SOFTWARE ARCHITECTURE
5. COMPILER

Tatsuya Hagino
hagino@sfc.keio.ac.jp

lecture URL

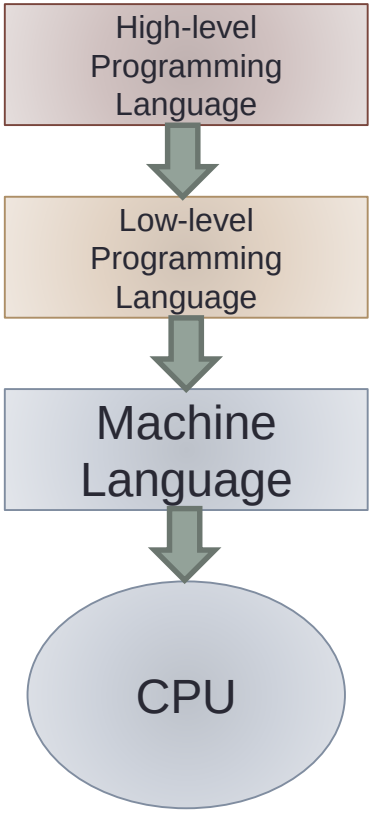
https://vu5.sfc.keio.ac.jp/slide/

Programming Language

- Programming Language
  - Artificial language to express instructions to computers

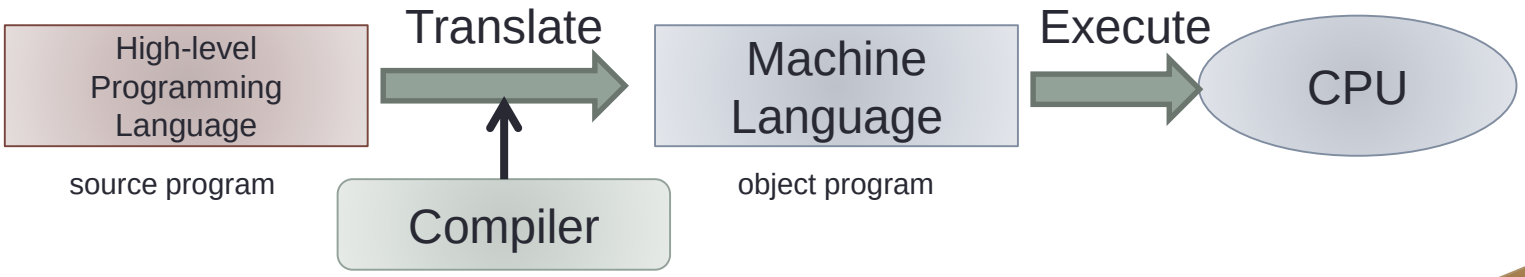
- Low-level programming language
  - Close to a machine language which computer can directly execute
  - Assembly language
  - Depends on CPU

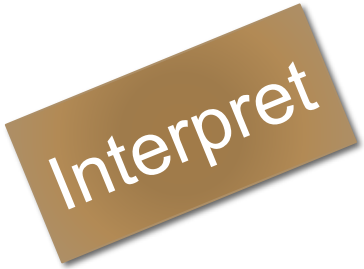
- High-level programming language
  - Independent from CPU or computer architecture
  - Easy for human to understand
  - Cannot be executed by computers directly.
  - Need translation to low-level languages



Execution of High-level Programs

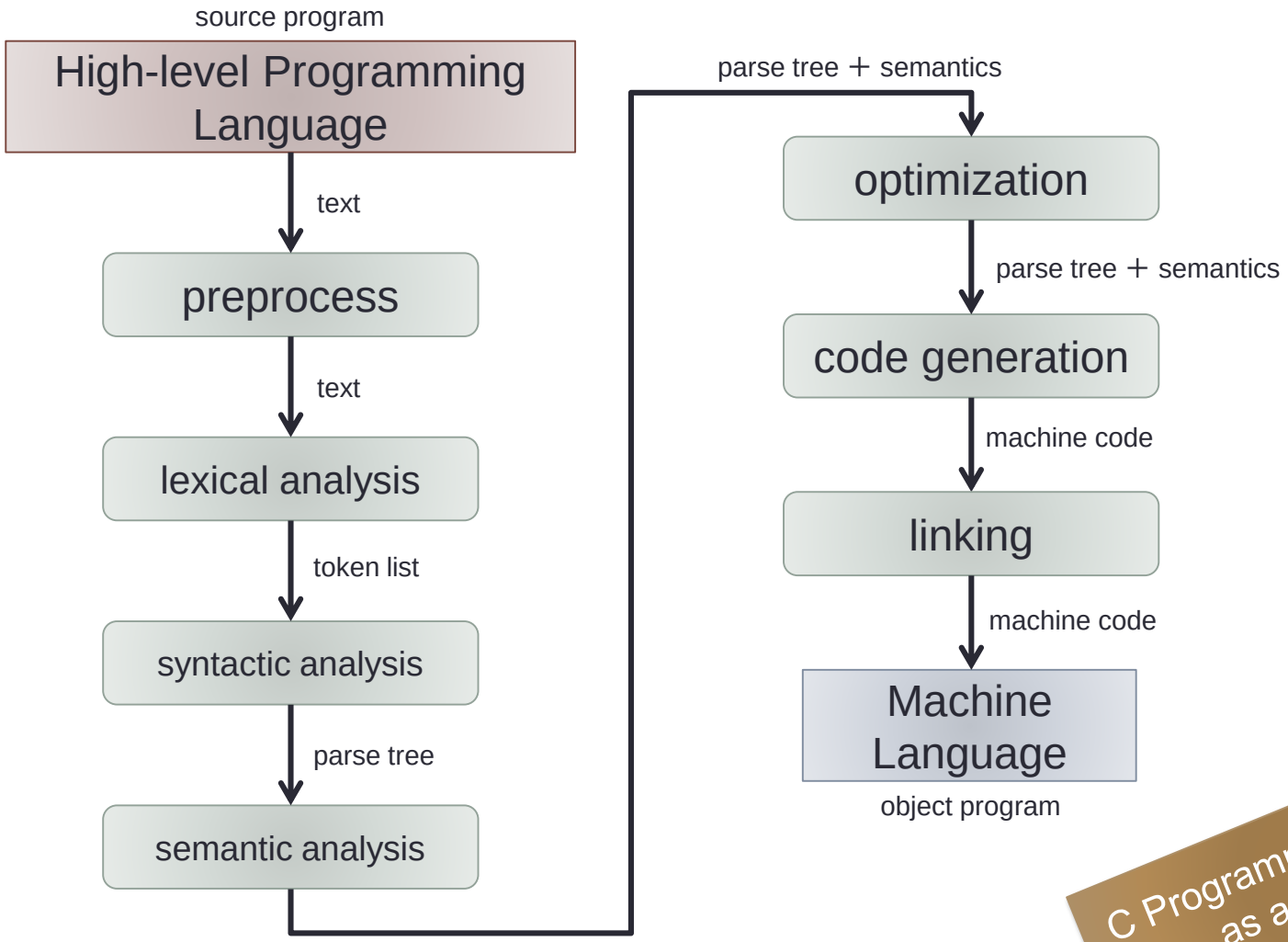
- Compiler
  - Translate programs into machine languages
  - Compilers are used for the translation





- Interpreter
  - Direct execution without translation
  - Slower execution compared with compiler
  - Easy and fast to execute when programs are changed

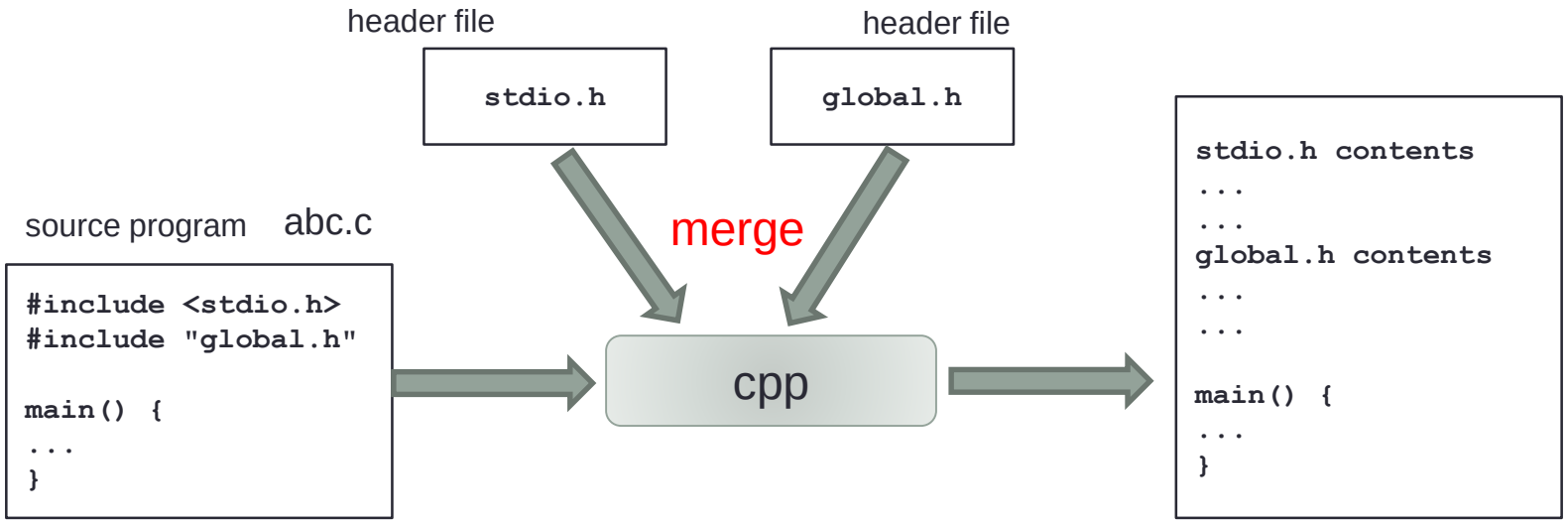
Compiler Components



C Programming Language as an example

Preprocess

- Preprocessor for C Programming Language
  - cpp (C Pre-Processor)
  - Lines starting with # are the instructions to C Pre-Processor
- Functions
  - Combine header files and other include files with the source code
  - Macro expansion
  - Conditional compilation



cpp Macro

- Give a name to a constant
  - **`#define MAX 100`**

```
for (i = 0; i < MAX; i++) {
  a[i] = b[MAX - i - 1];
}
```

macro expansion → cpp →

```
for (i = 0; i < 100; i++) {
  a[i] = b[100 - i - 1];
}
```

- Improve readability
- Easy to change the value

- Macro may take parameters
  - Parameters are replace with actual strings when expanded.
  - Not function calls but simple string replacement
  - **`#define DOUBLE(x) (x)*(x)`**

```
a = DOUBLE(5);
b = DOUBLE(a+1) + DOUBLE(c);
c = DOUBLE(DOUBLE(a+b));
```

macro expansion → cpp →

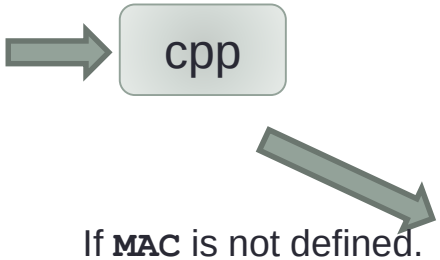
```
a = (5)*(5);
b = (a+1)*(a+1) + (c)*(c);
c = ((a+b)*(a+b))*((a+b)*(a+b));
```

Conditional Compilation

- Choose code depending on the condition

source program

```
  a = 3;
#ifdef MAC
  x = 10;
  n = 5;
#else
  x = 20;
  n = 4;
#endif
  for (i = 0; i < n; i++) {
    a += x;
  }
```



If **MAC** is defined.

source program for MacOS

```
a = 3;
x = 10;
n = 5;
for (i = 0; i < n; i++) {
  a += x;
}
```

If **MAC** is not defined.

source program for Windows

```
a = 3;
x = 20;
n = 4;
for (i = 0; i < n; i++) {
  a += x;
}
```

- Use a single source program for multiple machines

Lexical Analysis

- Lexical analysis
  - Divide a list of characters to tokens (e.g. variable names, identifiers, numbers, symbols, etc.)

```
for (i = 1; i <= 100; i++) {
  abc[i] = i;
}
```

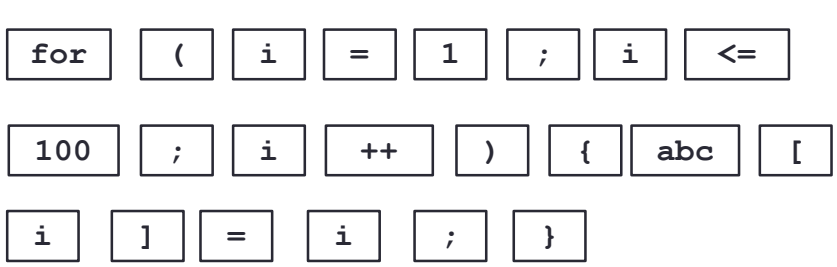
for	(	i	=	1	;	i	<=
100	;	i	++	)	{	abc	[
i	]	=	i	;	}		

- Token
  - Number: a list of numeric characters
  - Identifier: a list of alpha numeric characters which starts with an alphabet
  - Symbol: a special character or a list of special characters

- Lexical analysis returns a list of token kinds and values

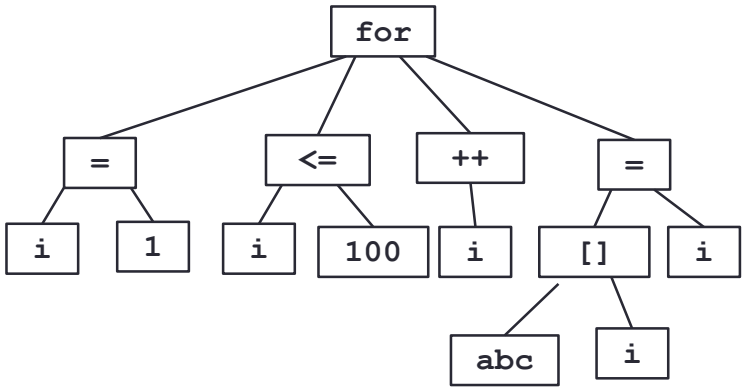
Syntactic Analysis

- Syntax analysis
  - Divide into meaningful groups
  - Group for expressions, statements, etx.
  - Create a parse tree from a list of tokens



syntactic analysis

parse tree



- Syntax
  - Defines what are correct statements and expressions
  - Written as a set of grammar rules
  - Context Free Grammar (CFG)
  - Syntactic analysis is done according to the grammar rules.

Grammar Rule

- Grammar rule for **`for`** statement

- Grammar rules are usually expressed in a context free grammar
  - BNF (Bacus Naur Form) is often used.

```
<for> ::= 'for' '(' <expr> ';' <expr> ';' <expr> ')' <statement>
```

- The grammar rules of C Programming Language can be divided to:
  - expression
  - statement
  - function
  - variable declaration
  - type declaration

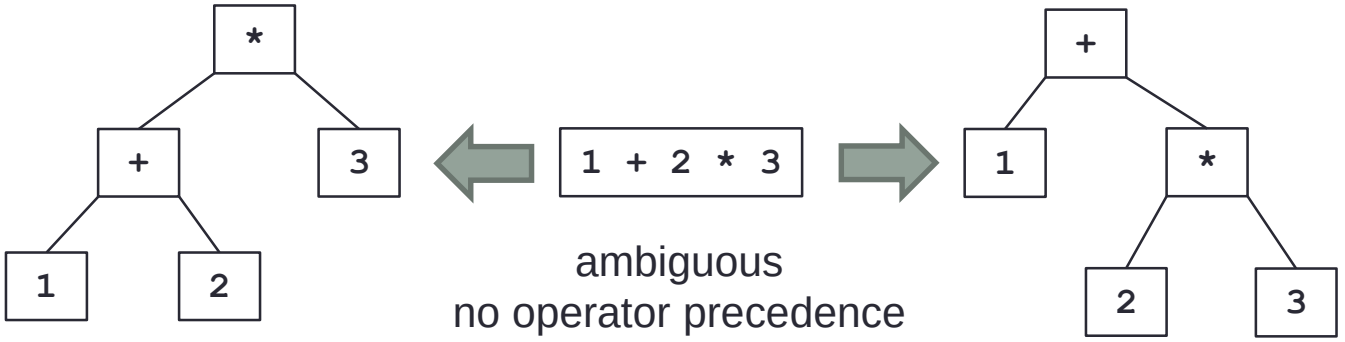
5 basic grammar for English

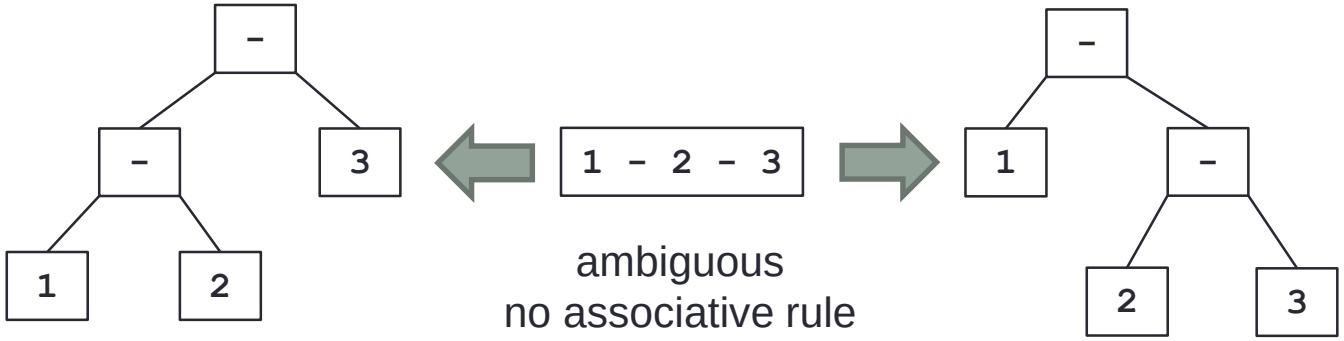
S V
S V C
S V O
S V O O
S V O C

Grammar Rule for Expressions

- Grammar rule for expressions

```
<expr> ::= <num> | <expr> '+' <expr> | <expr> '-' <expr>
                 | <expr> '*' <expr> | <expr> '/' <expr>
```

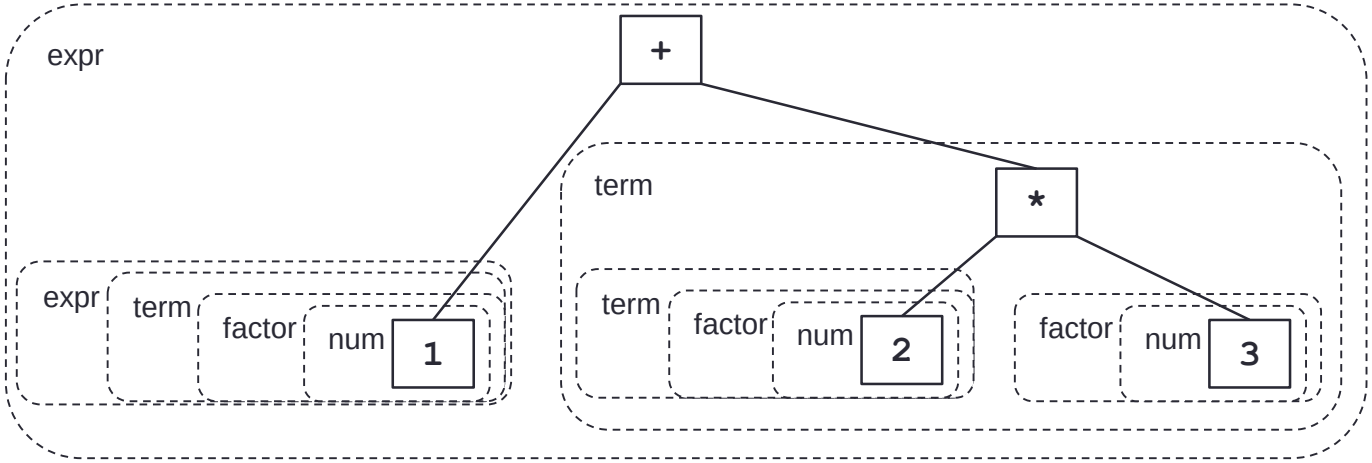




Unambiguous Grammar Rule for Expressions

- Divide into each precedence of expressions

```
<expr> ::= <expr> '+' <term> | <expr> '-' <term> | <term>

<term> ::= <term> '*' <factor> | <term> '/' <factor> | <factor>

<factor> ::= <num> | '-' <factor> | '(' <expr> ')'
```

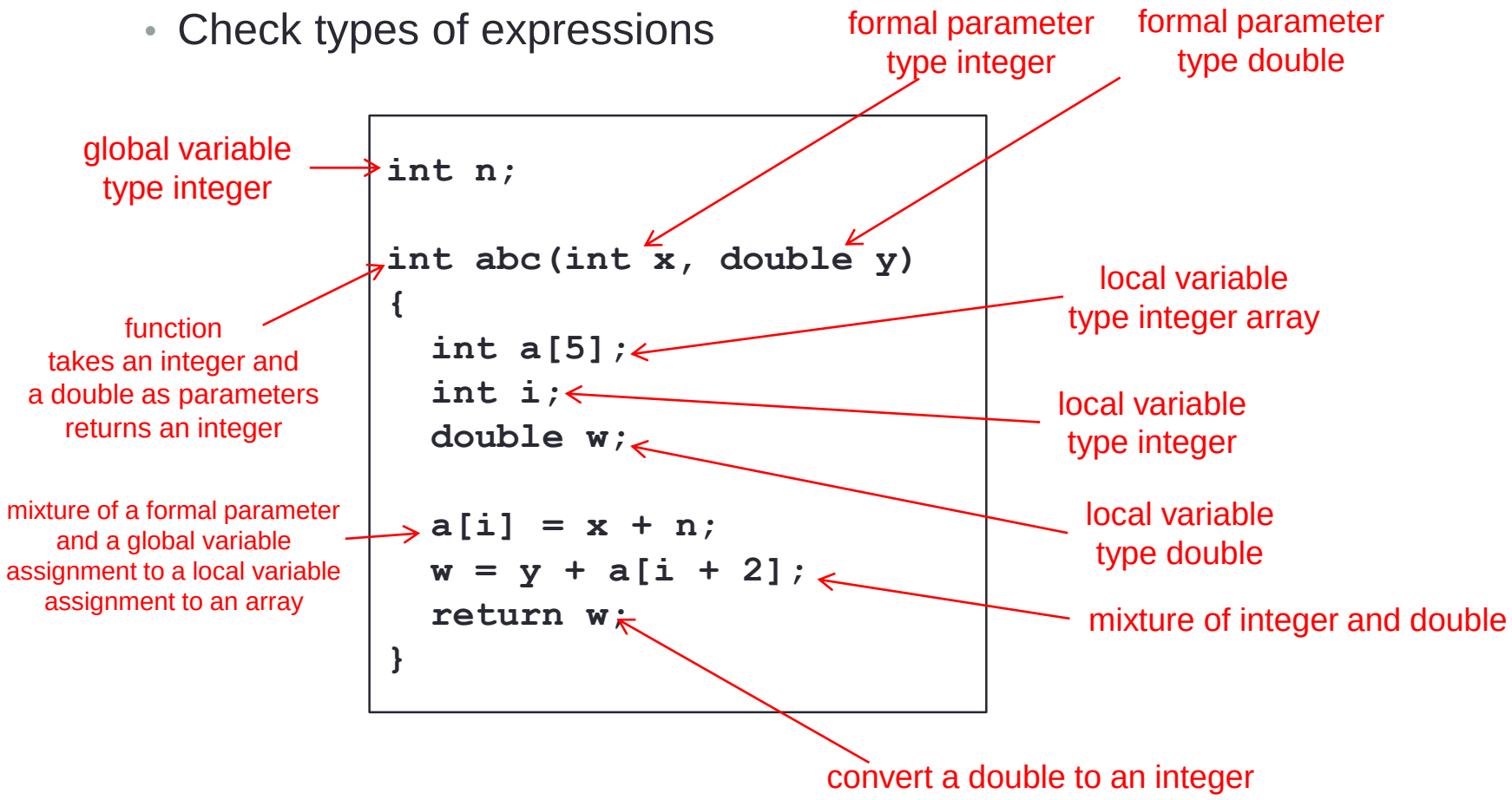


BNF for C Programming Language Statements

```
<statement> ::=

     <expr> ';' |

     'if' '(' <expr> ')' <statement> ( 'else' <statement> )? |

     'while' '(' <expr> ')' <statement> |

     'for' '(' <expr> ';' <expr> ';' <expr> ')' <statement> |

     'switch' '(' <expr> ')' '{'
           (('case' <constant> | 'default') ':' <statement>*)* '}' |

     'do' <statement> 'while' '(' <expr> ')' |

     '{' <statement>* '}'
```

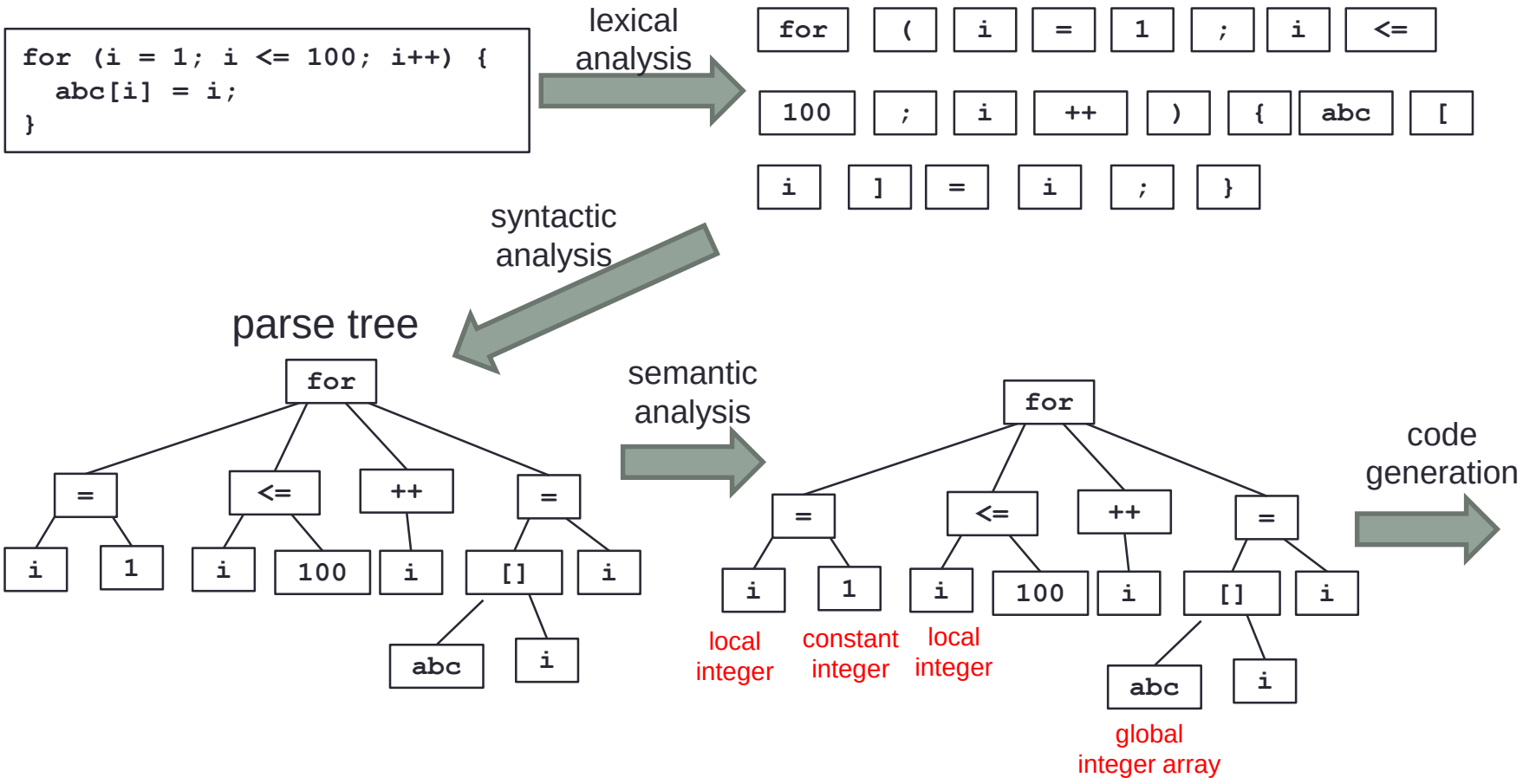
Semantic Analysis

- Even if a program is syntactically correct, it may not be translated
  - Variables need to be declared beforehand.
  - Check types of expressions



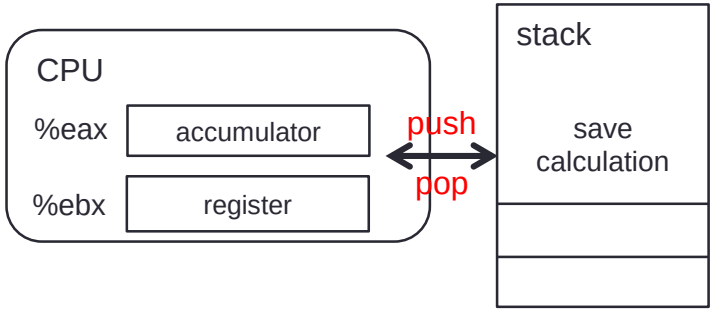
Code Generation

- After lexical analysis, syntactic analysis and semantic analysis, then object code can be generated.

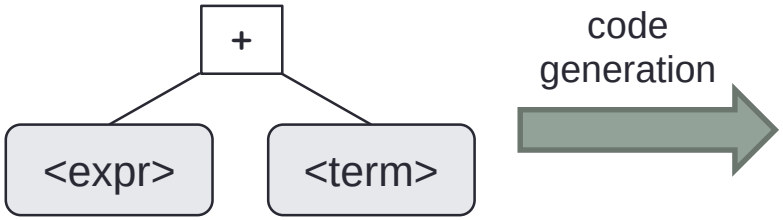


Code Generation

- Code generation
  - Generate code for the target CPU and architecture
  - Variable and parameter references are resolved.
- How to generate code
  - Generate code for parse tree
  - From top to bottom
- Example
  - Generate code for addition
  - i386 code as an example



The calculation result is always in %eax



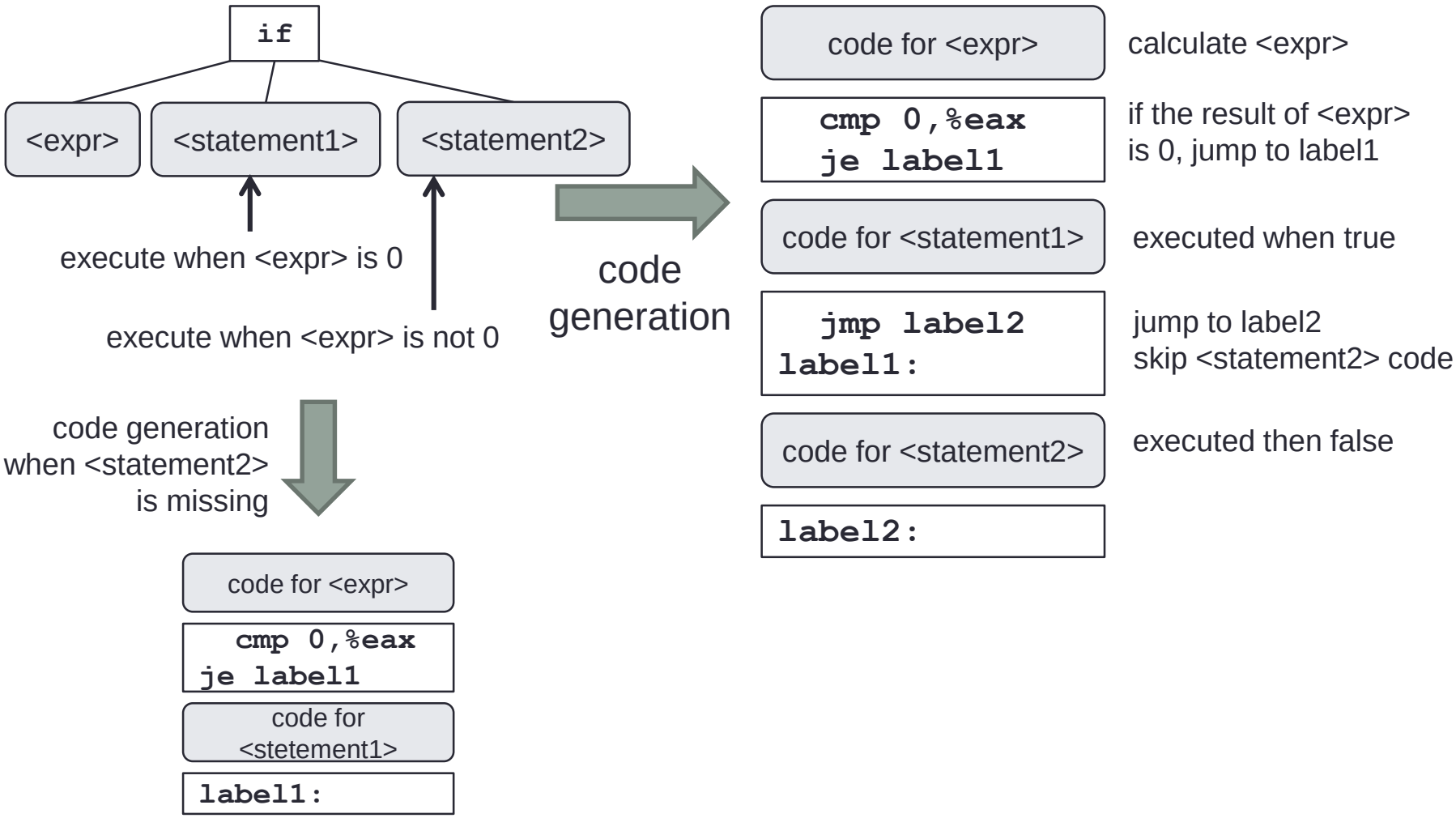

```
<expr> ::= <expr> '+' <term>
```

code for <expr>	calculate <expr>
`pushl %eax`	save the result of <expr>
code for <term>	calculate <term>

`movl %eax,%ebx`
`popl %eax`
`addl %ebx,%eax`
	use saved <expr>
	add <term> result

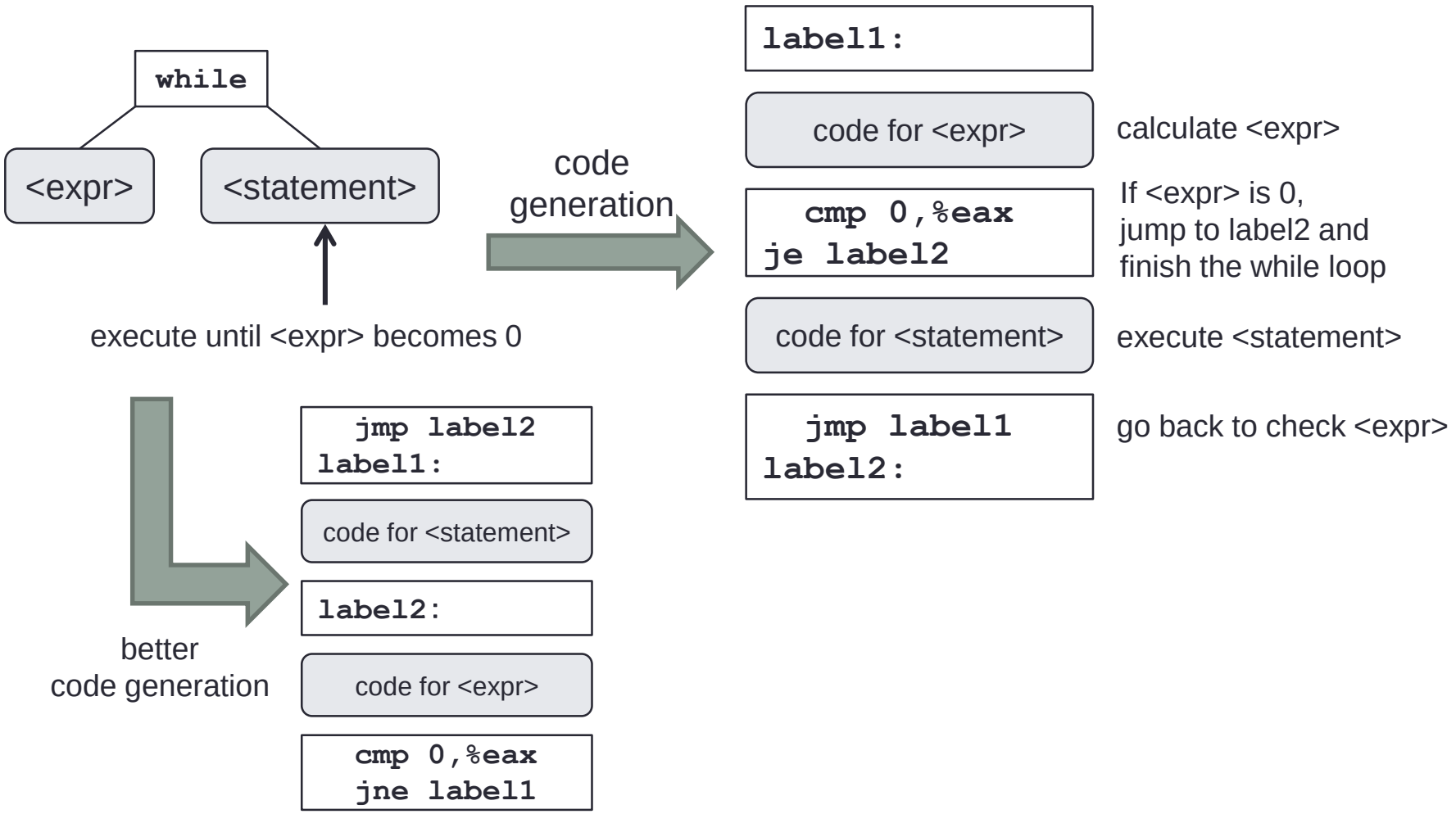
Code Generation for if Statement

```
<if> ::= 'if' '(' <expr> ')' <statement1> ( 'else' <statement2> )?
```



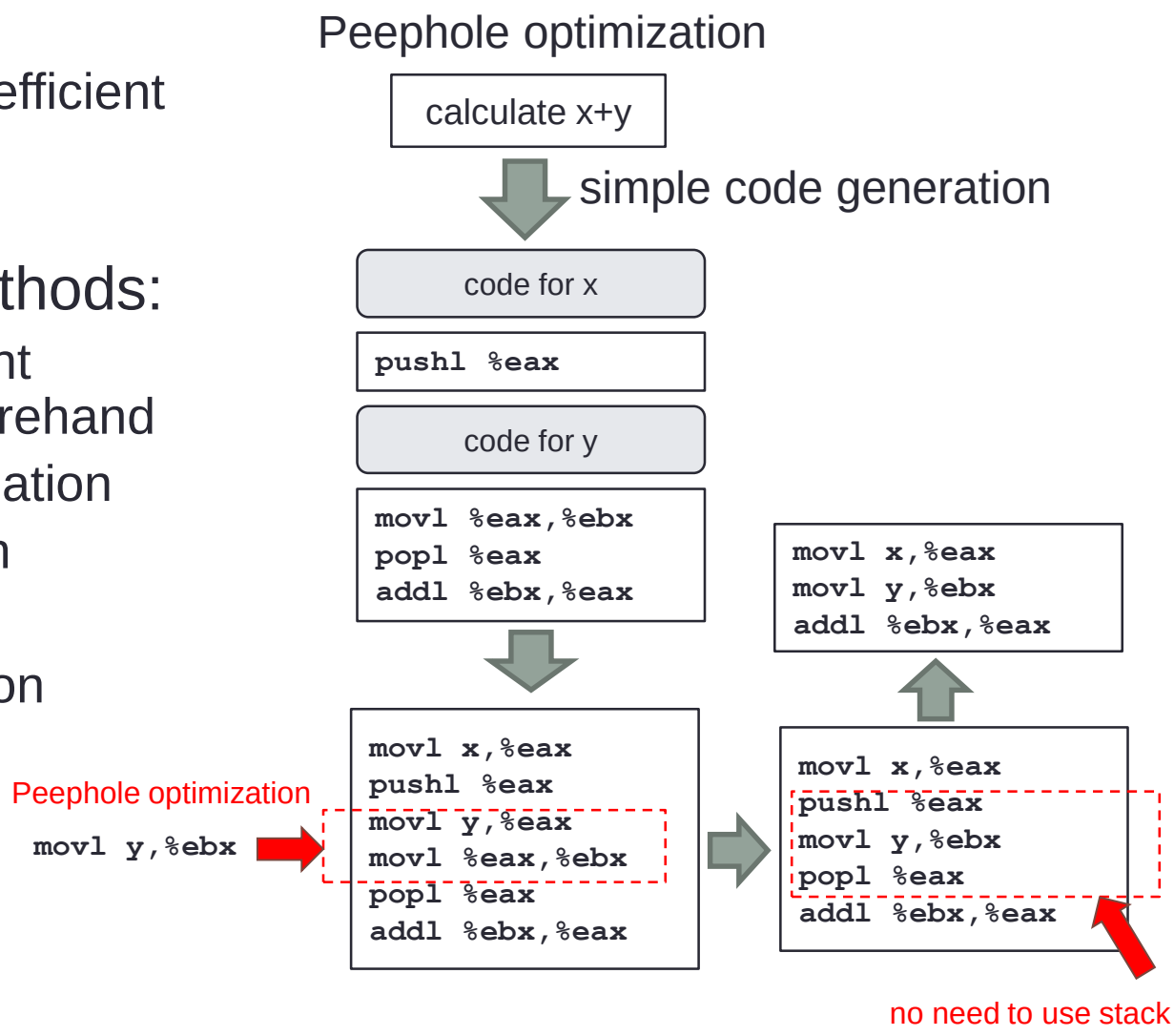
Code Generation for while Statement

```
<while> ::= 'while' '(' <expr> ')' <statement>
```



Optimization

- Optimization
  - Convert to more efficient code
- Optimization methods:
  - Calculate constant expressions beforehand
  - Peephole optimization
  - Loop optimization
  - Use registers
  - Global optimization



Loop Optimization

- Move out things which do not change in the loop
- Use addition rather than multiplication

```
for (i = 0; i < 1000; i++) {
  b = x * x + 3;
  a = a + b + i * 4;
}
```

move out things which do not change

```
b = x * x + 3;
for (i = 0; i < 1000; i++) {
  a = a + b + i * 4;
}
```

convert multiplication to addition

```
b = x * x + 3;
j = 0;
for (i = 0; i < 1000; i++) {
  a = a + b + j;
  j = j + 4;
}
```

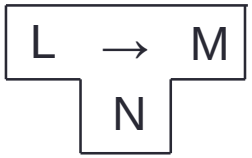
calculation cost

addition subtraction < multiplication < division

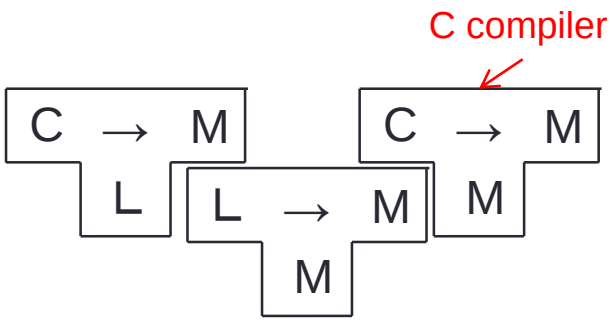
Compiler Programming

- C language compiler is written in a high-level language
  - The compiler needs to be compiled.
  - The first C language compiler needs to be written in a different language.
  - Once having C compiler, C language compiler may be written in C itself.

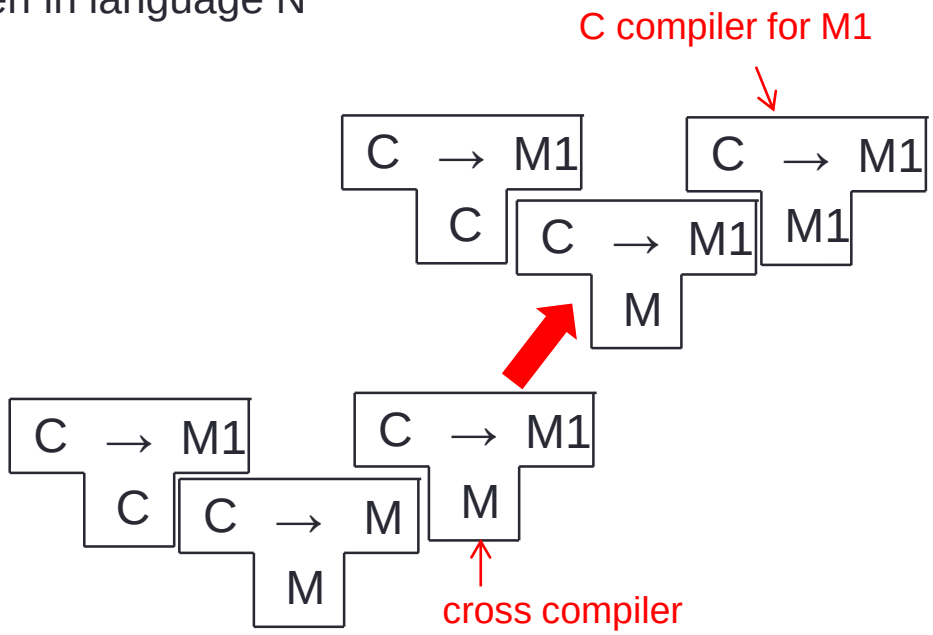
- T scheme



Compiler for language L to language M written in language N



C compiler written in L
use L compiler



port C compiler to M1

Compiler Compiler

- Generate compilers easily
  - Generate lexical analyzer
    - lex
    - Use regular expressions to express tokens
  - Generate syntactic analyzer
    - yacc
    - Yet Another Compiler Compiler
    - Syntactic analyzer is generated from CFG

Summary

- High-level Programming Language Processing
  - Interpreter
  - Compiler
- Compiler components
  - preprocess
  - lexical analysis
  - syntax analysis
  - code generation
  - optimization