How GCC Compiles Code

A look at how GCC does some of its magic

Davis Claiborne

LUG @ NC State

September 4, 2019

Linux Users Group
at NC State University

What is a compiler and why do you need it?

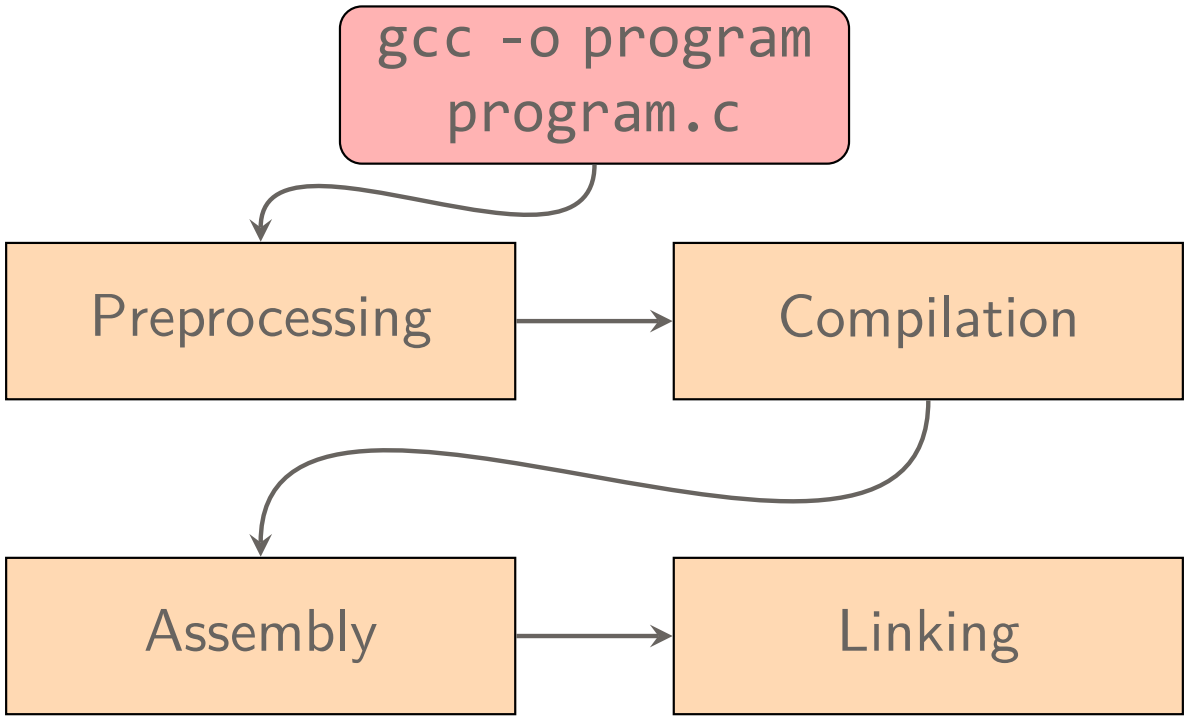
Compiler: Coverts code to format computer can interpret natively

Justification: Simpler code; portability

Program: GCC - common; widely available; system support

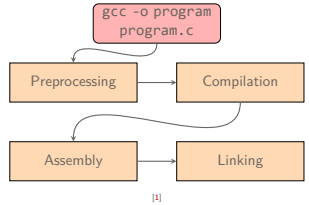
- A **compiler** is a tool that is used to translate a programming language, such as C, to a machine language that can be directly interpreted by the processor
- **Justification**: Why would you want to use a compiler?
  - If you've never written in assembly or machine code, try writing any somewhat complex program with it and that's all the justification you need - it is very complicated and tedious
  - Additionally, code can be compiled to virtually any platform or processor, instead of having to write different assembly for every target
- **Program**: We will be looking at GCC, the GNU Compiler Collection, as it is essentially the de-facto compiler for the most common platforms, and is widely available and free to use

Steps of compilation process



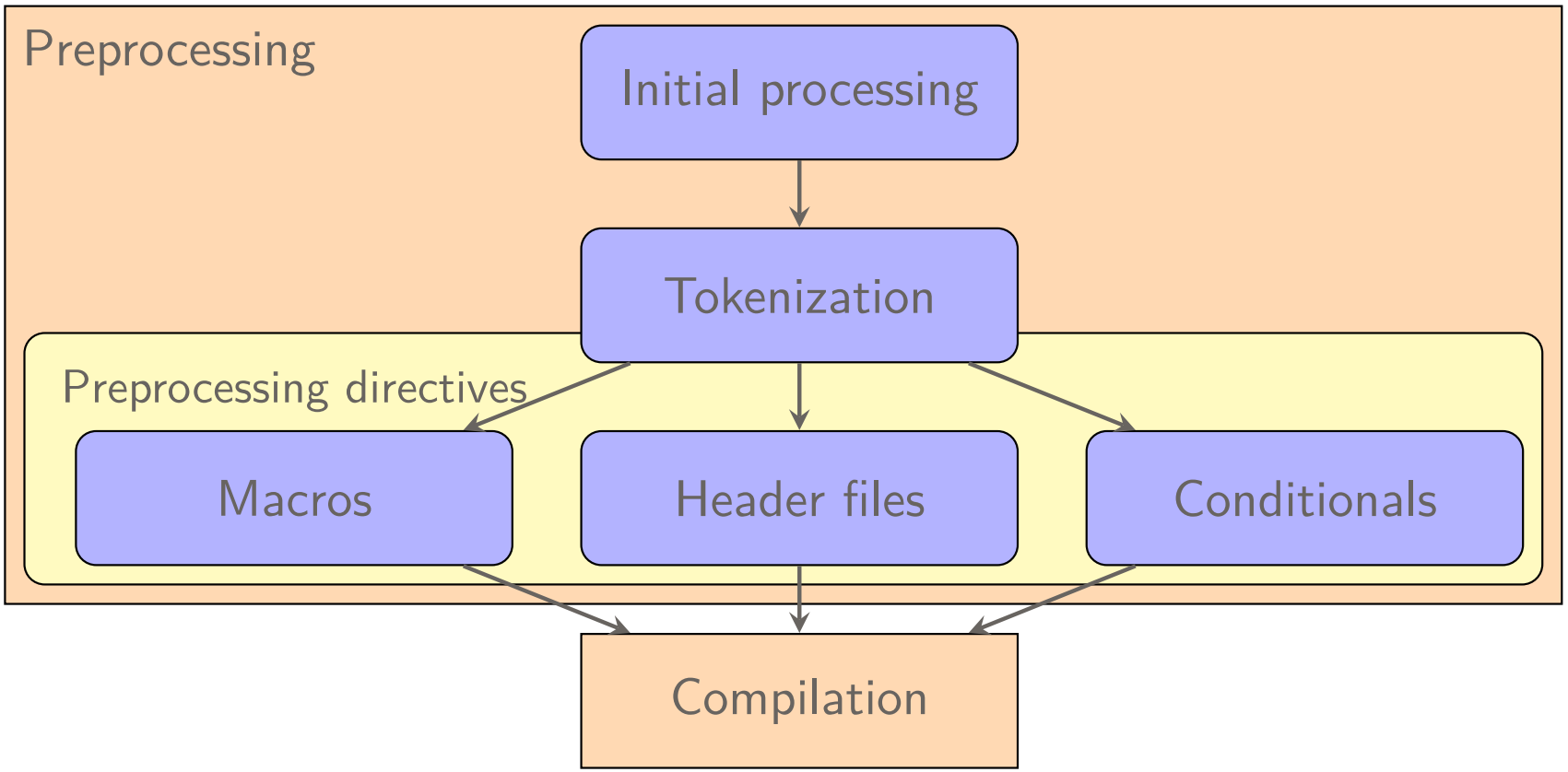
[1]

Steps of compilation process

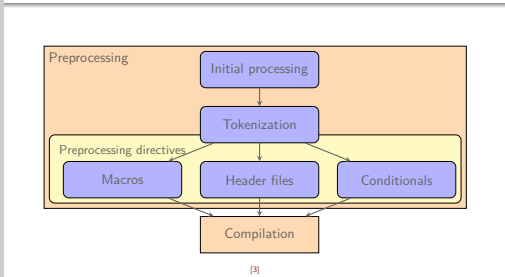


- The **preprocessing** stage is responsible for handling C macros, `#defines`, `#includes`, and other C preprocessors
- The **compilation** stage converts the preprocessed code to an intermediate representation, known as an AST, or abstract syntax tree, and performs optimizations using it
- The **assembly** stage converts the AST to an object file, which typically contains assembly code
- The **linking** stage essentially adds functions from external files and rearranges the assembly code so that functions are declared before they are used
- In this presentation, I will cover the first two stages.

Steps of preprocessing process

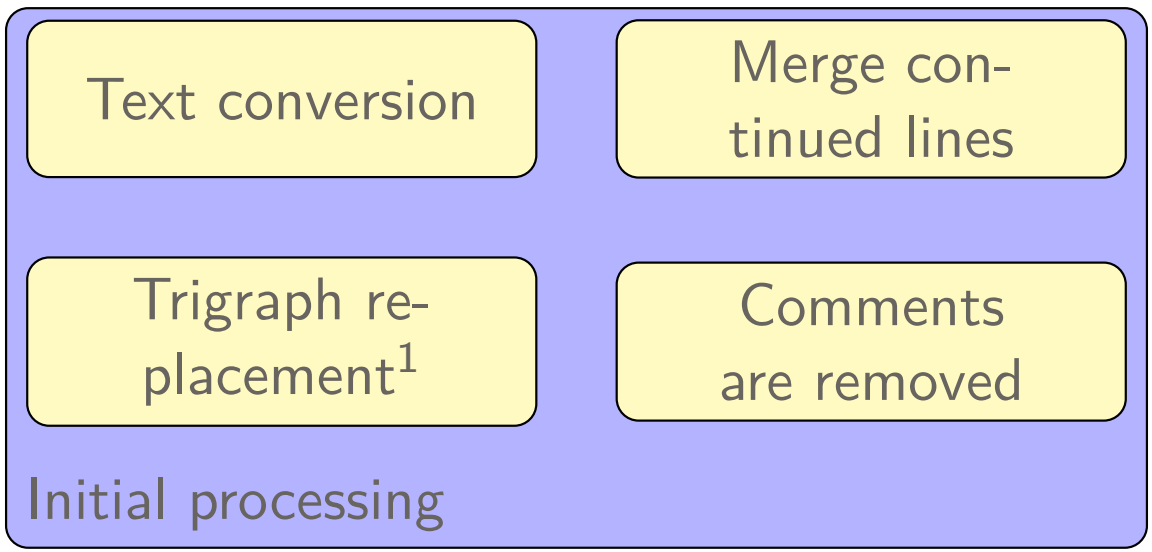


[3]



- The first step of the preprocessor is known as **initial processing**, which takes care of text encoding as well as several other basic tasks
- The next step is **tokenization**, which is where the compiler begins to split the raw text of the source file into discrete sections which can be operated on by preprocessor directives
- **Macros**, which are specified by `#define` calls, can be one of two things: plain text replacements, or function-like
- **Header files** are handled by `#include` - this preprocessor essentially just prepends the header mentioned to top of the file
- **Conditionals** are used to specify parts of code that may or may not be included in the compilation process by using `#ifdef` or `#if`, which will evaluate an expression, then include the code if that condition is true
- One thing to understand about the preprocessing step of compilation is that none of these steps are particularly syntax aware - they just do simple text replacement and operations

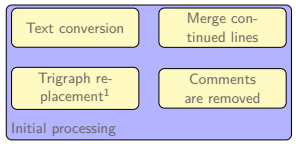
Initial processing



[3, sections 1.1 and 1.2]

[1] The following trigraphs are defined:

Trigraph	??(	??)	??<	??>	??=	??/	??'	??!	??-
Replacement	[	]	{	}	#	\	^	|	~



[3, sections 1.1 and 1.2]

[1] The following trigraphs are defined:

Trigraph	??(	??)	??<	??>	??=	??/	??'	??!	??-
Replacement	[	]	{	}	#	\	^	|	~

- During the **text conversion** portion of initial processing, the compiler attempts to convert the provided C file into UTF-8
- The compiler also takes care of **trigraphs**, which are three character sequences, starting with two question marks, that represent specific characters that are used in C but were not present on some very old computers
  - GCC actually ignores trigraphs by default since they are not widely supported by most compilers and can get you into trouble if you're not careful, but you can enable them with the `-std` or `-trigraph` options if desired
- Additionally, lines ending with a backslash are `merged` with the line following it
  - Technically speaking, lines can be continued anywhere, not just at white space, since no space is added
- Finally, all **comments** are removed and replaced with a single space

Tokenization

Two main parts to tokenization:

- Grouping
- Digraph replacement

Digraphs are similar to trigraphs, but are safer and do less

Digraph	<%	>%	<:	:>	%:	%:%:
Replacement	]	]	{	}	#	##

Why digraphs are safer than trigraphs:

```
// Will this work???/
do_magic_please()
```

becomes

```
// Will this work?\
do_magic_please()
```

If using trigraph replacement

[3, sections 1.3]

Tokenization

Two main parts to tokenization:

- Grouping
- Digraph replacement

Digraphs are similar to trigraphs, but are safer and do less

Digraph	<%	>%	<:	:>	%:	%:%:
Replacement	{	}	[	]	#	##

Why digraphs are safer than trigraphs:

```
// Will this work???/
do_magic_please()
```

becomes

```
// Will this work?\
do_magic_please()
```

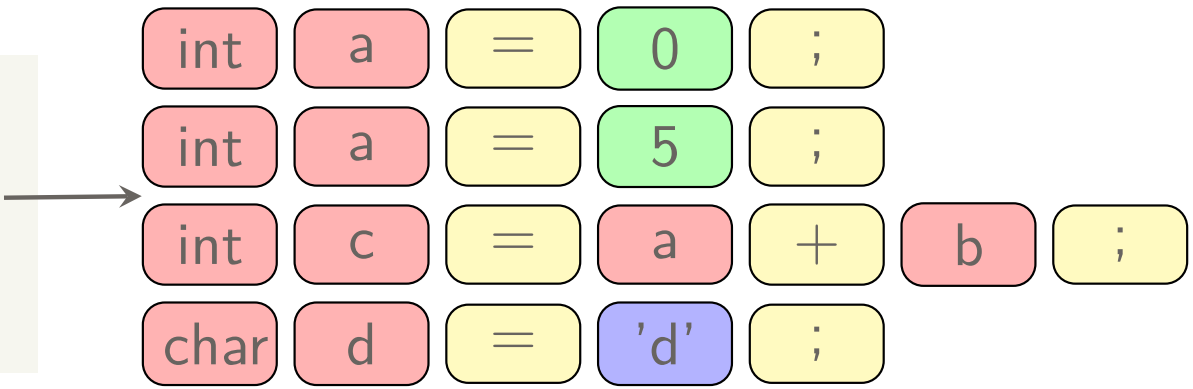
If using trigraph replacement

[3, sections 1.3]

- **Tokenization**, also known as lexical analysis, is a step of the compilation process where the text from the program is split up into various groupings.
- These **groupings** allow the compiler to be somewhat syntax aware, as the compiler groups the tokens into five broad categories: item identifiers, preprocessing numbers, string literals, punctuators, and other
- These groupings allow the compiler to treat different types differently - for instance, macro replacement does not occur within strings
- In the case of ambiguities, the tokenizer is *greedy*, meaning that it will try to match as much as possible first - this can sometimes result in syntax errors or undesired code
- Additionally, **digraph** conversion occurs during tokeniztion
  - Although digraphs accomplish less than trigraphs, they are safer to use than trigraphs: since conversion happens during tokenization, character grouping has already occurred, so lines can't accidentally get commented out like you see here

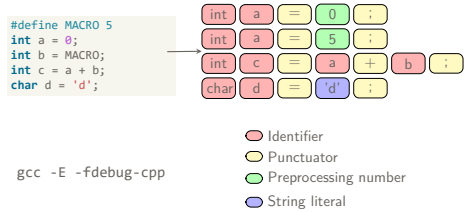
Examples of tokenization

```c
#define MACRO 5
int a = 0;
int b = MACRO;
int c = a + b;
char d = 'd';
```




```
gcc -E -fdebug-cpp
```

- Identifier
- Punctuator
- Preprocessing number
- String literal



- Here you can see a simple example of how a program was tokenized
- As you can see, C keywords are not special to the tokenizer
  - This is because C keywords can be used as macros, which can be useful - for instance, you could `#define const` to remove `const` from your code if using an older compiler that doesn't support it
  - The only exception to this rule is the `defined` preprocessor operator, in which case the compiler will throw an error during the preprocessing phase if used in a macro
- Astute observers may notice that the "other" grouping is not visible - only @, $, `, and control characters other than null fall into this category
- Null is treated as white space and warned about
- If you're curious how your code has been tokenized, you can use the `-fdebug-cpp` flag to see where the text has been broken up

The preprocessing language

Jobs of the preprocessing language:

- Header files
  - `#include <file.h>`
  - `#include "file.h"`
- Macro expansion
  - `#define TEXT val`
  - `#define FUNC( args ) body`
- Conditionals (`#if`, `#ifdef`, etc.)
- Other less commonly used directives

[3]

The preprocessing language

Jobs of the preprocessing language:

- Header files
  - `#include <file.h>`
  - `#include "file.h"`
- Macro expansion
  - `#define TEXT val`
  - `#define FUNC( args ) body`
- Conditionals (`#if`, `#ifdef`, etc.)
- Other less commonly used directives

[3]

- Before the compilation stage can happen, translation must occur
- One of the most simple preprocessor instructions is the `#include` command, which takes a single argument, the **header** file to include
  - The use of angled brackets indicates that the specified file is a system file, specified directories are searched through for it
  - Quotes indicate that the file is a user-defined file, so the preprocessor first looks for the specified file in the current directory, then in the specified "quote" directories, then finally in the same directories as it looks for system files

The preprocessing language

Jobs of the preprocessing language:

- Header files
  - `#include <file.h>`
  - `#include "file.h"`
- Macro expansion
  - `#define TEXT val`
  - `#define FUNC( args ) body`
- Conditionals (`#if`, `#ifdef`, etc.)
- Other less commonly used directives

[3]

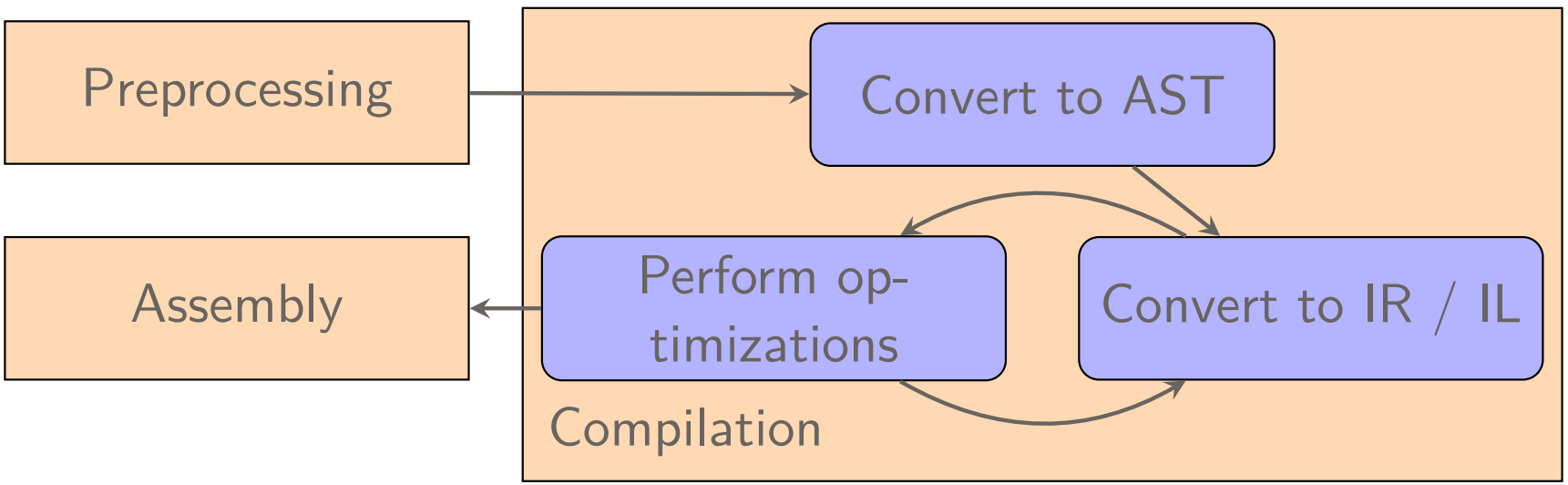
The preprocessing language

Jobs of the preprocessing language:
- Header files
  - `#include <file.h>`
  - `#include "file.h"`
- Macro expansion
  - `#define TEXT val`
  - `#define FUNC( args ) body`
- Conditionals (`#if`, `#ifdef`, etc.)
- Other less commonly used directives

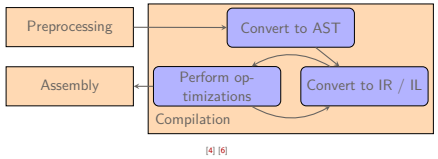
[3]

- Another task is **macro** expansion, which takes two forms:
  - The first variation acts essentially like a find and replace that the preprocessor performs
  - The second variation is a bit more complex, as it has the ability to take "parameters"
- **Conditionals** are the last commonly-used directive - these can evaluate statements containing expressions composed solely of macros - since, remember, the preprocessor has no real knowledge of C's syntax, so it can't evaluate variables, even constants
- There are other less common directives, such as `warning`, `error`, etc., though they are rarely used in most applications

Steps of compilation process



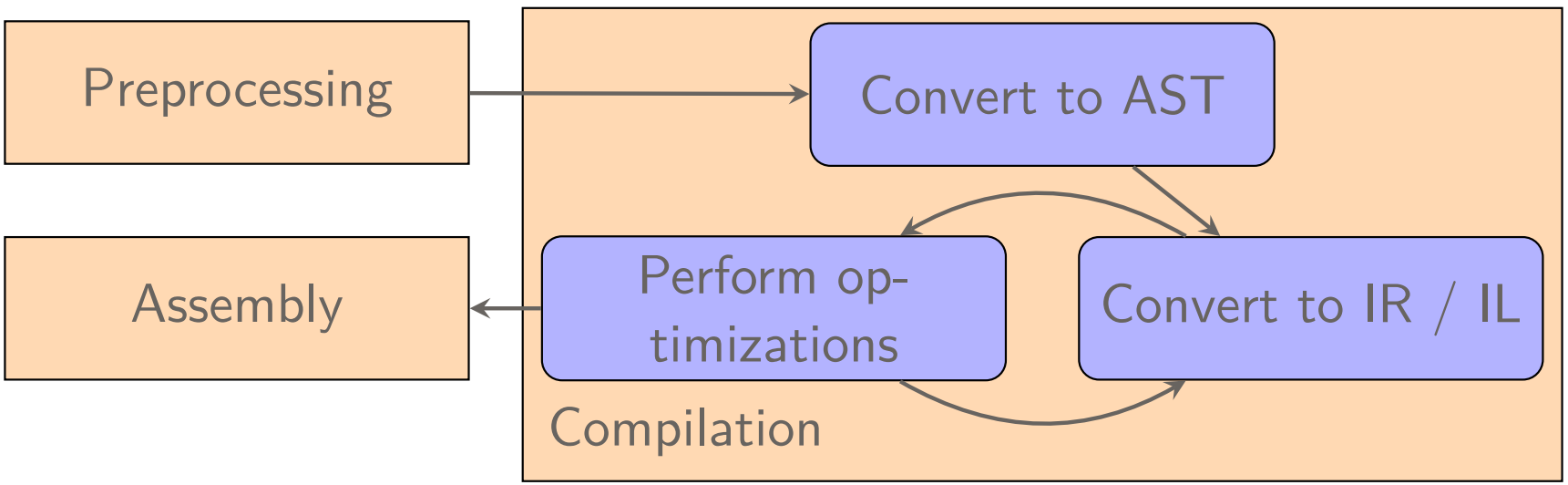
[4] [6]



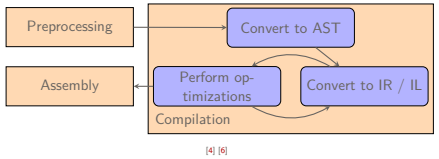
[4] [6]

- After the processing phase, the compiler moves on to **compilation**
- During compilation, the preprocessed tokens are parsed in order to be converted into an AST
- The **AST**, or abstract syntax tree, is used to check for syntax errors in the code, as well as generate the list of variables used in the program
- The AST is then used to generate an **IR**, or intermediate representation, also known as an **IL**, or intermediate language
  - While technically speaking, the AST is a type of IR, the distinction is made here to distinguish between the two phases of syntax checking and optimization

Steps of compilation process



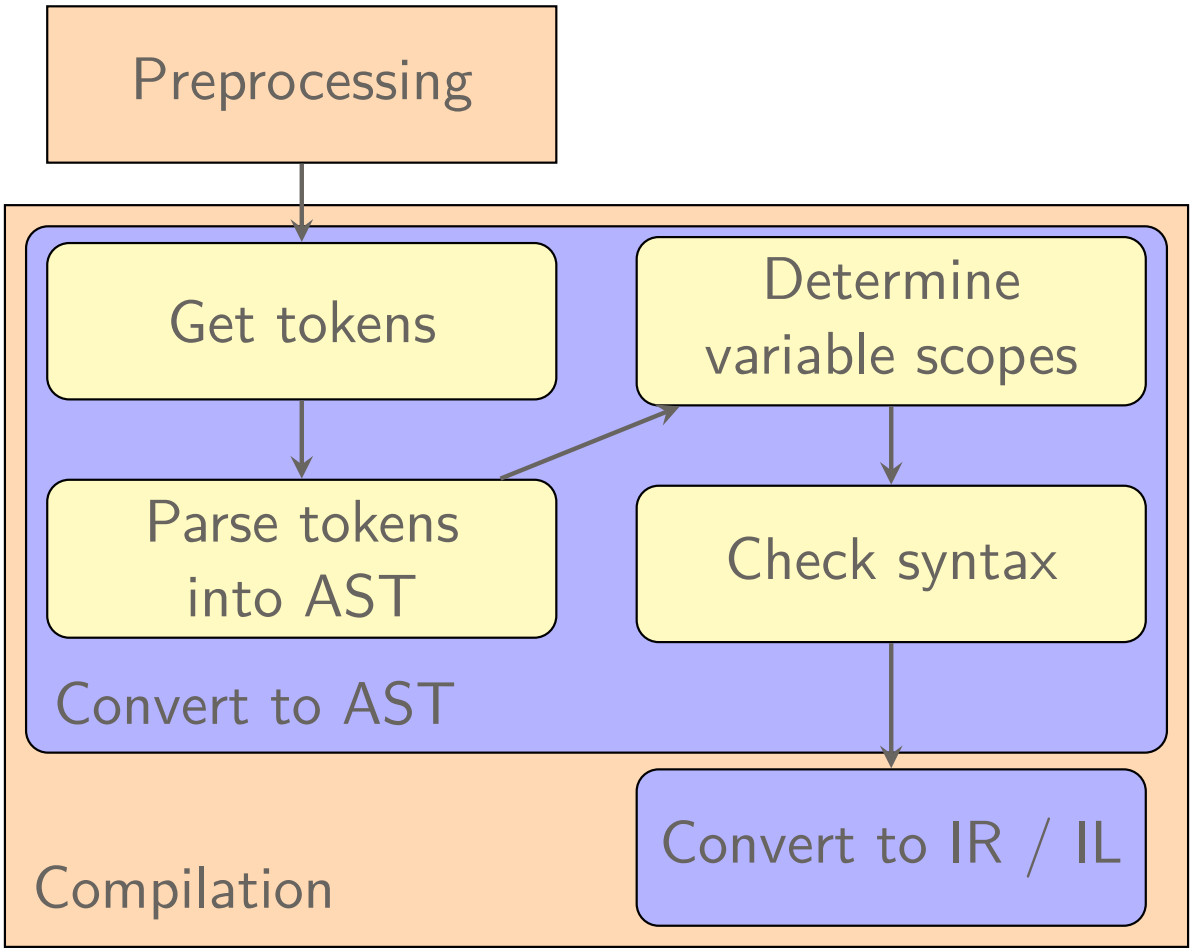
[4] [6]



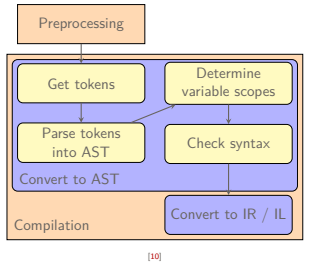
[4] [6]

- You may be wondering, what is an **IR** and why is it used?
  - IRs are essentially special languages that the compiler uses when generating code
  - They're used for two main reasons:
    - They are specifically designed so that the code can more easily be operated on, i.e. analyzed for optimizations, syntax errors, variables, etc.
    - So that platform-specific optimizations only have to be written once, based on the IR, instead of having to be written for each supported language
- As you can see, the code may be converted to an IR several times, as different IRs are better suited for different optimizations

Generating the AST



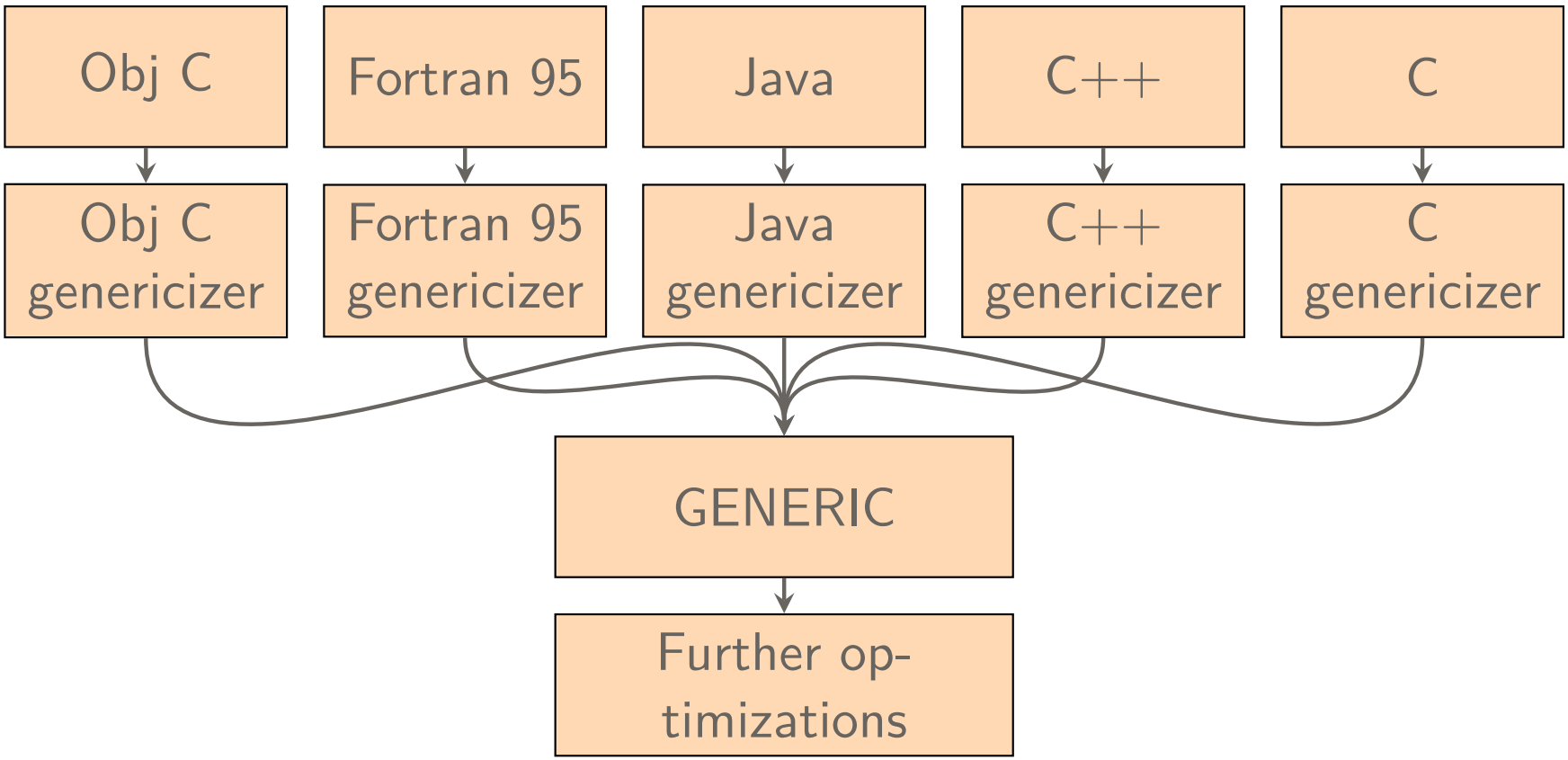
[10]



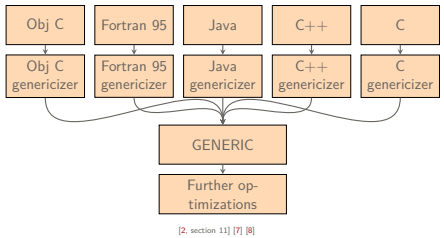
[10]

- The ordered, loosely-typed tokens generated during the preprocessing stage and arranged into an AST
- Next, each variable's scope is determined
  - The scope of each variable is required for later optimizations, as well to ensure that variables are defined in-scope
- Finally, the AST is analyzed in order to ensure syntax is correct
  - Correct number of arguments given to functions
  - Variable types are correct
  - Lvalues and rvalues are all valid
  - All variables are accessible from current scope
- Assuming there are no syntax errors, the AST is then converted to an IR for optimization

Some comments on the AST



[2, section 11] [7] [8]



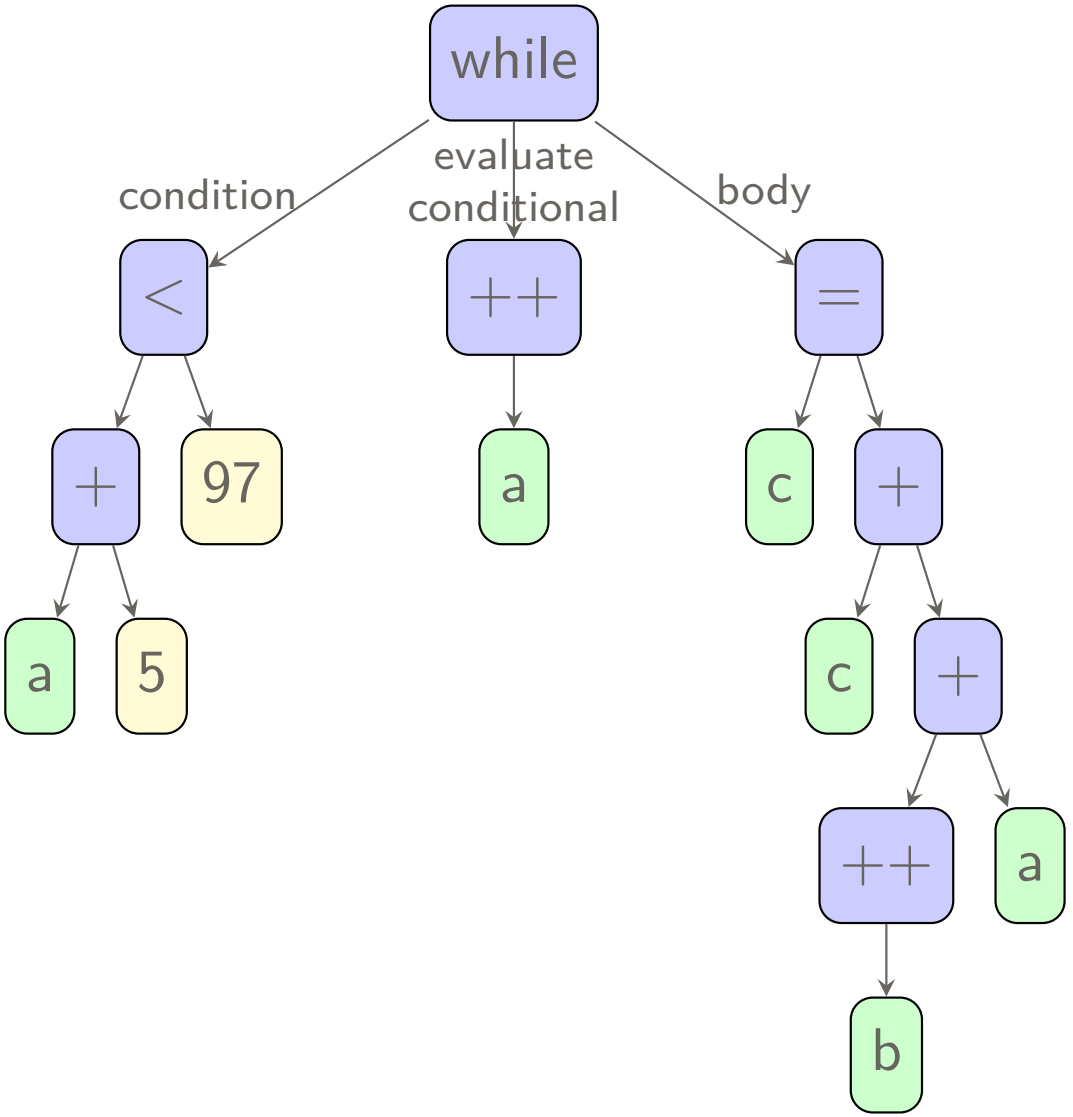
[2, section 11] [7] [8]

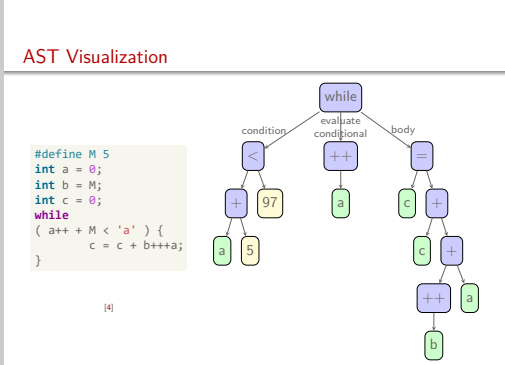
- The AST that gcc uses is intended to be language independent - it must be general enough such that any language being compiled can be represented by it
- With that in mind, the AST is fittingly called "`GENERIC`"
- Remember how each IR best suites a specific optimization? The way `GENERIC` represent code is best suited for "name space abstraction" of variables.
- Name space abstraction is the name given to how the compiler deals with the scope of variables - the scope and use of each variable influences how the compiler makes use of registers later on
- Technically speaking, no optimization actually goes on here for C code, since for C it is actually less efficient to optimize here, but it is still used as an intermediate step for generating the next IR

AST Visualization

```
#define M 5
int a = 0;
int b = M;
int c = 0;
while
( a++ + M < 'a' ) {
        c = c + b+++a;
}
```

[4]





- Here you can see an example of what a line would look like after having its tokens parsed and arranged into an AST
- You may have noticed there was some ambiguity in how the code could have been interpreted - b+++a could become b++ + a or b + ++a
- Because the tokenizer is greedy, the shown interpretation is correct
- There are many ways to parse the tokens and reach such an AST - one of the most common and easy to understand is through using Dijkstra's "shunting-yard" algorithm
- This algorithm includes most aspects of constructing the AST, including operator precedence and associativity
  - Associativity refers to how to determine order of operations for operators with equal precedence

GENERIC goes to GIMPLE

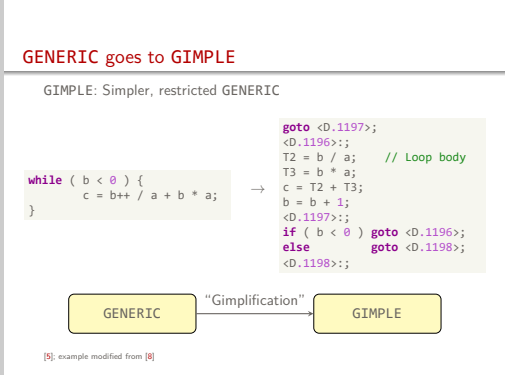
GIMPLE: Simpler, restricted GENERIC

```
while ( b < 0 ) {
        c = b++ / a + b * a;
}
```

→

```
goto <D.1197>;
<D.1196>:;
T2 = b / a;     // Loop body
T3 = b * a;
c = T2 + T3;
b = b + 1;
<D.1197>:;
if ( b < 0 ) goto <D.1196>;
else         goto <D.1198>;
<D.1198>:;
```

[5]; example modified from [8]



- The next step in compilation is converting the AST, represented with GENERIC, to the next IR, which is known as GIMPLE
- GIMPLE is a subset of GENERIC with one important distinction - all commands are "three address" commands, meaning that each GIMPLE command is guaranteed to reference three or fewer memory locations
- Additionally, some control flow structures, such as while-loops, are converted to more primitive structures
- The process of converting to GIMPLE is known as "gimplification."
- The reason that this is done is because it more closely resembles the operations that the machine code will be doing. Using a generic machine code-like language makes performing optimizations simpler and easier to do.
- The T2 and T3 seen in the above GIMPLE code are referred to as "expression temporaries" and are used to attempt to eliminate redundant calculations

Optimizing GIMPLE - Step 1: Control flow

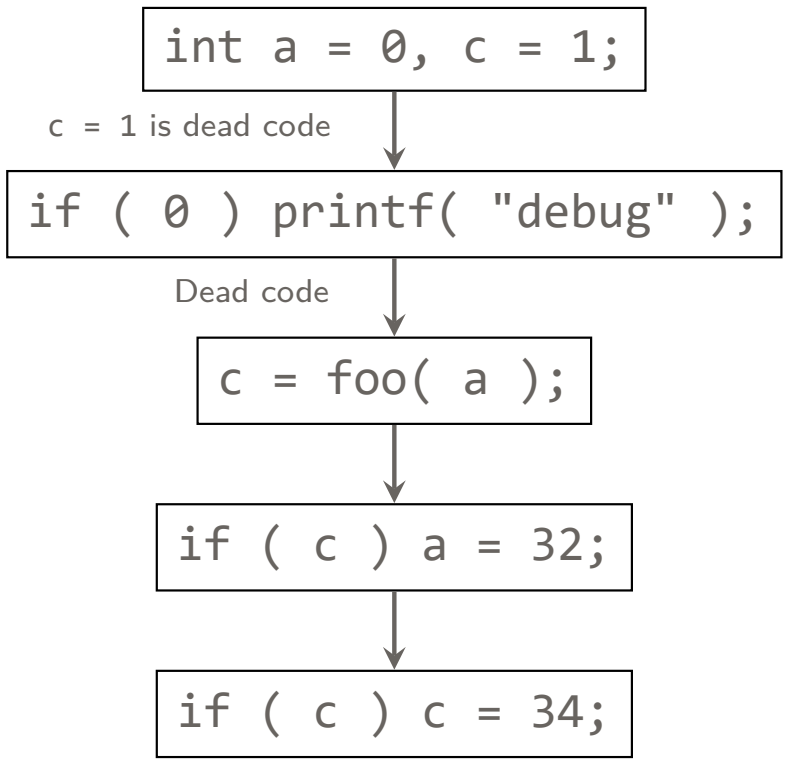
```c
#define DEBUG 0
int a = 0,
    c = 1;

if ( DEBUG ) {
        printf( "debug" );
}

c = foo( a );

if ( c ) {
        a = 32;
}

if ( c ) {
        c = 34;
}
```



[2, section 15] [8] [9]

Optimizing GIMPLE - Step 1: Control flow

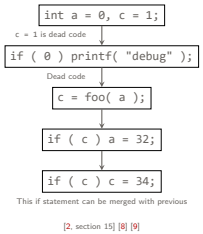
```
#define DEBUG 0
int a = 0,
    c = 1;

if ( DEBUG ) {
        printf( "debug" );
}

c = foo( a );

if ( c ) {
        a = 32;
}

if ( c ) {
        c = 34;
}
```



[2, section 15] [8] [9]

- The first step to optimizing GIMPLE code is to construct a control flow graph
- By constructing a control flow graph, it is easier to detect “dead” code that is never executed or is never used
- Additionally, the compiler can perform several test runs of the function to get usage information about the function, such as how many times each block was run, as well as how frequently a block of code was run relative to the rest of the blocks
  - This information can be used to guide decisions that the compiler makes regarding optimizing speed over space - for instance, if a block of code is run very many times, it is probably more important to optimize its speed
- For example, here you can `c = 1` is considered dead - this is because, assuming it’s not used in `foo`, it is automatically overwritten by the value returned by `foo`.

Optimizing GIMPLE - Step 2: SSA

C Code

```
c = 3;
c++;
c = 5;
```

GIMPLE code - post-cfg

```
c_1 = 3;
c_2 = c_1 + 1;
c_3 = 5;
```

Optimizing GIMPLE - Step 2: SSA

C Code

```
c = 3;
c++;
c = 5;
```

GIMPLE code - post-cfg

```
c_1 = 3;
c_2 = c_1 + 1;
c_3 = 5;
```

- You may be wondering, "How can does the control flow graph identify dead code?"
- The main way is through SSA, or "Single Static Analysis"
- The GIMPLE code is changed so that each variable only has one assignment - if a variable has multiple assignments, it is indicated using an underscore
- The compiler then analyzes each variable assignment to determine if it is necessary or not

References I

General:

[1] The Four Stages of Compiling a C Program
https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/

[2] GNU Compiler Collection (GCC) Internals https://gcc.gnu.org/onlinedocs/gccint/

Preprocessor:

[3] The C Preprocessor https://gcc.gnu.org/onlinedocs/cpp/

Compilation:

[4] AST representation in GCC http://icps.u-strasbg.fr/~pop/gcc-ast.html

[5] GENERIC and GIMPLE: A New Tree Representation for Entire Functions
https://ols.fedoraproject.org/GCC/Reprints-2003/jason.pdf

[6] GCC Translation Sequence and Gimple IR
https://www.cse.iitb.ac.in/~uday/courses/cs715-09/gcc-gimple.pdf

[7] The Conceptual Structure of GCC
http://www.cse.iitb.ac.in/grc/intdocs/gcc-conceptual-structure.pdf

[8] Tree SSA - A New Optimization Infrastructure for GCC ftp://gcc.gnu.org/pub/gcc/summit/2003/Tree%20SSA%20-%20A%20New%20optimization%20infrastructure.pdf

[9] Introduction to Compilers - Lecture 24: Control Flow Graphs
https://www.cs.cornell.edu/courses/cs412/2008sp/lectures/lec24.pdf

[10] New C Parser https://gcc.gnu.org/wiki/New_C_Parser