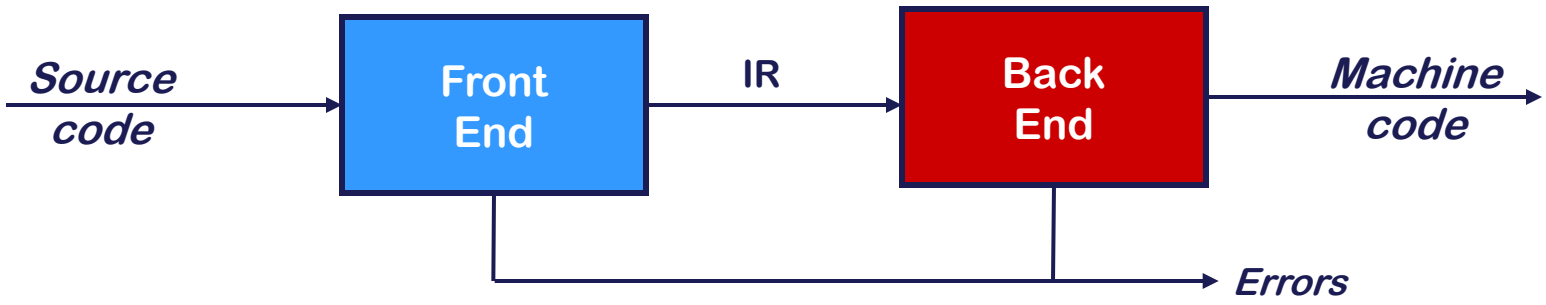
Compiler Design

Introduction to Compiler

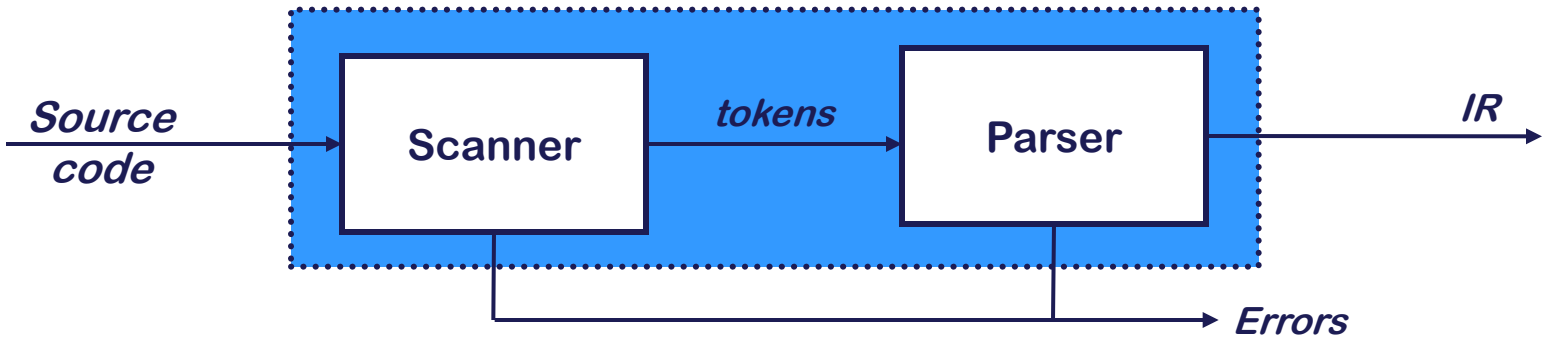
Hwansoo Han

Traditional Two-pass Compiler



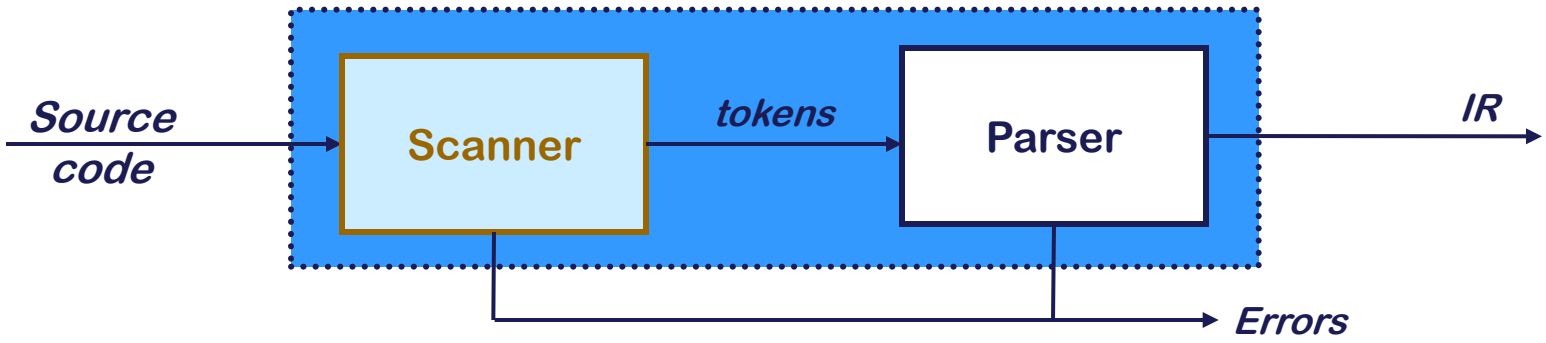
- **High level functions**
  - Recognize legal program, generate correct code (OS & linker can accept)
  - Manage the storage of all variables and code
- **Two passes**
  - Use an intermediate representation (IR)
  - Front end maps legal source code into IR - O(n) or O(n log n)
  - Back end maps IR into target machine code - typically NP-complete
  - Admits multiple front ends & multiple passes (*better code*)

Front End



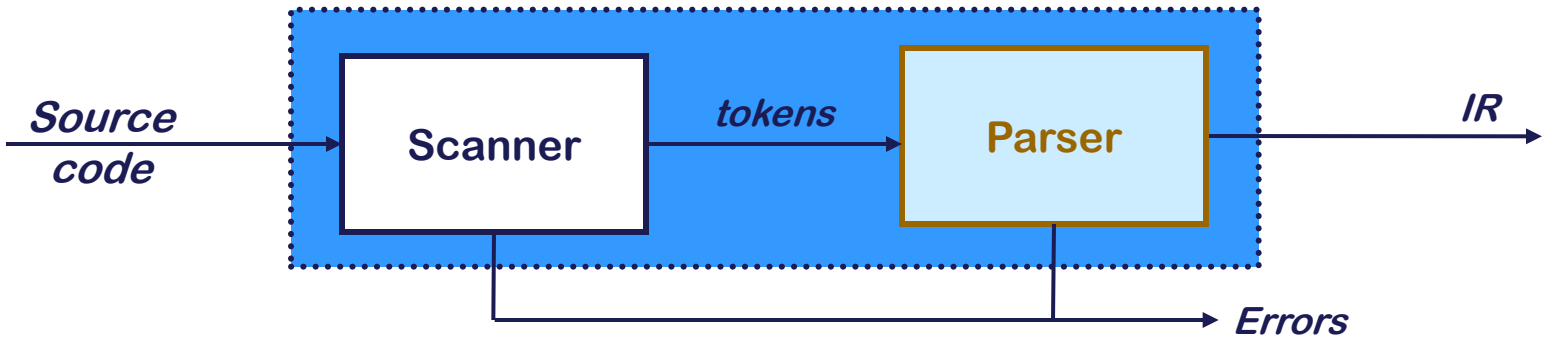
- **Responsibilities**
  - Recognize legal (& illegal) programs
  - Report errors in a useful way
  - Produce IR & preliminary storage map
  - Shape the code for the back end
  - Much of front end construction *can be automated*

Front End - Scanner



- **Scanner**
  - Maps character stream into words (basic units of syntax)
  - Produces tokens — a word & its part of speech
    - x = x + y ; *becomes* <id,x> = <id,x> + <id,y> ;
    - *word ≅ lexeme, part of speech ≅ token type*
  - Typical tokens include *number*, *identifier*, +, -, new, *while*, *if*
    - Scanner eliminates white space
  - Produced by automatic scanner generator

Front End - Parser



- **Parser**
  - Recognizes context-free syntax
  - Guides context-sensitive ("semantic") analysis
    - *E.g. type checking*
  - Builds IR for source program
  - Produced by automatic parser generators

Front End - example (1)

- **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. | -

S = *goal*

T = { number, id, +, - }

N = { *goal*, *expr*, *term*, *op* }

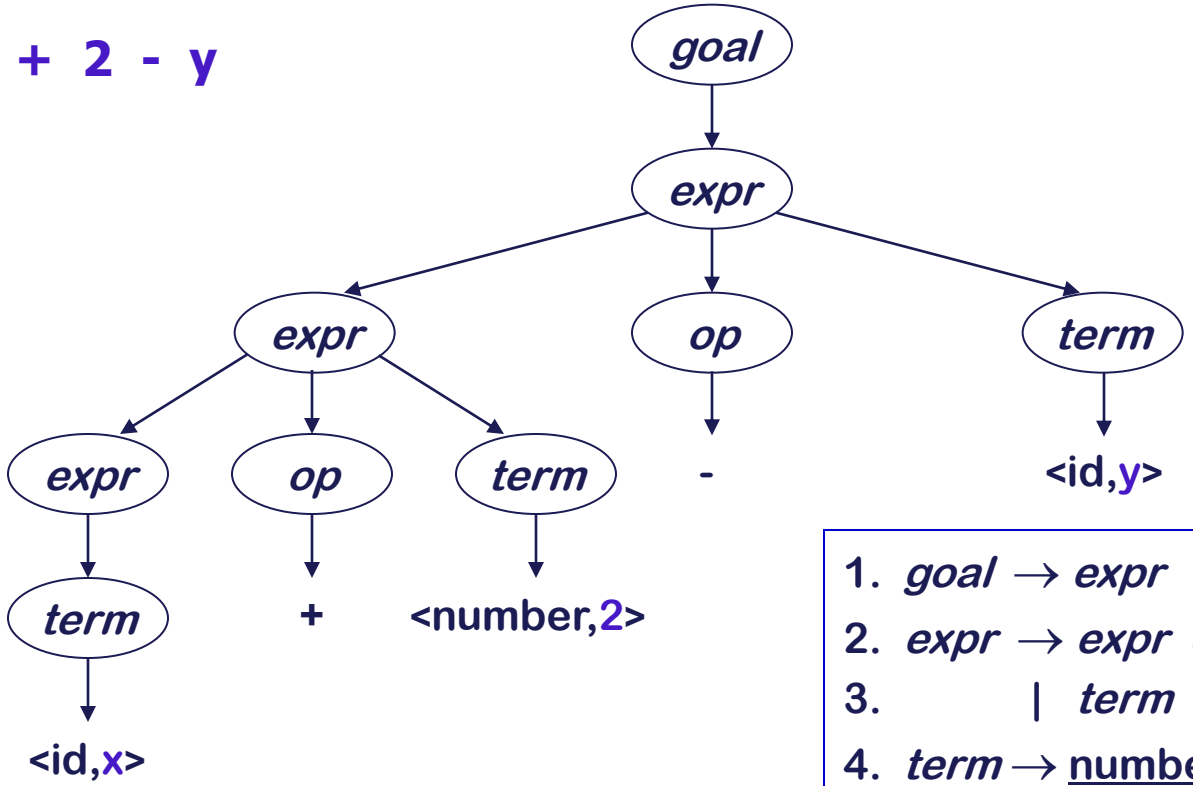
P = { 1, 2, 3, 4, 5, 6, 7}

- 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

Front End - example (2)

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

x + 2 - y

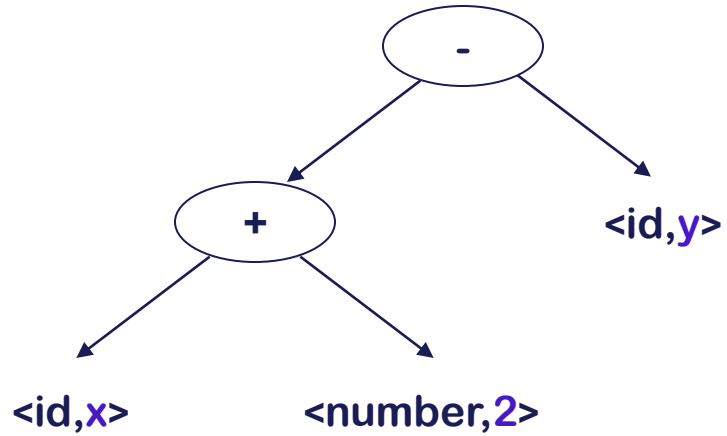


- This contains a lot of unneeded information.

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

Front End - example (3)

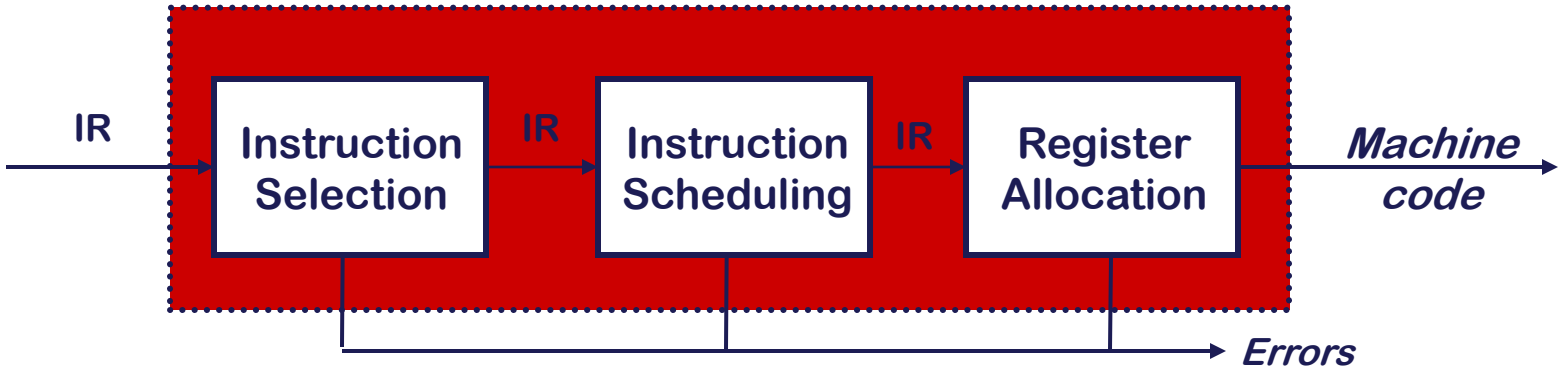
- **Compilers often use an *abstract syntax tree (AST)***



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

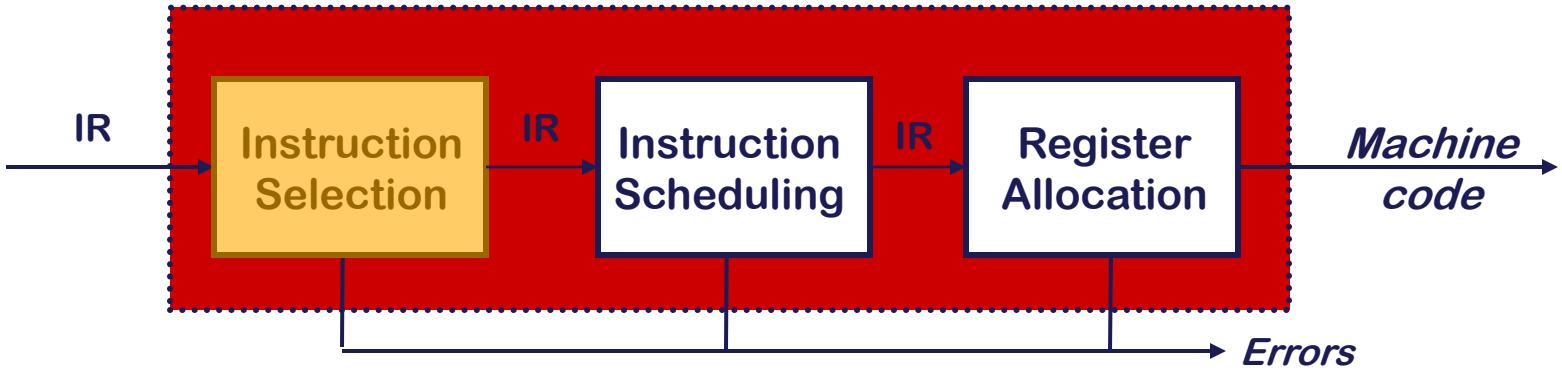
- This is much more concise
- ASTs are one kind of *intermediate representation (IR)*

Back End



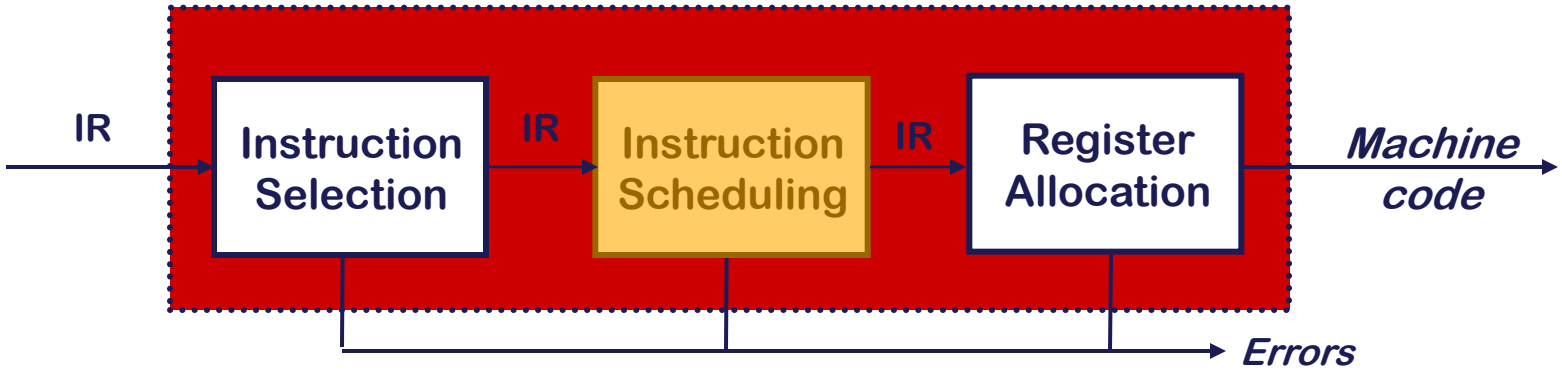
- **Responsibilities**
  - Translate IR into target machine code
  - Choose instructions to implement each IR operation
  - Decide which values to keep in registers
  - Find optimal order of instruction execution
  - Ensure conformance with system interfaces
- **Automation has been *less* successful in the back end**

Back End - Instruction selection



- **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

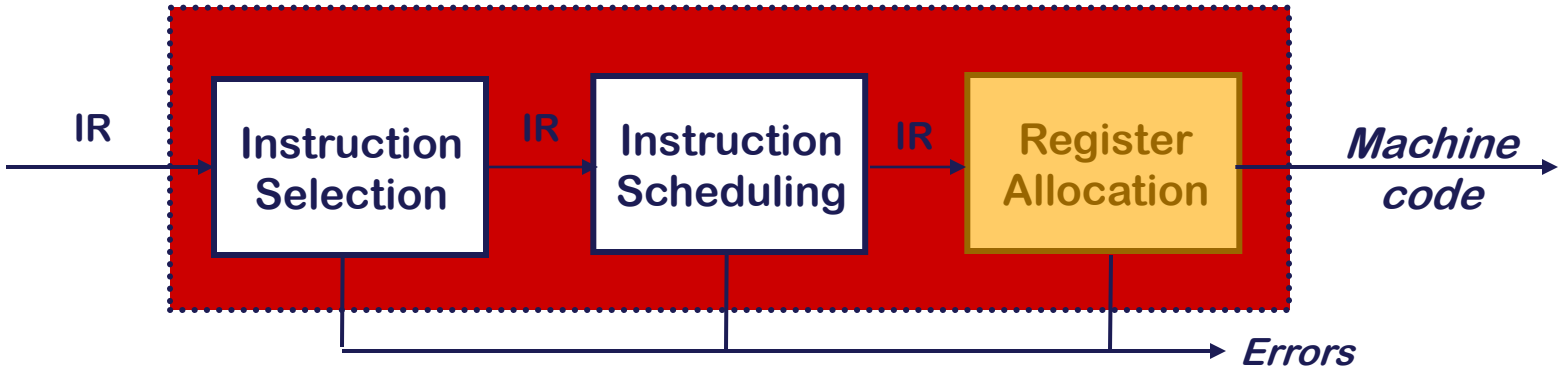
Back End - Instruction scheduling



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

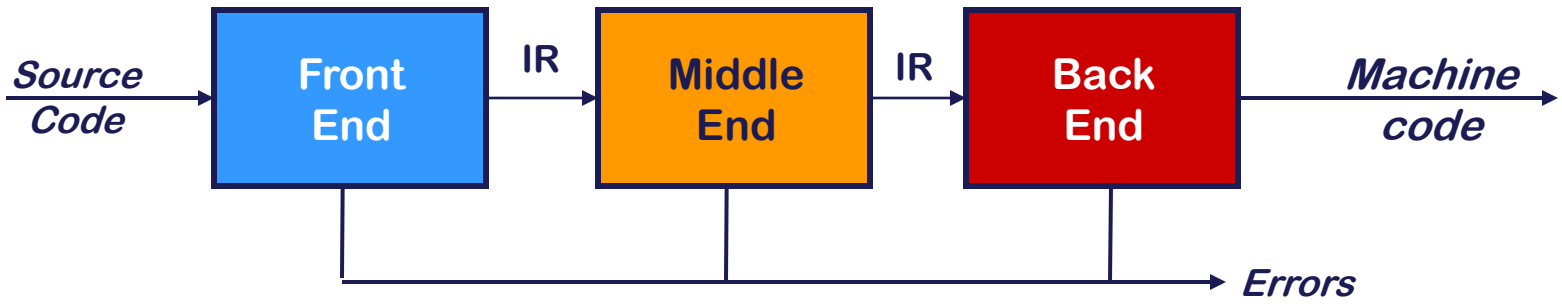
  - Optimal scheduling is NP-Complete in nearly all cases
  - Heuristic techniques are well developed

Back End - Register allocation



- **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

Optimizing Compiler

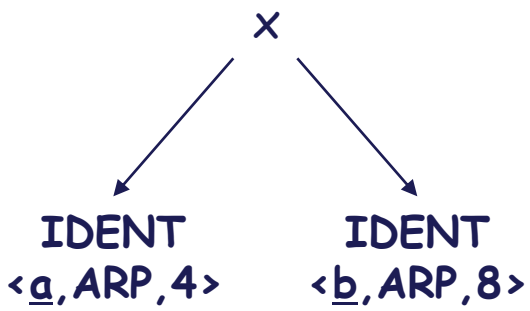


- **Code *Optimizations***
  - Analyzes IR and rewrites (or *transforms*) IR
  - Primary goal is to reduce
    - Execution time,
    - Space usage,
    - Power consumption, …
  - Must preserve "meaning" of the code

Instruction Selection Example

- **Simple Treewalk for initial code**
- **Peephole matching for desired code**

Tree



Treewalk Code

```
loadI    4      ⇒ r5
loadAO   r0,r5  ⇒ r6
loadI    8      ⇒ r7
loadAO   r0,r7  ⇒ r8
mult     r6,r8  ⇒ r9
```

Desired Code

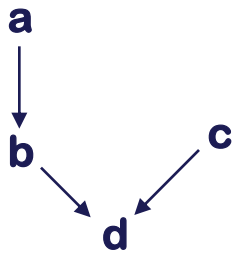
```
loadAI   r0,4   ⇒ r5
loadAI   r0,8   ⇒ r6
mult     r5,r6  ⇒ r7
```

Instruction Scheduling Example

- **Schedule Instructions considering**
  - Latency
  - Dependences
- **Generate fast executing code**

```
a:   loadAI    r0,@w   ⇒ r1
b:   add       r1,r1   ⇒ r1
c:   loadAI    r0,@x   ⇒ r2
d:   mult      r1,r2   ⇒ r1
```

The Code



The Precedence Graph

Register Allocation Example

- **Instruction selection assume infinite number of registers (virtual registers)**
- **Mapping virtual registers to physical registers**
  - Sometime need register spill/fill code

6 virtual registers

```
loadI   4      ⇒ r1
loadAO  r0,r1  ⇒ r2
loadI   8      ⇒ r3
loadAO  r0,r3  ⇒ r4
mult    r2,r4  ⇒ r5
```

(only r5 is used later)

3 physical registers

```
loadI   4      ⇒ r1
loadAO  r0,r1  ⇒ r1
loadI   8      ⇒ r2
loadAO  r0,r2  ⇒ r2
mult    r1,r2  ⇒ r2
```

(r5 is renamed with r2)

Summary

- **Front End**
  - Process high-level programming language
- **Middle End**
  - Apply optimization for speed, power, space, …
- **Back End**
  - Produce machine-level assembly code (or binary code)