Overview of the Course

Your teachers:

Schedule :

Two weekly lectures

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

Roberta Gori
roberta.gori@di.unipi.it

One weekly lecture for experiencing with practical applications

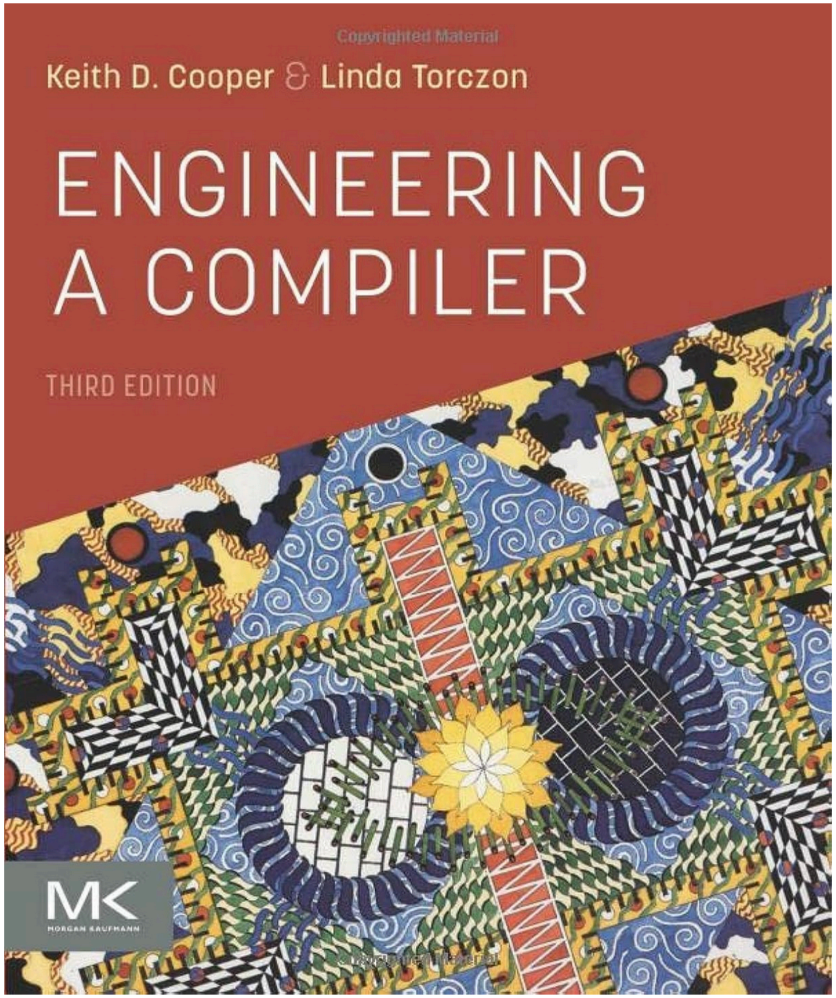
Thursday	11:00	13:00	lab I

Lorenzo Ceragioli
lorenzo.ceragioli@imtlucca.it

What we will see

- Formal languages (maybe a recall for someone):
  - 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: Abstract interpretation
- Register allocation

Our textbook

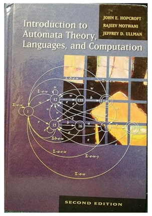


Other informations

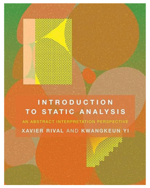
- web page, I will add there all the slides

www.di.unipi.it/~gori/Linguaggi-Compilatori2023

Material for specific topics:



- Introduction to Automata Theory, Languages, And Computation. Hopcroft, Motwani, Ullman



- Introduction to static Analysis: an abstract interpretation perspective Rival, Yi

- Principles of abstract interpretation Cousot

- 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

Program analysis for verification and optimization

- Static analysis to prove properties on program behaviors
- Abstract interpretation based techniques for proving correctness or devise bugs in programs:
  connections with Meta and real world tools

CAVEAT on the slides

- Apart from the first two weeks, we will follow the slides of the authors of the book
- These are slides for a course in U.S.A. which intend to give you basic concepts and ideas (they do not summarise the book but they give you more ideas and perspectives)
- All the details have to be found in the book
- Use the slides as a guide on the different topics

Final Exam

The exam will consist in

1. a project developed partly during the hours of thursdays
2. a seminar on a chosen topic between a list
3. an oral discussion mainly on the project

Please, actively partecipate to lectures

Compilers

- What is a compiler?
  - A program that takes other programs and prepare them for execution

- In particular, a program that takes a program and translate it in program written in a target language
  - The target language is in general the instruction set of an architecture
  - The target language can be a human-oriented programming language (Source-to-source translators)

AOT vs JIT

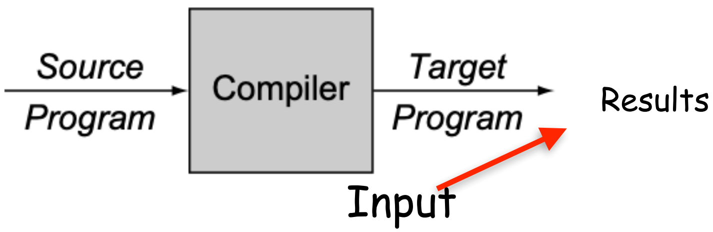
- Most compiler are designed to run in a separate step before execution (ahead-of-time AOT)

- Some new compilers translate the code to executable form at runtime (just-in-time JIT)

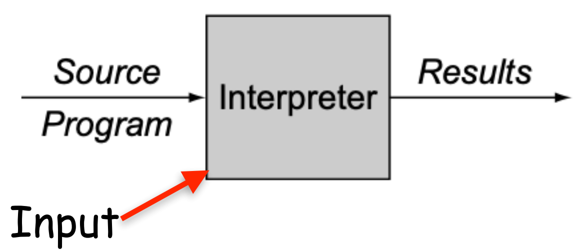
  the cost of the translation has to be summed to the one of the execution

Compilers vs Interpreters

- A compiler is a program that takes a program and translate it in a program written in an another language



- What is an interpreter?
  - A program that reads a program and an input and produces the results of executing that program on the input



It’s a property of the implementation !

- C and C++ are typically compiled,

  Pyton and Scheme are typically interpreted
- Java has a complex translation schema:

  - compiled to bytecode for the Java VM (by AOT compiler)

  - bytecode is interpreted or hybrid strategy is used (JIT compiler)

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 Compilers?

Deep Understanding of Programming Languages

Studying compilers provides a deeper insight into how programming languages work, including structures, optimizations, and resource management.

Performance Optimization

You will learn how to optimize code for speed and memory usage,

crucial for high-performance applications or resource-constrained environments.

Foundation for Language Development

Knowledge of compilers is essential for creating new programming languages or extending existing ones.

Development Tools and Automation

Compilers are the backbone of many tools like IDEs, debuggers, and static analyzers, enabling productivity improvements for developers.

Problem-Solving and Transferable Skills

Compilers involve complex algorithmic and data structure, applicable in various fields like OS development or search engines.

High Demand in the Job Market

Skills in compilers are sought after in industries working with custom languages, high-level compilers, or software optimization.

Research Opportunities

Still many open problems!

Compiler are interesting

- 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

Performance: reducing the price of language abstraction

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

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

Well written compilers make such 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, nei suoi primi mesi di vita, avesse tradotto qualsiasi ragionevole programma sorgente "scientifico" in un programma oggetto che una volta eseguito fosse piu' veloce solo la mets' della codifica a mano dello stesso programma, l'accettazione del nostro sistema di compilazione sarebbe stata in serio pericolo... Credo che se non fossimo riusciti a produrre programmi compilati efficienti, l'uso diffuso di linguaggi come il 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

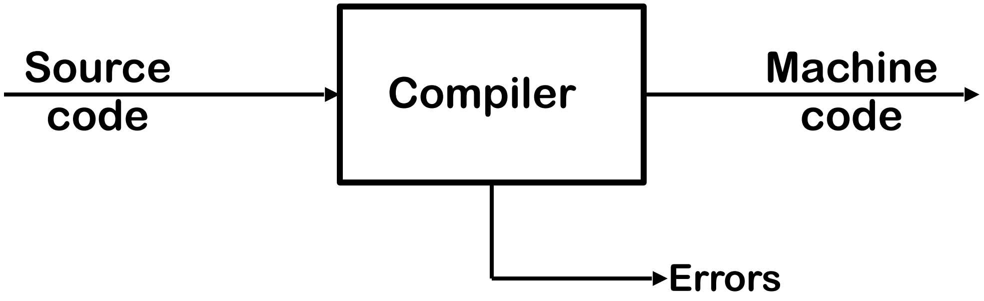
Fundamental Principles of Compilation

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

- The compiler must improve the input program

The View from 35,000 Feet

High-level View of a Compiler

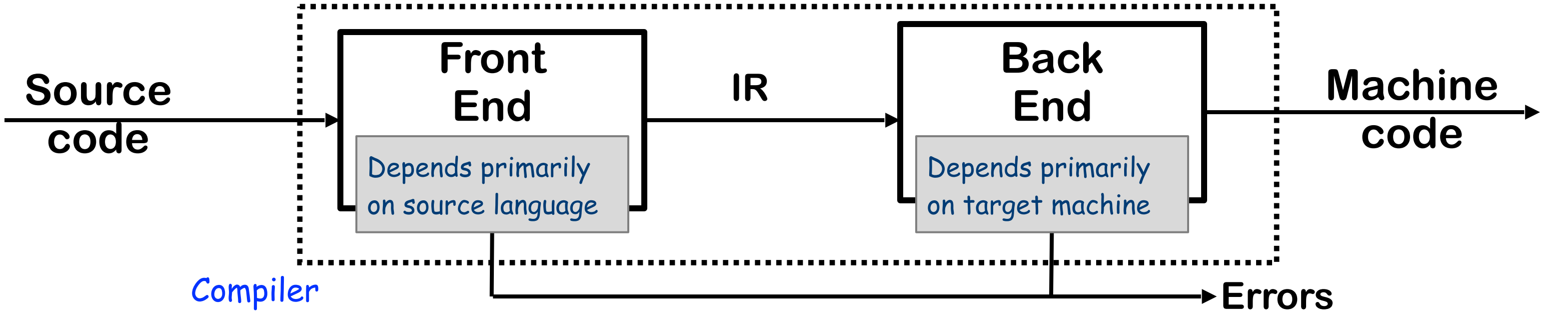


Implications

- 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

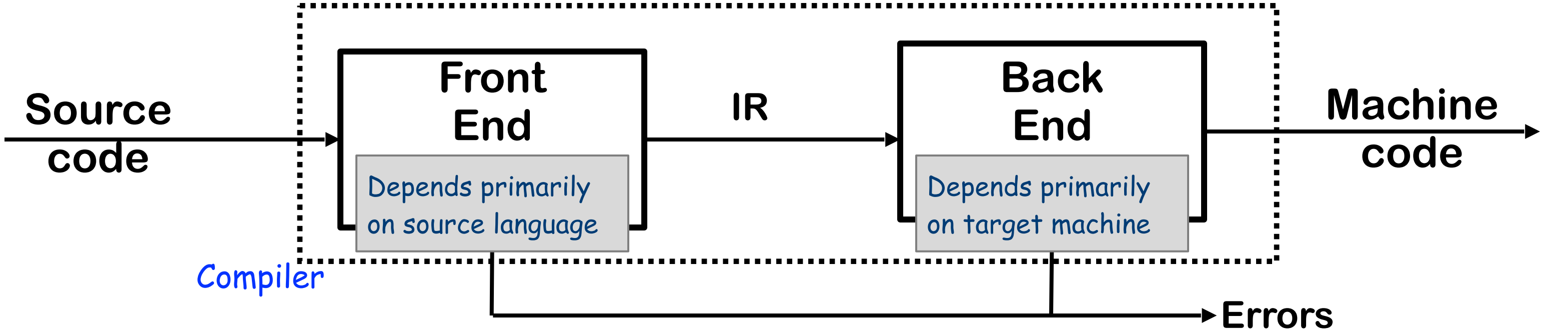


Implications of the division:

- 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)

Front end is O(n) or O(n log n) Back end is NP-Complete

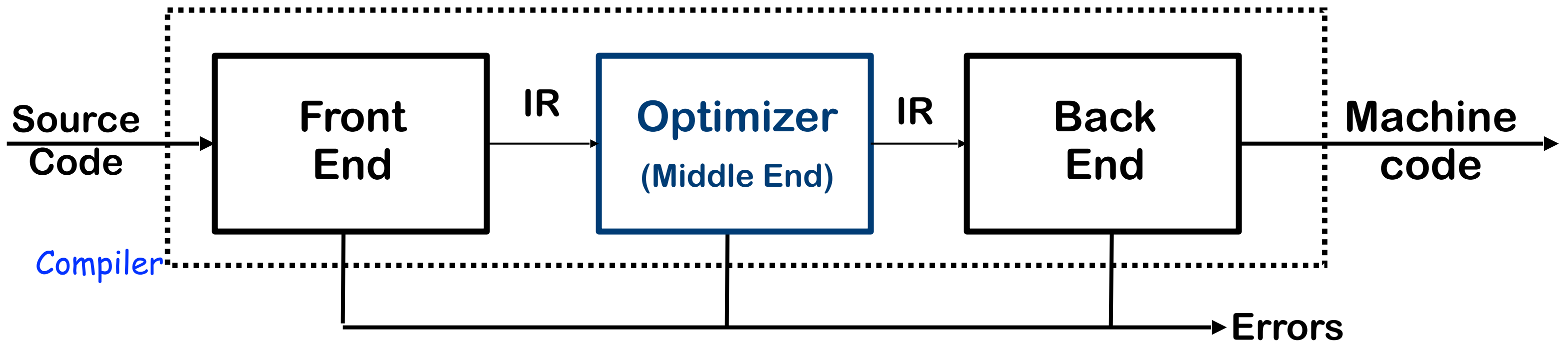
Advantages of two-pass compiler



Classic principle from software engineering: Separation of concerns

- The some architecture can target a different machine code
- The some architecture can target a source code
- The IR has to encode all the knowledge that the compiler has on the program

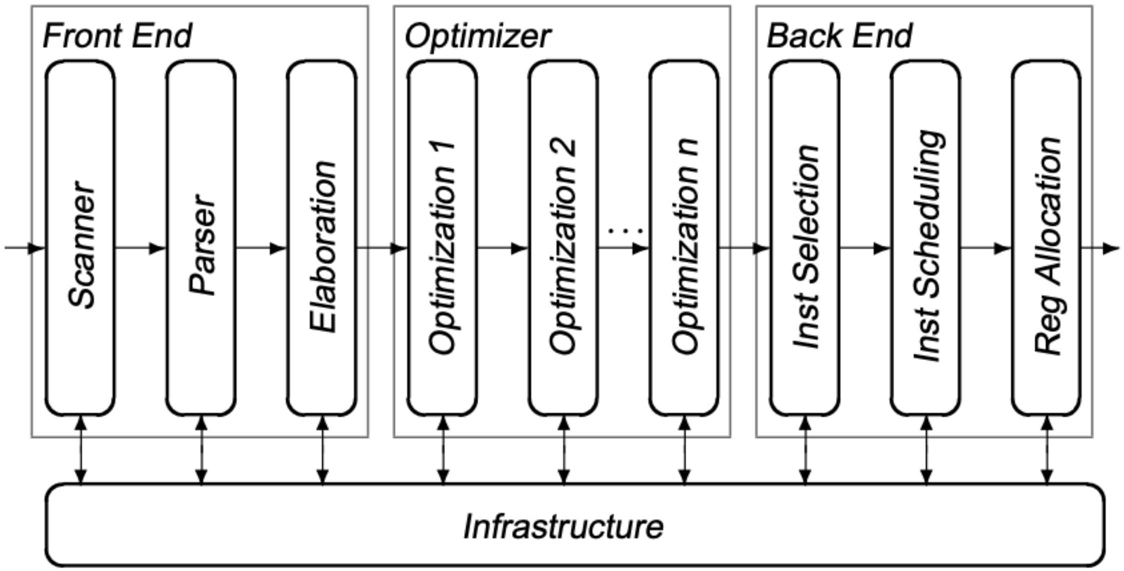
Traditional three-part Compiler



Code Improvement (or Optimization)

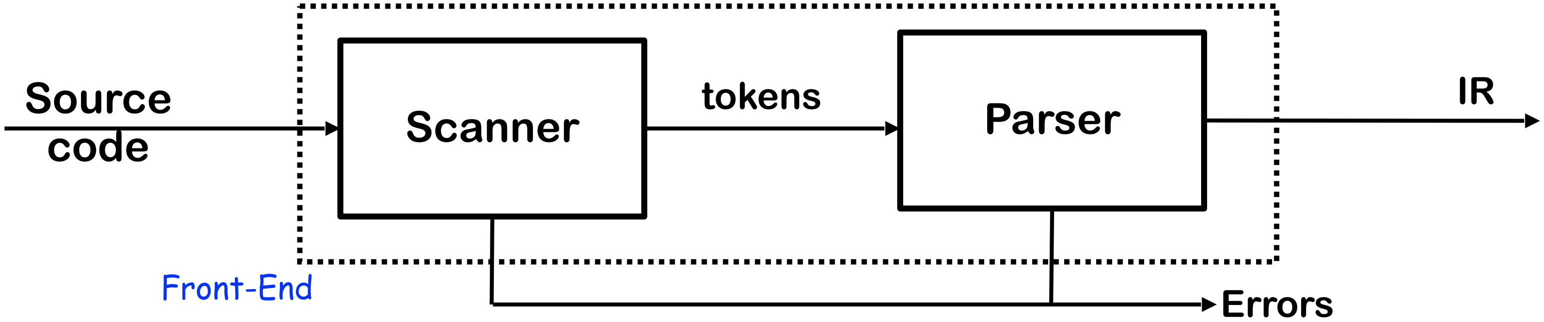
- analyzes IR and transform IR
- primary goal is to reduce running time of the compiled code
  - and/or reduce code space, power consumption, page faults.....
- Must preserve "meaning" of the code

The phases of a Compiler



- In the actual architecture each phase is divided into a series of passes
- The optimiser contains passes that use distinct analyses and transformation to improve the code

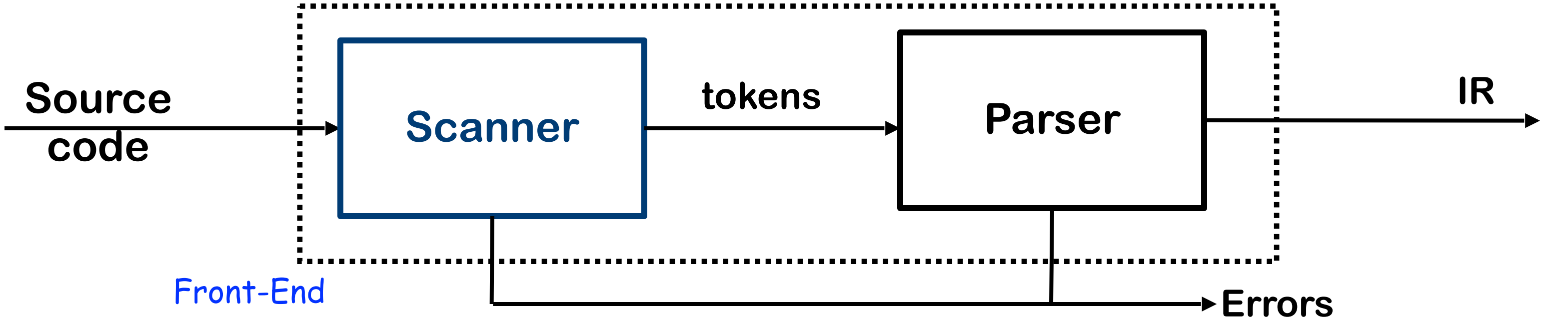
The Front End



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 stream characters into stream of words (Lexical analysis)
- It determines
- Produces pairs — a word & its part of speech

  x = x + y ;   becomes <id,x> = <id,x> + <id,y> ;
- Typical words include numbers, identifier, +, –, new, while, if
- Speed is important

Textbooks advocate automatic scanner generation

Commercial practice appears to be hand-coded scanners

Lexical analysis

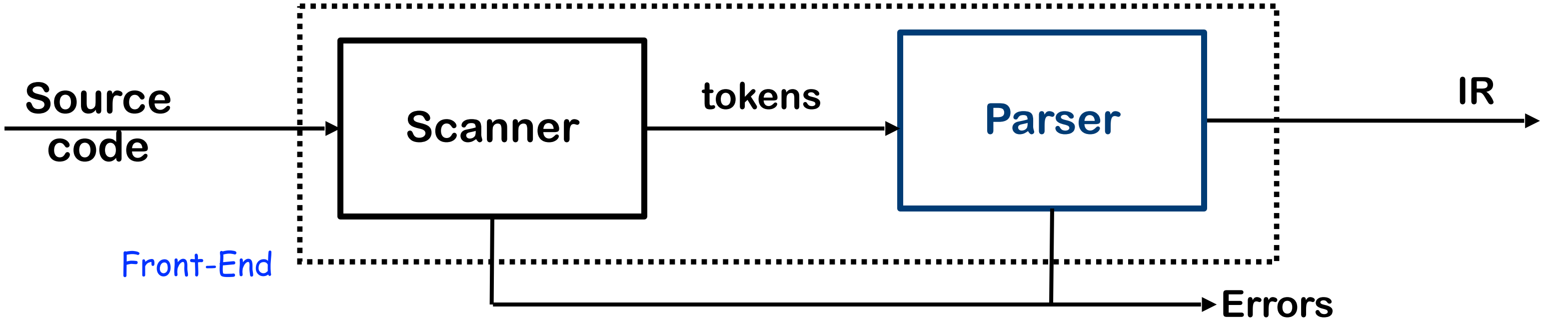
Split program into individual words that makes sense:

1g2h3i is neither a valid identifier nor a valid number

```
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

- Check the syntax & reports errors (Syntax Analysis)
- It determines if the stream of words is a sentence in the source language
- Builds IR for source program

Hand-coded parsers are fairly easy to build

Most books advocate using automatic parser generators

Grammars for the Front-End

To recognise words and sentences of the source language, the Front-End uses grammars like

SheepNoise → SheepNoise baa
| baa

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

It is written in a variant of Backus–Naur Form (BNF)

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

Grammars for simple expressions

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.		|	-

- It defines simple expressions with + & - over number and id
- This grammar 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

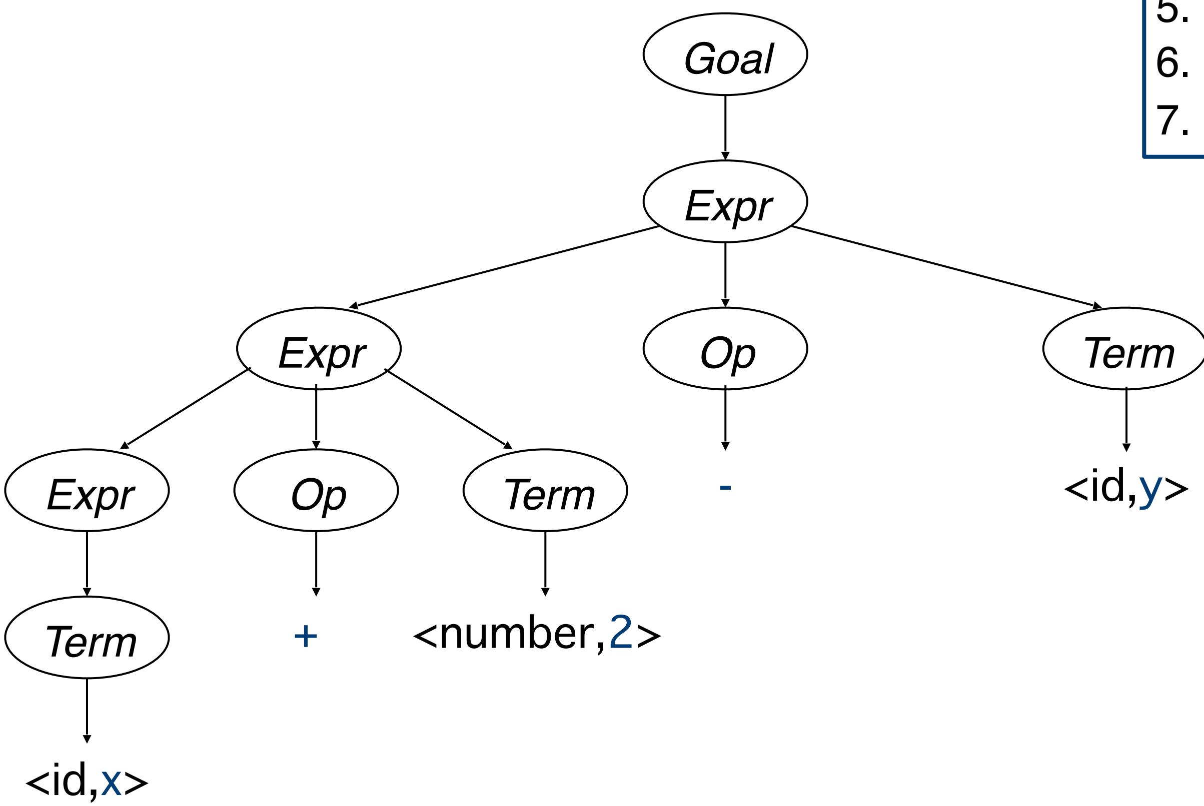
To recognize a valid sentence, we reverse this process and start from x+2-y

The Front End

To recognise if x + 2 - y

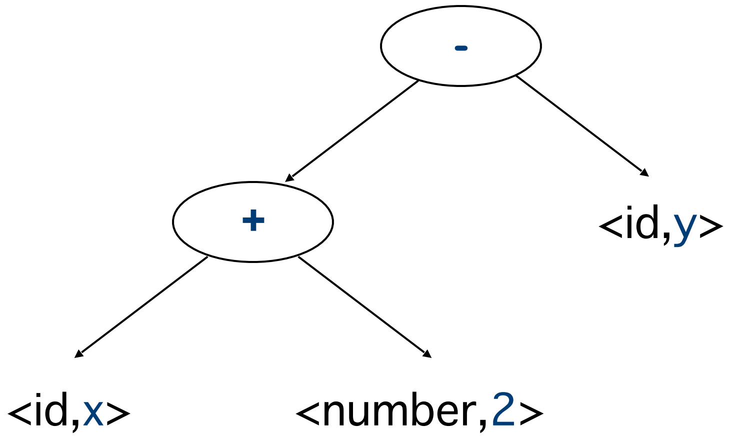
belongs to the language we construct the
parsing tree (or syntax tree)

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



The Front End

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



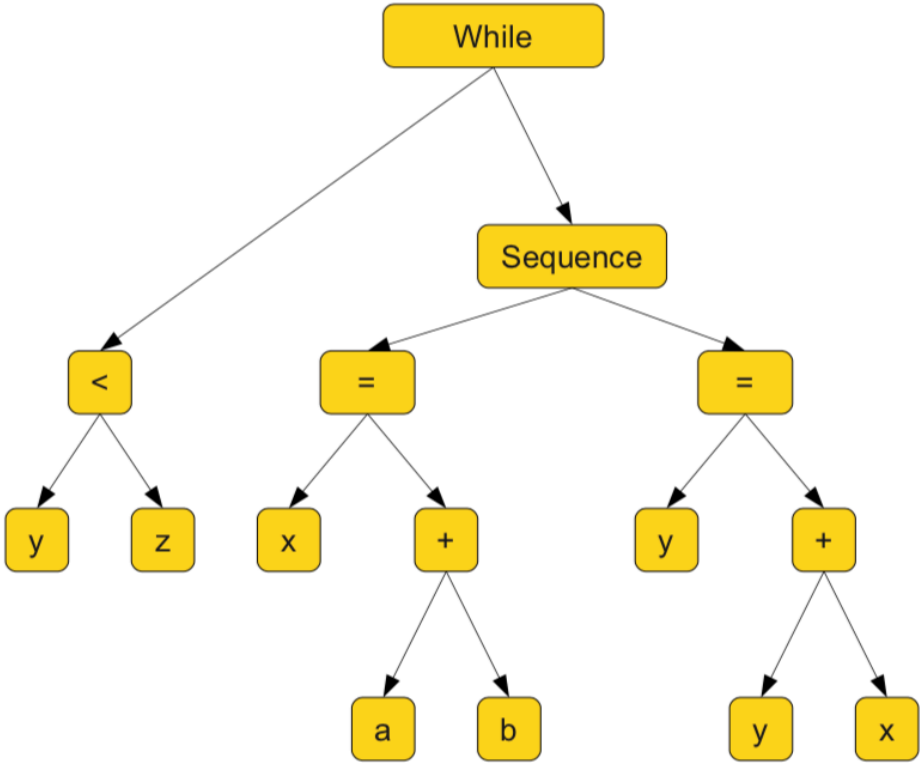
The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise

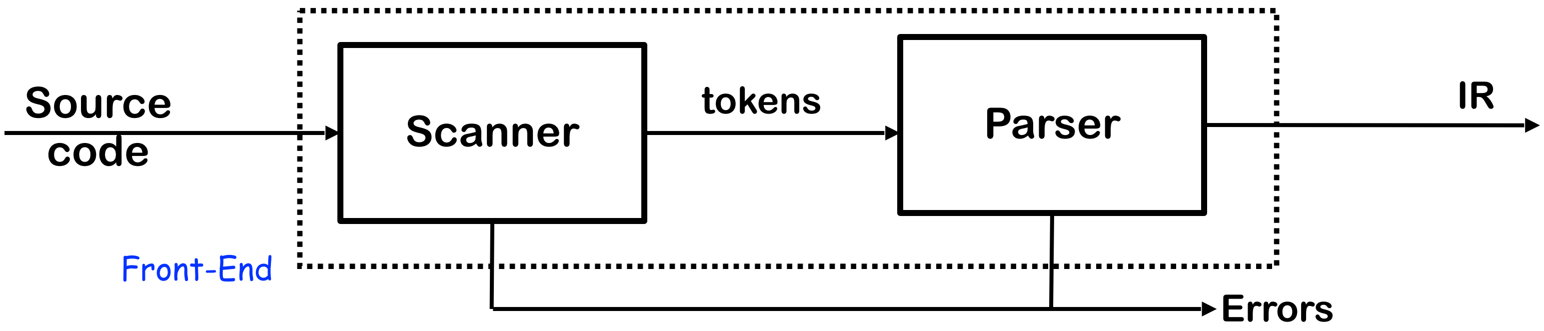
ASTs can be used as intermediate representation

Syntax analysis

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

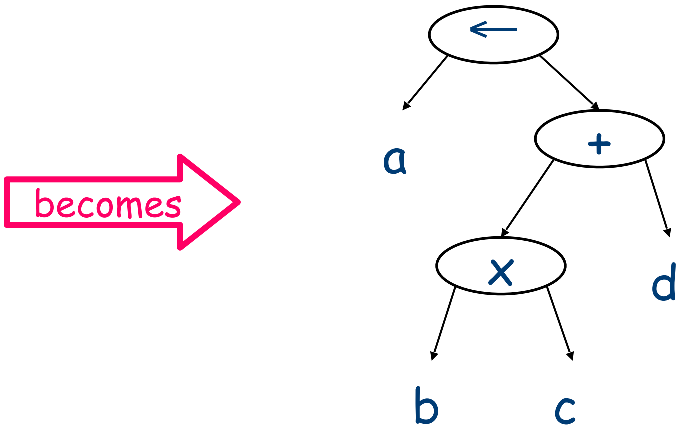


Front-End produces the IR

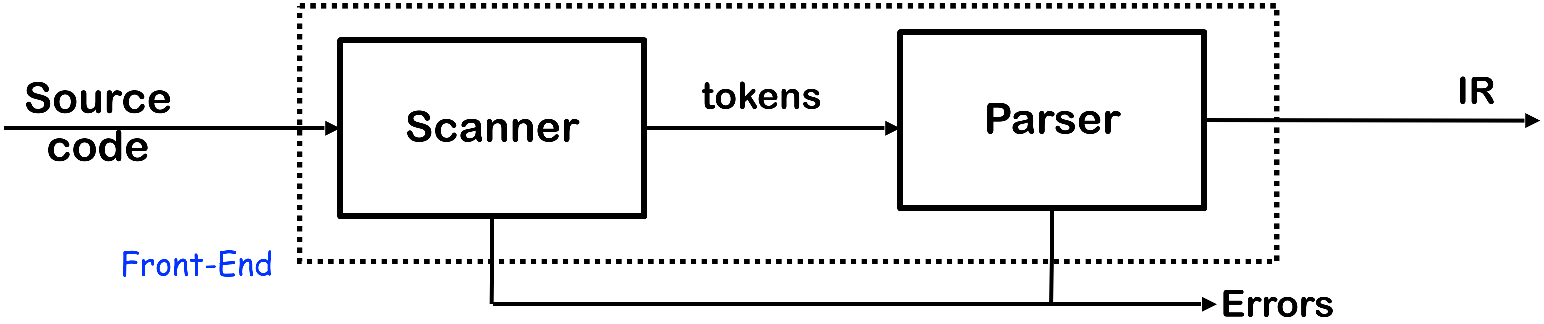


If the IR is the Abstract Syntax Tree

$a \leftarrow b \times c + d$



Front-End produces the IR



If the IR is the three address code

$a \leftarrow b \times c + d$

becomes

load @b $\Rightarrow r_1$

load @c $\Rightarrow r_2$

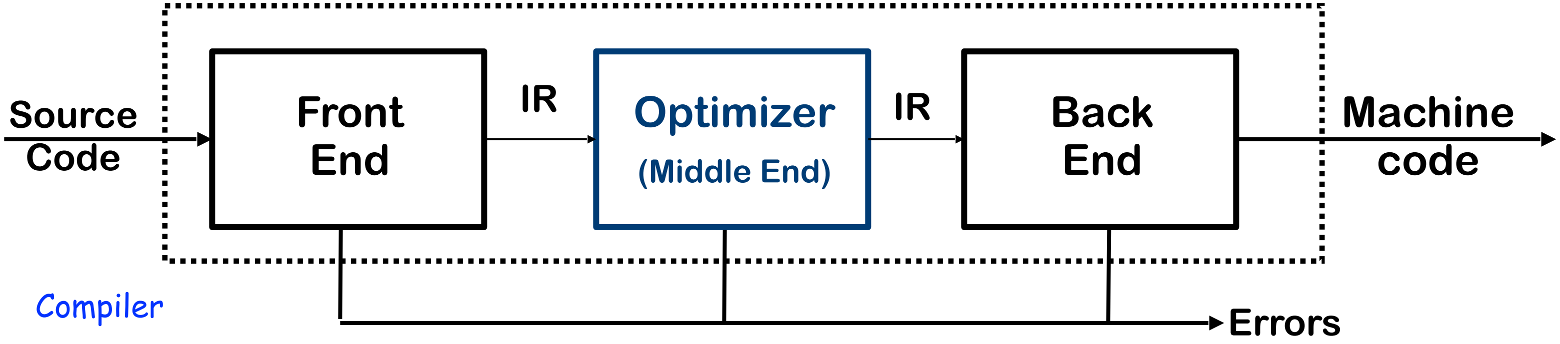
mult $r_1, r_2 \Rightarrow r_3$

load @d $\Rightarrow r_4$

add $r_3, r_4 \Rightarrow r_5$

store $r_5 \Rightarrow$ @a

The Optimizer



The IR emitted by the Front-End is generated by looking to each statement at the time

The IR program contains code that will work for any surrounding context

The optimizer can discover something on the context from the entire IR code and use this knowledge to improve the code

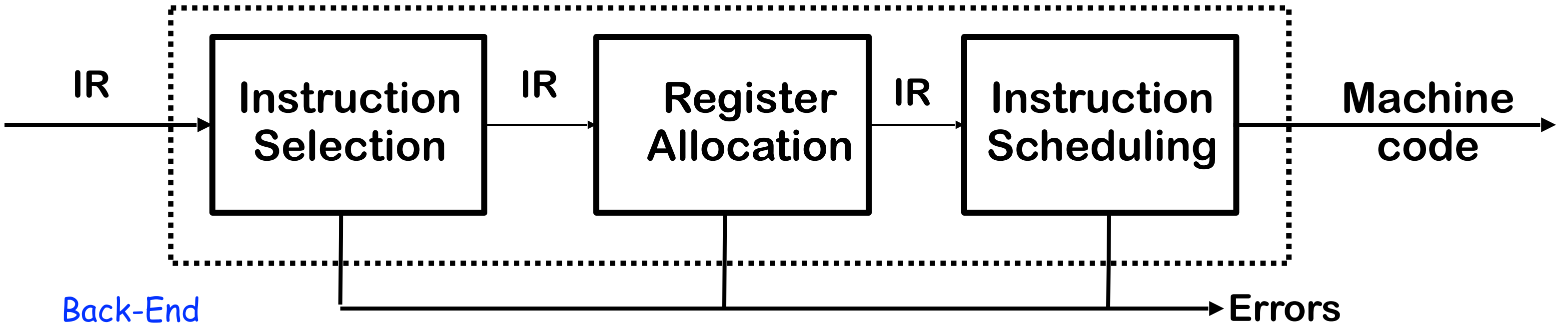
Example of optimizations: loop invariant

```
b ← ···
c ← ···
a ← 1
for i = 1 to n
  read d
  a ← a × 2 × b × c × d
  end
```

becomes

```
b ← ···
c ← ···
a ← 1
t ← 2 × b × c
for i = 1 to n
  read d
  a ← a × d × t
  end
```

The Back End



Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Reorder the instructions so that efficiency is gained

Automation has been less successful in the back end

About ILOC

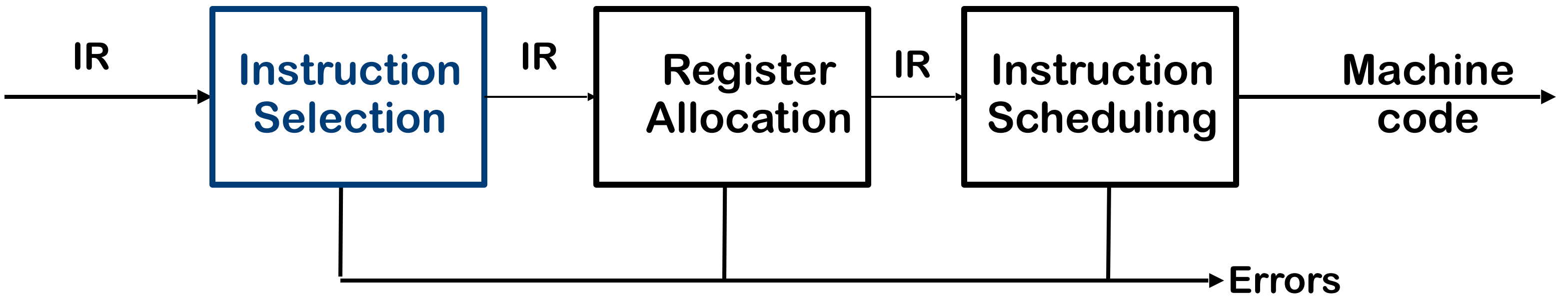
- ILOC (Intermediate Language for Optimizing Compiler) is an assembly language for a simple RISC machine.

ILOC Operation		Meaning
loadAI	$r_1, c_2 \Rightarrow r_3$	**Memory**$(r_1 + c_2) \rightarrow r_3$
loadI	$c_1 \Rightarrow r_2$	$c_1 \rightarrow r_2$
mult	$r_1, r_2 \Rightarrow r_3$	$r_1 \times r_2 \rightarrow r_3$
storeAI	$r_1 \Rightarrow r_2, c_3$	$r_1 \rightarrow$ **Memory**$(r_2 + c_3)$

Instruction selection for a = (a x 2 x b x c) x d

loadAI	r_{arp}, @a	⇒	r_a	// load ‘a’ (Memory(r1+c2) ->r3)
loadI	2	⇒	r_2	// constant 2 into r_2 (the constant c1 goes in register r2)
loadAI	r_{arp}, @b	⇒	r_b	// load ‘b’
loadAI	r_{arp}, @c	⇒	r_c	// load ‘c’
loadAI	r_{arp}, @d	⇒	r_d	// load ‘d’
mult	r_a, r_2	⇒	r_a	// $r_a \leftarrow a \times 2$
mult	r_a, r_b	⇒	r_a	// $r_a \leftarrow (a \times 2) \times b$
mult	r_a, r_c	⇒	r_a	// $r_a \leftarrow (a \times 2 \times b) \times c$
mult	r_a, r_d	⇒	r_a	// $r_a \leftarrow (a \times 2 \times b \times c) \times d$
storeAI	r_a	⇒	r_{arp},@a	// write r_a back to ‘a’ (r1-> Memory(r2+c3))

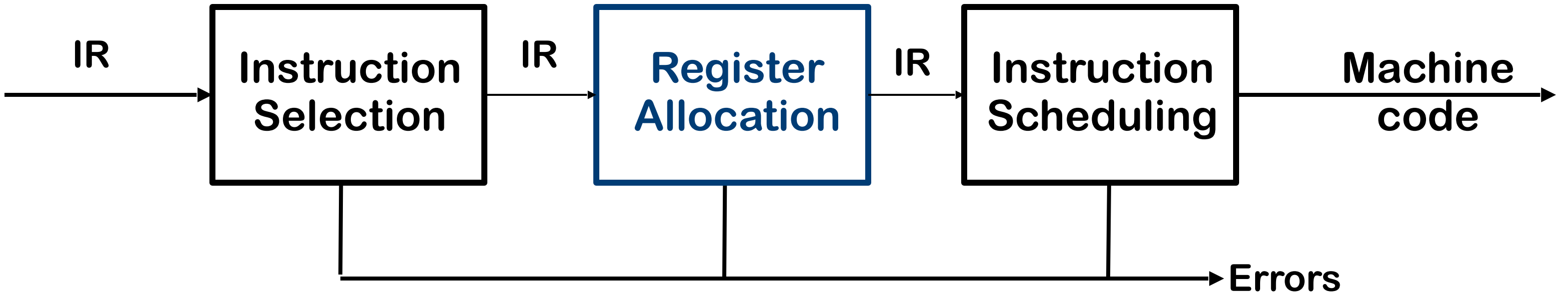
The Back End



Instruction Selection

- It has to translate the IR code into sequence of ISA instructions
- Take advantage of features of the target machine
- Assume an infinite number of (virtual) registers
- Usually viewed as a pattern matching problem
  - ad hoc methods, pattern matching
  - Form of the IR influences choice of technique
  - RISC architecture simplified this problem

The Back End



Register Allocation

- It has to map virtual to physics registers
- 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

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

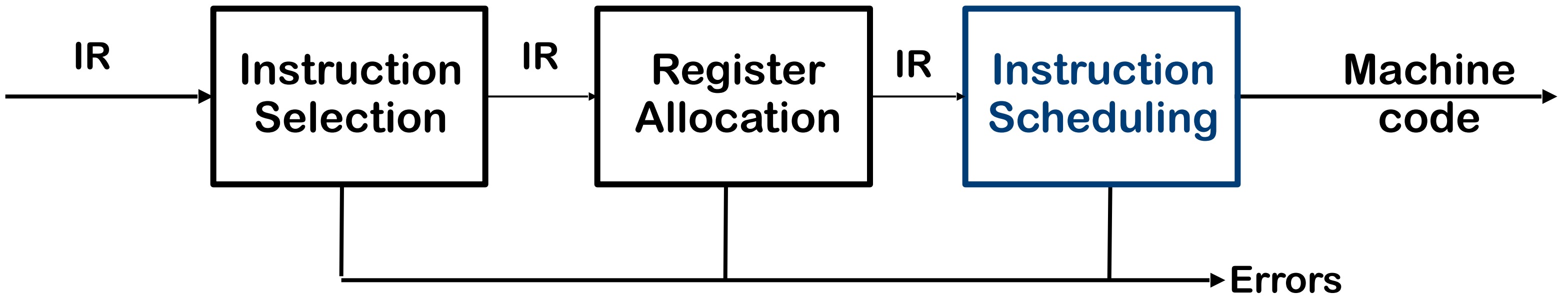
Use 6 registers!

```
loadAI   rarp, @a ⇒ ra       // load 'a'            (Memory(r1+c2) ->r3)
loadI    2        ⇒ r2       // constant 2 into r2  (the constant c1 goes in register 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'   (r1-> Memory(r2+c3))
```

Use 3 registers!

```
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'
```

The Back End



Instruction Scheduling

- It reorder the sequence of instructions to avoid 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

Before the instruction scheduling

LoadAI, storeAI	3 cycles
mult	2 cycles
others	1 cycle

- The original number of cycles

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

After the instruction scheduling

Start	End				
1	3	loadAI	r_{arp}, @a	$\Rightarrow r_1$	// load 'a'
2	4	loadAI	r_{arp}, @b	$\Rightarrow r_2$	// load 'b'
3	5	loadAI	r_{arp}, @c	$\Rightarrow r_3$	// load 'c'
4	4	add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow a \times 2$
5	6	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (a \times 2) \times b$
6	8	loadAI	r_{arp}, @d	$\Rightarrow r_2$	// load 'd'
7	8	mult	r_1, r_3	$\Rightarrow r_1$	// $r_1 \leftarrow (a \times 2 \times b) \times c$
9	10	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
11	13	storeAI	r_1	$\Rightarrow r_{arp}$, @a	// write r_a back to 'a'