CS406: Compilers

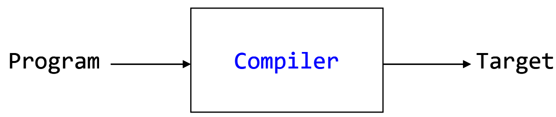
Spring 2020

Week1: Overview, Structure of a compiler

1

Intro to Compilers

- 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.

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
```


slide courtesy: Milind Kulkarni

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

4

slide courtesy: Milind Kulkarni

Compilers are *optimizers*

- Can perform optimizations to make a program more efficient

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

⇨

```
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
```


slide courtesy: Milind Kulkarni

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!

slide courtesy: Milind Kulkarni

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

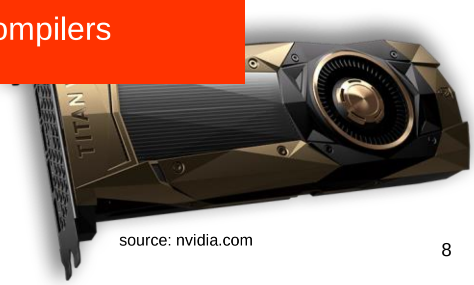
slide courtesy: Milind Kulkarni

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

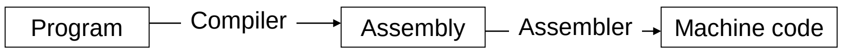
Some common compiler types

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

How would you categorize a compiler that handles SQL queries?

slide courtesy: Milind Kulkarni

HLL to Assembly

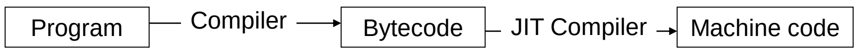


- Compiler converts program to assembly
- Assembler is machine-specific translator which converts assembly to machine code

```
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

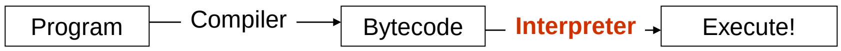
HLL to Bytecode to Assembly



- 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

HLL to Bytecode

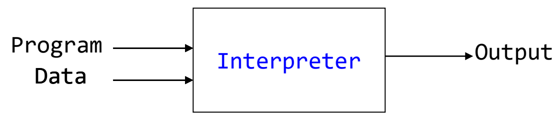


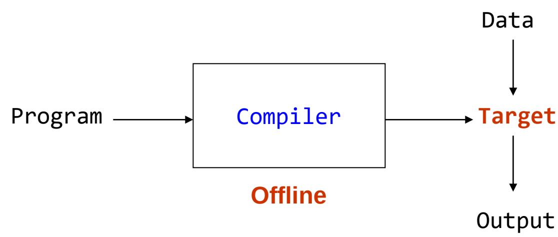
- Compiler converts program into machine independent bytecode
  - e.g. javac generates Java bytecode, C# compiler generates CIL
- Interpreter then executes bytecode “on-the-fly”
- Bytecode instructions are “executed” by invoking methods of the interpreter, rather than directly executing on the machine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

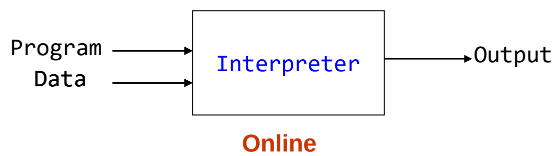
slide courtesy: Milind Kulkarni

Quick Detour: Interpreters

- Alternate way to implement programming languages







these are the two types of language processing systems

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?

15

Taking a peek at the history of how compilers and interpreters came into existence: In 1954, IBM came up with a hugely successful commercially available machine. 704 was the first mass produced machine with floating point hardware. As people started buying this machine and using it, they found that the software cost greatly exceeded the hardware cost (coding was done using assembly language then). Back then, hardware costed a Bomb and software was beating the hardware cost! So, a natural question was how to make software development more productive?

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)

16

The earliest effort in improving the productivity of developing software was called Speedcoding, developed by John Backus in 1953. Speedcode was the name of the programming language and it was a high-level programming language. The language provided pseudo-instructions for computing mathematical functions such as sine, logs etc. A resident software analyzed these instructions and called corresponding subroutines. So, this was an example of what we know of interpreters today.

This scheme aimed at ease-of-use at the expense of consuming system resources. For example, the interpreter consumed roughly 300 bytes, which was 30% of the memory of 704. As a result, the programs ran 10-20 times slower than handwritten programs.

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

17

Speedcoding was not popular but John backus thought it was promising and it gave rise to another project. Those days, the most important applications were weather prediction, finite element analysis, computational chemistry, computational physics, computational fluid dynamics etc. Programmers wrote formulas in a way that machines could understand and execute.

The problem with speedcoding was that the formulas were interpreted and hence, led to slower programs. John Backus thought that if the formulas were translated into a form that the machine can understand and execute, then it would solve the problem. So, the formula translation or Fortran 1 project started. Fortran 1 ran from 54 to 57 and ended up taking 3 years as against 1 year that they had predicted initially. So, people were not good at predicting how long complex software development would take then. People are not good now either. Some of you who read the “No Silver Bullets” paper would agree.

It was such a success that by 1958, 50% of all software developed was implemented in Fortran.

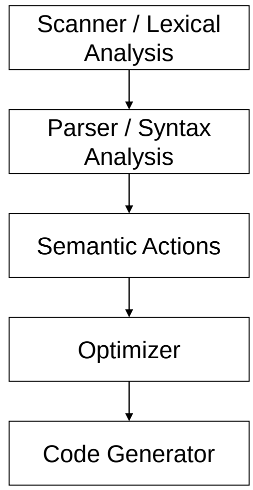
So, everybody thought that 1) Fortran raised the level of abstraction 2) Made it easier to use the machine 704.

Fortran 1 had a huge impact on computer science. This was the first compiler. It led to an enormous body of theoretical work. One of the attractive things about studying programming languages is that it involves a good mix of both theoretical and practical subjects. You need to have a good grasp of theory as well as good system building skills or engineering skills.

Fortran 1, also led to high-level programming languages such as BASIC. It influenced compiler design in such a fundamental way that today's compilers still preserve the structure of Fortran 1.

So, what is the structure of Fortran 1 compiler?

Structure of a Compiler



18

Fortran 1 compiler has 5 phases. Lexical analysis and Parsing take care of syntactic aspects. Semantic aspects takes care of things like types (can I assign a float to an int?), scope rules (what happens when we encounter a variable that is not defined yet?) Optimization phase deals with transforming the program into an equivalent one but that runs faster or uses less memory. Finally, the code generation phase deals with translating the program into another language. That another language might be machine code, bytecode, or another high-level programming language.

Scanner

- Analogy: Humans processing English text
  `Rama is a neighbor vs. Ra mais an eigh bor.`
- A compiler starts by seeing only program text

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

19

For some of the phases, we can have an analogy to how humans understand natural language.

The first step in understanding a program for a compiler or English language by a human is to recognize words (smallest unit above letters). Take the example English sentence: “Rama is a neighbor”. We immediately recognize the sentence as a group of 4 words. In addition, there are word separators (blank spaces), punctuations (full-stop), and special notations (capital letter R). Now, if you were given the other sentence, it doesn’t come to you immediately what that sentence is saying. You have to work a bit to align the spaces and understand.

The goal of lexical analysis or scanning is to recognize program text into ‘words’ or ‘tokens’ as it is called in compiler terminology. Take the example code snippet.

Scanner

- A compiler starts by seeing only program text

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


The lexical analyzer starts by seeing program text as a series of letters.

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’ ‘}’
```

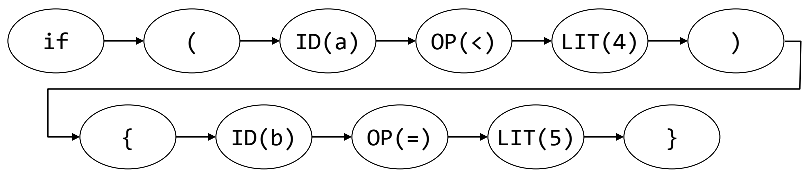
- Analogy: Humans processing English text
  - recognize words in `Rama is a neighbor.`
    - Rama, is, a, neighbor
    - Additional details such as punctuations(.), capitalizations (R), blank spaces.

21

The lexical analyzer then converts the program text into tokens (that the small-letter ‘i’ followed by small-letter ‘f’ is a token ‘if’, that the blankspace is a token, that ‘(‘ is a token, and so on.) Just as in English text, we had punctuations, we have ‘\n’s ‘\t’s and ‘ ‘s. We have operators ‘<‘ and ‘=‘. We have constant ‘4’. We have variables ‘a’, and ‘b’. We have keywords ‘if’. We have more punctuations ‘(‘, ‘)’, ‘{‘,’}’

Scanner

- 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

22

slide courtesy: Milind Kulkarni

So, the lexical analyzer produced a sequence, or a list of tokens as shown. We still do not know whether there is some structure to that sequence. If there is some structure, what is that structure?

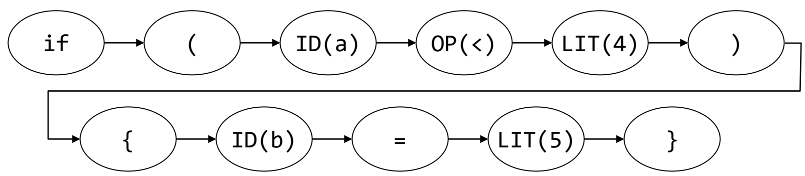
Exercise

Convert the following program text into tokens:

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

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”)



- Analogy: understand the English sentence structure

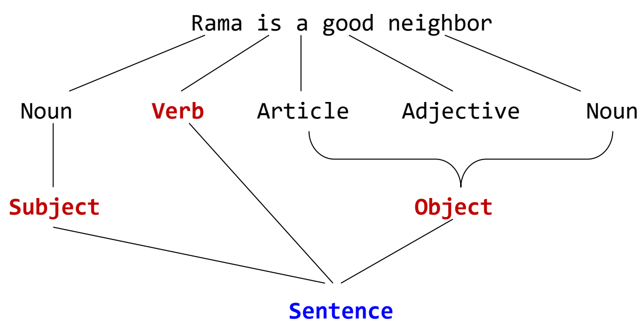
  `Rama is a good neighbor`

To recognize the structure, a parser or syntactic analyzer comes into picture. Our goal is to tell that the sequence of tokens is an if statement with a then block.

Coming back to the analogy of how humans recognize structure, we have the diagramming English sentences procedure. It is a simple procedure of drawing a tree structure and identifying elements within the structure.

Parser - Analogy

- Diagramming English sentences

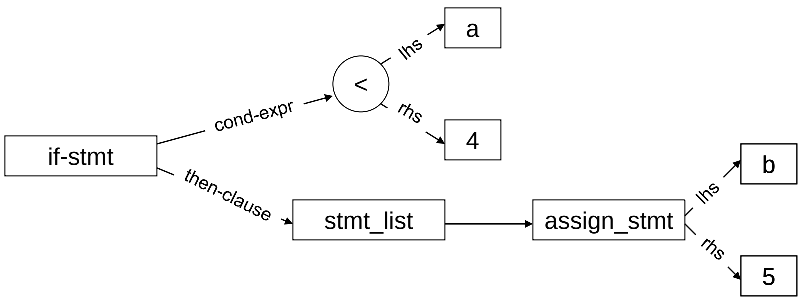


25

The first step is identifying the role of each word. Parsing groups the words into higher level constructs like Subject, Verb, and Objects. That sequence of Subject, Verb, and Object forms an entire sentence. This example of parsing an English sentence is followed in parsing program text as well.

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”)



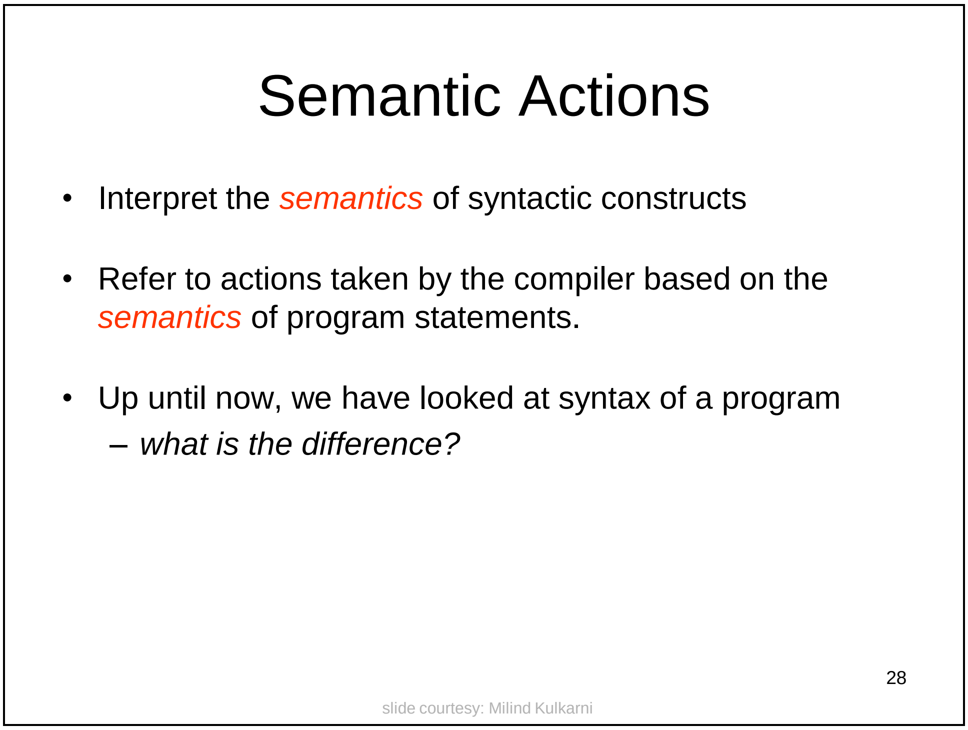
26

slide courtesy: Milind Kulkarni

Exercise

Draw the syntax tree for the following program stmt:

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



Once the structure is understood, we can try to understand the meaning of the sentence. Here we do not have an analogy. Because we do not know how humans understand the meaning of a sentence. We do know that humans first recognize words, sentences much like compilers do lexical analysis and syntactic analysis. For compilers, understanding the meaning of a syntactic structure is too hard. Compilers perform limited semantic analysis for the purpose of catching inconsistencies. They don't really know what the program is supposed to do. Semantic Actions refer to actions that the compiler takes based on the semantics of program statements. What is the difference between syntax and semantics?

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”

29

slide courtesy: Milind Kulkarni

Syntax refers to the grammatical structure of the language. Semantics refers to the meaning.

Semantic Actions - What

- What actions are taken by compiler based on the semantics of program statements ?
  - Examples:
    - Bind variables to their scopes:

```
Ram said Ram has a big heart
          Are they referring to the same person?
```

Programming languages have rules to resolve ambiguities like above:

```
int Ram = 1;
{
     int Ram = 2;
     ..
}
```

- Check for type inconsistencies

```
Ram left her home in the evening
```

30

Semantic Actions - What

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

```
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

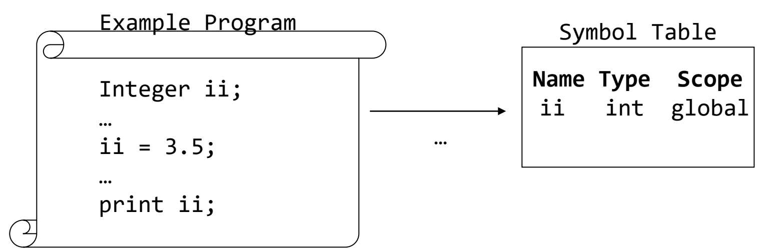
31

Semantic Actions - How

- 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
  - Variable declarations: types, scope
  - Function declarations: return types, # and type of arguments



33

slide courtesy: Milind Kulkarni

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!
```

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

34

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

37

slide courtesy: Milind Kulkarni

Here, making it more efficient may mean making it run faster or use less space or use less power or reduce number of network messages or reduce the number of database accesses or resource usage.

Optimizer - Analogy

Analogy: reducing word usage

```
                                                Dejavu
     Sunny felt a sense of ~~having experienced it before~~
  when his bike broke down.
```

Exercise: *is this rule correct?*

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

38

No strong analogy in English. Maybe like editing text to reduce number of words and find an equivalent word.

This rule would be incorrect for floating point X and Y. e.g. There is a special floating point value called NaN (not a number). As per floating point rules, 0 * NaN = NaN. So, if your optimizer inserted the above rule, it would be incorrect for floating point values.

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
```

→

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

39

slide courtesy: Milind Kulkarni

Code Gen produces assembly code most commonly. It can also produce other types of code such as bytecode, or another high-level language. Here, you are translating from machine independent code (IR) to machine-specific code. In this example, we need to select equivalent machine instruction(s) for “bge a, 4 done” which is an instruction in 3-operand code. In the translated code, we would get the first four instructions. Overall, the translated code uses 3 registers (r1-r3) and uses the ‘bge’ instruction.

Code Generation

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

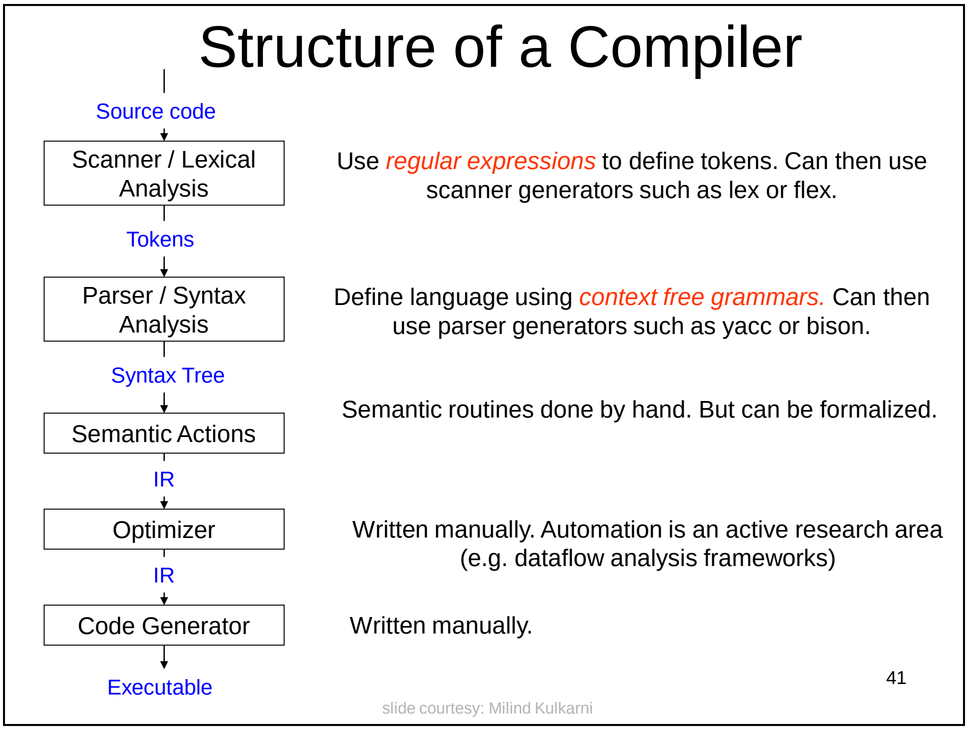
```
bge a, 4 done
mov 5, b
done: //done
```

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

40

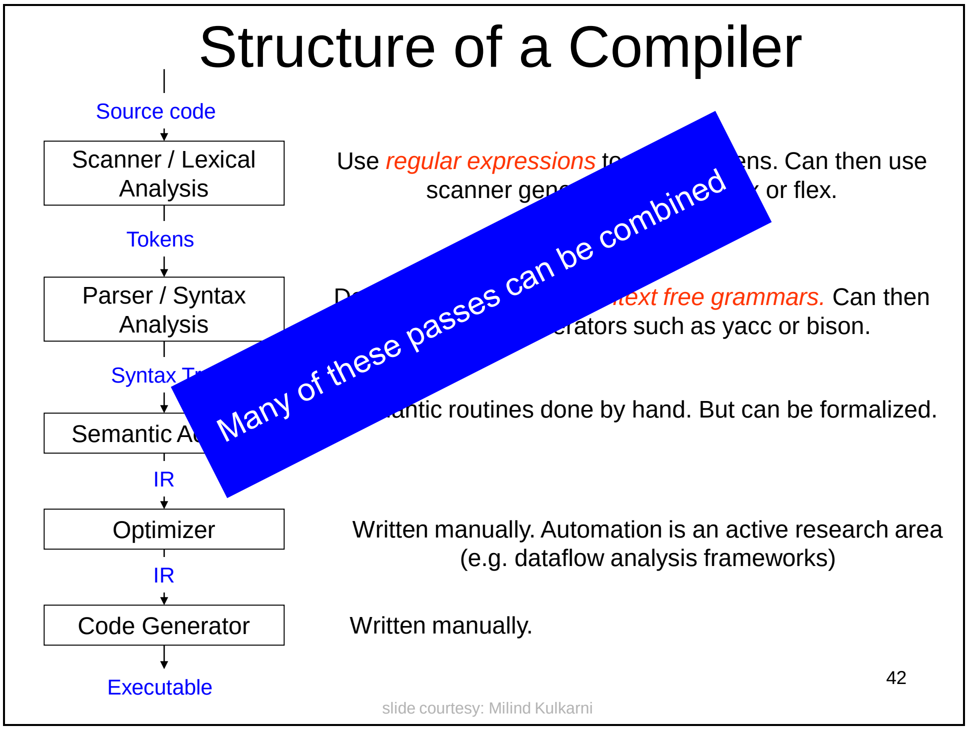
slide courtesy: Milind Kulkarni

We could also have a translation that uses at most 2 registers and that is using a different instruction (blt).



In summary, we have the modularization of compiler design into 5 stages. The first stage takes source code and produces tokens. The second stage takes tokens as input and produces a syntax tree. The third stage works on the syntax tree and produces IR. The fourth stage optimizes IR based on machine specific instructions and produces optimized IR. The last stage generates executable after acting on optimized IR. Also shown in the slide is a brief mention of how a compiler writer defines the ground rules for each stage.

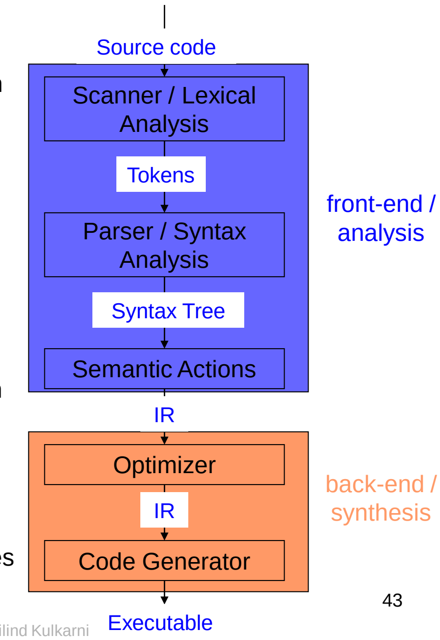
Is this organization of compiler's stages correct? What do you think?



Modern compilers combine many of these passes.

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

43

Design Considerations

- Compiler and 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

44

The design of compilers and programming languages have influence on each other. Higher-level languages are aimed programmer at productivity. In order to talk to machines, first, you have to translate that language into one that is closer to the language that machines understand. To do this translation or bride this gap, compilers must work hard.

Flexible languages such as those that allow dynamic typing are harder to compile. You have to catch the type errors at runtime. The check for conformity on the type rules can be done at compile-time or run-time. If it is done at compile-time they are statically-typed languages. If they are done at run-time they are dynamically typed languages.

If a language requires that you declare the type before its use, it is statically typed. Otherwise, it is dynamically typed

Compiler design is influenced by architectures. This is because the back-end part of the compiler is machine-dependent. The back-end design is simplified for RISC architectures because of a simpler set of instructions. If you have string manipulation instructions available in the assembly, the compiler must be able to recognize them in the source code. Is language-design influenced by architectures?

It is said that C brought a lot of its design features form PDP-11 architecture (CISC). What if you wanted to design an interpreted language that is compiled down to some high-level language, which is then compiled and a machine code is obtained? I wouldn’t care if I used CISC or RISC architecture.

Programming Languages and Real-world Usage

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

45

This slide asks few questions that help us understand how programming languages are used in the real world.

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

48

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)