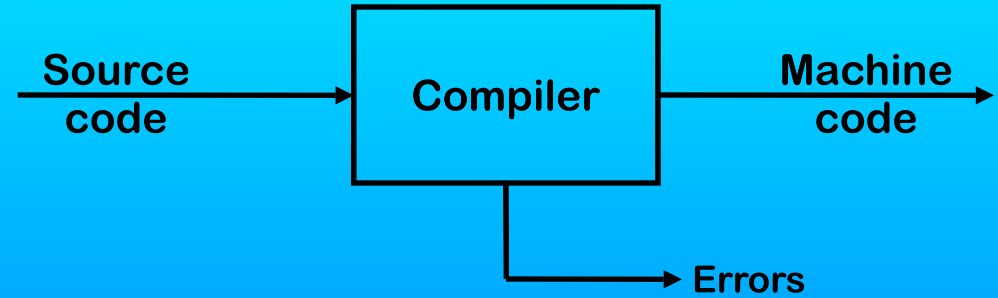
Compilers: An Overview

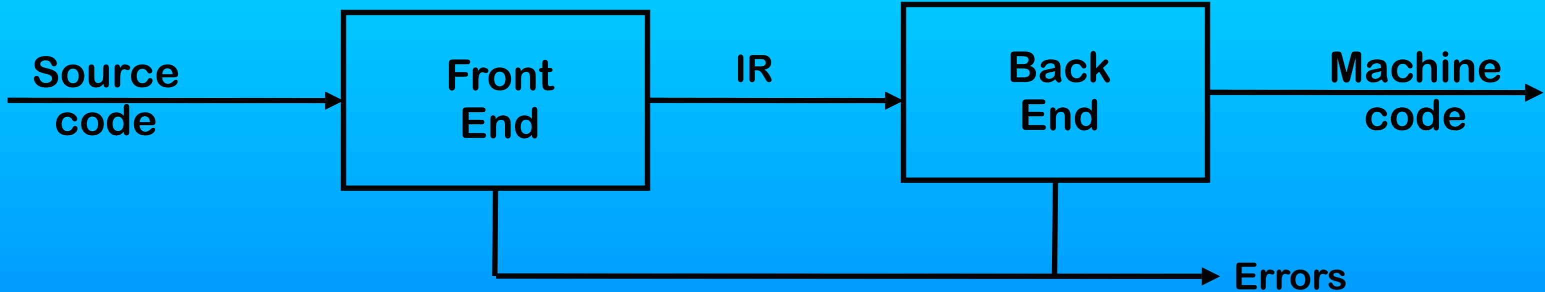
Dr. Stephen Blythe

High-level View of a Compiler



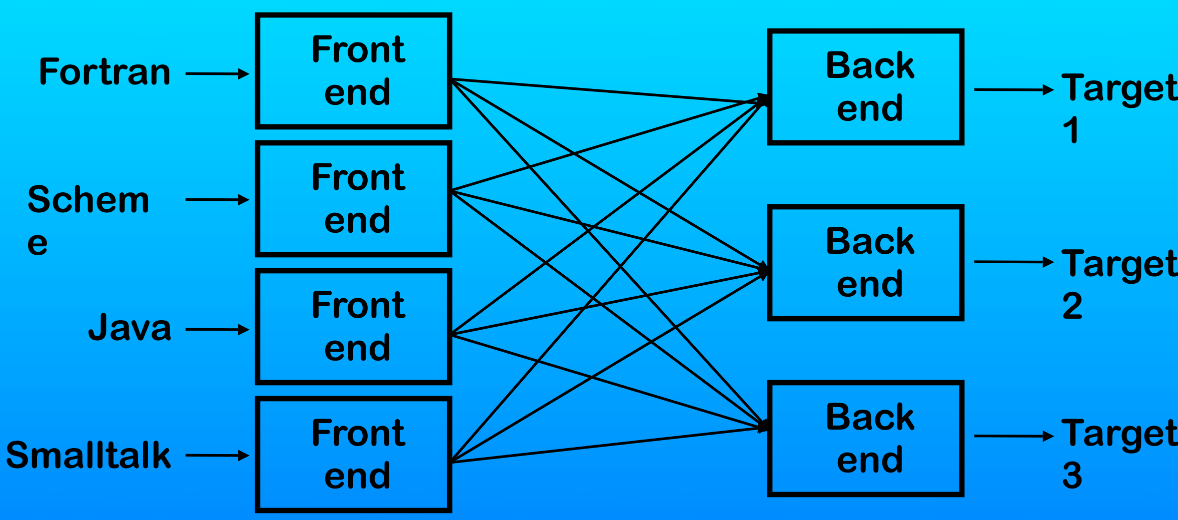
- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for machine code
- Big step up from assembly language translation

Traditional Two-pass Compiler



- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Allow multiple front ends & multiple passes
- Typically, front end is O(n log n), while back end is NP-Complete

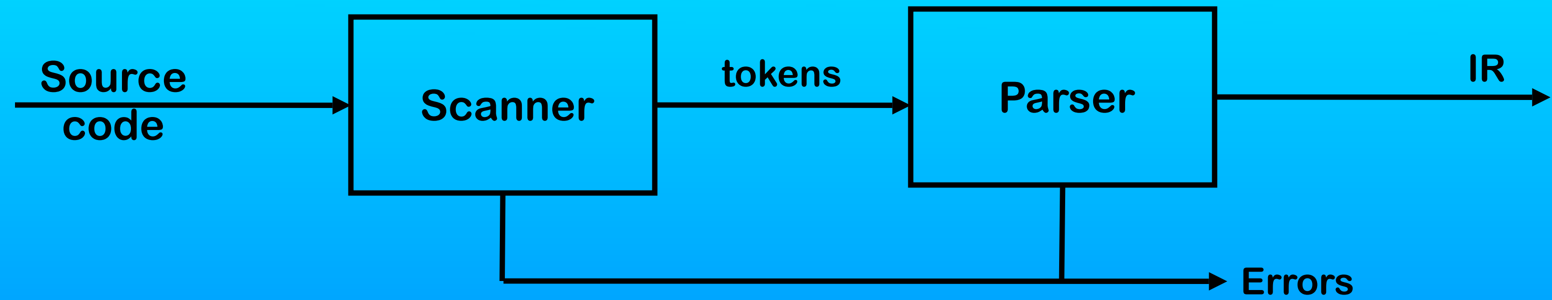
A Common Fallacy



Can we build n x m compilers with n+m components?

- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end
- Limited success in systems with very low-level IRs

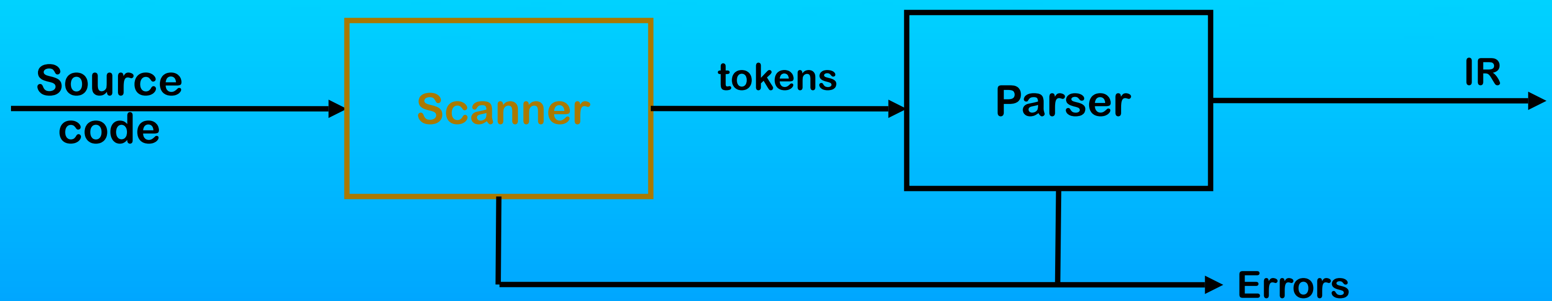
The Front End



Responsibilities

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

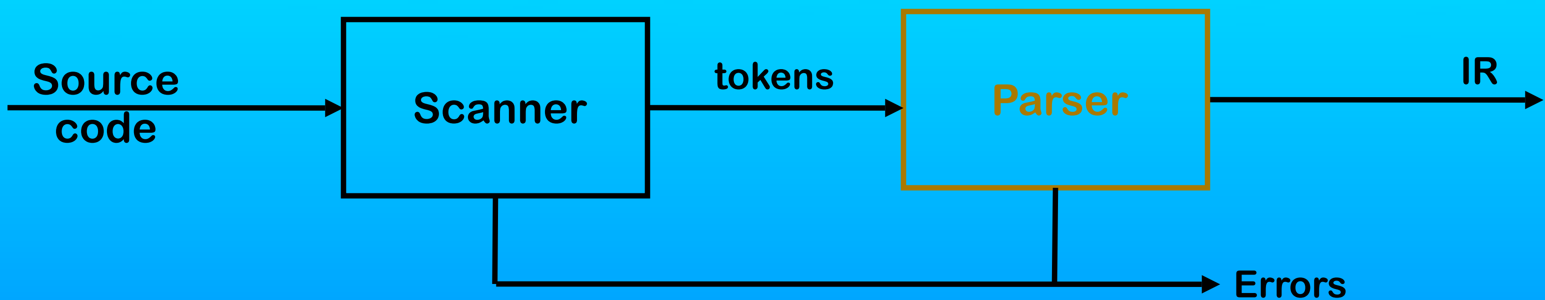
The Front End



Scanner

- Maps character stream into words—the basic unit of syntax
- 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
  - In casual speech, we call the pair a token
- Typical tokens include number, identifier, +, –, new, while, if
- Scanner eliminates white space
- Speed is important

The Front End



Parser

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

The Front End

Context-free syntax is specified with a grammar

$$\begin{aligned} \text{SheepNoise} \rightarrow & \ \text{SheepNoise} \ \underline{\text{baa}} \\ | & \ \underline{\text{baa}} \end{aligned}$$

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

1. goal → expr
2. expr → expr op term
3. | term
4. term → <u>number</u>
5. | <u>id</u>
6. op → +
7. | -

S = goal

T = { <u>number</u>, <u>id</u>, +, - }

N = { goal, expr, term, op }

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

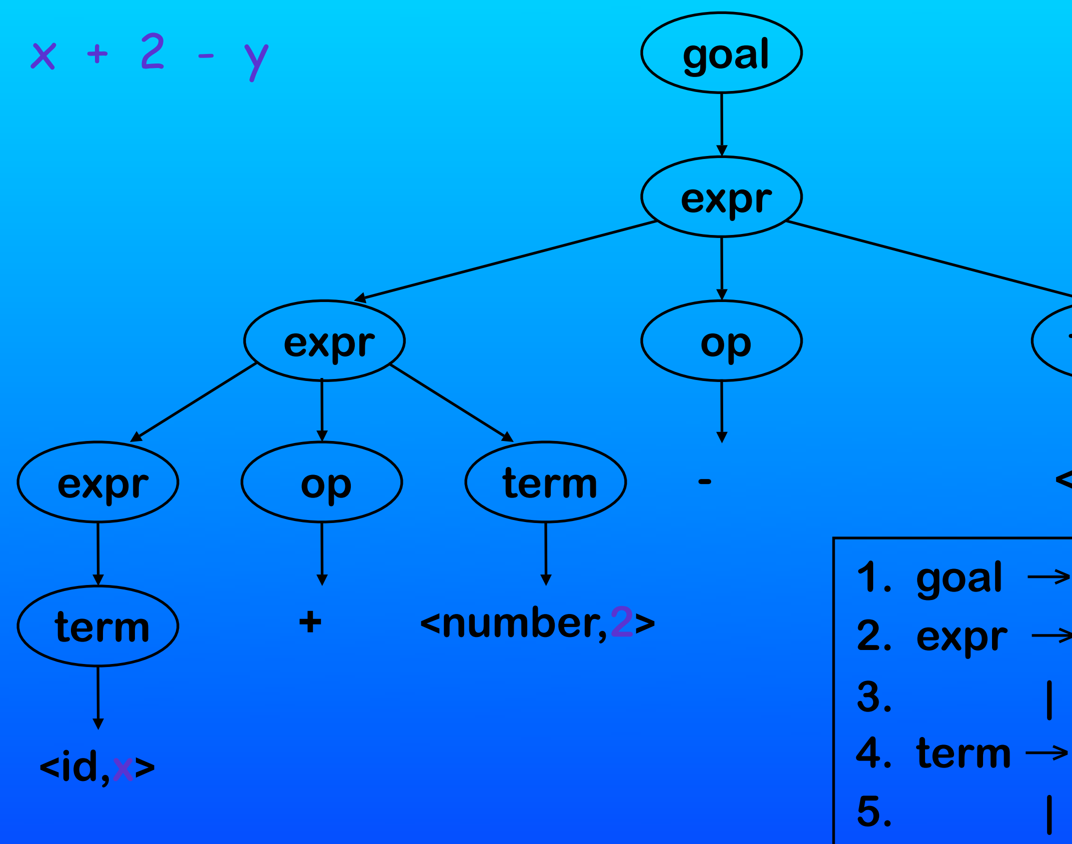
The Front End

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

1. goal → expr
2. expr → expr op term
3. | term
4. term → number
5. | id
6. op → +
7. | -

To recognize a valid sentence in some CFG, we build up a parse tree

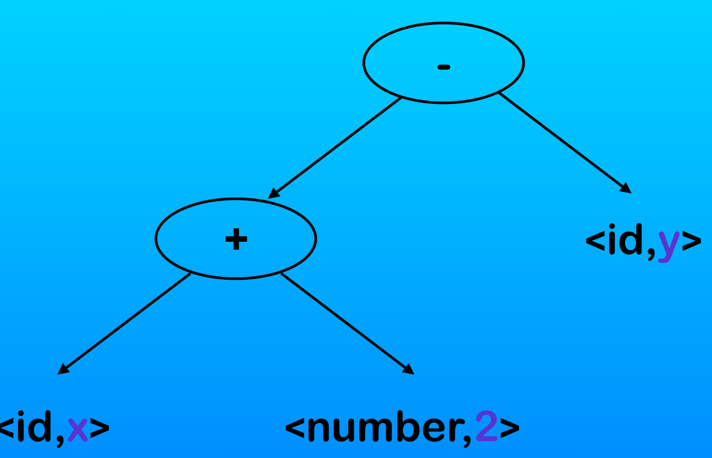
The Front End



1. goal → expr
2. expr → expr op term
3. | term
4. term → number
5. | id
6. op → +
7. | -

This contains a lot of unneeded information.

The Front End

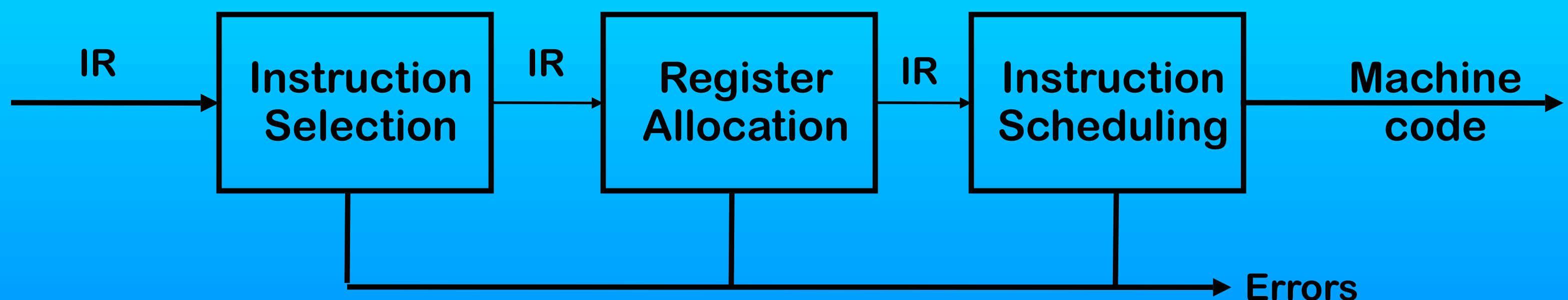


The AST summarizes grammatical structure, without including detail about the derivation

Compilers often use an abstract syntax tree (AST) instead

- This is much more concise
- ASTs are one kind of intermediate representation (IR)
- Also referred to as a directed acyclic graph (DAG)

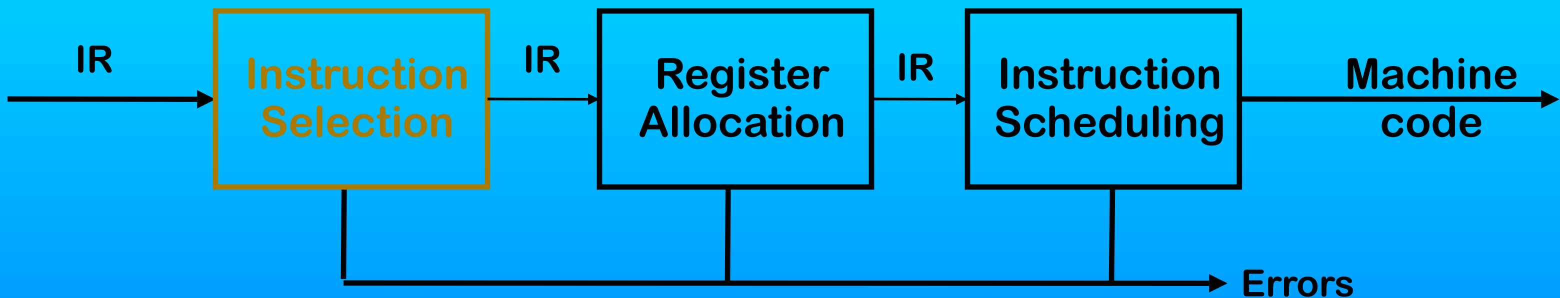
The Back End



Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

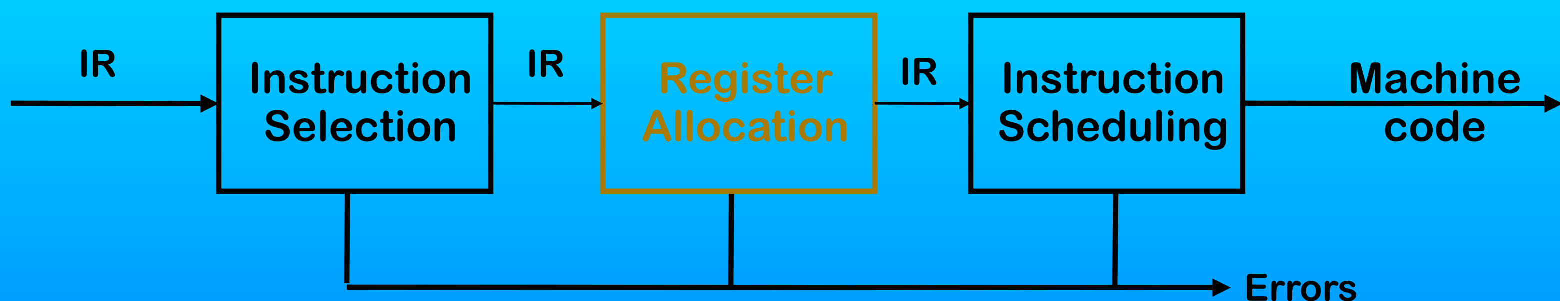
The Back End



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

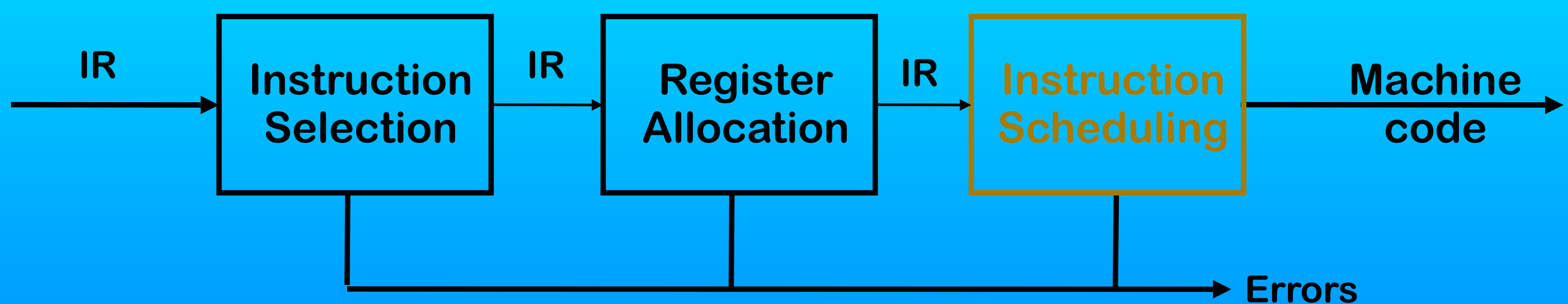
The Back End



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
  - Compilers approximate solutions to NP-Complete problems

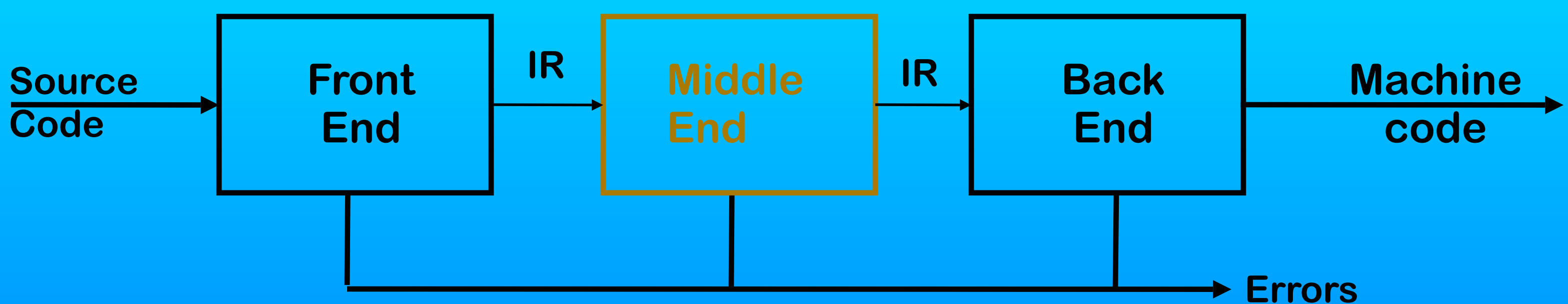
The Back End



Instruction Scheduling

- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables, thus changing the allocation
- Optimal scheduling is NP-Complete in nearly all cases
- Heuristic techniques are well developed

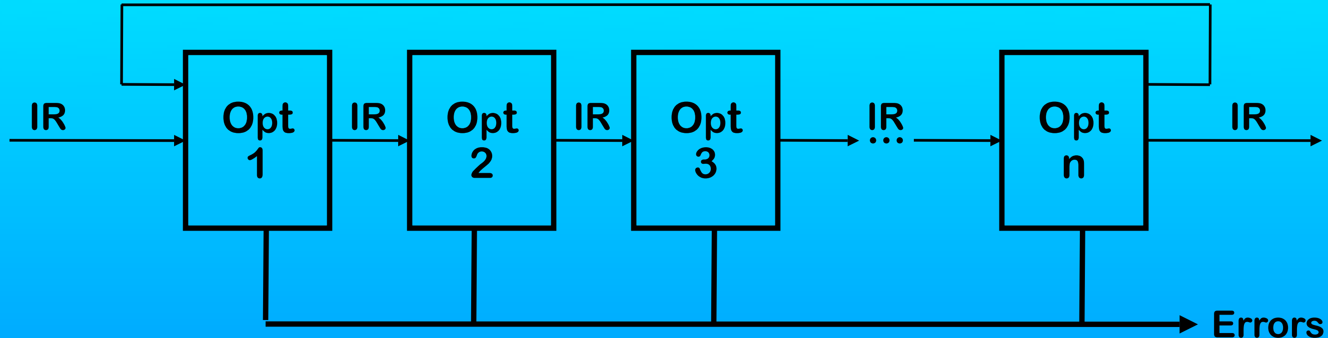
Traditional Three-pass Compiler



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
  - Measured by values of named variables

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

Example

- Optimization of Subscript Expressions in Fortran

Address(A(I,J)) = address(A(0,0)) + J * (column size) + I

Does the user realize a multiplication is generated here?

Example

- Optimization of Subscript Expressions in Fortran

Address(A(I,J)) = address(A(0,0)) + J * (column size) + I

Does the user realize a multiplication is generated here?

```
DO I = 1, M
      A(I,J) = A(I,J) + C
ENDDO
```

Example

- Optimization of Subscript Expressions in Fortran

Address(A(I,J)) = address(A(0,0)) + J * (column size) + I

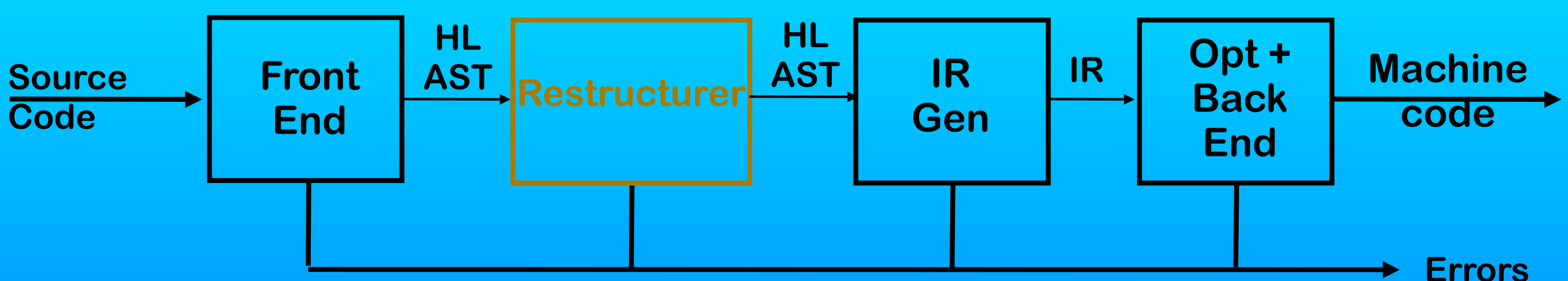
Does the user realize a multiplication is generated here?

```
DO I = 1, M
    A(I,J) = A(I,J) + C
ENDDO
```

→

```
compute addr(A(0,J))
DO I = 1, M
    add 1 to get addr(A(I,J))
    A(I,J) = A(I,J) + C
ENDDO
```

Modern Restructuring Compiler



Typical Restructuring Transformations:

- Blocking for memory hierarchy and register reuse
- Vectorization
- Parallelization
- All based on dependence
- Also full and partial inlining

Role of the Run-time System

- Memory management services
  - Allocate
    - In the heap or in an activation record (stack frame)
  - Deallocate
  - Collect garbage
- Run-time type checking
- Error processing
- Interface to the operating system
  - Input and output
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization