

Overview of Compilation (Objectives)

- The student will be able to name and describe the different phases of a compiler.
- The student will be able to name the principles of compiler design

What is a compiler?

- A compiler is just a program that takes other programs and converts them into a form that is understood by an assembler so that it can be assembled, linked and run on a computer



Principles of Compiler Design

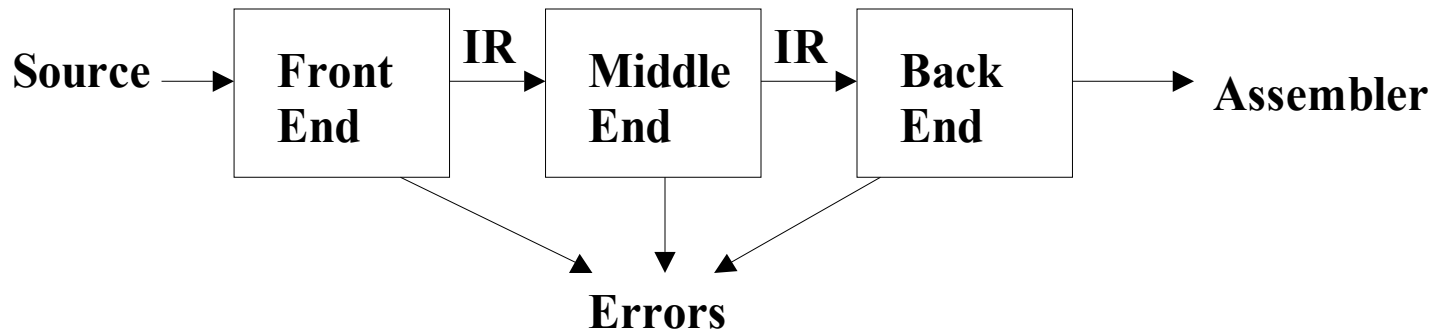
- The compiler must preserve the meaning of the program being compiled
 - The compiler must faithfully implement the defined semantics of a programming language.
- The compiler must improve the source code in a discernable way
 - A direct translation of a source program results in highly inefficient code.

Some Possible Constraints

- Code speed
 - Very fast code might be the highest need of an application
 - e.g., weather
- Code size
 - How much space the object code requires
 - e.g., embedded systems
- Feedback
 - How much feedback is given to the user when an error is encountered
- Compile time
 - programs need to be compiled as fast as possible
- Debugging support
 - code improvements may make debugging difficult

Overview of Compiler Phases

- Basic phases

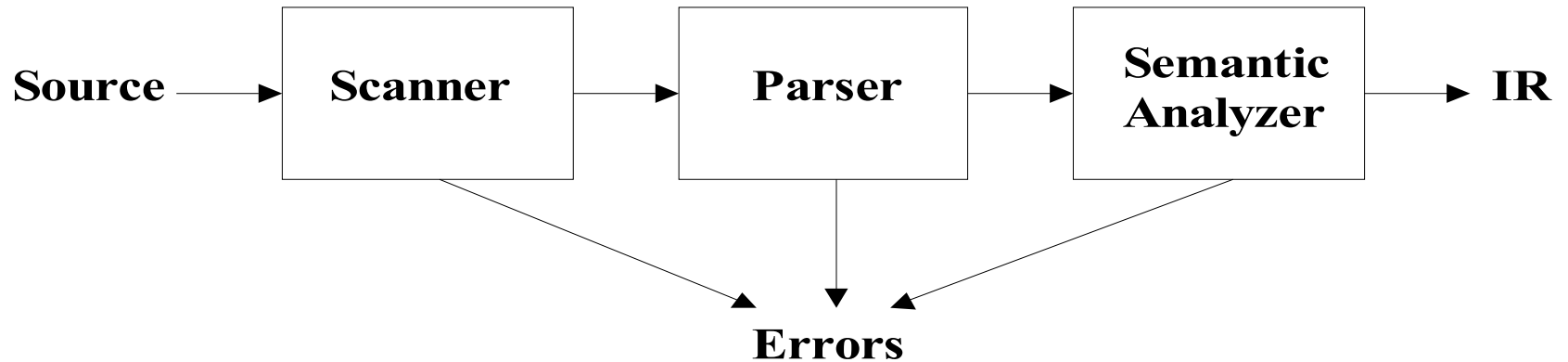


Front End

- **Syntax Analysis**

- determine if programs made up of valid sentences

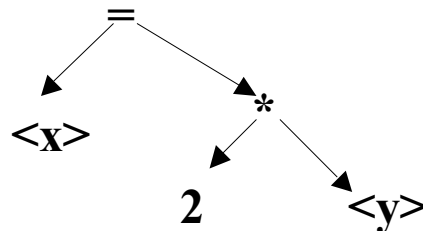
- scanner - valid words (reserved words, variables, etc.)
- parser - valid sentence structure (if statements, etc.)
- semantic analyzer - determine if sentences have meaning (type checking)



Front End

- Lexical Analysis
 - convert words in a program into tokens
 - `<var>`
 - `+, -, =`
 - `IF, THEN, FOR`
- Parsing
 - convert sentences into their structure

$x = 2 * y$

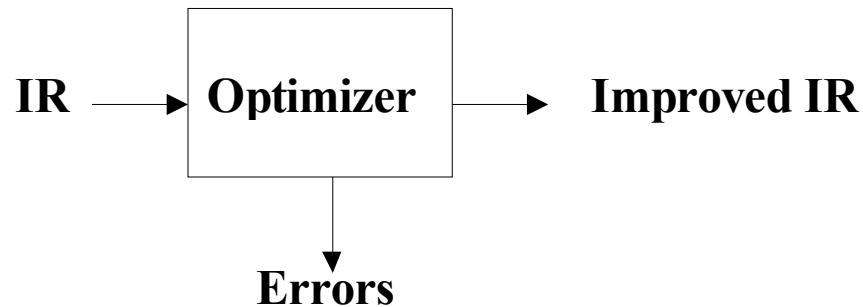


Front End

- Semantic Analysis (context-sensitive analysis)
 - after the program is parsed it is checked to make sure its meaning can be determined
 - type checking
 - number of parameters
 - variables declared
 - functions have prototypes
 - recursion supported or not
- If the program passes semantic analysis, it is converted into an intermediate representation (IR) that is used by the back end

Middle End

- Basic structure
 - often the IR is like assembler



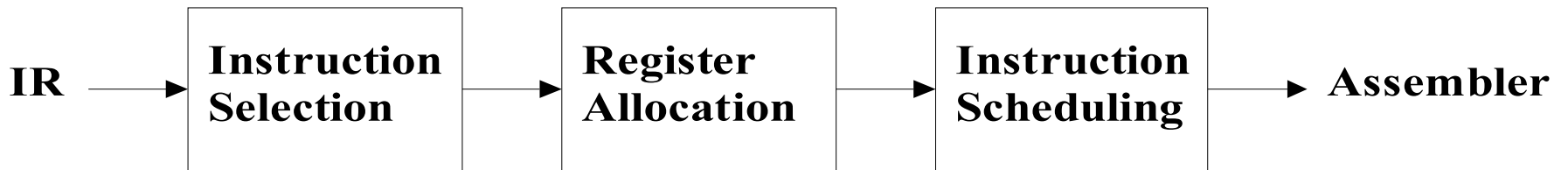
- Optimizer is machine-independent

Middle End

- Optimization
 - remove redundancy
 - remove useless code
 - move code out of loops
 - use constants where possible
 - use less expensive operations

Back End

- Basic phases



- Order of allocation and scheduling may be different

Example

- Consider the following

$$w = w * 2 * x * y * w * 2$$

- Compiler first must recognize
 - variable names
 - =, *
- Next it must determine that the statement is in the source language
- Then, it must make sure the types are correct

Example

- Next it must allocate space for the variables
 - stack?
 - data segment?
 - registers?
- Then, the front end might generate

loadAI	$r_{sp}, @w \rightarrow r_w$
loadI	$2 \rightarrow r_2$
mult	$r_w, r_2 \rightarrow r_{t1}$
loadAI	$r_{sp}, @x \rightarrow r_x$
mult	$r_x, r_{t1} \rightarrow r_{t2}$
loadAI	$r_{sp}, @y, r_y$
mult	$r_y, r_{t2} \rightarrow r_{t3}$
loadAI	$r_{sp}, @w \rightarrow r_w$
mult	$r_w, r_{t3} \rightarrow r_{t4}$
loadI	$2 \rightarrow r_2$
mult	$r_2, r_{t4} \rightarrow r_{t5}$
storeAI	$r_{t5} \rightarrow r_{sp}, r_w$

Example

- Optimization is next
 - eliminate extra loads of w and 2 and extra multiplication of $w*2$

```
loadAI  rsp, @w → rw
loadI   2 → r2
mult    rw, r2 → rt1
loadAI  rsp, @x → rx
mult    rx, rt1 → rt2
loadAI  rsp, @y, ry
mult    ry, rt2 → rt3
mult    rt1, rt3 → rt4
storeAI rt4 → rsp, rw
```

- Code generation converts the intermediate into the target machine's assembler.