# CS 333 Compiler Design

Dr. Anup Keshri (anup_keshri@bitmesra.ac.in)
Dr. Supratim Biswas (supratim.biswas@bitmesra.ac.in)

Department  of Computer Science  and  Engineering
BIT Mesra

# Course Information

- Read the basic course information : Course Objectives, Course Outcomes, CO-PO-PSO mapping; syllabus, from the home page.

  Course Objectives : 1. understand the need for compiler in computer engineering

  2. provide a thorough understanding of design, working and implementation of PL

  3. trace the major concept areas of language translation and compiler design

  4. create awareness of functioning & complexity of modern compilers

# Course Information

Course Outcomes : 1. analyze need for compiler for interfacing between user and m/c

2. Explain roles of several phases of compilation process

3. Create awareness of functioning / complexity of modern compilers

4. Outline major concept areas of language translation / compiler design

5. Develop a comprehensive compiler for a given language

6. Apply knowledge for developing tools for NLP

# Course Conduct

- 4 contact hrs / week : 3 lectures and 1 tutorial
- Prescribed Evaluation scheme will be followed
- Prepare for fill in the blanks type of questions in quizzes
- Attendance norms of the institute apply – do not approach the instructors for relaxation in attendance
- Direct positive correlation observed in FLAT between active presence and skill enhancement / better performace
- Course and the associated laboratory course CS 334 will be synchronized to reinforce the concepts and skills.

# Distinctive Features of a Compilers Course

- Unique course in CSE discipline. It bridges deep theoretical concepts to the most recent advances in architectures and operating systems
- Awareness of compiler features helps in increased programming productivity (invest time and effort to know the capabilities of your compiler)
- Two drivers for Research in compilers - design of new programming languages and invention of new architectures.
- Research has produced spectacular enhancements in technology. Designing a working compiler 3 decades back required several human years.
- Today, given m different languages and n different architectures, m*n compilers can be generated in hours / days.
- Research in compilers continues, specially for automatic optimization, automatic parallelization of sequential programs.
- The hupe leaps in hardware technology have not been efficiently expolited till date because software tools have not been able to catch up with the native computational powers of comtemporary High Performance Architectures.

# Complexity of Software Systems

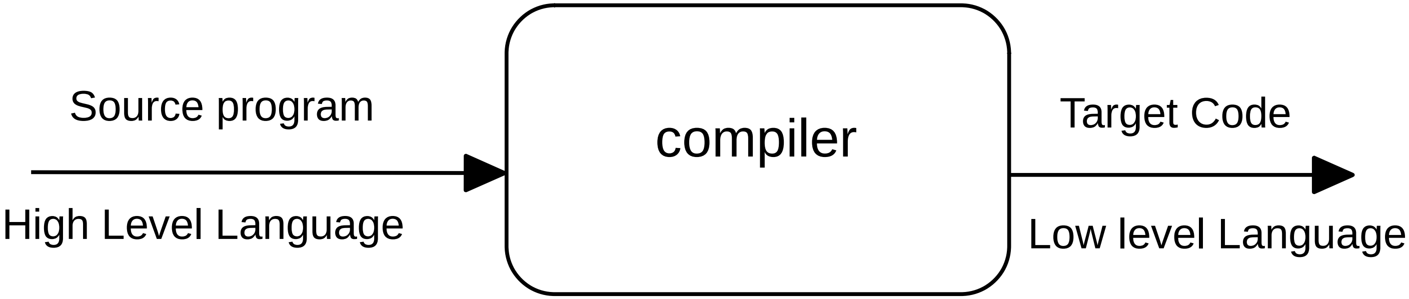
The following table gives an idea about the size and complexity of large software systems from 3 highly popular domains.

The information is only meant to be indicative as the figures are significantly larger today.

| Name | Category | 1st Rel | No of Dirs | No of Files | Lines of Code | White spaces | No of Langs | Main Language |
|---|---|---|---|---|---|---|---|---|
| Linux 4.9-rc.5 | OS Kernel | 1991 | 3830 | 45884 | 20659032 | 36% | 18 | C |
| Gcc 6.2.0 | Compiler Framework | 1987 | 5500 | 79499 | 10638200 | 33% | 33 | C, C++ |
| Mysql 5.7.16 | DBMS | 1995 | 1533 | 10656 | 3901791 | 29% | 27 | C++, C |

# What is a Compiler ?

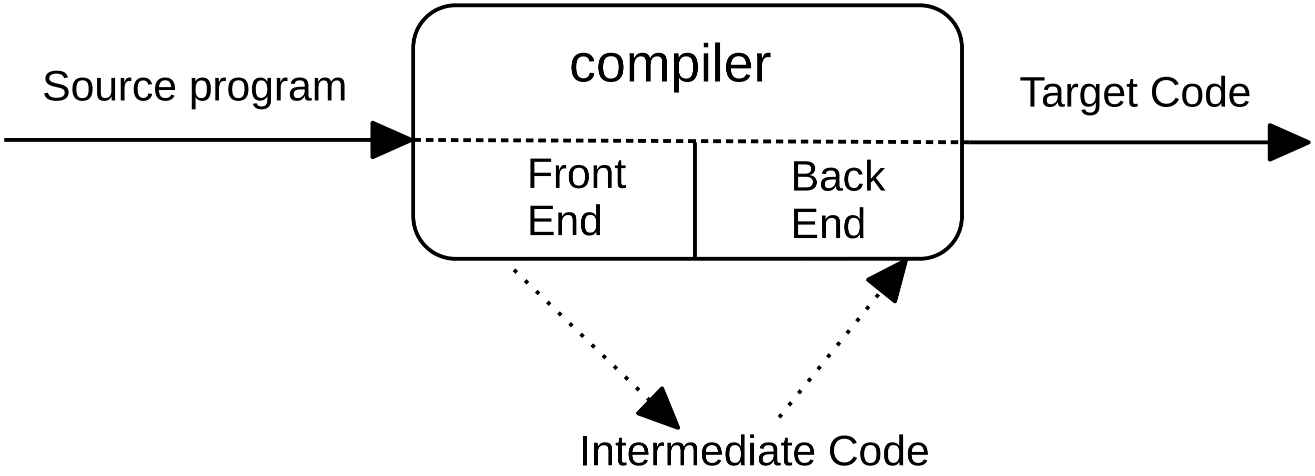
- A compiler is essentially a Translator from a High Level Language (HLL) such as C, C++, Java to a Low Level Language (LLL) such as assembly or machine language.



The difference in the level of abstractions used by HLL and LLL are huge. HLLs focus on application domain computations while LLLs depend on the target architecture.

- The translation process used in a compiler is non-trivial. A compiler slowly brings down the HLL computations to instructions supprted by an architecture.

# Another view of a Compiler

- 

- The front-end of a compiler performs HLL specific analyses and produces semantically equivalent code (intermediate code) whose abstraction level is significantly lower than HLL but still higher than LLL.

- The back-end of a compiler analyses the intermediate code produced by the front-end and is responsible for generating target code.

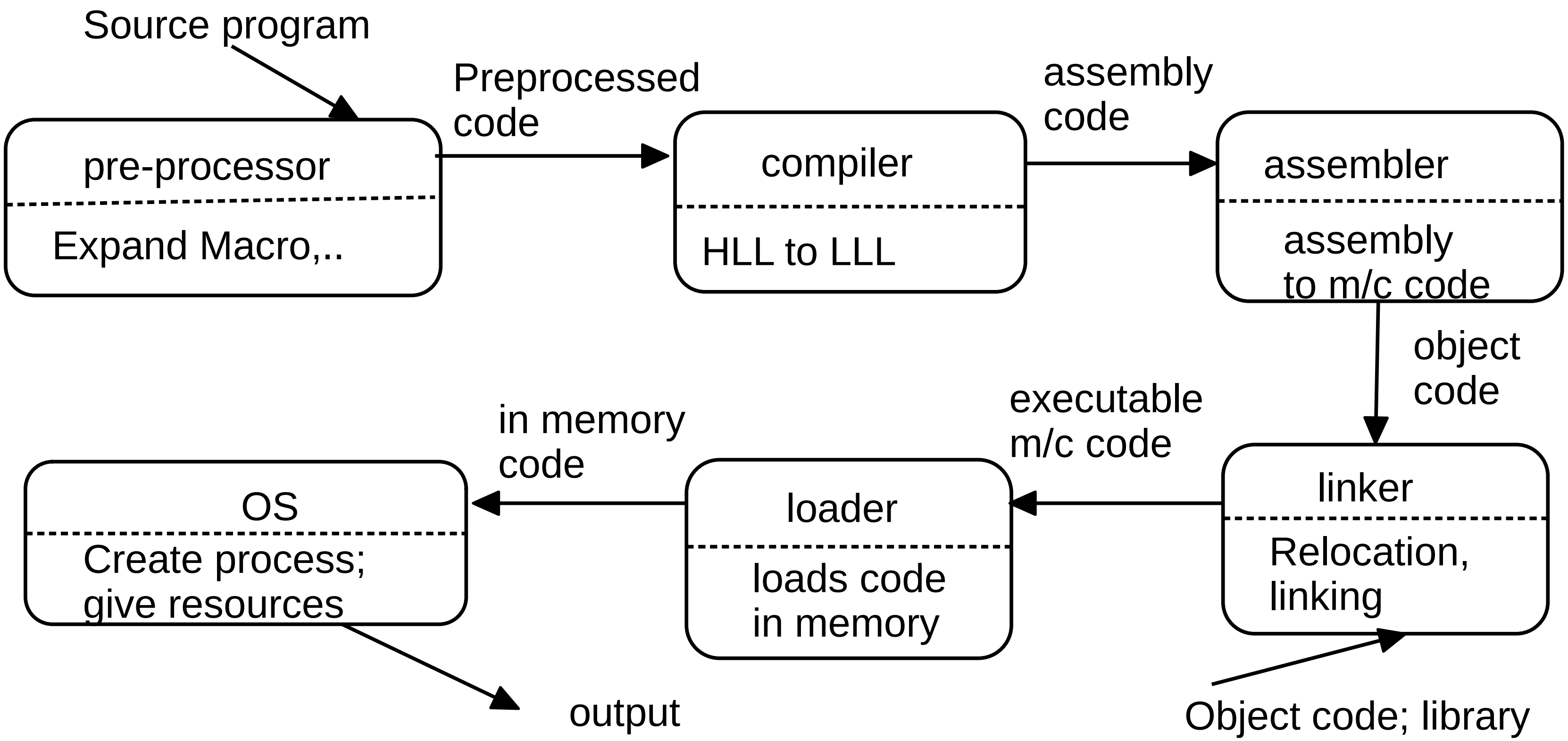
# Compiler Phase and Pass

Phase and pass are two commonly used terms in compiler design.

- The process of compilation is carried out in distinct logical steps, which are called phases.
- A compiler typically has 7 phases : lexical analysis, syntax analysis, semantic analysis, intermediate code generation, run time environment, code optimization and code generation
- A pass denotes a processing of the entire source code or its equivalent form. Industrial quality compilers use large number of passes in their translation process.

# Compiler and Support Software

A compiler is not an end to end software. It uses several system software tools to accomplish its goal.



# Other Models of Translation

A compiler is not the only way to translate code from HLL to LLL.

- Another mechanism, known as Intrepretation, is also used popularly. Such software are called Interpreters.
- In principle a HLL may be either compiled or interpreted, but in practice some languages are compiled and some are interpreted or same use both the models.
- Difference between Compiler and Interpreter
- Identify languages that are interpreted and languages that are compiled.

# THEME 1

## COMPILERS : EXAMPLE DRIVEN APPROACH

# Simple C program

```
#include <stdio.h>
int main()
{    int a[1000], i, j;
     int sum = 10000;
     for (i = 0; i < 1000; i++) a[i] = i;
     for (i = 0; i < 1000; i++)
          for (j = 0; j < i*i; j++)
                a[i] = a[i] + a[j];
     printf(" sum : %d \n", sum);
}
```

# C program – Manual Analysis

Let us manually analyse “firstprog.c”

- What is the role played by the statement ? #include <stdio.h>
- Someone has to supply the prototype of the i/o function printf() in order to check whether the call is valid
- Let us assume there exists a s/w program that will do this task.
- In real world this is task of a tool called C pre-processor, named as “cpp” which is called by C compiler when it encounters “#include... “ and similar other statements.
- For the manual processing we shall ignore this statement for the present.

# C program – Manual Analysis

The remaining statements in “firstprog.c”

- The pretty indented display of the text in the earlier slide was due to an editor which interpreted certain characters before displaying.
- The contents of the raw text in the file is shown below, where the symbol, ↵ represents newline and ↔ denotes a single white space. This is how a compiler sees the program at the first instance.

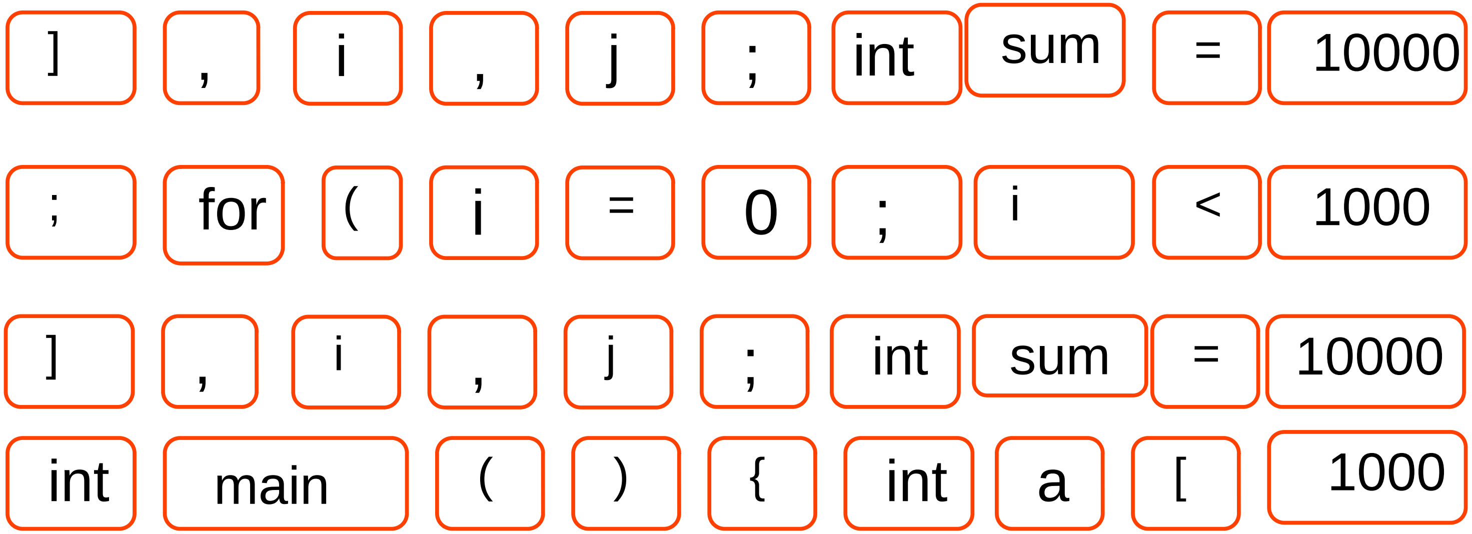
↵int↔main()↵{↵int↔ ↔a[1000],↔i,↔j;↵↔ ↔int
↔sum↔=↔10000;↵↔ ↔for↔(i↔=↔0;↔i↔<↔1
000;↔i++)↔a[i]↔=↔i;↵ ..............................

The first task is then to break the stream of characters shown above into meaningful units of language C.

# Basic Elements of C program

Breaking the program string into meaningful words of the language give the smallest logical units of the language.

- Why is “main” a word and not “main()” is decided by the programming language (PL) specifications and not by a compiler.
- The elements in the red boxes are called tokens (lexemes) and a compiler first converts the character stream into a sequence of tokens.
-



] , i , j ; int sum = 10000

; for ( i = 0 ; i < 1000

] , i , j ; int sum = 10000

int main ( ) { int a [ 1000

# Lexical Analysis

- The process of partitioning a program into its constituent smallest logical words is called as Lexical Analysis.
- Distinction between lexemes and tokens
- Lexeme : instance of the smallest word

  Examples : int main a sum for 1000 < ++ * = .......
- Token : groups of similar lexemes

  Examples : identifier = {i j a sum }

  keyword = {main int for }

  integer = {0 1000 10000}

  operator = {< ++ * = }

# Lexical Analysis

**Lexemes Tokens and Patterns**

- How to describe the lexemes that form the token identifier
- Different PL have variations in the specification of an identifier :

  string of alphanumeric characters with an alphabet as the first character

  string of alphanumeric characters with an alphabet as the first character but length is restricted to 31 characters

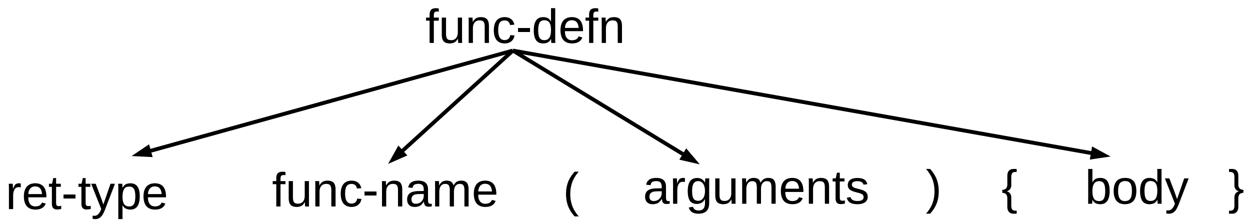
  string of alphabet or numerals or underline (_) characters with an alphabet or underline as the first character with length ≤ 31; characters beyond 31 are ignored

- Decscriptions as given above, may be formal or informal are called as patterns. A pattern when described formally gives a precise description of the underlying token. We are already familiar with the formal notation of regular expression which us used to describe patterns.
- The benefit of formal pattern description is that one can directly design a recognizer (DFA) from the patterns.

# Structure of a Program

Lexical analysis partitions an input program into a stream of tokens (reduces the size of the input).

- The immediate next task is to determine if the consecutive tokens, when grouped together, describe some structure of the PL.
- A PL defines the linguistic structures that constitute a program in the language. For example, a C program is a collection of functions and a function has a hierarchical structure as shown below.
- The structure of func-defn is defined using sub-structures, ret-type, func-name, arguments and body, as shown below.



func-defn

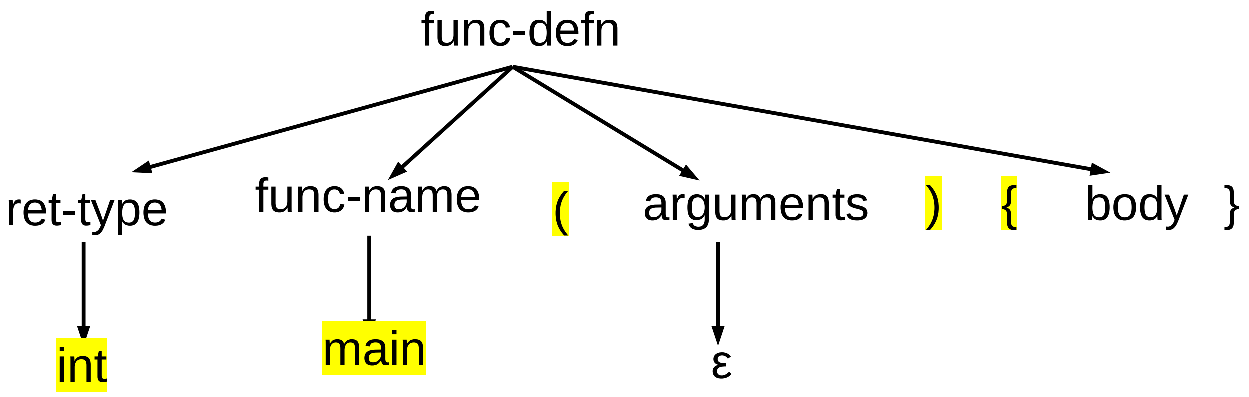
ret-type func-name ( arguments ) { body }

# Structure of a Program

Let  us intuitively apply the ideas to the C program to discover its structure.

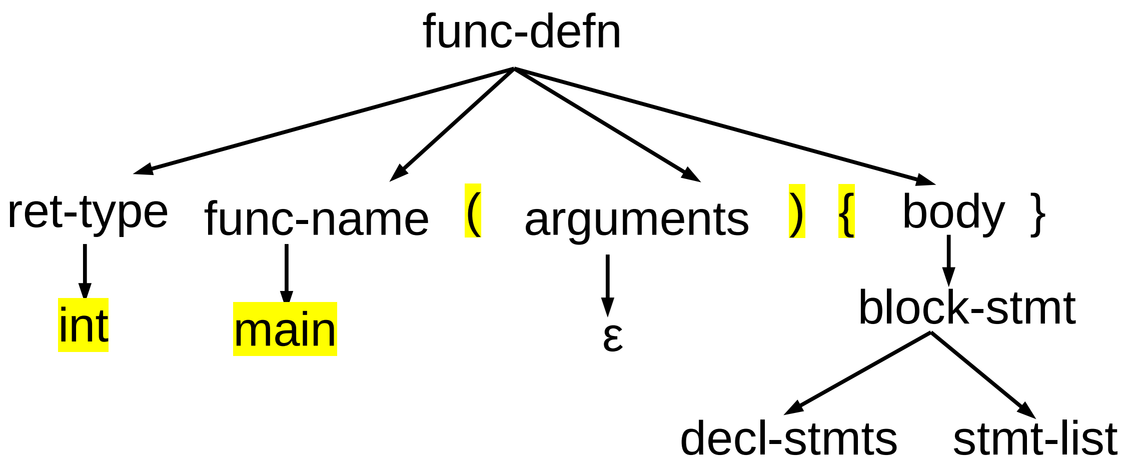
The input “int main() {“

partially matches the program structure as displayed in the tree below.



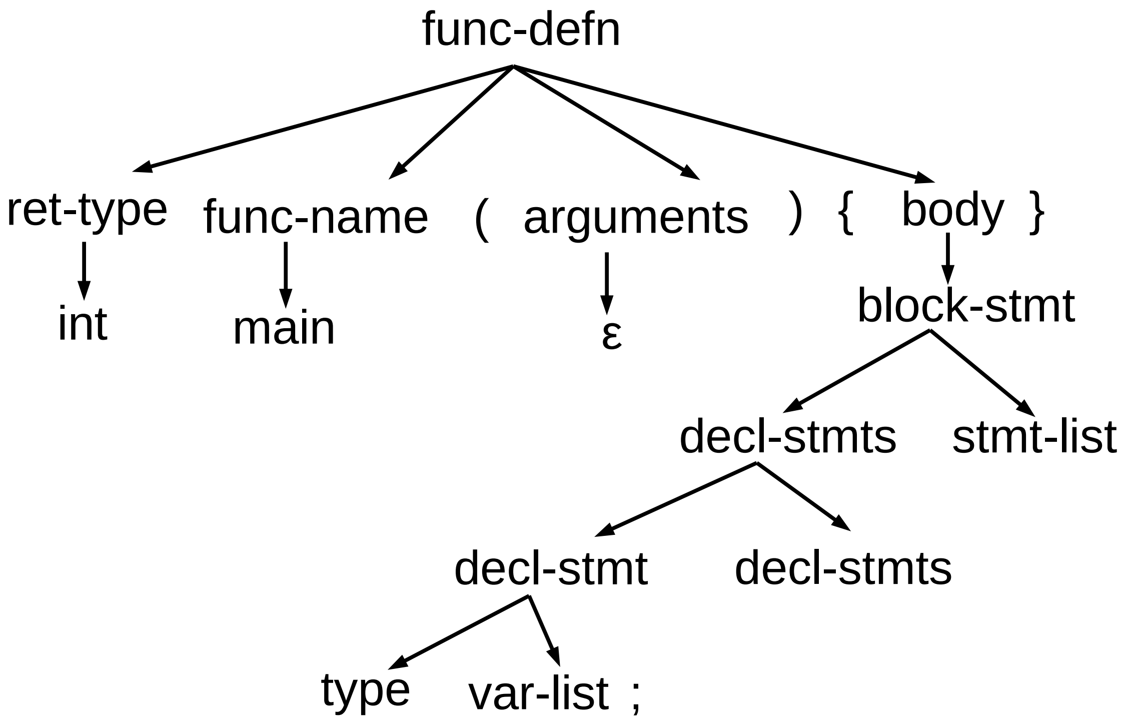
# Structure of a Program

The structure of a function body is further elaborated into a block-statement which in turn is specified by one or more declaration statements followed by a list of statements.



# Structure of a Program

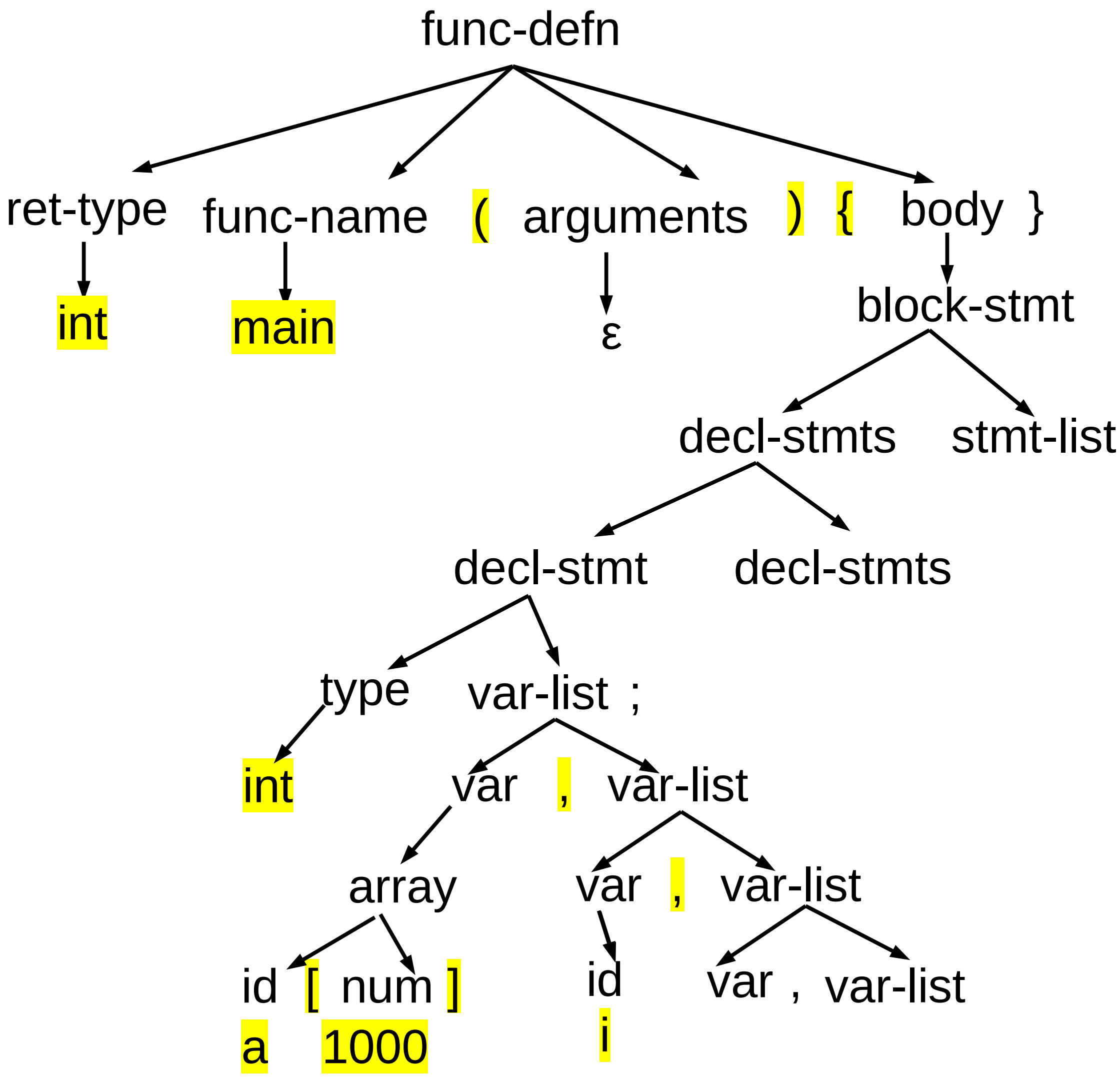
The structure of a single declaration statement is used to further extend the tree under decl-stmts – which comprises of a type specifier followed by a list of variable declarations.



The tree gets extended when the input examined is :

int main ( ) { int a [ 1000 ] , i ,     is shown in the next page.

# Structure of a Program



# Tree for Validating Program Structure

- The process of discovering language constructs in a stream of input tokens is called **Syntax Analysis**. Syntax analysis uses lexical analysis to convert lexemes to tokens and works on tokens only.
- A tree is a natural data structure to represent the structure of a program and is constructed incrementally as input tokens are examined one by one.
- Such a tree, often called a Parse Tree or an Abstract Syntax Tree (AST), is used for representation of discovered programming langauge structures.
- The tokens lie at the leaf nodes of the tree while the internal nodes are the names of language features rooted at this node.
- The structural linguistic features have to be clearly specified to the compiler designer.
- A CFG meets the requirement for specifying the syntactic aspects of language constructs.

# CFG for Syntax Specification

- As an illustration, we write a CFG underlying the structure validation process
- G = (N, T, P, S) where N = { func-defn, ret-type, func-name, arguments, body, block-stmt, decl-stmts, stmt-list, decl-stmt, type, var-list, var, array}
- T has the following symbols and tokens : ( ) { } int , ; id [ ] num
- The nonterminal func-defn is the start symbol S
- The production rules follow :

func-defn →ret-type func-name ( arguments ) { body }

- ret-type → int | void | ε

arguments → arg-list | ε

block-stmt → decl-stmts stmt-list

decl-stmt →type var-list ;

var → array | id

func-name → id

body → block-stmt | ε

decl-stmts → decl-stmt | ε

var-list →var , varlist

array → id [num ]

# Understanding the Meaning

# Semantic Analysis

# Semantic Analysis

# Semantic Analysis

# Intermediate Code Generation

# Intermediate Code Generation

# Intermediate Code Generation

# Intermediate Code Generation

# Intermediate Code Generation

# Run Time Environment

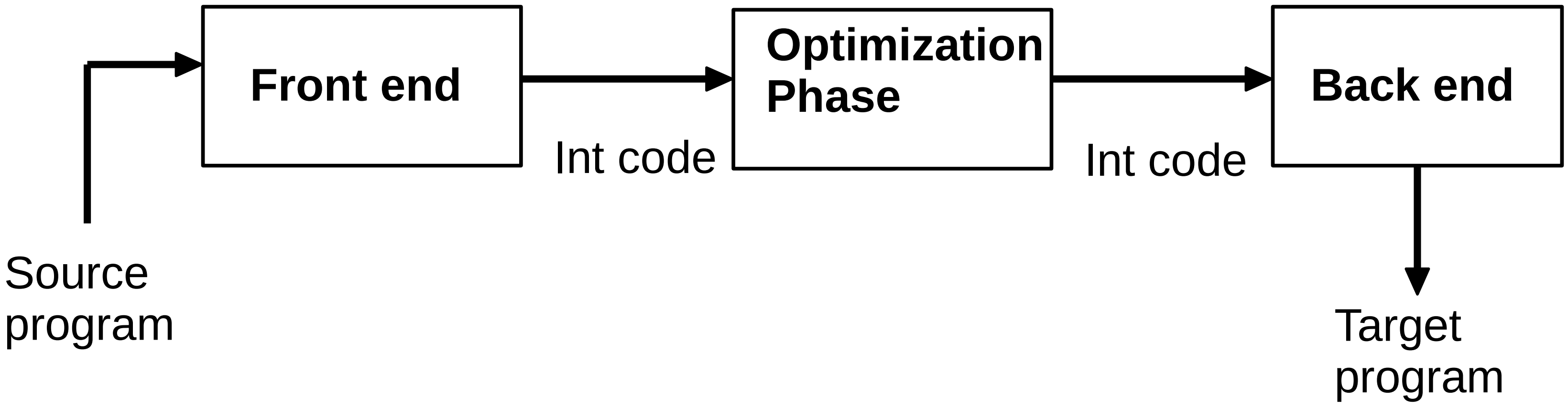
# Run Time Environment

# Run Time Environment

# Run Time Environment

# Compiler Optimizations

- Intermediate code optimization is an optional phase which is enabled by an user if so required.



- Purpose is to improve the quality of intermediate code generated by the front end of the compiler.
- Illustrate the capability of this phase through examples.

# Compiler Optimizations

- Consider the following C program.

```
 int main()
{   int a[1000], i, j;
    int sum = 10000;
    for (i = 0; i < 1000; i++) a[i] = i;
    for (i = 0; i < 1000; i++)
                    for (j = 0; j < i*i; j++)   a[i] = a[i] + a[j];
    printf(" sum : %d \n", sum);
}
```

# Compiler Optimizations

- Compile this program with

```
    $  gcc -S -fverbose-asm firstprog.c
 int main()
{   int a[1000], i, j;
   int sum = 10000;
   for (i = 0; i < 1000; i++) a[i] = i;
   for (i = 0; i < 1000; i++)
        for (j = 0; j < i*i; j++)   a[i] = a[i] + a[j];
   printf(" sum : %d \n", sum);
}
```

# Compiler Optimizations

- Compile this program with

```
$  gcc -S -fverbose-asm firstprog.c -o unopt.s
```

- Now compile using the O2  switch

```
$  gcc -S -fverbose-asm firstprog.c -O2 -o opt.s
```

- Compare the two assembly code side by side and observe.
- You may be surprised by the second assembly code.

# Manual Optimization

Consider the following C++ program :

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j;
  int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)  { d = a * b; x[i] = x[i-1] + d;  c = b*100;  }
    else  {d = a * a; x[(i+d)%10] = x[i]+x[i-1];} };
  if (a < 2) for (j = 1; j < 11; j++)  x[(j+5)%10] = x[j] + 5;
  else for (j = 0; j < 10; j++) cout << x[j] << " ";
  cout << endl;
  return 0;
}
```

# Constant Evaluation

**Task 1 : Evaluate compile time constants. What are the savings ?**

```
  int main()
  { int a = 2, b = 3, c = 40, d, i, j; int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
      { d = 6; // d = a * b;
        x[i] = x[i-1] + d;
        c = 300; // c = b*100;
  }
  else    {    d = 4; // d = a * a;
                x[(i+d)%10] = x[i]+x[i-1];} };
 if (False )   for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5; // 2 < 2
 else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
 return 0;
}
```

# Constant Propagation

**Task 2 : Evaluate compile time constants. What are the savings ?**

```
   int main()
   { int a = 2, b = 3, c = 40, d, i, j; int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
   for ( i = 1; i < 11; i+=2 )
   { if ( i%2)
       { d = 6; // d = a * b;
         x[i] = x[i-1] + 6;          // substitute 6 for d
         c = 300; // c = b*100;
  }
  else    {    d = 4; // d = a * a;
                 X [ (i+4) % 10] = x[i] + x[i-1];} };
 if (False )   for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5;     // 2 < 2
 else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
 return 0;
}
```

# Loop Invariant Code Motion

**Task 3 : Move loop invariant code out of loops. What are the savings ?**

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j; int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
 d = 6; // d = a * b;
 c = 300; // c = b*100;
 d = 4; // d = a * a;
 for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
         {  x[i] = x[i-1] + 6;  }
   else     { x[(i+4) % 10] = x[i] + x[i-1];} };
 if (False )    for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5;     // 2 < 2
  else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
return 0;
}
```

# Live Variables & Dead Code Removal

**Task 4 : Find variables that are live and remove dead code. What are the savings?**

```
   int main()
   { int a = 2, b = 3, c = 40, d, i, j;
     int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
     for ( i = 1; i < 11; i+=2 )
     { if ( i%2)
            {  x[i] = x[i-1] + 6;  }
      else     { x[(i+4) % 10] = x[i] + x[i-1];} };
     if (False )   for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5;     // 2 < 2
      else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

# Remove Unreachable Code

**Task 5 : Find variables that are unreachable, evaluate conditional expressions in control flow constructs and remove unreachable code. What are the savings?**

```
  int main()
  { int a = 2, b = 3, c = 40, d, i, j;
    int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    for ( i = 1; i < 11; i+=2 )
    { if ( i%2)
              {  x[i] = x[i-1] + 6;  }
     else     { x[(i+4) % 10] = x[i] + x[i-1];} };
      for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
 return 0;
}
```

# Remove Unused Definitions

**Task 6 : Find the definitions that are not used in the program. What are the savings?**

```
    int main()
    { // int a = 2, b = 3, c = 40, d,
      int  i, j;
      int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
      for ( i = 1; i < 11; i+=2 )
      { if ( i%2)
              {  x[i] = x[i-1] + 6;  }
       else     { x[(i+4) % 10] = x[i] + x[i-1];} };
        for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

# Final Optimized Program

**Task 6 : Find the definitions that are not used in the program. What are the savings?**

```
   int main()
   { int  i, j;
     int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
     for ( i = 1; i < 11; i+=2 )
     { if ( i%2)
            {  x[i] = x[i-1] + 6;  }
      else     { x[(i+4) % 10] = x[i] + x[i-1];} };
       for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
 return 0;
}
```

# Code Generation

# Code Generation

# Compiler Concepts Learned Through Example

# THEME 2

## STRUCTURE OF A COMPILER

# Design of a Compiler

- Regular expressions: **Finite Automata : Lexical Analyser**
- Context free grammars : **Pushdown automata : Parser or Syntax Analyser**
- Nested Symbol Tables : Scope and lifetime analysis : **Data structure and algorithms**
- Memory layout and activation records : Runtime environments : **Architecture and OS**

# Design of a Compiler

- Semantic Analysis [Context Sensitive Issues such as Type checking; intermediate code generation] : Syntax directed translation scheme : Tree, graph; **Data structure and algorithms.**
- Parameter passing mechanisms : Semantic Analyser : **Programming Languages.**
- Optimization of intermediate code : Optional pass : graph, lattice theory, solution of equations; **Discrete Structures**.
- Code generation : Compiler back-end design: **Algorithms and complexity; Architecture, Assembly language, OS.**

**Theory and Practice are intimately connected**

# THEME 3

## Experimentation With Gnu C Compiler

## Open Source Production Quality Compiler

# Simple C program

```
#include <stdio.h>
int main()
{    int a[1000], i, j;
     int sum = 10000;
     for (i = 0; i < 1000; i++) a[i] = i;
     for (i = 0; i < 1000; i++)
          for (j = 0; j < i*i; j++)
                a[i] = a[i] + a[j];
     printf(" sum : %d \n", sum);
}
```

# Compiler Features

- Most compilers provide a host of features for programming ease and enhanced productivity.
- Illustrated with Gnu C compiler (gcc) & C++ compiler (g++)
- List a few compiler options that are generally useful. Read

  the online manual, man gcc or info gcc for more details.
- -fverbose-asm -S -c -o

  -v -O2 -E

  -fump-tree-all and many many others
- Experiments to illustrate the working of a compiler through a simple example.

# Checking Correctness

The output produced after the 6 optimizations are performed in the order discussed are as follows.

| | |
|---|---|
| 10 16 30 36 50 56 70 76 90 96 | original source |
| 10 16 30 36 50 56 70 76 90 96 | after optimization 1 |
| 10 16 30 36 50 56 70 76 90 96 | after optimizations 1,2 |
| 10 16 30 36 50 56 70 76 90 96 | after optimization 1 to 3 |
| 10 16 30 36 50 56 70 76 90 96 | after optimization 1 to 4 |
| 10 16 30 36 50 56 70 76 90 96 | after optimization 1 to 5 |
| 10 16 30 36 50 56 70 76 90 96 | after optimization 1 to 6 |
| 10 16 30 36 50 56 70 76 90 96 | after optimization by gcc |

# Compiling for Dumps

Other necessary command line switches

- -O2 -fdump-tree-all
- -O3 enables -ftree-vectorize. Other flags must be enabled explicitl

Other useful options

- Suffixing -all to all dump switches
- -S to stop the compilation with assembly generation
- -fverbose-asm to see more detailed assembly dump