



# BCS 307 – Compiler Construction

Cr Hr. 3+1

# Agenda

- Introduction to Compilers
- Compilation vs Interpretation
- Implementation strategies
- Compiler Structure
- Phases of Compilation

## The point is...

- Execute this!

```
int num= 4;  
int factorial = 1;  
while (num > 1) {  
    factorial = factorial*num;  
    num=num-1;  
}
```

- How can computers execute this? Computers only know 1's and 0's

# Interpreters & Compilers

- Interpreter
  - A program that reads an source program and produces the results of executing that program
- Compiler
  - A program that translates a program from one language (the *source*) to another (the *target*)

# Common Issues

- Compilers and interpreters both must read the input – a stream of characters – and “understand” it; *analysis*

```
w h i l e ( k < l e n g t h ) { <nl> <tab> i f ( a [ k ] > 0 ) <nl> <tab>  
<tab>{ n P o s + + ; } <nl> <tab> }
```

# Interpreter

- Interpreter

- Execution engine
- Program execution interleaved with analysis

```
running = true;  
while (running) {  
    analyze next statement;  
    execute that statement;  
}
```

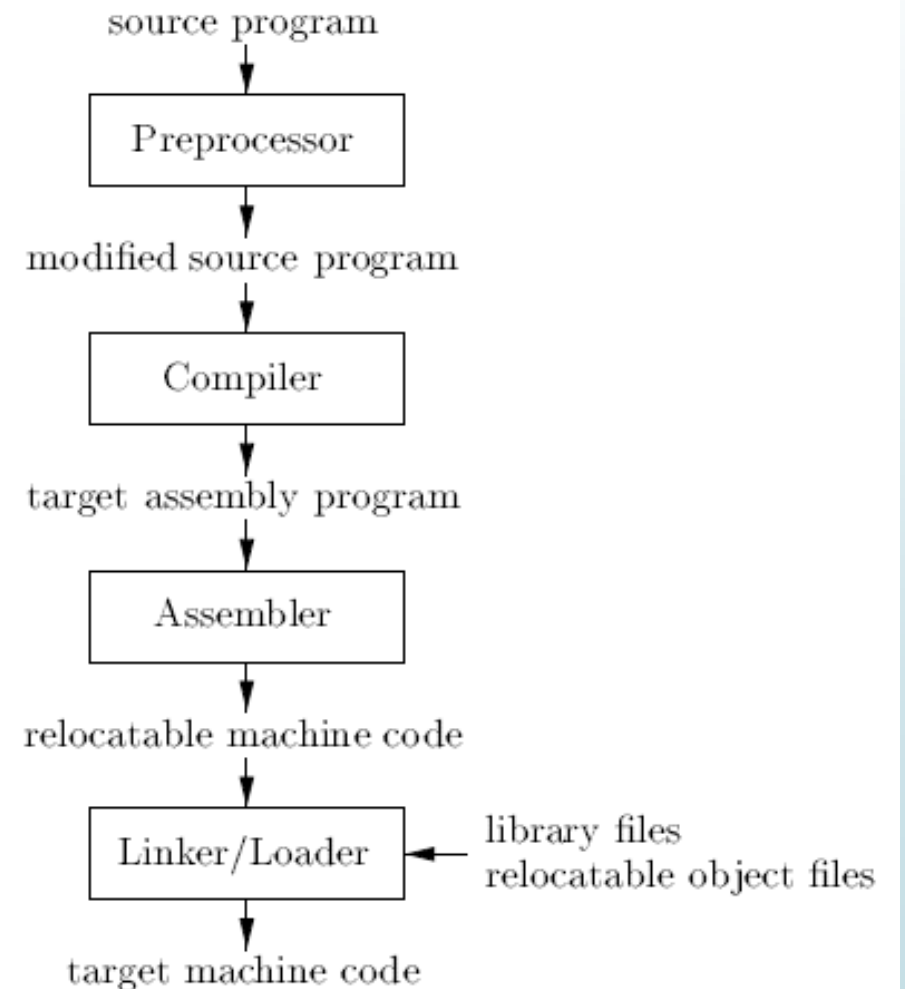
- Usually need repeated analysis of statements (particularly in loops, functions)
- But: immediate execution, good debugging & interaction

# Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier to execute or more efficient
  - Should “improve” the program in some fashion
- Offline process
  - Tradeoff: compile time overhead (preprocessing step) vs execution performance

# Processes Before and After Compilation

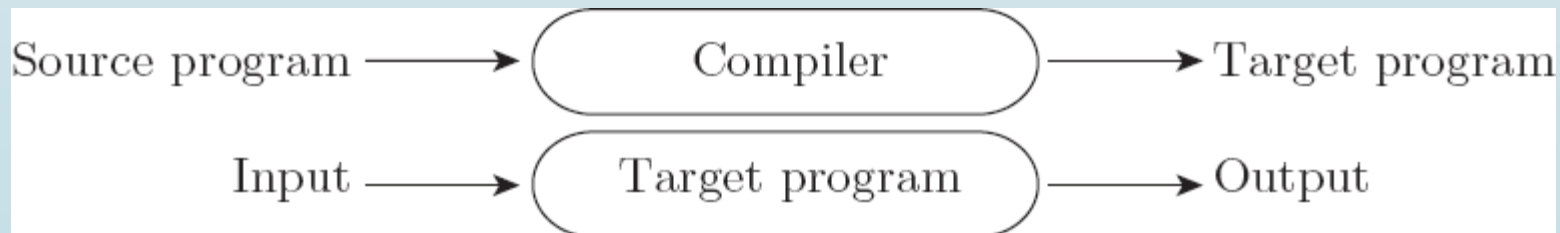
- A program usually goes through some process before and after compilation





# Compilation vs Interpretation (1)

- Not a clear-cut distinction
- Pure Compilation
  - The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



## ➡ Pure Interpretation

- ➡ Interpreter stays around for the execution of the program
- ➡ Interpreter keeps control during execution

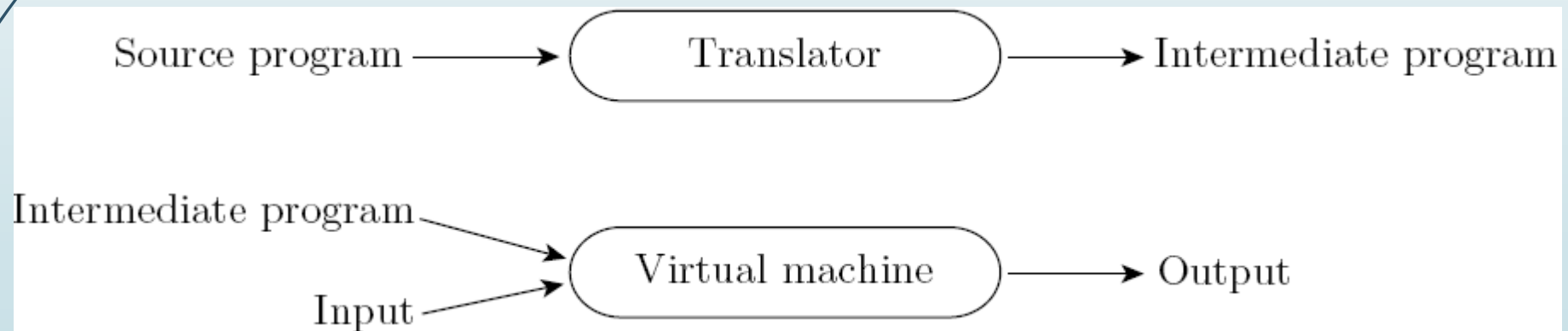


# Compilation vs Interpretation (3)

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)
- Compilation
  - Better performance

# Compilation vs Interpretation (4)

- Some language implementations include a both compilation and interpretation
- Compilation or simple pre-processing, followed by interpretation



# Implementation strategies (1)

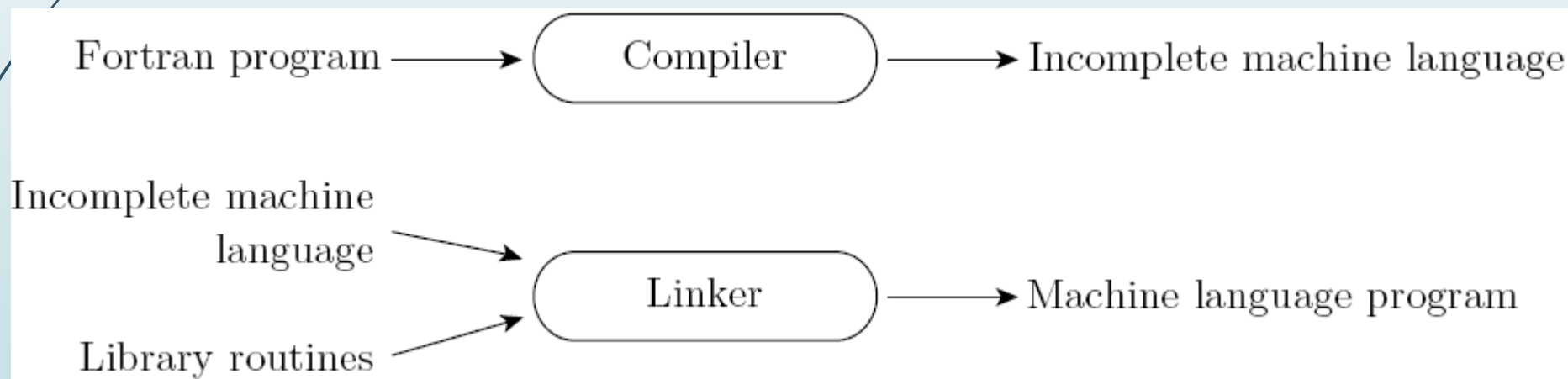
## ■ Preprocessor

- Removes comments and white space
- Groups characters into *tokens* (keywords, identifiers, numbers, symbols)
- Expands abbreviations in the style of a macro assembler
- Identifies higher-level syntactic structures (loops, subroutines)

# Implementation strategies (2)

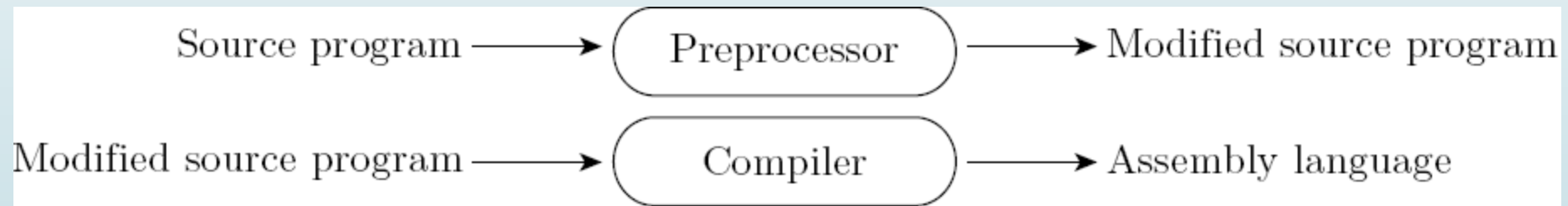
## Library of Routines and Linking

- Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:



# Implementation strategies (3)

- The C Preprocessor (conditional compilation)
  - Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source



# Dynamic and Just-in-Time Compilation

16

- In some cases a programming system may deliberately delay compilation until the last possible moment.
  - The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs.
  - The main C# compiler produces .NET Common Language Runtime (CLR), which is then translated into machine code immediately prior to execution.



# Implementations

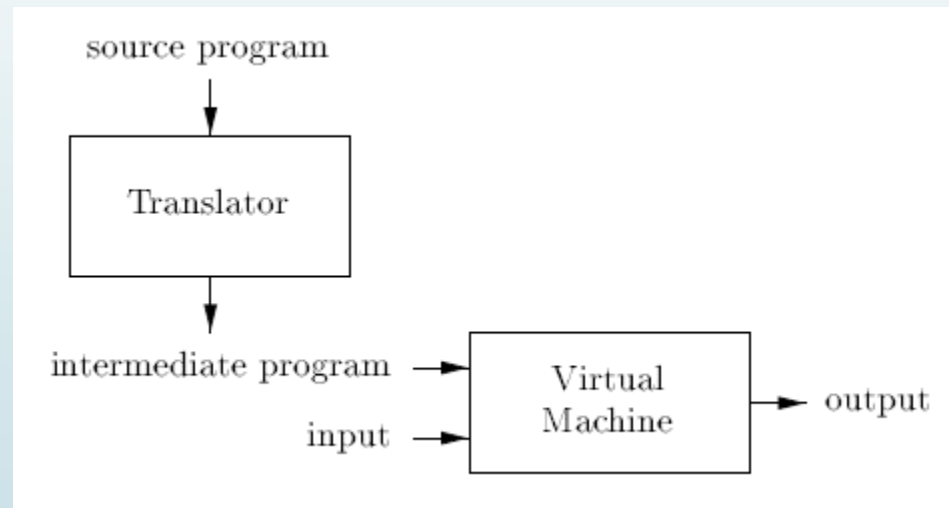
- Compilers
  - C#, C, C++, Java etc.
  - Strong need for optimization
- Interpreters
  - PERL, Python, Ruby, awk, sed, shells, Scheme/Lisp/ML, postscript/pdf, Java VM
  - Particularly effective if interpreter overhead is low

# Hybrid Approaches (1)

- Classic example: Java
  - Compile Java source to byte codes (.class files)
  - Execution
    - Interpret byte codes directly, or
    - Compile some or all byte codes to native code
      - Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code
- Variations used for .NET & implementations of dynamic and functional languages, e.g., JavaScript, Haskell

## Hybrid Approaches (2)

- A typical hybrid compilation



# Why Study Compilers? (1)

- Become a better programmer
  - Insight into interaction between languages, compilers, and hardware
  - Understanding of implementation techniques
  - Know the stuff in the debugger
  - Better intuition about what your code does

# Why Study Compilers? (2)

- Compiler techniques are everywhere
  - Parsing (little languages, interpreters, XML)
  - Software tools (verifiers, checkers, ...)
  - Database engines, query languages
  - AI, etc.: domain-specific languages
  - Text processing
    - Tex/LaTeX -> dvi -> Postscript -> pdf
  - Hardware: VHDL; model-checking tools
  - Mathematics (Mathematica, Matlab)

# Why Study Compilers? (3)

- ▶ Blend of theory and engineering
  - ▶ Applications of theory to practice
    - ▶ Parsing, scanning, static analysis
  - ▶ Some very difficult problems (NP-hard or worse)
    - ▶ Resource allocation, “optimization”, etc.
    - ▶ Need to come up with good-enough approximations/heuristics

# Why Study Compilers? (4)

- Ideas from many parts of CSE
  - AI: Greedy algorithms, heuristic search
  - Algorithms: graph algorithms, dynamic programming, approximation algorithms
  - Theory: Grammars, DFAs and PDAs, pattern matching, fixed-point algorithms
  - Systems: Allocation & naming, synchronization, locality
  - Architecture: pipelines, instruction set use, memory hierarchy management

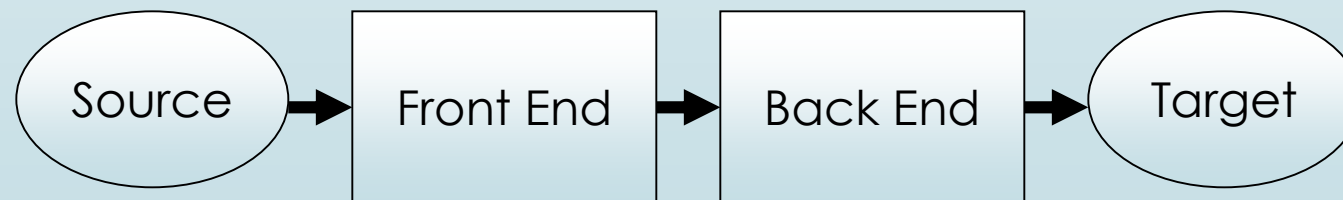
# Why Study Compilers? (5)

- ▶ You may write a compiler yourself
- ▶ You can write parsers and interpreters for little languages
  - ▶ XML, Command languages, configuration files,, ...



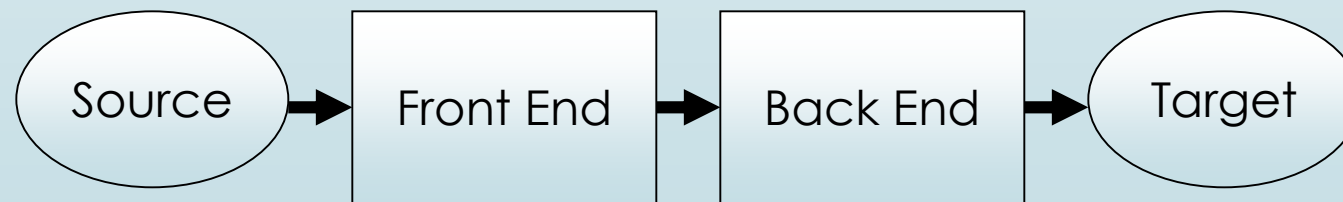
# Structure of a Compiler

- First approximation
  - Front end: analysis
    - Read source program and understand its structure and meaning
  - Back end: synthesis
    - Generate equivalent target language program



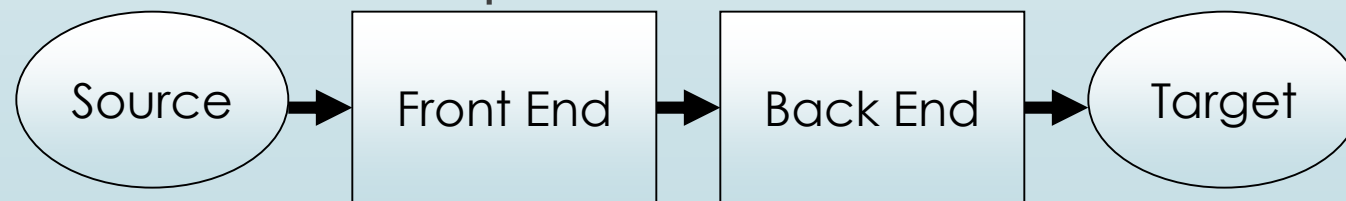
## Compiler must...

- recognize legal programs (& complain about illegal ones)
- generate correct code
- manage storage of all variables/data
- agree with OS & linker on target format



# Implications

- Need some sort of Intermediate Representation(s) (IR)
- Front end maps source into IR
- Back end maps IR to target machine code
- Often multiple IRs – higher level at first, lower level in later phases



# Programming Environment Tools

- Tools integrated in an Integrated Development Environment (IDE)

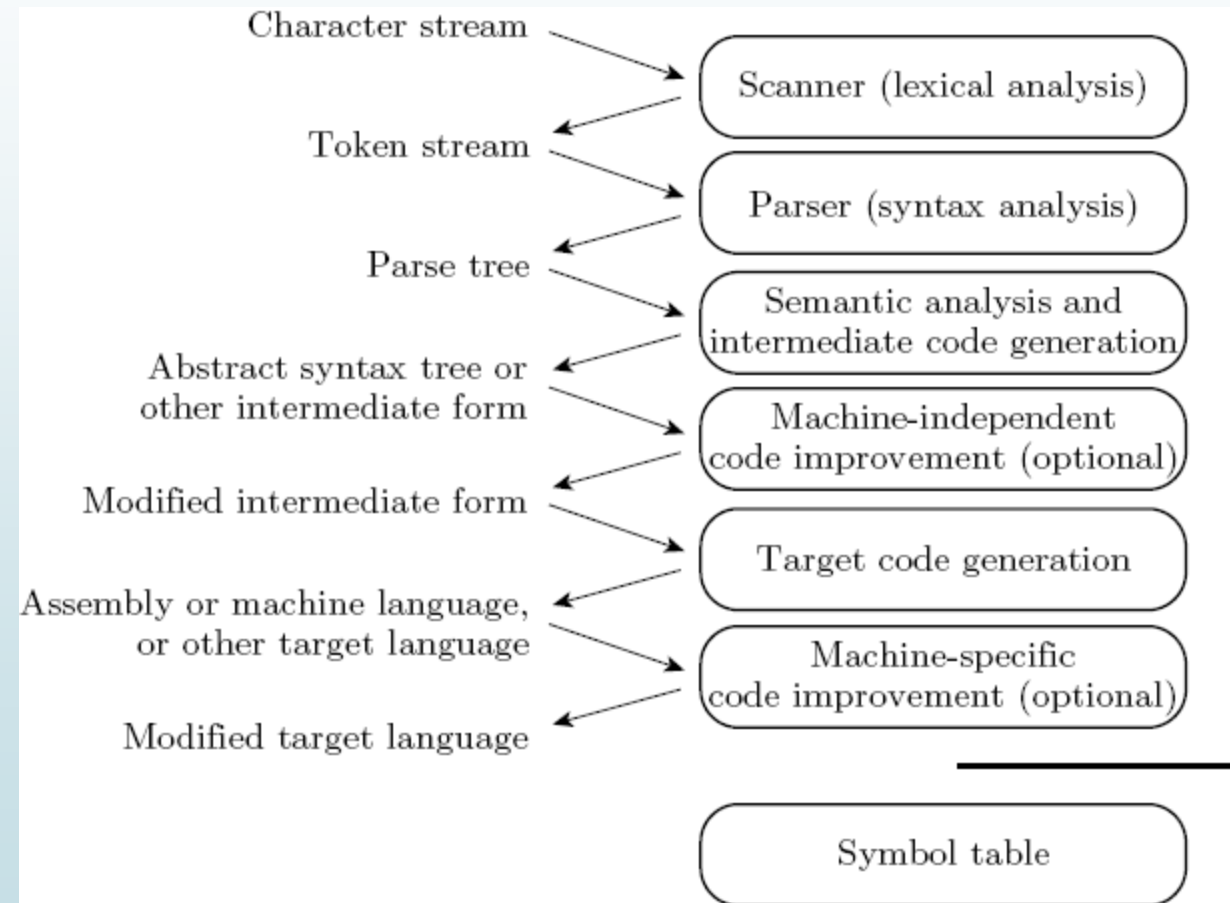
Type	Unix examples
Editors	vi, emacs
Pretty printers	cb, indent
Pre-processors (esp. macros)	cpp, m4, watfor
Debuggers	adb, sdb, dbx, gdb
Style checkers	lint, purify
Module management	make
Version management	sccs, rcs
Assemblers	as
Link editors, loaders	ld, ld-so
Perusal tools	More, less, od, nm
Program cross-reference	ctags

# Structure of Compiler

- Inside a compiler, there are two main parts: analysis and synthesis.
- Analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
  - If it detects either syntactically ill formed or semantically unsound source code, it must provide error messages.
  - It also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.
- Synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called the **front end** of the compiler; the synthesis part is the **back end**.

# Phases of Compilation

30



# Compilation Overview - Scanning

- Scanning is recognition of a *regular language*, with DFA (deterministic finite automaton).
- Divides the program into "tokens", the smallest meaningful units.
- It also saves complexity for later phases.

# Compilation Overview - Parsing

32

- Recognition of a *context-free language*, e.g., using Pushdown Automaton (PDA)
  - Parsing discovers the "context free" structure of the program.
  - Informally, it finds the structure you can describe with syntax diagrams.



# Compilation Overview - Semantic analysis

33

- The discovery of *meaning* in the program
  - The compiler actually does what is called **static** semantic analysis (at compile time).
  - Things like array subscript out of bounds can't be figured out until run time, so they are part of the program's **dynamic** semantics.

## Compilation Overview - Intermediate form

- After the program passes all checks, intermediate code may be generated.
  - IFs are often chosen for machine independence, ease of optimization, or compactness.
  - Often resemble machine code for some imaginary machine architecture.
  - Some compilers move the code through more than one IF.

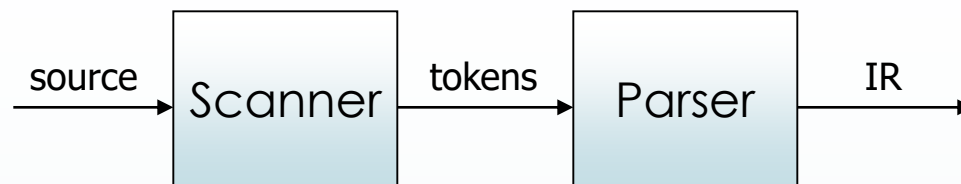
# Compilation Overview – Code Optimization and Generation (1)

- Takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - The optimization phase is optional
- **Code generation phase** produces assembly language or relocatable machine language (relative address is different from its absolute address)

## Compilation Overview – Code Optimization and Generation (2)

- Certain ***machine-specific optimizations*** (use of special instructions or addressing modes, etc.) may be performed during ***target code generation***
- ***Symbol table***: all phases rely on a symbol table that keeps track of all the identifiers in the program.

## Front End



- **Scanner:** Responsible for converting character stream to token stream
  - Also strips out white space, comments
- **Parser:** Reads token stream; generates IR Source language specified by a formal grammar
  - There are some tools that can read the grammar and generate scanner & parser

# Scanner Example

## ► Input text

```
// this statement does very little  
if (x >= y) y = 42;
```

## ► Token Stream

IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

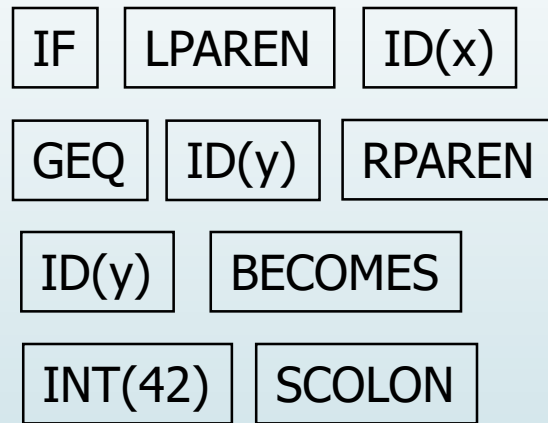
- Notes: tokens are atomic items, not character strings; comments & whitespace are *not* tokens (not true for all languages, cf. Python)

## Parser Output (IR)

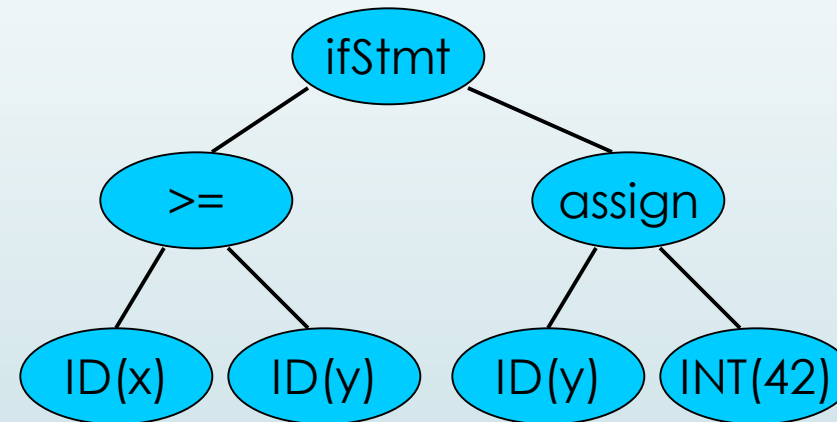
- Many different forms.
  - Engineering tradeoffs have changed over time (e.g., memory is almost free these days).
- Common output from a parser is an abstract syntax tree.
  - Essential meaning of the program without the syntactic noise.

# Parser Example

## Token Stream Input



## Abstract Syntax Tree





# Static Semantic Analysis

- During or (more commonly) after parsing
  - Type checking
  - Check language requirements like proper declarations, etc.
  - Preliminary resource allocation
  - Collect other information needed by back-end code generation

# Back-End

- Responsibilities
  - Translate IR into target machine code
  - Should produce “good” code
    - “good” = fast, compact, low power (pick some)
  - Should use machine resources effectively
    - Registers
    - Instructions & function units
    - Memory hierarchy

# Back-End Structure

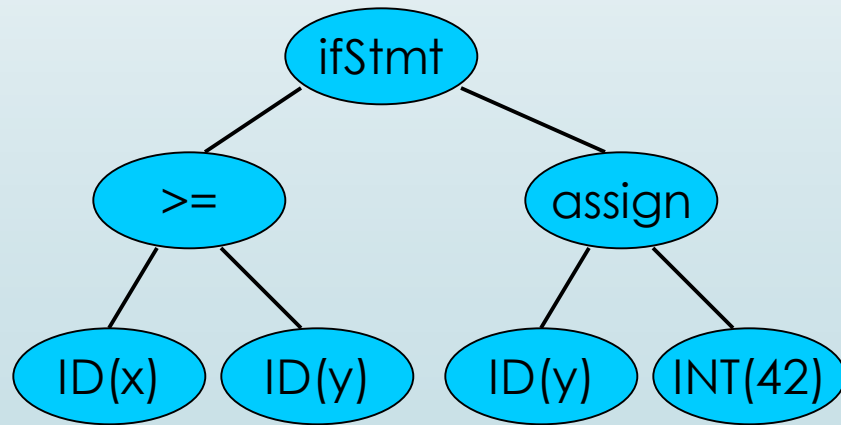
- Typically split into two major parts
  - “Optimization” – code improvements
    - Usually works on lower-level IR than AST
  - Code generation
    - Instruction selection & scheduling
    - Register allocation

# The Result

## Input

if (x >= y)

y = 42;



## Output

```
mov  eax,[ebp+16]
```

```
cmp  eax,[ebp-8]
```

```
jl   L17
```

```
mov  [ebp-8],42
```

```
L17:
```

# Some History (1)

- 1950's
  - FORTRAN I (1954) – competitive with hand-optimized code
- 1960's
  - New languages: ALGOL, LISP, COBOL, SIMULA
  - Formal notations for syntax, esp. BNF
  - Fundamental implementation techniques
    - Stack frames, recursive procedures, etc.

## Some History (2)

- 1970's
  - Syntax: formal methods for producing compiler front-ends;
- Late 1970's, 1980's
  - New languages (functional; object-oriented - Smalltalk)
  - New architectures (RISC machines, parallel machines, memory hierarchy)

## Some History (3)

### ► 1990s

- Techniques for compiling objects and classes, efficiency in the presence of dynamic program elements and short methods (Self, Smalltalk –JVMs, etc.)
- Just-in-time compilers (JITs)
- Compiler technology critical to effective use of new hardware (RISC, Itanium, parallel machines, complex memory hierarchies)

## Some History (4)

- Last decade
  - Compilation techniques in many new places.
    - Software analysis, verification, security.
  - Phased compilation – blurring the lines between “compile time” and “runtime”.
  - Dynamic languages – e.g., JavaScript, ...
  - Compilers for parallel systems.



# Course Project

- Compiler construction is best learnt by building it (at least some parts).
- Course project should implement Lexical Analyser and Syntax Analyser.
  - You can go further ...

# Programming Environments

- Whatever you want!
  - But you can use C# as you are already familiar with it, and its quick way to develop programs.
  - For IDE, you can use Visual Studio

## Some Resources

- The GNU Compiler Collection (gcc) consists of open-source compilers for C, C++, Fortran, Java, and other languages.
- Phoenix is a compiler-construction toolkit that provides an integrated framework for building the program analysis, code generation, and code optimization phases of compilers.

# Prerequisites

- Courses in:
  - Data structures & algorithms
    - Linked lists, dictionaries, trees, hash tables, Formal languages & automata
    - Regular expressions, finite automata, context-free grammars, maybe a little parsing
  - Machine organization
    - Assembly-level programming

# Grading Policy

- As devised by the university

# Online Lectures

- ▶ Lectures will be delivered online on MS Teams
- ▶ Lecture notes will be available on the same

# Communications

- At the end of each session, you will have time for questions.
- You may also post your queries on Teams, or my email.

# Books

- Include:

- Aho, Lam, Sethi, Ullman, "Compilers Principles, Tools and Techniques", 2<sup>nd</sup> edition
- Cooper & Torczon, *Engineering a Compiler*



# Questions?

- Make sure you understand the concepts. Ask questions where find problems.
  - As the course proceeds, try implementing the concepts as we move on.

# Upcoming Topics

- ▶ Language Basics
- ▶ Lexical analysis – scanning
- ❖ Its helpful to read the first few chapters of the book beforehand.