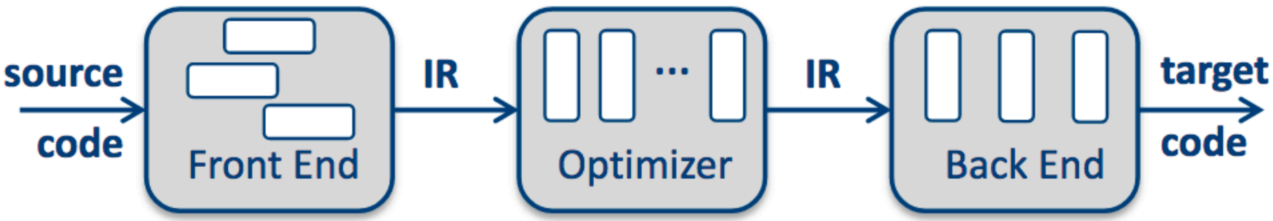
Overview of the Course



The organization of this course

Schedule :

Two weekly lectures

Thursday	14:00	16:00	L1
Friday	9:00	11:00	L1

Roberta Gori

roberta.gori@di.unipi.it

One weekly lecture for constructing a compiler

Wednesday	14:00	16:00	L1

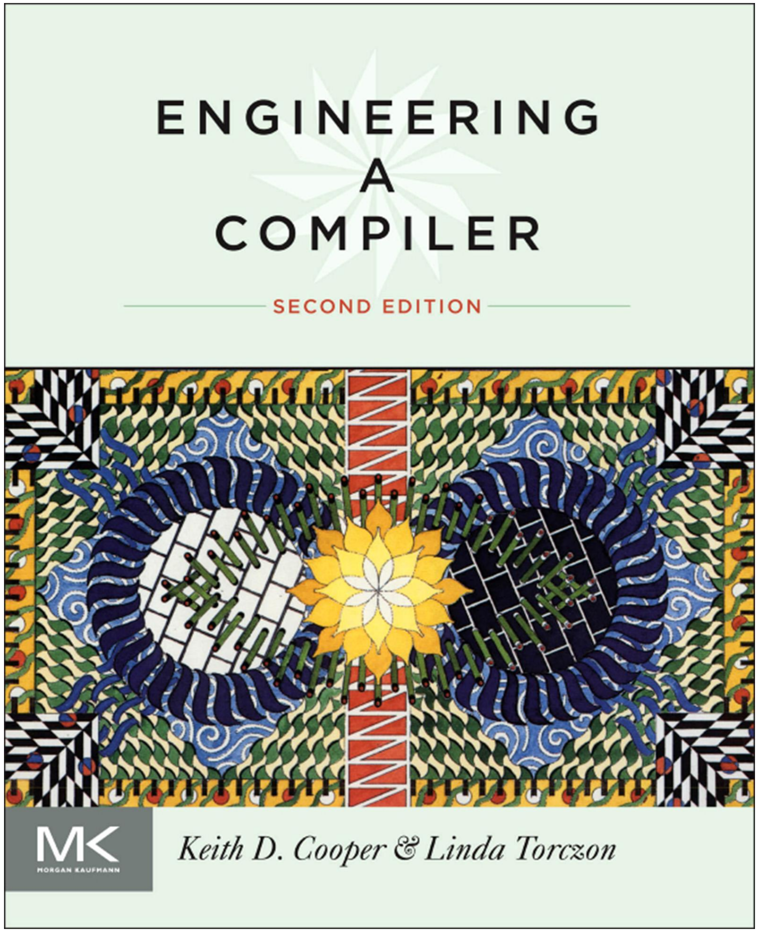
Letterio Galletta

letterio.galletta@imtlucca.it

What we will see

- A recall on formal languages:
    - Grammars, automata, theorems, regular and context free languages
    - Chomsky hierarchy
- Lexical analysis
- Parser
- Contextual analysis
- Intermediate representation
- Code shape
- Optimization
- Dataflow analysis
- More static analyses: control flow and abstract interpretation
- Register allocation
- Register allocation

Our textbook



Other information

- web page, I will add there all the slides

www.di.unipi.it/~gori/Linguaggi-Compilatori2021

Material for specific topics:

- Introduction to Automata Theory, Languages, And Computation.
  Hopcroft, Motwani, Ullman
- Fondamenti dell'Informatica. Linguaggi formali, calcolabilita' e complessita'.
  Dovier, Giacobazzi
  Bollati Boringhieri
- Principles of Program Analysis.
  Nielson,Nielson, Hankin
  Springer
- Static Inference of Numeric Invariants by Abstract Interpretation a tutorial by Antoine Mine on Abstract interpretation.

About this teacher

Roberta Gori
roberta.gori@di.unipi.it

My own research program

- Whole program analysis for verification and optimization
- Static analysis to discern program behavior
- Abstract interpretation based techniques

Compilers

- What is a compiler?
  - A program that translates an executable program in one language (source) into an executable program in another language (target)
  - The compiler should improve the program, in some way
- What is an interpreter?
  - A program that reads an executable program and an input and produces the results of executing that program on the input
- C and C++ are typically compiled,

  Pyton and Scheme are typically interpreted
- Java is complicated
  - Compiled to bytecodes (code for the Java VM)
  - which are then interpreted
  - Or a hybrid strategy is used
    → Just-in-time compilation

Common mis-statement:
X is an interpreted language
(or a compiled language)
It's a property of the implementation !

Compilers vs Interpreters

Compiler scans the whole program in one go.	Translates program one statement at a time.
It converts the source code into object code.	It does not convert source code into object code instead it scans it line by line
The translation is performed before executing	The translation and execution is performed at the same time
Good execution time.	Slow in executing the object code.
It does not require source code for later execution.	It requires source code for later execution.
The errors are shown at the end together.	Errors are shown line by line.

Why Study Compilation?

- Compilers are important
  - Responsible for many aspects of system performance
  - Attaining performance has become more difficult over time
    - → In 1980, typical code got 85% or more of peak performance
    - → Today, that number is closer to 5 to 10% of peak
    - → Compiler has become a prime determiner of performance
- Compilers are interesting
  - Compilers are complex program of millions of lines
  - Compilers include many applications of theory to practice
  - Writing a compiler exposes algorithmic & engineering issues
- Compilers are everywhere
  - Many practical applications have embedded languages
    - → Commands, macros, formatting tags …
  - Many applications have input formats that look like languages

Still many open problems!

Fundamental Principles of Compilation

- The compiler must preserve the meaning of the program being compiled
- The compiler must improve the input program

Reducing the Price of Abstraction

Computer Science is the art of creating virtual objects and making them useful.

- We invent abstractions and uses for them
- We invent ways to make them efficient
- Programming is the way we realize these inventions

Well written compilers make abstraction affordable

- Cost of executing code should reflect the underlying work rather than the way the programmer chose to write it
- Change in expression should bring small performance change
- Cannot expect compiler to devise better algorithms
  - Don't expect bubblesort to become quicksort

Making Languages Usable

It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that if we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus on the subject of the 1st FORTRAN compiler

Era nostra convinzione che se FORTRAN, durante i suoi primi mesi, avesse tradotto un qualsiasi programma sorgente "scientifico" ragionevole in un programma oggetto piu' efficiente solo della metà della sua controparte codificata a mano, allora l'accettazione del nostro linguaggio sarebbe stata in serio pericolo. Credo che se non fossimo riusciti a produrre programmi efficienti, l'uso diffuso di linguaggi come FORTRAN sarebbe stato seriamente ritardato.

Simple Examples

All data collected with gcc 4.1, -O3, running on a multiuser Intel T9600 @ 2.8 GHz

Which is faster?

All three loops have distinct performance.

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    A[i][j] = 0;
```

0.51 sec on 10,000 x 10,000 array

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    A[j][i] = 0;
```

1.65 sec on 10,000 x 10,000 array

```
p = &A[0][0];
t = n * n;
for (i=0; i<t; i++)
  *p++ = 0;
```

0.11 sec on 10,000 x 10,000 array

A good compiler should know these tradeoffs, on each target, and generate the best code.

Few real compilers do.

Conventional wisdom suggests using

```
bzero((void*) &A[0][0],(size_t) n*n*sizeof(int))
```

0.52 sec on 10,000 x 10,000 array

Intrinsic Merit

- Compiler construction poses challenging and interesting problems:
  - Compilers must process large inputs, perform complex algorithms, but also run quickly
  - Compilers have primary responsibility for run-time performance
  - Compilers are responsible for making it acceptable to use the full power of the programming language
  - Computer architects perpetually create new challenges for the compiler by building more complex machines
    - → Compilers must hide that complexity from the programmer
- A successful compiler requires mastery of the many complex interactions between its constituent parts

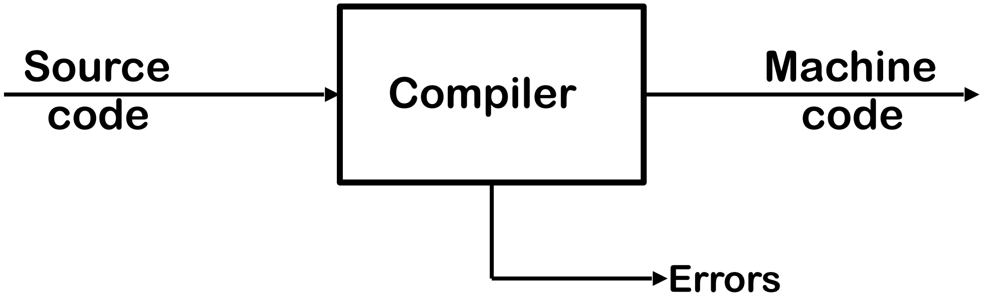
Intrinsic Interest

- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management Instruction set use

The View from 35,000 Feet

High-level View of a Compiler

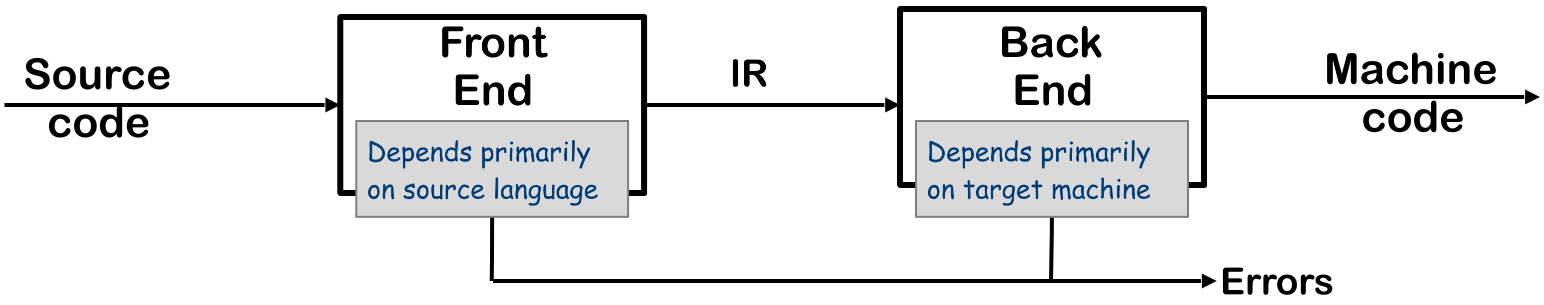


Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Big step up from assembly language

Traditional Two-pass Compiler



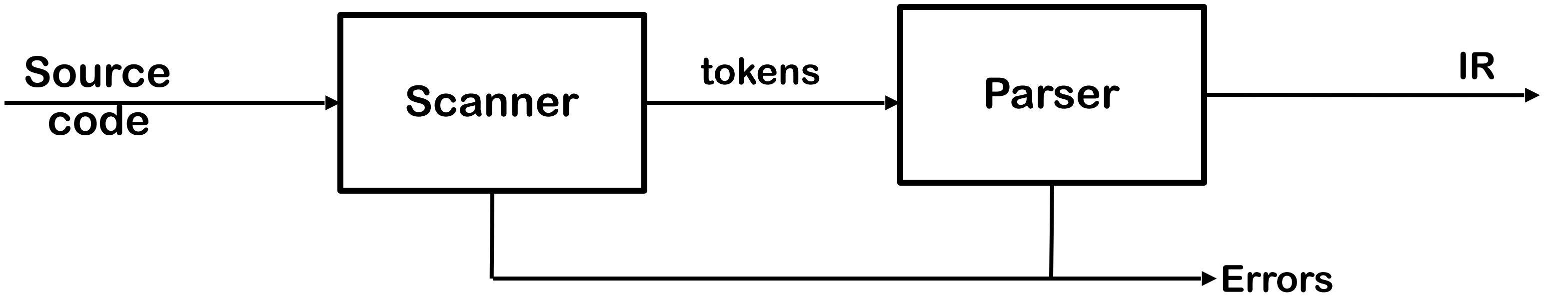
Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple passes (better code)

Classic principle from software engineering: Separation of concerns

Typically, front end is O(n) or O(n log n), while back end is NP-Complete

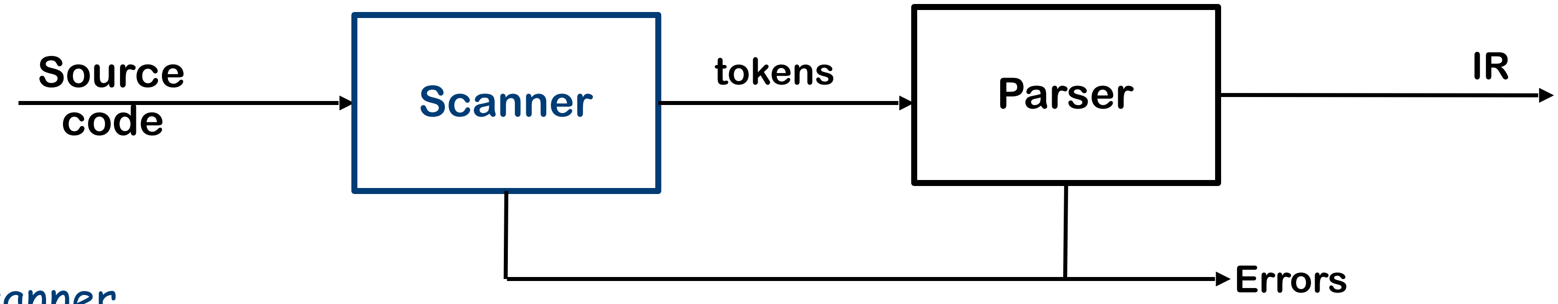
The Front End



Responsibilities

- Recognize legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the rest of the compiler
- Much of front end construction can be automated

The Front End



Scanner

- Maps character stream into words—the basic unit of syntax

  (Lexical analysis)
- Produces pairs — a word & its part of speech

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

  — word ≅ lexeme, part of speech ≅ token type, pair ≅ token ≅ a lexeme and a token type
- Typical tokens include number, identifier, +, –, new, while, if
- Speed is important

Textbooks advocate automatic scanner generation

Commercial practice appears to be hand-coded scanners

Lexical analysis

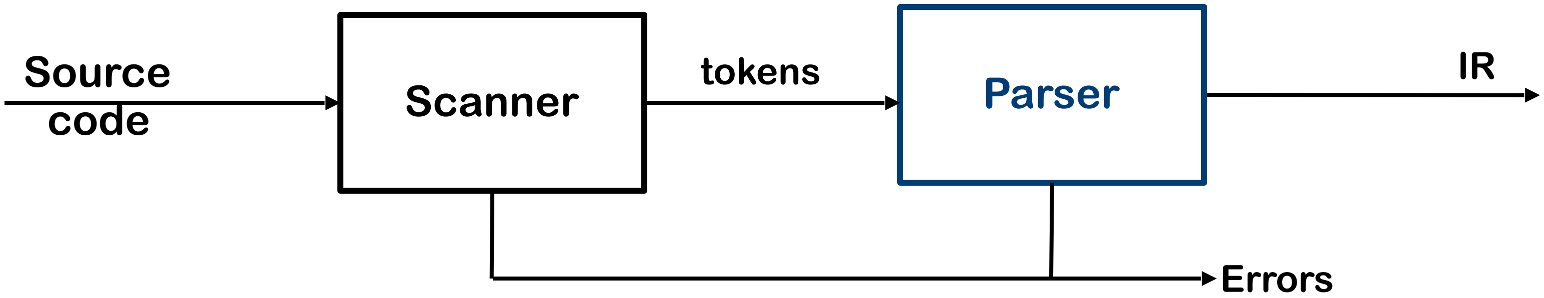
Split program to individual words that makes sense:

My mother coooooookes dinner not.

```
while (y < z) {
    int x = a + b;
    y += x; }
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

The Front End



Parser

- Recognizes context-free syntax & reports errors
  (Syntax Analysis)
- Guides context-sensitive ("semantic") analysis (type checking)
- Builds IR for source program

Hand-coded parsers are fairly easy to build

Most books advocate using automatic parser generators

The Front End

For Lexical and Syntact analysis we need grammars

$$\text{SheepNoise} \rightarrow \text{SheepNoise}\ \underline{\text{baa}}$$
$$\mid\ \underline{\text{baa}}$$

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

It is written in a variant of 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 or words
- P is a set of productions or rewrite rules $(P : N \rightarrow N \cup T)$

The Front End

Context-free syntax can be put to better use

S = *Goal*

T = { number, id, +, - }

N = { *Goal*, *Expr*, *Term*, *Op* }

P = { 1, 2, 3, 4, 5, 6, 7 }

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 & subtraction over number and id
- This grammar, like many, falls in a class called "context-free grammars", abbreviated *CFG*

The Front End

Given a CFG, we can derive sentences by repeated substitution

1.	*Goal* → *Expr*
2.	*Expr* → *Expr Op Term*
3.	| *Term*
4.	*Term* → number
5.	| id
6.	*Op* → +
7.	| -

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

A derivation

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

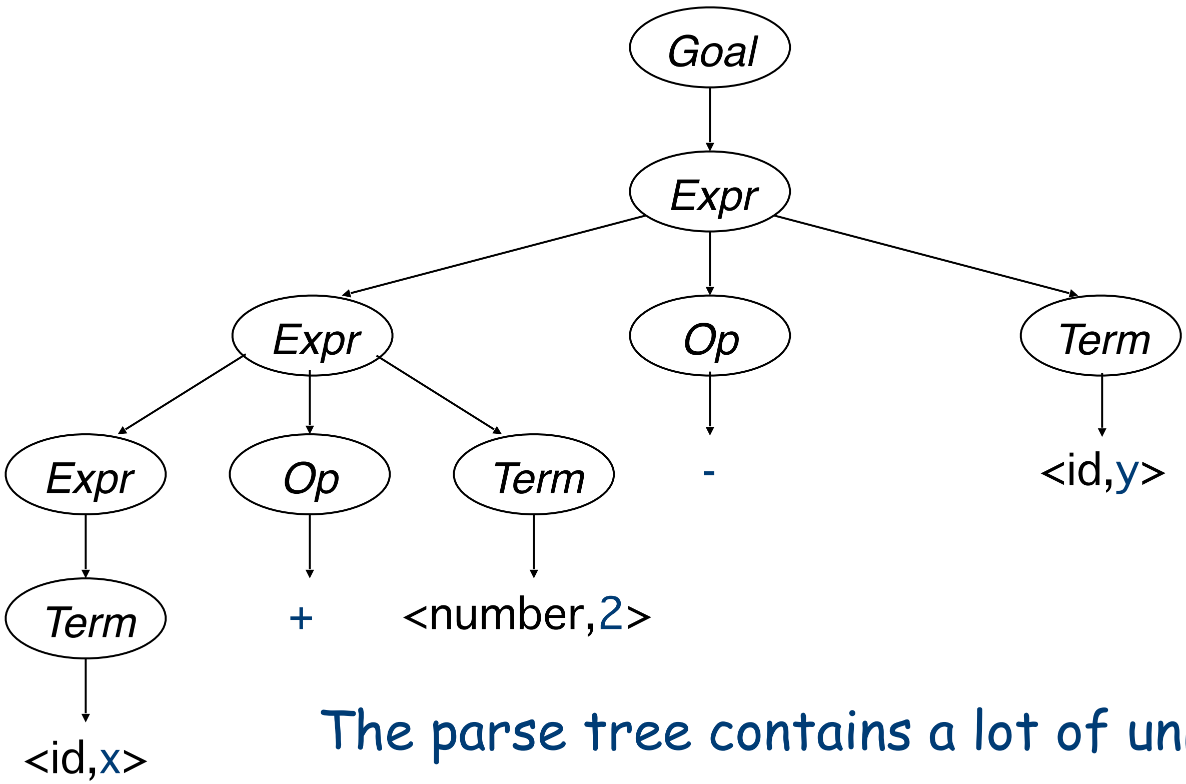
The Front End

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

x + 2 - y

1.	*Goal*	→	*Expr*
2.	*Expr*	→	*Expr Op Term*
3.		|	*Term*
4.	*Term*	→	number
5.		|	id
6.	*Op*	→	+
7.		|	-

The parse tree for x+2-y

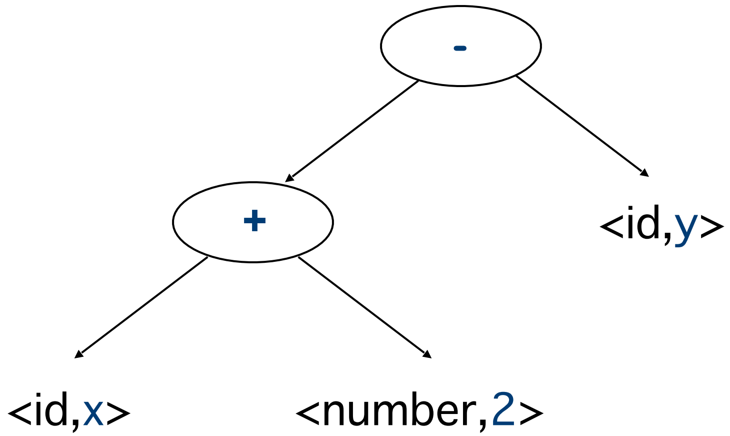


1. *Goal* → *Expr*
2. *Expr* → *Expr Op Term*
3. | *Term*
4. *Term* → number
5. | id
6. *Op* → +
7. | -

The parse tree contains a lot of unneeded information

The Front End

Compilers often use an abstract syntax tree instead of a parse tree



The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

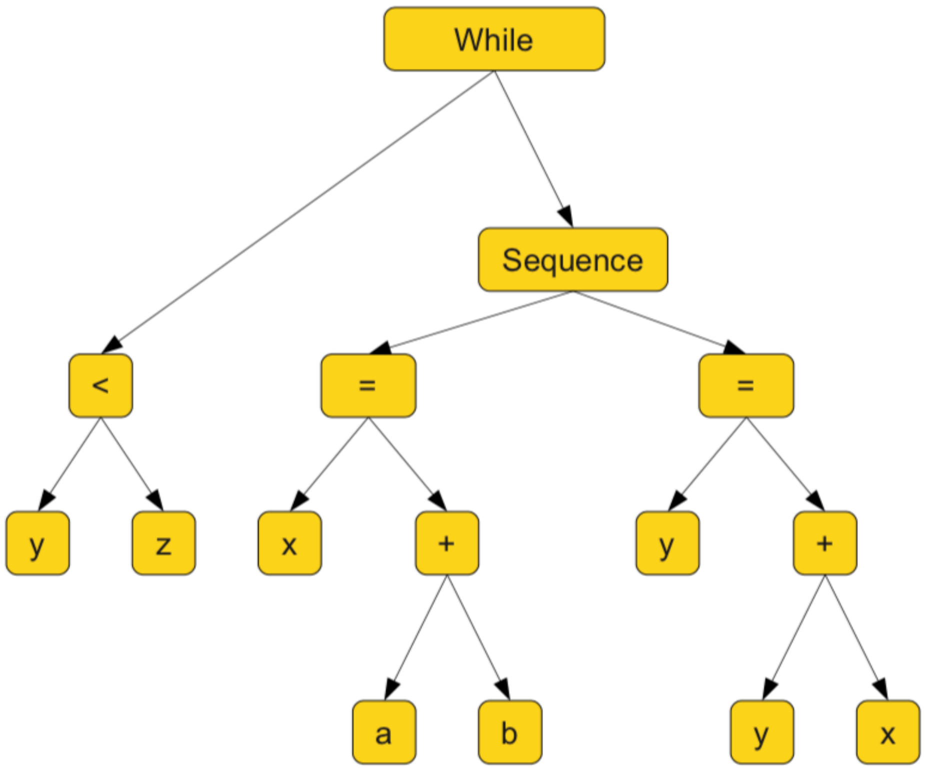
ASTs can be used as intermediate representation

Syntax analysis

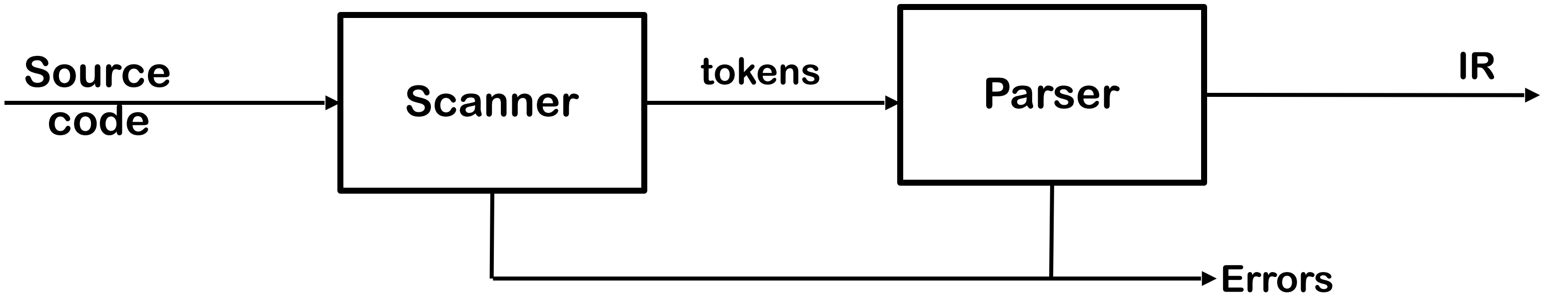
Split program to individual words that makes sense:

My mother cookes dinner

```
while (y < z) {
   int x = a + b;
   y += x; }
```



The Front End



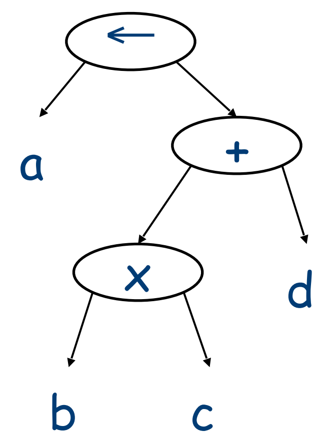
Next step:

Code shape determines many properties of resulting program

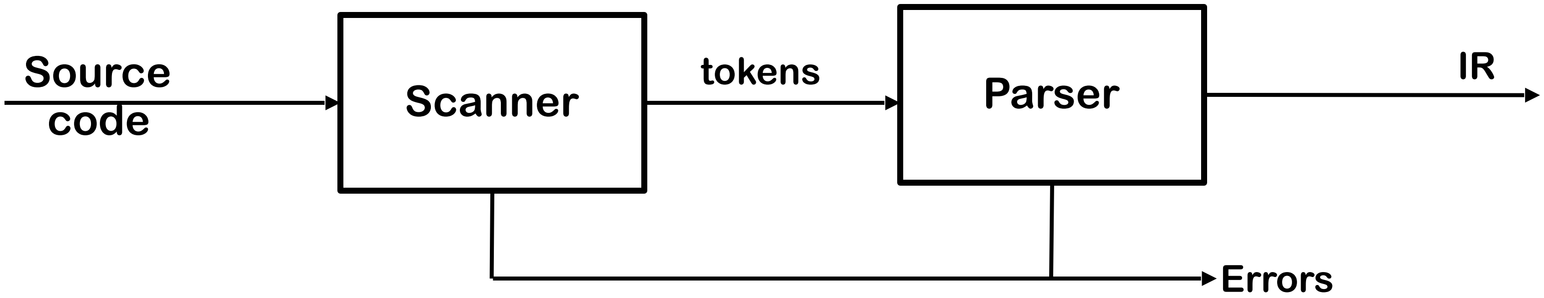
Recall the speed difference between different ways of writing a simple array initialization

$a \leftarrow b \times c + d$

becomes



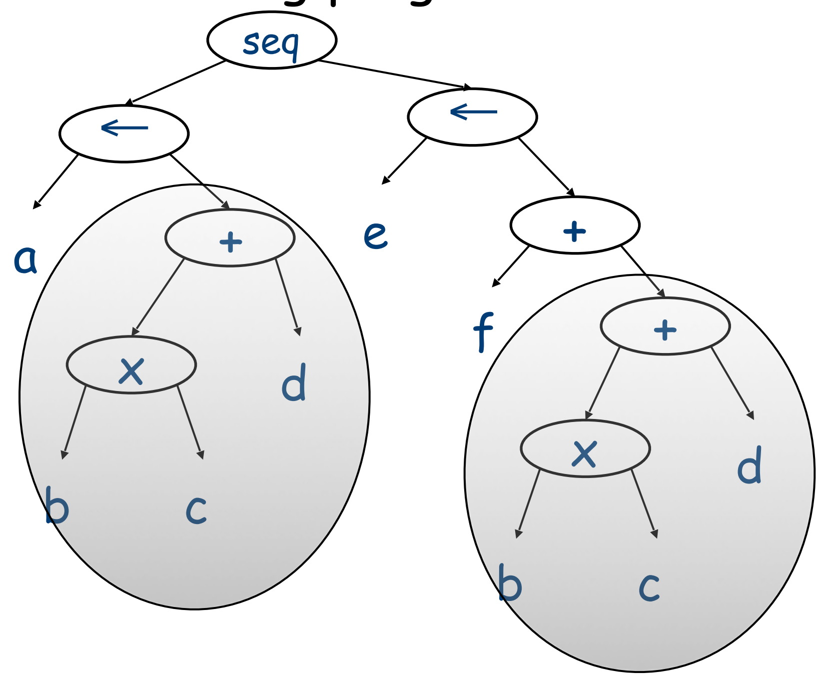
The Front End



Code shape determines many properties of resulting program

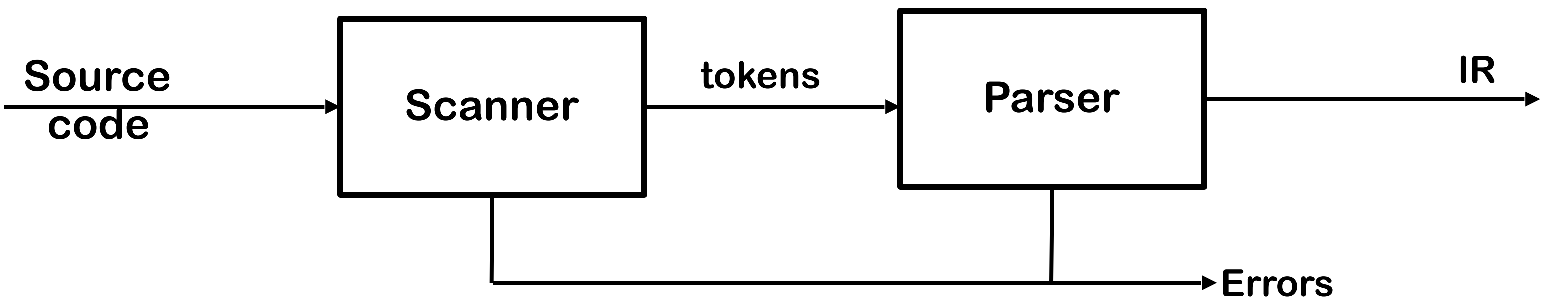
$a \leftarrow b \times c + d$

$e \leftarrow f + b \times c + d$



If you turn this AST into code, you will likely get duplication but getting it right takes a fair amount of effort ….

The Front End



Code shape determines many properties of resulting program

$a \leftarrow b \times c + d$

$e \leftarrow f + b \times c + d$

becomes

```
load  @b     ⇒ r1
load  @c     ⇒ r2
mult  r1,r2  ⇒ r3
load  @d     ⇒ r4
add   r3,r4  ⇒ r5
store r5     ⇒ @a
load  @f     ⇒ r6
add   r5,r6  ⇒ r7
store r7     ⇒ @e
```

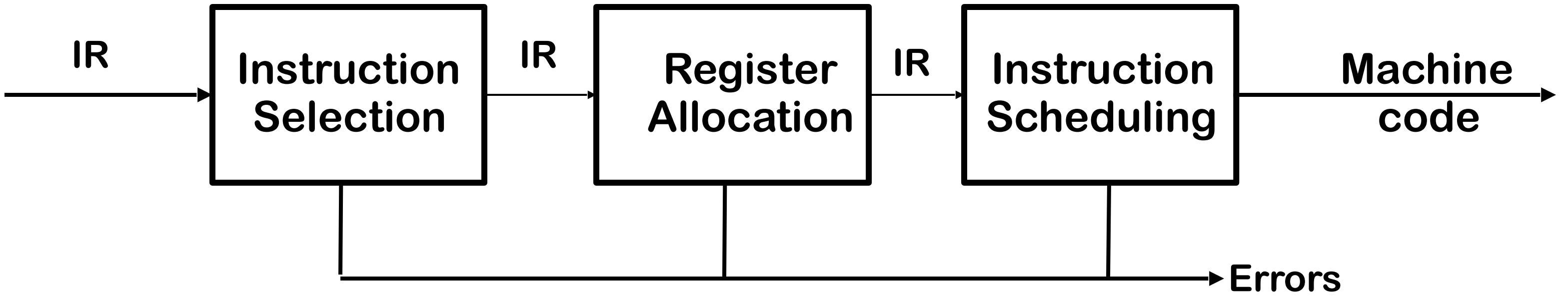
(the first five instructions) computes $b \times c + d$

(load @f and add $r_5,r_6 \Rightarrow r_7$) reuses $b \times c + d$

We would like to produce a code

for the common expression and then reuses it

The Back End



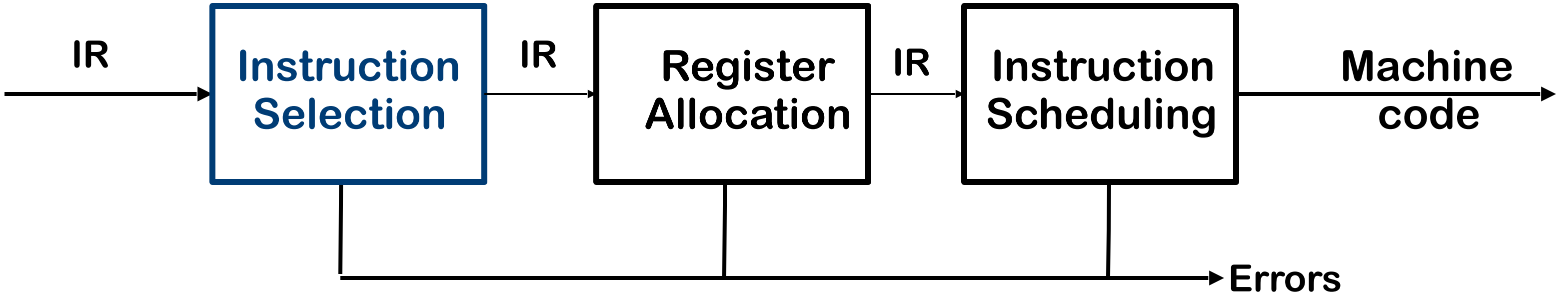
Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers

Automation has been less successful in the back end

Standard goal has become "locally optimal" code.

The Back End



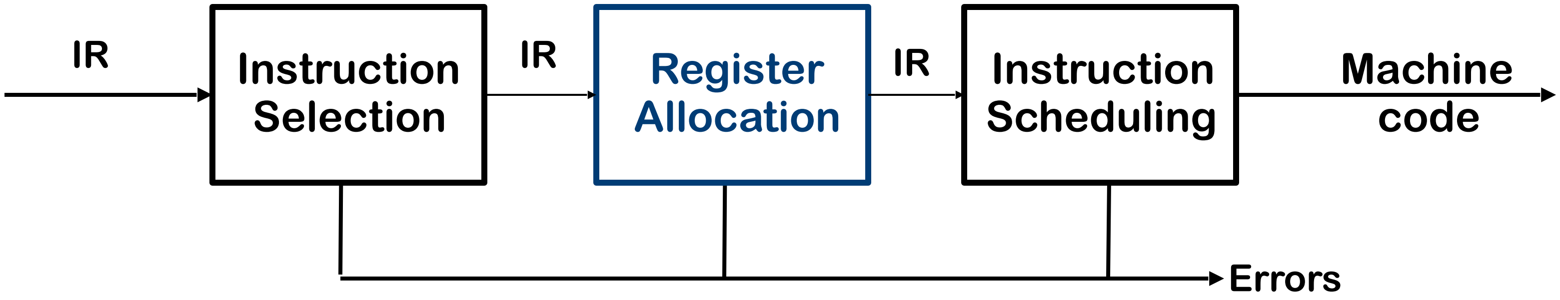
Instruction Selection

- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  - ad hoc methods, pattern matching, dynamic programming
  - Form of the IR influences choice of technique

This was the problem of the future in 1978

- Spurred by transition from PDP-11 to VAX-11
- RISC architecture simplified this problem

The Back End



Register Allocation

- Have each value in a register when it is used
- Manage a limited set of resources
- Can change instruction choices & insert LOADs & STOREs
- Optimal allocation is NP-Complete in most settings

Compilers approximate solutions to NP-Complete problems

About ILOC

- ILOC (Intermediate Language for Optimizing Compiler) is a notation used in the textbook to indicate an assembly language for a simple RISC machine.

- Most operations takes arguments that are registers

  add r1,r2 -> r3 (r1 + r2->r3)

- The memory operations loads and stores transfer values between memory and registers

  loadI c1->r2 (the constant c1 goes in register r2)

  loadAI r1,c2 -> r3 (Memory(r1+c2) ->r3)

  storeAI r1 -> r2,c3 (r1-> Memory(r2+c3))

Register allocation for a = (a x 2 x b x c) x d

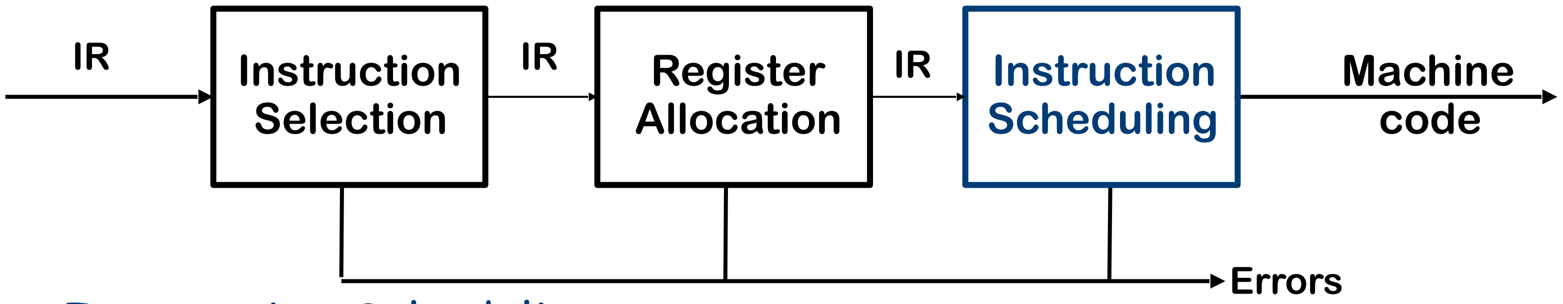
Use 6 registers!

```
loadAI   rarp, @a ⇒ ra        // load ‘a’
loadI    2        ⇒ r2        // constant 2 into r2
loadAI   rarp, @b ⇒ rb        // load ‘b’
loadAI   rarp, @c ⇒ rc        // load ‘c’
loadAI   rarp, @d ⇒ rd        // load ‘d’
mult     ra, r2   ⇒ ra        // ra ← a × 2
mult     ra, rb   ⇒ ra        // ra ← (a × 2) × b
mult     ra, rc   ⇒ ra        // ra ← (a × 2 × b) × c
mult     ra, rd   ⇒ ra        // ra ← (a × 2 × b × c) × d
storeAI  ra       ⇒ rarp,@a   // write ra back to ‘a’
```

```
loadAI   rarp, @a ⇒ r1        // load ‘a’
add      r1, r1   ⇒ r1        // r1 ← a × 2
loadAI   rarp, @b ⇒ r2        // load ‘b’
mult     r1, r2   ⇒ r1        // r1 ← (a × 2) × b
loadAI   rarp, @c ⇒ r2        // load ‘c’
mult     r1, r2   ⇒ r1        // r1 ← (a × 2 × b) × c
loadAI   rarp, @d ⇒ r2        // load ‘d’
mult     r1, r2   ⇒ r1        // r1 ← (a × 2 × b × c) × d
storeAI  r1       ⇒ rarp, @a  // write ra back to ‘a’
```

Use 3 registers!

The Back End



Instruction Scheduling

- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed

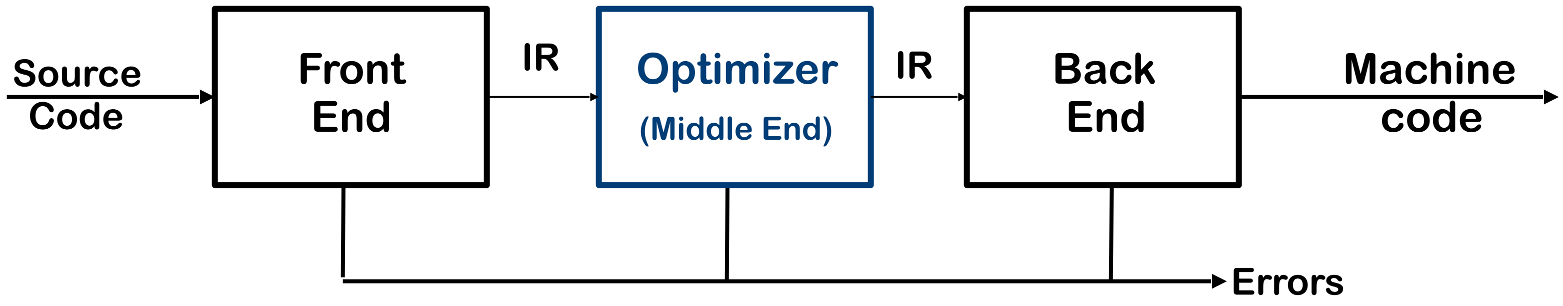
Instruction scheduling

LoadAI, storeAI	3 cycles
mult	2 cycles
others	1 cycle

- Reorder operations to reflect the target machine performance constraints

Start	End				
1	3	loadAI	r_{arp}, @a	$\Rightarrow r_1$	// load 'a'
4	4	add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow a \times 2$
5	7	loadAI	r_{arp}, @b	$\Rightarrow r_2$	// load 'b'
8	9	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (a \times 2) \times b$
10	12	loadAI	r_{arp}, @c	$\Rightarrow r_2$	// load 'c'
13	14	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (a \times 2 \times b) \times c$
15	17	loadAI	r_{arp}, @d	$\Rightarrow r_2$	// load 'd'
18	19	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
20	22	storeAI	r_1	$\Rightarrow r_{arp}$, @a	// write r_a back to 'a'

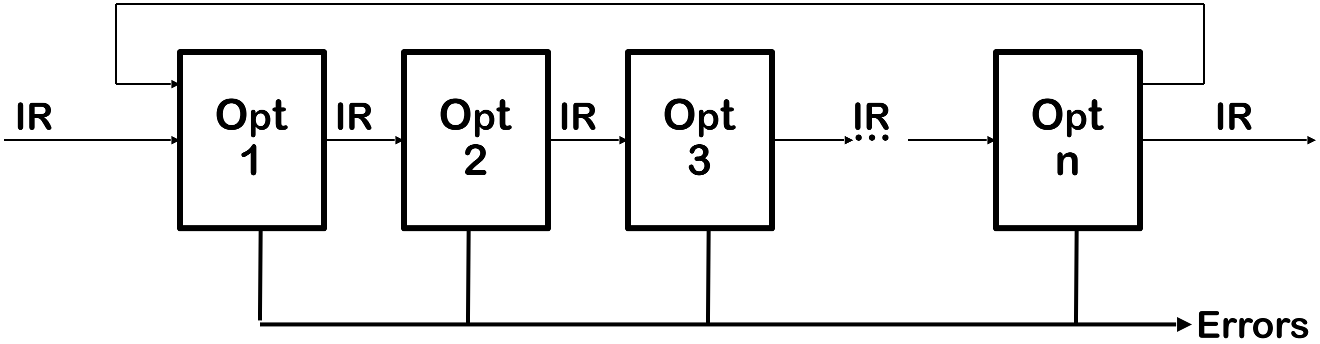
Traditional Three-part Compiler



Code Improvement (or Optimization)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code

The Optimizer (or Middle End)



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form