Introduction

Programming a Computer

- Machine language program
- Assembly language program
- High level language program

C Program Using System Call

```
#include <unistd.h>
int main() // first1.c
{
    char *str = "My first program\n";
    write(1, str, 17);  // STDOUT_FILENO=1
    _exit(0) ;
}
```

Assembly Language Translation

```
  .file   "first1.c"
  .section  .rodata
.LC0:
  .string  "My first program\n"
  .text
.globl main
  .type  main, @function
main:
  pushl  %ebp
  movl  %esp, %ebp
  andl  $-16, %esp
  subl  $32, %esp
```

```
movl  $.LC0, 28(%esp)
movl  $17, 8(%esp)
movl  28(%esp), %eax
movl  %eax, 4(%esp)
movl  $1, (%esp)
call  write
movl  $0, (%esp)
call  _exit
```

Using Software Interrupt

```
#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
        movl  $(STDOUT_FILENO), %ebx
        movl  $L1, %ecx
        movl  $(L2-L1), %edx
        int   $128
        movl  $(SYS_exit), %eax
        movl  $0, %ebx
        int   $128
```

Preprocessor, assembler and Linker

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

An Assembly Language Program: Intel-64 Xeon

```
#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
        movq  $(STDOUT_FILENO), %rdi
        movq  $L1, %rsi
        movq  $(L2-L1), %rdx
        syscall
        movl  $(SYS_exit), %eax
        movq  $0, %rdi
        syscall
```

Preprocessor, assembler and Linker

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

Simple Library: Printing an Integer

```
#define BUFF 20
void print_int(int n){ // print_int.c
    char buff[BUFF], zero='0';
    int i=0, saveN, j, k, bytes;

    saveN = n;
    if(n == 0) buff[i++]=zero;
    else{
       if(n < 0) {
          buff[i++]='-';
          n = -n;
       }
```

```
    while(n){
        int dig = n%10;
        buff[i++] = (char)(zero+dig);
        n /= 10;
    }
    if(buff[0] == '-') j = 1;
    else j = 0;
    k=i-1;
    while(j<k){
        char temp=buff[j];
        buff[j++] = buff[k];
        buff[k--] = temp;
    }
}
```

```
    buff[i]='\n';
    bytes = i+1;

    __asm__ __volatile__ (
          "movl $4, %%eax \n\t"
          "movl $1, %%ebx \n\t"
          "int $128 \n\t"
          :
          :"c"(buff), "d"(bytes)
    ) ;  // $4: write, $1: on stdin
}
```

Printing an Integer: print_int.h

```
#ifndef _MYPRINTINT_H
#define _MYPRINTINT_H
void print_int(int);
#endif
```

Printing an Integer: main

```
#include <stdio.h>
#include "print_int.h"
int main()
{
    int n;

    printf("Enter an integer: ");
    scanf("%d", &n);
    print_int(n);
    return 0;
}
```

Creating a Library

```
$ cc -Wall -c print_int.c
$ ar -rcs libprint_int.a print_int.o
$ cc -Wall -c main_print_int.c
$ cc  main_print_int.o -L. -lprint_int
$ ./a.out
Enter an integer: -123
-123
$
```

A Simple Makefile

```
a.out: main_print_int.o libprint_int.a
        cc main_print_int.o -L. -lprint_int

main_print_int.o: main_print_int.c print_int.h

libprint_int.a: print_int.o
                ar -rcs libprint_int.a print_int.o

print_int.o:    print_int.c print_int.h

clean:
        rm a.out main_print_int.o libprint_int.a print_int.o
```

Note

We may copy the library to a standard directory as a superuser. In that case specifying the library path is not necessary.

```
# cp libprint_int.a /usr/lib
# cc main_print_int.o -lprint_int
```

Shared Library

Following are steps for creating a shared library:

```
$ cc -Wall -fPIC -c print_int.c
$ cc -shared -Wl,-soname,libprint_int.so
  -o libprint_int.so print_int.o
```

Perform the following steps as superuser.

Shared Library

`cp libprint_int.so /usr/lib/`

`ldconfig -n /usr/lib/`

The soft-link `libprint_int.so.1` is created under `/usr/lib`. Final compilation:

$ `cc main_print_int.o -lprint_int`

The new `./a.out` does not contain the code of `print_int()`. But it contains code for the corresponding `plt` (procedure linkage table).

Types of High-Level Languages

- Imperative Programming Language
- Functional Programming Language
- Logic Programming Language
- Object-Oriented Programming Language
- Scripting Language

Imperative Languages

Close to *von Neuman architecture*

```
x = x + 5;
```

Read data from a memory location, transform the data and write it back to a memory location.

Examples are: *Fortran*, *Algol*, *PL/I*, *Pascal*, *C* etc[a]

[a]Most of the commercial general purpose programming languages have imperative features.

Note

The presence of the variable '`x`' to the right of the *assignment operator* corresponds to a memory read, and the '`x`' to the left of the operator corresponds to a memory write. The addition operation takes place in the CPU[a]

[a] A smart compiler may keep the variable in a CPU register and by-pass the von Neumam bottle-neck.

Functional Languages

A *pure functional program* may be viewed as a mathematical function, where the output is a function value of the input. There is no high-level notion of *state* (modification of the value of a memory element).

Functional Language Examples

Examples of pure functional languages are:

Miranda, *Haskell* etc.

Lisp, *SML*, *CAML* are functional languages with imperative features.

Features of a Functional Language

A function is a first-class object. It can be used as a *parameter*, can be *returned* as a *value*, can be *assigned* to a variable.

A higher-order function takes a function as a parameter and it may return a function as a value.

Features of a Functional Language

Some functions may be polymorphic i.e. independent of any *concrete type* e.g. length of a list. Recursion and lazy evaluation (unlike *call-by-value*) of parameters are important features.

Functional Program: an OCAML Example

```
$ ocaml
Objective Caml version 3.10.1
# let compose f g = fun x -> g(f(x));;
val compose :
('a -> 'b) -> ('b -> 'c) -> 'a -> 'c =
<fun>
# let rec fact x = if x = 0 then 1
                    else x*fact(x-1);;
val fact :  int -> int = <fun>
# let factOf = compose fact;;
val factOf :  (int -> '_a) -> int -> '_a =
<fun>
```

Functional Program: an OCAML Example

```
# let succ = fun x -> x+1;;
val succ : int -> int = <fun>
# let factOfsucc = factOf succ;;
val factOfsucc : int -> int = <fun>
# factOfsucc 0;;
- : int = 2
# factOfsucc 5;;
- : int = 121
```

An OCAML File: succOfFact.ml

```
let compose f g = fun x -> g(f(x));;
let rec fact x = if x = 0 then 1
                   else x*fact(x-1);;
let factOf = compose fact;;
let succ = fun x -> x+1;;
let succOfFact = factOf succ;;
let main() =
    print_string("Enter a +ve integer: ");
    let n = read_int() in
    Printf.printf "\nsuccOfFact(%d) = %d\n" n
                                     (succOfFact n);;
```

Usin OCAML File: succOfFact.ml

```
$ ocaml
        Objective Caml version 3.10.1


# #use "succOfFact.ml";;
val compose : ('a -> 'b) -> ('b -> 'c) ->
                                  'a -> 'c = <fun>
val fact : int -> int = <fun>
val factOf : (int -> '_a) -> int -> '_a = <fun>
val succ : int -> int = <fun>
val succOfFact : int -> int = <fun>
val main : unit -> unit = <fun>
```

Usin OCAML File: `succOfFact.ml`

```
# main();;
Enter a +ve integer: 5


succOfFact(5) = 121
- : unit = ()
#
```

Logic Programming Languages

A logic program is a collection of *axioms* and *inference rules*. An implementation of a logic program has an inference mechanism. The user of a logic program specifies a *goal*. The implementation tries to infer the goal from the axioms and the inference rules by suitable choice of values for the variables.

A Prolog Predicate

```
$ gprolog
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999-2007 Daniel Diaz
| ?- append(X,Y, [a,b,c]).

X = []
Y = [a,b,c] ?  <Ctrl C>
Prolog interruption (h for help) ?  e
$
```

A Prolog Predicate

```
$ gprolog
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999-2007 Daniel Diaz
| ?- append(X,Y, [a,b,c]).

X = []
Y = [a,b,c] ?  h
Action (; for next solution, a for all
solutions, RET to stop) ?  ;
```

More Output

```
X = [a]
Y = [b,c] ?  a

X = [a,b]
Y = [c]

X = [a,b,c]
Y = []

no
| ?-
```

A Prolog Program

```
| ?- [user].
compiling user for byte code...
fact(0,1).
fact(N, Val) :-
      N > 0,
      M is N-1,
      fact(M, Prev),
      Val is N*Prev.

Use EOF (Ctrl-D).
```

A Prolog Program

```
user compiled, 8 lines read - 796 bytes
written, 75261 ms

yes
| ?- fact(0, V).

V = 1 ?  ;

(1 ms) no
| ?- fact(5, Val).

Val = 120 ?
```

Prolog Program

The first line of the program '`fact(0,1).`' is called an *axioms or fact*.
The last five lines '`fact(N, Val) :- N > 0, M is N-1, fact(M, Prev), Val is N*Prev.`' is called an *inference rule*.
Finally, '`fact(5, Val).`' is called a *goal* or a *query*.

Prolog Program

The *axiom* says that "factorial 0 is 1". The *inference rule* in Horn clause declares, "if `N>0` & `M` is `N-1` & factorial of `M` is `Prev` & `Val` is `N*Prev`, then factorial of `N` is `Val`". The prolog interpreter infers the goal and prints the value of `Val = 120`.

Prolog Program File: `fact.pl`

```
% fact.pl
% Computes factorial
fact(0,1). % unit clause
fact(N, Val) :-  % rule
         N > 0,
         M is N-1,
         fact(M, Prev),
         Val is N*Prev.
```

Compiling Prolog Program File

```
| ?- consult('fact.pl').
compiling ..../fact.pl for byte code...
..../fact.pl compiled, 10 lines read -
            868 bytes written, 7 ms

yes
| ?- fact(5,V).

V = 120 ? ;
```

Compiling Prolog Program File

```
| ?- listing.

fact(0, 1). % unit clause
fact(A, B) :- % rule
        A > 0,
        C is A - 1,
        fact(C, D),
        B is A * D.

(1 ms) yes
| ?-
```

Compiling Prolog Program File

```
$ gplc fact.pl
$ ./fact
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999-2007 Daniel Diaz
| ?- fact(5,X).

X = 120 ?
```

Example: myConcat.pl

```
myConcat([], X, X).
myConcat([H|T], X, [H|Z]) :-
                myConcat(T, X, Z).
| ?- consult('myConcat').
....
| ?-
myConcat([1,2], [a,b,c], [1,2,a,b,c]).
yes
| ?- myConcat([1,2], X, [1,2,a,b,c]).
X = [a,b,c]
```

Example: `myConcat.pl`

```
| ?- myConcat([1,2], [a,b,c], X).
X = [1,2,a,b,c]
| ?- myConcat([1,2], X, [2,a,b,c]).
no
```

Example: isPrefix.pl

```
% prefix.pl
isPrefix([],X).
isPrefix([H|X], [H|Y]) :- isPrefix(X,Y).
```

Example: Another factorial

```
% fact1.pl
factorial(0,V,V).
factorial(N,A,V) :- N > 0,
                    A1 is N*A,
                    N1 is N-1,
                    factorial(N1,A1,V).
```

Object Oriented Languages

We call that a programming language has object oriented features if it supports *data and procedure encapsulation*, *inheritance* of objects, *polymorphism* etc.

Object Oriented Languages

Simula is the oldest language with basic object oriented features. Present day major OO languages with procedural features are - C++, Java, Python etc. OCAML is a functional language with OO features.

A C++ Program

```
// complex.h
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>

class complex {
  private:
      double real, imag ;

  public:
          // Default constructor
      complex() ;
```

```
        // Second constructor
    complex(double r, double i) ;
        // Copy constructor
    complex(const complex &c) ;
        // Methods
    complex addComplex(const complex &c);
        // Operator Overloading
    complex operator+(const complex &c) ;
        // Outstream
    friend std::ostream& operator<<(
           std::ostream& os, const complex &s
                                     );
} ;
#endif
```

A C++ Program

```
// complex.c++: implementation
#include <iostream>
#include <math.h>
#include "complex.h"

complex::complex() {
    real = imag = 0.0 ;
}
complex::complex(double r, double i){
    real = r ; imag = i;
}
```

```
complex::complex(const complex &c){
    real = c.real ; imag = c.imag ;
}
complex complex::addComplex(const complex &c){
    return complex(real+c.real, imag+c.imag);
}
complex complex::operator+(const complex &c){
    return complex(real + c.real, imag + c.imag) ;
}
std::ostream& operator<<(
     std::ostream &sout, const complex &c){
    if(c.imag)
       if(c.imag < 0) sout<<c.real<<"-j"<<-c.imag ;
       else sout<<c.real<<"+j"<<c.imag ;
```

```
    else sout << c.real ;
    return sout ;
}
```

A C++ Program

```
// testComplex.c++ : Testing complex
#include <iostream>
#include "complex.h"
using std::cout;

int main() {
    complex c, d(1.0,2.0), e(3.0,4.0);
    c = d.addComplex(e) ;
    cout << c << "\n" ;
    c=complex(); cout<<c<<"\n";
    c=d+e; cout<<c<<"\n";
}
```

Compiling C++ Program

```
$ g++ -Wall -c complex.c++
$ g++ -Wall complex.o testComplex.c++
$ ./a.out
4+j6
0
4+j6
```

Scripting Languages

According to Wikipedia: *A scripting language ... is a programming language that controls a software application.* Script is distinct from the application programs written in different language(s). Scripts, unlike the applications, are often (but not always) interpreted from the source code.

Examples

- Job control languages : JCL, bash etc.
- Text processing languages: awk, sed, grep, Perl.
- Web browser and web server scripting: ECMAScript, JavaScript
- Embeddable/Extension languages: Tcl
- General purpose scripting languages: Perl, Python, Ruby, Tcl

A bash Script

```
#!/bin/bash
while [ 1 ]
do
   ls -l ./
   sleep 5
done
```

Python Interpreter

```
$ python
Python 2.3.4 (\#1, Nov  4 2004, 14:06:56)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)$]$ on linux2
Type "help", "copyright", "credits" or "license"....
>>> 10 + 2
12
>>> 2**200
1606938044258990275541962092341162602522202993782
                                     792835301376L
>>>
```

Mathematics Library

```
>>> import math
>>> math.pi
3.141592653589793l
>>> math.sin(30*math.pi/180)
0.49999999999999994
>>> math.sqrt(2)
1.4142135623730951
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: math domain error
```

String

```
>>> iit = 'IIT Kharagpur'
>>> len(iit)
13
>>> iit[0], iit[4], iit[-1], iit[-3]
('I', 'K', 'r', 'p')
>>> iit[4:7]
'Kha'
>>> iit[-5:-1]
'agpu'
>>> iit + ' IIT Bombay'
'IIT Kharagpur IIT Bombay'
```

Note

A string cannot be modified (immutable) in Python.

```
>>> iit[4] = 'T'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item
                                 assignment
```

String

```
>>> iit.find('har')
5
>>> iit.split(' ')
['IIT', 'Kharagpur']
>>> iit.lower()
'iit kharagpur'
>>> line = 'This line has five words.'
>>> line.split(' ')
['This', 'line', 'has', 'five', 'words.']
```

Python List

- A Python *list* is a sequence of objects.
- Objects may be of different types.
- The order of an objects in a list is determined by its position.
- This is a *mutable* data i.e. a list can be modified.

List

```
>>> l = [105, 10.5, "105"]
>>> len(l)
3
>>> l[1]
10.5
>>> l = l + ['iit', 2**10]
>>> l.pop(1)
10.5
>>> l
[105, '105', 'iit', 1024]
```

Input/Output

```
>>> x = raw_input("Enter a +ve integer: ")
Enter a +ve integer: 5
>>> if x.isdigit():
...     print int(x)
... else:
...     print x
...
5
```

Python Program

```
# This is the first program (pyProg1.py)
x = raw_input("Enter a +ve integer: ")
if x.isdigit():
   print int(x)
else:
   print x

$ python pyProg1.py
Enter a +ve integer:  5
5
```

Python Program

```
$ python pyProg1.py
Enter a +ve integer:  goutam
goutam
```

Python Script

```
#! /usr/bin/python
# This is the first program (pyProg1.py)
x = raw_input("Enter a +ve integer: ")
if x.isdigit():
   print int(x)
else:
   print x
```

Change Mode

```
$ ./pyProg1.py
bash: ./pyProg1.py: Permission denied
$ ls -l pyProg1.py
-rw-rw-r-- 1 goutam goutam 135 Aug 22 10:10 pyPr
$ chmod 764 pyProg1.py
$ ls -l pyProg1.py
-rwxrw-r-- 1 goutam goutam 135 Aug 22 10:10 pyPr
$ ./pyProg1.py
Enter a +ve integer: 5
5
```

Factorial Function

```
#! /usr/bin/python
# fact1.py calculates factorial of a number
x = raw_input("Enter a +ve integer: ")
if x.isdigit():
   n = int(x)
   fact = 1
   for i in list(range(n+1))[1:]:
       fact *= (i+1)
   print n, '!=', fact
else:
   print "Not a number"
```

Factorial Function

```
#! /usr/bin/python
# fact2.py calculates factorial of a number
x = raw_input("Enter a +ve integer: ")
if x.isdigit():
   n = int(x)
   i, fact = 1, 1
   while i <= n:
     fact = fact*i
     i = i + 1
   print n, "!=", fact
else:
   print 'not a number'
```

Note

The assignment is done in parallel. The scope of `while` is determined by indentation.

Using Command Line

```
#! /usr/bin/python
# fact3.py uses command line arguments
import string, sys
if len(sys.argv)==1:
    print 'Less arguments'
    sys.exit(0)
for m in sys.argv[1:]:
    try:
        n=int(m)
    except:
        print m, "is not a number"
    else:
```

```
i,fact = 1,1
while i <= n:
    fact = fact*i
    i = i + 1
print n,'! = ', fact
```

Running Python Script

```
$ ./fact3.py 3 5 7 goutam
2 ! =  6
5 ! =  120
7 ! =  5040
goutam is not a number
```

List of Primes

```
#! /usr/bin/python
# listPrime.py list of primes within 1..n
def isPrime(n):
    prime = True
    if n <= 1:
        return False
    else:
        i=2
        while i<n :
```

```
            if n%i == 0:
                prime = False
                break
            else:
                i += 1
        return prime
x = raw_input("Enter a natural number: ")
if x.isdigit():
    pList=[]
    n = int(x)
    count = 0
```

```
   for i in list(range(n+1))[2:]:
       if isPrime(i) == True:
          pList.append(i)
          count += 1
   print pList, ":", count
else:
   print 'not a number'
```

Running the Program

```
$ ./listPrime.py
Enter a natural number: 100
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
 79, 83, 89, 97] : 25
```

Recursion

```
#! /usr/bin/python
# fact4.py uses command line arguments
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

```
x = raw_input("Enter a +ve integer: ")
if x.isdigit():
    n = int(x)
    print n, "!=", fact(n)
else:
    print 'Not a number'
```

List Operation

```
#! /usr/bin/python
# list1.py list operations
#
import sys
def insertOrd(l, data):
    llen = len(l)
    if l == [] or l[llen-1] < data:
        l.append(data)
        return
```

```
    else:
        l.append(l[llen-1])
        i = llen-2
        while i > -1 and l[i] > data:
            l[i+1]=l[i]
            i -= 1
        else:
            l[i+1]=data
            return

def delOrd(l, data):
```

```
    i=0
    llen = len(l)
    while i < llen and data > l[i]:
          i += 1
    if i < llen and data == l[i]:
          l.pop(i)
    return

l = []
while True:
      x = raw_input("Enter data, terminate with \
```

```
      if x == 'end':
         break
      elif x.isdigit():
           n = int(x)
           insertOrd(l, n)
      else:
           print 'wrong data'
           sys.exit(0)
print 'List is: ', l
y = raw_input("Enter the data to delete: ")
if y.isdigit():
```

```
    delOrd(l, int(y))
    print 'List after delete: ', l
else:
    print 'wrong data to delete'
```

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 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

Formal language specifications e.g. regular expression, formal grammar, automaton etc. are used to specify the syntax of a language.

Description of Syntax

Regular language specification is used to specify different syntactic categories.

Restricted subclass of the *context-free grammar* e.g. SLR or LALR(1) or LR(1) is used to specify the structure of a syntactically correct program.

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. operational semantics, axiomatic semantics, denotational semantics, grammar rule with attributes etc.

Users of Specification

- Programmer - often uses an informal description of the language construct.
- Implementor of the language for a target machine/language.
- People who want to verify a piece of program or who want to automate the program writing (synthesis).

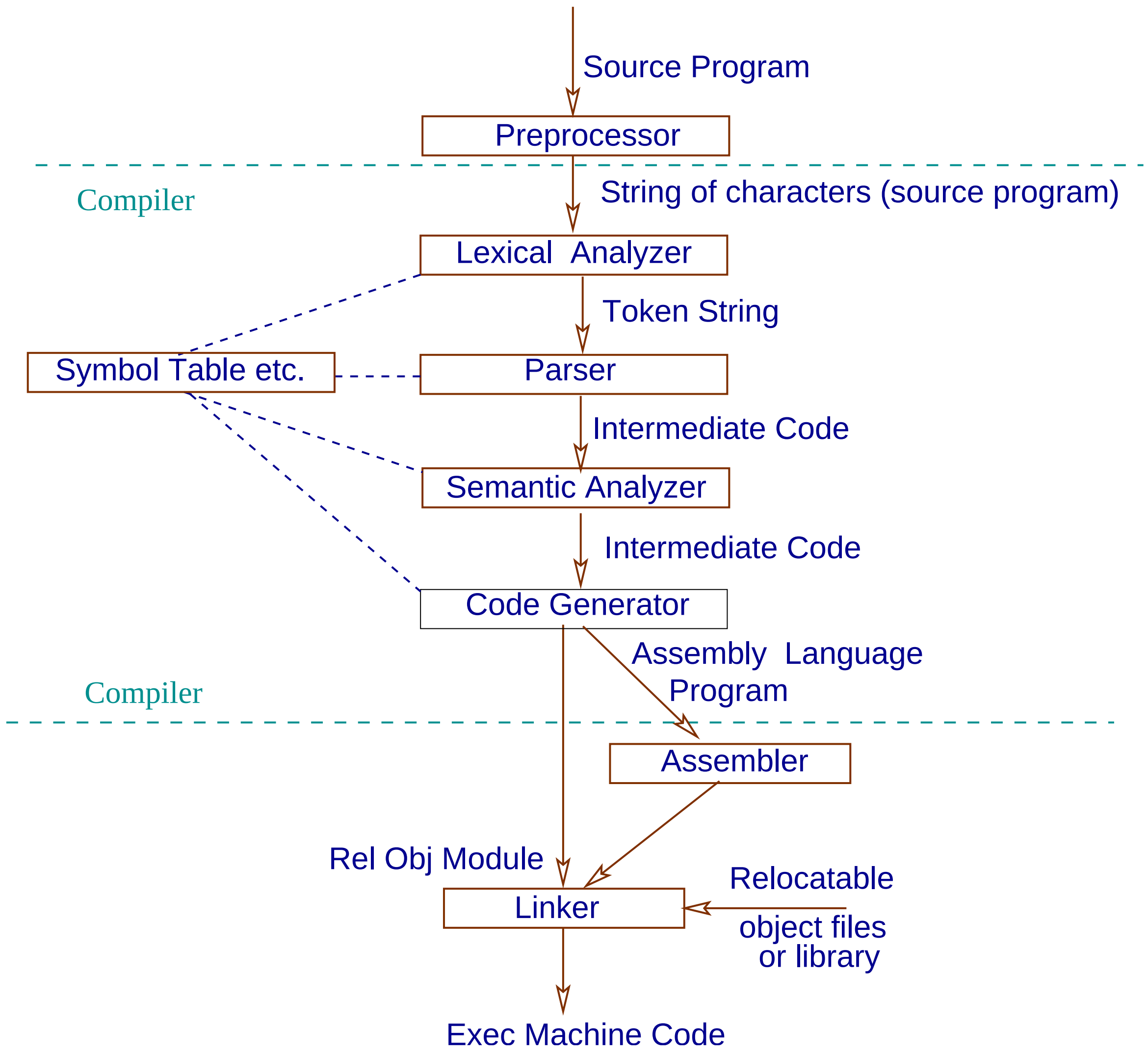
Source and Target

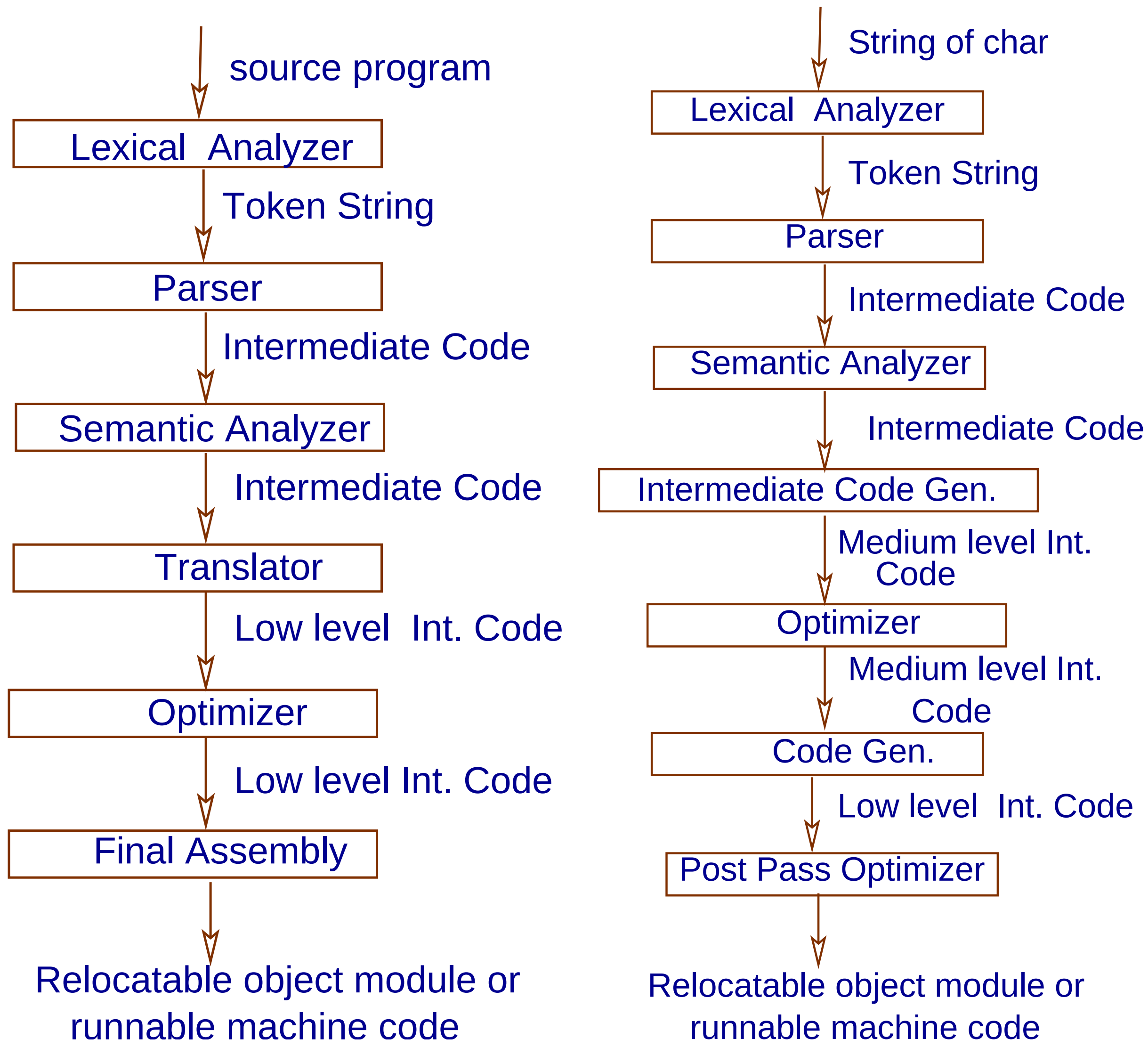
- There are different types of high level languages to be translated.
- Also there are different targets to which a language may have to be translated e.g. another high-level language, assembly languages of different machines, machine language of different actual or virtual machines etc.

- Actual machines may have wide range architectures e.g. CISC, RISC (simple pipeline, super-scalar, dynamic scheduling), VLIW/EPIC etc.

Basic Phases of Compilation

- Preprocessing before the compilation.
- Lexical analysis - identification of the *syntactic* symbols of the language and their attributes.
- Syntax checking and static semantic analysis.
- Code generation and Code improvement.
- Target code generation and improvement.


Source Program
Preprocessor
Compiler
String of characters (source program)
Lexical Analyzer
Token String
Symbol Table etc.
Parser
Intermediate Code
Semantic Analyzer
Intermediate Code
Code Generator
Assembly Language
Program
Compiler
Assembler
Rel Obj Module
Relocatable
Linker
object files
or library
Exec Machine Code


source program
Lexical Analyzer
Token String
Parser
Intermediate Code
Semantic Analyzer
Intermediate Code
Translator
Low level Int. Code
Optimizer
Low level Int. Code
Final Assembly
Relocatable object module or
runnable machine code
String of char
Lexical Analyzer
Token String
Parser
Intermediate Code
Semantic Analyzer
Intermediate Code
Intermediate Code Gen.
Medium level Int.
Code
Optimizer
Medium level Int.
Code
Code Gen.
Low level Int. Code
Post Pass Optimizer
Relocatable object module or
runnable machine code

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* for an alphabet of the language may be a number along with its *attribute*.

A Example

Consider the following C function.

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

Scanner or Lexical Analyzer

The scanner uses the *finite automaton* model to identify different tokens. Softwares are available that takes the specification of the *tokens* (elements of different syntactic categories) in the form of *regular expressions (extended)* and generates a program that works as the scanner. The process is completely automated.

Syntactic Category, Token and Attribute

String	Type	Token	Attribute
"`double`"	keyword	302	
"`CtoF`"	identifier	401	"`CtoF`"
"`(`"	delimiter	40	
"`double`"	keyword	302	
"`cel`"	identifier	401	"`cel`"
"`)`"	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 string* generated by the scannar, forms a valid program. It uses restricted class of *context-free grammars* to specify the language constructs.

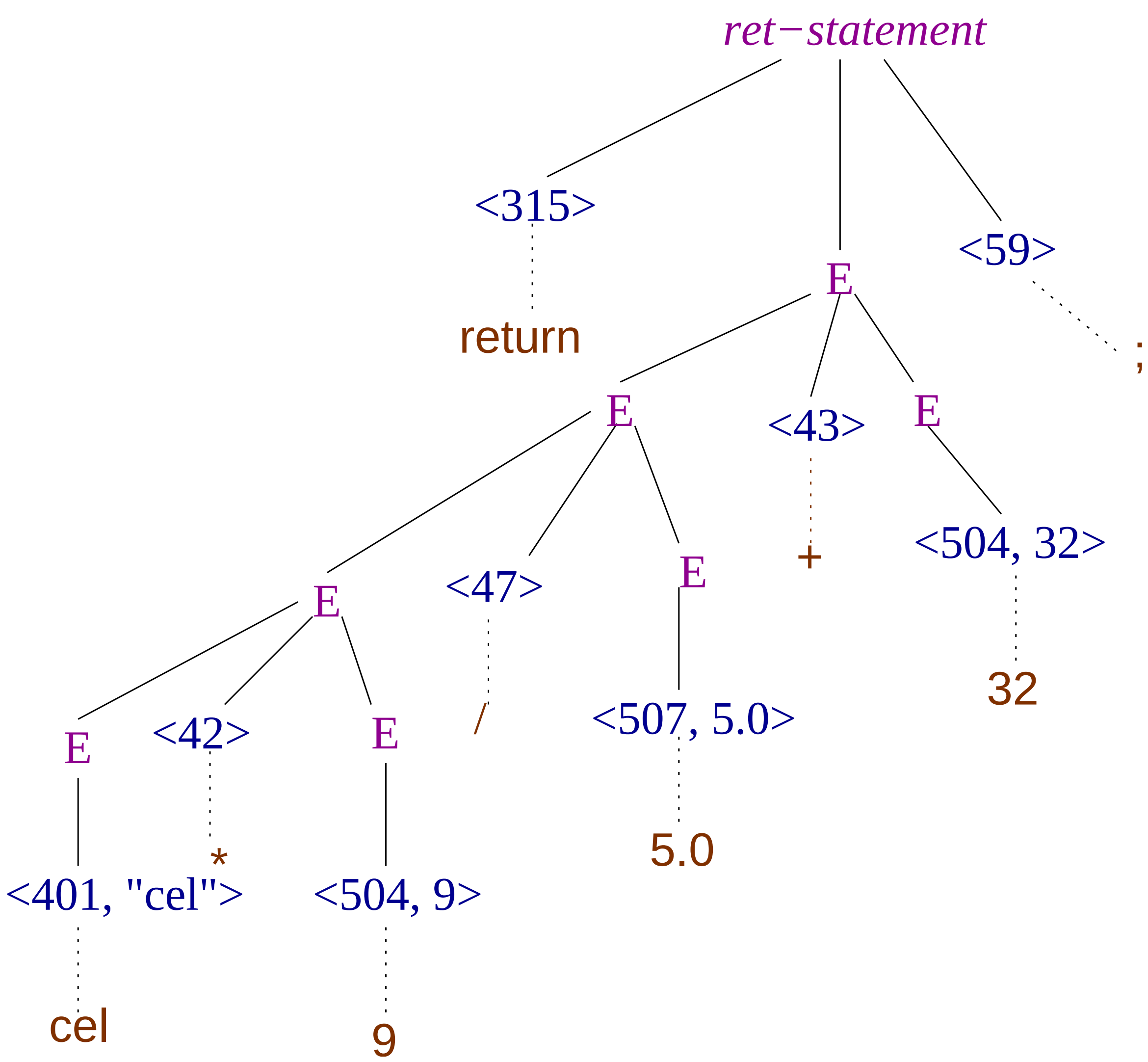
Context-Free Grammar

$$
\begin{aligned}
\textit{function-definition} &\rightarrow \textit{decl-spec decl comp-stat} \\
\textit{decl-spec} &\rightarrow \textit{type-spec} \mid \cdots \\
\textit{type-spec} &\rightarrow \texttt{double} \mid \cdots \\
\textit{decl} &\rightarrow \textit{d-decl} \mid \cdots \\
\textit{d-decl} &\rightarrow \textit{ident} \mid \textit{ident}\ \texttt{(}\ \textit{par-list}\ \texttt{)} \\
\textit{par-list} &\rightarrow \textit{par-dcl} \mid \cdots \\
\textit{par-dcl} &\rightarrow \textit{decl-spec decl} \mid \cdots
\end{aligned}
$$

Expression Grammar

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid$$
$$-E \mid var \mid \mathit{float\text{-}cons} \mid \mathit{int\text{-}cons} \mid \cdots$$

Parse Tree


ret−statement
<315>
return
E
<59>
;
E
<43>
E
+
<504, 32>
32
E
<47>
E
/
<507, 5.0>
5.0
E
<42>
E
*
<401, "cel">
cel
<504, 9>
9

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 example are - variables, named constants, function names, type names, labels etc.

Semantic Analysis

The symbol table corresponding to the function `CtoF` should have an entry for the variable `cel` with its type and other information.
The constant 9 of type `int`. It is to be converted to 9.0 of type `double` before it can be multiplied with `cel`. Similar is the case for 32.

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 regiaters help the compiler to improve the code.

Generation GCC Intermediate Code

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

The generated file is: ctof.c.004t.gimple

GCC Intermediate Code - gimple

```
CtoF (double cel) {
  double D.1248;
  double D.1249;
  double D.1250;

  D.1249 = cel * 9.0e+0;
  D.1250 = D.1249 / 5.0e+0;
  D.1248 = D.1250 + 3.2e+1;
  return D.1248;
}
```

Generation GCC Low-level Intermediate Code

$ `cc -Wall -fdump-rtl-expand -S ctof.c`

The generated file is: ctof.c.128r.expand

IA 32 Assembly Code

```
    .file   "ctof.c"
    .text
.globl CtoF
    .type   CtoF, @function
CtoF:
    pushl   %ebp               # Save the base pointer
    movl    %esp, %ebp         # New base pointer points to
                               # address of old base pointer
    subl    $8, %esp           # easp <-- esp - 8
    movl    8(%ebp), %eax      # eax <-- lower 4 byte of the
                               # M[ebp + 4] is the return ad
    movl    %eax, -8(%ebp)     # M[ebp - 8] <-- eax, lower 4
```

```
                                  # of the parameter
movl    12(%ebp), %eax            # eax <-- higher 4 byte of th
movl    %eax, -4(%ebp)            # M[ebp - 4] <-- eax, higher
                                  # of the parameter
                                  # The 8-byte parameter cel is
                                  # M[ebp - 8]
fldl    -8(%ebp)                  # Push floating-point value i
                                  # in the floating-point stack
fldl    .LC0                      # Push 9.0 on the fp stack
fmulp   %st, %st(1)               # Compute cel*9.0 and push in
                                  # stack
fldl    .LC1                      # Push 5.0 on the fp stack
fdivrp   %st, %st(1)              # Compute (cel*9.0)/5.0 and p
                                  # fp stack
```

```
    fldl    .LC2             # Push 32.0 in the fp stack
    faddp   %st, %st(1)      # Add 32.0 and pop fp stack
    leave
    ret
    .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
    .ident    "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
    .section    .note.GNU-stack,"",@progbits
```

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

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:
    pushl    %ebp
    flds     .LC0
    movl     %esp, %ebp
    fmull    8(%ebp)
    popl     %ebp
    fdivs    .LC1
```

```
    fadds   .LC2
    ret
    .size   CtoF, .-CtoF
    .section    .rodata.cst4,"aM",@progbits,4
    .align 4
.LC0:
    .long   1091567616
    .align 4
.LC1:
    .long   1084227584
    .align 4
.LC2:
    .long   1107296256
    .ident   "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
```

```
.section    .note.GNU-stack,"",@progbits
```

64-bit Intel Code (Xeon)

```
    .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 pointe
.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 dat
    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
```

Improved 64-bit Intel Code (Xeon)

```
    .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
```