Abstract View

The Structure of a Compiler

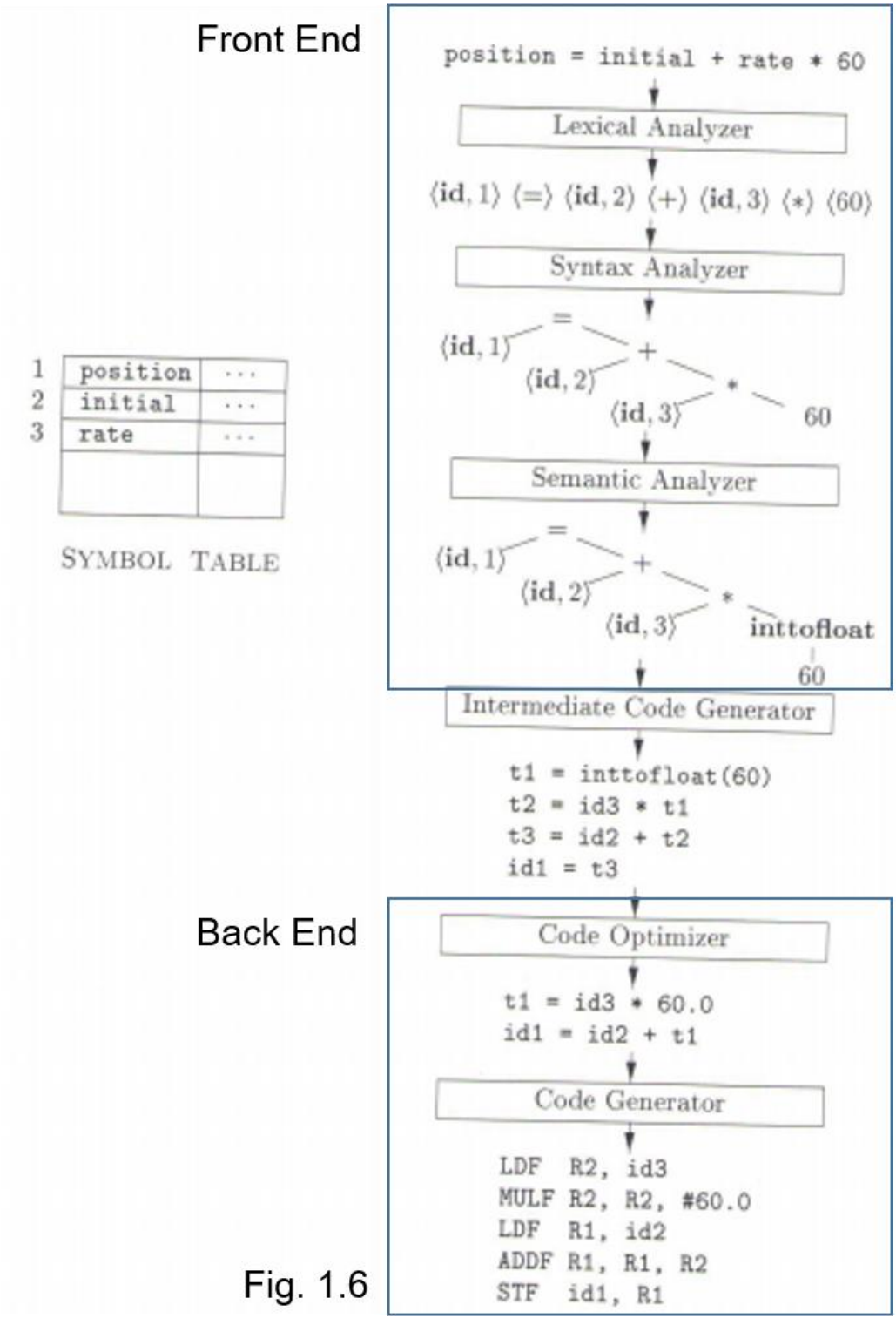
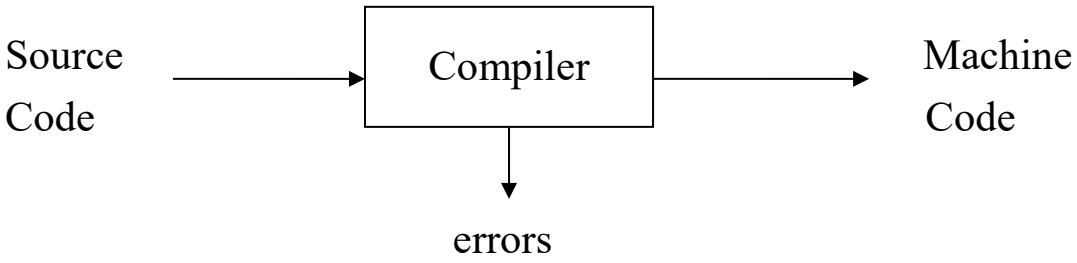


Fig. 1.6

Abstract View

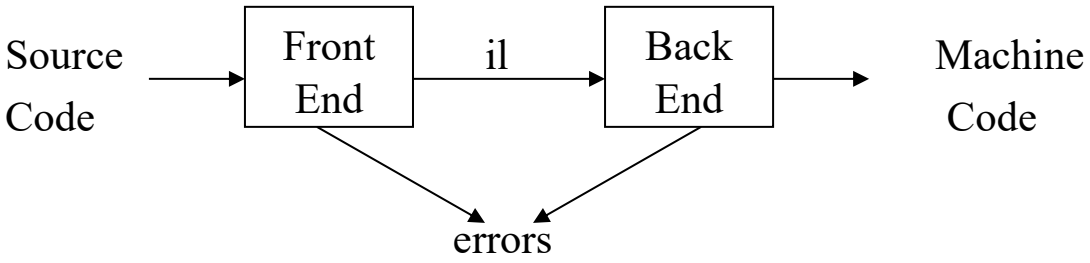


Implications:

- 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 notation

Traditional two pass compiler



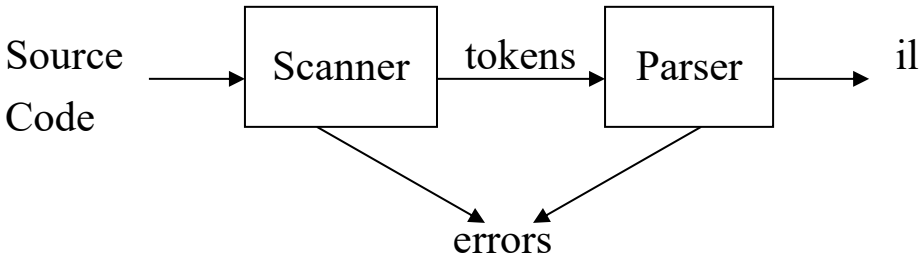
Implications:

- 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 => better code

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

Back end is NP-complete

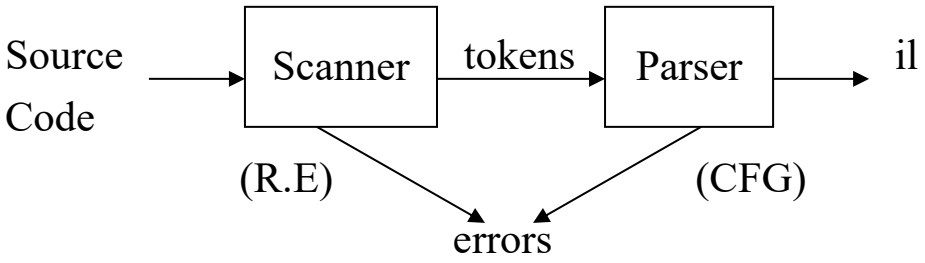
Front end



Responsibilities:

- Recognize legal procedure
- Report errors
- Produce il
- Preliminary storage map
- Shape the code for the back end

Scanner



Scanner

- Maps characters into tokens – the basic unit of syntax or terminal symbols in grammar

  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)

Terminology:

pattern – rule that describes a set of strings belongs to a token.

lexeme – a sequence of characters in the source program that is matched by the pattern for a token

alphabet – any finite set of symbols eg. {0, 1} is the binary alphabet

string – a finite sequence of symbols drawn from an alphabet

language – any countable set of strings over some fixed alphabet

Specifying patterns

A scanner must recognize various parts of the language syntax.

Some parts are easy:

white space

Some combination of *blank* and *tab*

keywords and operators

Specified as literal patterns – do, end

comments

Opening and closing delimiters - /* … */

Other parts are much harder:

identifiers

Alphabet 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 decimals) “E” (+ or -)digits from 0-9

complex – “(” real “.” real “)”

We need a powerful notation to specify these patterns.

Definitions of operations on languages

Operation	Definition
Union of L and M Written $L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M Written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L Written L^*	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L Written L^+	$L^+ = \cup_{i=1}^{\infty} L^i$

Aho, Sethi, and Ullman, Figure 3.6

Ex. 3.3 : L = {A, B, …, Z, a, b, …, z} and D = {0,1, …, 9}

1. $L \cup D$
2. LD
3. L^4
4. L^*
5. $L (L \cup D)^*$
6. D^+

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 $\sum$, where $\sum$ means finite set of symbols):

1. ε is a RE denoting the set $\{\varepsilon\}$
2. if a $\in \sum$, 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) | (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 → (a|b|c|…|z|A|B|C|…|Z)

digit → (0|1|2|3|4|5|6|7|8|9)

id → letter(letter|digit)*

numbers

integer → (+|–|ε)(0| (1|2|3|4|5|6|7|8|9) (digit)*)

decimal → integer . (digit)*

real → (integer|decimal) E (+|–|)(digit)$^+$

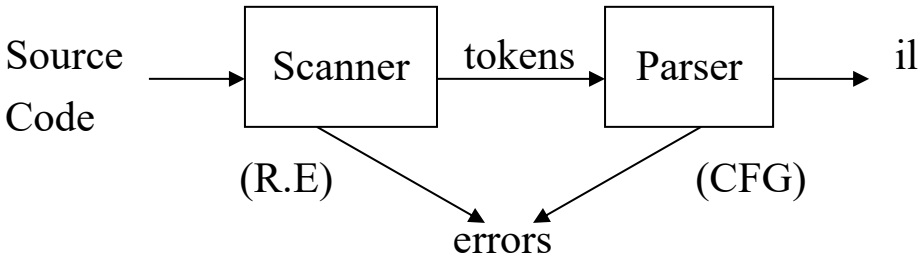
complex → “(” real “,” real “)”

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

<sheep noise> ::= baa
| baa <sheep noise>

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

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

Example

Context-free syntax specifying expression

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

Parse Tree

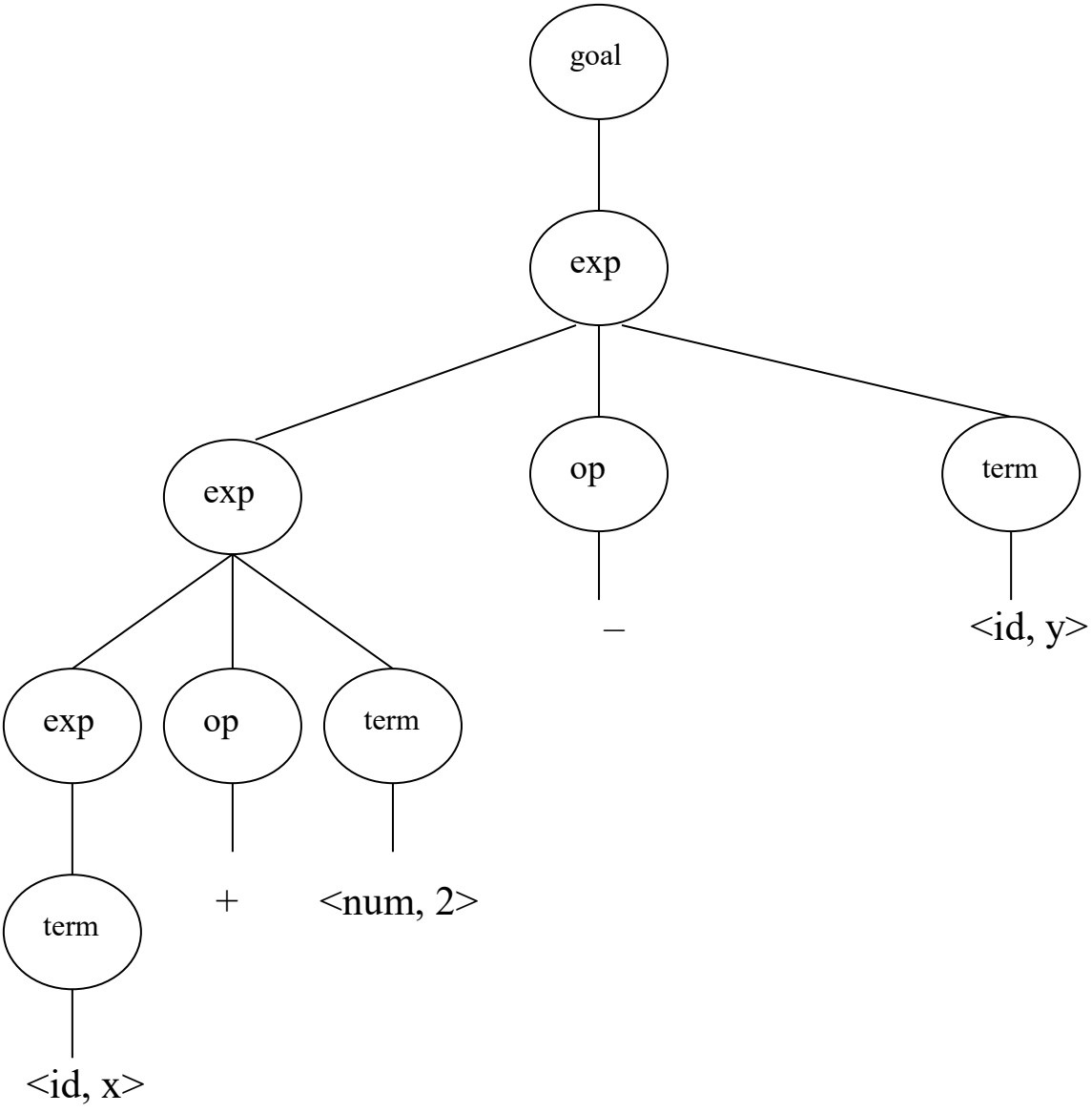
Given a grammar, valid sentences can be derived by repeated substitution.

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
4	x + 2 – y

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

Parsing Tree

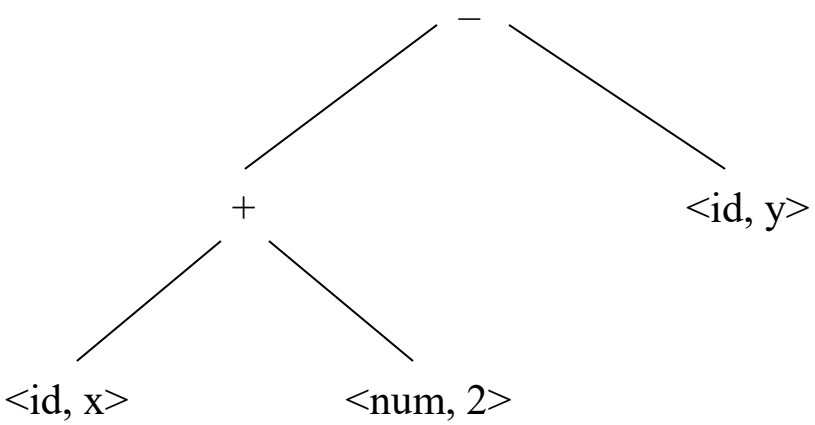
A parsing process can be represented by a tree, called a parsing tree or a syntax tree.



Obviously, this contains a lot of unneeded information

Abstract syntax tree

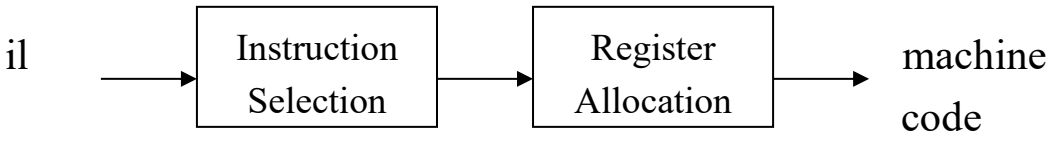
So, compilers often use an abstract syntax tree



This is much more concise.

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

Back end

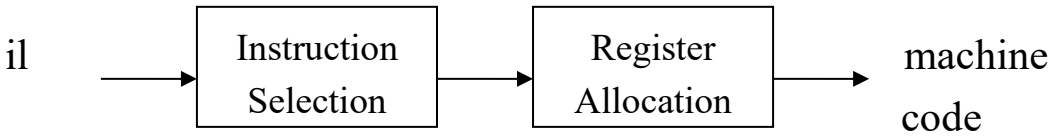


Responsibilities

- 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

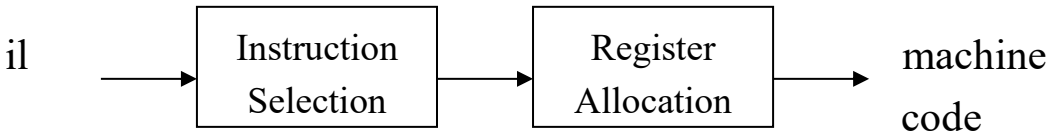
Instruction Selection



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

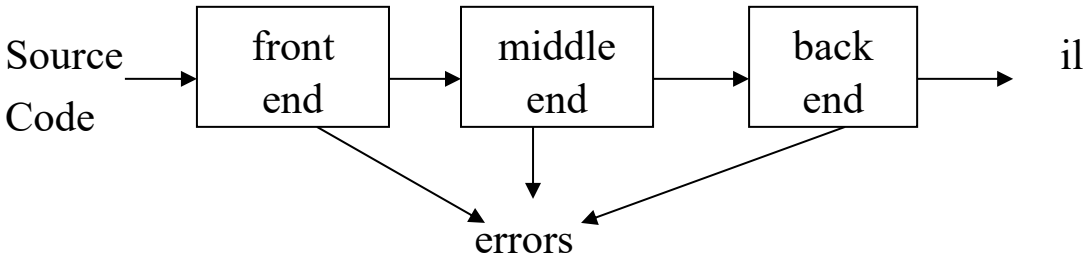


Register Allocation

- 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

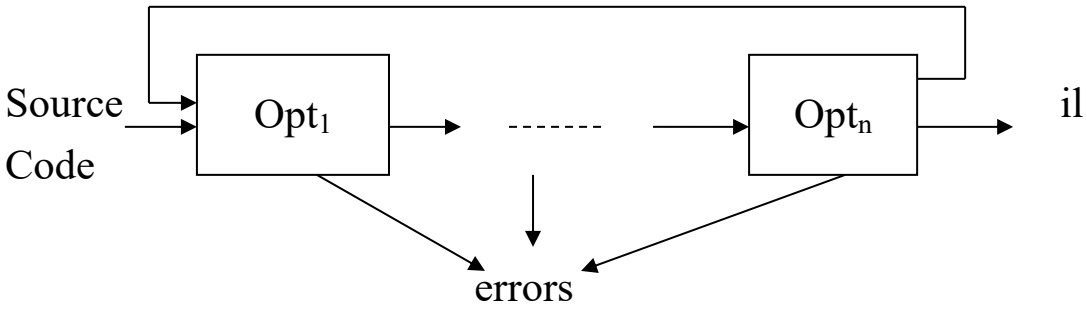
Traditional three pass compiler



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)