Introduction to Compilation

Wei Wang

Course Instructor

- Name:
  - Wei Wang
- Office:
  - NPB 3.210
- Email:
  - wei.wang@utsa.edu
  - Always include “CS5363” in the title
- Research Areas:
  - Compiler, Architecture, Cloud and SE

Course Meetings, Web Pages, etc.

- Meetings:
  - TR 7:30-8:45pm
- Office Hours:
  - Mon 3:00-5:00pm, and Thu 4:00-5:00pm
  - In Zoom, link posted in Blackboard
- Website

  https://wwang.github.io/teaching/Spring2021/CS5363/Syllabus/general_info.html

Online Teaching

- Lectures:
  - I will record and post lectures in Panopto.
- Exams:
  - Will be delivered through Blackboard
  - For privacy reason, I can’t proctor the exam as in-person teaching.
- Assignments and Projects
  - Submit through Blackboard

Course Textbooks

- Reference Book
  - Compilers: Principles, Techniques, and Tools, 2nd Edition,
    - by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
    - AKA “Dragon Book”

Course Topics

- Formal Languages and Automaton
- Lexical Analysis
- Parsing
- Code Generation
- Compiler Optimization
- Functional Languages

Grading Scheme

- Mid-Term Exam: 20%
- Final Exam: 20%
- Assignments: 20%
- Projects: 35%
  - Develop a compiler for a C-like language
- Course participation and other extra point opportunities: 5%
  - Be active in class

Other Related Information

- Mid-Term Exam: Mar 18th Thursday, in-class
- Final Exam: Thu, May 13, 07:00 pm - 08:50 pm
- All exam days are fixed.
  - Plan your travel accordingly
  - No make-up exam will be given unless university-sanctioned reasons.
- Prerequisites:
  - You must be able to program in C and C++.
  - CS2233 Discrete Math and CS3343 Algo.
  - It is better if you have taken OS, Arch and undergrad PL.
- Late submission docked with 10% if late within a week. No submissions accepted after a week.

More on the Course Project

- The project consists a number of phases
  - Lexical Analysis
  - Parser
  - Semantic Analysis (two phases)
  - Code Generation
  - Documentation
  - Except Lexical analysis and Documentation, each phase takes about a week of full time programming.
- Do the implementation yourself!
- Must have a functional compiler to get a B or above.

PL and Compilers: An Introduction

Overview and History (1)

- Cause
  - Software for early computers was written in assembly language
  - The benefits of reusing software on different CPUs started to become significantly greater than the cost of writing a compiler
- The first real compiler
  - FORTRAN compilers of the late 1950s
  - 18 person-years to build

11

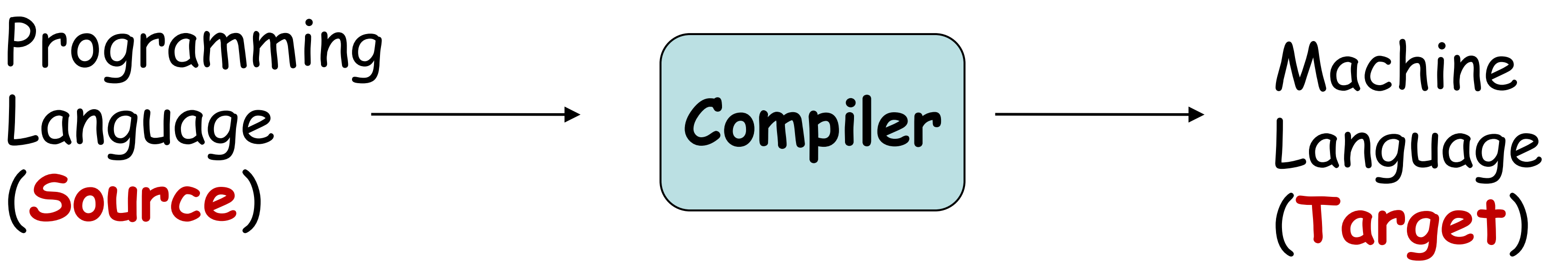
Overview and History (2)

- **Compiler technology**
  - is more broadly applicable and has been employed in rather unexpected areas.
    - ✓ Text-formatting languages like nroff and troff; preprocessor packages like eqn, tbl, pic
    - ✓ Silicon compiler for the creation of VLSI circuits
    - ✓ Command languages of OS
    - ✓ Query languages of Database systems

12

What Do Compilers Do (1)

- A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
  - ✓Ignore machine-dependent details for programmer



What Do Compilers Do (2)

- Compilers may generate three types of code:
  - Pure Machine Code
    - Machine instruction set without assuming the existence of any operating system or library.
    - Mostly being OS or embedded applications.
  - Augmented Machine Code
    - Code with OS routines and runtime support routines.

What Do Compilers Do (2)

- Compilers may generate three types of code:
  - Virtual Machine Code
    - Virtual instructions, can be run on any architecture with a virtual machine interpreter or a just-in-time compiler
    - Ex. Java

What Do Compilers Do (3)

- Another way that compilers differ from one another is in the format of the target machine code they generate:
  - Assembly or other source format
  - Re-locatable binary
    - Relative address
    - A linkage step is required
  - Absolute binary
    - Absolute address
    - Can be executed directly

16

Interpreters & Compilers

- Interpreter
  - A program that reads a 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)

Commonalities

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

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

Interpreter

- Interpreter
  - Execution engine
  - Program execution interleaved with analysis

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

  - May involve repeated analysis of some statements (loops, functions)

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

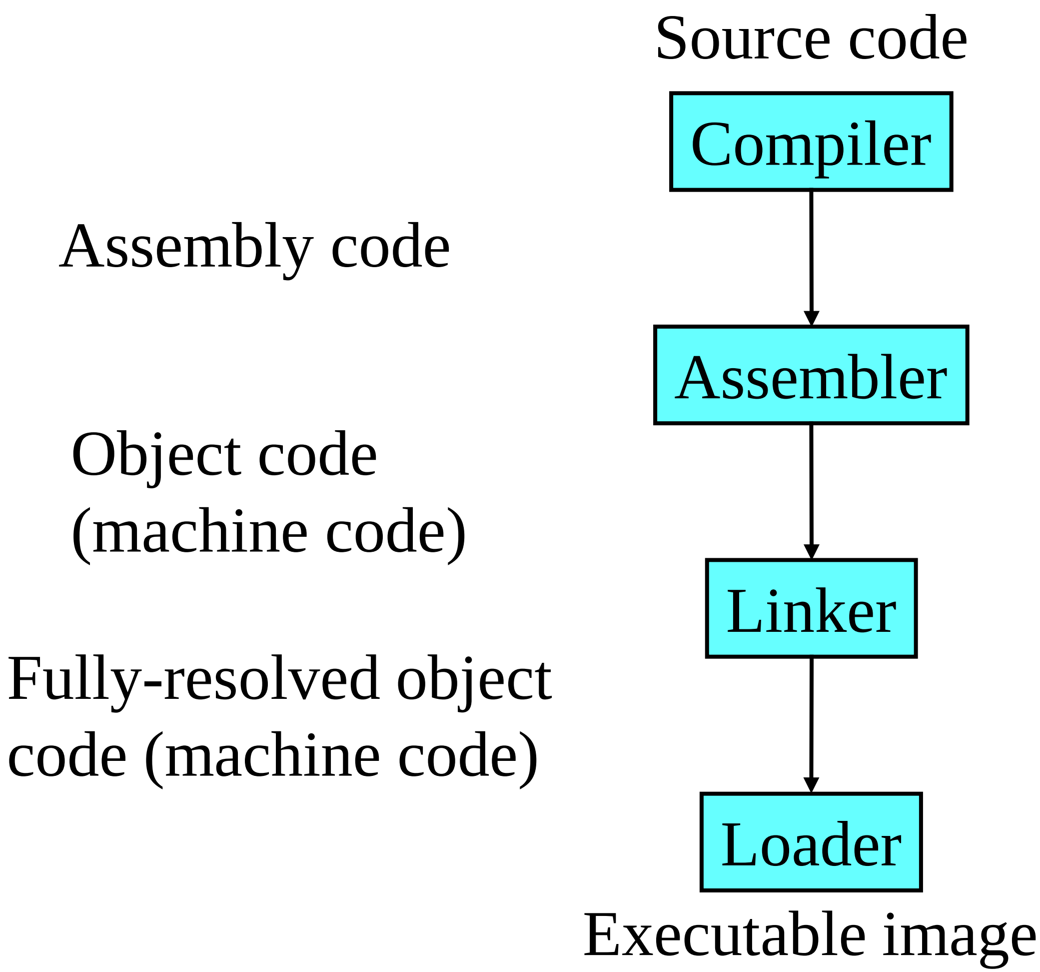
Typical Implementations

- Compilers
  - FORTRAN, C, C++, Java, COBOL, etc. etc.
  - Strong need for optimization, etc.
- Interpreters
  - PERL, Python, awk, sed, sh, csh, postscript printer, Java VM
  - Effective if interpreter overhead is low relative to execution cost of language statements

Hybrid approaches

- Well-known example: Java
  - Compile Java source to byte codes – Java Virtual Machine language (.class files)
  - Execution
    - ✓Interpret byte codes directly, or
    - ✓Compile some or all byte codes to native code
      - **(particularly for execution hot spots)**
      - **Just-In-Time compiler (JIT)**
- Variation: VS.NET
  - Compilers generate MSIL
  - All IL compiled to native code before execution

Compilers: The Big picture

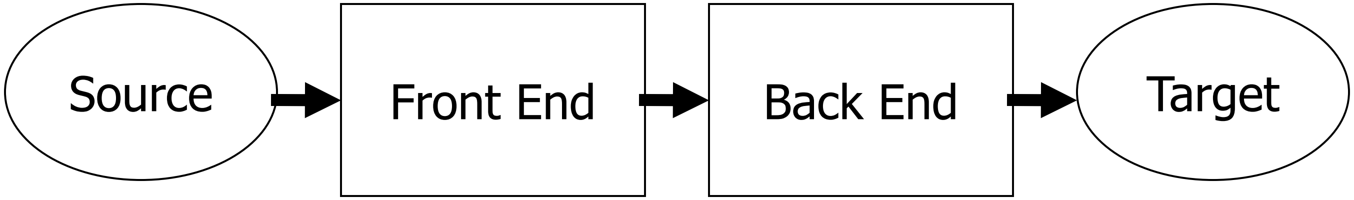


Idea: Translate in Steps

- Series of program representations
- Intermediate representations optimized for program manipulations of various kinds (checking, optimization)
- Become more machine-specific, less language-specific as translation proceeds

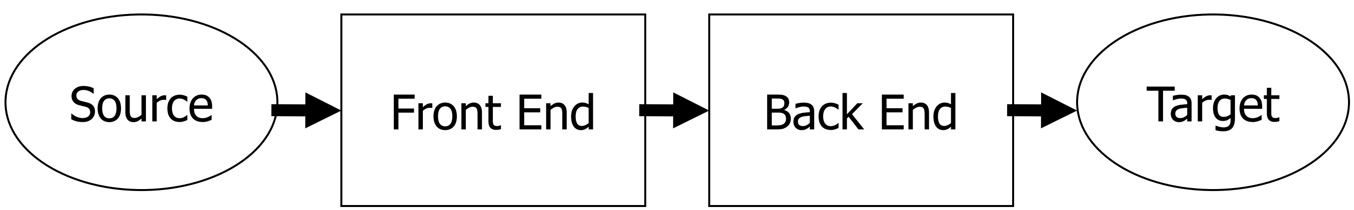
Structure of a Compiler

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



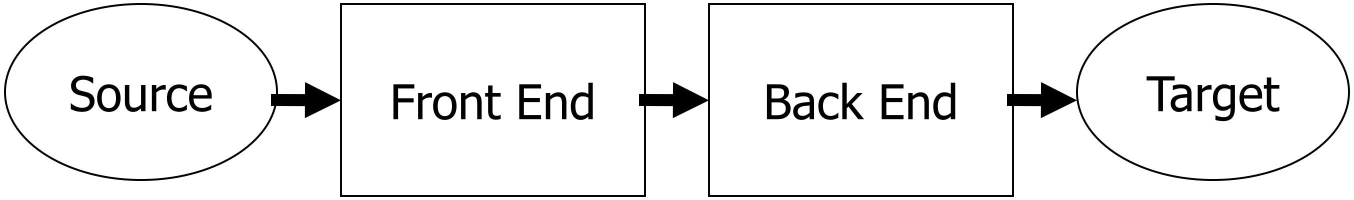
Implications

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

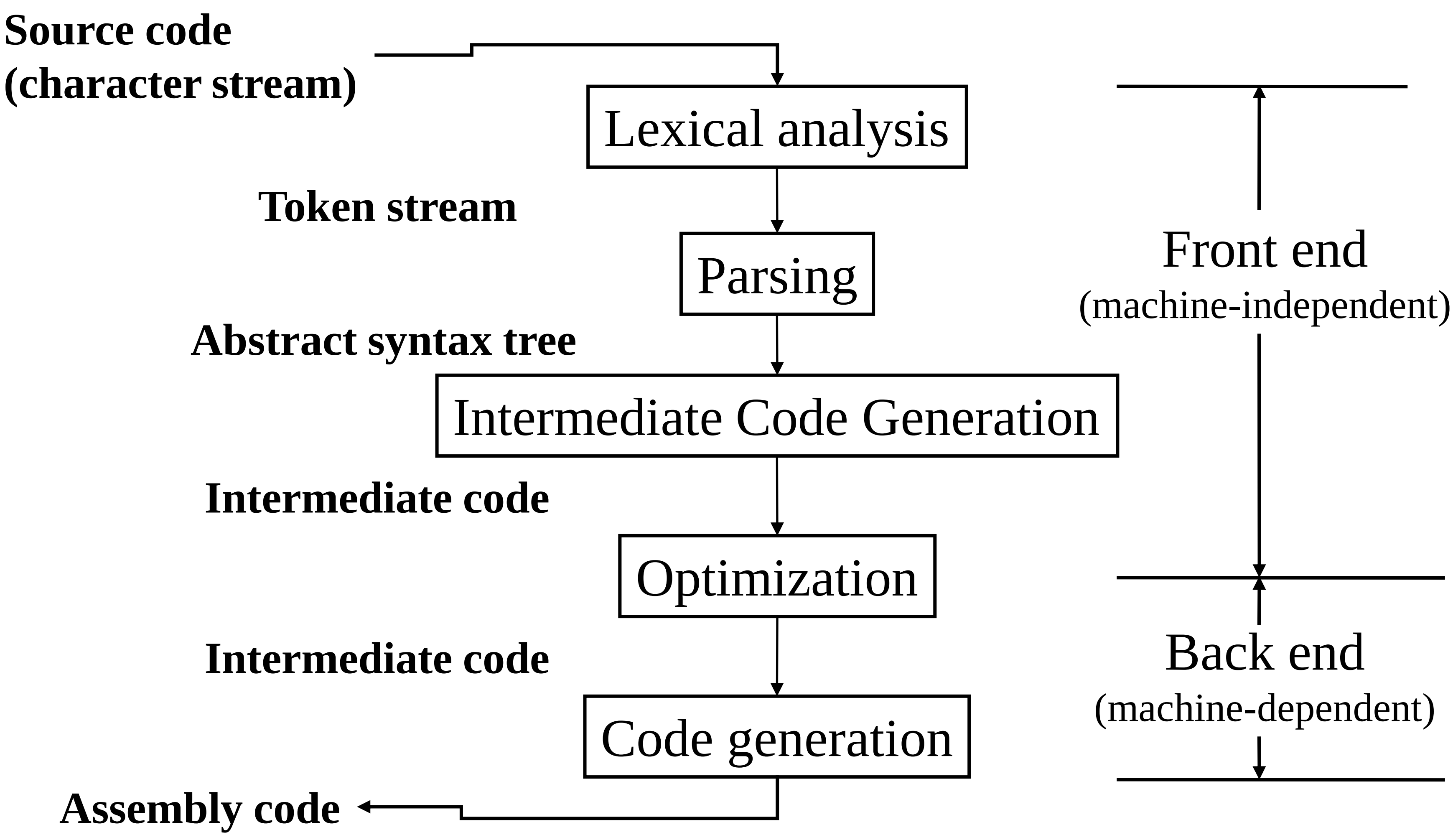


More Implications

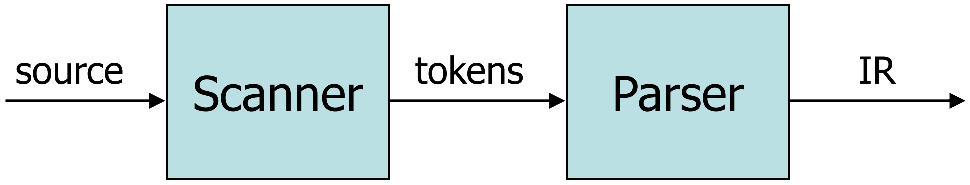
- Need some sort of Intermediate Representation (IR)
- Front end maps source into IR
- Back end maps IR to target machine code



Standard Compiler Structure



Front End



- Split into two parts
  - Scanner: Responsible for converting character stream to token stream
    - ✓ Also strips out white space, comments
  - Parser: Reads token stream; generates IR
- Both of these can be generated automatically
  - Source language specified by a formal grammar
  - Tools read the grammar and generate scanner & parser (either table-driven or hard coded)

Tokens

- Token stream: Each significant lexical chunk of the program is represented by a token
  - Operators & Punctuation: {}[]!+-=*;: …
  - Keywords: if while return goto
  - Identifiers: id & actual name
  - Constants: kind & value; int, floating-point character, string, …

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

  – Note: tokens are atomic items, not character strings

Parser Output (IR)

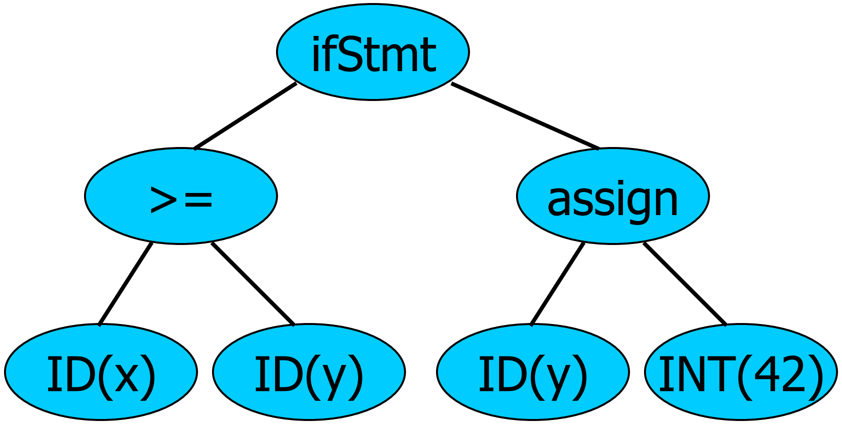
- Many different forms
  - (Engineering tradeoffs)
- Common output from a parser is an abstract syntax tree
  - Essential meaning of the program without the syntactic noise

Parser Example

- Token Stream Input

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

- Abstract Syntax Tree



Static Semantic Analysis

- During or (more common) after parsing
  - Type checking
  - Check for language requirements like “declare before use”, type compatibility
  - Preliminary resource allocation
  - Collect other information needed by back end analysis and code generation

Back End

- Responsibilities
  - Translate IR into target machine code
  - Should produce fast, compact code
  - Should use machine resources effectively
    - ✓Registers
    - ✓Instructions
    - ✓Memory hierarchy

Back End Structure

- Typically split into two major parts with sub phases
  - “Optimization” – code improvements
    - ✓May well translate parser IR into another IR
  - 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:
```

Example (Output assembly code)

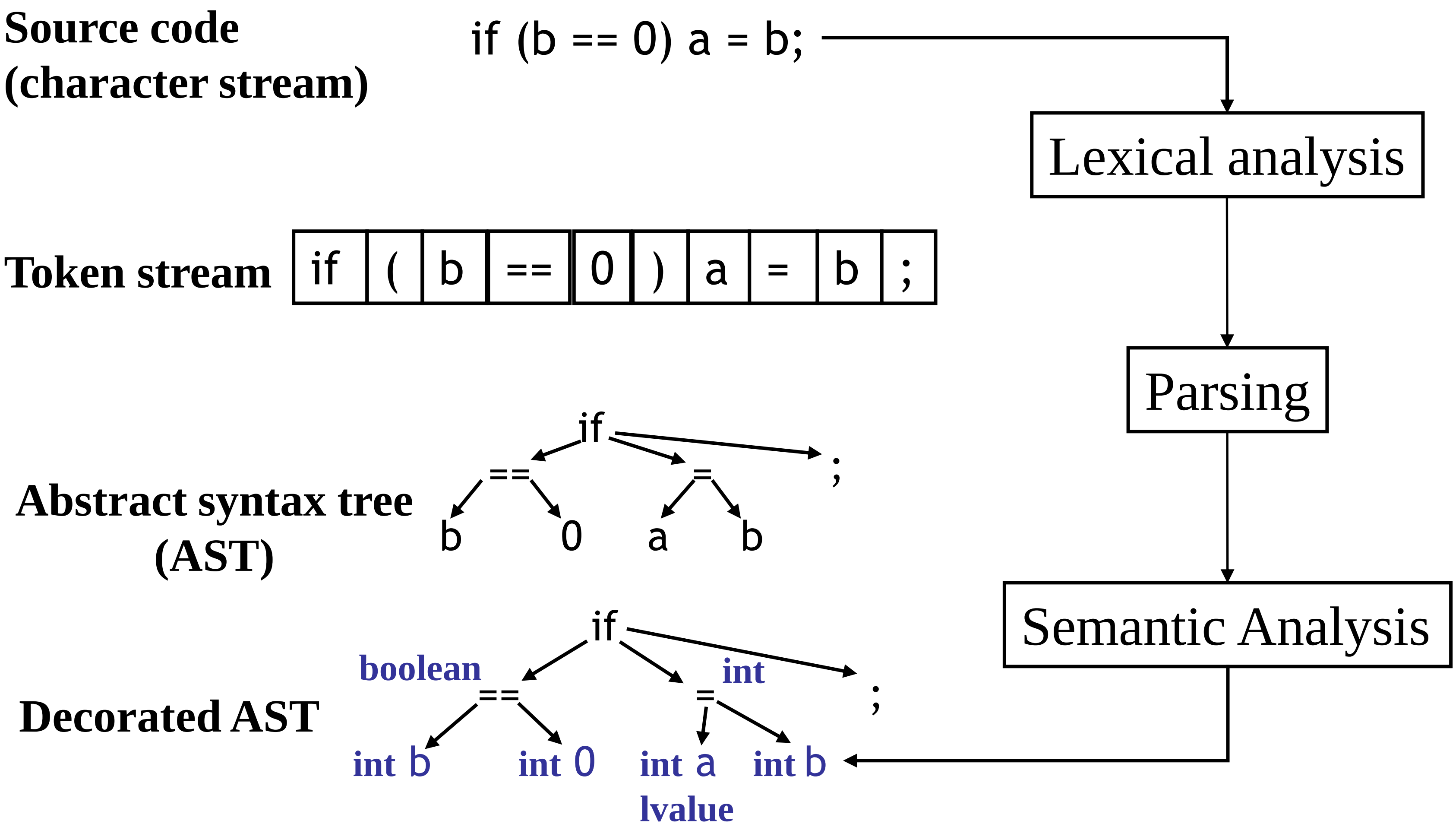
Unoptimized Code

```
    lda $30,-32($30)
    stq $26,0($30)
    stq $15,8($30)
    bis $30,$30,$15
    bis $16,$16,$1
    stl $1,16($15)
    lds $f1,16($15)
    sts $f1,24($15)
    ldl $5,24($15)
    bis $5,$5,$2
    s4addq $2,0,$3
    ldl $4,16($15)
    mull $4,$3,$2
    ldl $3,16($15)
    addq $3,1,$4
    mull $2,$4,$2
    ldl $3,16($15)
    addq $3,1,$4
    mull $2,$4,$2
    stl $2,20($15)
    ldl $0,20($15)
    br $31,$33
$33:
    bis $15,$15,$30
    ldq $26,0($30)
    ldq $15,8($30)
    addq $30,32,$30
    ret $31,($26),1
```

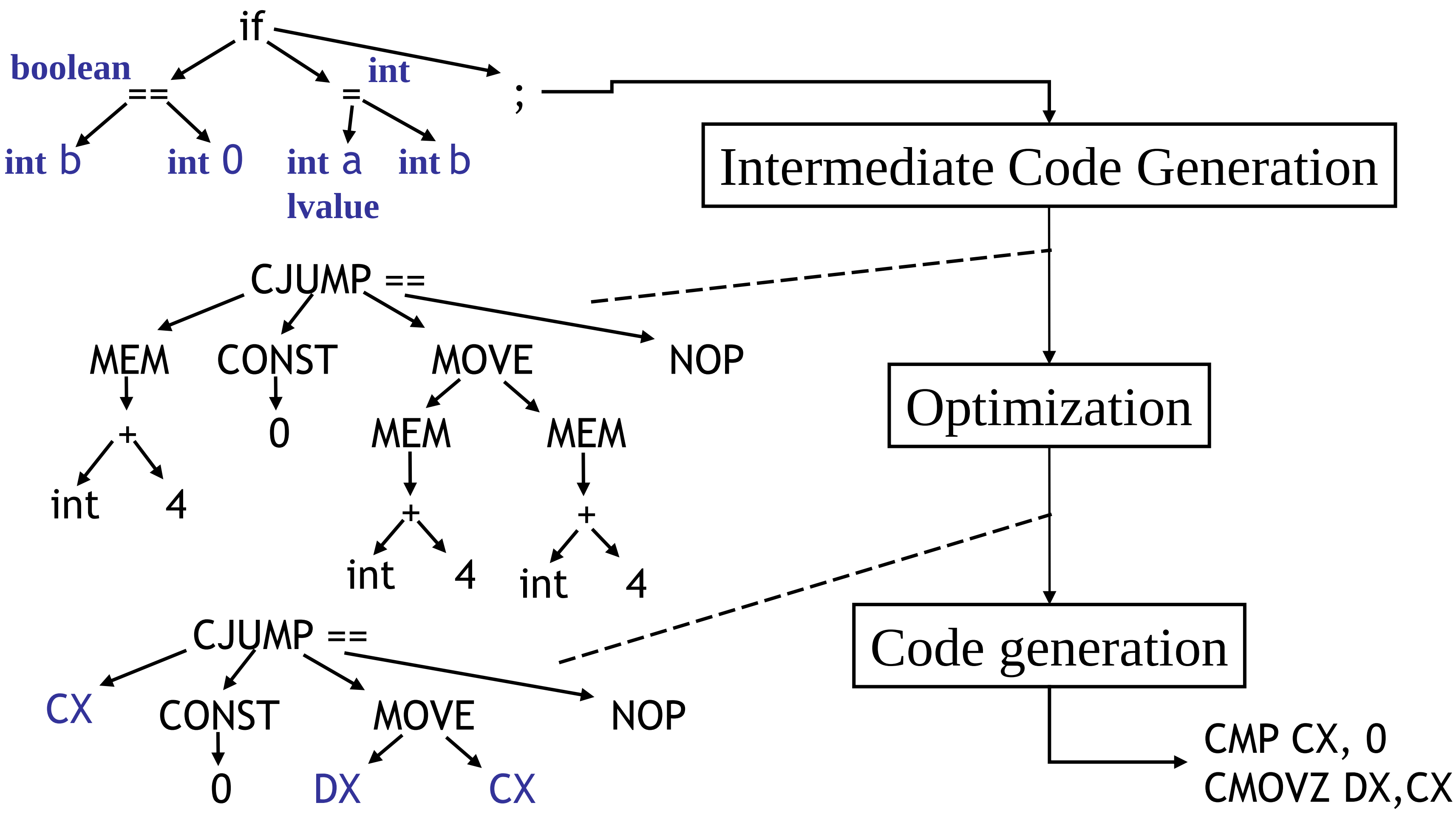
Optimized Code

```
s4addq $16,0,$0
mull $16,$0,$0
addq $16,1,$16
mull $0,$16,$0
mull $0,$16,$0
ret $31,($26),1
```

Compilation in a Nutshell 1



Compilation in a Nutshell 2



Compiler Design and Programming Language Design

- An interesting aspect is how programming language design and compiler design influence one another.

- Programming languages that are easy to compile have many advantages

41

Compiler Design and Programming Language Design(2)

- Languages such as Snobol and APL are usually considered non-compilable
- What attributes must be found in a programming language to allow compilation?
  - Can the scope and binding of each identifier reference be determined before execution begins?
  - Can the type of object be determined before execution begins?
  - Can existing program text be changed or added to during execution?

Computer Architecture and Compiler Design

- Compilers should exploit the hardware-specific feature and computing capability to optimize code.
- The problems encountered in modern computing platforms:
  - Instruction sets for some popular architectures are highly non-uniform.

43

Computer Architecture and Compiler Design

- High-level programming language operations are not always easy to support.
  - ✓Ex. exceptions, threads, dynamic heap access …
- Exploiting architectural features such as cache, distributed processors and memory
- Effective use of a large number of processors

Compiler Design Considerations

- Debugging Compilers
  - Designed to aid in the development and debugging of programs.
- Optimizing Compilers
  - Designed to produce efficient target code
- Re-targetable Compilers
  - A compiler whose target architecture can be changed without its machine-independent components having to be rewritten.

45

Why Study Compilers? (1)

- Compiler techniques are everywhere
  - Parsing (little languages, interpreters)
  - Database engines
  - AI: domain-specific languages
  - Text processing
    - ✓ Tex/LaTex -> dvi -> Postscript -> pdf
  - Hardware: VHDL; model-checking tools
  - Mathematics (Mathematica, Matlab)

Why Study Compilers? (2)

- Fascinating blend of theory and engineering
  - Direct 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 solutions
- The crucial part of our computer systems.
  - Security and performance rely on compilers.

Why Study Compilers? (3)

- 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 & hierarchy management, instruction set use

Why Study Compilers? (4)

- Renewed interest in compiler research
  - Today's systems are becoming heterogeneous and exascale.
    - We are using CPU, GPU, FPGA and ASICs.
    - We are forced to run with hundreds of thousands of processors.
  - Code generation, resource management, programmability, all need to be revisited
  - There is actual a requirement to rethink the way we design the whole computer system.

Class Summary

- Compilers: Introduction
  - Why Compilers?
  - Input and Output
  - Structure of Compilers
  - Compiler Design

Next Class

- Foundation of Compilers
  - Formal Languages
  - Grammars
  - Automatons