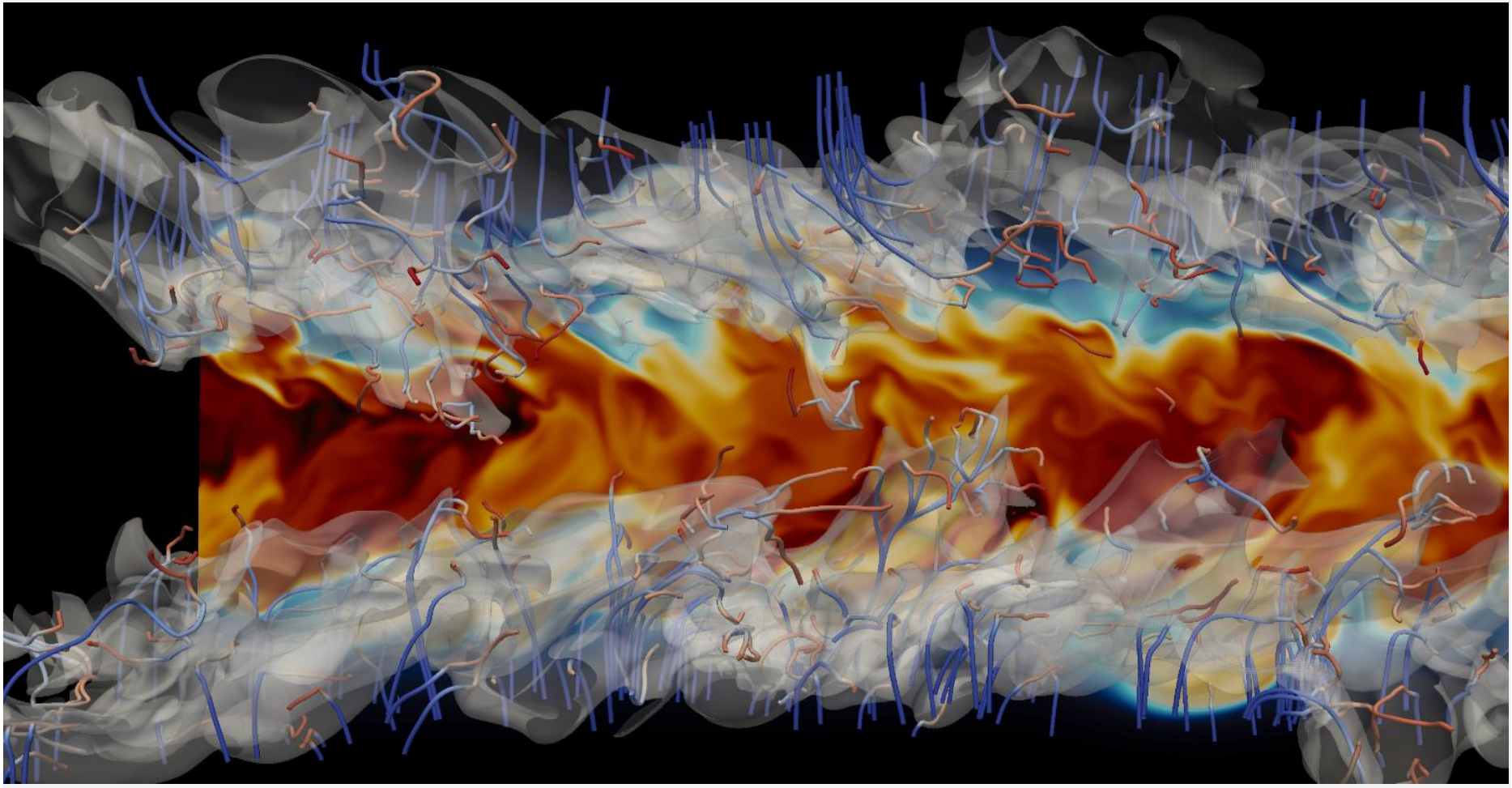
Software Tools for UNIX/Linux Systems

Part 7: Compiler

C. Hasse

TECHNISCHE UNIVERSITÄT DARMSTADT

STFS



Agenda

1 **Introduction**

2 **Preprocessor**

3 **Compiler**

4 **Assembler**

5 **Libraries and linker**

Disclaimer

The examples discussed during this session are based on the

GNU Compiler Collection (GCC).

The concepts presented are valid for other compilers as well (usually commercial) e.g. Intel, PGI, Borland.

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

Source: en.wikipedia.org/wiki/Compiler

TECHNISCHE UNIVERSITÄT DARMSTADT

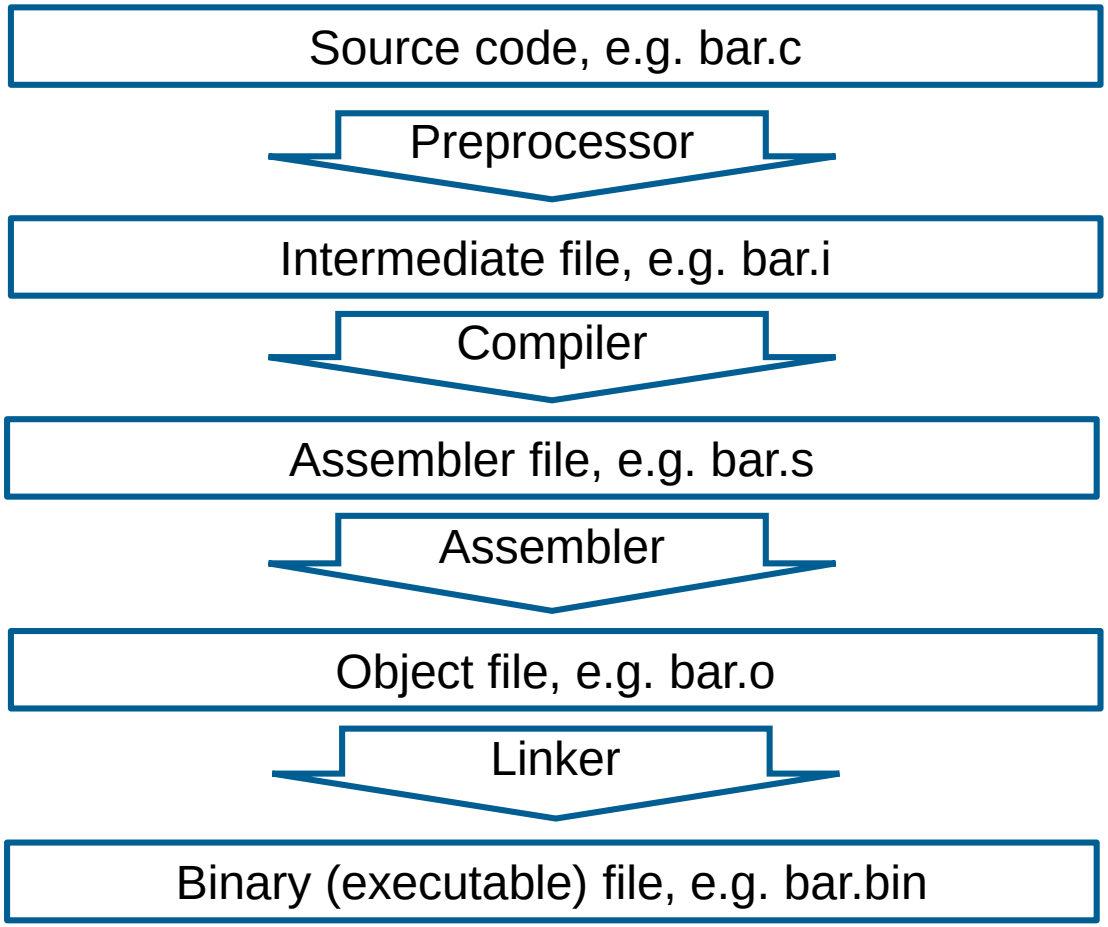
$> cat introduction01.c

```
1   #include <stdio.h>
2
3   int main()
4   {
5       printf( "hello world!\n" );
6       return 0;
7   }
```

$> gcc introduction01.c
$> ./a.out

Compiler

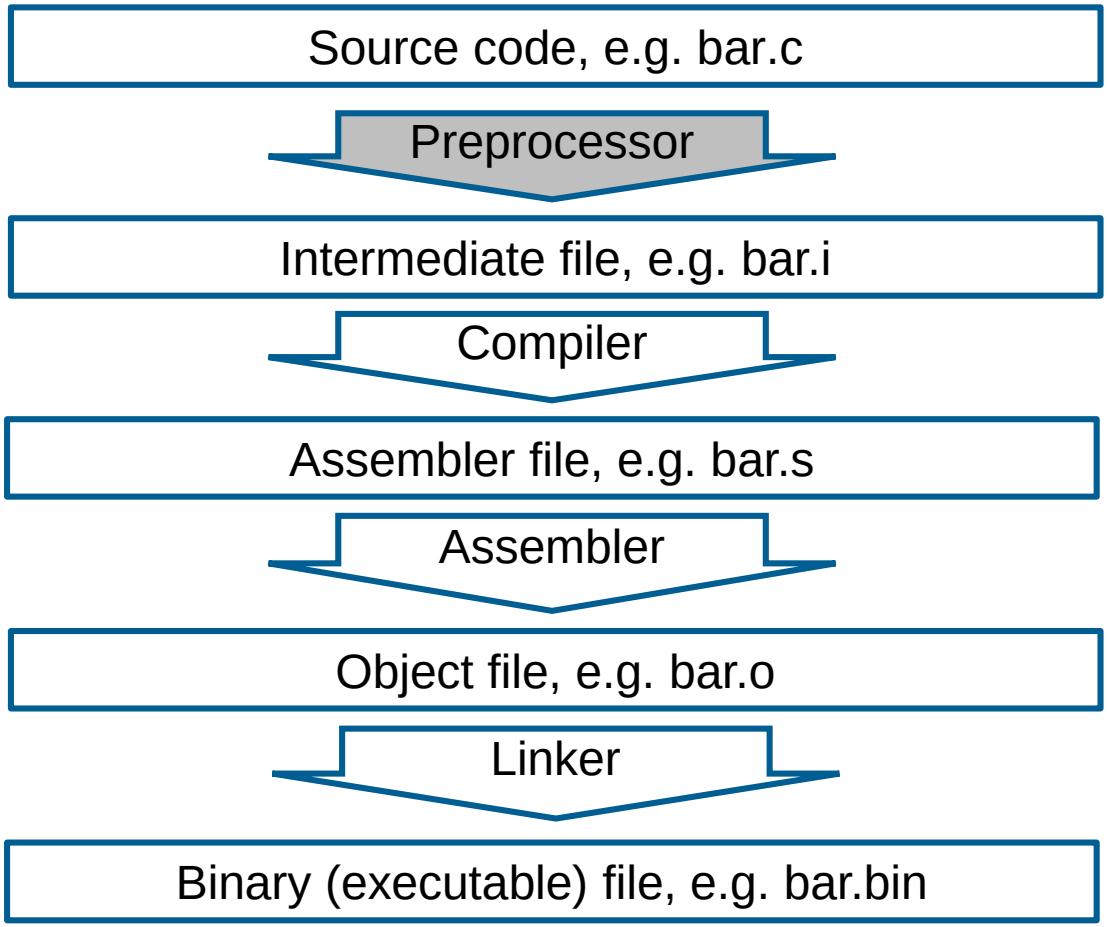
Flag	Meaning
-o FILE	Specify output FILE (other than "a.out")
-c	Compile only
-ON	Optimize (level N)
-g	Add debug information
-p	Add profiler information
-I PATH	Search PATH for include files
-L PATH	Search PATH for libraries
-l LIB	Link against LIB
-E	Run preprocessor only
-save-temps	Save any intermediate files


Source code, e.g. bar.c
Preprocessor
Intermediate file, e.g. bar.i
Compiler
Assembler file, e.g. bar.s
Assembler
Object file, e.g. bar.o
Linker
Binary (executable) file, e.g. bar.bin

Lets have a look at the intermediate steps:

```
$>  gcc -save-temps introduction01.c
$>  ls
```

Preprocessor



Preprocessor

TECHNISCHE UNIVERSITÄT DARMSTADT

For the following examples we want to run the preprocessor only:

```
$>  gcc -E FILENAME
```

This way we will see the output of the preprocessor directly.

TECHNISCHE UNIVERSITÄT DARMSTADT

#define

Creates a macro – a global constant or function to reduce code size and improve readability.

```
$>  cat preprocessor01.c
```

```
1   #define M_PI 3.1415
2   #define Y(X) X*X;
3
4   double myX = M_PI * 2;
5   double myResult = Y(myX);
```

```
$>  gcc -E preprocessor01.c
```

TECHNISCHE UNIVERSITÄT DARMSTADT

#include

Replaces line with files contents. Improves code reusability.

```
$>  cat preprocessor02.c
```

```
1   double first = 1;
2   #include "preprocessor02.h"
3   double third = 3;
4   double myX = M_PI;
```

```
$>  cat preprocessor02.h
```

```
1   #define M_PI 3.1415
2   double second = 2;
```

```
$>  gcc -E preprocessor02.c
```

TECHNISCHE UNIVERSITÄT DARMSTADT

#ifdef #ifndef #eldef

Checks whether macro was or was not defined already. Excessively used to avoid double including files.

```
$>  cp preprocessor02.h preprocessor03.h
$>  cat preprocessor03.c
```

```
1   #include "preprocessor03.h"
2   #include "preprocessor03.h"
```

```
$>  gcc -E preprocessor03.c
```

$> cat preprocessor03.h

```
1  #ifndef COMPILER03_H
2  #define COMPILER03_H
3  #define M_PI 3.1415
4  double second = 2;
5  #endif
```

$> gcc -E preprocessor03.c

#if #ifn #else #elif

Compare defined values to toggle code segments on/off

$> cat preprocessor04.c

```
1   #define DEBUG 4
2
3   #if DEBUG > 2
4      printf( "DEBUG mode is verbose" );
5   #elif DEBUG > 0
6      printf( "DEBUG mode is on" );
7   #else
8      printf( "DEBUG mode is off" );
9   #endif
```

$> gcc –E compiler05.c

TECHNISCHE UNIVERSITÄT DARMSTADT

Remark:

#define can be passed via command line as well:

```
$>  cat preprocessor04.c
```

```
1   #if DEBUG > 2
2      printf( "DEBUG mode is verbose" );
3   #elif DEBUG > 0
4      printf( "DEBUG mode is on" );
5   #else
6      printf( "DEBUG mode is off" );
7   #endif
```

```
$>  gcc -E -D DEBUG=2 compiler05.c
```

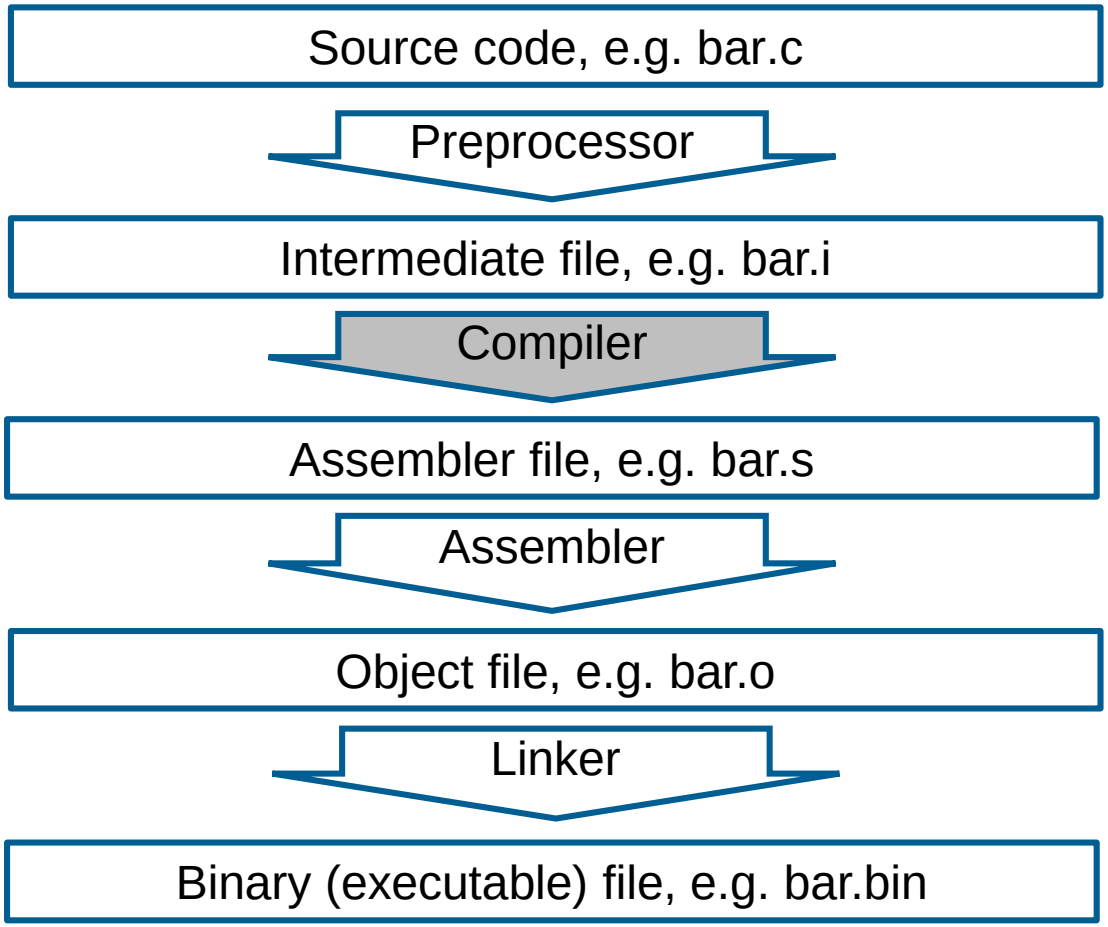
TECHNISCHE UNIVERSITÄT DARMSTADT

$> cat preprocessor05.c

```
1   #define GREETER "hello world"
2   #define NL "\n"
3   #define NUMBER 3.1415
4
5   printf(GREETER NL );
6   double myNumber = NUMBER;
```

$> gcc -E compiler06.c

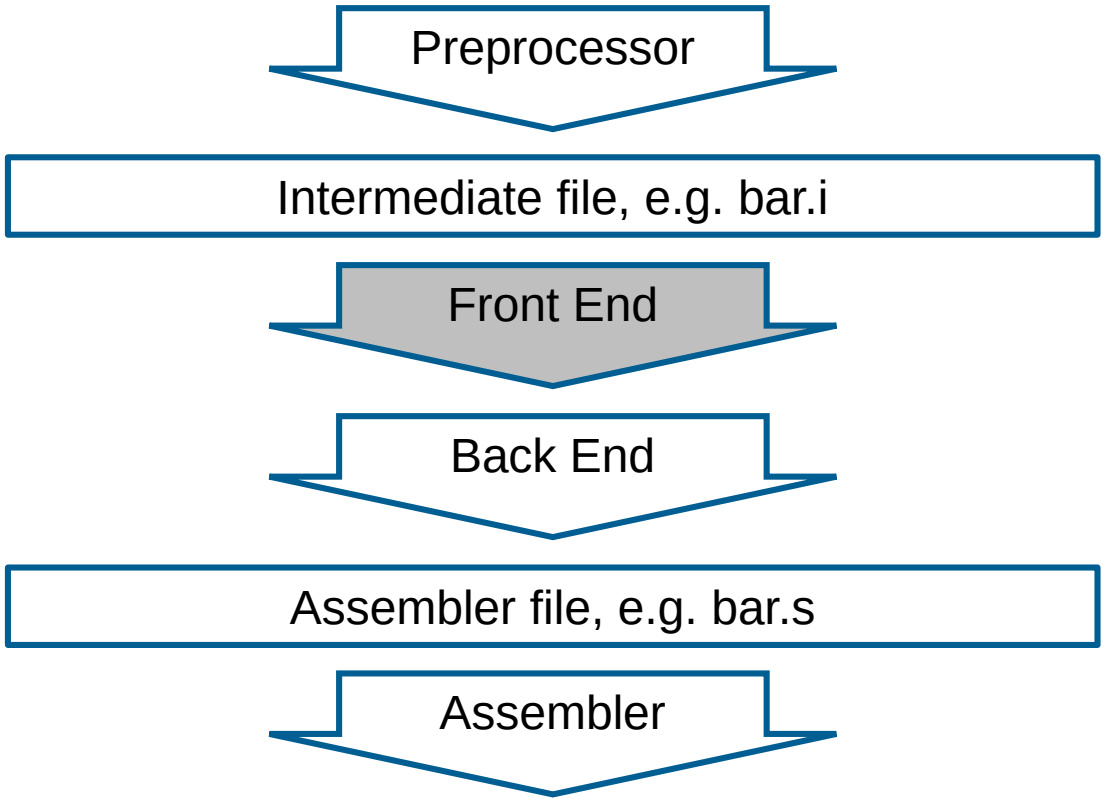
Compiler


Source code, e.g. bar.c
Preprocessor
Intermediate file, e.g. bar.i
Compiler
Assembler file, e.g. bar.s
Assembler
Object file, e.g. bar.o
Linker
Binary (executable) file, e.g. bar.bin

The compiler accomplishes different tasks:

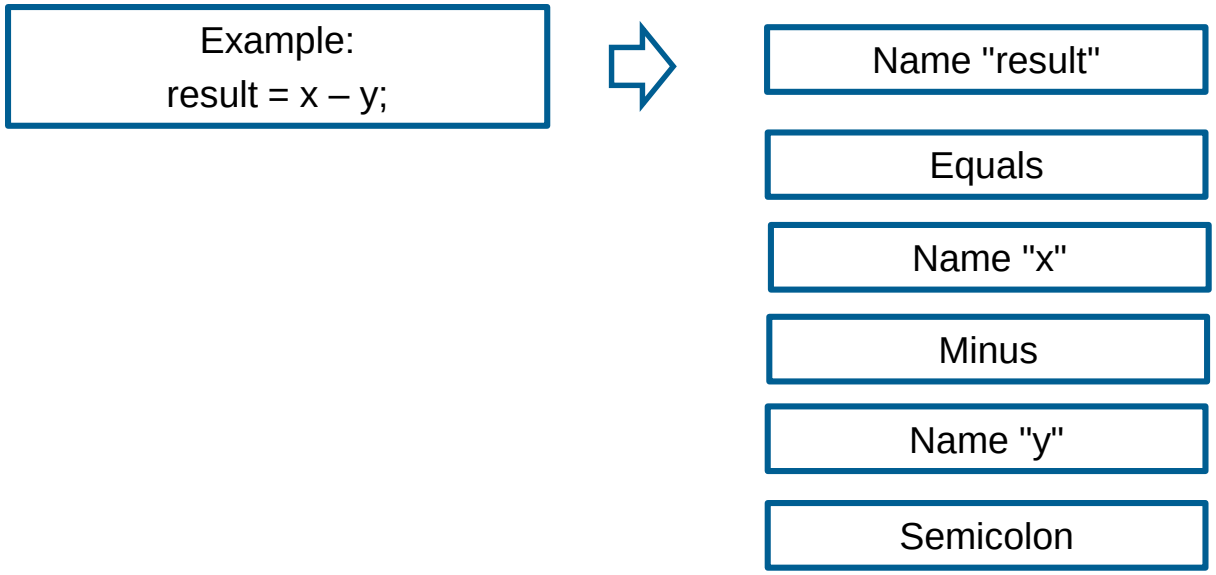
- Front end
  - Lexical analysis
  - Syntactic analysis
  - Semantic analysis
- Back end
  - Code analysis (internal)
  - Code optimization
  - Code generation (assembler)

Compiler


Preprocessor
Intermediate file, e.g. bar.i
Front End
Back End
Assembler file, e.g. bar.s
Assembler

Lexical analysis:

Convert sequence of characters (your code) into a sequence of tokens.



Example: lexical analysis

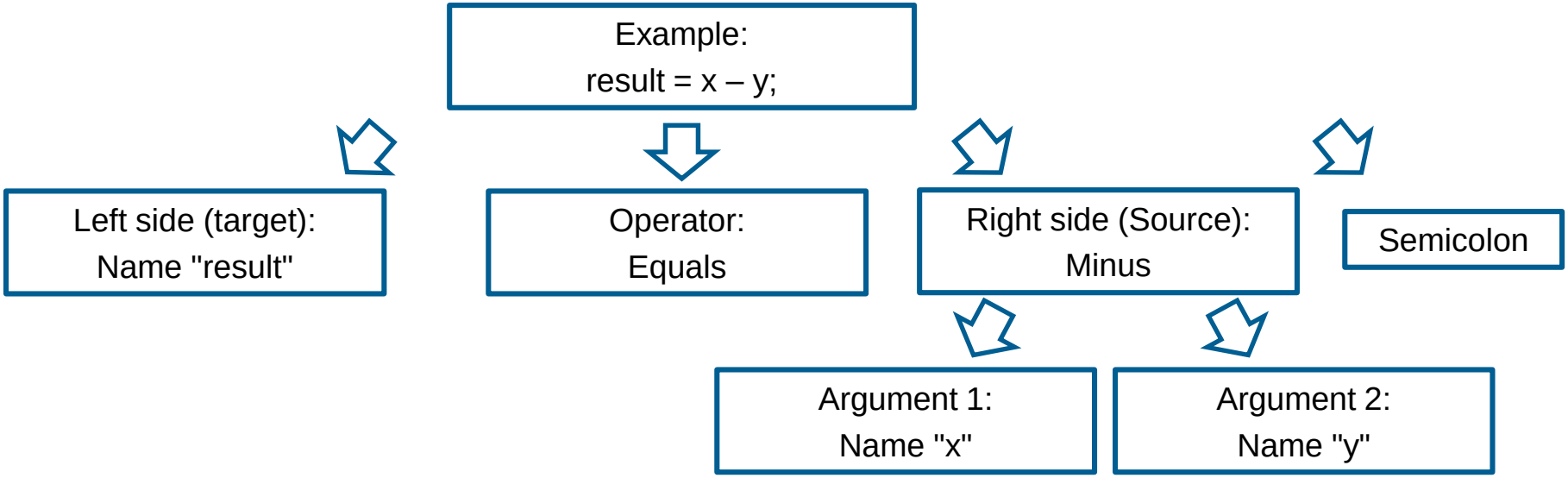
```
$>  cat lexical01.c

1   #include stdio.h
2
3   int main()
4   {
5      printf( "%f\n", 1e+% );
6      return 0;
7   }

$>  gcc -Wall lexical01.c
```

Syntactic analysis:

Analyzing the code according to the rules of the language and resolving the syntactic relation of the constituents.



Syntactic analysis:

I *is* going to the concert tonight.

Incorrect grammar but still understandible for english speakers – compilers are not as forgiving as humans…

Example: syntactic analysis

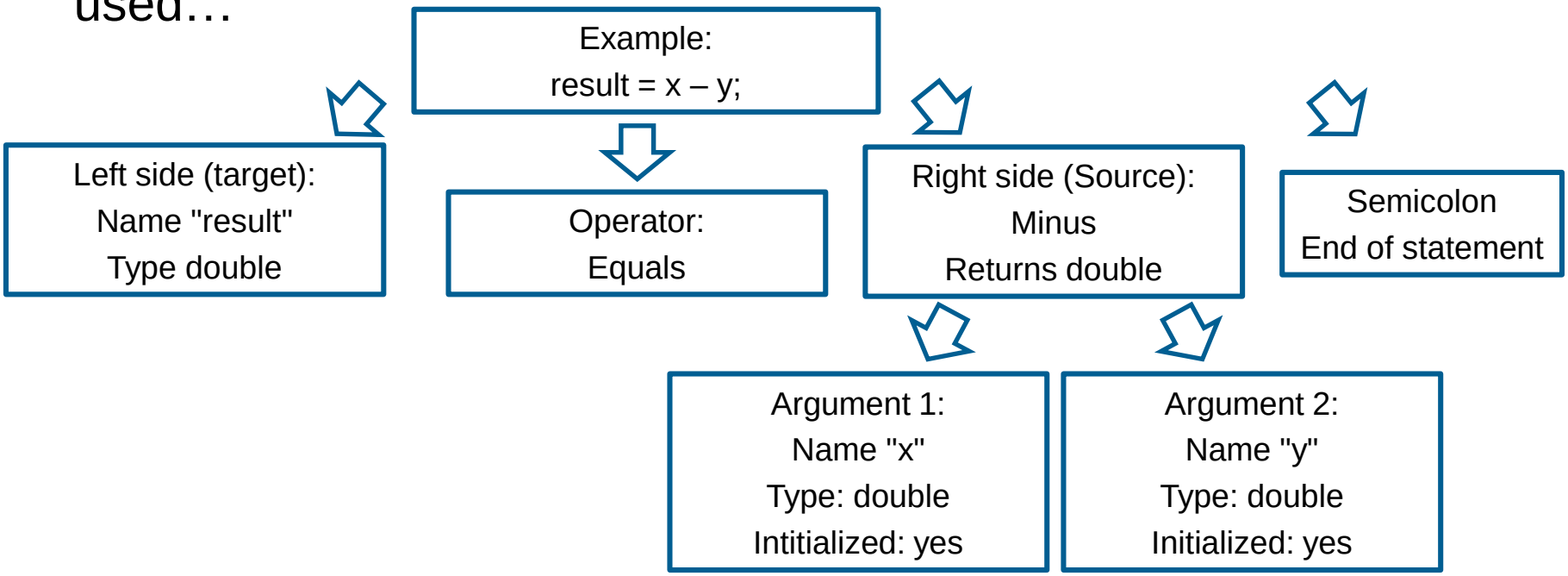
```
$>  cat syntactic01.c

1   #include stdio.h
2
3   int main()
4   {
5      printf( "%f\n", 1.0 )
6      return 0;
7   }

$>  gcc -Wall syntactic01.c
```

Semantic analysis:

Adding and checking semantic information on parse tree. Examples are: type checks, variables initialized before being used…



Semantic analysis:

My refrigerator just drove a car to Chicago.

Perfect syntax but listeners will question your sanity.

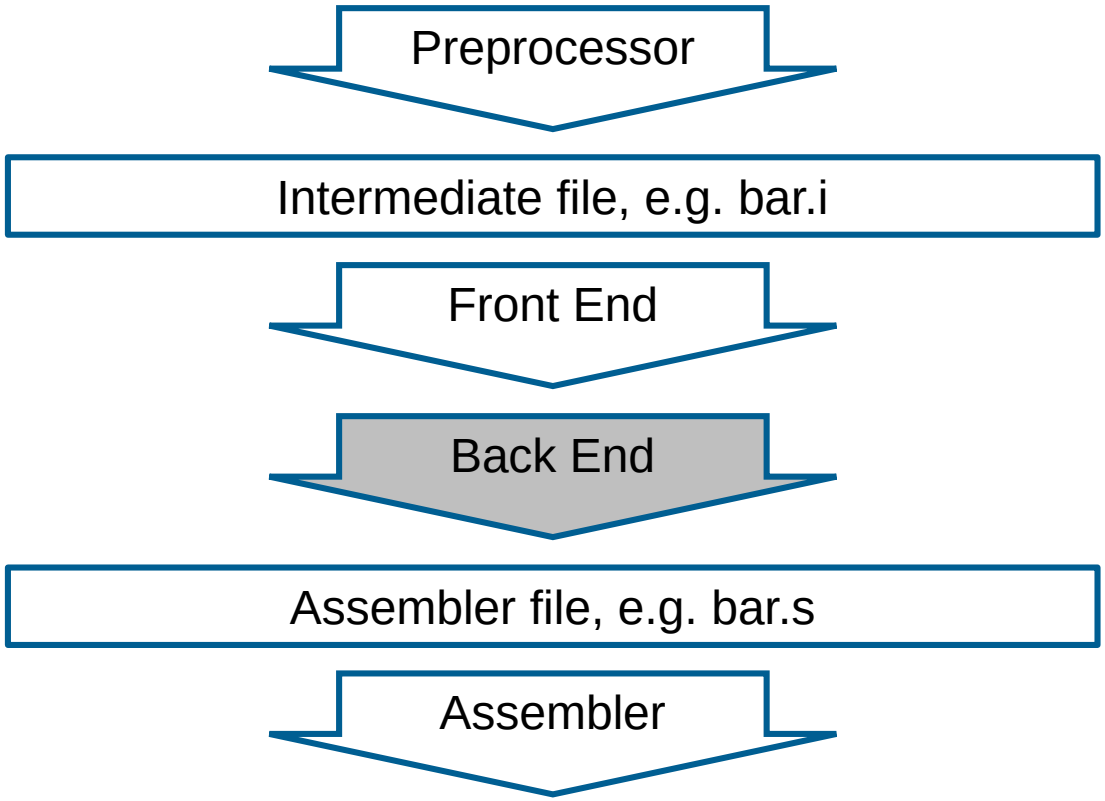
Example: semantic analysis

```
$>  cat semantic01.c

1   #include stdio.h
2
3   int main()
4   {
5       printf( "%f\n", myFloat );
6       float myFloat = 3.1415;
7       return 0;
8   }

$>  gcc -Wall semantic01.c
```

Compiler


Preprocessor
Intermediate file, e.g. bar.i
Front End
Back End
Assembler file, e.g. bar.s
Assembler

Code analysis, e.g.

- Data flow
- Use-define chains
- Dependence analysis
- Pointer analysis
- Escape analysis

Code optimization, e.g.

- Inline expansion
- Dead code elimination
- Loop transformation

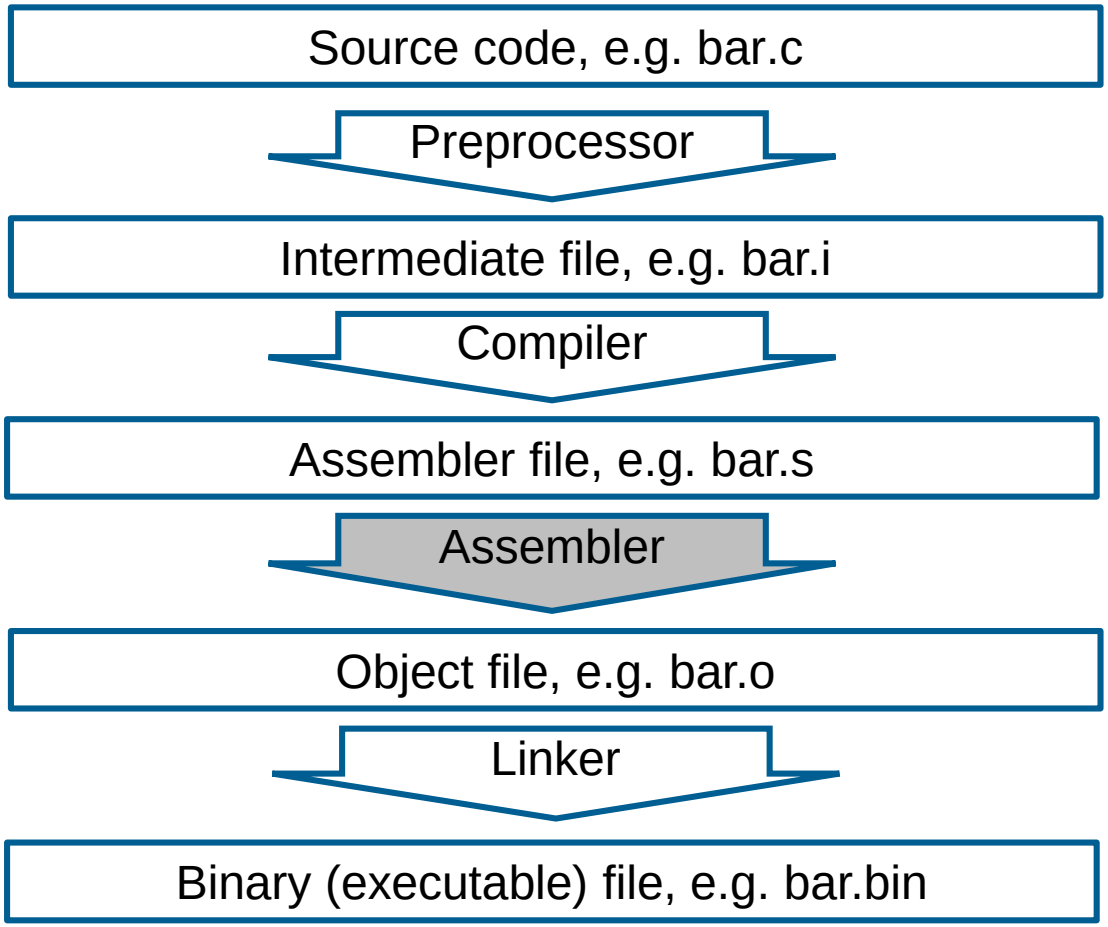
Code generation

Use internal code representation (optimized and error-checked tags in code tree) to generate assembler language.

Here, the first machine specific translation takes place (assembler code is different for different systems e.g. 32-Bit vs. 64-Bit but similar for one family)

=> see introduction01.s

Assembler


Source code, e.g. bar.c
Preprocessor
Intermediate file, e.g. bar.i
Compiler
Assembler file, e.g. bar.s
Assembler
Object file, e.g. bar.o
Linker
Binary (executable) file, e.g. bar.bin

Assembler

Translates assembler code to machine code.

Result of automated code generation (compiler) hard to read, thus example of self written assembler follows.

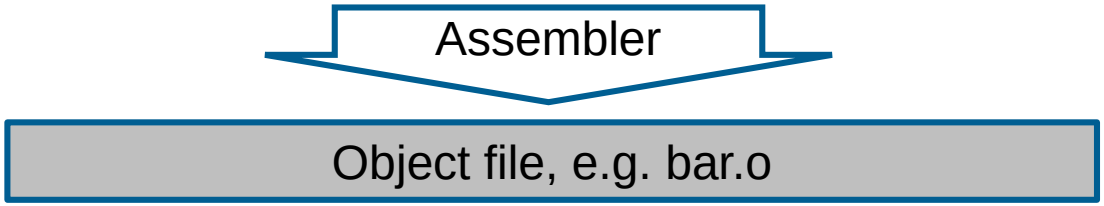
nasm –f elf64 assembler01.asm

ld –lc –I/lib/ld-linux-x86-64.so.2 assembler01.o

Assembler does not touch the code anymore! Direct translation into binary format.

The assembler code is human readable (ascii) representation of machine code. Beyond the assembler only binary representation

Object file



- Object file is a file containing machine code
- Machine code is a set of instruction that can be executed on a specific architecture
- No abstraction present (like variable, object…)
- Object file is not directly executable
- In object file more functions can be packed
- Standard format for object file on UNIX systems is the “ELF” (see next slides)

Interlude

As do other files, binaries must follow certain conventions regarding their structure.

Different operating systems do have different conventions.

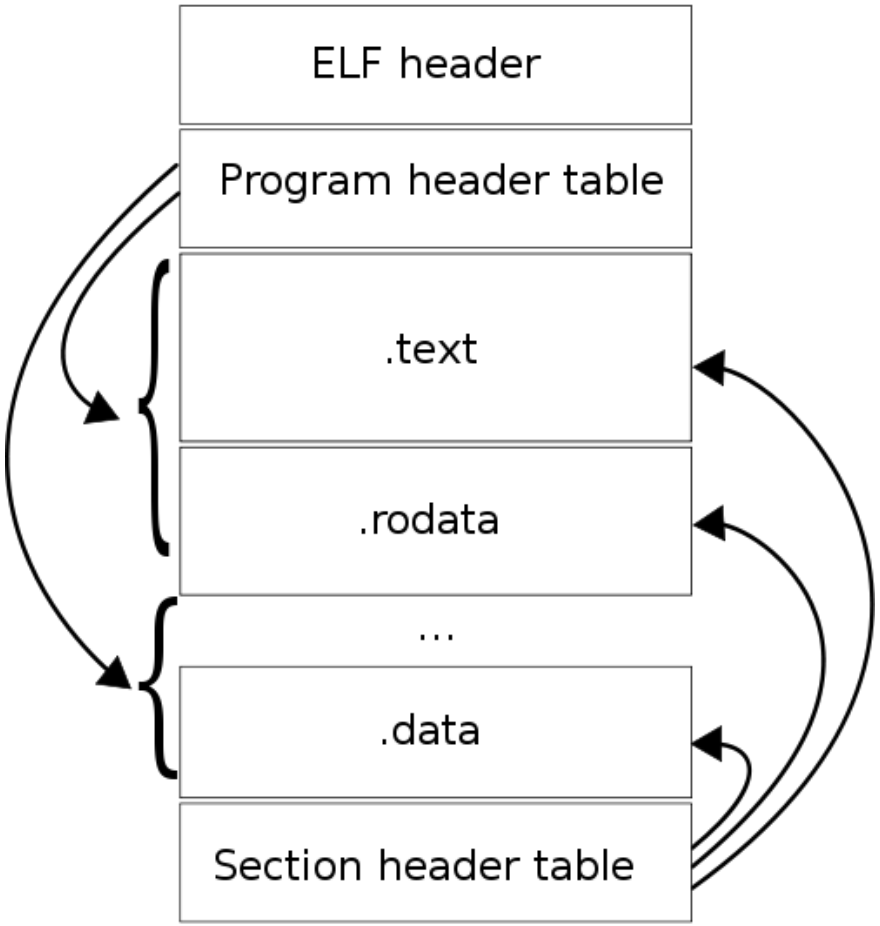
Example: Running Linux 32-Bit application under Linux 64-Bit:

Error: wrong ELF class

Interlude - ELF

E xecutable and
L inkable
F ormat

File format for executables, object code, libraries…

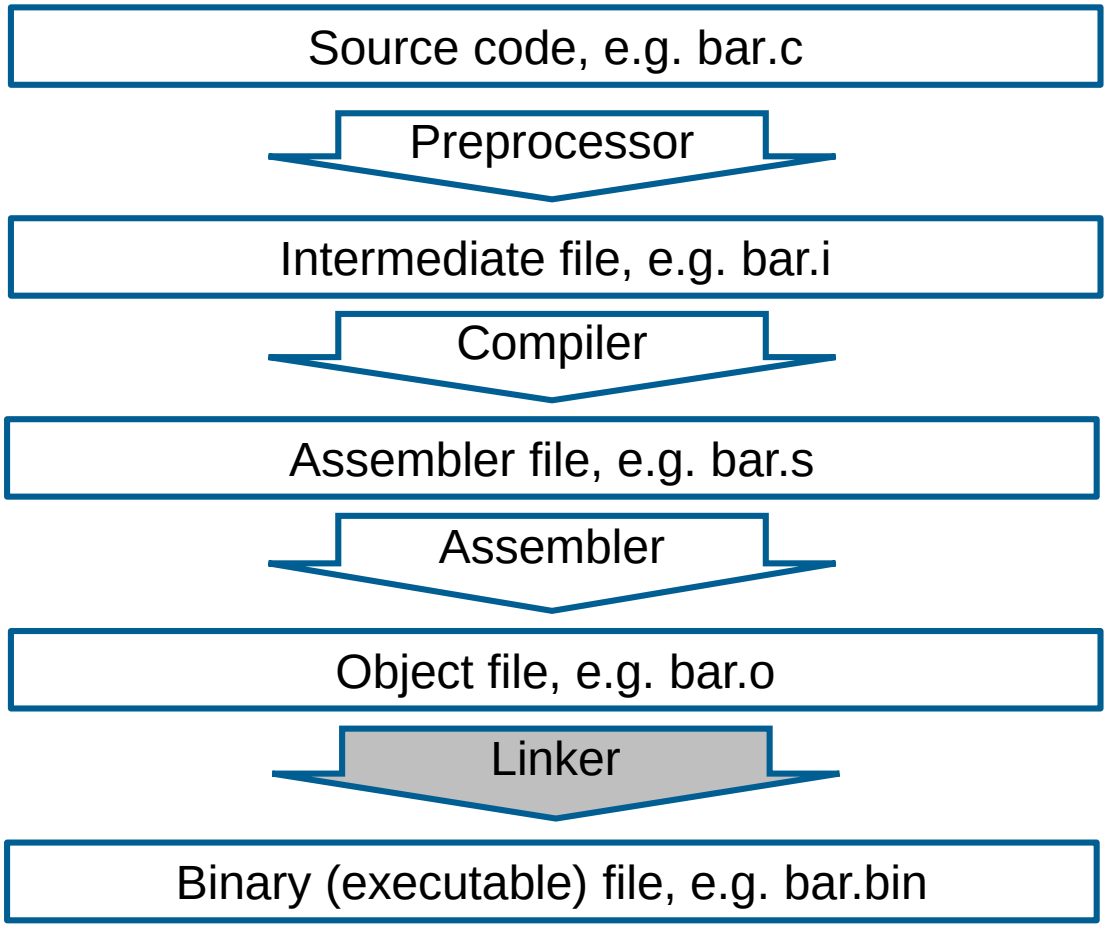


Interlude

Tools:

nm	show symbol table of object code file
readelf	read ELF-table of binary file
size	list section sizes and total sizes
ldd	print shared library dependecy

Linker


Source code, e.g. bar.c
Preprocessor
Intermediate file, e.g. bar.i
Compiler
Assembler file, e.g. bar.s
Assembler
Object file, e.g. bar.o
Linker
Binary (executable) file, e.g. bar.bin

Linker

Not all source code shipped/rebuild for every project – parts are distributed as library and linked to own object code.

Two concepts:

- Static linking
- Dynamic linking

What are the benefits?

Dynamic linking allows for memory sharing, fast loading applications. Only very limited cases where static linking is the way to go.

Linker

Linker already used for assembler example:

ld –lc –I/lib/ld-linux-x86-64.so.2 assembler01.o

Flags already known from compiler – in fact compiler passes „–l“ and „–L“ flags directly to linker.

Tools:

ldd	show libraries linked dynamically to binary