COMP3173 Compiler Construction Introduction

Dr. Zhiyuan Li

Outline

- Course Information
  - Contact
  - Outline
  - Assessment
  - Textbook

Contact

Section	Instructor	TA	Timetable	
1001	Dr. Zhiyuan Li	Ms. Cici Chong Chen	Lecture	Wed. 10:00-11:50 T5-701
	goliathli@uic.edu.cn	chongchen@uic.edu.cn	Lecture	Fri. 9:00-9:50 T4-401
	T3-502-R6	T3-602-R25-H8	Tutorial	Tue. 17:00-17:50 TBD
1002	Dr. Wentao Cheng	Ms. Ruby Zhou	Lecture	Tue. 11:00-11:50 T5-505
	wentaocheng@uic.edu.cn	rubinzhou@uic.edu.cn	Lecture	Wed. 10:00-11:50 T29-405
	T7-102-R13	T3-502-R26-H5	Tutorial	TBD
1003	Dr. Monica Wen Chen	Ms. Lily Chengyan Lin	Lecture	Mon. 10:00-11:50 T29-303
	wenchen@uic.edu.cn	chengyanlin@uic.edu.cn	Lecture	Tue. 10:00-10:50 T5-705
	T6-302-R5	T3-602-R25	Tutorial	Wed. 12:00-12:50 TBD

Assessment

- Assignment 20%
- Project 40%
- Final exam 40%
- MR sections and FE sections will be graded separately.
- Low final exam score will downgrade the letter grade.

Final Exam Score	Highest Letter Grade
0 - 19	F
20 - 24	D
25 - 30	C-

Regulations

- Plagiarism is not allowed.
- “But why do I have to state this to Y3 students?” - Goliath

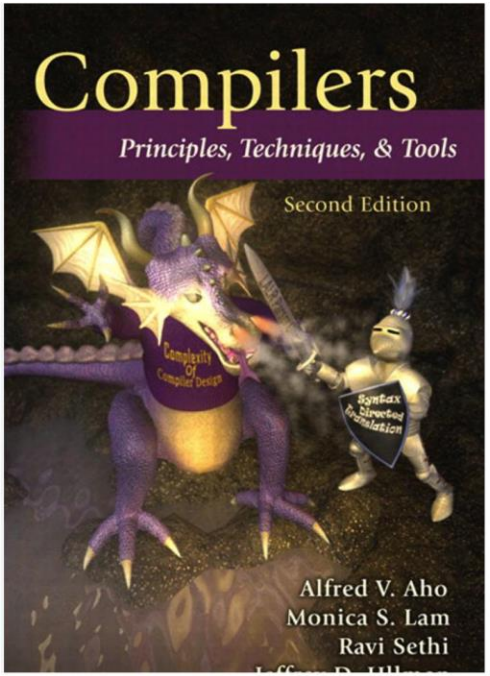
- If you have difficulties in the project, you can do some parts and get the most of the marks.
- Relax!

LLM

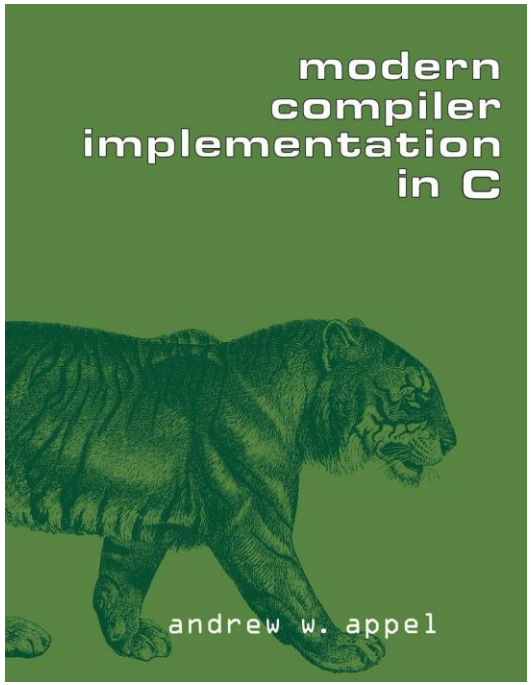
- For LLMs, students should follow these.
  - Consider LLM as **"a 24-hour teacher"**.
  - LLMs / Teachers can answer you questions.
  - LLMs / Teachers cannot do assignments for you.
  - Students cannot use LLMs / teachers in exams.
- For example, if you want to know how to do 17×23
  - you can ask LLM / teacher "how to calculate multiplication?"
  - If you ask "what is 17×23", your teacher will not answer.

Textbook

- Dragon book



- Tiger book



Some Reasons

There are some good reasons why you need to study this subject.

- To understand developing tools.
- The contents are related to some other subjects in CS.
- Useful in data language specifications.
- Create your own programming language.

Introduction to Compiler construction

Outline

- Definition
- Main phases
- Analysis
- Intermediate code generation
- Synthesis
- Some other components

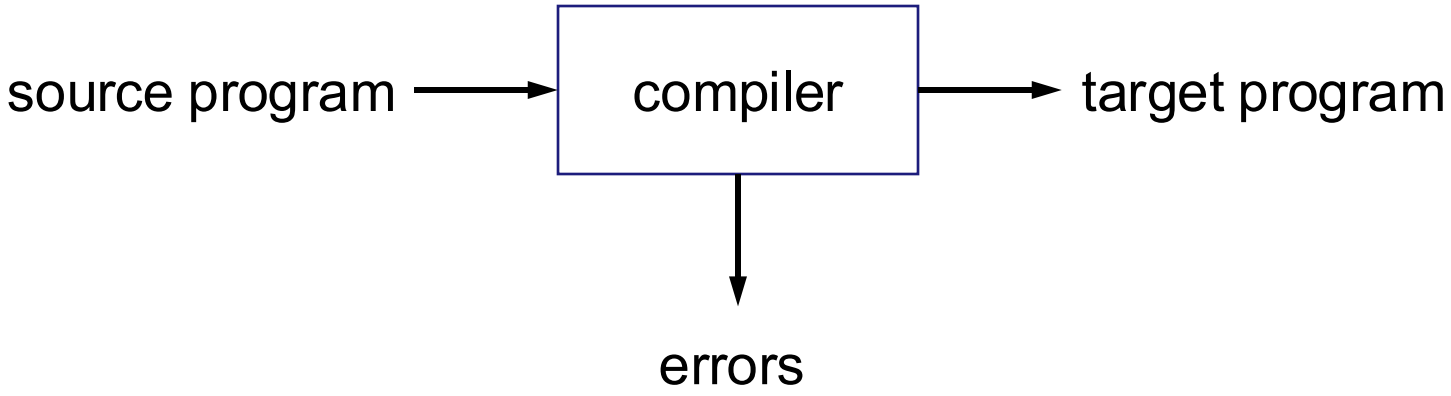
Definition

- A **compiler** is a program that reads a program written in one programming language (the ***source*** language) and ***translates*** it into an ***equivalent*** program in another programming language (the ***target*** language).

- "Equivalent" means the target program does exactly the same thing as the source program.
- Target language can be
  - some other general purpose programming languages, like C, C++, Java;
  - machine languages understood by microprocessors; or
  - other domain-specific languages, like PostScript used by printers.

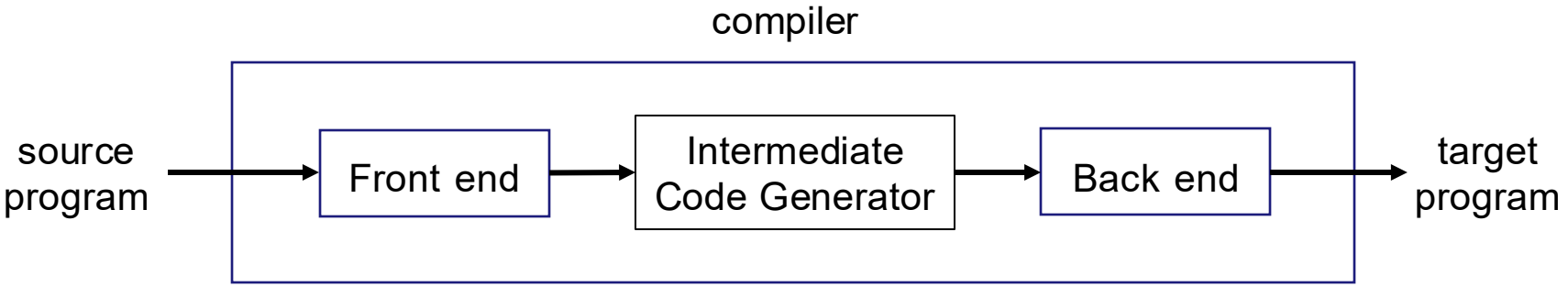
Error Handling

- The source program may contain some errors.
- If errors occur the compiler must detect all errors.



Main Phases

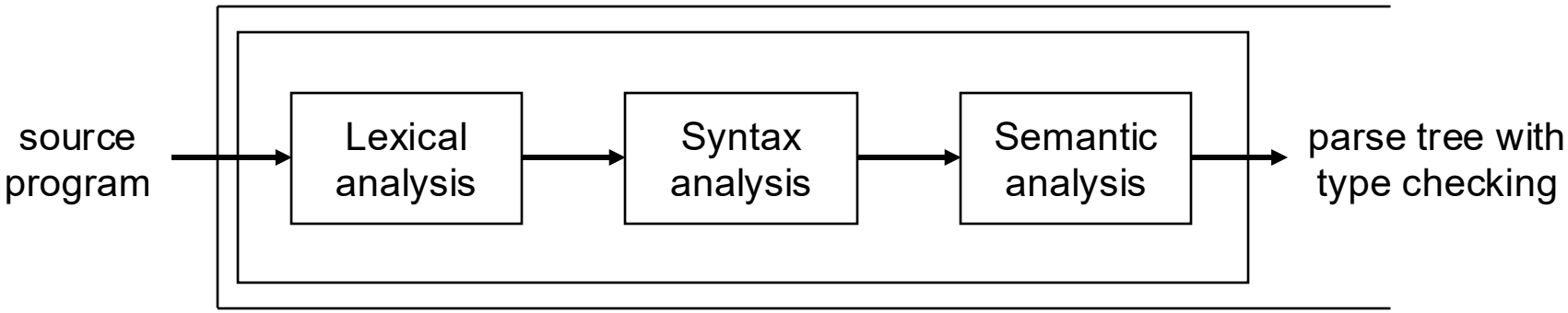
- A compiler is not one big piece of software. It has a sequence of phases.
- On the high level:
  - Input: source program
  - Source Code Analysis (front end)
  - Intermediate Code Generator
  - Synthesis (*back end*) Output: target program



Analysis (*Front End*)

Analysis comes in three phases:

- Lexical analysis
- Syntax analysis
- Semantic analysis



Lexical Analysis

- The program does lexical analysis is called ***lexer*** or ***scanner***.
- Reads the input program character by character as a stream.
- Splits the stream into ***lexemes***, means “lexical elements”.
- Classifies each lexeme into a category of lexemes, called ***token***.
- Same as ***tokenizer*** in natural language processing. It splits the text into words.
- In most of the programming languages, a lexer skips spaces and comments.
- We use ***regular expressions*** to do lexical analysis.

Lexical Analysis

- For example, the following piece of code in C language

```
num=1+23;
```

has 9 characters, 6 lexemes, and 5 different tokens.

No.	Lexeme	Token
1	num	identifier
2	=	ass_opt
3	1	int_literal
4	+	plus_opt
5	23	int_literal
6	;	semicolon

Tokens

- Here is a list of common tokens:
  - identifier: names the programmer chooses, like variable names;
  - keywords: names already in the programming language, like `if`;
  - separator: punctuation characters and paired-delimiters, like `{}`;
  - operator: symbols operate on arguments and produce results, like +;
  - literal: numeric, logical, textual, reference literals, like `true`, `123`;
  - comment

Syntax Analysis

- The program which does syntax analysis is called a ***parser***.
- Uses a ***grammar*** to analyze the ***form*** of tokens.
- And groups the tokens into a nested hierarchical structure.
- The structure is called a ***parse tree***.
- The parse tree shows the structure of the program.
- Internal vertices are called ***non-terminals***. Leaves are called ***terminals***.
- Same as using English grammar to check if the nouns, verbs, adjectives, etc., in a sentence are in the correct order.

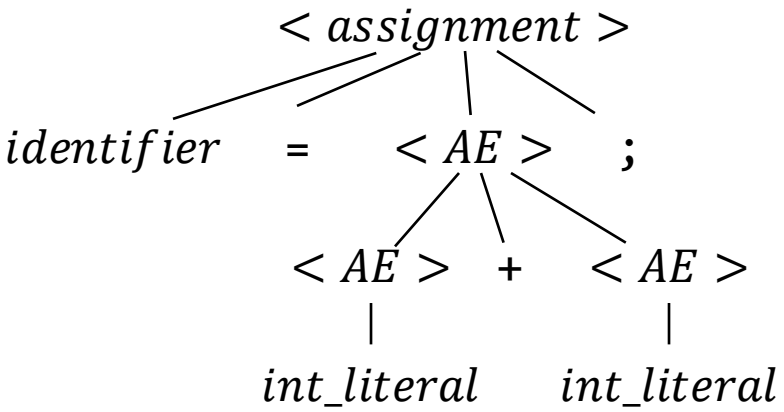
Syntax Analysis

- For example, the following piece of code in C language

```
num=1+23;
```

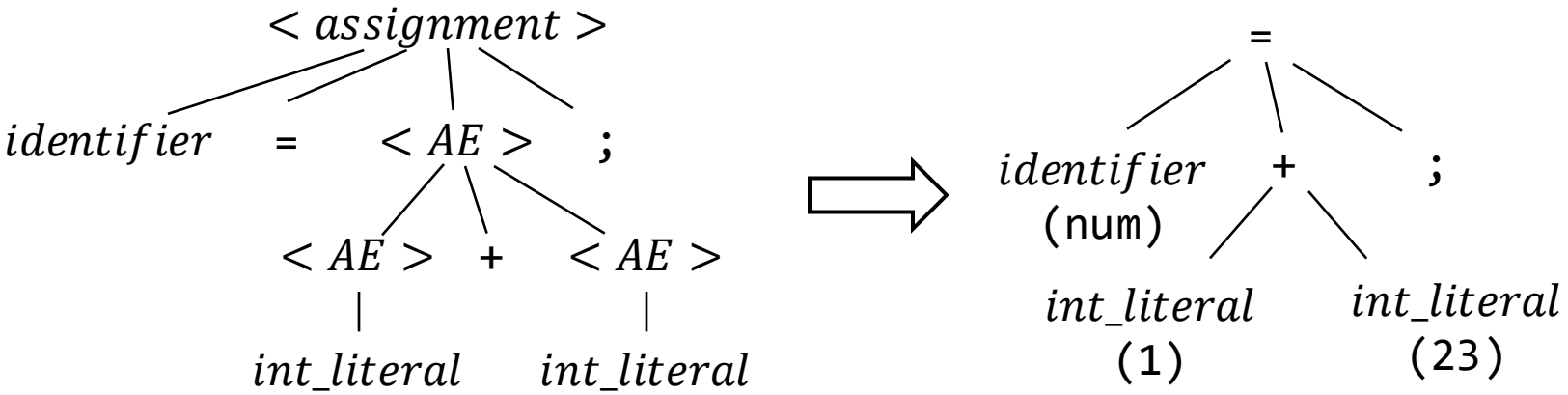
we can use the following grammar to generate the parse tree.

$$< assign > \rightarrow identifier = < AE >;$$
$$< AE > \rightarrow < AE > + < AE >$$
$$\rightarrow identifier$$
$$\rightarrow int\_literal$$



Syntax Analysis

- Sometimes the parse tree can be extremely large.
- To simplify the structure, we can remove the internal vertices and let the operators be the parents.
- This is called ***syntax tree***.
- Similar to expression trees (in Discrete Math).



Semantic Analysis

- It takes a parse tree as input.
- Checks the “meaning” of the program.
- Searching errors in the program. For example,
  - undefined variables,
  - uninitialized variables,
  - multiple declaration,
  - types of variables for operations and functions, etc.
- Semantic Analysis also does ***type conversions*** for many programming languages. For example in C,

```
float x=3.5;
int y=x;
```

the compiler adds a type conversion on x before signing the value to y.

Errors

- Lexical error: some characters do not form any lexeme.
- Syntax error: the code does not follow the grammar.
  - In other words, tokens are not in the correct order.
- Semantic error: the code is meaningless.

Sample Code	Error type	Reason
`int 2esd=5;`	Lexical error	`2esd` cannot be a variable name.
`int x 5;`	Syntax error	A variable cannot be followed by a number directly.
`int x="Hello";`	Semantic error	`x` is an integer.
`int x 5=;`	Syntax error	Wrong order
`int x=5;` `int x=3;`	Semantic error	Multiple declaration

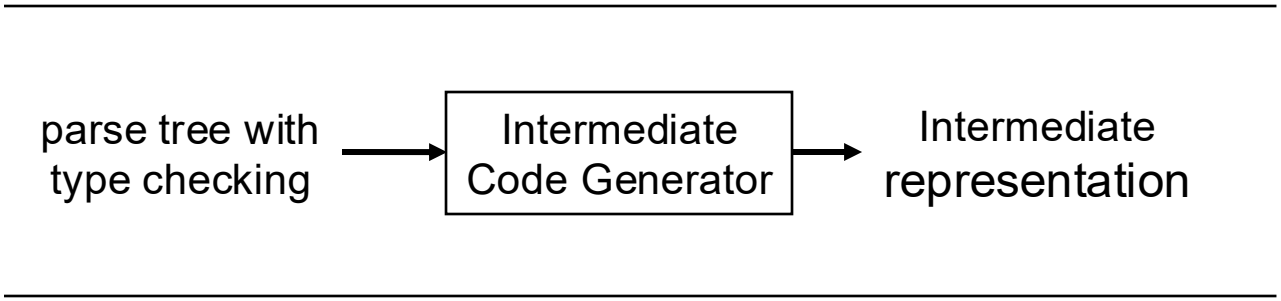
Other Applications for Source Code Analysis

Code analysis is also useful in other applications. For example,

- Structure editors, like Eclipse analyzes the code and highlight in different colors, detect some errors, allow users jump between matching parenthesis, etc.
- Pretty printers can nicely format your code with correct indentation, break lines, and different fonts for different parts of the program.
- Static checkers can check your program for bugs without running it.
- Interpreters directly perform the operations in the source program instead of producing a target program, like Python, web browsers.

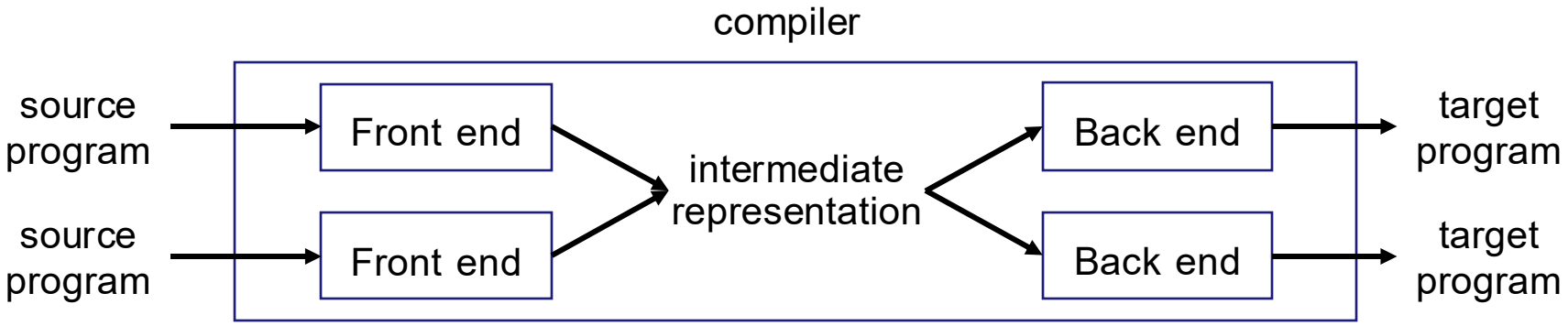
Intermediate Code Generator

- The parse tree with type checking after the semantic analysis is very high-level. It's not easy to convert a parse tree to target language directly.
- The generator takes a parse tree with type checking as input and generates an intermediate code.



Intermediate Code Generator

- More importantly, the intermediate code can act as an interface in between front end and back end. The compiler can be expanded to support more source languages or target languages by only implementing the front end or the back end.
- For example, GCC can compile C, C++, Java, Object-C, Fortran, etc. on Windows, Linux, Android, iOS, etc.



Intermediate Code Generator

- The intermediate code only exists in the compiler.
- The representation of intermediate code is totally up to designer of the compiler.
- A good intermediate code needs to be
  - convenient to be produced from the parse tree with type checking;
  - convenient to be translated into the target language;
  - simple and have clear meaning, so that optimization can be done easily.
- The most commonly used intermediate language is call ***single static assignment*** (SSA).
- In this course, we will use ***three-address code***, which is easier than SSA.

Three-address Code

- In three-address code, each line contains at most 3 different variables and does at most 1 operation.
- For example,

```
id1=id2+id3*2;
```

where the variables are floating points. The three-address code will be like

```
temp1=float(2)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3
```

Intermediate Code Generator

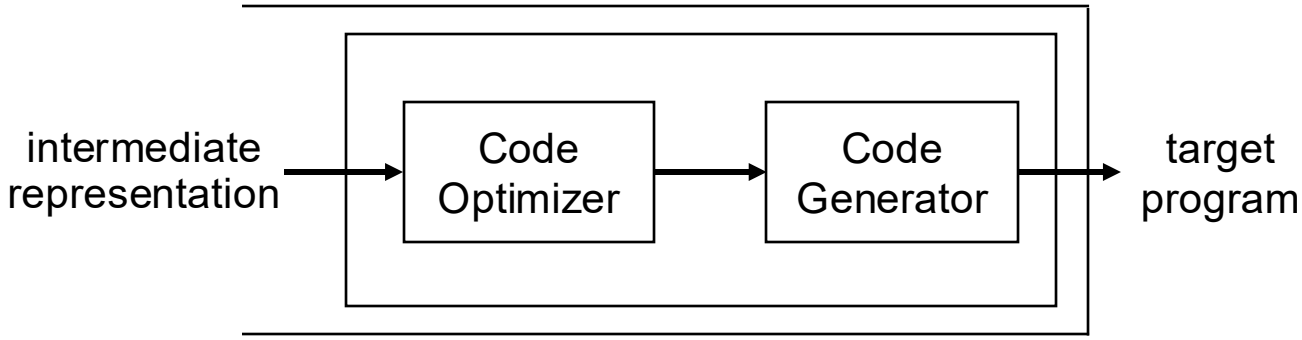
- To generate correct intermediate code, the generator does two things.
  - Decide the order in which operations have to occur (multiplication before addition in the above example).
  - Generate (variable) names for intermediate results (all the variables $tempX$ in the above example).

Note:

- The intermediate code generator (also back end) is only used when the target language is a low-level language, machine code or assembly language for example.
- If the target language is another high-level language, the parse tree will be directly translated into the target language.

Synthesis (*Back End*)

- The back end of a compiler takes the intermediate code as input and generates a program in the target language.
  - Code Optimizer
  - Code Generator
  - Machine-dependent code optimization (optional)



Code Optimization

- The code optimization phase tries to rewrite the intermediate code with a “better” performance.
- Usually “better” means “faster”. But sometimes it means “smaller” for embedded systems.
- For example, the three-address code we have above can be rewritten as

```
temp1=id3*2.0
id1=id2+temp1
```

- Code optimization is a very complex topic.
- Some compilers take hours or even days to optimize the code.
- In this course, we only introduce some simple optimizations that usually improve code a lot without consuming too many resources.

Code Generation

- The code generator takes the optimized intermediate code from the optimizer and generates the target code.
- It does three thing.
  - ***Instruction selection***
  - ***Instruction scheduling***
  - ***Register allocation***
- These are all optimization problems and ***NP-hard***, meaning that there is no existing algorithm which finds the optimal solution in polynomial time.
- In real applications, we use ***approximations*** to find some good enough solutions instead of the optimal ones.

Instruction Selection

- Select the instructions from the target language.
- Usually, there are multiple ways to implement one instructions.
- Consider a sequence of instructions, the possibility becomes **exponential**.
- For example in C, these three operations do the same thing. But the last one is much faster.
- Things will be complicated when multiple operations are nested with each other.
- It's hard to say which one is the fastest on the right.

```
x*2
x+x
x<<1
```

```
x*3
x+x+x
(x<<1)+x
```

Instruction Scheduling

- Arrange the instruction in the optimal order.
- The importance is caused by instruction pipelining within a microprocessor.
- The following two pieces of code are pipelined differently if the pipeline has two stages(fetch and execute).

```
MULI R1, R2
MULI R2, R5
MULI R3, R4
```

Clock cycle	1	2	3	4	5
Fetch	MULI R1, R2		MULI R2, R5	MULI R3, R4	
Execute		MULI R1, R2		MULI R2, R5	MULI R3, R4

```
MULI R1, R2
MULI R3, R4
MULI R2, R5
```

Clock cycle	1	2	3	4
Fetch	MULI R1, R2	MULI R3, R4	MULI R2, R5	
Execute		MULI R1, R2	MULI R3, R4	MULI R2, R5

- The number of possible arrangement can go up to **factorial** !

Register Allocation

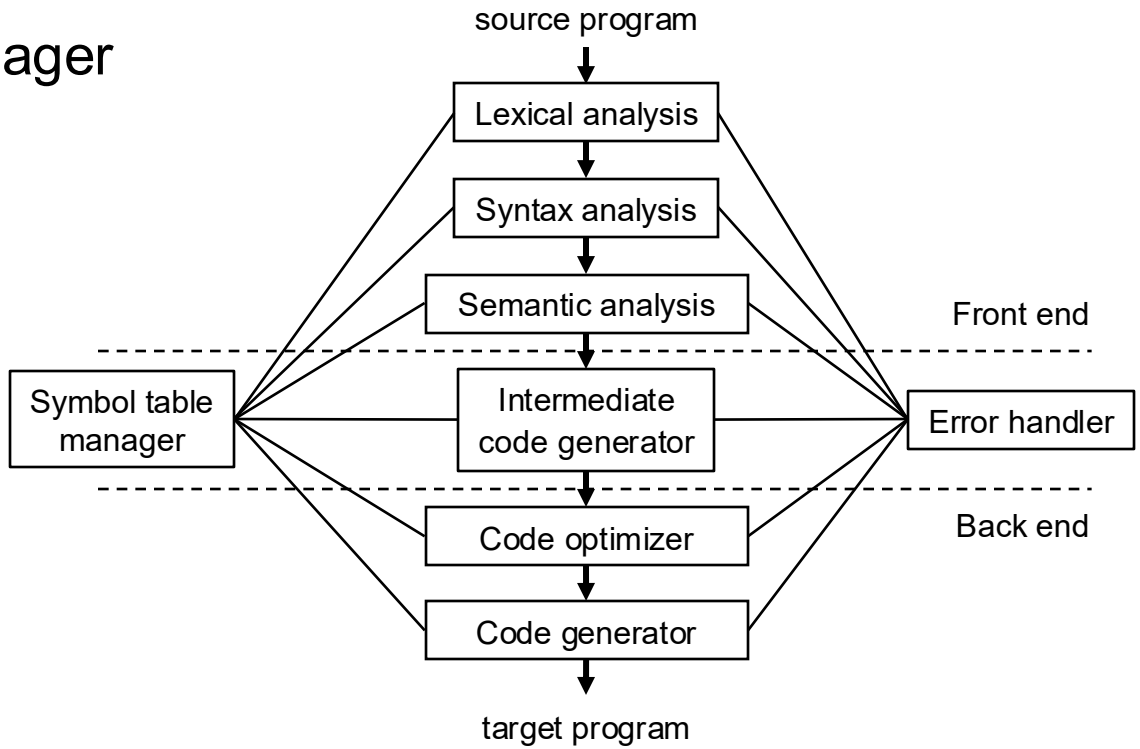
- Register allocator decides ***where*** and ***for how long*** each variable (from the intermediate code) is saved in the register of a microprocessor. Registers are **very small**.
- There is a tradeoff between re-using the variables in the future, versus save register space for other computations. Is it more efficient to
  - keep the value in the register;
  - move the value to memory, save register space, but needs additional computation;
  - simply remove the value and reload it when it is needed again?
- For example

```
y=2*x;
z=2*y;
  ⋮
w=3*y;
```

Other Components

Usually, a compiler also has some other components which work in both front end and back end.

- Symbol table manager
- Error handler



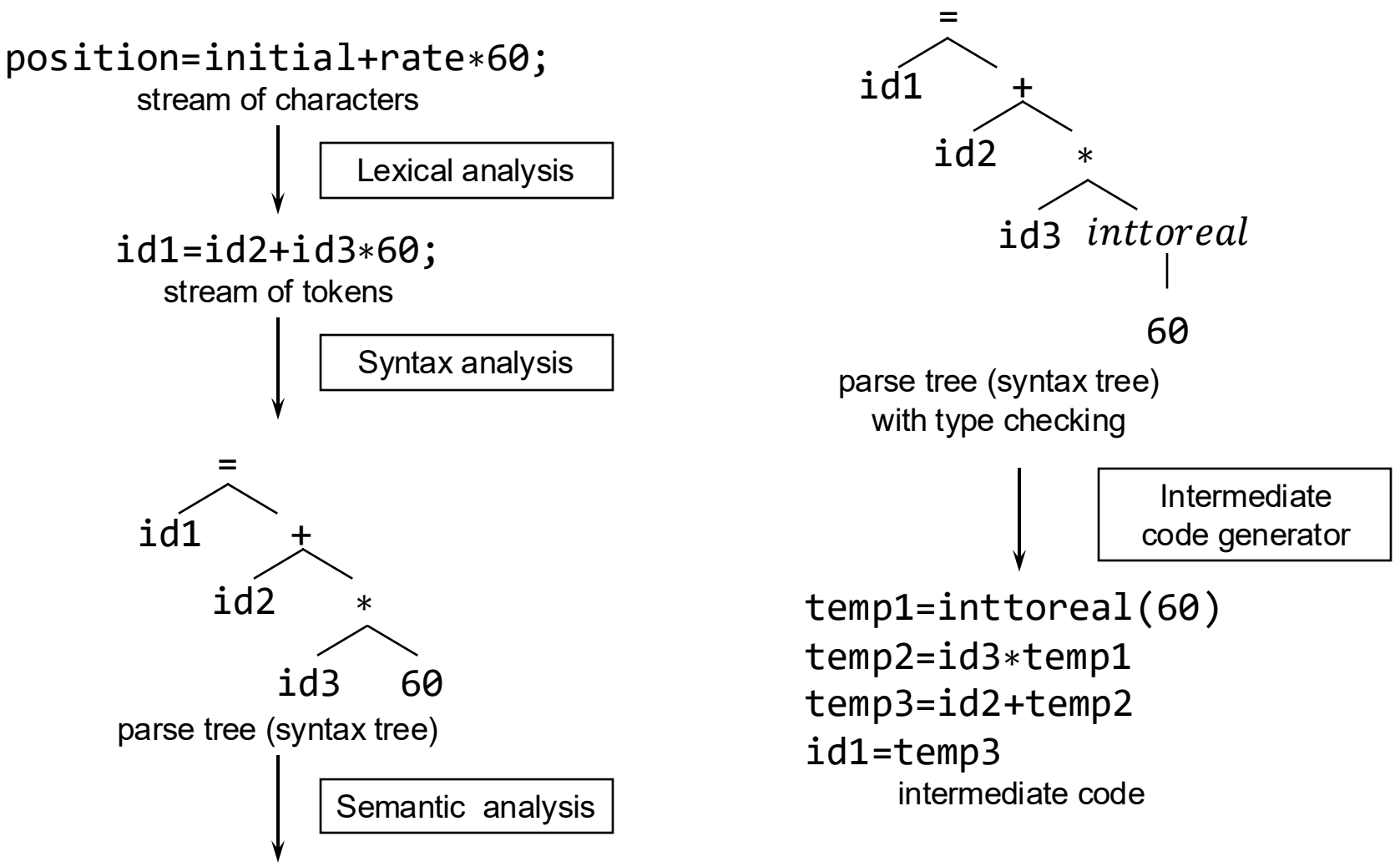
Symbol Table Manager

- The ***symbol table*** is a data structure in the compiler.
- It records the ***attributes*** of all ***identifiers*** in the source program.
- Recall that, identifiers are user defined names in the program.
- When the lexer recognizes an identifier, it inserts the identifier into the symbol table.
- The attribute of an identifier consists of its type, its scope, and the its line number.
- These information is added by different phases.
- The front end can use the symbol table to check errors.
- The back end can use the symbol table to decide memory space for each variable on the stack.

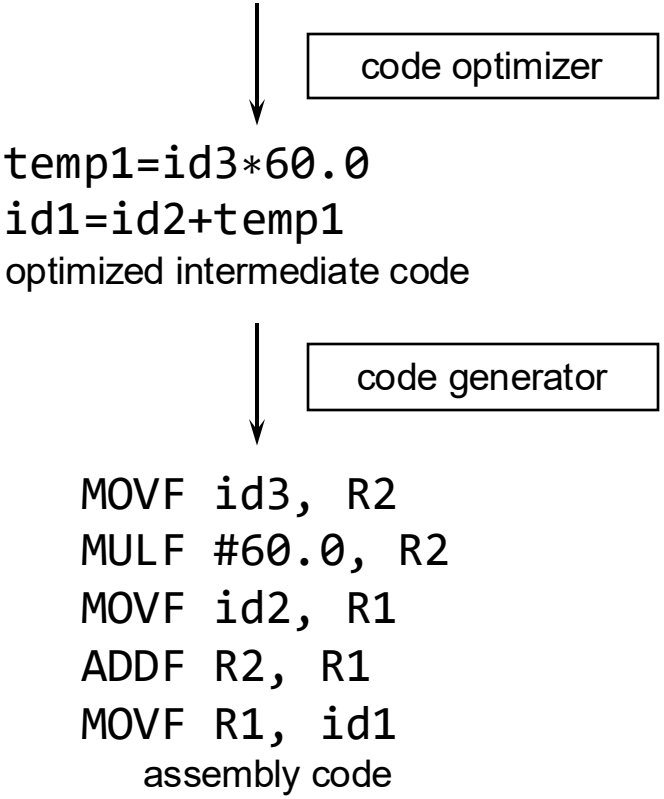
Error Handler

- A simple compiler can stop at the first error encountered.
- But this not efficient because errors can be multiple.
- In general, a compiler wants to detect as many errors as possible.
- An ***error handler*** tries to ***recover*** the errors so that the compiler can proceed on the rest of the program.
- The error handler is mostly used in the front end for analysis purposes.

Overall flow



Overall flow



Note:

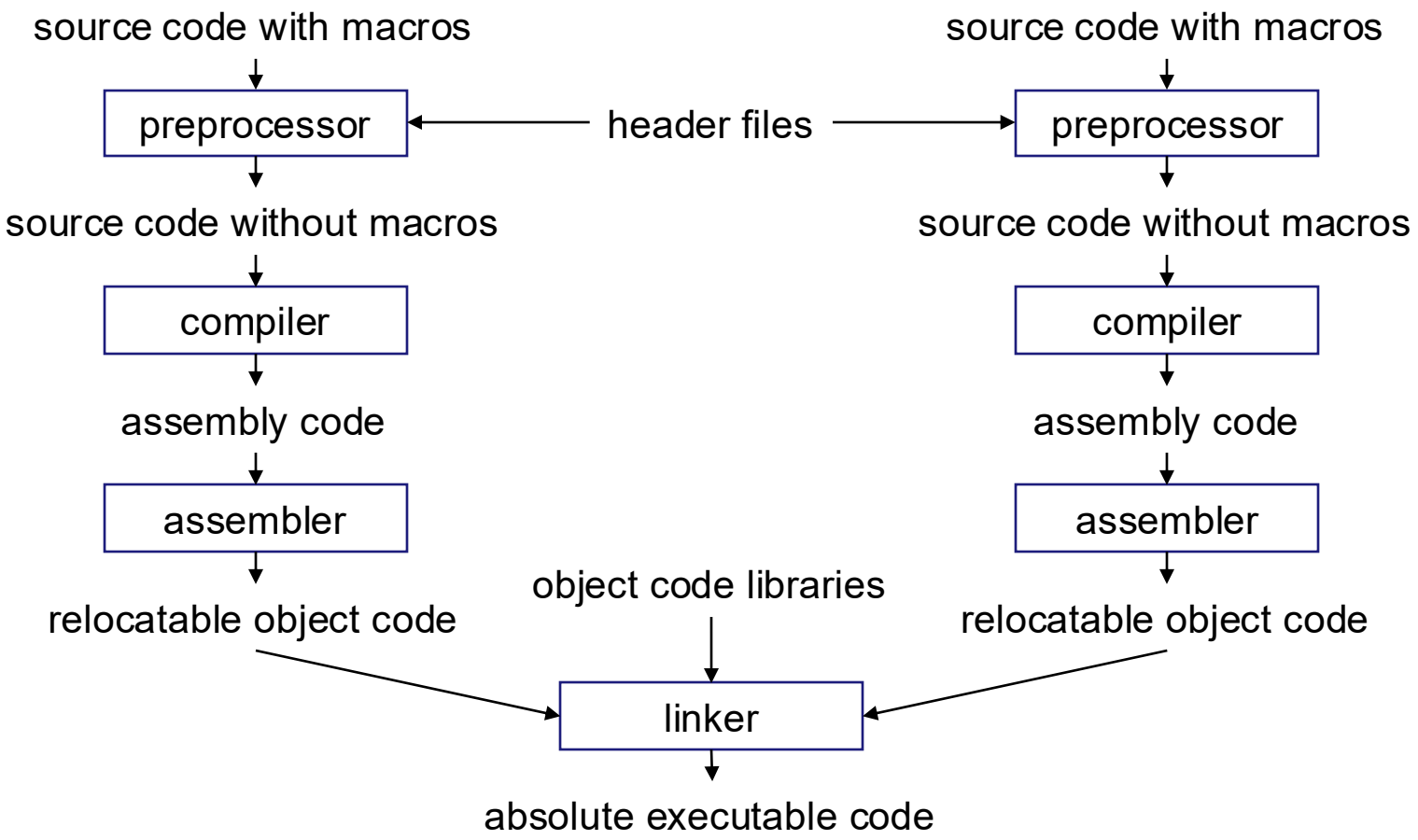
- The symbol table also keep tracking the attributes for each identifier.
- `MOVF id3, R2` means moving a floating point from `id3` to register 2.
- Similar for `MULF` and `ADDF`.

Other Phases

For some programming languages, like C, a compiler must work with some other phases outside the compiler.

- ***Preprocessor***
  - Works before the compiler.
  - Handles the macros and file inclusion by simply copy and paste.
- ***Assembler***
  - Works after the compiler.
  - Takes assembly codes as input and generates binary instruction that can be executed by the microprocessor.
  - The instruction are ***object code*** and stored in ***object files***.
  - Object files are ***relocatable***.
- ***Linker***
  - Modifies the memory addresses in the object files.
  - Generates absolute executable code.

The Whole Picture



The End of Lecture 1