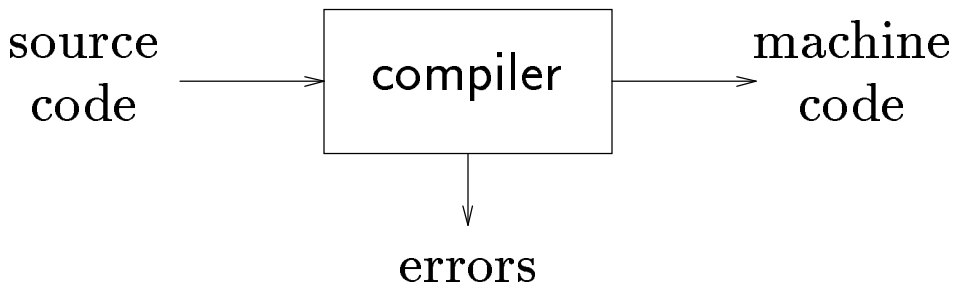


Abstract view

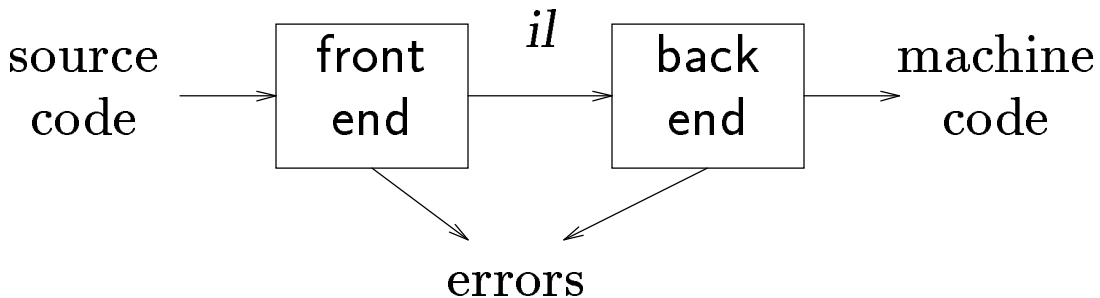


Implications:

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

Big step up from assembler – higher level notations

Traditional two pass compiler



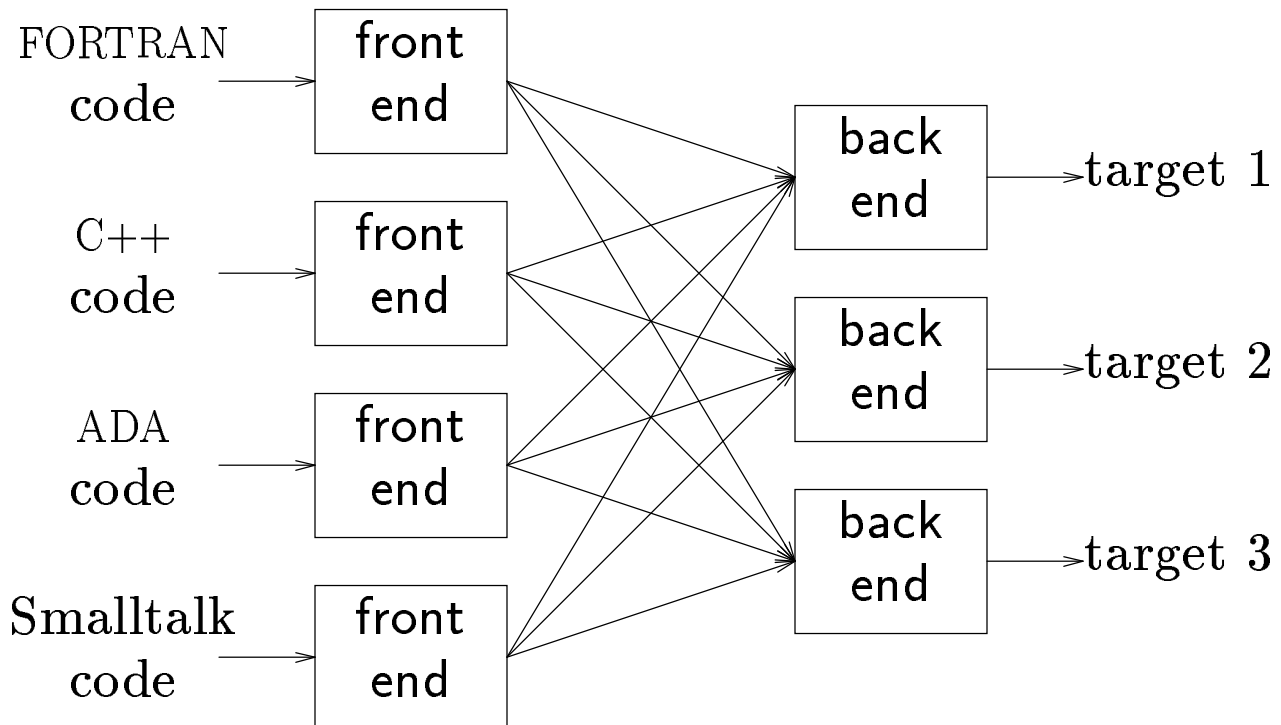
Implications:

- intermediate language (*il*)
- front end maps legal code into *il*
- back end maps *il* onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes \Rightarrow better code

Front end is $O(n)$ or $O(n \log n)$

Back end is NP-Complete

A fallacy

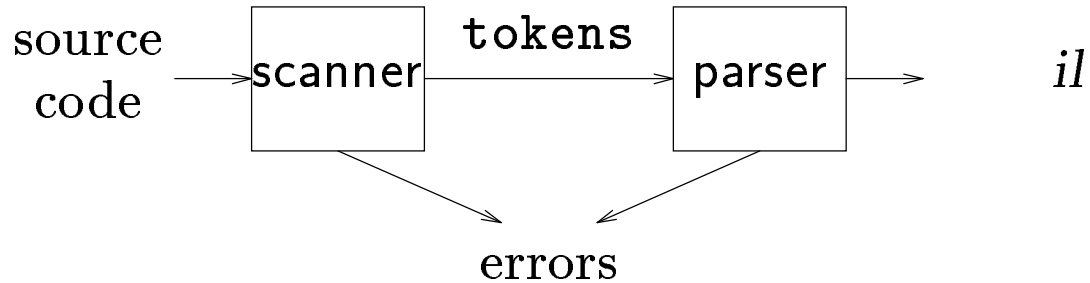


Can we build $n \times m$ compilers with $n + m$ components?

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

Limited success with low-level ils

Front end

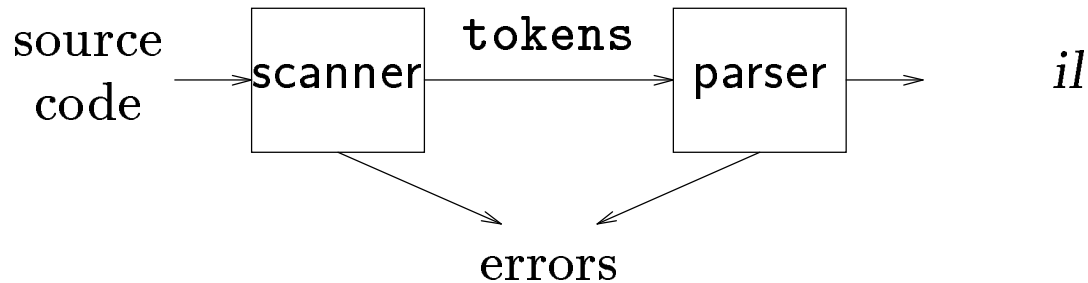


Responsibilities:

- recognize legal procedure
- report errors
- produce *il*
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Scanner



Scanner

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

`x = x + y;`

becomes

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

- character string 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 (**lex**)

Specifying patterns

A scanner must recognize various parts of the language's syntax.

Some parts are easy:

white space

some combination of `< \b >` and `tab`

keywords and operators

specified as literal patterns — `do`, `end`

comments

opening and closing delimiters — `/* ... */`

Specifying patterns

Other parts are much harder:

identifiers

alphabetic followed by k alphanumerics

(-, \$, &, ...)

numbers

integers — 0 or digit from 1-9 followed by
digits from 0-9

decimals — integer “.” digits from 0-9

reals — (integer or decimal) “E” (+ or -) digits
from 0-9

complex — “(” real “,” real “)”

We need a powerful notation to specify these patterns.

Definitions

Operation	Definition
<i>union of L and M written $L \cup M$</i>	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
<i>concatenation of L and M written LM</i>	$LM = \{st \mid s \in L \text{ and } t \in M\}$
<i>Kleene closure of L written L^*</i>	$L^* = \cup_{i=0}^{\infty} L^i$
<i>positive closure of L written L^+</i>	$L^+ = \cup_{i=1}^{\infty} L^i$

Aho, Sethi, and Ullman, Figure 3.8

Regular expressions

Patterns are often specified as *regular languages*.

Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*.

Regular expressions (over an alphabet Σ):

1. ϵ is a RE denoting the set $\{\epsilon\}$
2. if $a \in \Sigma$, then a is a RE denoting $\{a\}$
3. if r and s are REs, denoting $L(r)$ and $L(s)$,
then:

(r) is a RE denoting $L(r)$

$(r) \mid (s)$ is a RE denoting $L(r) \cup L(s)$

$(r)(s)$ is a RE denoting $L(r)L(s)$

$(r)^*$ is a RE denoting $L(r)^*$

If we adopt a *precedence* for operators, the extra parentheses can go away. We assume *closure*, then *concatenation*, then *alternation* as the order of precedence.

RE examples

identifier

$letter \rightarrow (a \mid b \mid c \mid \dots \mid z \mid A \mid B \mid C \mid \dots \mid Z)$

$digit \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

$id \rightarrow letter (letter \mid digit)^*$

numbers

$integer \rightarrow$

$(+ \mid - \mid \epsilon) (0 \mid (1 \mid 2 \mid 3 \mid \dots \mid 9) (digit)^*)$

$decimal \rightarrow integer . (digit)^*$

$real \rightarrow (integer \mid decimal) \mathbf{E} (+ \mid -) (digit)^+$

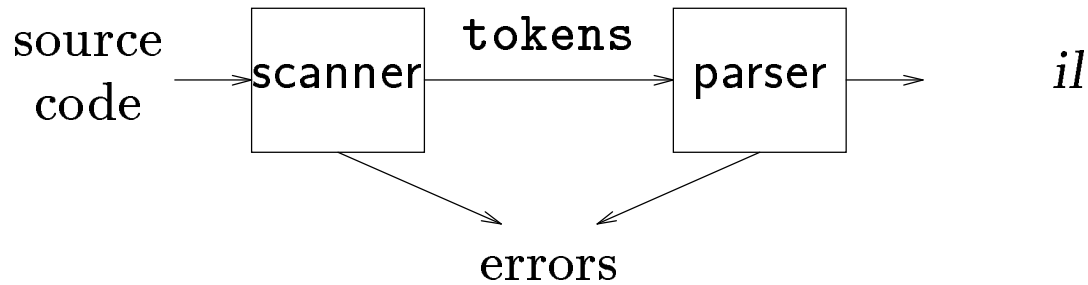
$complex \rightarrow "(real \text{ , } real)"$

Numbers can get much more complicated

Most programming language tokens can be described with regular expressions.

We can use regular expressions to automatically build scanners.

Parser



Parser:

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

Parser generators mechanize much of the work

Grammar

Context-free syntax is specified with a *grammar*.

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

This grammar defines the set of noises that a sheep makes under normal circumstances.

The format is called *Backus-Naur form*. (BNF)

Formally, 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* or *rewrite rules*

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

Substitution

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  |         | -
```

This grammar defines simple expressions with addition and subtraction over the tokens **id** and **number**.

$$S = \langle \text{goal} \rangle$$

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

$$N = \langle \text{goal} \rangle, \langle \text{expr} \rangle, \langle \text{term} \rangle, \langle \text{op} \rangle$$

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

Parse tree

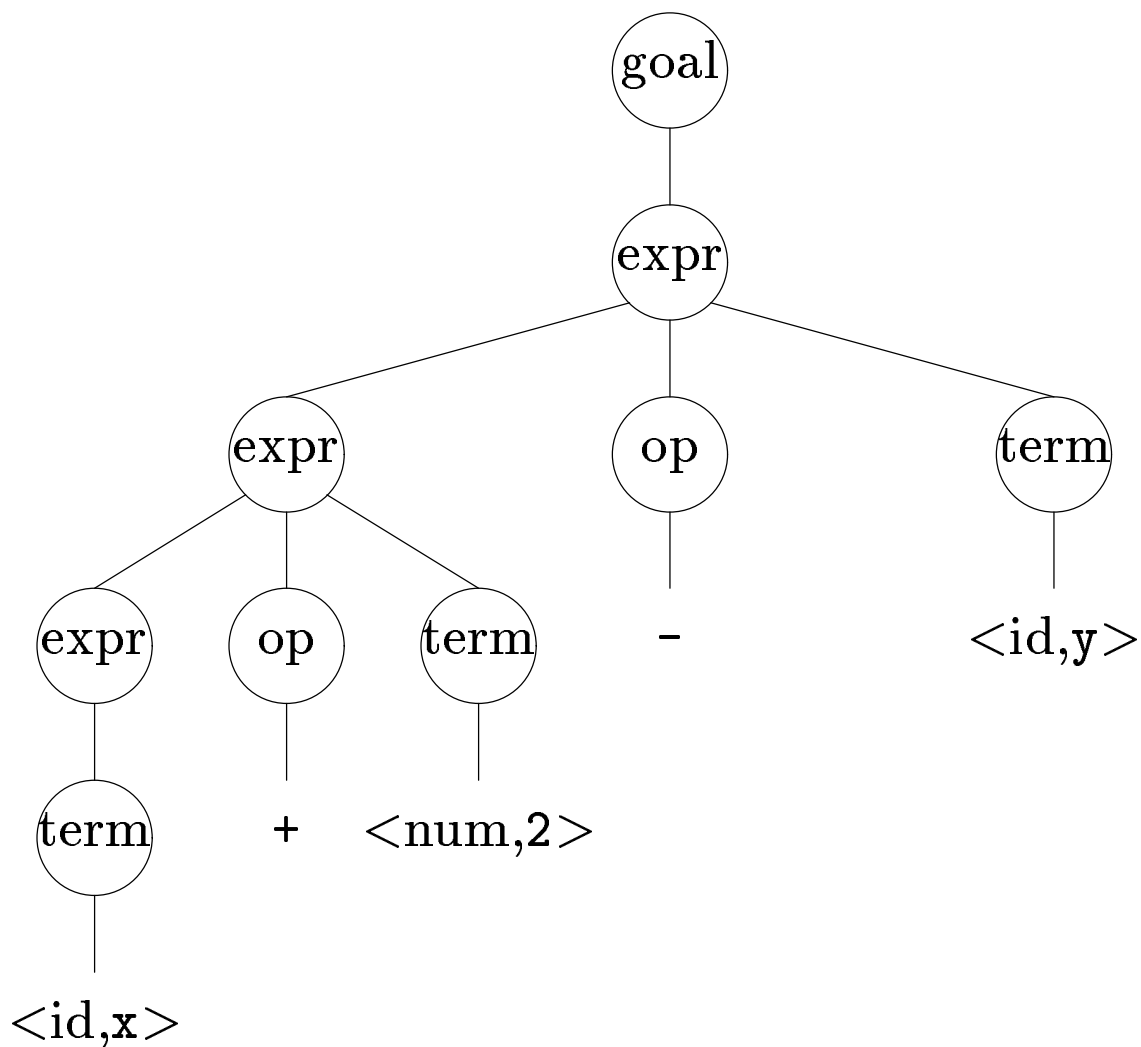
Given a grammar, valid sentences can be derived by repeated substitution.

Prod'n.	Result
	$\langle \text{goal} \rangle$
1	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{term} \rangle$
5	$\langle \text{expr} \rangle \langle \text{op} \rangle y$
7	$\langle \text{expr} \rangle - y$
2	$\langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{term} \rangle - y$
4	$\langle \text{expr} \rangle \langle \text{op} \rangle 2 - y$
6	$\langle \text{expr} \rangle + 2 - y$
3	$\langle \text{term} \rangle + 2 - y$
5	$x + 2 - y$

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

Parse tree

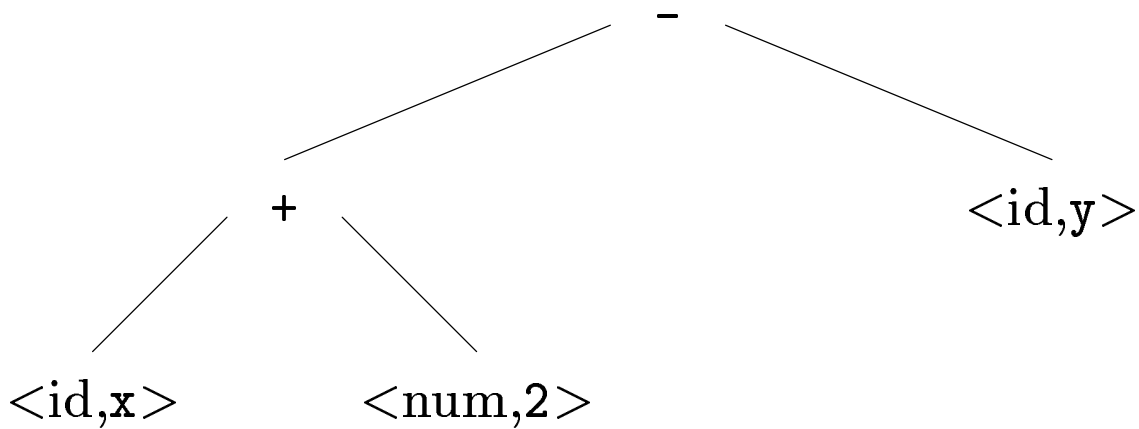
A parse can be represented by a tree, called a *parse tree* or a *syntax tree*.



Obviously, this contains a lot of unneeded information.

Abstract syntax tree

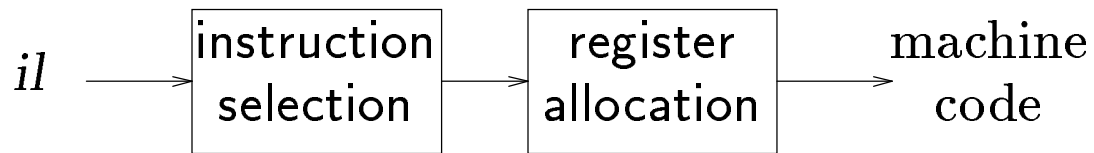
So, compilers often use an *abstract syntax tree*.



This is much more concise.

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

Back end

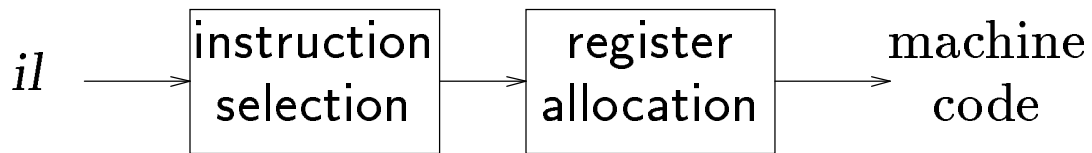


Responsibilities

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

Automation has been less successful here

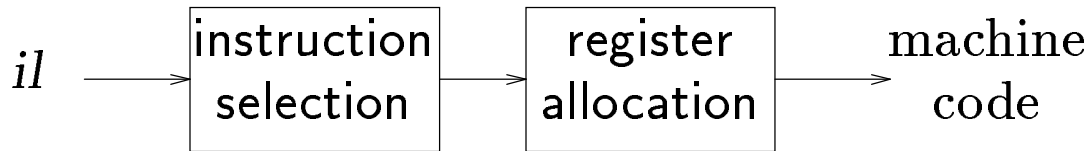
Instruction selection



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

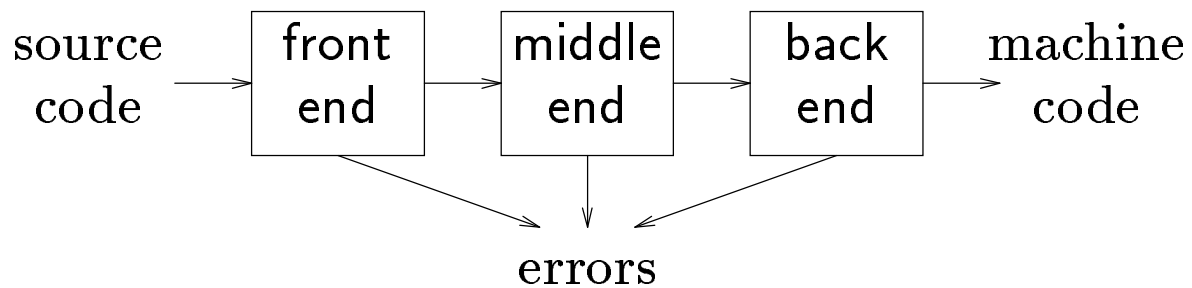


Register Allocation

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
 - ⇒ NP-complete for 1 or k registers

Modern allocators often use an analogy to graph coloring

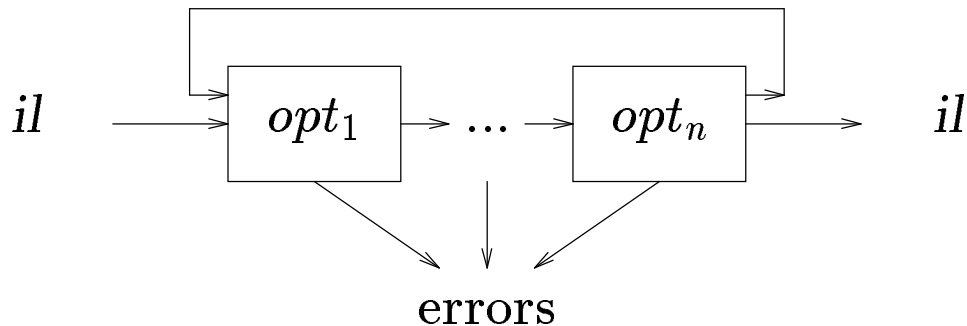
Traditional three pass compiler



Code Improvement

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

Optimizer (middle end)



Modern optimizers are usually built as a set of passes.

Typical passes

- discover & propagate constant values
- reduction of operator strength
- common subexpression elimination
- redundant computation elimination
- encode an idiom in some powerful instruction
- move computation to less frequently executed place (*e.g.*, out of loops)