Compiler Construction
1. Introduction

Oscar Nierstrasz

Compiler Construction

Lecturers	Prof. Oscar Nierstrasz, Dr. Mircea Lungu
Assistants	Jan Kurš, Boris Spasojević
Lectures	E8 001, Fridays @ 10h15-12h00
Exercises	E8 001, Fridays @ 12h00-13h00
WWW	scg.unibe.ch/teaching/cc

MSc registration Spring 2015

MASTER IN COMPUTER SCIENCE

JMCS students

- Register on Academia for **teaching units** by **March 13, 2015**
- Register on Academia for **exams** by **May 15, 2015**
- Request reimbursement of travel expenses by **June 30, 2015**

NB: Hosted JMCS students *(e.g. CS bachelor students etc.)* must additionally:

- Request for Academia access by **February 28, 2015**

Roadmap

- > Overview
- > Front end
- > Back end
- > Multi-pass compilers
- > Example: compiler and interpreter for a toy language

See *Modern compiler implementation in Java* (Second edition), chapter 1.

Roadmap

> **Overview**
> Front end
> Back end
> Multi-pass compilers
> Example: compiler and interpreter for a toy language

Textbook

> Andrew W. Appel, ***Modern compiler implementation in Java*** (Second edition), Cambridge University Press, New York, NY, USA, 2002, with Jens Palsberg.

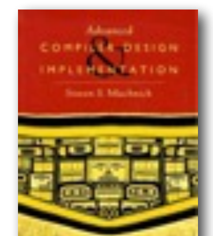
Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/

Other recommended sources

> **Compilers: Principles, Techniques, and Tools**, Aho, Sethi and Ullman
> —http://dragonbook.stanford.edu/

> **Parsing Techniques**, Grune and Jacobs
> —http://www.cs.vu.nl/~dick/PT2Ed.html

> **Advanced Compiler Design and Implementation**, Muchnik

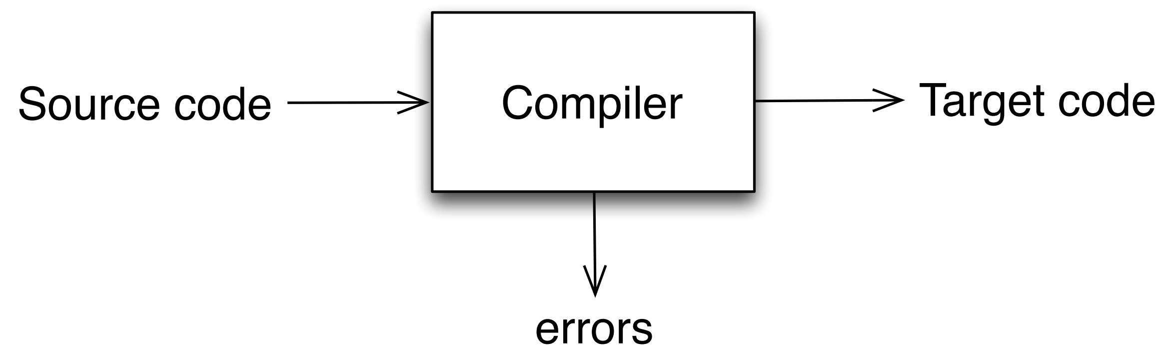


Schedule

1	20-Feb-15	**Introduction**
2	27-Feb-15	**Lexical Analysis**
3	06-Mar-15	**Parsing**
4	13-Mar-15	**Parsing in Practice**
5	20-Mar-15	**Semantic Analysis**
6	27-Mar-15	**Intermediate Representation**
	03-Apr-15	***Good Friday***
	10-Apr-15	***Spring break***
7	17-Apr-15	**Optimization**
8	24-Apr-15	**Code Generation**
9	01-May-15	**Bytecode and Virtual Machines**
10	08-May-15	**PEGs, Packrats and Parser Combinators**
11	15-May-15	**Program Transformation**
12	22-May-15	***Project Presentations***
13	*29-May-15*	***Final Exam***

What is a compiler?

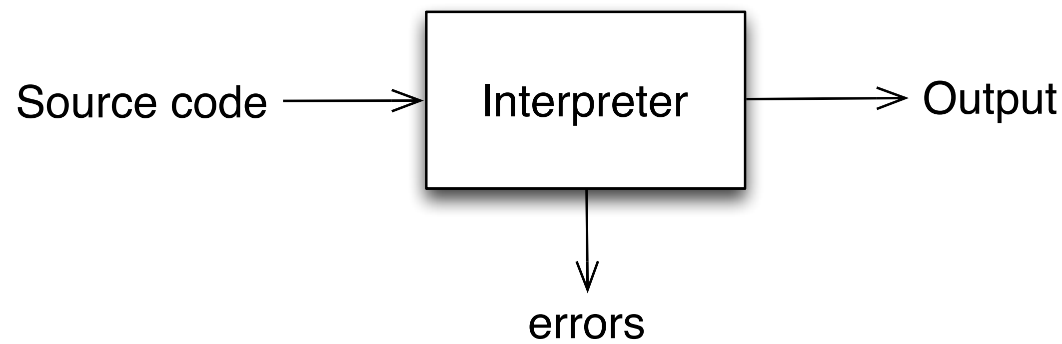
a program that translates an *executable* program in one language into an *executable* program in another language



Translates "source code" into "target code".
We expect the program produced by the compiler to be "better", in some way, than the original.

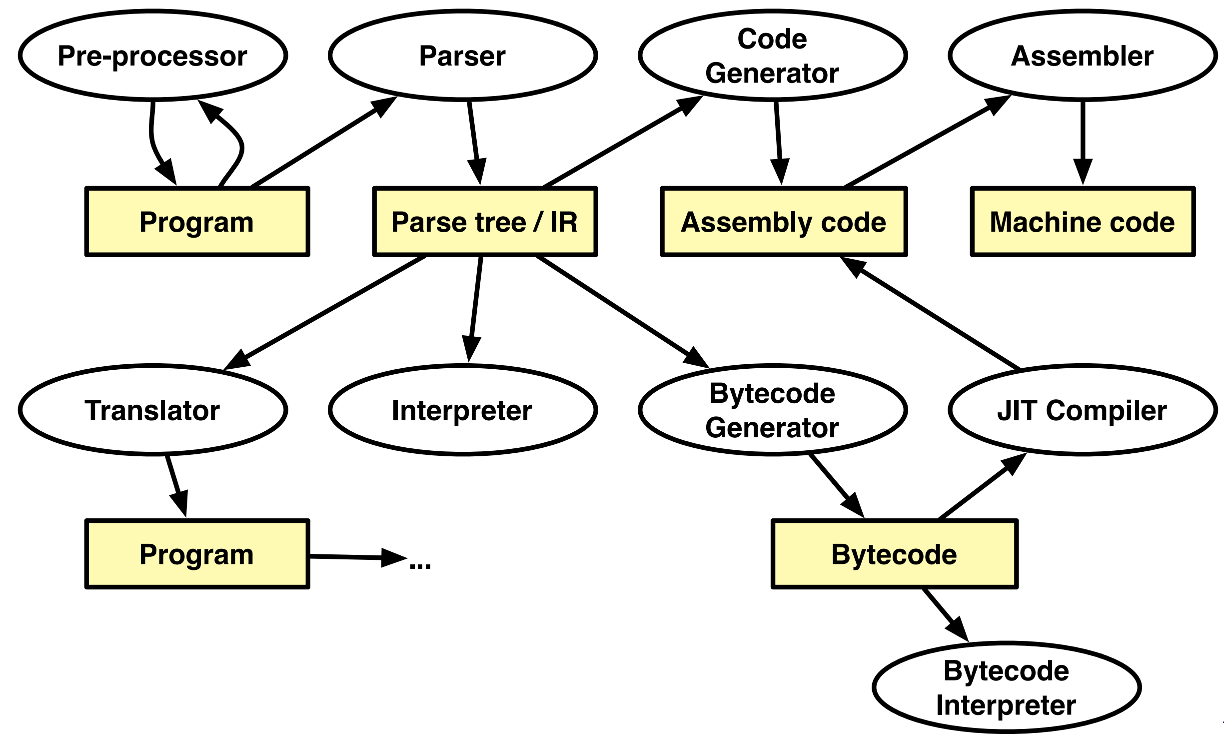
What is an interpreter?

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.

Implementing Compilers, Interpreters …



This picture offers a high-level overview of the different approaches to implementing languages

Why do we care?

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 for analysis*
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 ⇒ changes in compilers
  —new features pose new problems
  —changing costs lead to different concerns
  —old solutions need re-engineering

> Innovations in compilers should prompt changes in architecture
  —New languages and features

For example, computationally expensive but simpler scannerless parsing techniques are undergoing a renaissance.

What qualities are important in a compiler?

- Correct code
- Output runs fast
- Compiler runs fast
- Compile time proportional to program size
- Support for separate compilation
- Good diagnostics for syntax errors
- Works well with the debugger
- Good diagnostics for flow anomalies
- Cross language calls
- Consistent, predictable optimization

Each of these shapes your feelings about the correct contents of this course

A bit of history

> **1952:** First compiler (linker/loader) written by Grace Hopper for **A-0** programming language

> **1957:** First complete compiler for **FORTRAN** by John Backus and team

> **1960: COBOL** compilers for multiple architectures

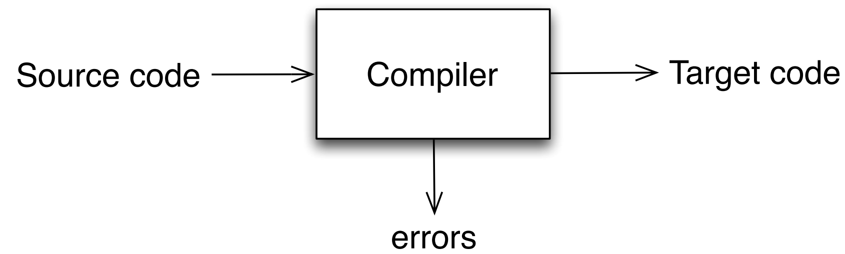
> **1962:** First self-hosting compiler for **LISP**

http://en.wikipedia.org/wiki/Compiler ;-)

A compiler was originally a program that "compiled" subroutines [a link-loader]. When in 1954 the combination "algebraic compiler" came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

— Bauer and Eickel [1975]

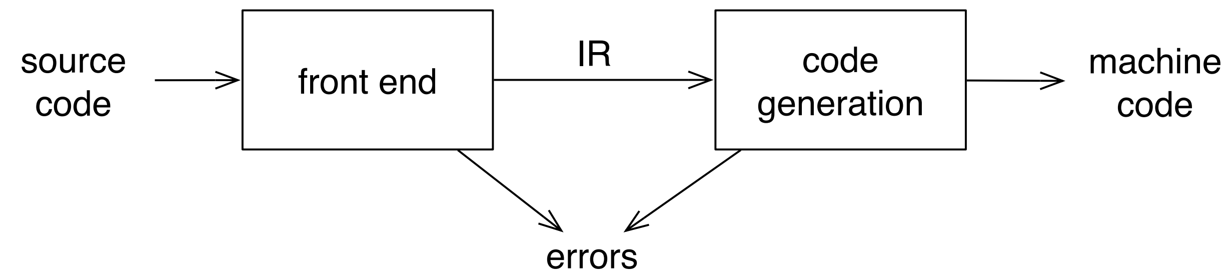
Abstract view



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

Big step up from assembler — higher level notations

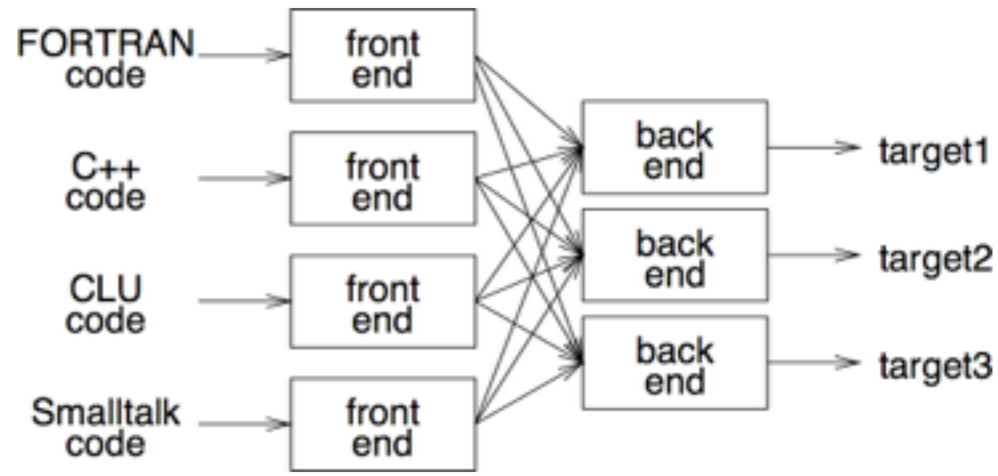
Traditional two pass compiler



- front end maps legal code into IR
- intermediate representation (IR)
- back end maps IR onto target machine
- simplifies retargeting
- allows multiple front ends
- multiple passes ⇒ better code

A classical compiler consists of a front end that parses the source code into an intermediate representation, and a back end that generates executable code.

A fallacy!



Front-end, IR and back-end must encode knowledge needed for all n×m combinations!

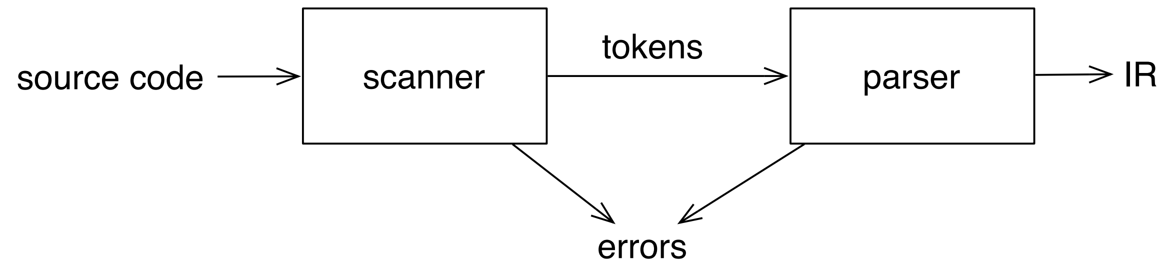
- must encode all the knowledge in each front end
- must represent all the features in one IR
- must handle all the features in each back end

Limited success with low-level IRs

Roadmap

> Overview
> **Front end**
> Back end
> Multi-pass compilers
> Example: compiler and interpreter for a toy language

Front end

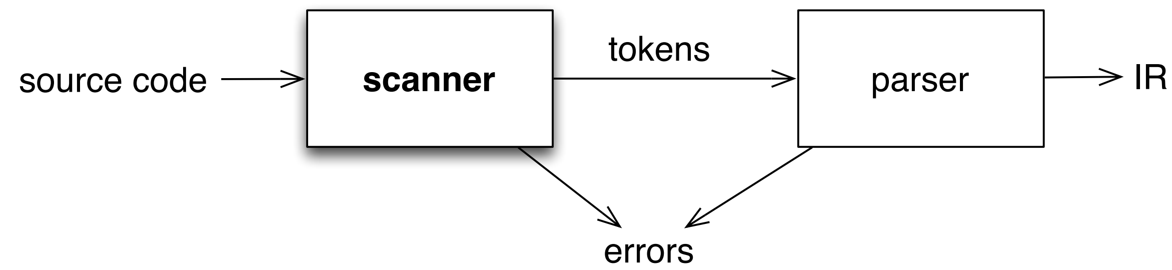


- recognize legal code
- report errors
- produce IR
- preliminary storage map
- shape code for the back end

Much of front end construction can be automated

- preliminary storage map => not only prepare symbol table, but decide what part of storage different names (entities) should be mapped to (local, global, automatic etc)
- shape code for the back end => decide how different parts of code are organized (in the IR)

Scanner

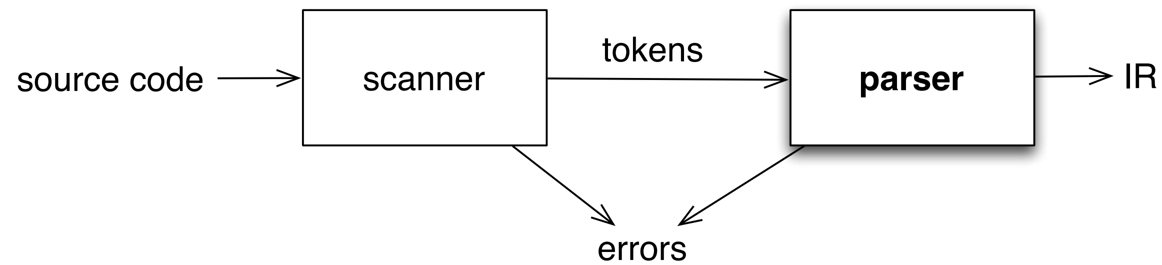


- map characters to ***tokens***
- character string value for a token is a ***lexeme***
- eliminate white space

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


- character string value for a *token* is a *lexeme*
- *Typical tokens: id, number, do, end …*
- *Key issue is speed*

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

Context-free grammars

Context-free syntax is specified with a *grammar*, usually in *Backus-Naur form* (BNF)

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

A grammar $G = (S,N,T,P)$

- S is the <u>*start-symbol*</u>
- N is a set of <u>*non-terminal symbols*</u>
- T is a set of terminal symbols
- P is a set of <u>*productions*</u> — $P: N \rightarrow (N \cup T)^*$

Called “context-free” because rules for non-terminals can be written without regard for the context in which they appear.

Deriving valid sentences

Production	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`

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

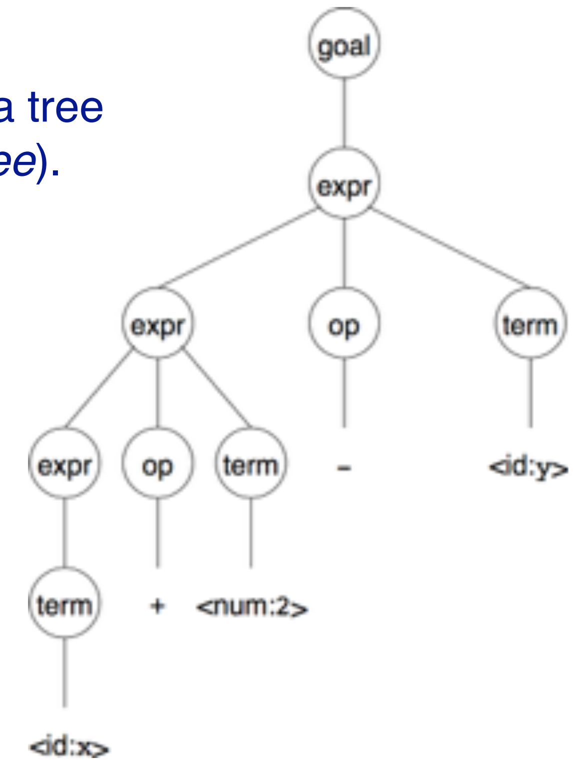
To *recognize* a valid sentence in some CFG, we *reverse* this process and build up a *parse*.

The parse is the sequence of productions needed to parse the input.

The *parse tree* is something else …

Parse trees

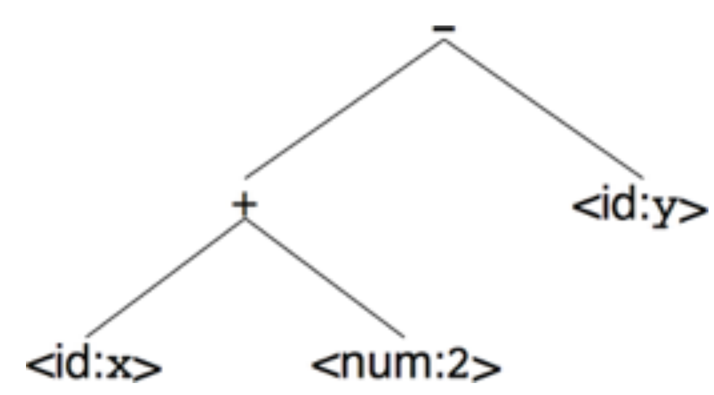
A parse can be represented by a tree called a *parse tree* (or *syntax tree*).



Obviously, this contains a lot of unnecessary information

Abstract syntax trees

So, compilers often use an ***abstract syntax tree*** (AST).

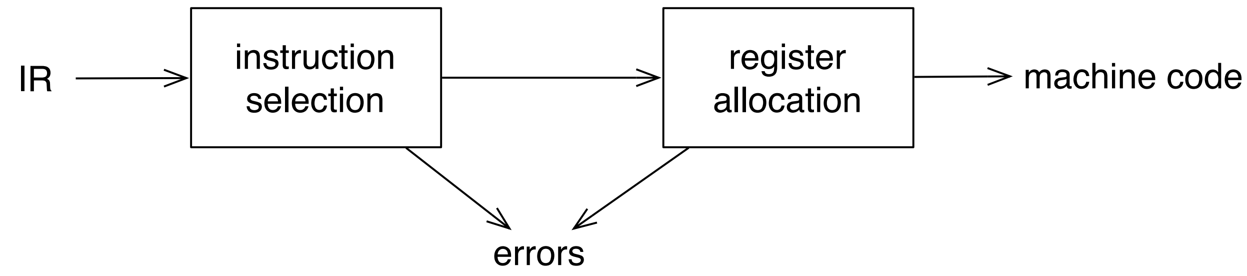


ASTs are often used as an IR.

Roadmap

- Overview
- Front end
- **Back end**
- Multi-pass compilers
- Example: compiler and interpreter for a toy language

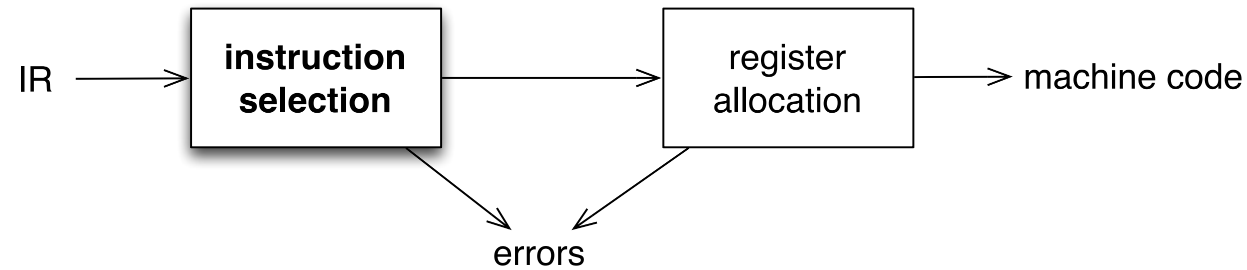
Back end



- 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

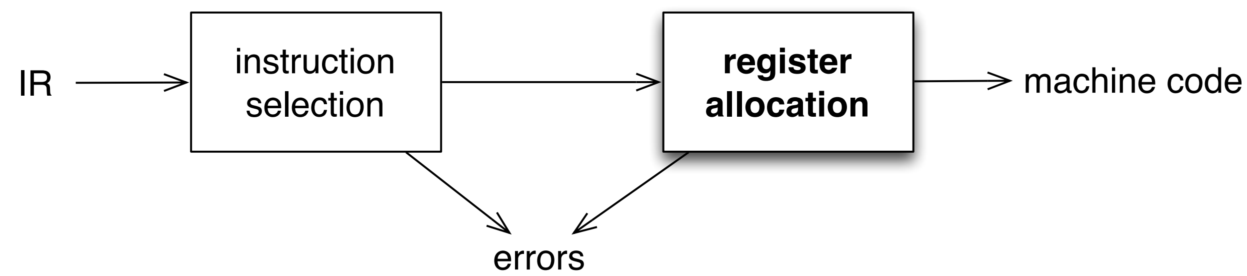
Automation has been less successful here

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



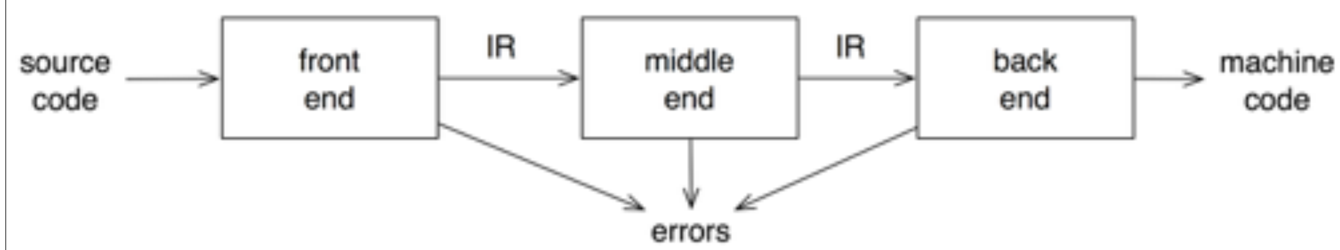
- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult

Modern allocators often use an analogy to graph coloring

Roadmap

> Overview
> Front end
> Back end
> **Multi-pass compilers**
> Example: compiler and interpreter for a toy language

Traditional three-pass compiler

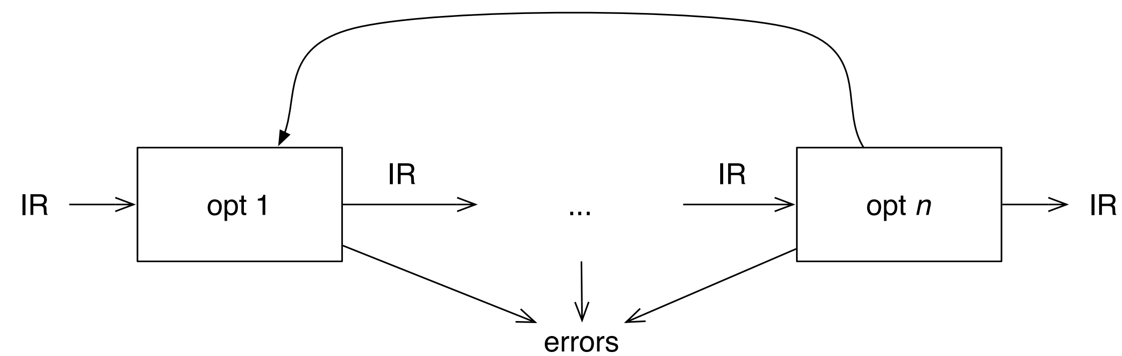


- analyzes and changes IR
- goal is to reduce runtime *(optimization)*
- must preserve results

Code improvement and optimization

Optimizer (middle end)

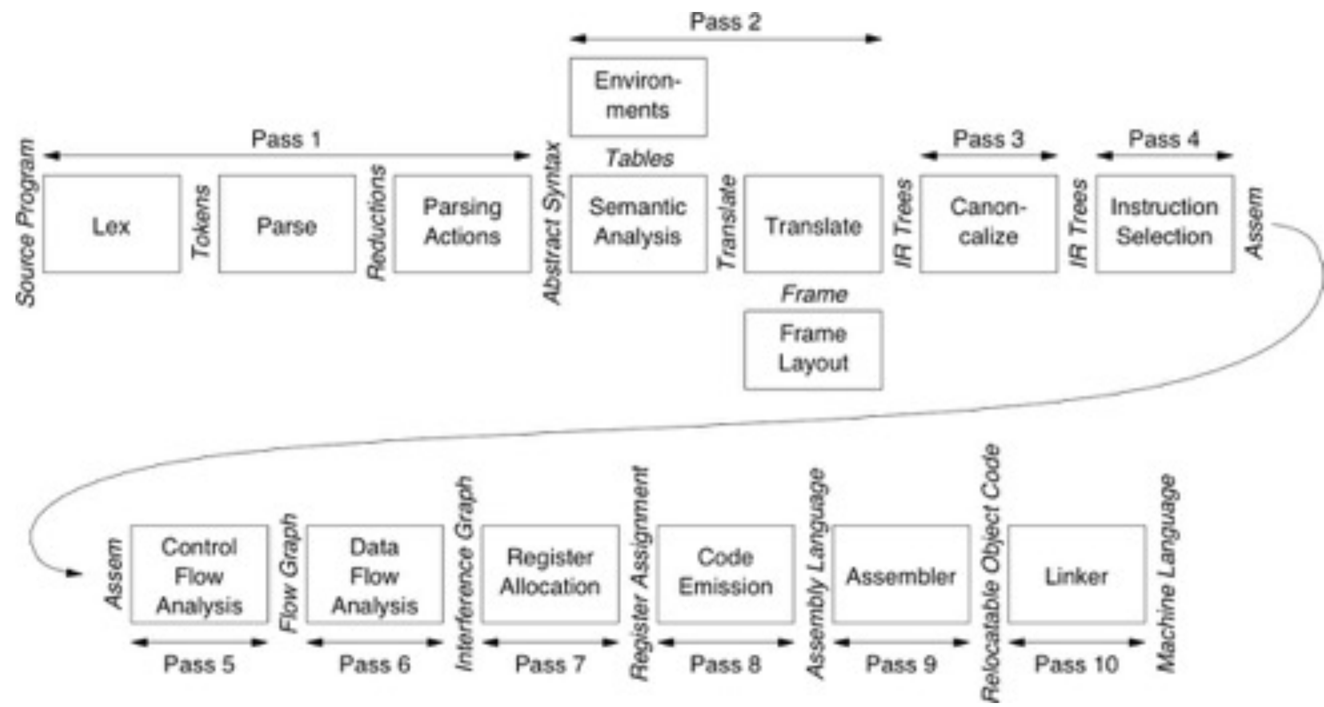
Modern optimizers are usually built as a set of passes



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

- constant propagation and folding *(evaluate and propagate constant expressions at compile time)*
- code motion *(move code that does not need to be reevaluated out of loops)*
- reduction of operator strength *(replace slow operations by equivalent faster ones)*
- common sub-expression elimination *(evaluate once and store)*
- redundant store elimination *(detect when values are stored repeatedly and eliminate)*
- dead code elimination *(eliminate code that can never be executed)*

The MiniJava compiler



Cf. MCIJ 2d edn p 4

Compiler phases

Lex	Break source file into individual words, or *tokens*
Parse	Analyse the phrase structure of program
Parsing Actions	Build a piece of *abstract syntax tree* 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 *intermediate representation trees* (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 *control flow graph* 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., *liveness analysis* calculates places where each variable holds a still-needed (live) 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

Roadmap

> Overview
> Front end
> Back end
> Multi-pass compilers
> **Example: compiler and interpreter for a toy language**

A straight-line programming language (no loops or conditionals):

Stm	→	Stm ; Stm	*CompoundStm*
Stm	→	`id` := Exp	*AssignStm*
Stm	→	`print` (ExpList)	*PrintStm*
Exp	→	`id`	*IdExp*
Exp	→	`num`	*NumExp*
Exp	→	Exp Binop Exp	*OpExp*
Exp	→	(Stm , Exp)	*EseqExp*
ExpList	→	Exp , ExpList	*PairExpList*
ExpList	→	Exp	*LastExpList*
Binop	→	+	*Plus*
Binop	→	–	*Minus*
Binop	→	×	*Times*
Binop	→	/	*Div*

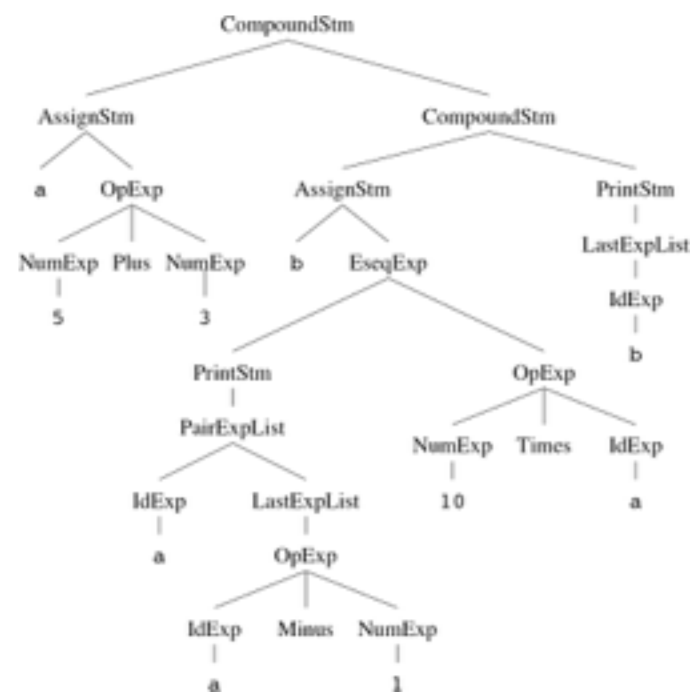
```
a := 5 + 3; b := (print(a,a—1),10×a); print(b)
```

prints

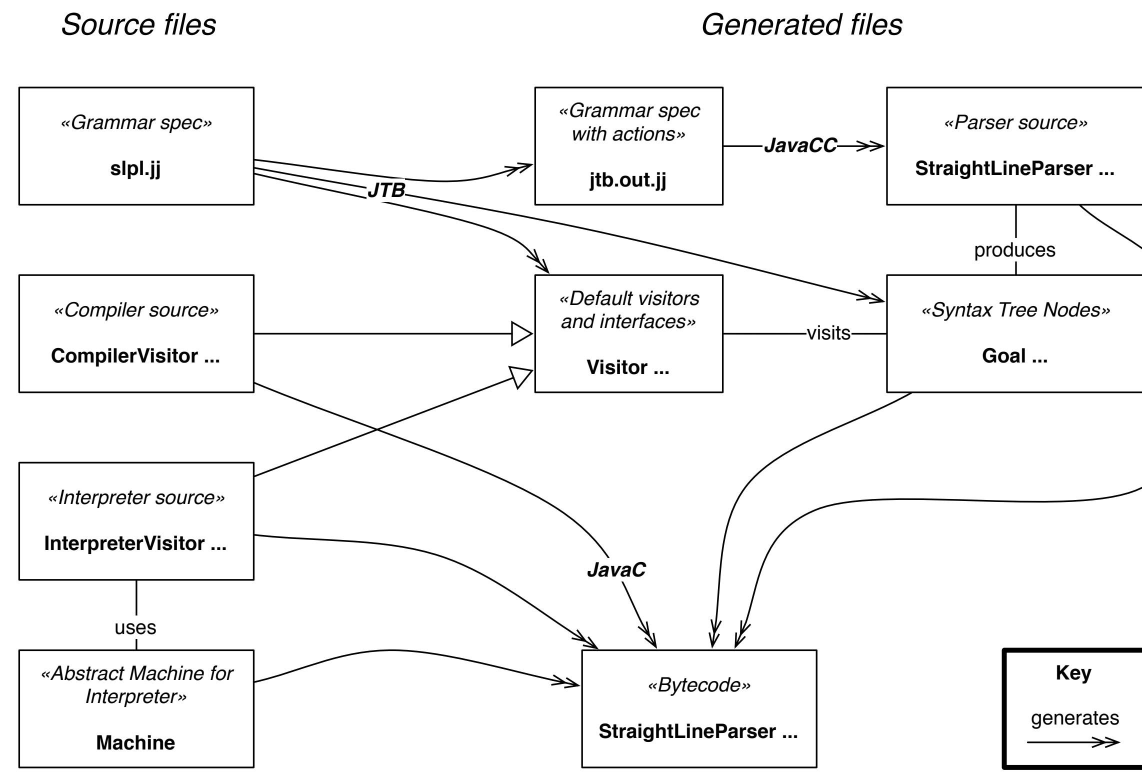
```
8 7
80
```

Tree representation

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



Straightline Interpreter and Compiler Files

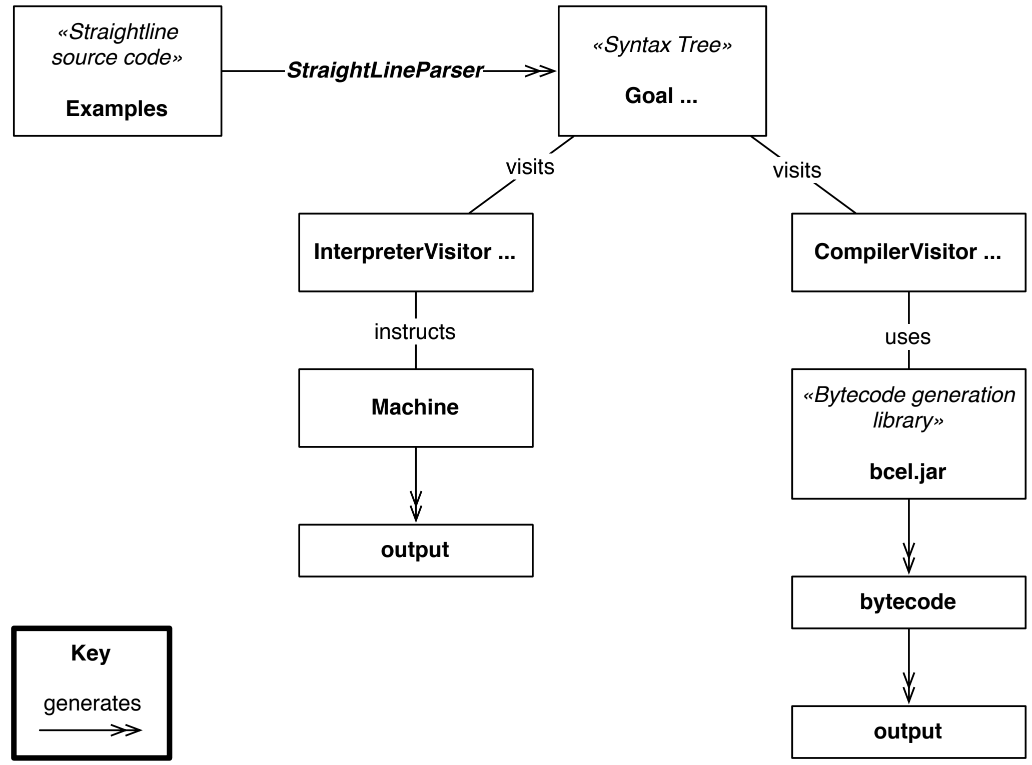


Java classes for trees

```
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;}
}
```

Straightline Interpreter and Compiler Runtime



What you should know!

- *What is the difference between a compiler and an interpreter?*
- *What are important qualities of compilers?*
- *Why are compilers commonly split into multiple passes?*
- *What are the typical responsibilities of the different parts of a modern compiler?*
- *How are context-free grammars specified?*
- *What is “abstract” about an abstract syntax tree?*
- *What is intermediate representation and what is it for?*
- *Why is optimization a separate activity?*

Can you answer these questions?

- ✎ *Is Java compiled or interpreted? What about Smalltalk? Ruby? PHP? Are you sure?*
- ✎ *What are the key differences between modern compilers and compilers written in the 1970s?*
- ✎ *Why is it hard for compilers to generate good error messages?*
- ✎ *What is "context-free" about a context-free grammar?*