~~CS406~~

CS323: Compilers

Spring 2023

Week1: Overview, Structure of a compiler

Why Study Compilers?

Job Postings: 2022 LLVM Developers' Meeting (swoogo.com)

Company Contact: Mike Edwards - llvmjobs2022@modul

Job Title: AI Compiler Engineer

Very Very Exciting Jobs!

Job Title: Senior Apple GPU Compiler Backer

Job Description: As a member of the AGX compiler for current and future Apple GPUs room for growth that works on every Apple

Company Description: Arm's processors are shipped in billions of produ with unique code generation challenges. LLVM is a foundational code ge Learning accelerators. For example, in the past year, 75 Arm engineer performance optimization, security hardening, support for new instruction

Company Contact: Kristof Beyls - kristof.beyls@arm.com

Job Title: Many LLVM-related jobs at Arm

Job Description: Your skills and knowledge of compiler fundament contribute to the LLVM community will help us develop innovative techn security of the entire field of computing.

Arm always has lots of LLVM-related job vacancies open.

Company Description: MATLAB® and Simulink® are the pro languages using state-of-the-art compiler technologies such as stringent demands—for speed, memory, area, standards com customers to implement their ideas and enable them to deplo customers and products span domains including Deep Lear Generation.

Company Contact: Akshatha Bhat - akshathb@mathworks.com

Job Title: Compiler Engineer LLVM, Senior Software Engineer - JI

ink® are the programming enable our customers to im

Company Description: Founded in 1987, Huawei is a leading global pro technology infrastructure and smart devices. We invest heavily in f technological breakthroughs that drive the world forward. We have more t more than 170 countries and regions. Huawei's Heterogeneous Compiler L fastest growing teams in the field of compilation technology.

Company Contact: Shivani Bhardwaj - shivani.bhardwaj@huawei.com

Job Title: Junior Compiler Software Engineer

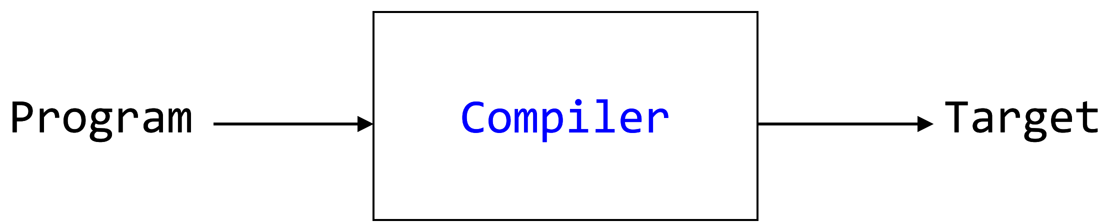
Job Description: Be a team player in a fast-paced R&D environment, wher - LLVM-based compilers targeting next-generation mobile, network, or ser

- Few disciplines with **deep theory + practice**

“..Theory and practice are two sides of the same coin..” - Jeff Ullman, ACM Turing Award lecture.

Intro to Compilers

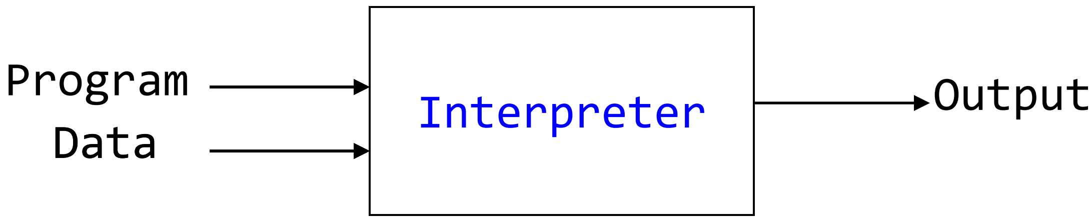
- One way to implement *programming languages*
  - Programming languages are notations for specifying computations to machines

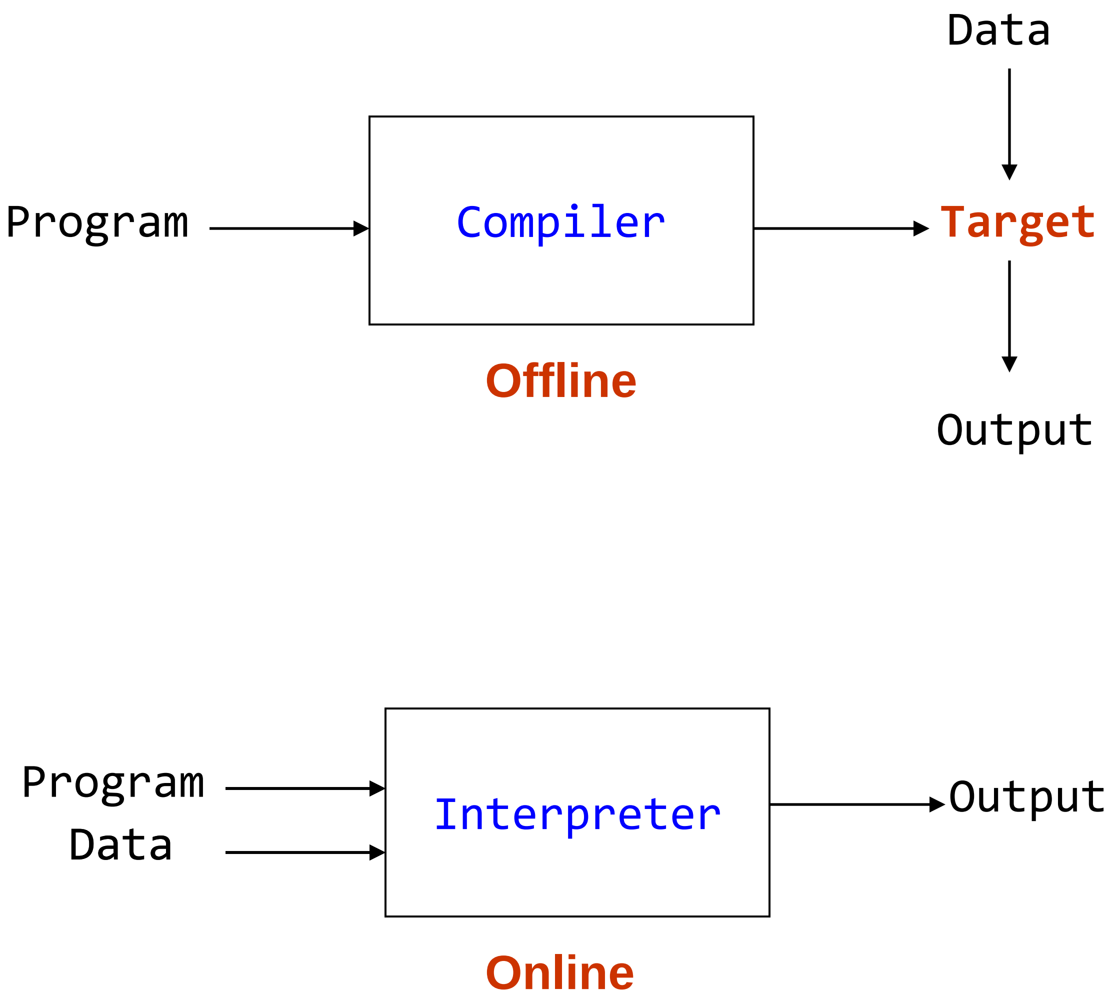


  - *Target* can be an assembly code, executable, another source program etc.

Intro to Compilers

- Alternate way to implement programming languages





these are the two types of language processing systems

What is a Compiler?

Traditionally: Program that analyzes and **translates** from a high-level language (e.g. C++) to low-level assembly language that can be executed by the hardware

```
int a, b;
a = 3;
if (a < 4) {
    b = 2;
} else {
    b = 3;
}
```

```
    var a
    var b
    mov 3 a
    mov 4 r1
    cmpi a r1
    jge l_e
    mov 2 b
    jmp l_d
l_e:mov 3 b
l_d:;done
```

Compilers are *translators*

- Fortran
- C
- C++
- Java
- Text processing language
- HTML/XML
- Command & Scripting Languages
- Natural Language
- Domain Specific Language

translate →

- Machine code
- Virtual machine code
- Transformed source code
- Augmented source code
- Low-level commands
- Semantic components
- Another language

Compilers are *optimizers*

- Can perform optimizations to make a program more efficient

```
int a, b, c;
b = a + 3;
```
~~c = a + 3;~~

```
c = b;
```

```
var a
var b
var c
mov a r1
addi 3 r1
mov r1 b
mov a r2
addi 3 r2
mov r2 c
```

```
var a
var b
var c
mov a r1
addi 3 r1
mov r1 b
mov r1 c
```

Why do we need compilers?

- Compilers provide *portability*
- Old days: whenever a new machine was built, programs had to be rewritten to support new instruction sets
- IBM System/360 (1964): Common Instruction Set Architecture (ISA) --- programs could be run on any machine which supported ISA
  - Common ISA is a huge deal (note continued existence of x86)
- But still a problem: when new ISA is introduced (EPIC) or new extensions added (x86-64), programs would have to be rewritten
- Compilers bridge this gap: write new compiler for an ISA, and then simply recompile programs!

Why do we need compilers?

- Compilers enable high-performance and productivity
- Old: programmers wrote in assembly language, architectures were simple (no pipelines, caches, etc.)
  - Close match between programs and machines --- easier to achieve performance
- New: programmers write in high level languages (Ruby, Python), architectures are complex (superscalar, out-of-order execution, multicore)
- Compilers are needed to bridge this *semantic gap*
  - Compilers let programmers write in high level languages and still get good performance on complex architectures

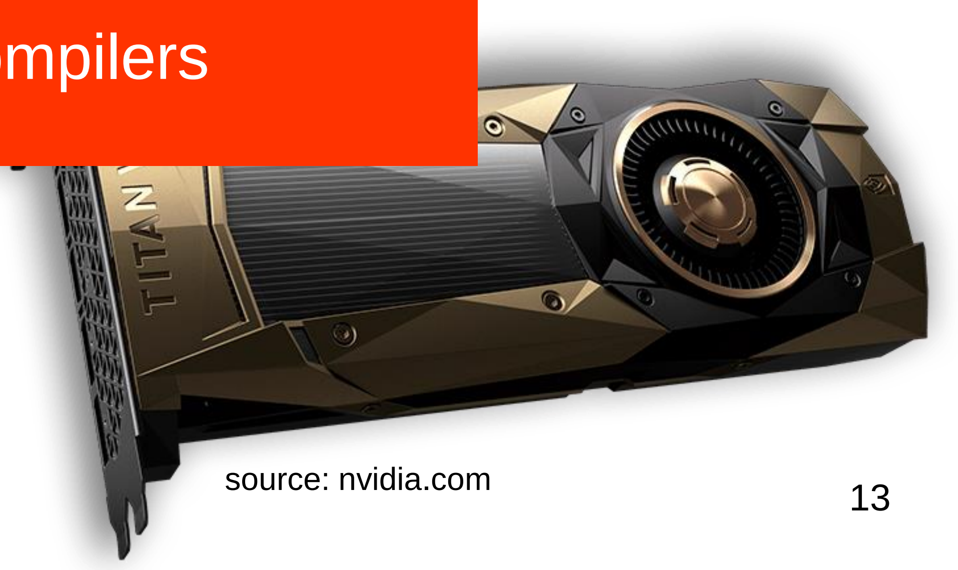
slide courtesy: Milind Kulkarni

Semantic Gap

- Python code that actually runs on GPU

```
import pycuda
import pycuda.autoinit from pycuda.tools import
make_default_context
c = make_default_context()
d = c.get_device()
……
```

Impossible without Compilers



source: nvidia.com

Example: Compilers as Translators

1. High level language $\Longrightarrow$ assembly language **(e.g. gcc)**
2. High level language $\Longrightarrow$ machine independent bytecode **(e.g. javac)**
3. Bytecode $\Longrightarrow$ native machine code **(e.g. java's JIT compiler)**
4. High level language $\Longrightarrow$ High level language
   (e.g. domain-specific languages, many research languages)

How would you categorize a compiler that handles SQL queries?

HLL to Assembly

Program — Compiler → Assembly — Assembler → Machine code

- Compiler converts program to assembly
- Assembler is machine-specific translator which converts assembly to machine code

```
add $7 $8 $9 ($7 = $8 + $9 ) => 000000 00111 01000 01001 00000 100000
```

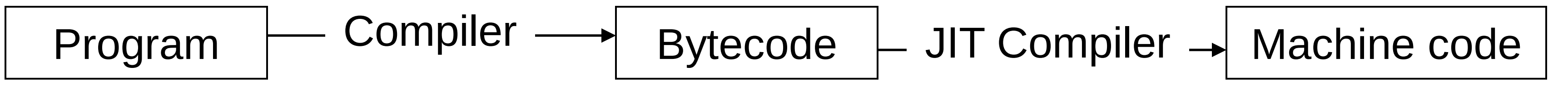
- Conversion is usually one-to-one with some exceptions
  - Program locations
  - Variable names

HLL to Bytecode

Program — Compiler → Bytecode — **Interpreter** → Execute!

- Compiler converts program into machine independent bytecode
  - e.g. javac generates Java bytecode, C# compiler generates CIL
- Interpreter then executes bytecode "on-the-fly"
- Bytecode instructions are "executed" by invoking methods of the interpreter, rather than directly executing on the machine
- Aside: what are the pros and cons of this approach?

HLL to Bytecode to Assembly



- Compiler converts program into machine independent bytecode
  - e.g. javac generates Java bytecode, C# compiler generates CIL
- Just-in-time compiler compiles code *while program executes* to produce machine code
  - Is this better or worse than a compiler which generates machine code directly from the program?

slide courtesy: Milind Kulkarni

History

- 1954: IBM 704
  – Huge success
  – Could do complex math

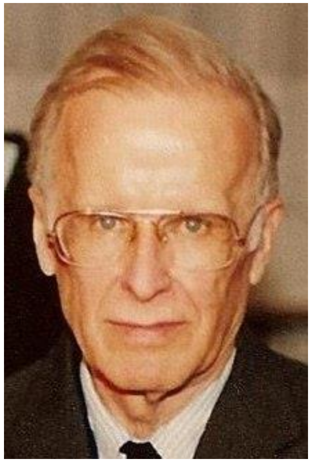
  – Software cost > Hardware cost

Source: IBM Italy,
https://commons.wikimedia.org/w/index.php?curid=48929471

How can we improve the efficiency of creating software?

History

- 1953: Speedcoding
  - *High-level programming language* by John Backus
  - Early form of *interpreters*
  - Greatly reduced programming effort



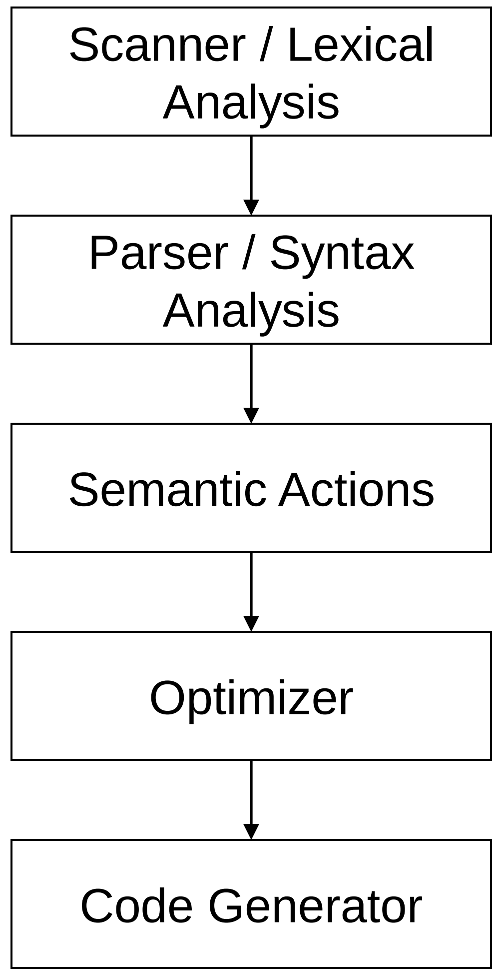
  - About 10x-20x slower
  - Consumed lot of memory (~300 bytes = about 30% RAM)

Fortran I

- 1957: Fortran released
  - Building the compiler took 3 years
  - Very successful: by 1958, 50% of all software created was written in Fortran
- Influenced the design of:
  - high-level programming languages e.g. BASIC
  - practical compilers

Today's compilers still preserve the structure of Fortran I

Structure of a Compiler



Scanner

- Analogy: Humans processing English text

```
Rama is a neighbor.
```

```
Ra mais an eigh bor.
```

You have to do some work to align the spaces and understand the sentence.

Scanner

- Consider the program text

```
if ( a < 4) {
    b = 5
}
```

  - Has *tokens* that are:
    1. keywords – `if`
    2. Punctuation marks – `(, ), {, }, blankspaces, tab space (\t), newlines (\n)`
    3. Identifiers – `a, b`
    4. Constants/Literals – `4, 5`
    5. Operators - `<, =`

Scanner

- A compiler starts by seeing only program text

```
if ( a < 4) {
    b = 5
}
```

- as a series of letters

```
'i' 'f' ' ' '(' 'a' '<' '4' ')'
' ' '{' '\n' '\t' 'b' '=' '5'
'\n' '}'
```

Scanner

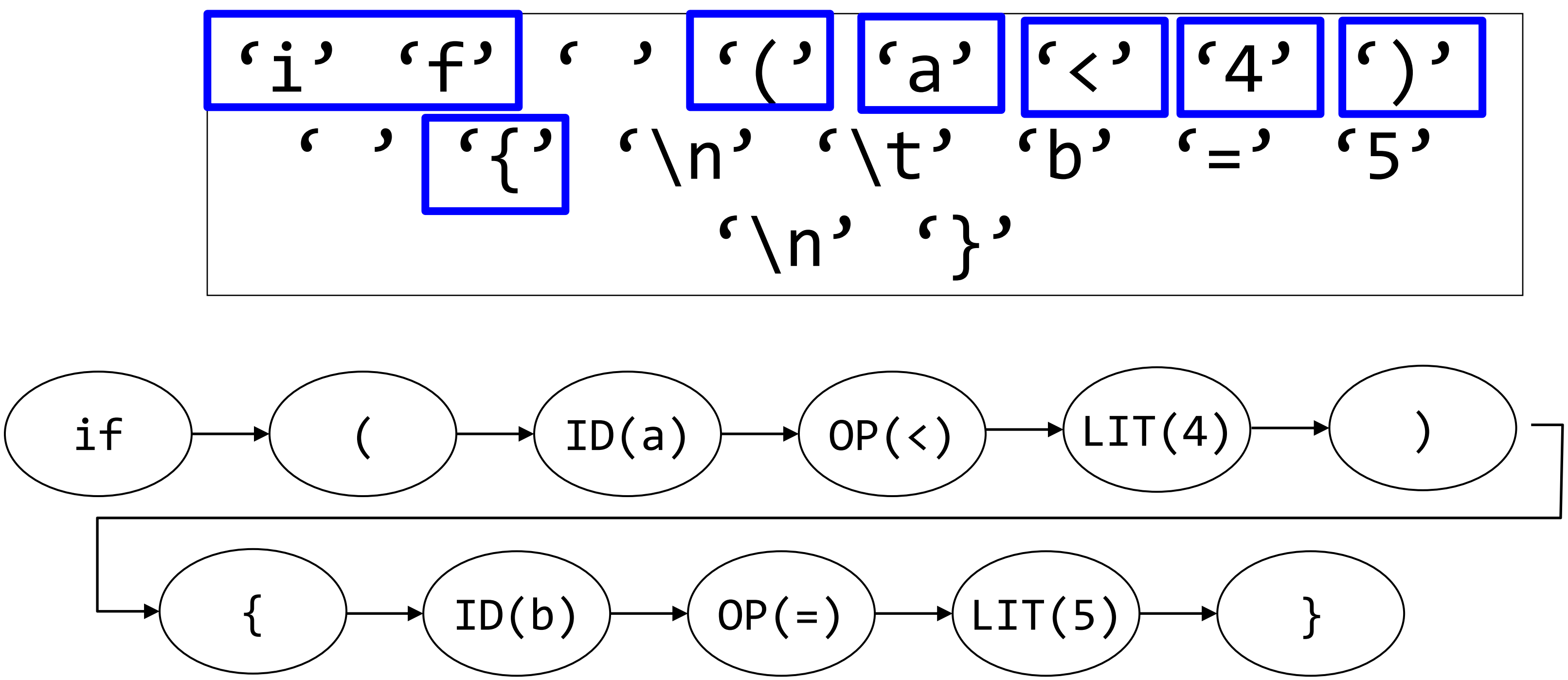
- A compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*

```
‘i’ ‘f’ ‘ ’ ‘(’ ‘a’ ‘<’ ‘4’ ‘)’
 ‘ ’ ‘{’ ‘\n’ ‘\t’ ‘b’ ‘=’ ‘5’
          ‘\n’ ‘}’
```

- Analogy: Humans processing English text
  - recognize words in `Rama is a neighbor.`
    - Rama, is, a, neighbor
    - Additional details such as punctuations(.), capitalizations (R), blank spaces.

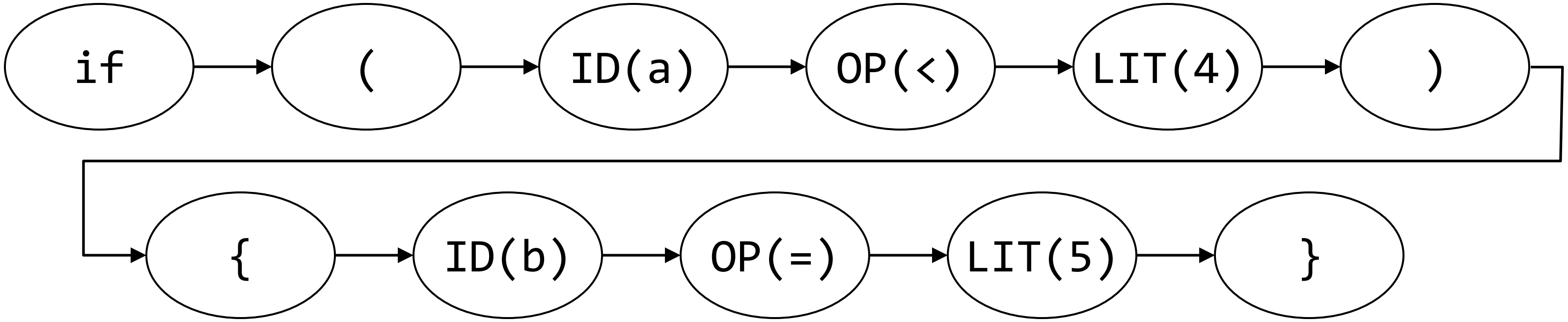
Scanner

- A compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*



Scanner - Summary

- A compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*



- But we still don't know what the *syntactic structure* of the program is

slide courtesy: Milind Kulkarni

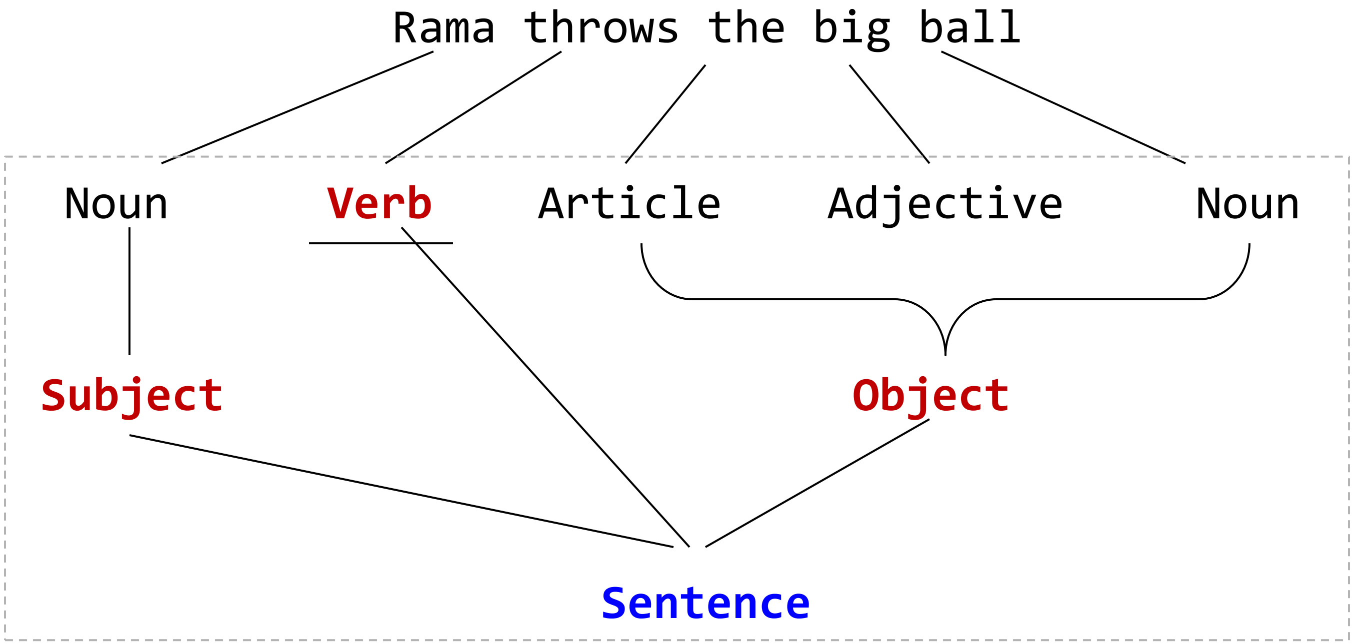
Exercise

Convert the following program text into tokens:

```
c = a + b * 60
```

Parser - Analogy

- Diagramming English sentences

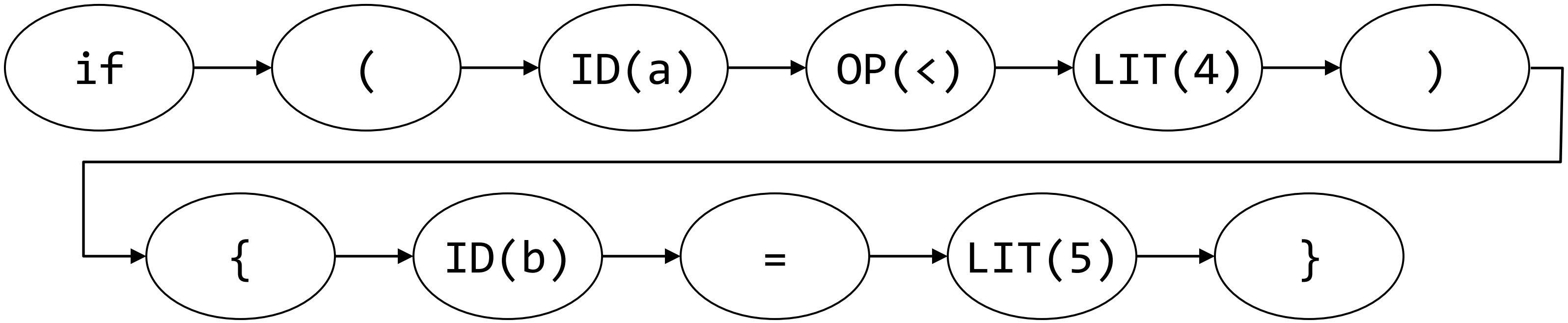


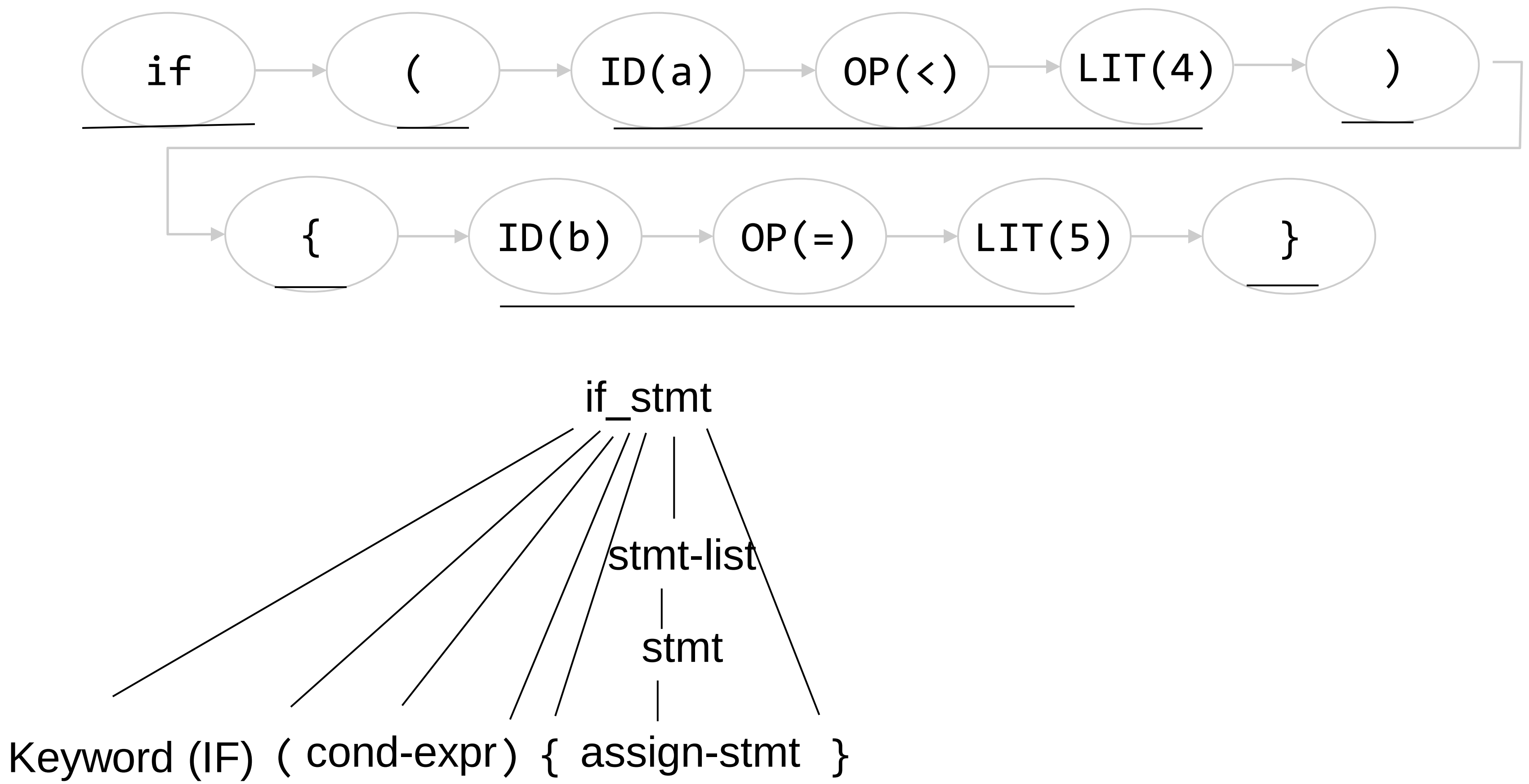
Tree structure (inverted)

Group lower-level language elements to higher-level language elements

Parser

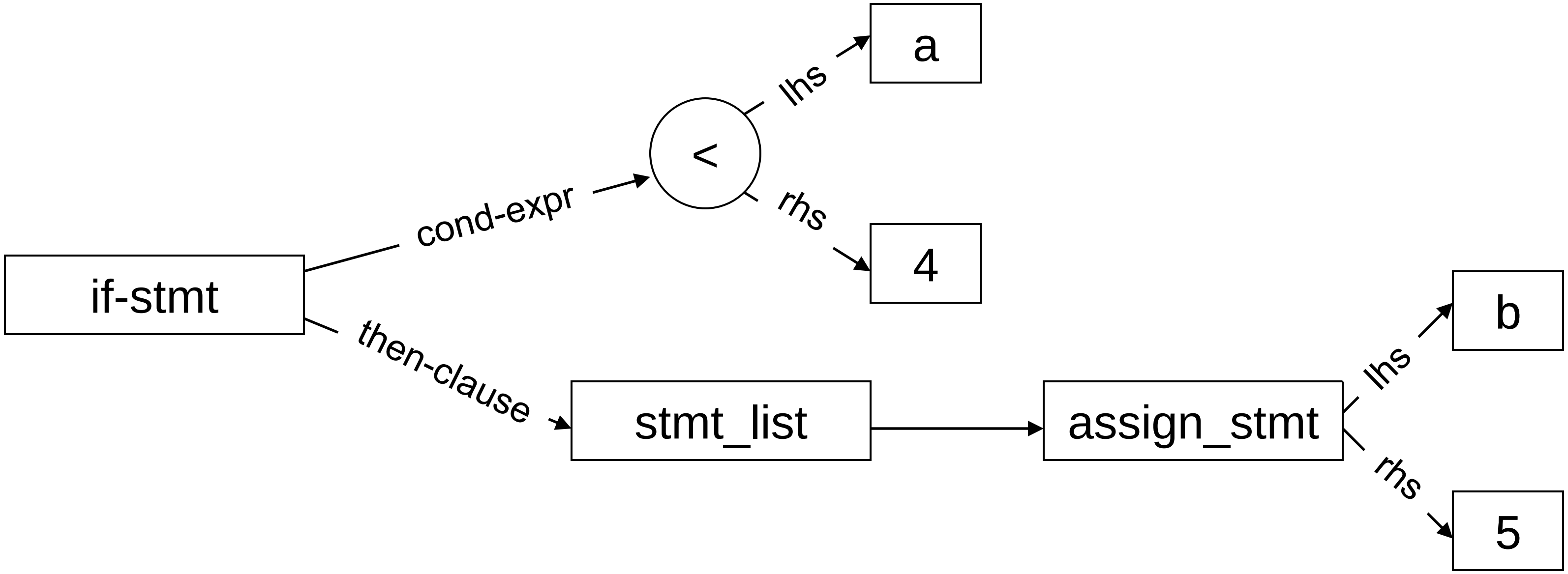
- Converts a string of tokens into *parse tree* or *abstract syntax tree*
- Captures syntactic structure of the code (i.e. “this is an `if` statement, with a `then`-block”




if
(
ID(a)
OP(<)
LIT(4)
)
{
ID(b)
OP(=)
LIT(5)
}
if_stmt
stmt-list
stmt
Keyword (IF)
(
cond-expr
)
{
assign-stmt
}

Parser

- Converts a string of tokens into *parse tree* or *abstract syntax tree*
- Captures syntactic structure of the code (i.e. “this is an `if` statement, with a `then`-block”)

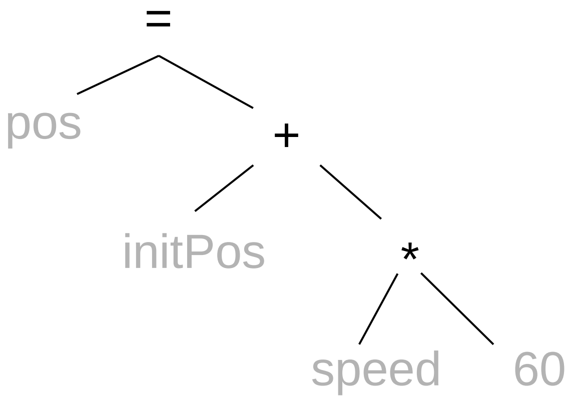


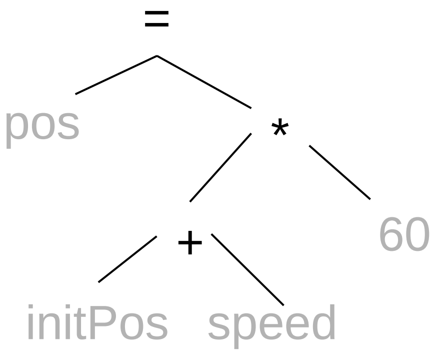
slide courtesy: Milind Kulkarni

Exercise

What is the correct abstract syntax tree for the following program statement?

```
pos = initPos + speed * 60
```





Semantic Actions

- Interpret the *semantics* of syntactic constructs
- Refer to actions taken by the compiler based on the *semantics* of program statements.
- Up until now, we have looked at syntax of a program
  - *what is the difference?*

Syntax vs. Semantics

- Syntax: "grammatical" structure of language
  – What symbols, in what order, is a legal part of the language?
    - But something that is syntactically correct may mean nothing!
      – "Colorless green ideas sleep furiously."

- Semantics: meaning of language
  – What does a particular set of symbols, in a particular order *mean?*
    - What does it mean to be an if statement?
      – "evaluate the conditional, if the conditional is true, execute the then clause, otherwise execute the else clause"

Semantic Actions – What is done?

- What actions are taken by compiler based on the semantics of program statements ?
  - Examples (analogy):

```
Ram said Ram has a big heart.
```

```
Are they referring to the same person Ram?
```

Programming languages have rules to resolve ambiguities like above: **bind variables to their scopes**:

```
int Ram = 1;
//some code here
{
    int Ram = 2;
    ..
}
//some code here
```

Semantic Actions – What is done?

- What actions are taken by compiler based on the semantics of program statements ?
  - Examples:

```
Ram left her home in the evening

        Usual naming conventions indicate that
        there is a "type mismatch" between 'Ram'
        and 'her': they refer to different types.
```

  - Programming languages have rules to enforce types
  - Check for type inconsistencies

Semantic Actions – How is it done?

- What actions are taken by compiler based on the semantics of program statements ?
  - Building a *symbol table*
  - Generating *intermediate representations*

Symbol Tables

- A list of every declaration in the program, along with other information
  1. Variable declarations: types, scope
  2. Function declarations: return types, # and type of arguments

Example Program

```
Integer ii;
…
ii = 3.5;
…
print ii;
```

…

Symbol Table

Name	**Type**	**Scope**
ii	int	global

Intermediate Representation

- Also called *IR*
- A (relatively) low level representation of the program
  - But not machine-specific!
- One example: *three address code*

```
        bge a, 4, done
        mov 5, b
done: //done!
```

```
if ( a < 4) {
    b = 5
}
```

  – Each instruction can take at most three operands (variables, literals, or labels)
  – Note: no registers!

slide courtesy: Milind Kulkarni

Exercise

Explain the semantics of the following program stmt:

```
pos = initPos + speed * 60
```

A Note on Semantics

- How do you define semantics?
  - **Static semantics:** properties of programs
    - All variables must have type
    - Expressions must use consistent types
    - Can define using *attribute grammars*
  - **Execution semantics:** how does a program execute?
    - Defined through *operational* or *denotational* semantics
    - Beyond the scope of this course!
  - For many languages, “the compiler is the specification”

Optimizer

- Transforms code to make it more efficient
- Different kinds, operating at different levels
  - High-level optimizations
    - Loop interchange, parallelization
    - Operates at level of AST, or even source code
  - Scalar optimizations
    - Dead code elimination, common sub-expression elimination
    - Operates on IR
  - Local optimizations
    - Strength reduction, constant folding
    - Operates on small sequences of instructions

slide courtesy: Milind Kulkarni

Optimizer

Reducing word usage (Analogy):

Dejavu

Sunny felt a sense of ~~having experienced it before~~ when his bike started making a hissing sound

Exercise: *is this optimization correct?*

`X = Y * 0` is the same as `X = 0`

Code Generation

- Generate assembly from intermediate representation
  - Select which instruction to use
  - Select which register to use
  - Schedule instructions

```
if ( a < 4) {
    b = 5
}
```

```
bge a, 4 done
mov 5, b
done: //done
```

```
ld  a, r1
mov 4, r2
cmp r1, r2
bge done
mov 5, r3
st r3, b
done:
```

slide courtesy: Milind Kulkarni

Code Generation

- Generate assembly from intermediate representation
  - Select which instruction to use
  - Select which register to use
  - Schedule instructions

```
bge a, 4 done
mov 5, b
done: //done
```

```
if ( a < 4) {
    b = 5
}
```

Previous slide

`mov 4, r1`	`ld  a, r1`
`ld  a, r2`	`mov 4, r2`
`cmp r1, r2`	`cmp r1, r2`
`blt done`	`bge done`
`mov 5, r1`	`mov 5, r3`
`st r1, b`	`st r3, b`
`done:`	`done:`

slide courtesy: Milind Kulkarni

Structure of a Compiler

Source code

Scanner / Lexical Analysis

Use *regular expressions* to define tokens. Can then use scanner generators such as lex or flex.

Tokens

Parser / Syntax Analysis

Define language using *context free grammars.* Can then use parser generators such as yacc or bison.

Syntax Tree

Semantic Actions

Semantic routines done by hand. But can be formalized.

IR

Optimizer

Written manually. Automation is an active research area (e.g. dataflow analysis frameworks)

IR

Code Generator

Written manually.

Executable

slide courtesy: Milind Kulkarni

Structure of a Compiler

Source code

Scanner / Lexical Analysis — Use *regular expressions* to … ens. Can then use scanner gen… or flex.

Tokens

Parser / Syntax Analysis — D… *…text free grammars.* Can then … erators such as yacc or bison.

Syntax T…

Semantic A… — …antic routines done by hand. But can be formalized.

IR

Optimizer — Written manually. Automation is an active research area (e.g. dataflow analysis frameworks)

IR

Code Generator — Written manually.

Executable

Many of these passes can be combined

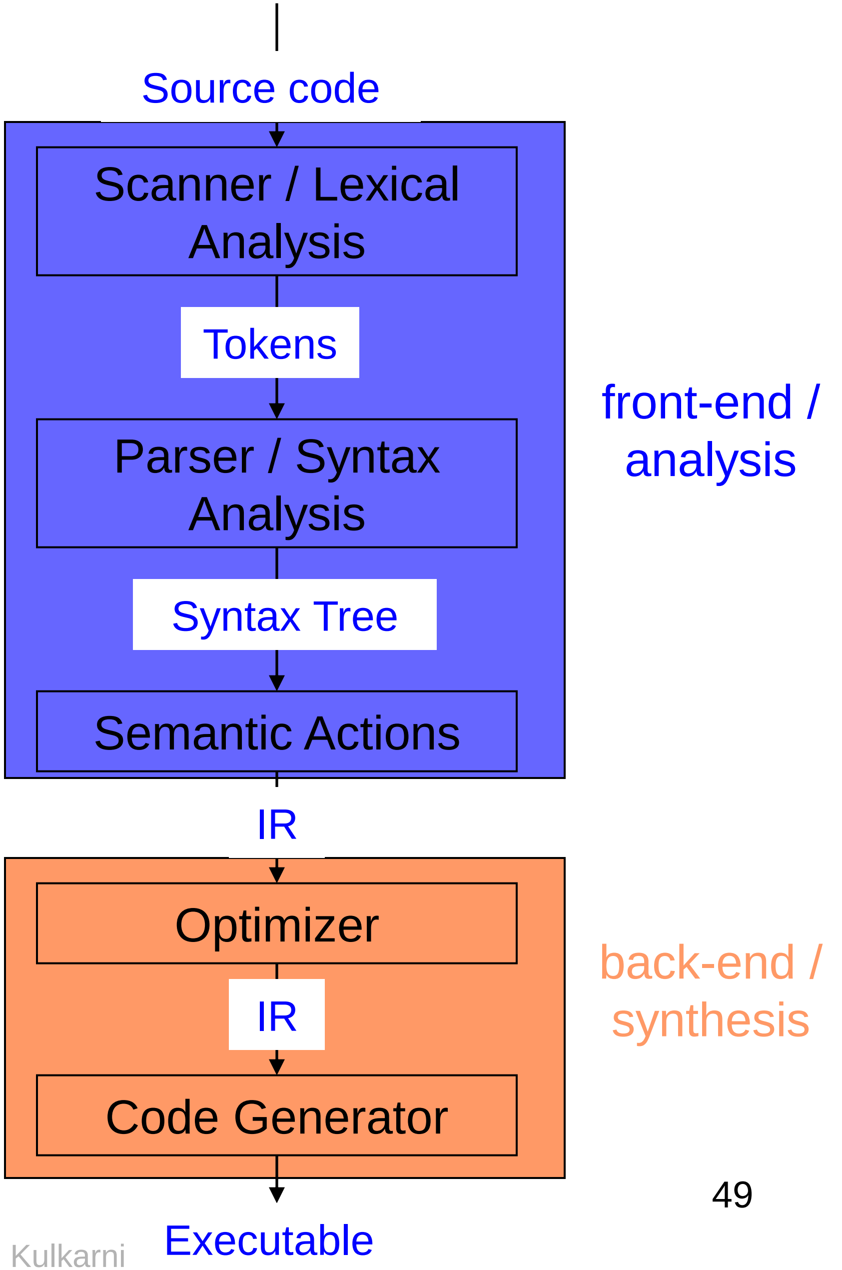
Front-end vs. Back-end

• Scanner + Parser + Semantic actions + (high level) optimizations called the *front-end* of a compiler

• IR level optimizations and code generation (instruction selection, scheduling, register allocation) called the *back-end* of a compiler

• Can build multiple front-ends for a particular back-end
 - •e.g. gcc or g++ or many front-ends which generate common intermediate language (CIL)

• Can build multiple back-ends for a particular front-end
 - •gcc allows targeting different architectures



slide courtesy: Milind Kulkarni

Design Considerations

- Compiler and programming language designs influence each other
  - Higher level languages are harder to compile
    - More work to bridge the gap between language and assembly
  - Flexible languages are often harder to compile
    - Dynamic typing (Ruby, Python) makes a language very flexible, but it is hard for a compiler to catch errors (in fact, many simply won't)
  - Influenced by architectures
    - RISC vs. CISC

Programming Languages and Real-world Usage

- Why are there so many programming languages?
- Why are there new languages?
- What is a good programming language?

Programming Languages and Real-world Usage

- Why are there so many programming languages?
  - Distinct often conflicting requirements of the application domain

Scientific Computing	Floating-Point Arithmetic, Parallelism Support, Array Manipulation	FORTRAN
Business Applications	No data loss (persistence), Reporting capabilities, Data analysis tools	SQL
Systems Programming	Fine-grained control of system resources, real-time constraints	C/C++

Programming Languages and Real-world Usage

- Why are there new languages?
  - To fill a technology gap
    - E.g. arrival of Web and Java
    - Java's design closely resembled that of C++

Training a programmer on a new programming language is a dominant cost

- Widely-used languages are slow to change
- Easy to start a new language

Programming Languages and Real-world Usage

- What is a good Programming Language?

  No universally accepted argument

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 1 (Sections: 1.1 to 1.3, 1.5)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 1 (Sections 1.1 to 1.3, 1.5)