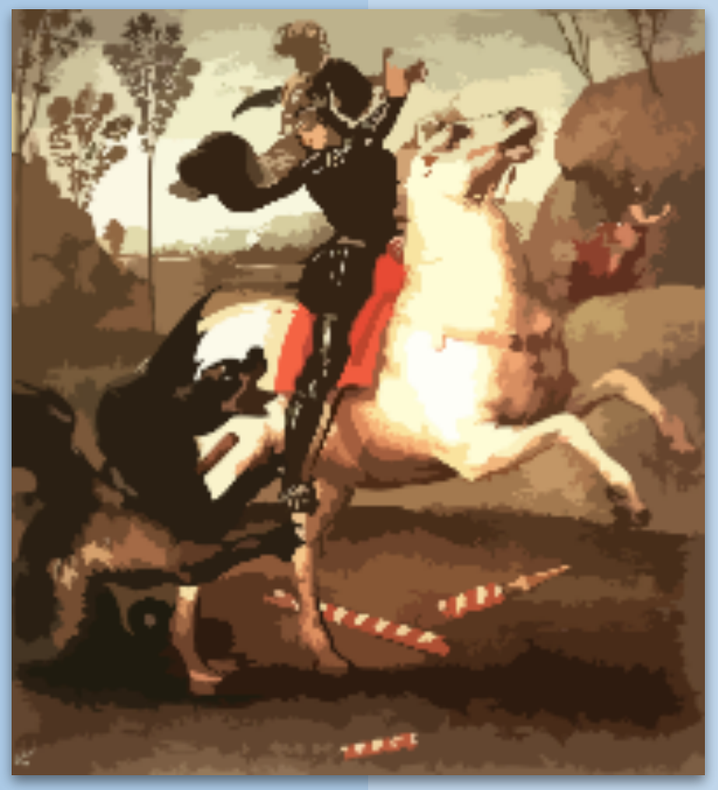
b
UNIVERSITÄT
BERN

Compiler Construction
1. Introduction

Oscar Nierstrasz



Compiler Construction

Lecturers	Prof. Oscar Nierstrasz, Dr. Mohammad Ghafari
Assistants	Manuel Leuenberger, Rathesan Iyadurai
Lectures	E8 001, Fridays @ 10h15-12h00
Exercises	E8 001, Fridays @ 12h00-13h00
WWW	scg.unibe.ch/teaching/cc

This is a note (a hidden slide). You will find some of these scattered around the PDF versions of the slides.

Roadmap

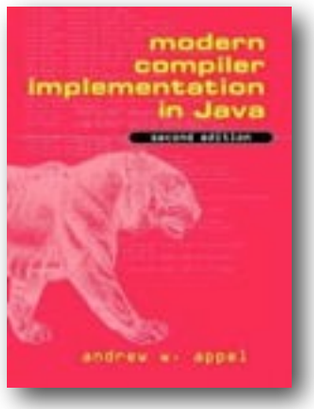
- Overview
- Front end
- Back end
- Multi-pass compilers
- Example: compiler and interpreter for a toy language

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

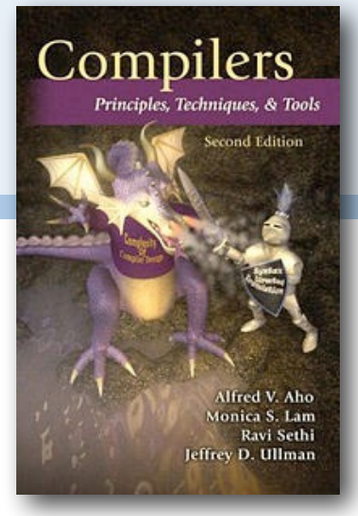


> Andrew W. Appel, ***Modern compiler implementation in Java*** (Second edition), Cambridge University Press, New York, NY, USA, 2002, with Jens Palsberg.

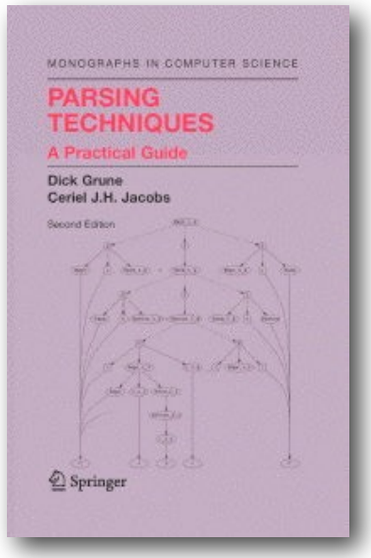
Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/

The book by Appel is the main source for this lecture series, and many of the slides have been adapted from similar courses by Palsberg and Hosking,

Other recommended sources

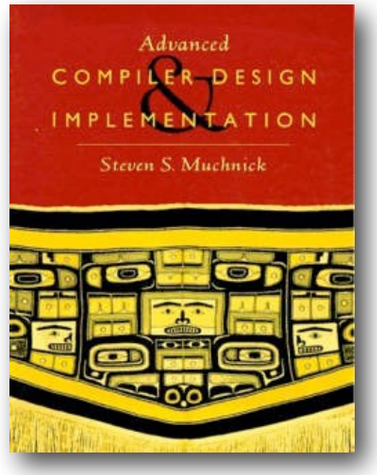


- **Compilers: Principles, Techniques, and Tools**, Aho, Sethi and Ullman
  - —http://dragonbook.stanford.edu/



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

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

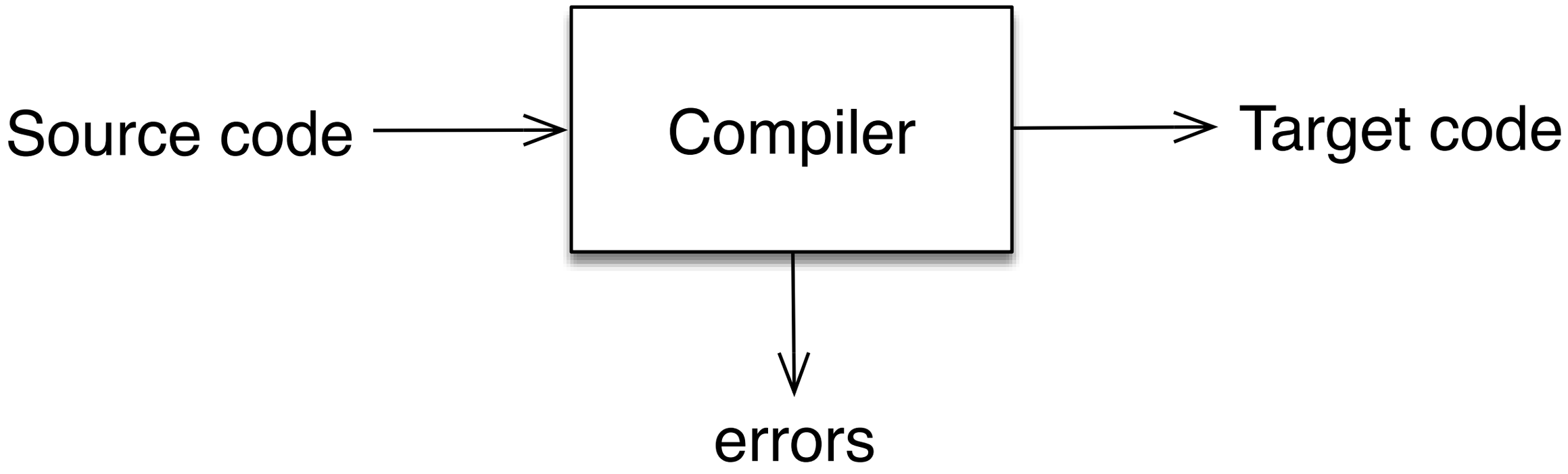


Schedule

1	22-Feb-19	**Introduction**
2	01-Mar-19	**Lexical Analysis**
3	08-Mar-19	**Parsing**
4	15-Mar-19	**Parsing in Practice**
5	22-Mar-19	**Intermediate Representation**
6	29-Mar-19	**Optimization**
7	05-Apr-19	**Code Generation**
8	12-Apr-19	**Bytecode and Virtual Machines**
	19-Apr-19	***Good Friday***
	26-Apr-19	***Spring break***
9	03-May-19	**PEGs, Packrats and Parser Combinators**
10	10-May-19	**Truffle — a language implementation framework**
11	17-May-19	**Program Transformation**
12	24-May-19	**Compiling R — a case study**
13	*31-May-19*	***Final Exam***

What is a compiler?

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



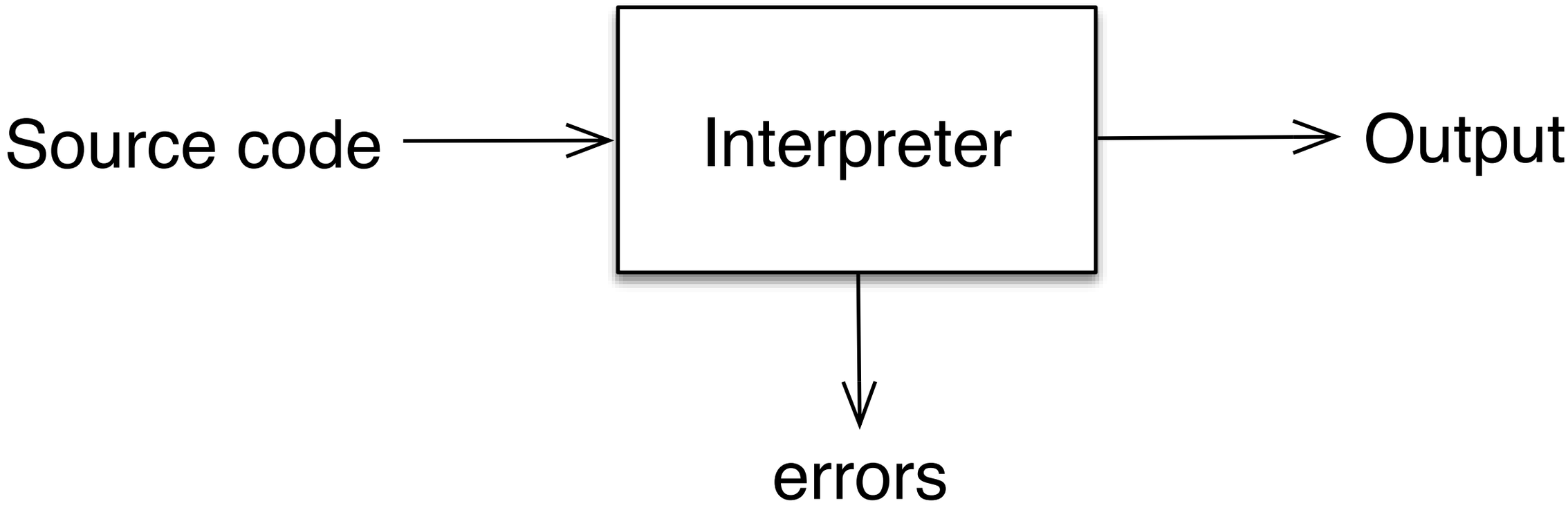
A compiler is a translator from (executable) source code to some new form of executable target code. The target is typically machine code, or virtual machine bytecode.

We expect the target to be “better” in some way, for example, it should be execute efficiently, and possibly eliminate anomalies such as dead code.

Note that a compiler *does not execute the program.*

What is an interpreter?

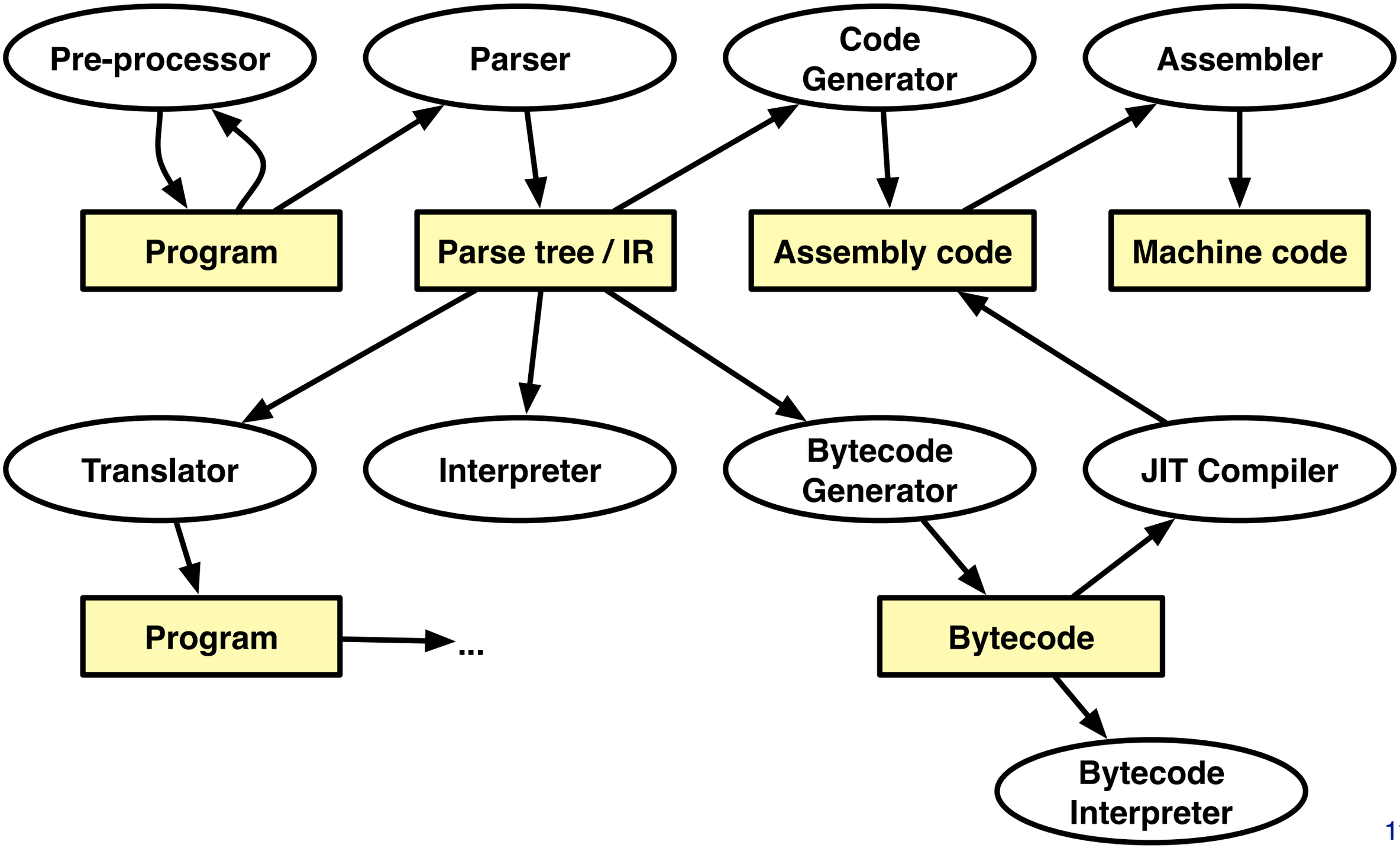
a program that reads an *executable* program and produces the *results* of running that program



The job of an interpreter is to execute source code, possibly consuming input, and producing output. An interpreter typically will translate the source code to some intermediate representation along the way, but this intermediate form will not necessarily be stored as an artifact.

In contrast to a compiler, an interpreter does execute the source program.

Implementing Compilers, Interpreters …



The “program” in the previous slide refers to the program source code. It can be pre-processed as much as needed. To do anything “useful” it has to be parsed in to some Intermediate Representation (IR/Parse Tree). The IR can be taken by a translator to produce code in a different language, or an interpreter which can execute it directly, or a bytecode/assembly generator to generate code for execution on a VM (bytecode interpreter or JIT) or a real machine (generating machine code from the assembly code).

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

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

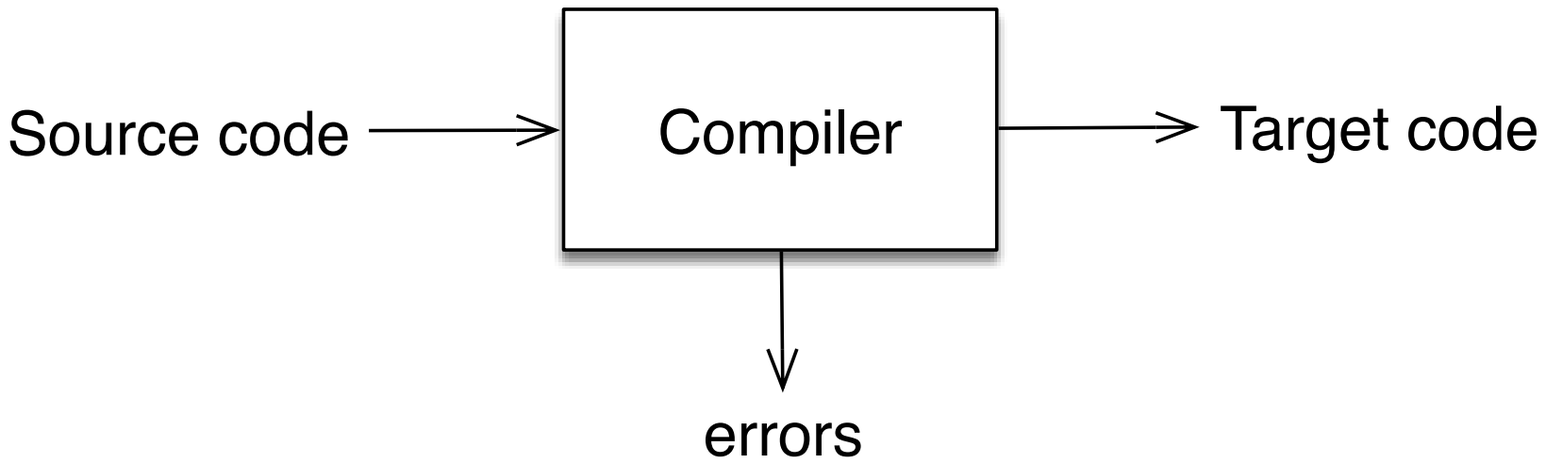
- **1960: COBOL** compilers for multiple architectures

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

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]

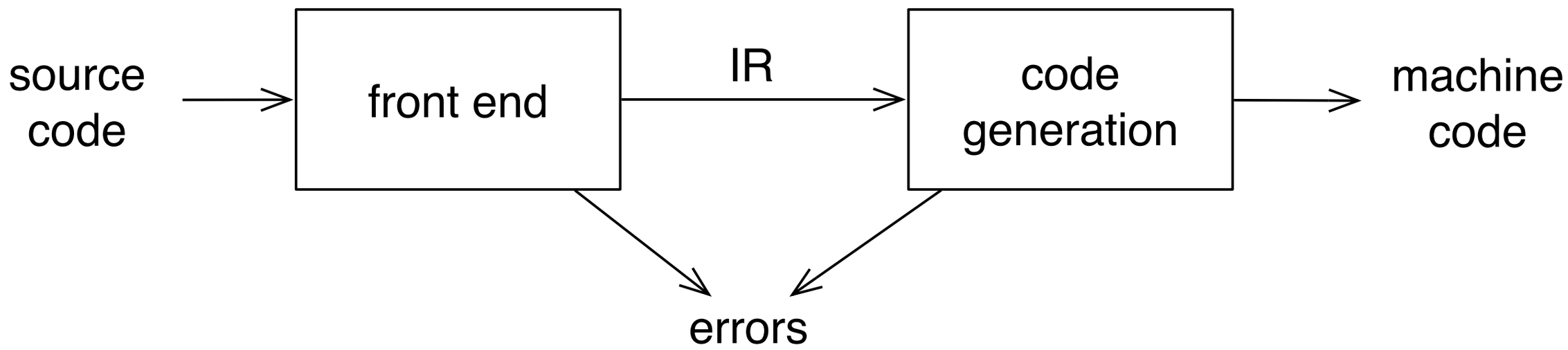
Abstract view



- 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

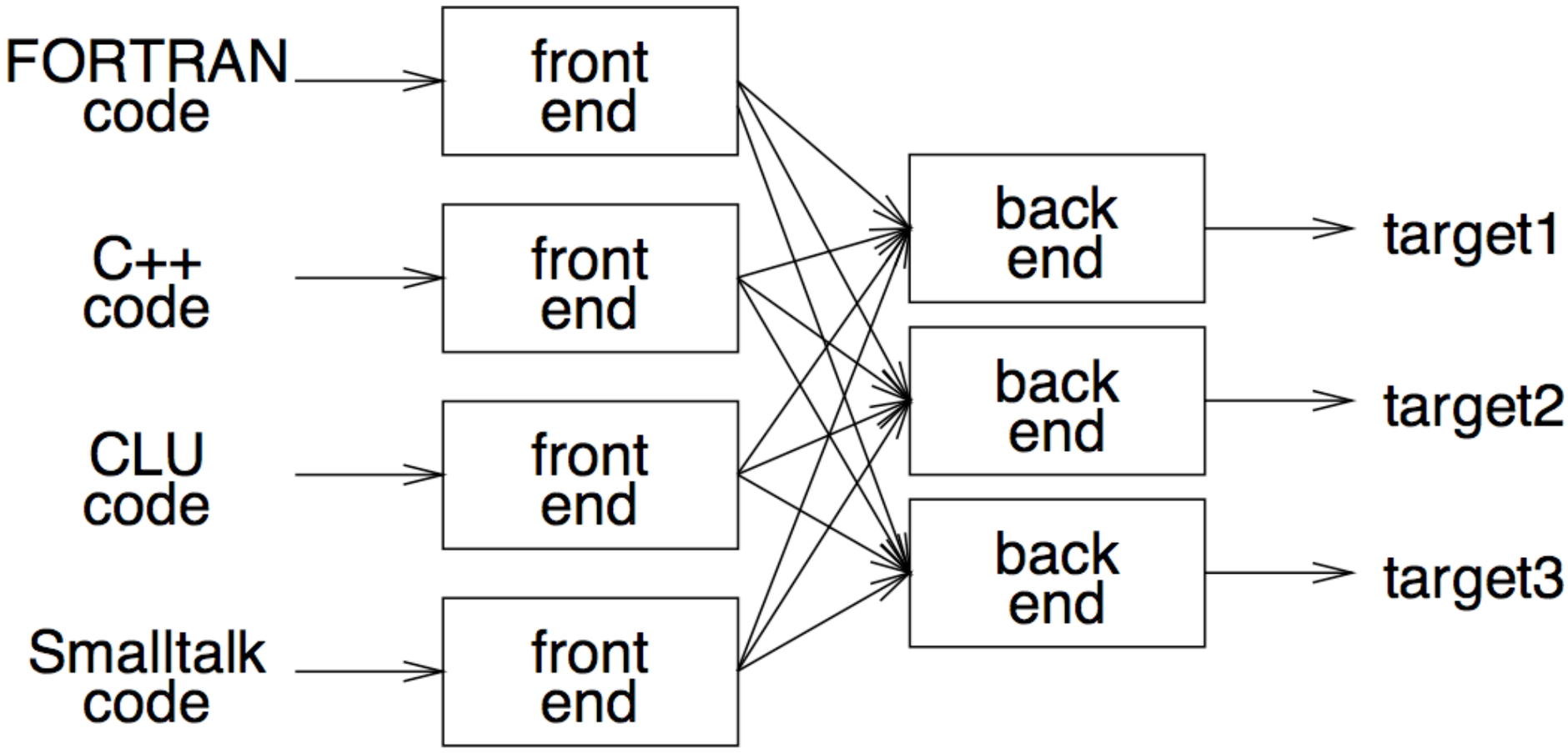
Traditional two pass compiler



- front end maps legal code into IR (syntax)
- intermediate representation (IR)
- back end maps IR onto target machine (semantics)
- 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. The front end is therefore more concerned with language *syntax*, while the back end deals with its *semantics*.

A fallacy!



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

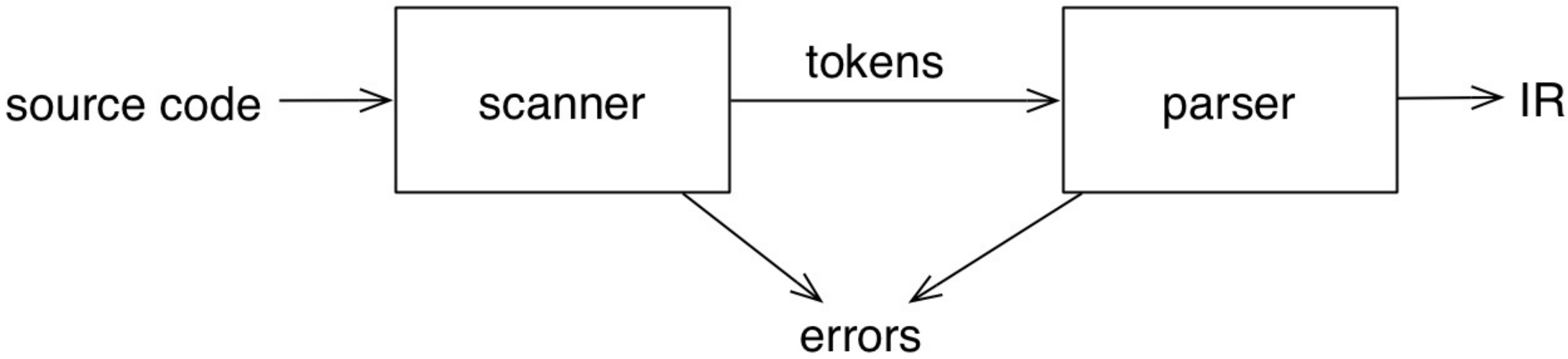
Using different front ends for a back end (or the other way around) is a nice side effect of the front end/back end separation, but it is not the *reason* for it.

The reason is a clear separation of concerns: the front end should deal with all things to do with the language and source code; the back end should deal with all things to do with the target architecture.

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

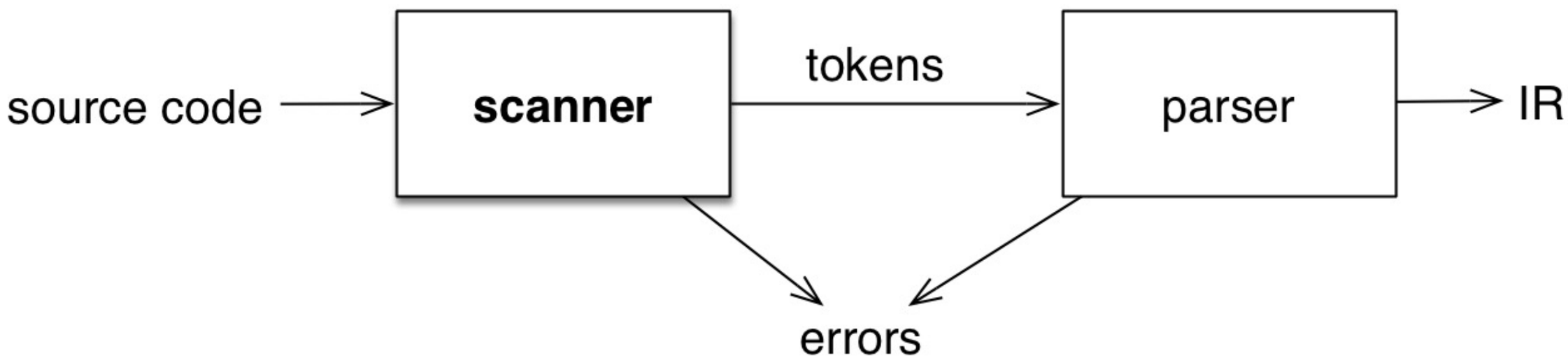
Much of front end construction can be automated

The front end of a compiler is classically split into a *scanner*, which is responsible for converting the source code into a stream of tokens of the source language, and a *parser*, which is responsible for recognizing structure in the stream of tokens.

The preliminary *storage map* not only tracks which tokens represent names of programmer-defined entities such as variables and procedures (i.e., the *symbol table*), but also decides what part of storage different names (entities) should be mapped to (local, global, automatic etc.).

The front end also shapes code for the back end, i.e., by deciding how different parts of code are organized in the IR.

Scanner



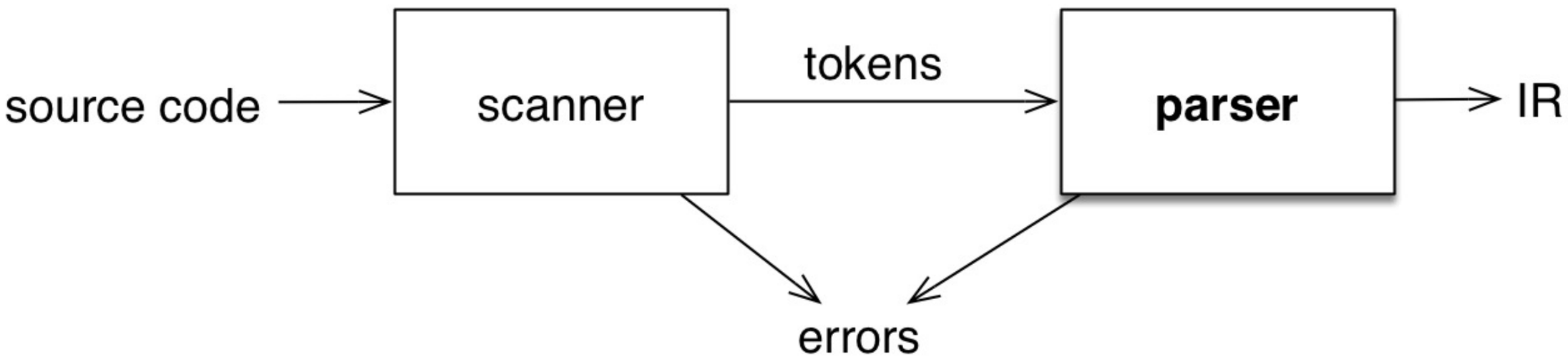
- map characters to *tokens*
- character string value for a token is a *lexeme*
- eliminate white space

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

The scanner converts the source code — i.e., a stream of characters — into a stream of semantically significant tokens, such as keywords, identifiers, numbers, strings, etc.

Each token consists of a type (e.g., keyword), and its string value (e.g., “class”).

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

The job of the parser is to *recognize structure in the stream of tokens* produced by the scanner. Typically it builds a *parse tree* representing this structure, but it may also produce another kind of IR (intermediate representation) more suitable for the backend.

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

Such a grammar is called “context-free” because rules for non-terminals can be written without regard for the context in which they appear: a number is always a number, regardless of where it occurs in the program.

Unfortunately modern programming languages often have *context-sensitive* features that make parsing more complicated. An example is a *here document*, which is terminated by a special string specified at the beginning. (It is context-sensitive because the terminator depends on a string specified earlier in the program.)

https://en.wikipedia.org/wiki/Here_document

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.

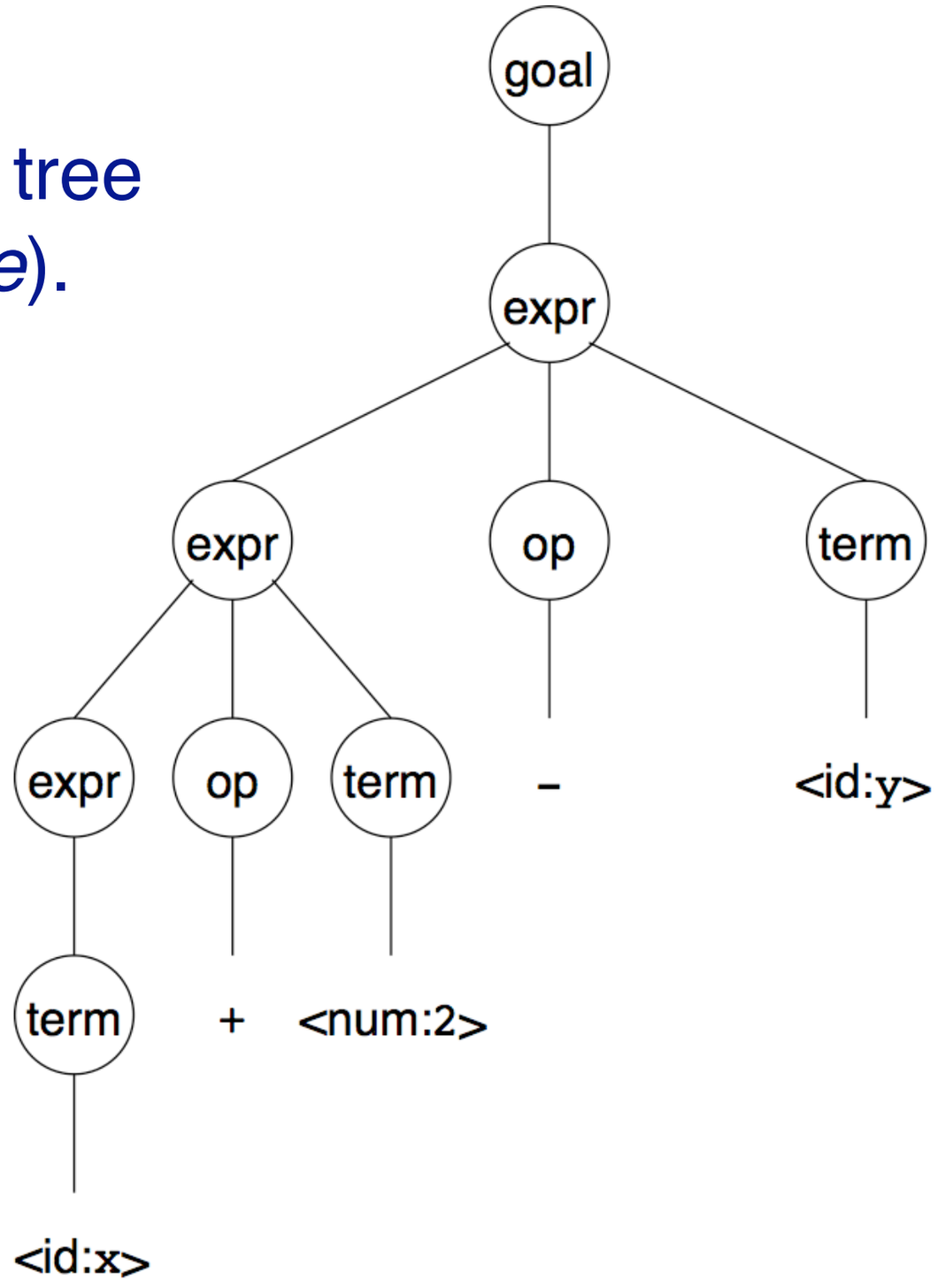
To *recognize* a valid sentence in some CFG, we *reverse* this process and build up a *parse*.

Formal grammars were invented to represent all the possible sentences of a given language. The grammar is used to *derive* (i.e., to *generate*) all these strings. In compilers we do the opposite; we want to use the grammar to take an input string (a sentence of the language) and *recognize* its structure, that is, to reconstruct its derivation, or to *parse* it.

The *parse* is the sequence of productions needed to parse the input. The *parse tree* is a structure representing this derivation.

Parse trees

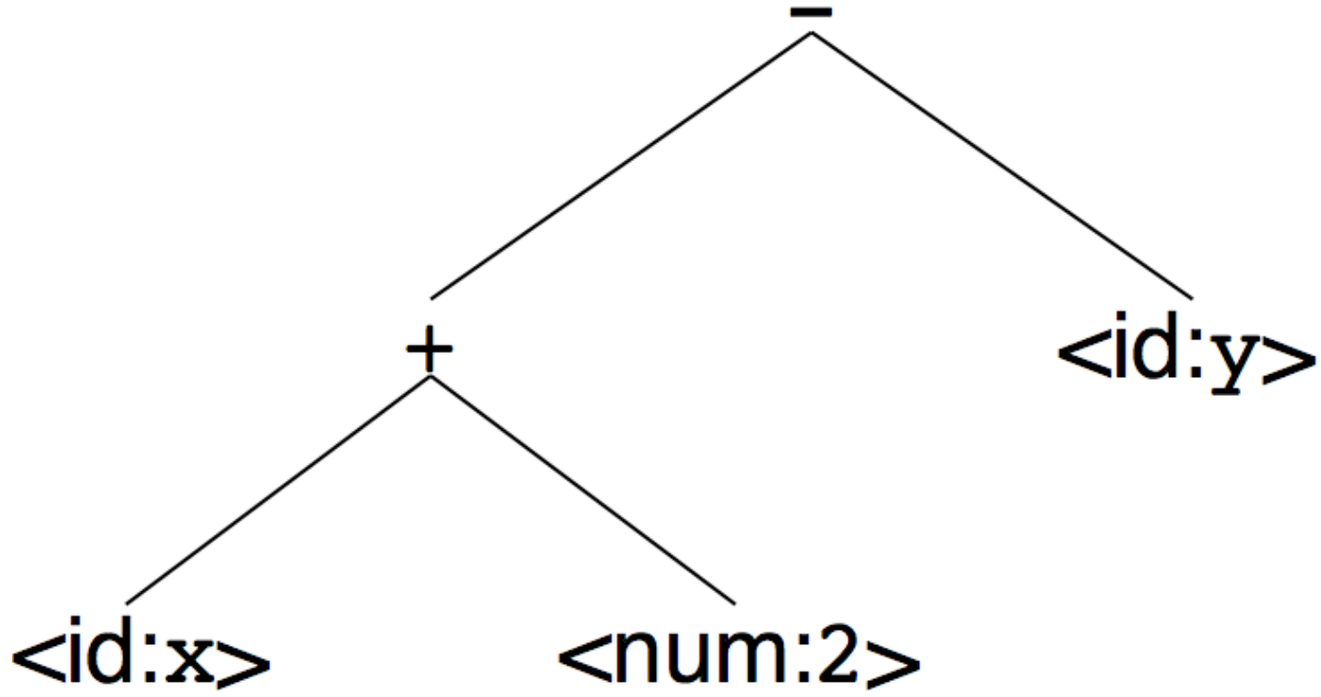
A parse can be represented by a tree called a *parse tree* (or *syntax tree*).



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.

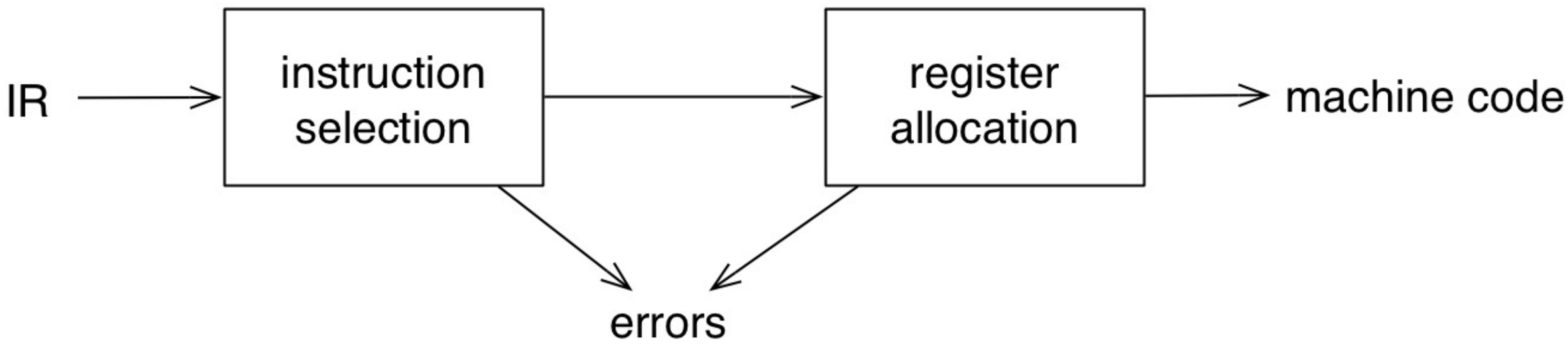
A concrete syntax tree contains a node for every single rule in the derivation of a program from its grammar. This usual contains far to much information, as grammars typically contain many rules that exists solely to resolve ambiguity. For this reason a parser often constructs instead an abstract syntax tree (AST) that eliminates uninteresting nodes.

https://en.wikipedia.org/wiki/Abstract_syntax_tree

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

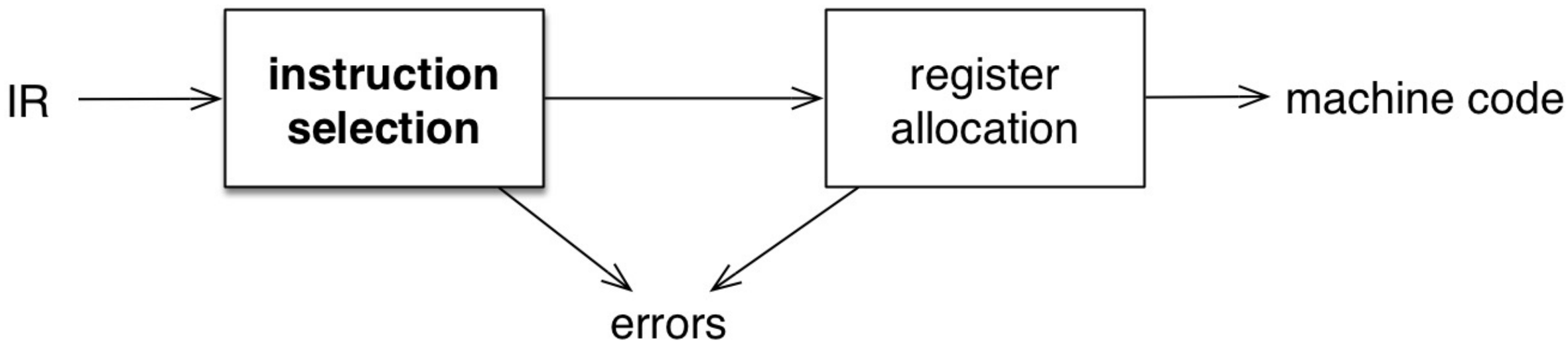
Automation has been less successful here

The back end of a compiler is responsible for transforming the intermediate representation produced by the front end into executable machine code. This entails efficient generation of machine instructions as well as allocation of registers.

Typically there are a very *limited number of registers* available to machine instructions, so the back end must figure out how to *minimize data transfers* between main memory and registers.

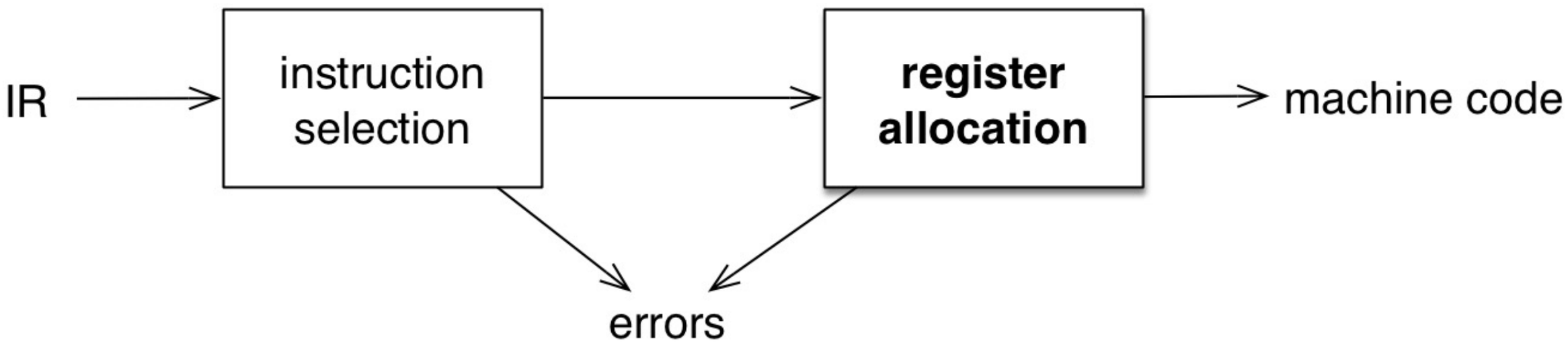
A challenge common with the front end is to process as much of the input as possible and generate useful feedback in case of errors, rather than simply failing on the first error.

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



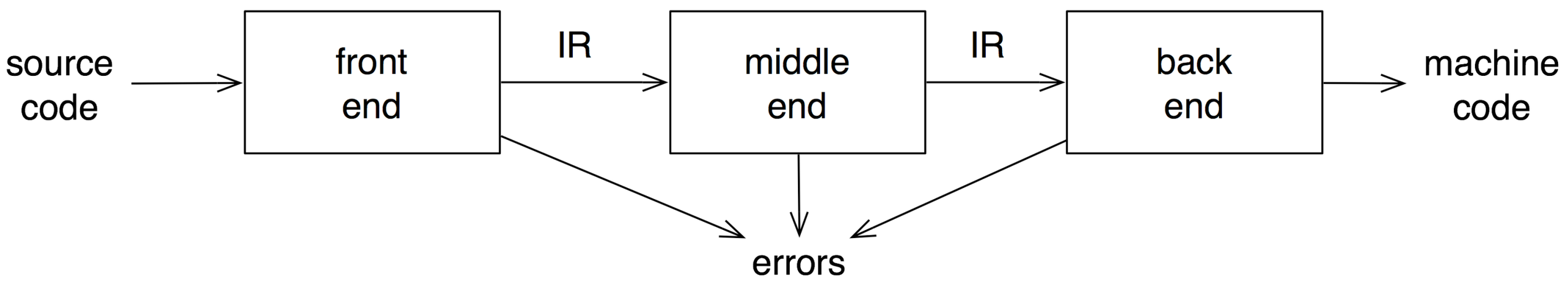
- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult

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

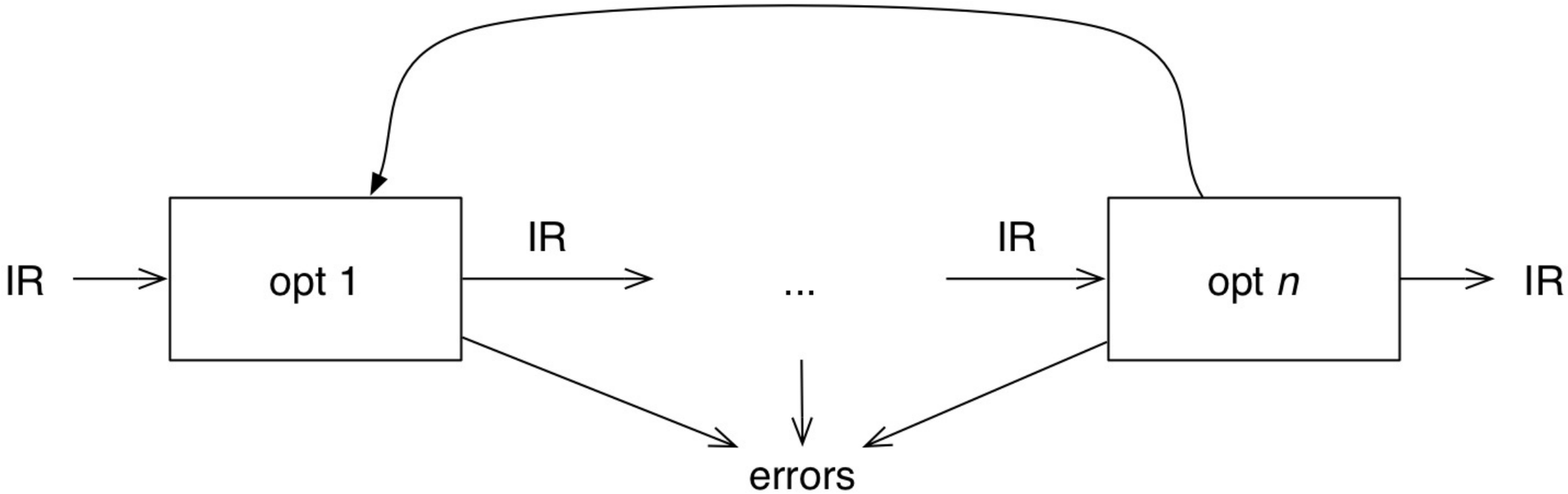
This, again, has to do with separation of concerns: the front end deals with the language, the back end deals with the target architecture, so who deals with the IR? Well, the “middle end”.

The middle end is responsible for massaging and optimizing the IR so that the back end can do a good job producing efficient machine code.

Choosing the “right” IR is not an easy task: source code is too high level for performing many analyses, and machine code is too low level. Sometimes an AST is used (*i.e.,* closer to source code), and sometimes a special kind of abstract instruction set (*i.e.,* closer to machine code).

Optimizer (middle end)

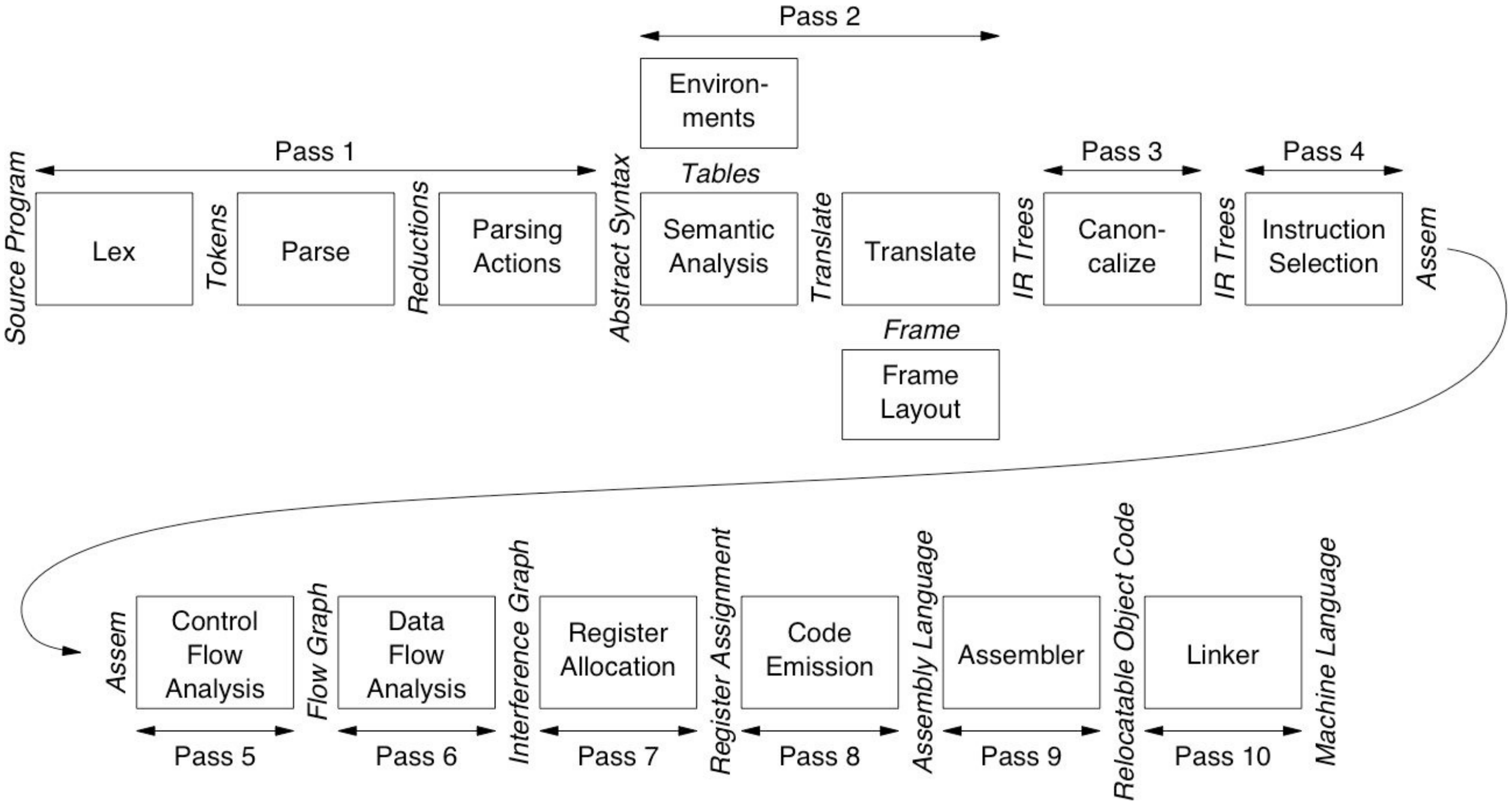
Modern optimizers are usually built as a set of passes



- 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 re-evaluated 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



This figure is from the main textbook for this course, “Modern compiler implementation in Java” (Second edition), 2002, by Andrew Appel. MiniJava is the example programming language used as a running example.

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)
```

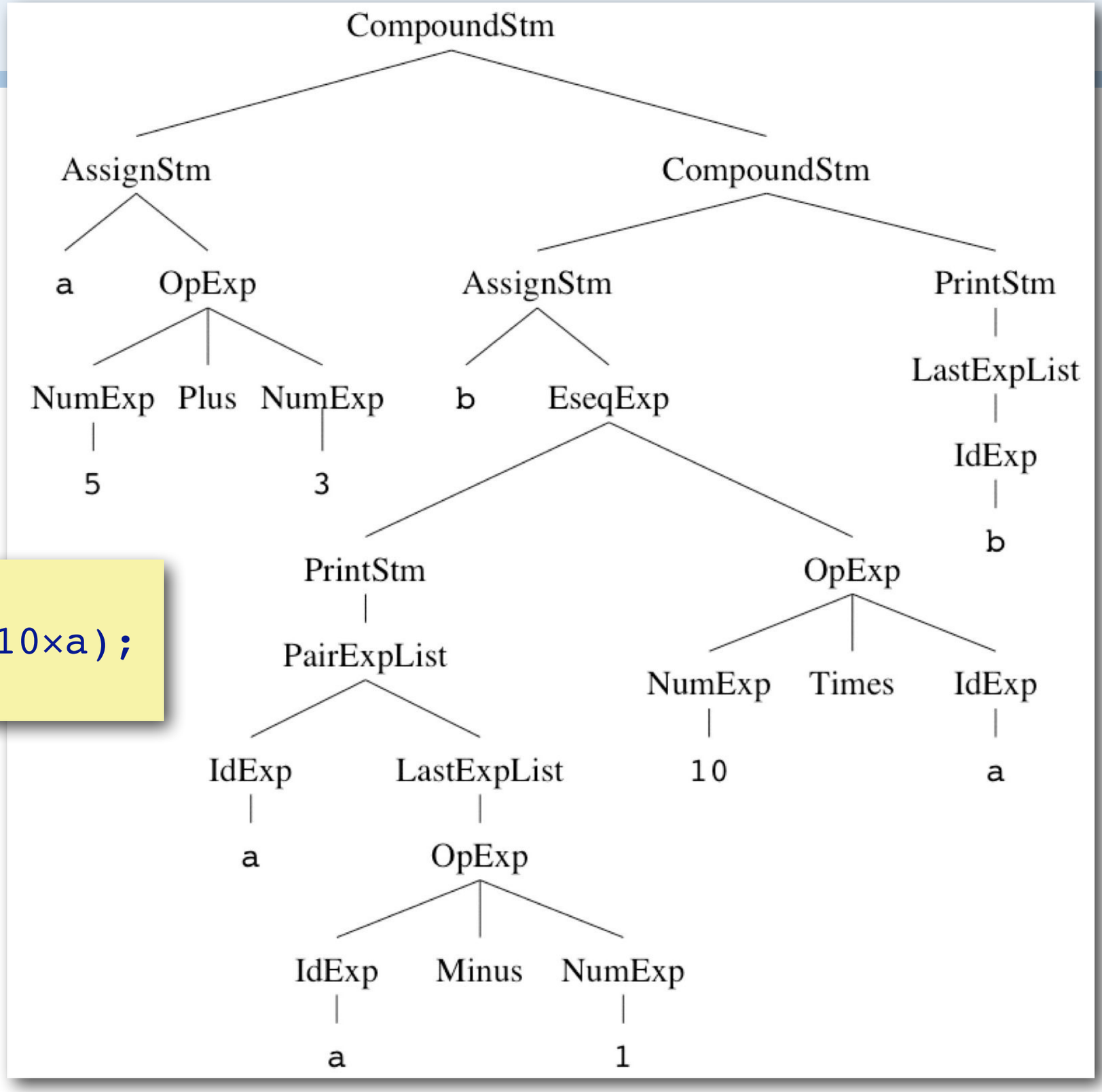
prints

```
8 7
80
```

A “straight-line” programming language consists purely of sequential code without any loops, branches or conditionals. This toy language allows you to compute simple arithemtic expressions, assign them to variables labeled “a” to “z”, and print out intermediate values.

Tree representation

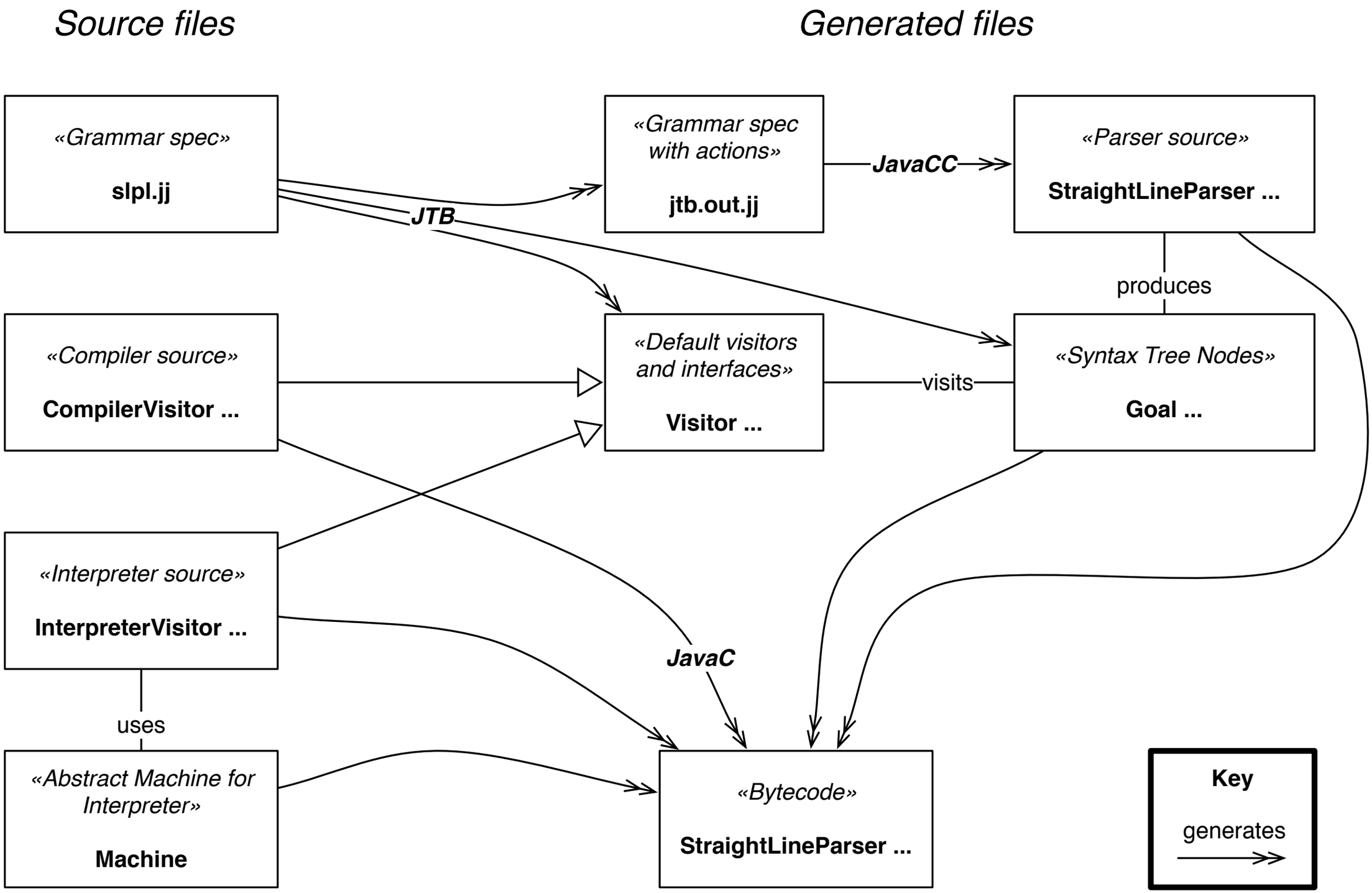
```
a := 5 + 3;
b := (print(a,a–1),10×a);
print(b)
```



The parse tree represents the parse of the given source code. Note that this is not an AST but a concrete syntax tree: it contains a node for every grammar rule used in the parse.

Exercise: what might an equivalent AST look like?

Straightline Interpreter and Compiler Files



In this course we will see two implementations of the straight-line language. One is an interpreter and the other a compiler (to Java bytecode). We will see them in the lectures on “Parsing in Practice” and “Code Generation”.

This diagram shows the input files at the left. From the grammar specification file, `slpl.jj`, the Java Tree Builder (JTB) generates (1) an augmented grammar specification with actions to build a (concrete syntax) parse tree from a given source file. JTB also generates (2) Java source code for the syntax tree nodes, and (3) default visitors and interfaces to visit a parse tree.

The augmented grammar file is processed by JavaCC (the Java “Compiler Compiler”) to generate the source code of a `StraightLineParser`, *i.e.*, which parses straight-line code and produces parse trees.

The straight-line compiler and interpreter are then implemented as visitors of the parse tree. The `CompilerVisitor` will visit parse trees and produce Java bytecode, whereas the `InterpreterVisitor` will interpret the parse tree with the help of a specially designed “abstract machine” for the straight-line language.

The source code is available here:

```
git clone git://scg.unibe.ch/lectures-cc-examples
```

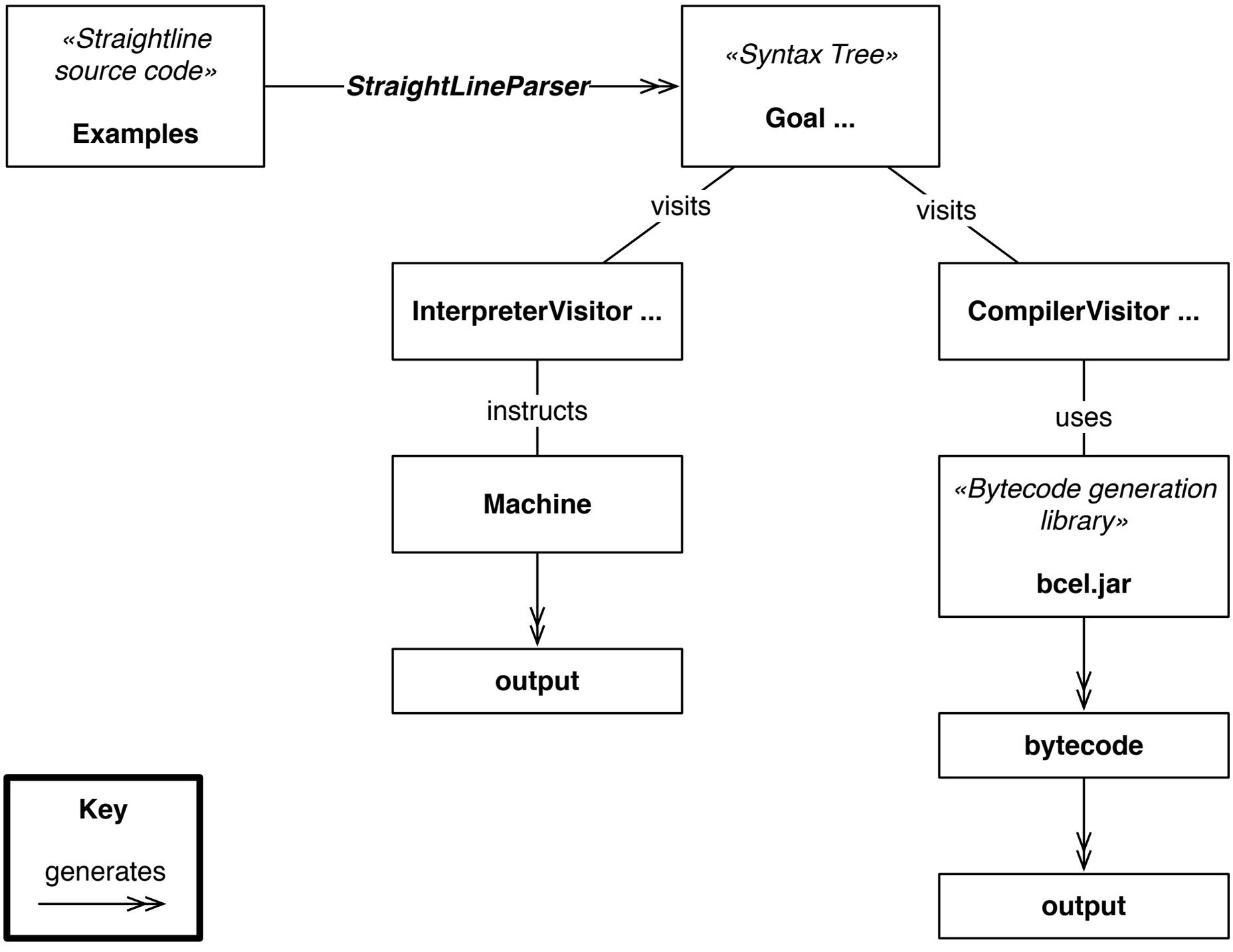
Java classes for trees

```
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;}
}
```

Here are the syntax tree nodes automatically generated by JTB for the straight-line language.

Straightline Interpreter and Compiler Runtime



Here we see the run-time view of both the interpreter and the compiler. In both cases source code is parse to generate a parse tree as an intermediate representation.

The interpreter visits the tree and produces instructions for the tailor-made abstract machine, and producing the output of running the straight-line code.

The compiler, on the other hand, visits the tree and uses the BCEL bytecode generation library to produce as output executable Java bytecode representing the straight-line code. Executing this bytecode should then produce the same output as that of the interpreter.

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?*