

Introduction

Programming a Computer

- High level language (HLL) program: easy for human understanding and writing program.
- Machine language program: a CPU can understand only this form and act.
- Assembly language program: intermediate form, a readable version of machine language program (and more).

A Compiler

- A **compiler** is a program that accepts a **HLL program** (source language) as input. It **translates** the program to a **target language program** preserving the **semantics**.
- An example of source language may be C++ and the target language may be the **assembly** or **machine language** of some processors e.g. **x86-64**, or some **virtual machine**.

Compiling a Compiler

- A compiler for a language A may be written as a HLL program CP_A in some language B .
- Use a compiler of B , C_B , to compile CP_A to generate the executable code, C_A , of a compiler for A .
- If $A = B$, we are compiling a new compiler for the language A - a process known as bootstrapping.

An Interpreter

- An **interpreter** is a type of translator that works in a different mode.
- Normally it does not produce a complete translated version of the machine code. It **interprets** the translated **intermediate representation** of the HLL program, and performs necessary **action on data**.

An Interpreter

```
#!/usr/bin/python3
# Recursive function to compute gcd()

def gcd(a, b):
    if(b == 0): return a
    return gcd(b, a%b)

m = int(input("Enter a +ve integer: "))
n = int(input("Enter a +ve integer: "))
print('gcd('+str(m)+' , '+str(n)+' ) = ' +str(gcd(m,n)))
```

An Interpreter

```
$ ./gcd.py
```

```
Enter a +ve integer: 12
```

```
Enter a +ve integer: 18
```

```
gcd(12,18) = 6
```

C Program

```
#include <stdio.h>
int main() // first0.c
{
    printf("My first program\n");
    return 0;
}
```

Assembly Language Program

```
$ cc -Wall -S first0.c ⇒ first0.s
```

```
.file    "first0.c"  
.text  
.section .rodata  
.LC0:  
.string  "My first program"  
.text  
.globl   main  
.type    main, @function  
main:  
.LFB0:  
.cfi_startproc
```

```
endbr64
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $0, %eax
popq    %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
```

```
.size    main, .-main
.ident   "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 8
.long    1f - 0f
.long    4f - 1f
.long    5
0:
.string   "GNU"
1:
.align 8
.long    0xc0000002
.long    3f - 2f
```

```
2:
    .long    0x3
3:
    .align 8
4:
```

Object File

```
$ cc -c first0.s ⇒ first0.o  
$ objdump -d first0.o | less
```

```
0000000000000000 <main>:
```

0:	f3 0f 1e fa	endbr64	
4:	55	push	%rbp
5:	48 89 e5	mov	%rsp,%rbp
8:	48 8d 3d 00 00 00 00	lea	0x0(%rip),%rdi
f:	e8 00 00 00 00	callq	14 <main+0x14>
14:	b8 00 00 00 00	mov	\$0x0,%eax
19:	5d	pop	%rbp
1a:	c3	retq	

#

Executable File

```
$ cc first0.o ⇒ a.out
```

```
$ objdump -d a.out | less
```

```
00000000000001149 <main>:
```

```
1149:  f3 0f 1e fa                endbr64
114d:  55                        push  %rbp
114e:  48 89 e5                mov   %rsp,%rbp
1151:  48 8d 3d ac 0e 00 00    lea   0xeac(%rip),%rdi
1158:  e8 f3 fe ff ff        callq 1050 <puts@plt>
115d:  b8 00 00 00 00        mov   $0x0,%eax
1162:  5d                        pop   %rbp
1163:  c3                      retq
```

a.out file contains more code than this.

Using Software Interrupt: x86-64

```
#include <asm/unistd.h>
#include <syscall.h>
#define STDOUT_FILENO 1

.file "first.S"
.section          .rodata
L1:
    .string "My First program\n"
L2:
.text
.globl _start
```

```
_start:
    movl    $(SYS_write), %eax        # eax <-- 1 (write)
                                         # parameters to 'write'
    movq    $(STDOUT_FILENO), %rdi    # rdi <-- 1 (stdout)
    movq    $L1, %rsi                 # rsi <-- starting
                                         # address of string
    movq    $(L2-L1), %rdx             # rdx <-- L2 - L1
                                         # string length

    syscall                           # software interrupt
                                         # user process requesting
                                         # OS for service
    movl    $(SYS_exit), %eax          # eax <-- 60 (exit)
                                         # parameters to exit
```

```
movq  $0, %rdi          # rdi <-- 0
syscall                  # software interrupt

ret                      # return
```

Preprocessor, assembler and Linker

```
$ /lib/cpp first.S first.s  
$ as -o first.o first.s  
$ ld first.o  
$ ./a.out  
My first program
```

```
first.s
```

```
.file "first.S"
```

```
.section .rodata
```

```
L1:
```

```
    .string "My First program\n"
```

```
L2:
```

```
.text
```

```
.globl _start
```

```
_start:
```

```
    movl $(1), %eax # eax <-- 1 (write)
```

```
                                # parameters to 'write'
```

```
    movq $(1), %rdi # rdi <-- 1 (stdout)
```

```
movq $L1, %rsi # rsi <-- starting
                # address of string
movq $(L2-L1), %rdx # rdx <-- L2 - L1
                # string length
syscall # software interrupt
                # user process requesting
                # OS for service
movl $(60), %eax # eax <-- 60 (exit)
                # parameters to exit
movq $0, %rdi # rdi <-- 0
syscall # software interrupt
ret # return
```

```
objdump -d first.o
```

```
first.o:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

```
0000000000000000 <_start>:
```

```
   0: b8 01 00 00 00      mov     $0x1,%eax
   5: 48 c7 c7 01 00 00 00 mov     $0x1,%rdi
  c: 48 c7 c6 00 00 00 00 mov     $0x0,%rsi
 13: 48 c7 c2 12 00 00 00 mov     $0x12,%rdx
 1a: 0f 05              syscall
 1c: b8 3c 00 00 00      mov     $0x3c,%eax
 21: 48 c7 c7 00 00 00 00 mov     $0x0,%rdi
 28: 0f 05              syscall
 2a: c3                retq
```

```
objdump -d a.out
```

```
a.out:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

```
0000000000401000 <_start>:
```

```
401000:  b8 01 00 00 00      mov    $0x1,%eax
401005:  48 c7 c7 01 00 00 00  mov    $0x1,%rdi
40100c:  48 c7 c6 00 20 40 00  mov    $0x402000,%rsi
401013:  48 c7 c2 12 00 00 00  mov    $0x12,%rdx
40101a:  0f 05              syscall
40101c:  b8 3c 00 00 00      mov    $0x3c,%eax
401021:  48 c7 c7 00 00 00 00  mov    $0x0,%rdi
401028:  0f 05              syscall
40102a:  c3                retq
```

Description/Specification of a Language

- Description of a well-formed program - **syntax** of a language.
- Description of the meaning of different constructs and their composition as a whole program - **semantics** of a language.

Description of Syntax

Syntax of a programming language is specified and verified in two stages.

1. Identification of the **tokens** (atoms of different syntactic categories) from the character stream of a program.
2. Correctness of the syntactic structure of the program from the stream of **tokens**.

Description of Syntax

- Tokens of a programming language are specified using regular expressions.
- Token identification is done by software where at its core there is a modified DFA or an NFA.

Description of Syntax

Structure of a programming language is specified using **restricted class** of context-free grammars e.g. LL(1), LALR(1), or LR(1).

Note

There are structural features of a programming language that are not specified by the grammar rules for efficiency reason and are handled differently.

Description of Meaning: Semantics

- Informal or semi-formal description by natural language and mathematical notations.
- Formal descriptions e.g. grammar rule with attributes, other formal specifications of semantics.

Users of Specification

- Programmer - often uses an informal description of the language construct.
- People who write language translators.
- People who want to verify a piece of program or who want to automate program writing (synthesis).

Source and Target

A source language is usually a high-level language. But there are different types of high level languages.

Imperative languages like C, object oriented languages like Java, functional languages like Haskell, logic programming languages like Prolog, languages for parallel and distributed programming etc.

Source and Target

- The target languages also have a wide spectrum. **Assembly** and **machine languages** of different architecture, byte code of a **virtual machines**, or even another **HLL**.
- Different machine architectures demand different types of code generation and improvements.

Front-end and Back-end

- The part of the compiler that analyzes the structure of the source program, extracts the semantic information and produces an internal representation of it is known as the front-end.
- The part that uses the internal representation and synthesizes the semantically equivalent target program is called the back-end.

Compiler and Interpreter

- A compiler and an interpreter mainly differ in their back-ends.
- A compiler generates the target code from the intermediate representation.
- The target code can be executed again and again.

Compiler and Interpreter

- An **interpreter** on the other hand performs action specified by the **program fragment** (extracted in the **intermediate form**) on its data.
- The **code generating** back-end of a compiler is replaced by a set of routines for **interpretation**.

Compiler and Interpreter

- Usually it is expected that the **compiled code** is more time **efficient**.
- But an interpreter may have **better error reporting** as the source program is available during interpretation.
- An interpreter may be **more portable** and **easier to write (!)**. People also claim that an interpreter is better in terms of **security**.

Basic Phases of Compilation

- **Read the program text** - this is the most time consuming part as it involves I/O.
- **Preprocessing** - a phase before the actual compilation. It may involve inclusion (reading) of several files.
- **Lexical analysis** - identification of the syntactic symbols of the language and collecting their attributes.

Basic Phases of Compilation

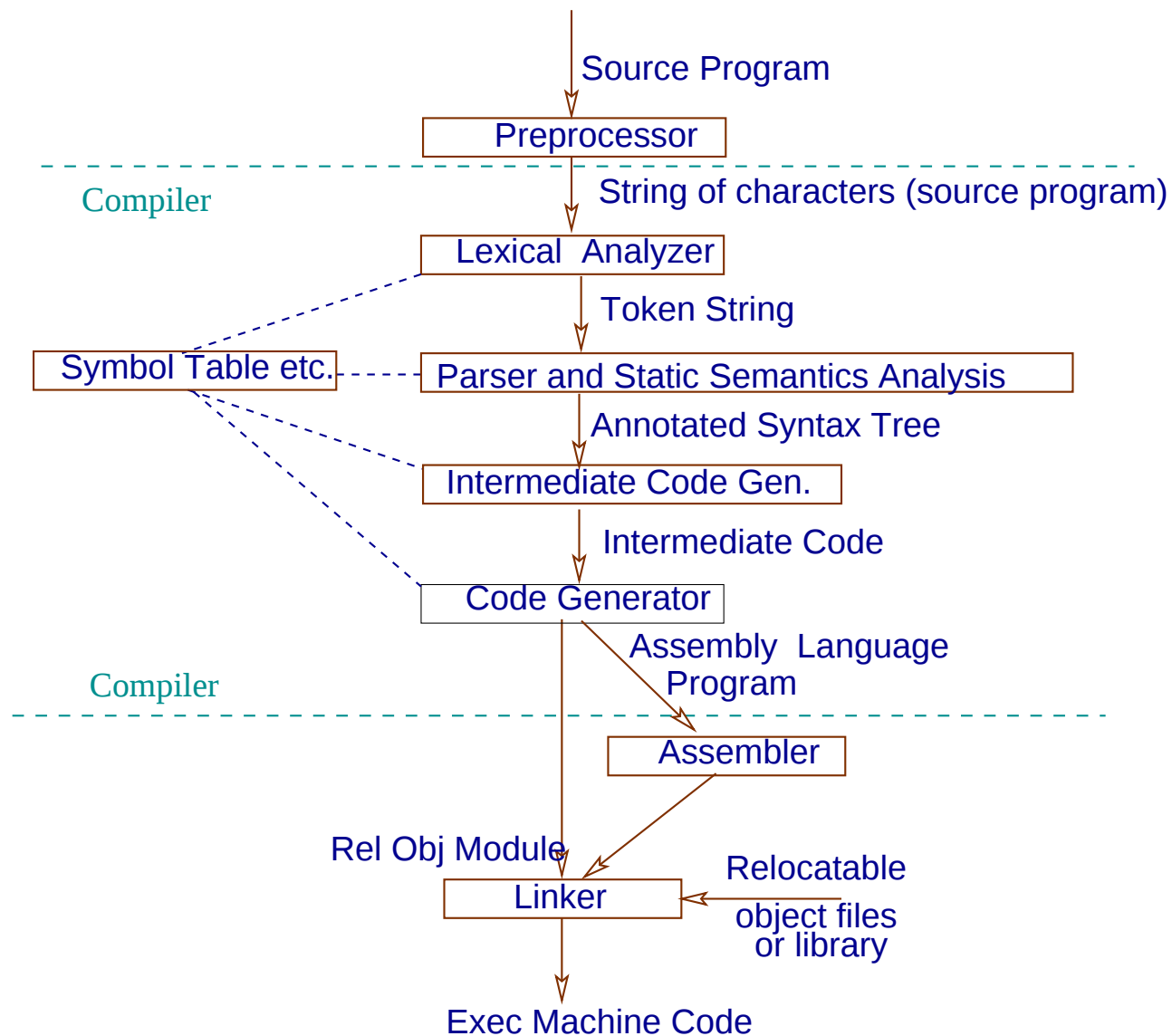
- Syntax checking and static semantic analysis
 - the stream of token is parsed to form the parse tree or syntax tree. The semantic information is collected from the context and the syntax tree is annotated.

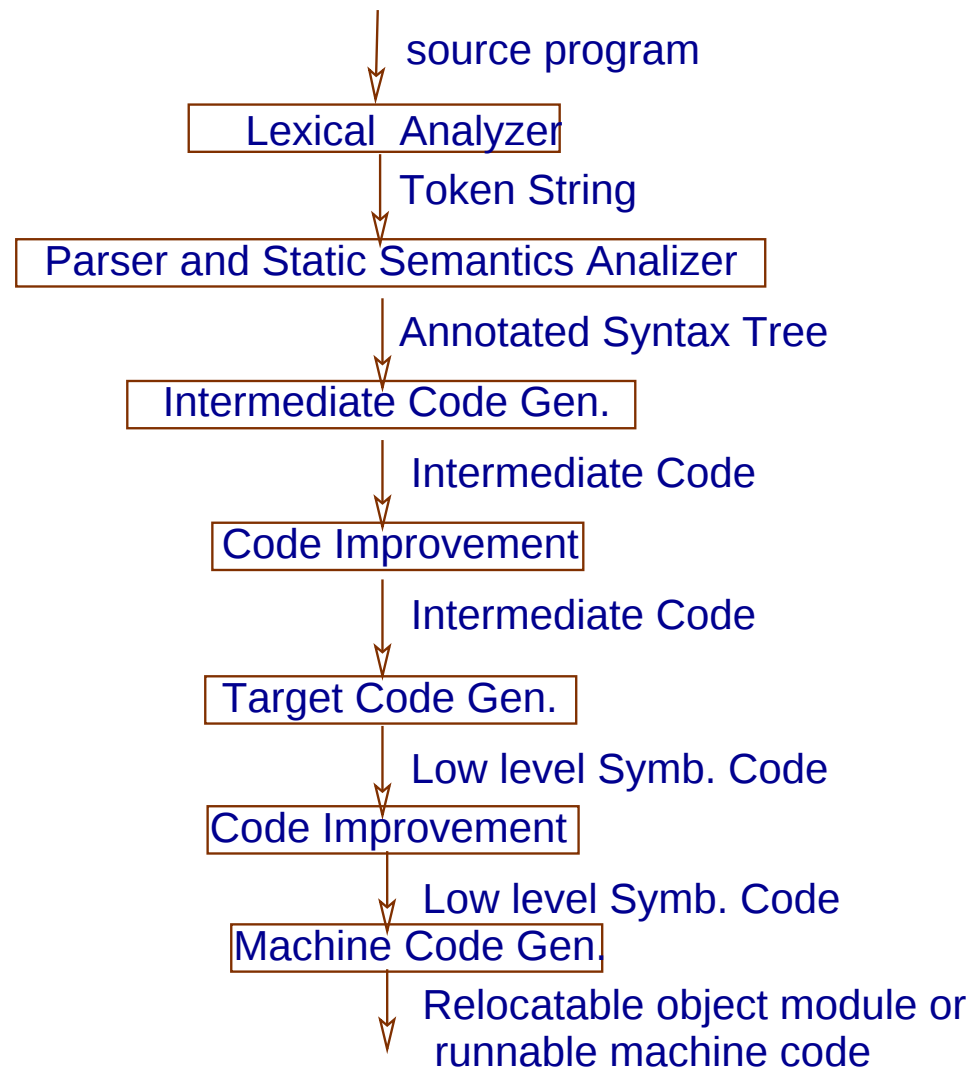
Basic Phases of Compilation

- **Intermediate code generation** and **code improvement** - language specific constructs are translated to more general and simple constructs. As an example a long expressions is broken down to expressions with fixed number of parameters. Different optimizations are performed on this intermediate code.

Basic Phases of Compilation

- Symbolic target code generation and architecture specific code improvement - the intermediate representation is translated to target code in symbolic form, and architecture specific code optimization is performed.
- Low level machine code generation.





Independence of Front and back Ends

- In an idealistic situation the **front-end** of a compiler does not know anything of the **target language**. Its job is to transform the **source program** to intermediate representation.
- Similarly, the **back-end** is not aware of the **source language**. It works on the intermediate representation to generate the **target code**.

Compiler and Interpreter

- If the intermediate representation of the source program and the input data to it is available, the ‘translator’ can perform the action on the data specified by the source program using the intermediate representation. There is no need to generate the target code explicitly.
- This is what is done by an interpreter.

Scanner or Lexical Analyzer

A **scanner** or **lexical analyzer** breaks the program text (string of ASCII characters) into the **alphabet** of the language (into **syntactic categories**) called a **tokens**.

A **token** may be encoded as a number and it may have one or more **attributes**.

An Example

Consider the following C function.

```
double CtoF(double cel) {  
    return cel * 9 / 5.0 + 32 ;  
}
```

Scanner or Lexical Analyzer

- A scanner uses the **finite automaton** model to identify different tokens.
- Software e.g. **flex** takes the **specifications** of **tokens** as **regular expressions** and generates a program that works as the **scanner**.
- The process is **completely automated**.

Scanner or Lexical Analyzer

- A **syntax analyzer** does not differentiate between different identifiers or different integer constants. They are identified as **identifier token** and **integer token**.
- But the actual values are preserved as **attributes** for use in subsequent phases.

Syntactic Category, Token and Attribute

String	Type	Token	Attribute
"double"	keyword	302	"CtoF"
"CtoF"	identifier	401	
"("	delimiter	40	
"double"	keyword	302	"cel"
"cel"	identifier	401	
")"	delimiter	41	
"{"	delimiter	123	
"return"	keyword	315	

String	Type	Token	Attribute
“cel”	identifier	401	“cel”
“*”	operator	42	
“9”	int-numeral	504	9
“/”	operator	47	
“5.0”	double-numeral	507	5.0
“+”	operator	43	
“32”	int-numeral	504	32
“;”	delimiter	59	
“}”	delimiter	125	

Parser or Syntax Analyzer

- A **parser** or **syntax analyzer** checks whether the **token stream** generated by the scanner, forms a **valid program**.
- It uses a restricted class of **context-free grammars** as model to specify the language constructs.
- The restriction is primarily from the **efficiency consideration**.

Context-Free Grammar

function-definition $\rightarrow$ decl-spec decl comp-stat

decl-spec $\rightarrow$ type-spec | $\dots$

type-spec $\rightarrow$ **double** | $\dots$

decl $\rightarrow$ d-decl | $\dots$

d-decl $\rightarrow$ ident | ident (**par-list**)

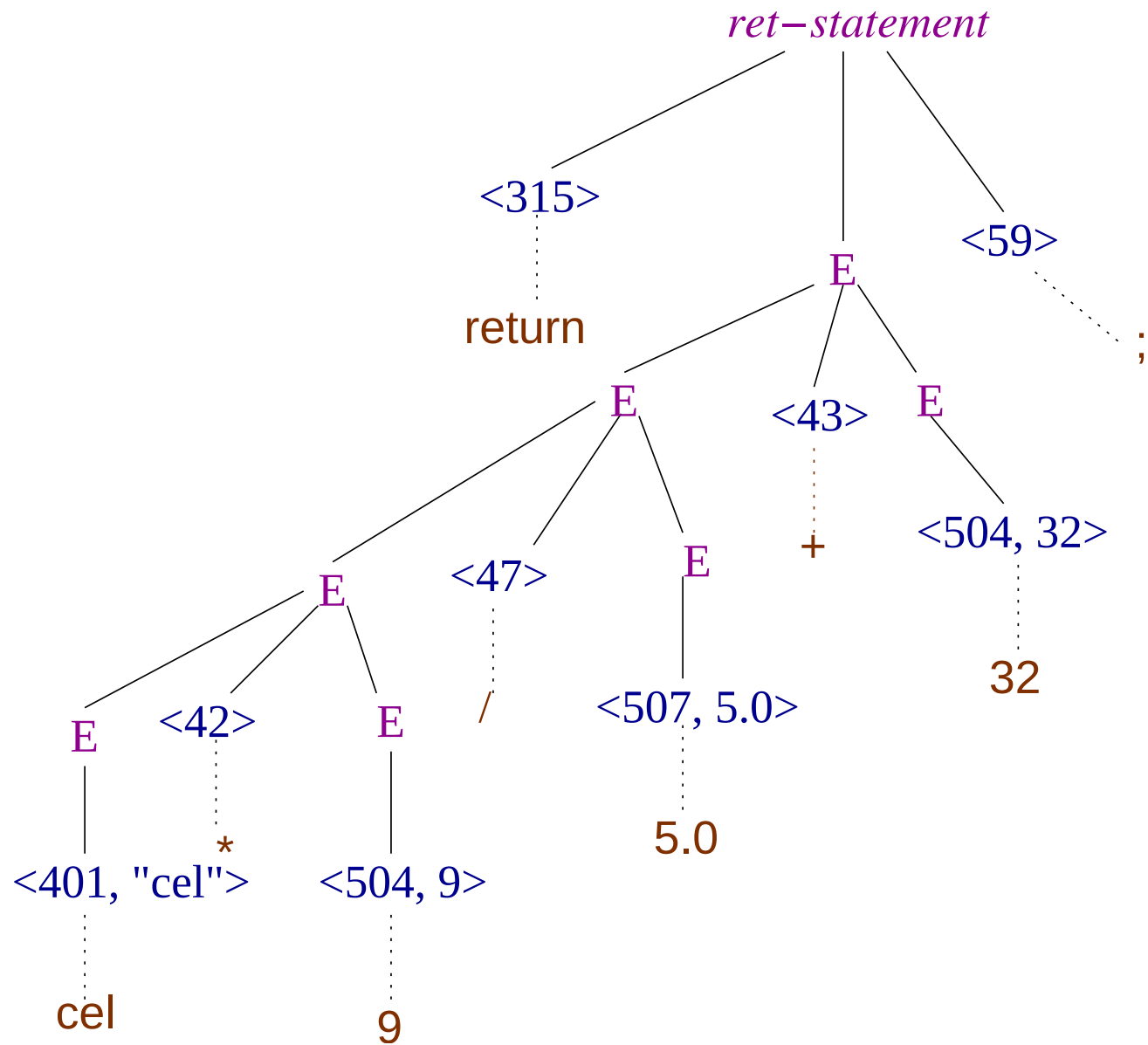
par-list $\rightarrow$ par-dcl | $\dots$

par-dcl $\rightarrow$ decl-spec decl | $\dots$

Expression Grammar

$$\begin{aligned} E \rightarrow & E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \\ & -E \mid \text{var} \mid \text{float-cons} \mid \text{int-cons} \mid \dots \end{aligned}$$

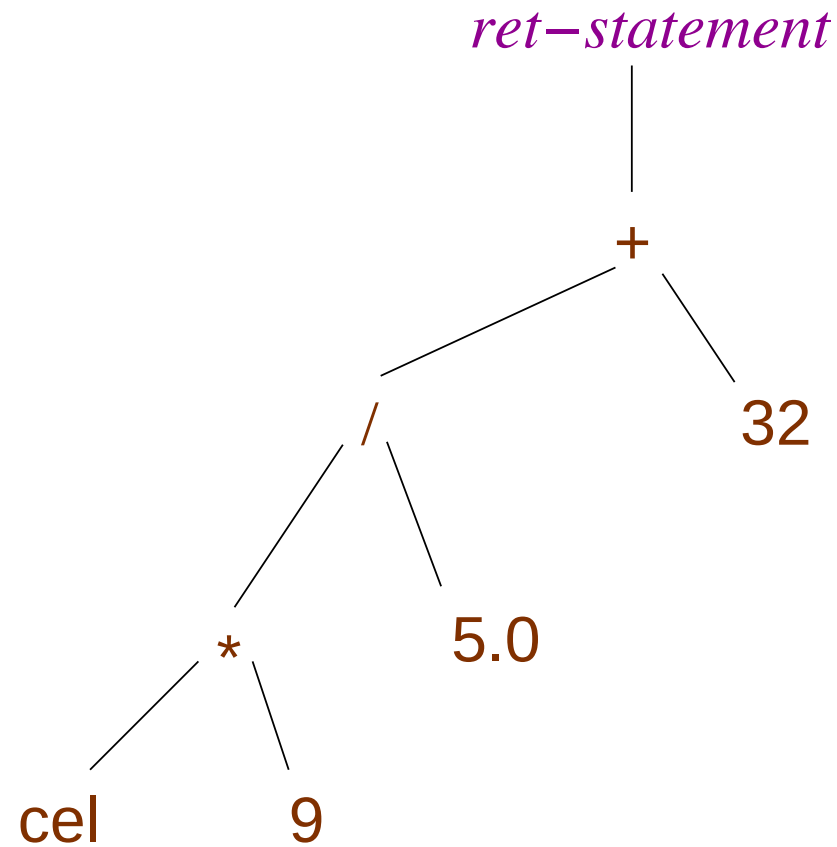
Parse Tree



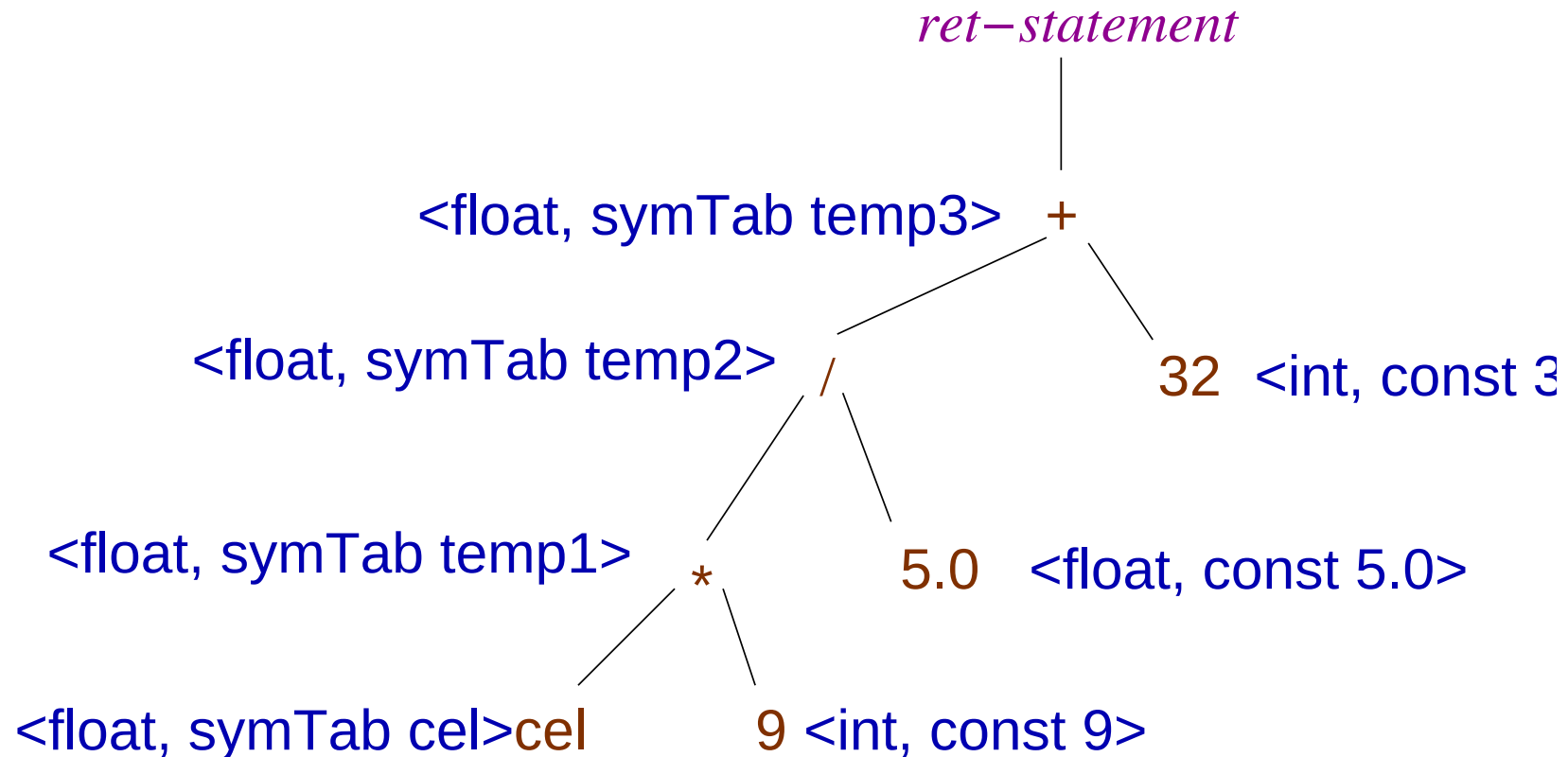
Abstract Syntax Tree

- The **parse tree** generated during syntax checking is often stored in a modified and compact form, known as an **abstract syntax tree (AST)**.
- Semantic information are stored in the nodes of AST (**annotated AST**) as **attributes**. It is used for **intermediate code generation**, **error checking** and **code improvement**.

Abstract Syntax Tree (AST)



Annotated AST



Symbol Table

The compiler maintains an important data structure called the **symbol table** to store variety of names and their attributes it encounters. A few examples are - **variables**, **named constants**, **function names**, **type names**, **labels** etc.

Symbol Table

- The **symbol table** corresponding to the function **CtoF** should have an entry for the parameter **cel** with its type.
- The return type of the function etc. information.

Semantic Analysis

The constant **9** is of type **int**. It is to be converted to **9.0** of type **double** (completely different representation) before it can be multiplied with **cel**.
Similar is the case for **32**.

Semantic Analysis

It is not enough to know that $x = x + 5;$ is syntactically correct.

- The operation is meaningful only if the variable x is declared as a datatype where the binary operation '+' is defined.
- Even if x is a number, the generated code will be different depending on whether it is an int, float or an address.

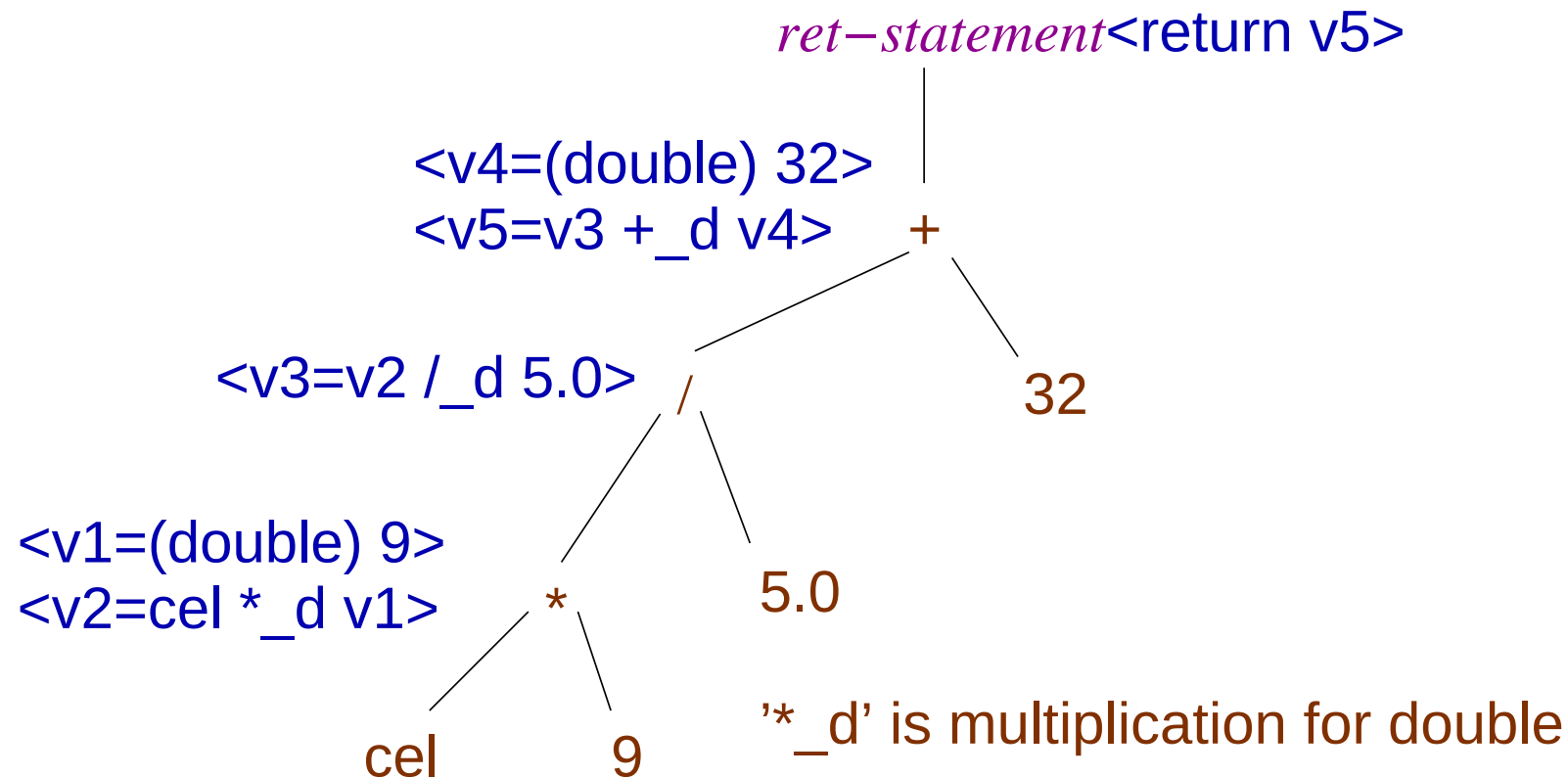
Intermediate Code Generation

- The language specific **AST** may be translated to other intermediate constructs such as **3-address codes**.
- The form of the intermediate code should be suitable for **code improvement** and **target code generation**.
- A **interpreter** may directly work on the AST.

Intermediate Code Generation

- A **while-loop** in a **C-like language** may be replaced by **test**, and **conditional** and **unconditional jumps**.
- A compiler may use more than one intermediate representations for different phases.

Intermediate Code



Intermediate Code

param cel

v1 = (double) 9 # compile time

v2 = cel *_d v1

v3 = v2 /_d 5.0

v4 = (double) 32 # compile time

v5 = v3 +_d v4

return v5

Note

$v1, v2, v3, v4, v5$ are called virtual registers. Finally they will be mapped to actual registers or memory locations. The distinct names of the virtual registers helps better in code improvement (SSA - static single assignment).

GCC IC - GIMPLE

```
$ cc -Wall -fdump-tree-gimple -S ctof.c
```

```
CtoF (double cel)
{
    double D.1796;

    _1 = cel * 9.0e+0;
    _2 = _1 / 5.0e+0;
    D.1796 = _2 + 3.2e+1;
    return D.1796;
}
```

Raw GIMPLE Code

```
$ cc -Wall -fdump-tree-gimple-raw -S ctof.c
```

```
CtoF (double cel)
```

```
gimple_bind <
```

```
  double D.1796;
```

```
  gimple_assign <mult_expr, _1, cel, 9.0e+0, NULL>
```

```
  gimple_assign <rdiv_expr, _2, _1, 5.0e+0, NULL>
```

```
  gimple_assign <plus_expr, D.1796, _2, 3.2e+1, NULL>
```

```
  gimple_return <D.1796 NULL>
```

```
>
```

Intermediate Code Improvement

- Different code improvement transformations are performed on the intermediate code.
- A few examples are constant propagation, constant folding, strength reduction, copy propagation, elimination of common sub-expression, $\dots$, code in-lining etc.

Target Code Generation

- Program variables and temporary variables are allocated to memory and registers.
- Translates the intermediate code to a target language code e.g. sequence of assembly language instructions of a machine.

Target Code Improvement

- The sequence of **target code** e.g. assembly language code of an architecture may be **modified** to **improve** the quality of code.
- It may **replace** a sequence of instructions by a **better sequence** to make the code faster on an architecture.

64-bit Intel Code

```
.file    "ctof.c"
.text
.globl CtoF
.type    CtoF, @function
CtoF:
.LFB2:
    pushq    %rbp                # save old base pointer
.LCFI0:
    movq     %rsp, %rbp          # rbp <-- rsp new base pointer
.LCFI1:
    movsd    %xmm0, -8(%rbp)      # cel <-- xmm0 (parameter)
    movsd    -8(%rbp), %xmm1     # xmm1 <-- cel
```

```
movsd    .LC0(%rip), %xmm0 # xmm0 <--- 9.0, PC relative
                                # addressing of read-only data
mulsd    %xmm0, %xmm1      # xmm1 <-- cel*9.0
movsd    .LC1(%rip), %xmm0 # xmm0 <-- 5.0
divsd    %xmm0, %xmm1      # xmm1 <-- cel*9.0/5.0
movsd    .LC2(%rip), %xmm0 # xmm0 <-- 32.0
addsd    %xmm1, %xmm0      # xmm0 <-- cel*9.0/5.0+32.0
                                # return value in xmm0

leave
ret
.LFE2:
.size    CtoF, .-CtoF
.section          .rodata
.align 8
```

```
.LC0:
    .long    0
    .long    1075970048
    .align   8
.LC1:
    .long    0
    .long    1075052544
    .align   8
.LC2:
    .long    0
    .long    1077936128
```

9.0 in IEEE-754 Double Prec.

63

0 | 1000 0000 010 | 0010 0000 0000 0000 0000

31

0000 0000 0000 0000 0000 0000 0000 0000

Exponent Bias: 1023,

Actual exponent: $1026 - 1023 = 3$.

$9.0_{10} = 1001.0_2 = 1.001 \times 2^3$.

9.0 and .LC0

Interpreted as integer we have the higher order 32-bits as $2^{30} + 2^{21} + 2^{17} = 1075970048$ and the lower order 32-bits as 0.

In the **little-endian** (lsb) data storage, lower bytes comes first.

9.0 and .LC0

```
.align 8  
.LC0:  
.long    0  
.long    1075970048  
is 9.0.
```

```
Improved Code $ cc -Wall -S -O2 ctof.c
```

```
.file    "ctof.c"
.text
.p2align 4,,15
.globl CtoF
.type    CtoF, @function
CtoF:
.LFB2:
    mulsd    .LC0(%rip), %xmm0
    divsd    .LC1(%rip), %xmm0
    addsd    .LC2(%rip), %xmm0
    ret
.LFE2:
```

```
.size    CtoF, .-CtoF
.section          .rodata.cst8,"aM",@progbits,8
.align 8
.LC0:
    .long    0
    .long    1075970048
    .align 8
.LC1:
    .long    0
    .long    1075052544
    .align 8
.LC2:
    .long    0
    .long    1077936128
```

Constant Folding

```
#include <stdio.h>

int main(){ // constFold.c
    int n;

    n = 24*25;
    printf("n: %d\n", n);
    return 0;
}
```

Constant Folding

```
$ cc -Wall -S constFold.c
```

```
movl    $600, -4(%rbp)    # n <-- 600, constant folding
movl    -4(%rbp), %eax    # eax <-- n
movl    %eax, %esi        # 2nd argument, data in n
leaq    .LC0(%rip), %rdi  # 1st argument, format
movl    $0, %eax
call    printf@PLT
movl    $0, %eax
```

Constant Propagation

```
$ cc -Wall -O2 -S constFold.c
```

```
movl $600, %edx      # 3rd argument 600 directly
movl $1, %edi        # edi <-- 1, 1st argument
                     # Stack overflow flag (?)

xorl %eax, %eax
leaq .LC0(%rip), %rsi # 2nd argument, format
call __printf_chk@PLT
xorl %eax, %eax
```

Strength Reduction

```
#include <stdio.h>

int main(){ // strengthRed.c
    int n, m;

    scanf("%d", &n);
    m = 48*n;
    printf("m: %d\n", m);
    return 0;
}
```

Strength Reduction

```
$ cc -Wall -S strengthRed.c
```

```
movl    -16(%rbp), %edx    # edx <-- n
movl    %edx, %eax         # eax <-- edx (n)
addl    %eax, %eax         # eax <-- 2n
addl    %edx, %eax         # eax <-- eax + edx (3n)
sall    $4, %eax           # eax <-- 2**4*(3*n)
                                # arithmetic shift left
                                # by 4-bits
```

Strength Reduction: not Beneficial

```
#include <stdio.h>

int main(){ // strengthRed1.c
    int n, m;

    scanf("%d", &n);
    m = 47*n;           // 48 replaced by 47
    printf("m: %d\n", m);
    return 0;
}
```

Strength Reduction: not Beneficial

```
$ cc -Wall -S strengthRed1.c
```

```
movl    -16(%rbp), %eax
```

```
imull   $47, %eax, %eax
```

Common Subexpression

```
#include <stdio.h>
int main(){ // commonSubExp.c
    int n, a, b;

    scanf("%d", &n);
    a = 47*n;
    b = 47*n+2;
    printf("a: %d, b: %d\n", a, b);
    return 0;
}
```

Common Subexpression: no Elimination

```
$ cc -Wall -S commonSubExp.c
```

```
movl    -20(%rbp), %eax    # eax <-- n
imull   $47, %eax, %eax    # eax <-- 47*n
movl    %eax, -16(%rbp)    # a <-- eax (47n)
movl    -20(%rbp), %eax    # eax <-- n
imull   $47, %eax, %eax    # eax <-- 47*n
addl    $2, %eax           # eax <-- eax + 2 (47n+2)
movl    %eax, -12(%rbp)    # b <-- eax (47n+2)
```

Common Subexpression: Elimination

```
$ cc -Wall -O2 -S commonSubExp.c
```

```
imull $47, 4(%rsp), %edx # edx <-- 47*n (param a)
leaq   .LC1(%rip), %rsi  # rsi <-- 2nd param (format)
movl   $1, %edi          # edi <-- 1, 1st param
leal   2(%rdx), %ecx      # ecx <-- rdx + 2 (param b)
                                # rdx contains 47*n
```

Loop-Invariant Code

```
#include <stdio.h>
int main(){ // loopInv.c
    int n, sum=0, i;

    scanf("%d", &n);
    for(i=1; i<5*n+7; ++i) sum = sum+i;
    printf("sum: %d\n", sum);
    return 0;
}
```

Loop-Invariant Code: no Code Motion

```
$ cc -Wall -S loopInv.c
```

```
    movl    $1, -12(%rbp)      # i <-- 1
    jmp     .L2                # goto .L2
.L3:
    movl    -12(%rbp), %eax     # eax <-- i
    addl    %eax, -16(%rbp)     # sum <-- sum + i
    addl    $1, -12(%rbp)      # i++
.L2:
    movl    -20(%rbp), %edx     # edx <-- n
    movl    %edx, %eax          # eax <-- edx (n)
    sall    $2, %eax            # eax <-- 4*n
    addl    %edx, %eax          # eax <-- 4n+n
```

```
addl    $6, %eax           # eax <-- 5n+6 (!)
cmpl    %eax, -12(%rbp)    # compare i and 5n+6
jle     .L3                # goto .L3 if i <= 5n+6
```

Loop-Invariant Code: Code Motion

```
$ cc -Wall -O2 -S loopInv.c
```

```
    movl    4(%rsp), %eax          # eax <-- n
    leal    (%rax,%rax,4), %ecx     # ecx <-- 4n+n
    addl    $7, %ecx              # ecx <-- 5n+7
    movl    $1, %eax              # eax <-- 1 (i)
    xorl    %edx, %edx            # edx <-- 0 (sum)
.L3:
    addl    %eax, %edx             # edx <-- edx (sum)+eax (i)
    addl    $1, %eax              # i <-- i+1
    cmpl    %eax, %ecx            # compare i and 5n+7
    jne     .L3                   # if i not = 5n+7
                                #      goto .L3
```