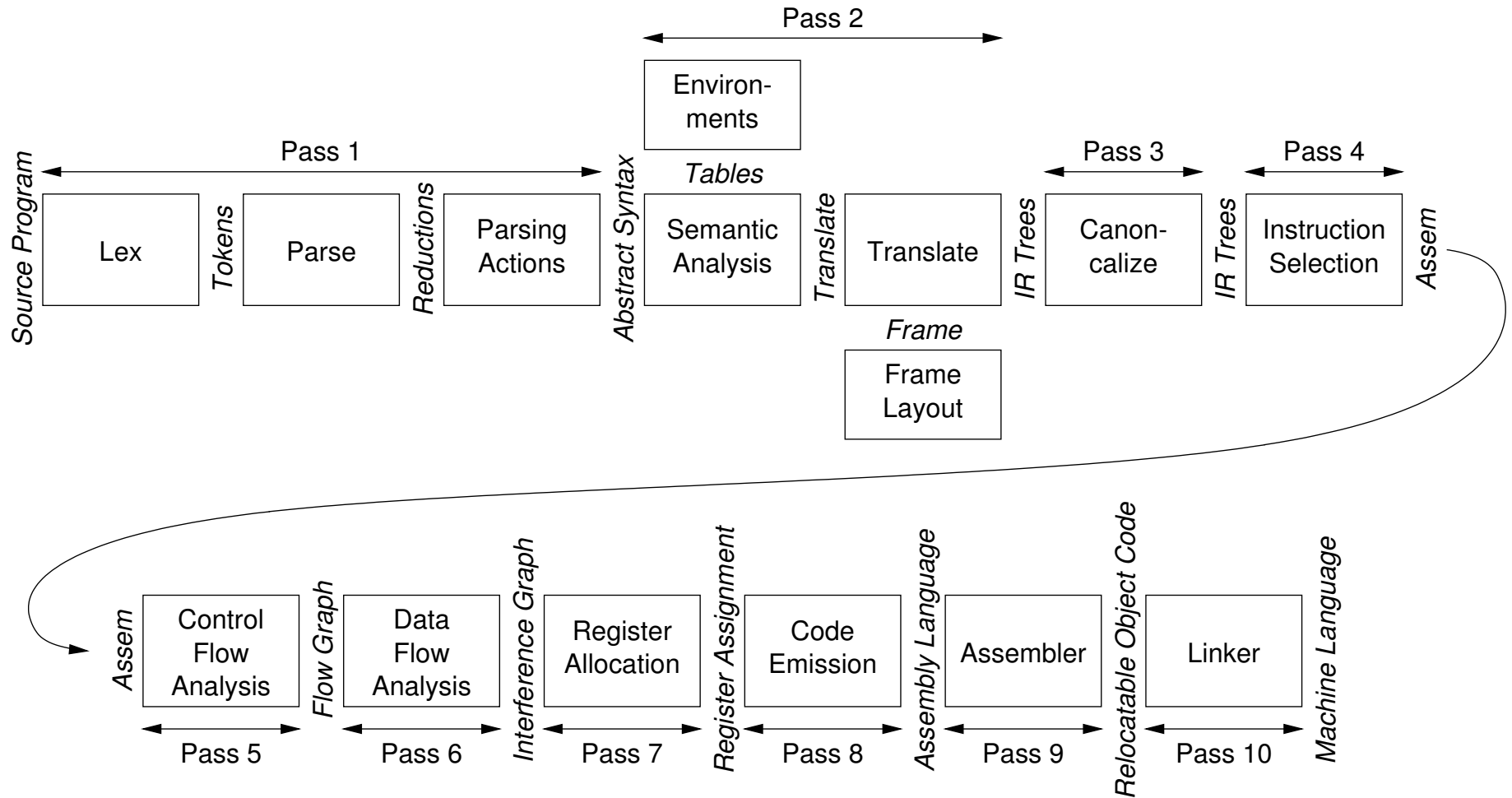


*Modern optimizers are usually built as a set of passes*

Typical passes

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

# The MiniJava compiler



# The MiniJava compiler phases

Lex	Break source file into individual words, or <i>tokens</i>
Parse	Analyse the phrase structure of program
Parsing Actions	Build a piece of <i>abstract syntax tree</i> for each phrase
Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase
Frame Layout	Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way
Translate	Produce <i>intermediate representation trees</i> (IR trees), a notation that is not tied to any particular source language or target machine
Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases
Instruction Selection	Group IR-tree nodes into clumps that correspond to actions of target-machine instructions
Control Flow Analysis	Analyse sequence of instructions into <i>control flow graph</i> showing all possible flows of control program might follow when it runs
Data Flow Analysis	Gather information about flow of data through variables of program; e.g., <i>liveness analysis</i> calculates places where each variable holds a still-needed ( <i>live</i> ) value
Register Allocation	Choose registers for variables and temporary values; variables not simultaneously live can share same register
Code Emission	Replace temporary names in each machine instruction with registers

# A straight-line programming language

---

<i>Stm</i>	$\rightarrow$ <i>Stm ; Stm</i>	CompoundStm
<i>Stm</i>	$\rightarrow$ <i>id := Exp</i>	AssignStm
<i>Stm</i>	$\rightarrow$ <i>print ( ExpList )</i>	PrintStm
<i>Exp</i>	$\rightarrow$ <i>id</i>	IdExp
<i>Exp</i>	$\rightarrow$ <i>num</i>	NumExp
<i>Exp</i>	$\rightarrow$ <i>Exp Binop Exp</i>	OpExp
<i>Exp</i>	$\rightarrow$ <i>( Stm , Exp )</i>	EseqExp
<i>ExpList</i>	$\rightarrow$ <i>Exp , ExpList</i>	PairExpList
<i>ExpList</i>	$\rightarrow$ <i>Exp</i>	LastExpList
<i>Binop</i>	$\rightarrow$ <i>+</i>	Plus
<i>Binop</i>	$\rightarrow$ <i>-</i>	Minus
<i>Binop</i>	$\rightarrow$ <i>×</i>	Times
<i>Binop</i>	$\rightarrow$ <i>/</i>	Div

An example straight-line program:

$a := 5 + 3; b := (\text{print}(a, a - 1), 10 \times a); \text{print}(b)$

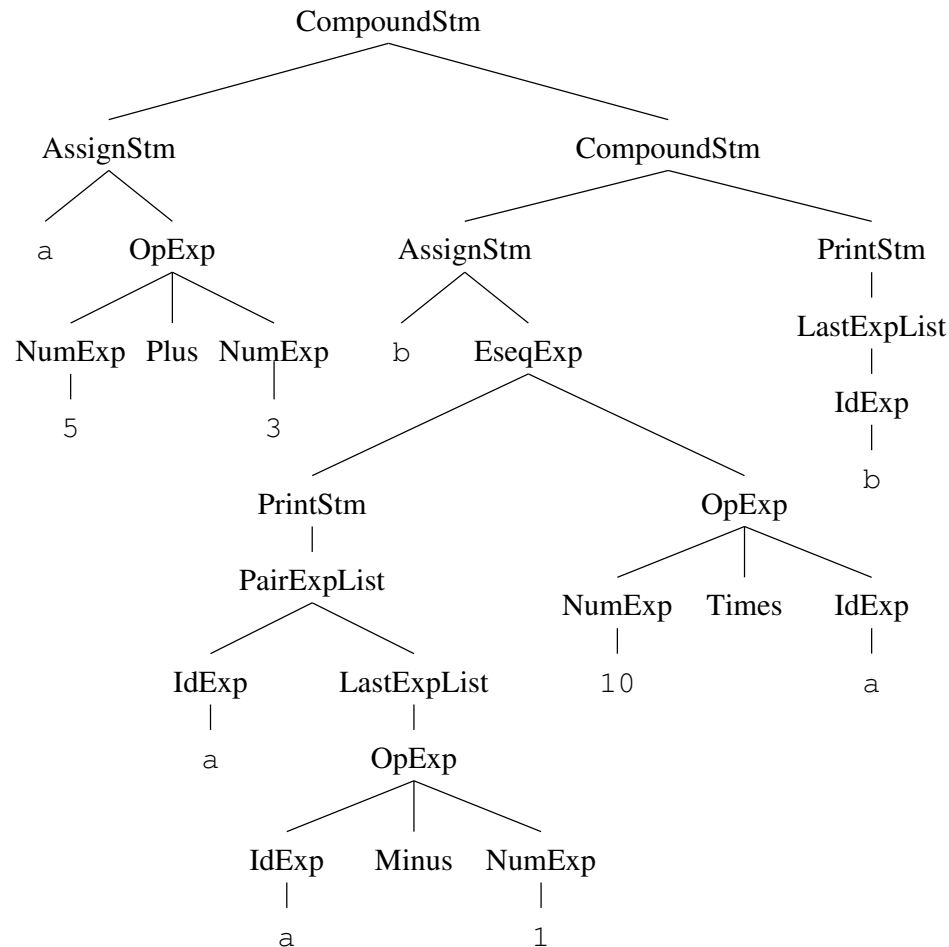
prints:

8 7

80

# Tree representation

`a := 5 + 3; b := (print(a, a - 1), 10 × a); print(b)`



This convenient internal representation is convenient for a compiler to use.

# Java classes for trees

---

```
abstract class Stm {}
class CompoundStm extends Stm {
    Stm stm1, stm2;
    CompoundStm(Stm s1, Stm s2)
    { stm1=s1; stm2=s2; }
}
class AssignStm extends Stm {
    String id; Exp exp;
    AssignStm(String i, Exp e)
    { id=i; exp=e; }
}
class PrintStm extends Stm {
    ExpList exps;
    PrintStm(ExpList e)
    { exps=e; }
}

abstract class Exp {}
class IdExp extends Exp {
    String id;
    IdExp(String i) {id=i;}
}
```

```
class NumExp extends Exp {
    int num;
    NumExp(int n) {num=n;}
}
class OpExp extends Exp {
    Exp left, right; int oper;
    final static int
        Plus=1,Minus=2,Times=3,Div=4;
    OpExp(Exp l, int o, Exp r)
    { left=l; oper=o; right=r; }
}
class EseqExp extends Exp {
    Stm stm; Exp exp;
    EseqExp(Stm s, Exp e)
    { stm=s; exp=e; }
}
abstract class ExpList {}
class PairExpList extends ExpList {
    Exp head; ExpList tail;
    public PairExpList(Exp h, ExpList t)
    { head=h; tail=t; }
}
class LastExpList extends ExpList {
    Exp head;
    public LastExpList(Exp h) {head=h;}
}
```