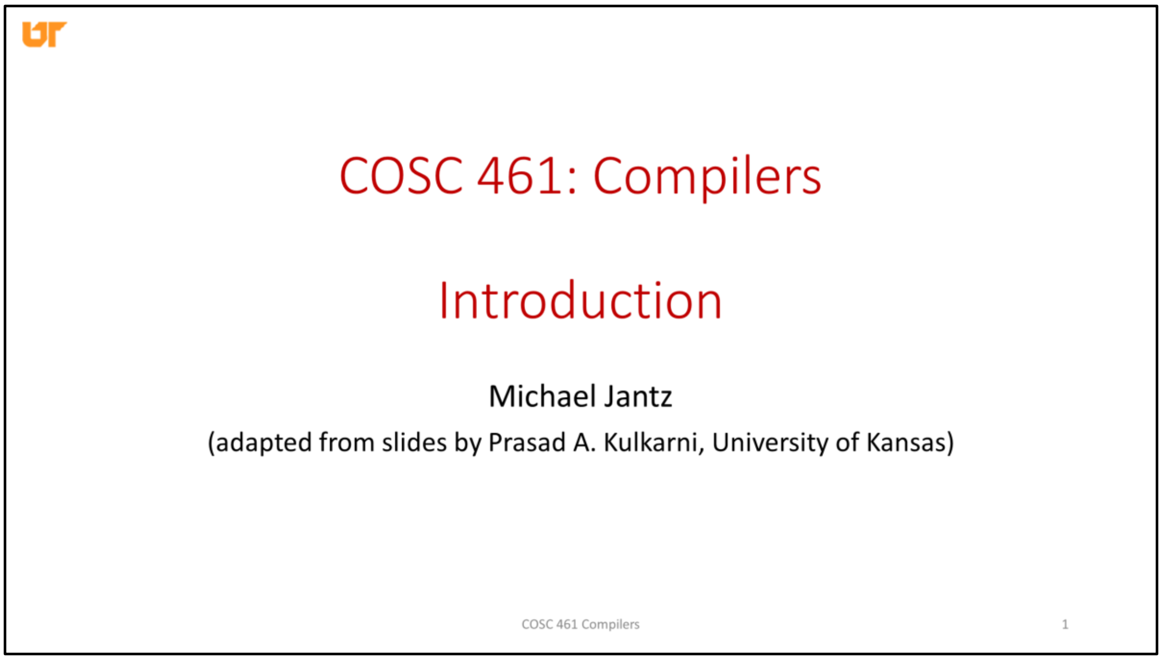
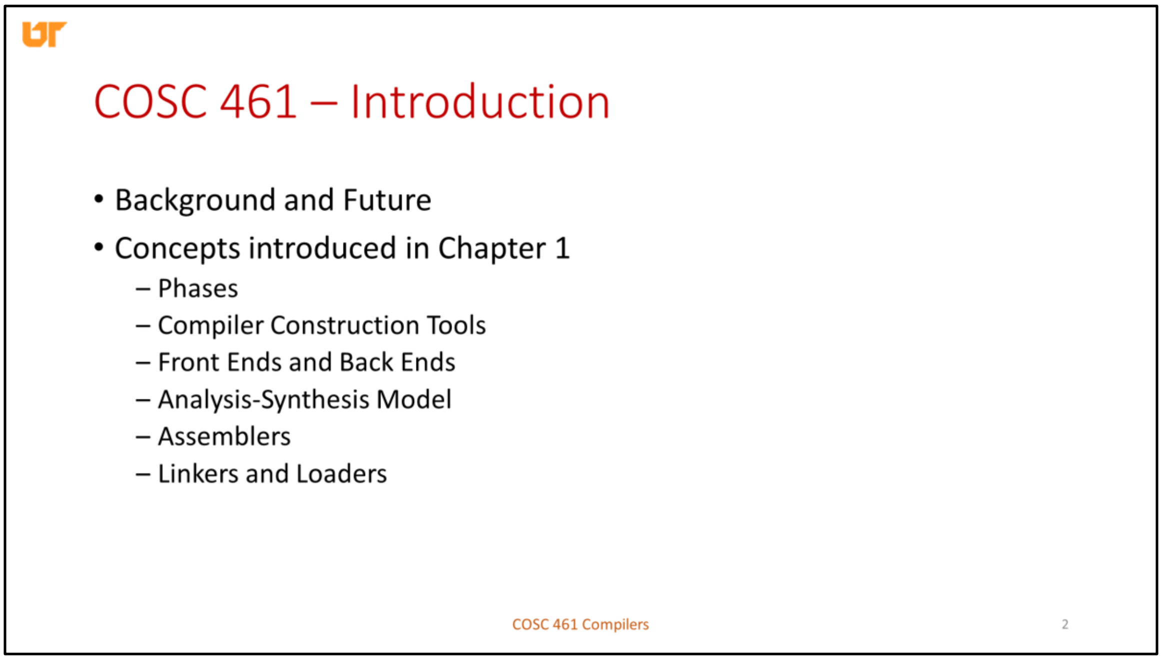


COSC 461: Compilers

Introduction

Michael Jantz

(adapted from slides by Prasad A. Kulkarni, University of Kansas)



COSC 461 – Introduction

- Background and Future
- Concepts introduced in Chapter 1
  - Phases
  - Compiler Construction Tools
  - Front Ends and Back Ends
  - Analysis-Synthesis Model
  - Assemblers
  - Linkers and Loaders

COSC 461 Compilers 2

Chapter 1 is basically an introduction.

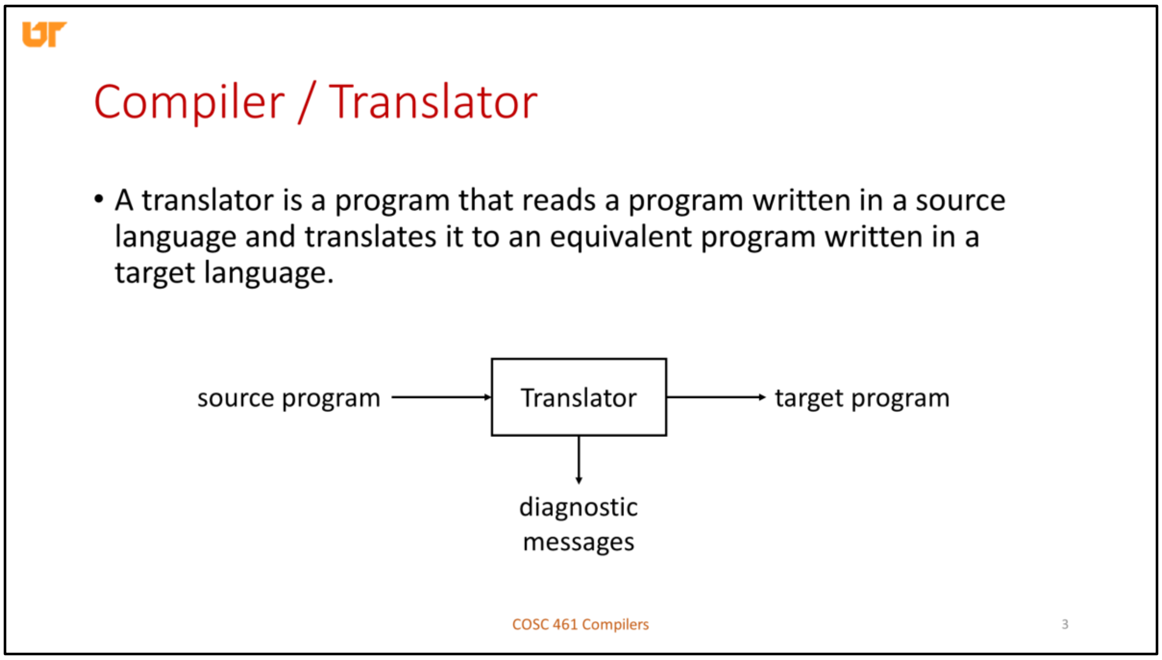
We will discuss the history and importance of compilers.

We will introduce some concepts that will be referenced or used throughout the text.

Q. How many of you have used a compiler before?
A. Should be all.

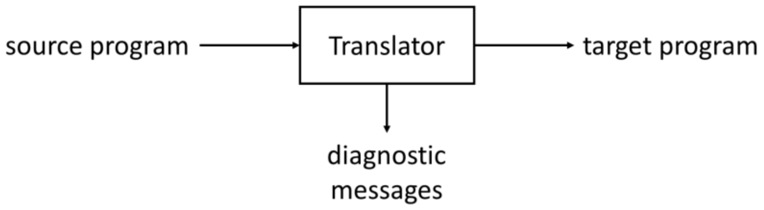
Q. What compiler and for what languages?
Q. Why did you compile the code?

Q. Is the terminal (or bash shell) a compiler? How about Java? Does Java use a compiler?



Compiler / Translator

- A translator is a program that reads a program written in a source language and translates it to an equivalent program written in a target language.



A compiler is a translator program ...

There are different kinds of translator programs

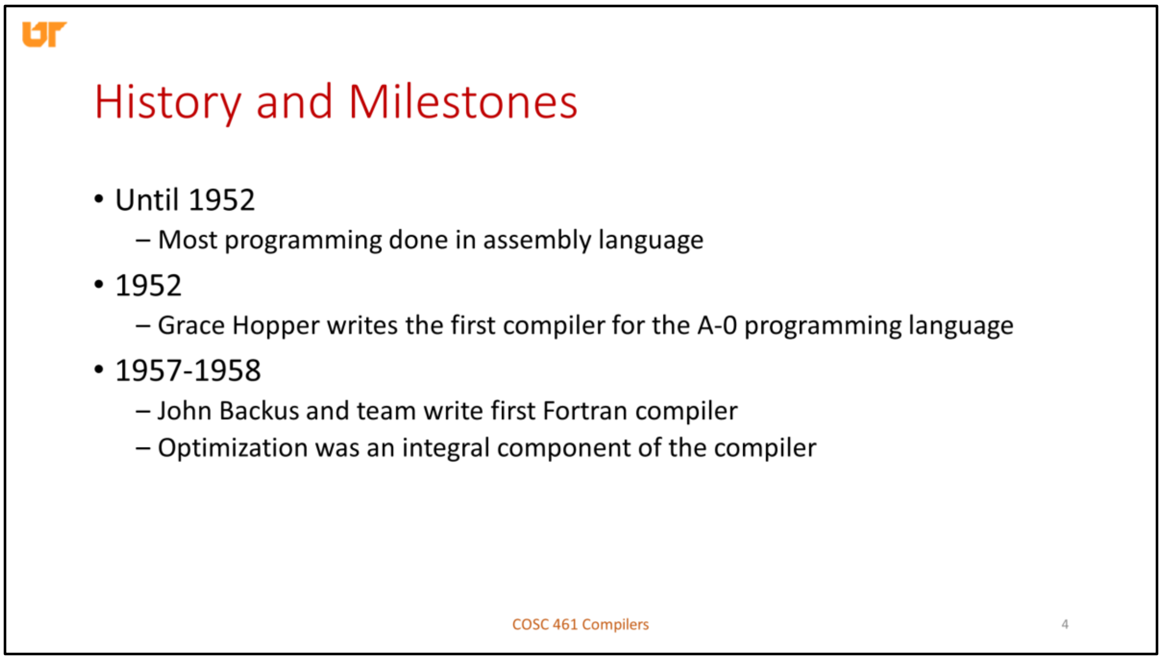
One type of translator might be a source-to-source translator which may be useful for high-level optimizations, or even for languages, like C++ for which the translator converts it to C code.

Another might be a binary-to-binary translator is useful for low-level optimizations, and for running programs on different architectures, like that done in VMs.

A compiler is just one type of translator.

Many of the principles learned in a compilers class can be applied to other applications involving translation or parsing.

Typically, a compiler's source language is a high level programming language and its target language is assembly or machine code.



History and Milestones

- Until 1952
  - Most programming done in assembly language
- 1952
  - Grace Hopper writes the first compiler for the A-0 programming language
- 1957-1958
  - John Backus and team write first Fortran compiler
  - Optimization was an integral component of the compiler

Now, I want to briefly discuss the history and some of the milestones in compilation

In the early days of computing, hardware was expensive, and programmers were cheap.

High-level languages were not popular since programmer productivity was not a major issue. Performance was critical since memory as well as CPU time was extremely limited.

The first compiler was written in 1952 by a woman named Grace Hopper. Grace Hopper is a very important person in the history of computing.

She helped popularize machine-independent program representations.

She is credited with popularizing the term "debugging" for fixing computer problems.

One of the most prestigious awards in computing is named after her – and there is a very large conference for young women in computer science and other computer-related fields is named after her.

Anyway, her compiler is notable because it was the first one, but it was not very popular since compiled programs were much slower than hand-written assembly programs.

John Backus and his team set out to write a compiler for Fortran with the goal of generating code with performance comparable to hand-written assembly.

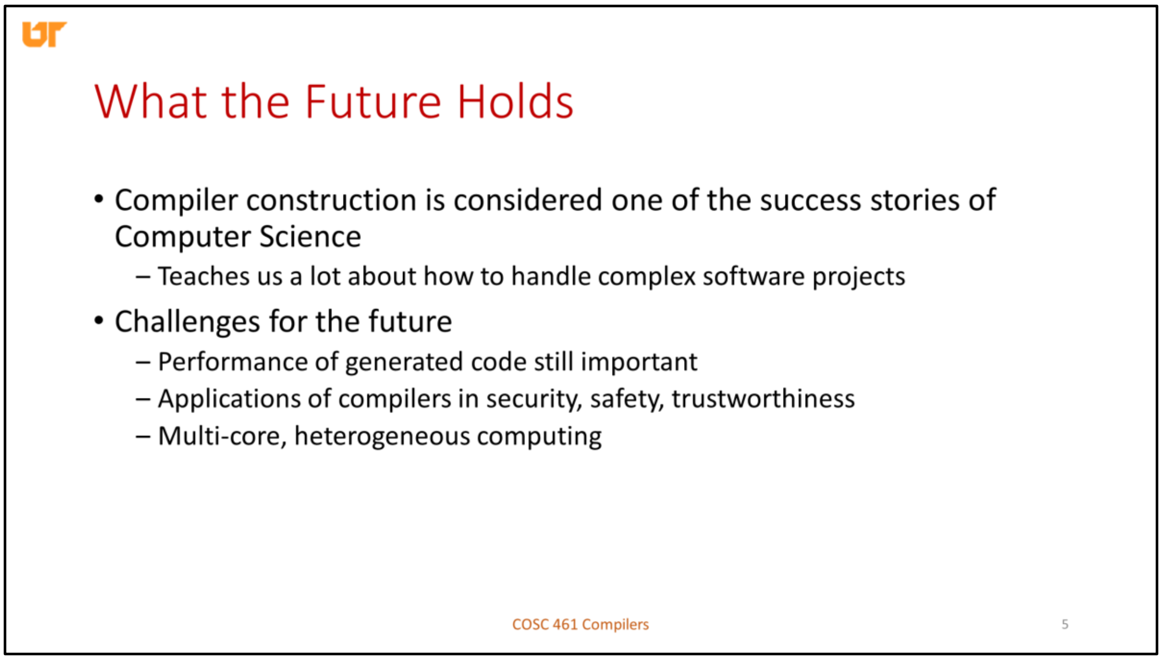
This compiler took 9 people working 24 months to complete. Compilers are much easier to implement today – we can even implement a compiler as a student project in a one semester.

The initial version of Backus' compiler was buggy, but the project was mostly considered a success.

This was the first compiler to be integrated with an optimizer, so the code it produced actually provided decent performance in many cases.

Many of the same compiler optimizations introduced in the first FORTRAN compiler are still used even today.

Q. Compilers are ubiquitous today – but what areas still see a restricted use of compilers?
A. Where performance is still so critical that that achieved by current state-of-the art compiler still falls short and resources are limited. Embedded systems, real-time systems, critical kernels in other high performance applications. These programs may still be hand-written in assembly.



What the Future Holds

- Compiler construction is considered one of the success stories of Computer Science
  - Teaches us a lot about how to handle complex software projects
- Challenges for the future
  - Performance of generated code still important
  - Applications of compilers in security, safety, trustworthiness
  - Multi-core, heterogeneous computing

COSC 461 Compilers 5

Compiler construction is one thing the software developers got right.
From the automata theory, to the automatic components, to the optimizer, the compiler is one piece of complicated software that we know how to build.

While compiler construction is considered a success and many of the problems in compilation have been solved, there are still many open problems for the future.

Some of our challenges going forward include these items:

Performance -- because no level of performance is ever enough.
There are also important applications in security, safety, and trustworthiness.

In the changing security environment, we no longer depend on compilers just to produce optimized code, but also to guaranty certain properties in the code that is generated.
Most important are safety properties of the code, so we can guaranty that the code generated cannot be exploited for computer break-ins.

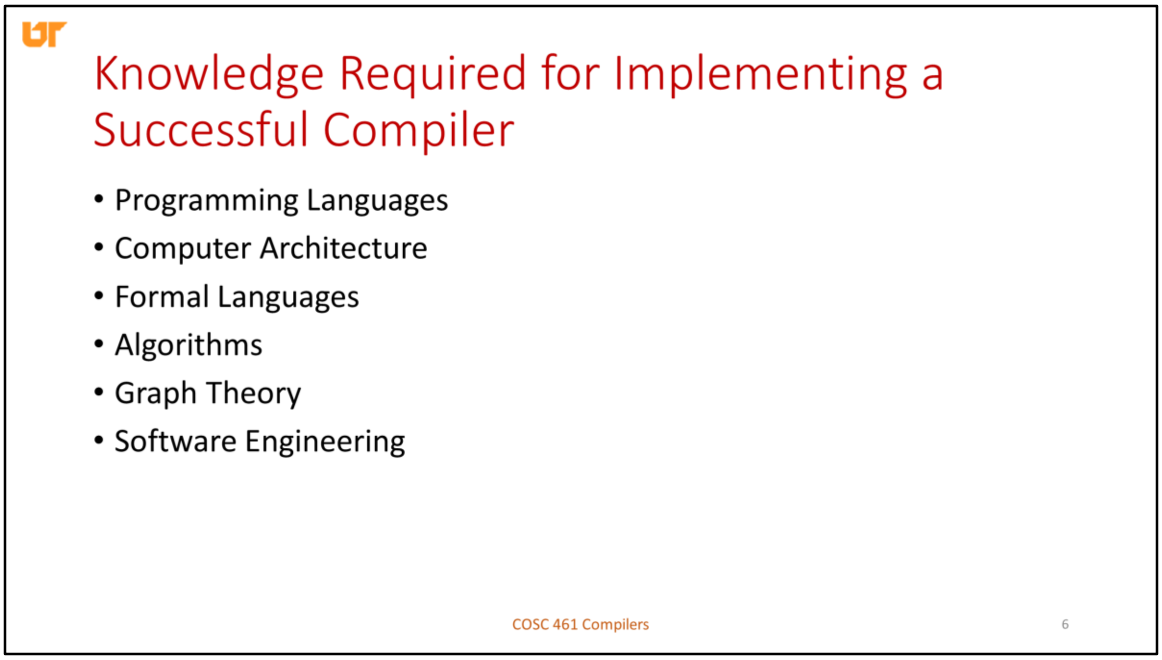
And, over the last decade or two, the world has been moving away from single-core architectures towards multi-core, and current compilers struggle to take full advantage of the newer architectures.

Q. When I say that the world is moving from single-core processors to multi-core machines, what do I mean?
A.   All chips now consist of multiple cores.

Q. More important question: Why this sudden change in direction?
A. Power constraints, unable to scale frequency. Transistor size still scaling.

Q. Why is it difficult to generate high-performance code for multicore systems?
A. With multi-core, we can no longer rely on newer chips to automatically lead to better performance. So, we have to write and generate code that executes across machine cores in parallel -- and with this -- there are a number of issues, such as race conditions, which makes debugging the code more difficult, and we need programs that actually use all of the available cores. Thus, people will increasingly rely on compilers to fill the gap and provide better performance on future machines.

But, before we get there, we need to learn the basics of compiler construction. That is what this class is about. In the later parts of this course we will concentrate on future interest areas in compilers.



Knowledge Required for Implementing a Successful Compiler

- Programming Languages
- Computer Architecture
- Formal Languages
- Algorithms
- Graph Theory
- Software Engineering

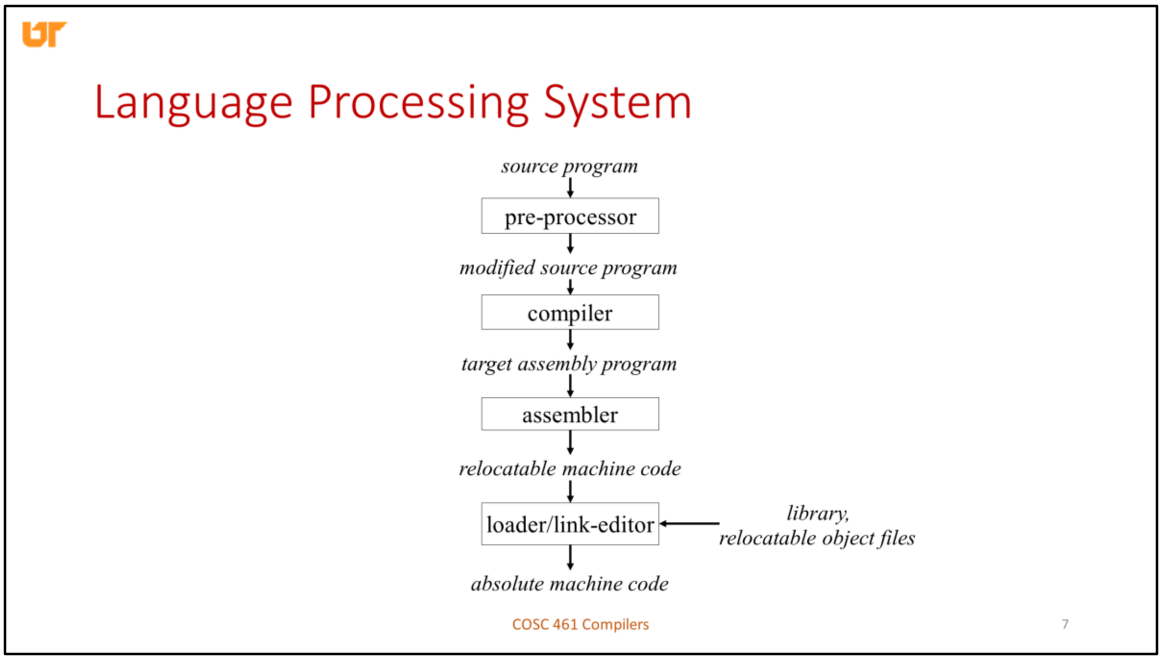
There is a wide range of knowledge required for implementing a successful compiler.

In particular, the compiler is a bridge between programming languages and computer architecture.

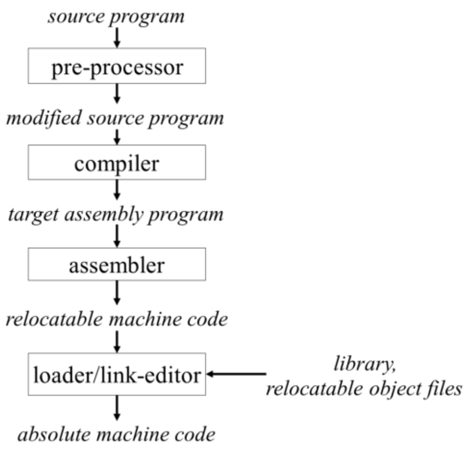
Programming languages – stack frames, scope, type checking, etc.
Computer architecture – target language, code generation, machine-specific optimizations

I want to clarify that when I say architecture, usually I mean the instruction set architecture – which has to do with the language the hardware understands – but not the micro-architecture. For instance, Intel has been using x86 as its ISA for some time, but it continues to update micro-architecture for its individual chips. (write this up)

Formal languages - lexical analysis and parsing
Algorithms - analysis for optimizations, enforcing semantic constraints
Graph theory - flow analysis, optimizations
Software engineering - large amount of code to be written



Language Processing System



COSC 461 Compilers

7

In addition to compilers, several other programs might be required to create an executable program.

This figure shows an overview of a language processing system for a language like C.

The preprocessor phase is responsible for expanding macros and pulling in included files.

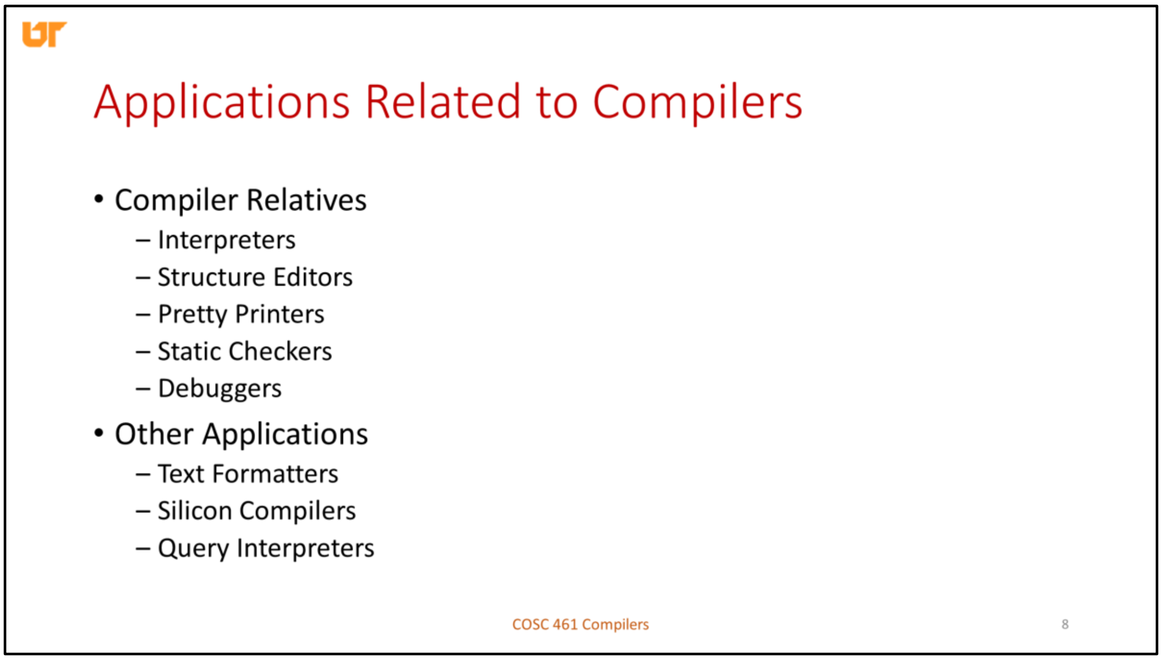
The preprocessed program is then fed to a compiler, which produces an assembly-language program as its output.

The assembly code at this point is just symbolic machine code, which is easier to read and debug than machine code.
Next, the assembly program is fed to an assembler, which produces re-locatable machine code. The code is now in a numeric format (usually binary) but it is still 'relocatable' because it will have placeholders for the linker to fill in with absolute addresses.

If you do have some unresolved addresses in your machine code – maybe because you're compiling a large program over multiple files – you would use a link editor to resolve external addresses, and create an executable from object files and libraries.

Finally, a loader loads instructions and data into memory so the program can be executed.

Write a small program ``hello.c'' to show different compiler steps.
'gcc -E hello.c' --> produces file after preprocessing
'gcc -S hello.c' --> produce assembly file
'gcc -c hello.c' --> object code file after assembly (before linking)
'gcc -o hello.exe hello.c' --> executable file after linking
'./hello.exe' --> loading and execution


Applications Related to Compilers

- Compiler Relatives
  - Interpreters
  - Structure Editors
  - Pretty Printers
  - Static Checkers
  - Debuggers
- Other Applications
  - Text Formatters
  - Silicon Compilers
  - Query Interpreters

COSC 461 Compilers 8

Here we have a list of applications related to compilers

An interpreter is another kind of language processor – but it has some important differences from a compiler, which will discuss in a little bit.

A structure editors is an editing program that is aware of the documents underlying structure. Programs like dreamweaver and powerpoint are structure editors. Some common programming editors like vim and emacs do this to some extent with syntax highlighting. These are also called syntax-directed editors.

A pretty printer is just a program that prints a document so its structure is more easily recognized. There are pretty printers for programming languages. The 'indent' program, which is available on most Linux machines, tries to indent your C code to a given specification, and is an example of a pretty printer.

Static checkers find bugs in a program that a compiler does not (perhaps due to efficiency reasons or due to the definition of the programming language). Lint is an example of a static checker for C.

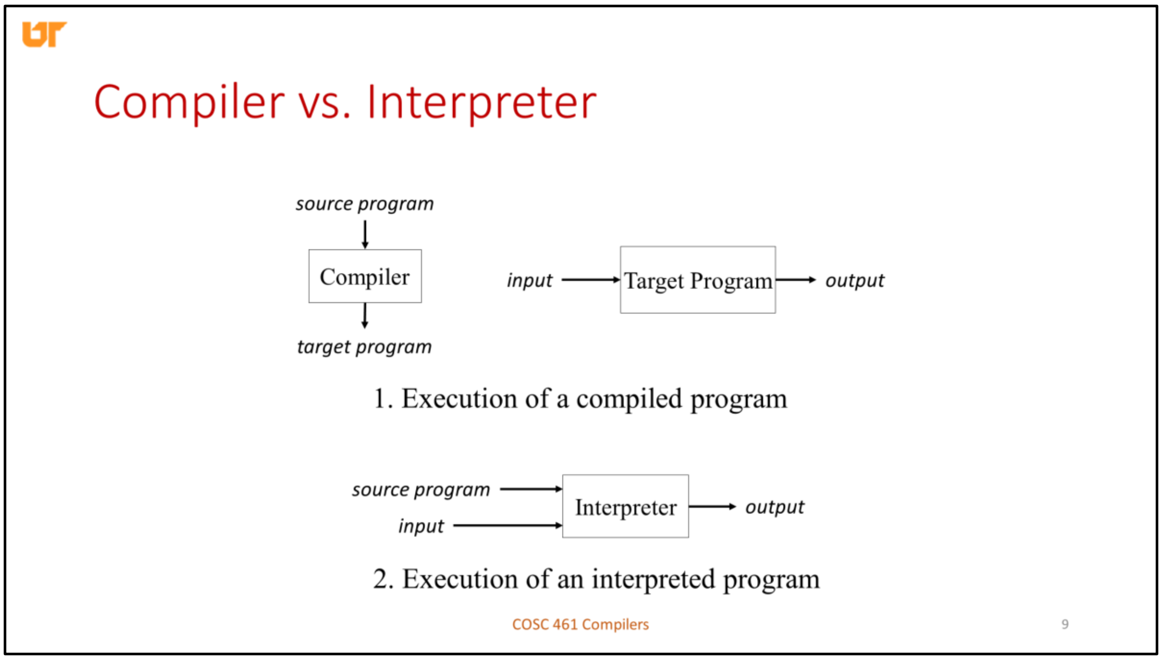
Lint detects the following errors:
1. Unused \#include directives, variables, and procedures
2. Memory usage after its deallocation
3. Unused assignments
4. Usage of a variable value before its initialization
5. Deallocation of nonallocated memory
6. Usage of pointers when writing in constant data segments
7. Nonequivalent macro redefinitions
8. Unreached code
9. Conformity of the usage of value types in unions
10. Implicit casts of actual arguments.

Debuggers provide an interface for diagnosing the execution of a program. One of the most popular debuggers is gdb.
Debuggers are related because they need to maintain a similar symbol table as a compiler. Debuggers are your best friends when trying to comprehend large code bases.
Q. Has anybody used gdb? Or any other debugger?

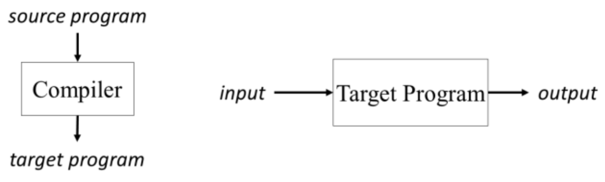
Text formatters use a batch mode for producing papers or documents. They use a language to specify the text formatting commands. latex is an example of a text formatter.

Silicon compilers automatically take a high-level specification and produce a circuit design for a chip.

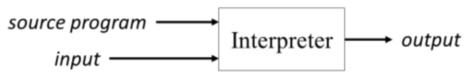
Query interpreters provide an interface to a database. They translate predicates containing relations into commands to interface with a database.



Compiler vs. Interpreter



1. Execution of a compiled program



2. Execution of an interpreted program

COSC 461 Compilers 9

This slide shows models for how a compiler works and how an interpreter works.

A compiled program operates in two steps:
1. the compiler translates the source program to a program in the machine language of the target processor,
2. The target program can then be directly executed.

Interpreters, on the other hand, perform the operations implied by the source program, one source statement at a time.

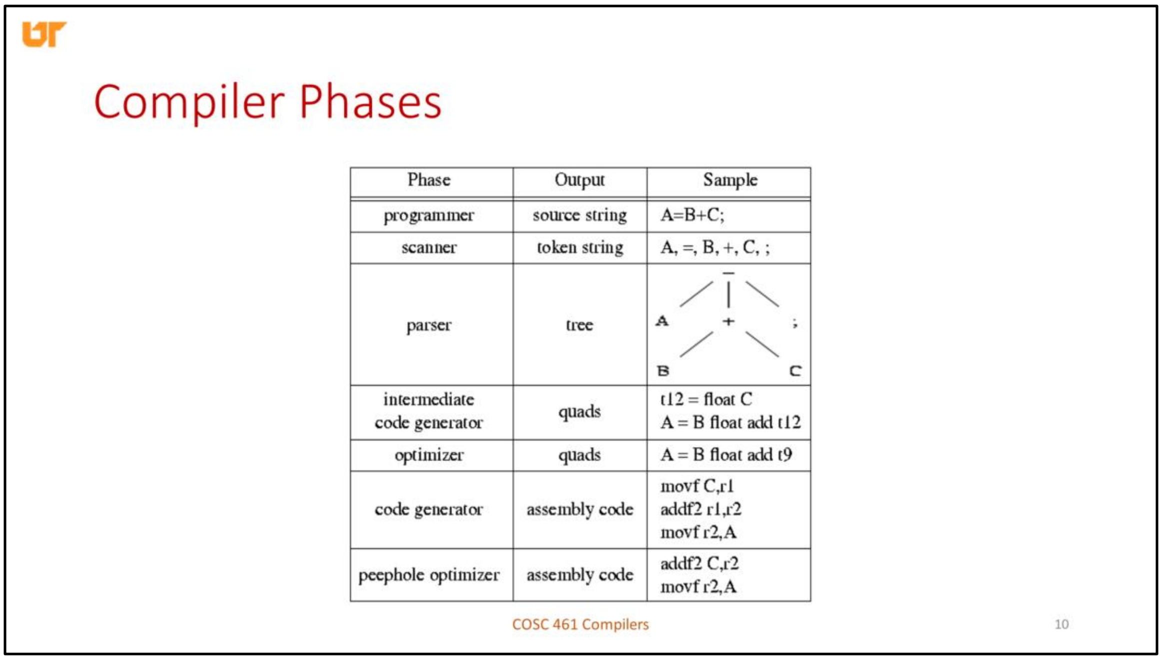
The machine language target program produced by the compiler is usually much faster than an interpreter at mapping input to output. An interpreter can, however, be faster than the combined time required for compilation followed by execution depending on how long your program actually runs. An interpreter can also provide better error diagnotics.

Q. Why would we use interpretation? A. Flexibility, portability.

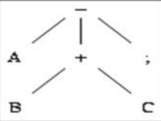
Q. Do you know of any software interpreter? How about hardware interpreter?

A computer can be viewed as a hardware interpreter of its instruction set.
Interpreters are used for command languages, AI languages like LISP, APL,
Interpreters give more flexibility at the cost of efficiency.

[May show operation of an interpreter on a small C program. Compiler produces assembly, while the interpreter has a fetch-decode-execute loop.]



Compiler Phases

Phase	Output	Sample
programmer	source string	A=B+C;
scanner	token string	A, =, B, +, C, ;
parser	tree	

intermediate code generator	quads	t12 = float C
		A = B float add t12

optimizer	quads	A = B float add t9
code generator	assembly code	movf C,r1 addf2 r1,r2 movf r2,A
peephole optimizer	assembly code	addf2 C,r2 movf r2,A

COSC 461 Compilers 10

Up to now, we've only considered the compiler itself as a black box. If we examine the compilation process in more detail, we see that compilers operate as a series of phases. This table shows the important phases in a typical optimizing compiler.

The programmer will produce the original source code. From there, the programmer invokes the compiler.

The scanner scans the input code and breaks it up into tokens. It will create entries in the symbol table to associate variable names with their tokens. Most compilers will typically invoke the scanner as a routine as its needed. Few errors will occur at this level. Really, the only thing that could go wrong is if we have an invalid character in the input stream.
Q. Name one character that is not a part of the C language?
A.   '@' or '\$' will be detected as invalid character by the lexer.

A parser has two responsibilities: (1) check for correct syntax and (2) present the program in a form or data structure (e.g. parse tree) from which it can be translated.

After parsing, we generate intermediate code. Don't worry about why it's called ‘quads’ – that’s just a name for the intermediate code in this table. We will soon discuss why intermediate code is generated first.

Next, an optimizer is applied to make transformations on the intermediate code that will eventually improve the performance of the final generated code.

A code generator converts the intermediate program representation into machine instructions.

And then it uses a peephole optimizer to optimize the machine code.

So notice, in this compiler, the optimizations are to both the intermediate code and the machine code.

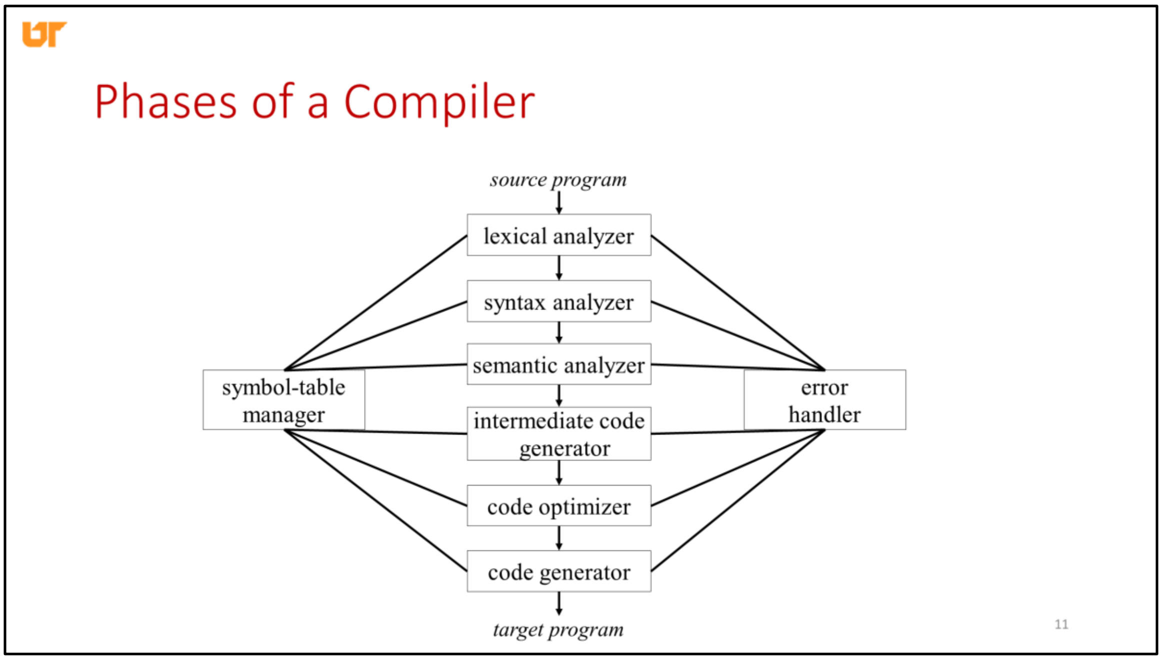
To understand why this might be necessary, consider this example:

[Show examples of machine-independent (unreachable code) and
 machine-dependent (multiply-add instruction) optimizations.]

```
\begin{verbatim}
 1. foo(){
      printf(...);
      return;
      printf(...); // unreachable code
    }

 2. a = a + b*c;
    MUL b, c, temp
    ADD a, temp, a
    =>
    MULADD b, c, a
\end{verbatim}
```

It is generally better to apply optimizations to machine independent code (if possible).



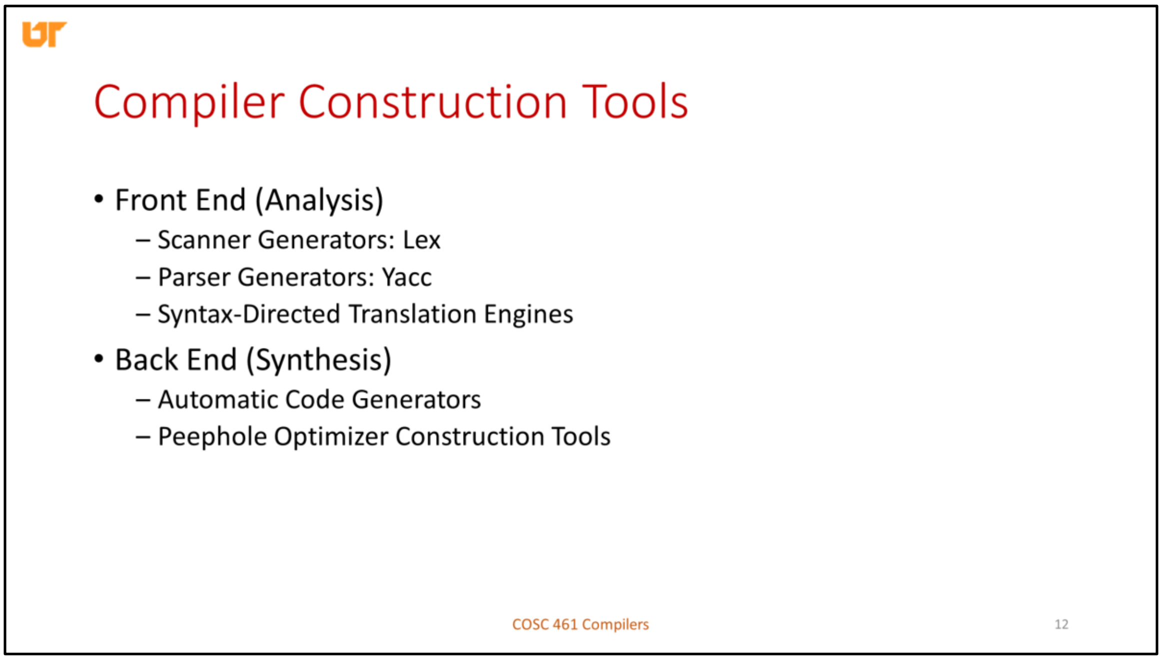
Phases of a Compiler

This figure shows how all these phases work together to translate a source program to a target program.

Can see that some phases put things into the symbol table and other phases take things out.
All phases have to deal with error handling, but the syntax and semantic analysis have to deal with it the most.

Some compilers, like GCC and the compiler I use, perform most of the optimizations after generating intermediate code.

We will discuss this more later.



Compiler Construction Tools

- Front End (Analysis)
  - Scanner Generators: Lex
  - Parser Generators: Yacc
  - Syntax-Directed Translation Engines
- Back End (Synthesis)
  - Automatic Code Generators
  - Peephole Optimizer Construction Tools

COSC 461 Compilers 12

In compiler construction, we talk about the phases of the compiler generally being part of one of two major components: the front end or the back end.

The front end is basically responsible for analysis: that is breaking up the parts of the program and imposing a structure on them. And the back end (or synthesis part) builds up the target program from the intermediate language and the information in the symbol table.

Many tools have been created to help automatically produce various components of a compiler.

This is the main reason why compiler construction now is less time consuming.

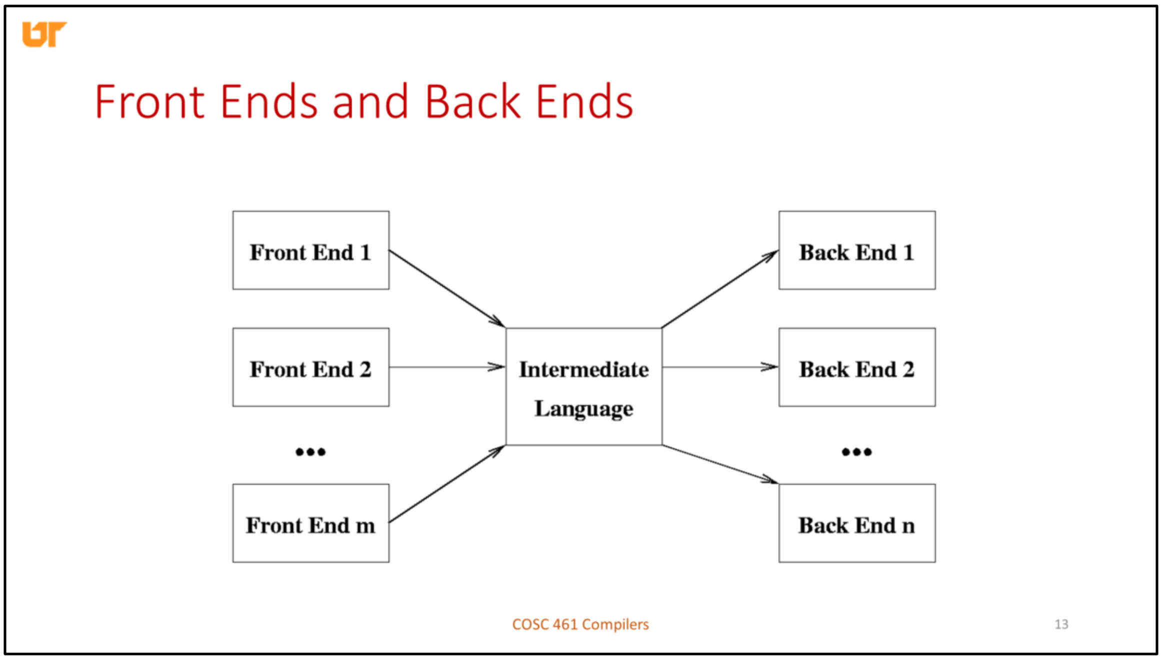
In the front end, scanner generators (like lex) produce lexical analyzers, which use specifications based on regular expressions.

Parser generators (like yacc) produce syntax analyzers, which use specifications based on context free grammars.

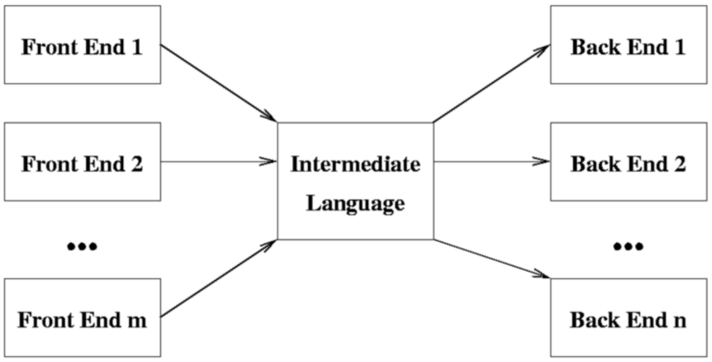
And syntax-directed translation engines are used to produce a collection of routines that walk a parse tree generating intermediate code.

In the back end, automatic code generators take a collection of rules to translate each intermediate language operation into target language operations (machine instructions).

And peephole optimizer construction tools read in rules or patterns to automatically produce a peephole optimizer.



Front Ends and Back Ends



This figure shows a common organization for many compilers.

So, essentially we write front ends for different source languages so that they all map to the same intermediate form. Then, we write backends for different machine architectures that operate on this intermediate form.

The idea is to reduce the effort by reducing the number of compilers needed.

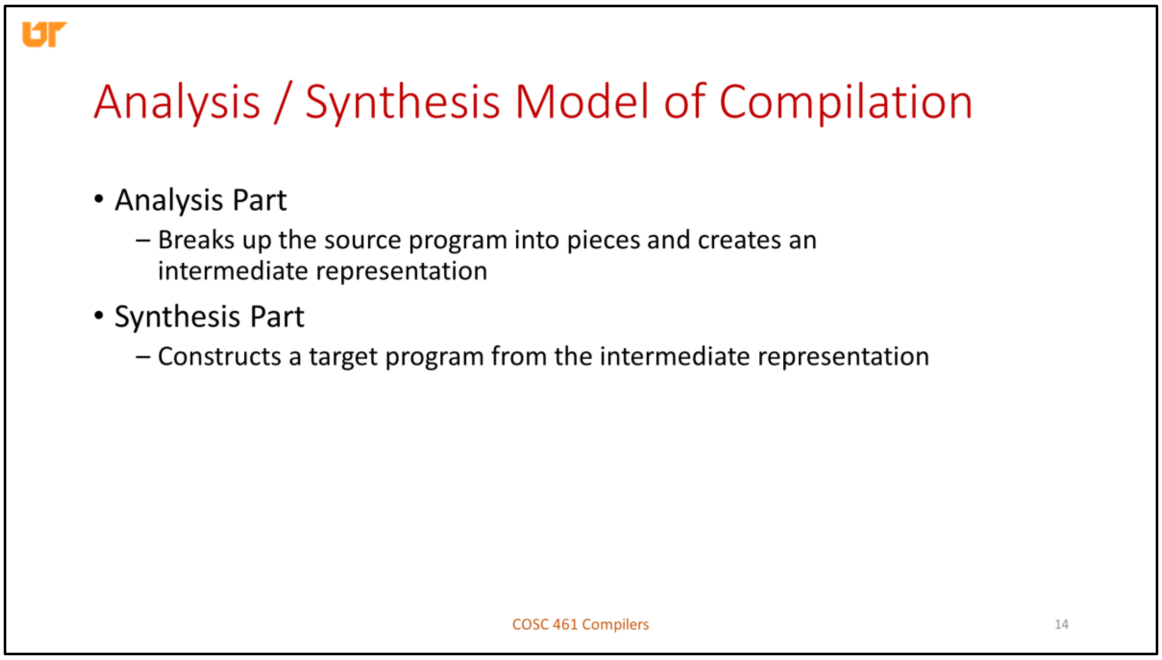
So, for instance, if we want to compile C to ARM and x86, we could write two different compilers: one that compiles C to ARM and one that compiles C to x86. But, with this design, now write write a front end that compiles C to the intermediate form, and two backends – one that translates the intermediate form to ARM and another that compiles it to x86.

What is this benefit of this design?

Without this approach one requires m*n compilers, where m is the number of source languages and n is the number of architectures. With this approach, one only needs m+n compilers since the intermediate language is the common medium for translation.

UNCOL stands for UNiversal Computer Oriented Language. It was an attempt to define a common intermediate language for all compiled source languages and all computer architectures. It was not too successful. So – there are still a lot of different intermediate languages around.

Still, current systems try to exploit this idea as much as possible. GCC uses a common intermediate language for similar languages – and LLVM includes an intermediate form that has been integrated with many different languages and architectures.



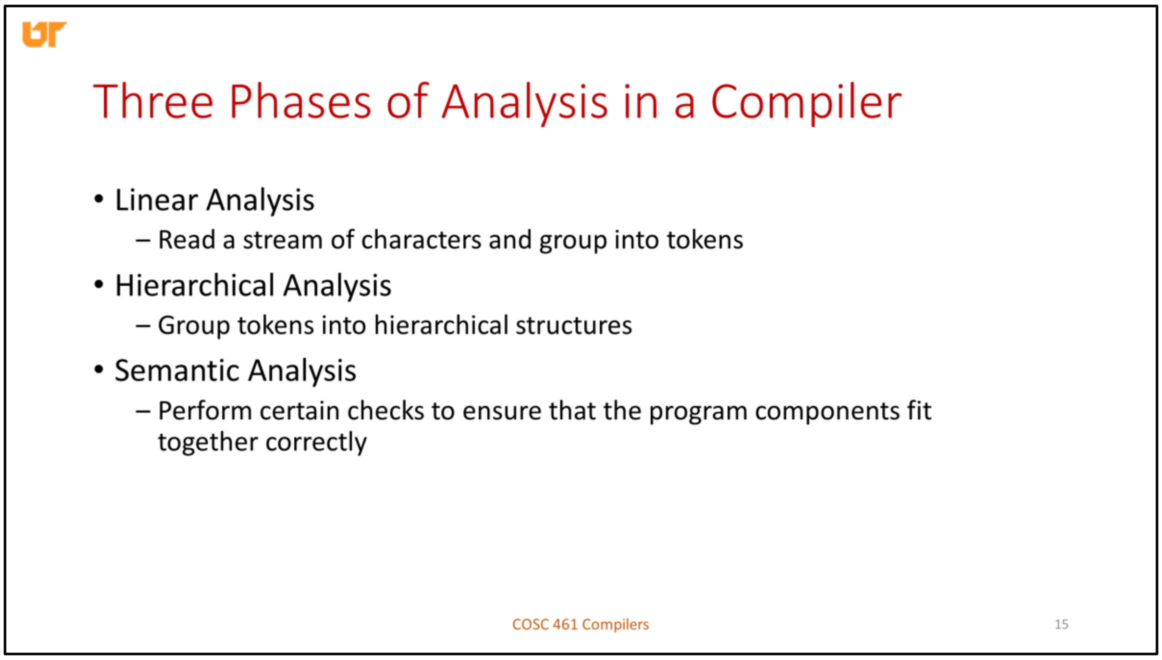
Analysis / Synthesis Model of Compilation

- Analysis Part
  - Breaks up the source program into pieces and creates an intermediate representation
- Synthesis Part
  - Constructs a target program from the intermediate representation

So, given this organization, we can talk about two main parts of the compiler: the analysis part and the synthesis part

Analysis means to examine by breaking up into smaller pieces. So, the analysis part breaks the source program into smaller pieces and imposes a structure on them. Using this structure, it builds an intermediate representation of the source program. The analysis part corresponds to the front end of the compiler.

And synthesis means to build something up from smaller pieces. So, the synthesis part constructs the desired target program from the intermediate representation. This corresponds to the compiler backend.

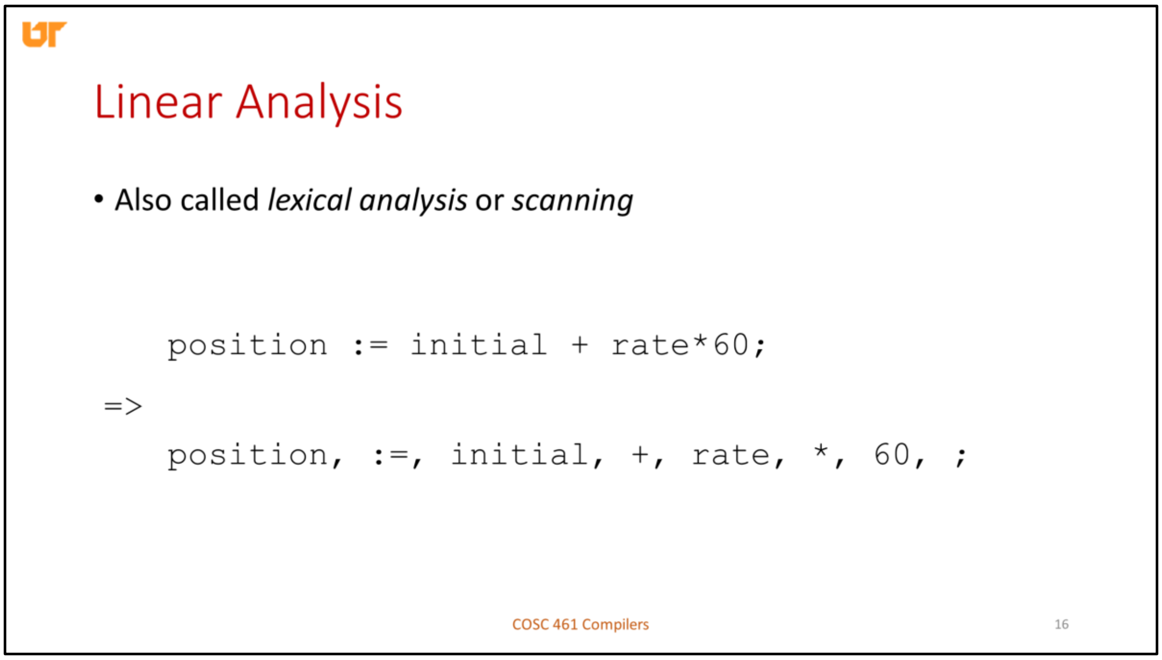


Three Phases of Analysis in a Compiler

- Linear Analysis
  - Read a stream of characters and group into tokens
- Hierarchical Analysis
  - Group tokens into hierarchical structures
- Semantic Analysis
  - Perform certain checks to ensure that the program components fit together correctly

COSC 461 Compilers 15

There are three parts of the analysis phase that are generally integrated together, that is, they are performed all at once.

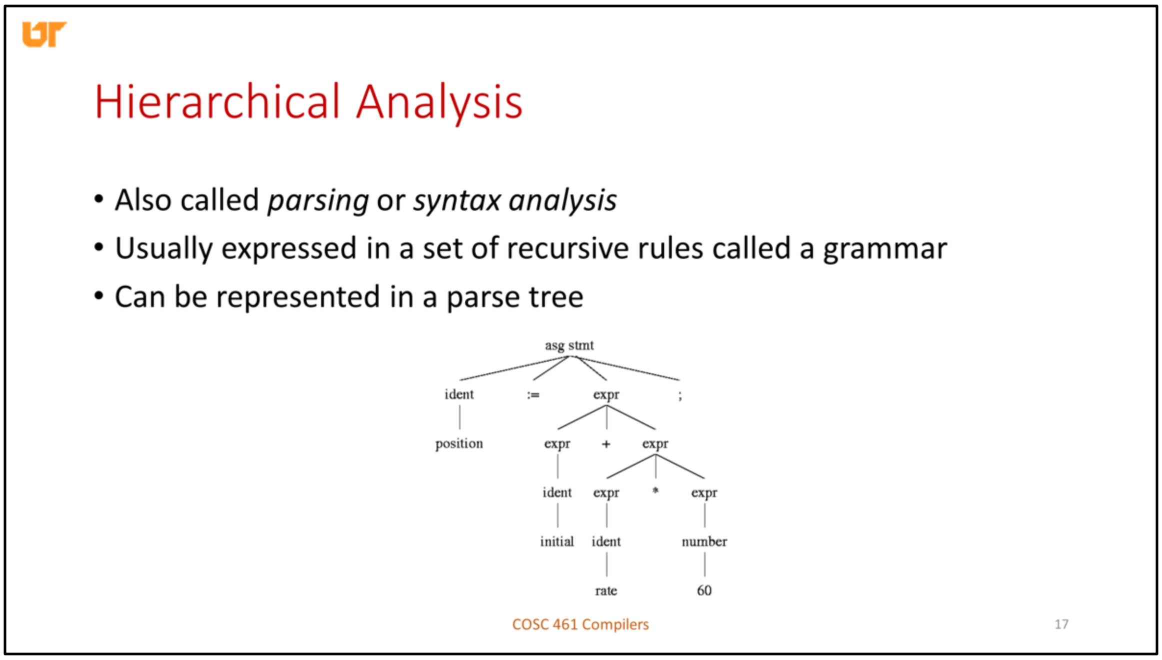


Linear Analysis

- Also called *lexical analysis* or *scanning*

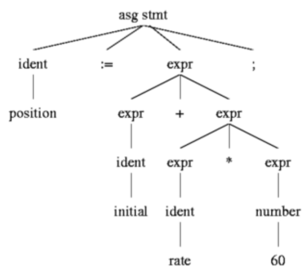
```
    position := initial + rate*60;
=>
    position, :=, initial, +, rate, *, 60, ;
```

The linear analysis phase, which is also called lexical analysis or scanning, breaks the stream of input characters into tokens. Blanks and other white space is generally ignored, except as token separators. So, in this example, if this is your input program, the scanner might break it up into tokens that look like this.



Hierarchical Analysis

- Also called *parsing* or *syntax analysis*
- Usually expressed in a set of recursive rules called a grammar
- Can be represented in a parse tree



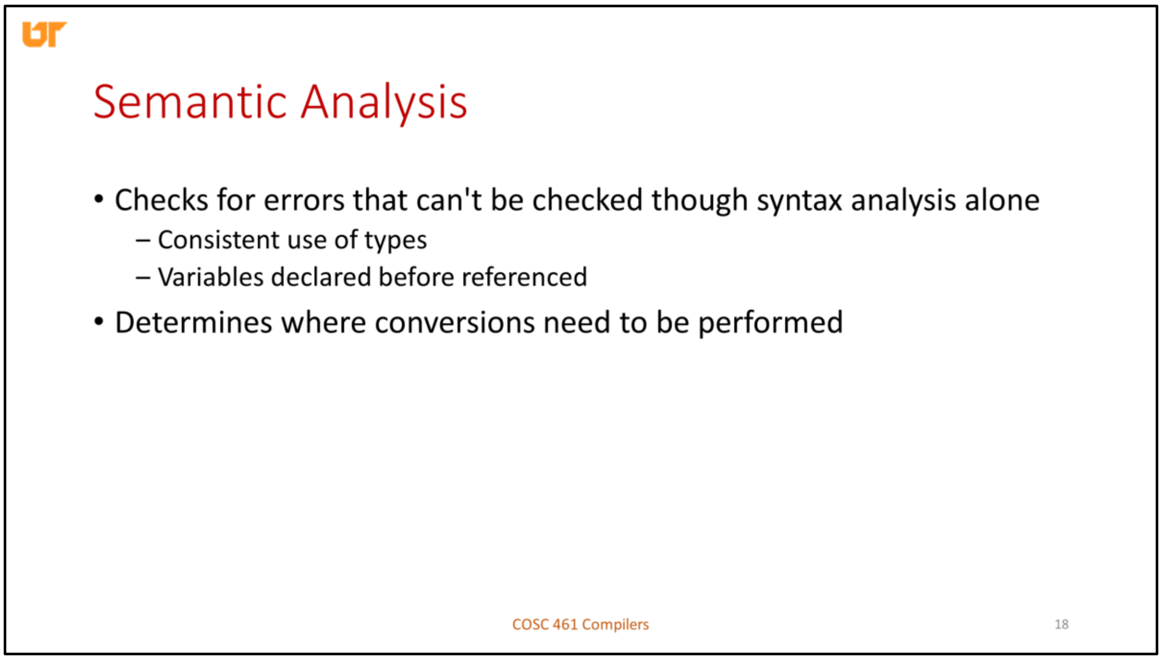
COSC 461 Compilers 17

Next, we have the hierarchical analysis phase, which is also called parsing or syntax analysis.

When we say syntax, we are referring to the structure of the language.

This structure is usually expressed by a set of rules called a grammar.

This phase uses the tokens from the linear analysis phase and creates an intermediate form representing the grammatical structure of the token stream. This form might be any data structure, but we usually use some sort of tree structure. We'll discuss more about this phase later in the course.



Semantic Analysis

- Checks for errors that can't be checked though syntax analysis alone
  - Consistent use of types
  - Variables declared before referenced
- Determines where conversions need to be performed

COSC 461 Compilers 18

And we also have the semantic analysis phase. In contrast to syntax, which again, deals with language structure, semantics deals with the meaning of words in the language.

Not all of the rules in a programming language can be specified as syntax alone. There are often cases where it might be helpful to employ semantic analysis to construct our intermediate form.

An important part of this phase is type checking. In a statically typed language (like C), this means that this phase would check that each operator has operands of the appropriate type, and that each variable has been declared as having some type before it is used.

Relatedly, semantic analysis deals with making sure variables are declared before they're referenced.

It’s also important for knowing which conversions need to be performed.

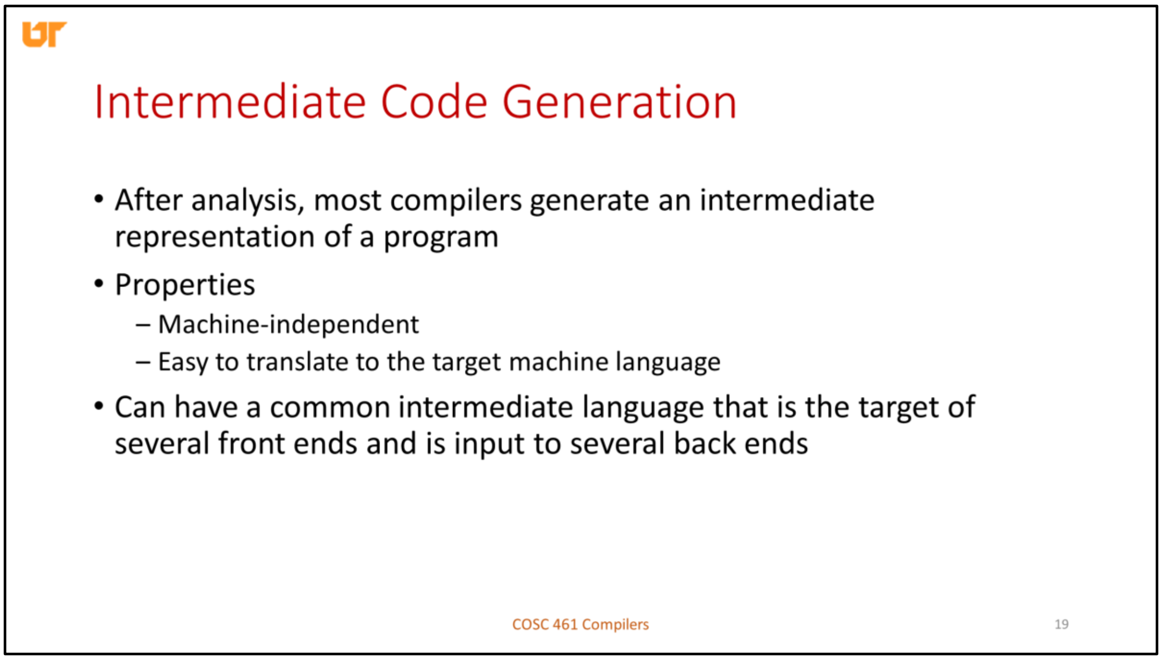
Example:

```
\begin{verbatim}
int a = 10;
double b;

b = a; // will need to be converted to b = (double)a;
\end{verbatim}
```

Q. Do we need a conversion here? Why do we need this conversion?
Yes. The type conversion will change the format of the operands from integer to floating point or vice versa. The target language might use different instructions depending on the type of the data.

This implicit type conversion is called coersion.



Intermediate Code Generation

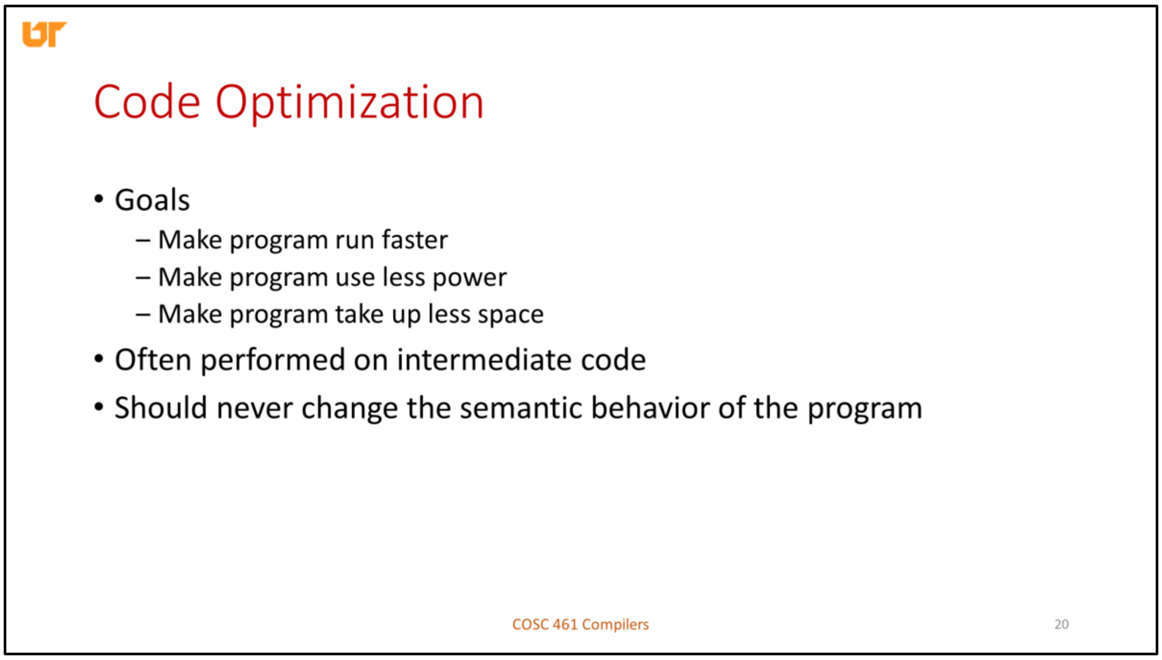
- After analysis, most compilers generate an intermediate representation of a program
- Properties
  - Machine-independent
  - Easy to translate to the target machine language
- Can have a common intermediate language that is the target of several front ends and is input to several back ends

COSC 461 Compilers 19

After analysis, most compilers generate an IR of the program. We will see there are many different types of intermediate representations (trees, graphs, stack, accumulator, triples, quadruples, etc.).

The most important properties of an intermediate representation are that its machine independent, and that it’s easy to translate to the target machine language.

And, as I mentioned before, the strategy of having an intermediate language facilitates the translation of many source languages to many target architectures.


Code Optimization

- Goals
  - Make program run faster
  - Make program use less power
  - Make program take up less space
- Often performed on intermediate code
- Should never change the semantic behavior of the program

COSC 461 Compilers 20

Now, most compilers apply some optimizations to the program code. “Optimization” is sort of a misnomer since a compiler cannot guarantee that it can produce optimal code. A better name for optimizations might be “code improver” because these are transformations that are applied to the code to try to improve it in some way.

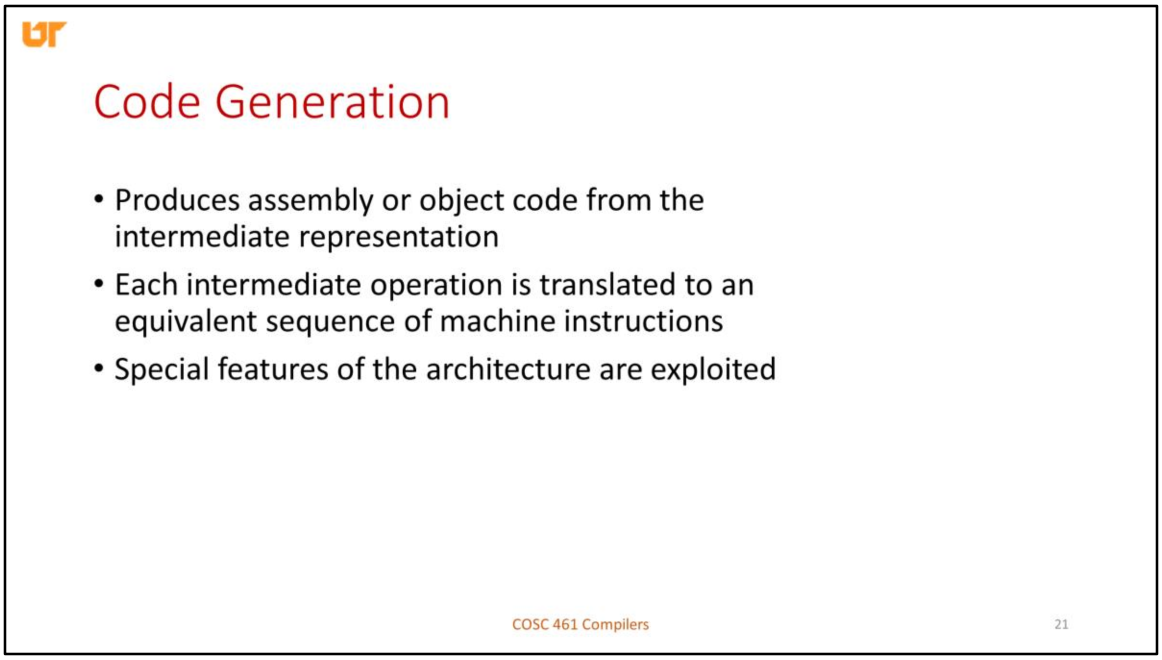
Specifically, there might be optimizations that try to make the program run faster or use less power or take up less space .

Q. Now, other than the obvious benefit of taking up less space on your disk, why else might it be important to have your program take up less space?
A. If your machine has a limited code or instruction cache, a smaller program will enable faster execution because of fewer cache misses.

Code optimization has been a hot topic of research for a long period of time. Architectures and machines are constantly changing and producing efficient code for machines is a challenge.
Multicore systems and embedded systems both pose unique challenges for optimization (multicore might want to increase parallelism for performance – while on an embedded device its more important to reduce code size and save battery life).

Many compilers will use the intermediate form to implement optimizations because any optimizations performed on the intermediate code will be machine independent.

Another challenge in code optimization is making sure to write your optimizations so that they are always safe – that is they never change the semantic behavior of the program.



Code Generation

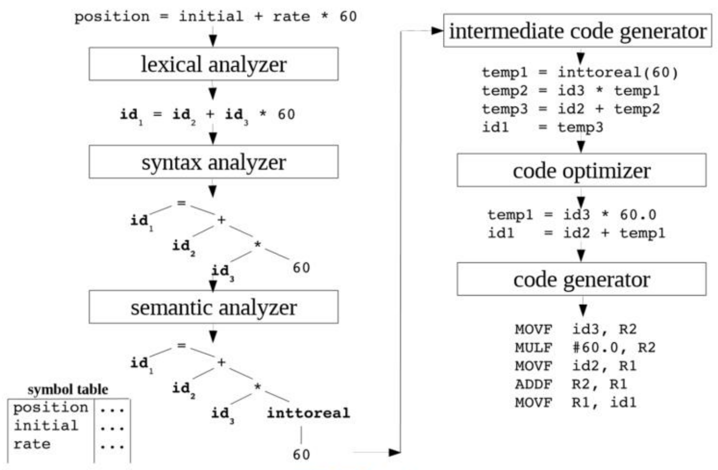
- Produces assembly or object code from the intermediate representation
- Each intermediate operation is translated to an equivalent sequence of machine instructions
- Special features of the architecture are exploited

And the final phase in compilation is code generation. The main purpose of this phase is to produce assembly or object code from the intermediate representation.

In this phase, each intermediate operation is translated to an equivalent sequence of machine instructions.

And, we might apply machine-specific optimizations to take advantage of special features of each architecture.

Translation of a Statement



```
temp1 = inttoreal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1   = temp3
```

```
temp1 = id3 * 60.0
id1   = id2 + temp1
```

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

COSC 461 Compilers 22

This figure shows how the whole process works together for a single statement.

The compiler initially breaks the input statement into tokens using the lexical analyzer.

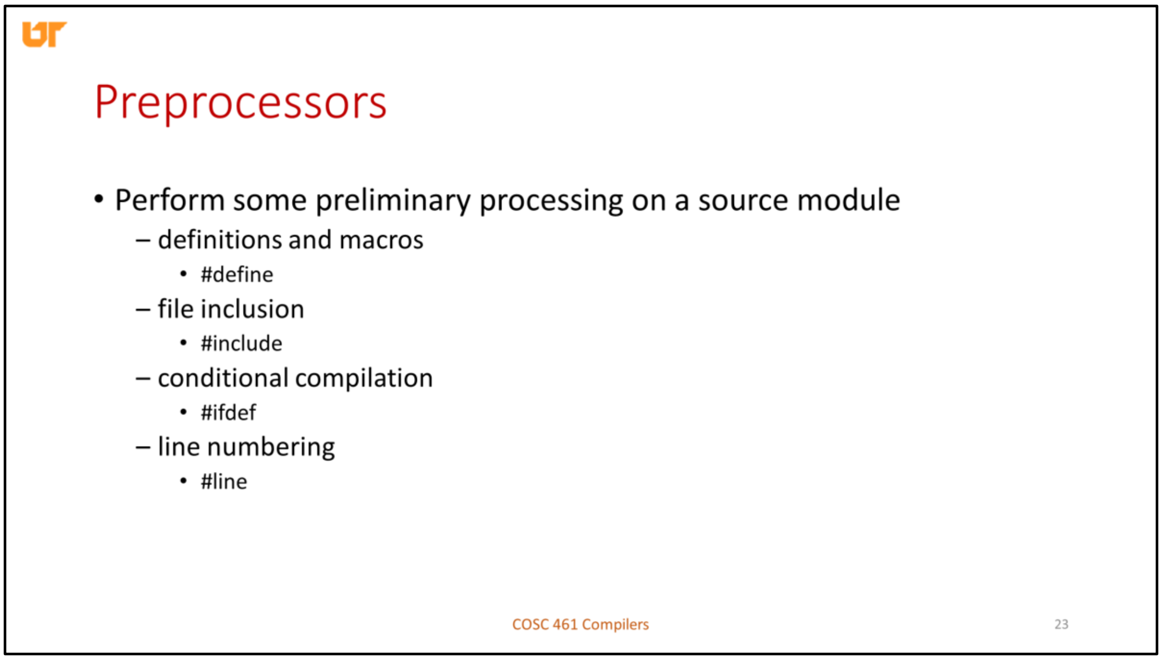
Then the syntax analysis checks if the order of the tokens meets the syntactic rules of the language, and parses the tokens into a tree structure. This phase helps with the translation as well since you can see that with the parse tree we can impose some order on how the operations are done.

The semantic analyzer checks for rules that cannot be checked by syntax alone. In this case, the semantic analyzer recognizes that the constant 60 needs to be converted from type ‘int’ to type ‘real’.

From the form resulting from semantic analysis, the compiler generates intermediate code for the statement. This code is machine independent at this point.

Next, the compiler applies some optimizations to this intermediate form. In this case, the optimizer was able to remove two of the four instructions.

And finally, the code generator maps this intermediate code to assembly or machine code for the target architecture – and it's not shown here – but this code may be further optimized by machine dependent optimizations.


Preprocessors

- Perform some preliminary processing on a source module
  - definitions and macros
    - #define
  - file inclusion
    - #include
  - conditional compilation
    - #ifdef
  - line numbering
    - #line

COSC 461 Compilers 23

Before moving on, let's quickly discuss some of the other parts of a typical language processing system.

Many languages, such as C and C++, use a preprocessor to perform some preliminary processing on the source code.

In C, it is common to setup definitions and macros. Definitions are used to define constants and macros are used to accomplish abstraction without the overhead of calls and returns.

```
int add_2(int x) {
  return (x + 2);
}
versus
#define add_2(x) (x+2)
```

With a function, have to create a context for the function we're calling and jump to the function code. Then, when it's done executing, it has to return, which involves saving the return value and jumping to the old code. If we use a macro, the preprocessor inlines this code for us and we avoid the overhead of a function call.

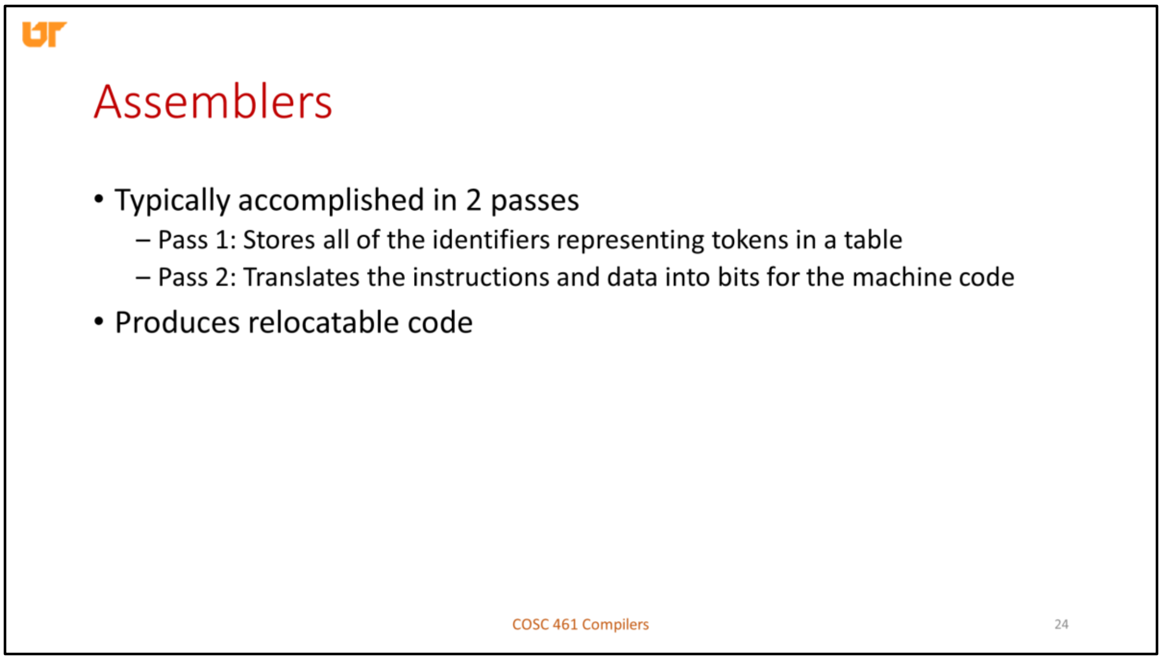
While they can make your code more efficient, macros can also be difficult to debug. It is always a good practice to surround a macro definition by parentheses so you don't have to debug errors due to unexpected precedence rules.

Show example from macros.c. max macro returns the max of the two arguments.
So, in this example, we print out the result of max(2,1) * 3.
Explain tertiary if.
Q: What is the expected output of this program?
Show program after preprocessor and ask again.

Another function of the preprocessor is file inclusion. File inclusion is often used to access header files, which are often used to define interfaces to functions.
For example:
#include "header.h"

Preprocessors also enable conditional compilation, which can be used to distinguish options that you know at compile time.
This is often used to make programs more easily portable across different operating systems or create different versions of a program for different systems.
[Go over macros1.c in examples directory.] Remember to compile with –DGOOD and –DBAD to produce different versions.

Finally, line numbering is used to indicate the line number of the source file. This is particularly useful for dealing with included files for reporting errors. (show example in line.c)
This can be useful when you're writing a program that generates code for a high-level language. For instance. yacc and lex, which we will use later in this course make use of this directive.



Assemblers

- Typically accomplished in 2 passes
  - Pass 1: Stores all of the identifiers representing tokens in a table
  - Pass 2: Translates the instructions and data into bits for the machine code
- Produces relocatable code

COSC 461 Compilers 24

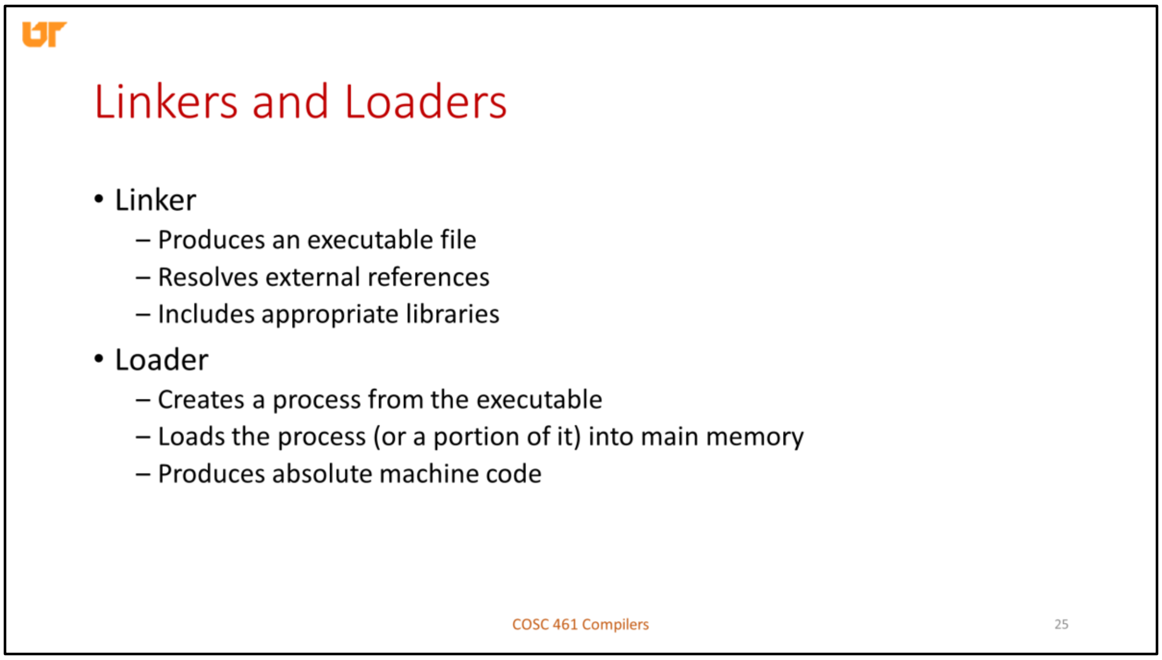
The assembler takes the assembly code produced by the compiler and produces numeric code (binary).

Assemblers typically make 2 passes through the input assembly code.

The first pass is to associate labels with relative locations. The second pass uses this information to generate the machine code that's in a binary format.

Two passes are helpful because, if you're scanning the code sequentially, branches may refer to forward labels (that is labels that have not been resolved yet). Two passes simplifies the resolution of these labels.

When we say the code is relocatable, that means it's allowed to be placed at different locations in memory and still execute correctly. The assembler puts in placeholders for addresses that cannot be resolved. The linker and loader programs are used later to resolve all the unknown addresses.



Linkers and Loaders

- Linker
  - Produces an executable file
  - Resolves external references
  - Includes appropriate libraries
- Loader
  - Creates a process from the executable
  - Loads the process (or a portion of it) into main memory
  - Produces absolute machine code

COSC 461 Compilers 25

The linker is in charge of resolving external references. This includes resolving the location of global variables and procedures.
A linker will only try to resolve references to variables and functions that have not been defined.
The linker produces an executable file.
[Go over hello.c and print.c in examples directory.]

To actually execute the program, you need a loader.
There is often confusion about linkers and loaders.
In fact, the linker on unix systems is called ld.
The main function of the loader is to create a process from the executable file.
The process will be loaded by the operating system into main memory where it can be executed.

Q. What does a process contain?
A. Go over typical layout of a process in virtual memory.
For machines with virtual memory, the locations are resolved at run time.