# Compiler Design

**Preet Kanwal**

Department of Computer Science & Engineering

Teaching Assistant : Gautham P Atreyas

# Compiler Design

## Unit 1

## The Phases of a Compiler

**Preet Kanwal**

Department of Computer Science & Engineering

# Compiler Design

## Lecture Overview

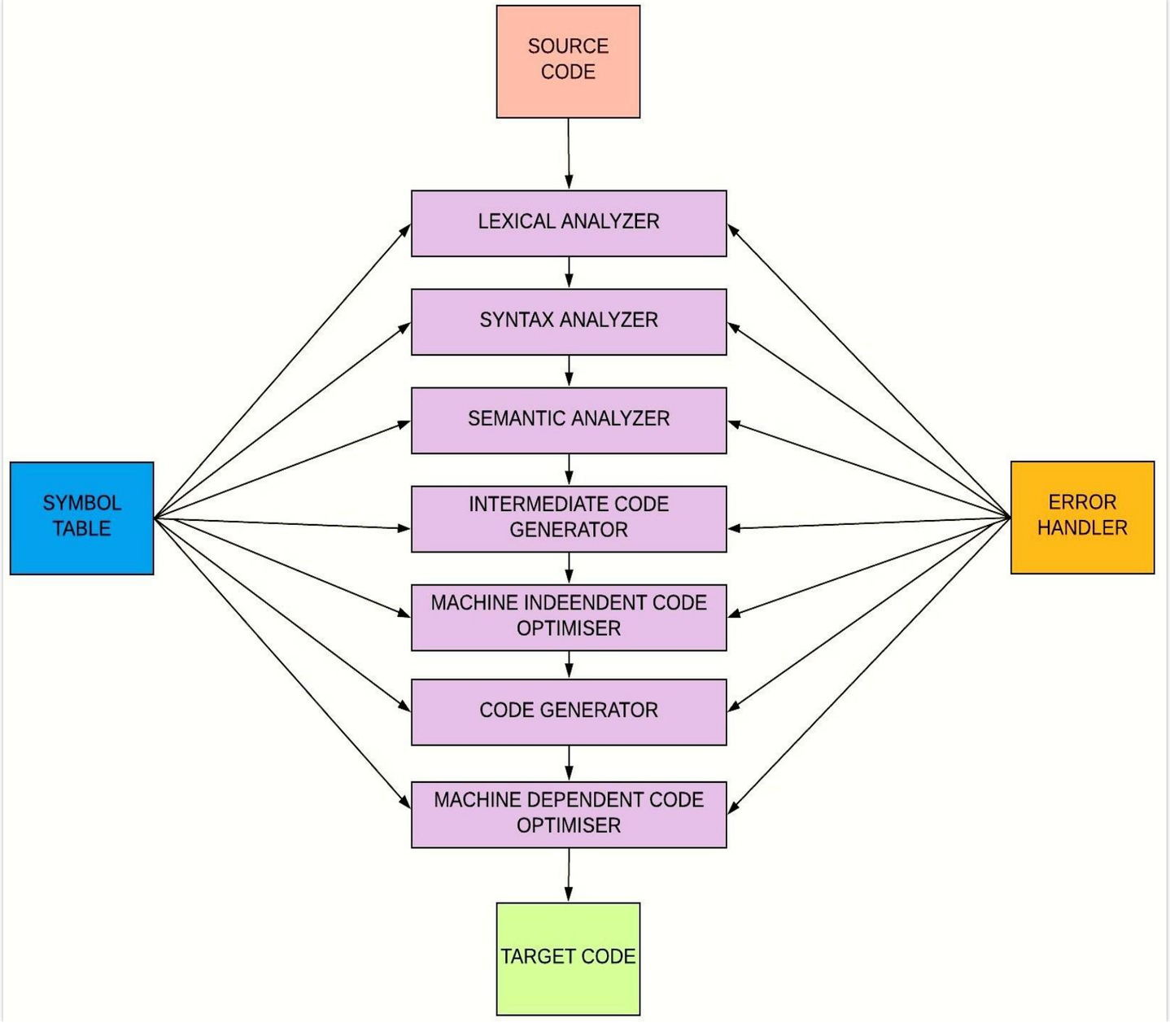
PES UNIVERSITY
CELEBRATING 50 YEARS

### In this lecture, you will learn about:

- Symbol Table
- Construction of Symbol Table
- Symbol Table and Scope

# Compiler Design

## Phases of a Compiler




- The compiler does its work in seven different phases.
- Each of them have access to the entities known as “Symbol Table” and “Error Handler”
- They will be explained in upcoming slides.
- Each phase transforms one representation of the source program into another.

# Compiler Design

## Symbol Table

PES UNIVERSITY
CELEBRATING 50 YEARS

### What is a symbol table?

**Data structure containing a record for each variable name with fields for the attributes of the name. The information is collected incrementally by analysis phase of a compiler and used by the synthesis phase to generate the target code.**

### How does it look like?

**Attributes provides the information about:**

- **Storage allocation**
- **Type**
- **Scope (where in the program its value may be used)**
- **Procedure names**
  - **Number and types of its arguments**
  - **Method of passing arguments (call by value or reference)**

# Compiler Design

## Symbol Table

PES UNIVERSITY
CELEBRATING 50 YEARS

**Why symbol table?**

- Used during all phases of compilation
- Maintains information about many source language constructs
- Used directly in the code generation phases
- Efficient storage and access important in practice
- Typically need to support multiple declarations of the same identifier

**Who creates it?**

- In some cases, the lexical analyzer can create an entry when it identifies a lexeme
- More often, the lexical analyzer can only return a token with a pointer to the lexeme, to the parser. Only the parser can decide whether to use a previously created symbol-table entry or create a new one
- Note: lexemes and tokens are explained in later topics

# Symbol Table

## When and where is it used?

- Lexical Analysis Time
  - Lexical Analyzer scans program
  - Finds Symbols
  - Adds Symbols to symbol table
- Syntactic Analysis Time
  - Information about each symbol is filled in/updated
- Semantic Analysis Time
  - Used for type checking

# Symbol Table

## More about symbol tables

- Each piece of info associated with a name is called an attribute.
- Attributes are language dependent.
- They include:
  - Actual characters of the name
  - Type
  - Storage allocation info (number of bytes).
  - Line number where declared
  - Lines where referenced
  - Scope

# Symbol Table

## Different Classes of Symbols have different attributes

- Variable, Type, Constant, Parameter, Record:
  - Type, storage, name, scope.
- Procedure or Function:
  - Number of parameters, parameters themselves, result type.
- Array:
  - # of Dimensions, array bounds.
- File:
  - record size, record type

## Other attributes:

- A scope of a variable can be represented either by
  - A number (Scope is just one of attributes)
  - Constructing different symbol tables for different scopes

- Object Oriented Languages have symbol classes like
  - Method names, class names, object names.
  - Scoping is VERY important (inheritance).

# Compiler Design

## Constructing the Symbol Table

There are three main operations to be carried out on the symbol table:

- determining whether a string has already been stored
- inserting an entry for a string
- deleting a string when it goes out of scope

This requires three functions:

- **lookup(s)**: returns the index of the entry for string s, or 0 if there is no entry
- **insert(s,t)**: add a new entry for string s (of token t), and return its index
- **delete(s)**: deletes s from the table (or, typically, hides it)

# Compiler Design

## Constructing the Symbol Table

The symbol table can be constructed using 2 mechanisms:

Linear lists and Hash tables

- Each scheme is evaluated on the basis of time required to add ` n ` entries and make ` e ` inquiries.
- A linear list is the simplest to implement, but its performance is poor when ` n ` and ` e ` are large.
- Hashing schemes provide better performance for greater programming effort and space overhead.

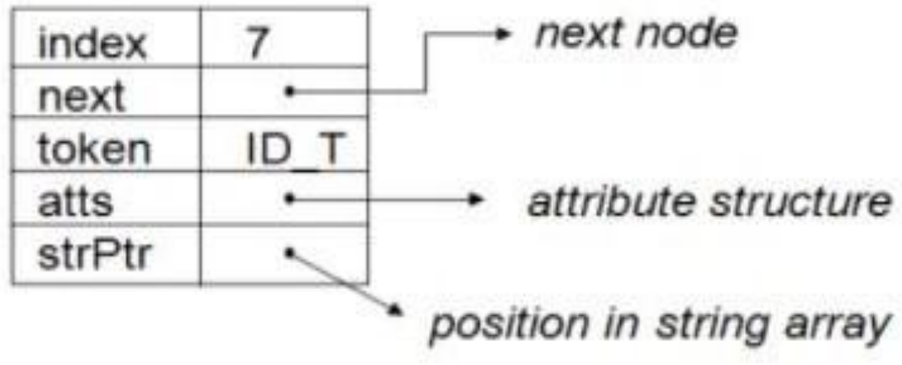
# Compiler Design

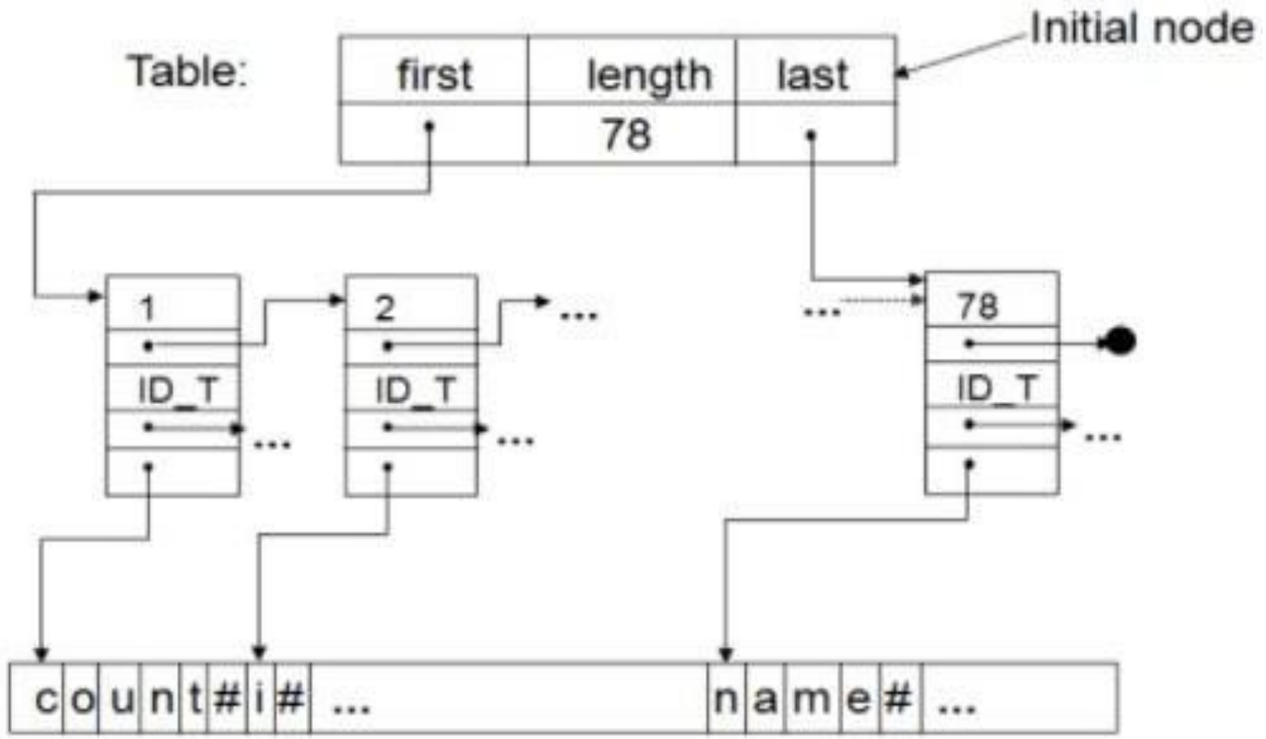
## Constructing the Symbol Table - Example

1. float Fun2(int I, float j,){
2. int k,e;
3. float z;
4. ;
5. e=0;
6. k = I *j+k;
7. z = fun1(k,e,);
8. *z; }

| Name | Token | Dtype | Value | Size | Scope | Other Attribute | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | Declared | Referred | Other |
| Fun2 | TK_ID | procname | | | | 1 | | |
| I | TK_ID | Int | | 4 | 1 | 1 | 6 | parameter |
| j | TK_ID | Int | | 4 | 1 | 1 | 6 | parameter |
| k | TK_ID | Int | | 4 | 0 | 2 | 6,7 | argument |
| e | TK_ID | Int | 0 | 4 | 0 | 2 | 7 | argument |
| z | TK_ID | Float | | 4 | 0 | 3 | 7,8 | return |
| fun1 | TK_ID | procname | | | | 7 | | proccall |

# Constructing the Symbol Table - Linked List Implementation



# Compiler Design

## Symbol Table and Scope

**The scope of a declaration is the portion of a program to which the declaration applies. The scope of identifier ` x ` refers to the scope of that particular declaration of ` x `.**

- **Symbol tables typically need to support multiple declarations of the same identifier within a program.**
- **Subclasses can redeclare a method name to override a method in the superclass.**
- **If blocks can be nested, several declarations of the same identifier can appear within a single block.**

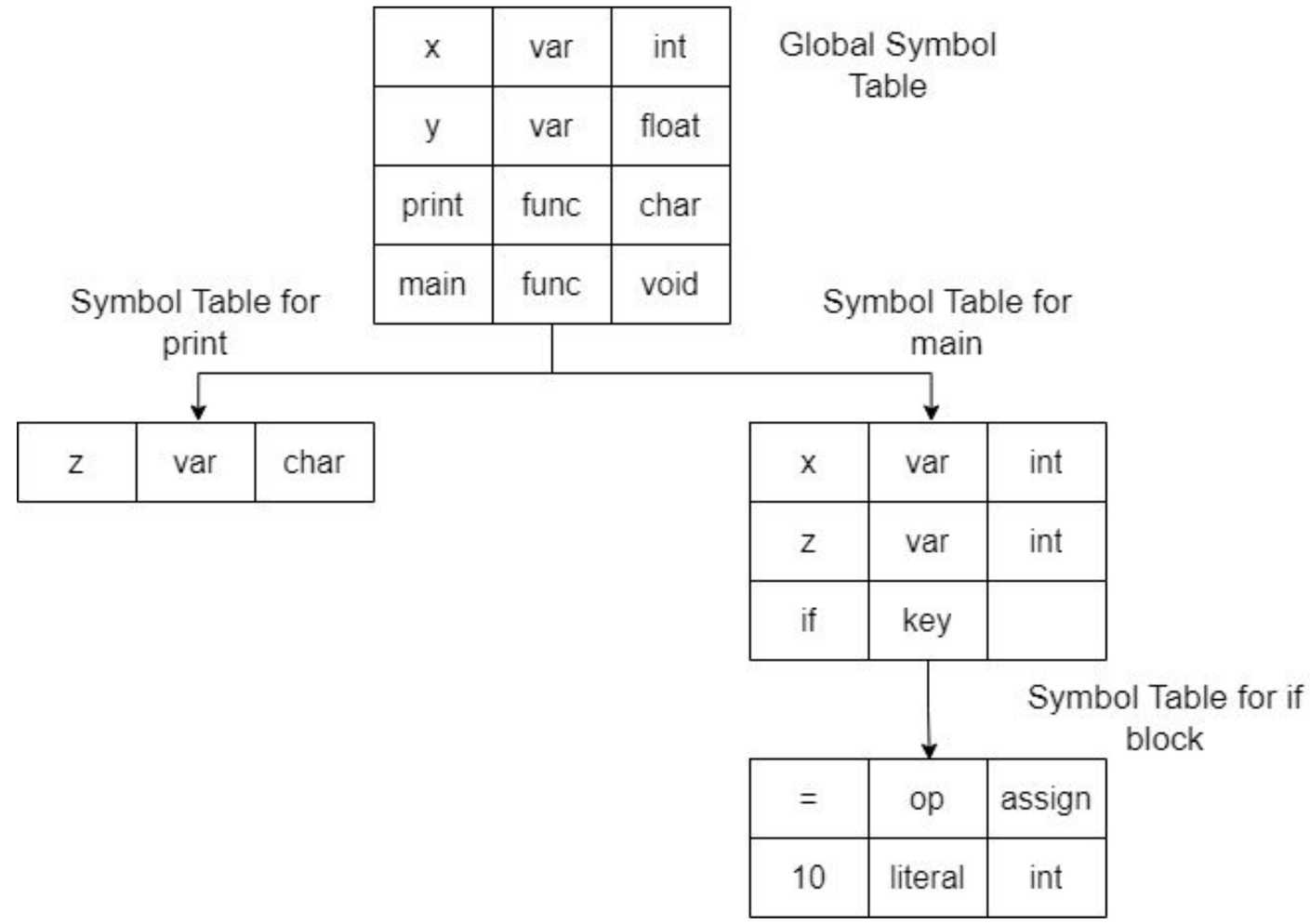
**We shall implement scopes by setting up a separate symbol table for each scope**

.

# Compiler Design

## Symbol Table and Scope

```
int x;
float y;

char print() {
   char z;
}

void main() {
   int x, z;
   if ( x ) {
      y = 10;
   }
}
```



# THANK YOU

**Preet Kanwal**

Department of Computer Science & Engineering

**preetkanwal@pes.edu**

# Compiler Design

**Preet Kanwal**

Department of Computer Science & Engineering

Teaching Assistant : Gautham P Atreyas

# Compiler Design

## Unit 1

## The Phases of a Compiler

**Preet Kanwal**

Department of Computer Science & Engineering

# Compiler Design

## Lecture Overview

PES UNIVERSITY
CELEBRATING 50 YEARS

### In this lecture, you will learn about:

- Challenges in scanning
- C Language Grammar

# Compiler Design

## Challenges in Scanning - Syntactic Sugar

**Syntactic Sugar : Syntax is designed to make things easier to read or express. Example:**

- In C : a[i] is a syntactic sugar for *(a+i)
- In C : a+=b is equivalent to a = a + b
- In C# : var x = expr (compiler deduces type of x from expr)

**Compilers expand sugared constructs into fundamental constructs (Desugaring).**

# Compiler Design

## Challenges in Scanning - Whitespaces

PES UNIVERSITY
CELEBRATING 50 YEARS

**Example 1: In FORTRAN, whitespace is irrelevant. For eg:**

```
DO 5 I = 1,25
DO5I   = 1.25
```

**This makes it difficult to decide where to partition the input.**

# Compiler Design

## Challenges in Scanning - Overloaded operator

### Example 2 : Different uses of the same characters ` > ` and ` < ` in C++

Template syntax: a<b>

Stream syntax: cin>>var;

Binary right shift syntax: a>>4

Nested Template : A<B<C>>D;

## Challenges in Scanning - Keywords

### Example 3: When keywords are used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

This makes it difficult to name lexemes.

C and C++ Lexers require lexical feedback to differentiate between typedef names and identifiers.

```
int foo;
typedef int foo;
foo x;
```

# Compiler Design

## Challenges in Scanning - Scope

PES UNIVERSITY
CELEBRATING 50 YEARS

### Example 4 : Scanning in Python: scope is handled using whitespace

### This requires whitespace tokens - Special tokens inserted to indicate changes in levels of indentation.

- NEWLINE marks the end of a line.
- INDENT indicates an increase in indentation.
- DEDENT indicates a decrease in indentation.

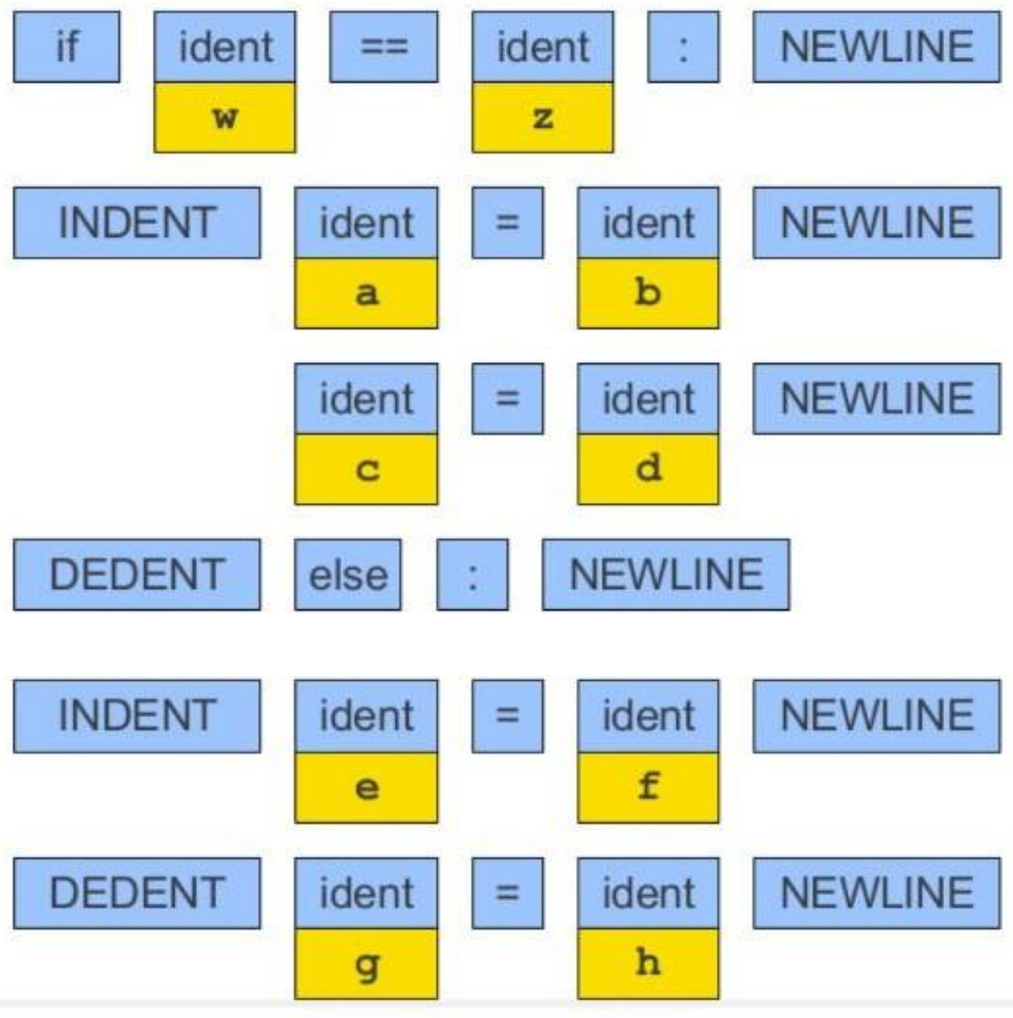
Note that INDENT and DEDENT encode only the change in indentation, and not the total amount of indentation.

Reference:
https://docs.python.org/3/reference/lexical_analysis.html

# Challenges in Scanning - Scope

## Example 4: Scanning in python (cont.)

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```



# Compiler Design

## C Language Grammar

**To define the C Language Grammar, let us first define the following:**

| | | |
|---|---|---|
| P | : | Program Beginning |
| S | : | Statement |
| Declr | : | Declaration |
| Assign | : | Assignment |
| Cond | : | Condition |
| UnaryExpr | : | Unary Expression |
| Type | : | Data type |
| ListVar | : | List of variables |
| X | : | can take any identifier or assignment |
| RelOp | : | Relational Operator |

# Compiler Design

## C Language Grammar

P → S

S → Declr; S | Assign; S | if (Cond) {S} S | while (Cond) {S} S |
if (Cond) {S} else {S} S | for (Assign; Cond; UnaryExpr) {S} S |
return E; S | λ

Declr → Type ListVar

Type → int | float

ListVar → X | ListVar, X

X → id | Assign

Assign → id = E

Cond → E RelOp E

RelOp → < | > | <= | >= | == | !=

UnaryExpr → E++ | ++E | E-- | --E

Recall that
E -> E + T | E – T | T
T -> T * F | T / F | F
F -> G ^ F | G
G -> ( E ) | id | num

# THANK YOU

**Preet Kanwal**

Department of Computer Science & Engineering

preetkanwal@pes.edu

# Compiler Design

**Preet Kanwal**

Department of Computer Science & Engineering

Teaching Assistant : Gautham P Atreyas

# Compiler Design

## Unit 1

## The Phases of a Compiler

**Preet Kanwal**

Department of Computer Science & Engineering

# Compiler Design

## Lecture Overview

PES UNIVERSITY
CELEBRATING 50 YEARS

### In this lecture, you will learn about:

- C Language Grammar
- Running an input through the different phases of a compiler
- Simple IR Optimizations
- Syntax Tree vs Parse Tree

# Compiler Design

## C Language Grammar

P → S

S → Declr; S | Assign; S | if (Cond) {S} S | while (Cond) {S} S |
if (Cond) {S} else {S} S | for (Assign; Cond; UnaryExpr) {S} S |
return E; S | λ

Declr → Type ListVar

Type → int | float

ListVar → X | ListVar, X

X → id | Assign

Assign → id = E

Cond → E RelOp E

RelOp → < | > | <= | >= | == | !=

UnaryExpr → E++ | ++E | E-- | --E

Recall that
E -> E + T | E – T | T
T -> T * F | T / F | F
F -> G ^ F | G
G -> ( E ) | id | num

# Running an input through different phases of a compiler

**Example 1 : Consider the following piece of code:**

```
while (number > 0)
{
   factorial = factorial * number;
   --number;
}
```

**We will now run this code through the different phases of a compiler and see what the output will be at each stage.**

# Compiler Design

## Running an input through different phases of a compiler

Before we start, there are a few terminologies that should be known. These will be explained in the later slides and units, but are important for this exercise.

- **Token**: a structure, that represents a sequence of characters which cannot be broken down further
- **Lexeme**: the actual sequence of characters in the source program
- **Pattern**: rule that defines the structure for a lexeme to be an instance of a token
- **Syntax**: describes the proper form of a programming language
- **Semantics**: defines what the program means
- **Syntax Tree**: tree-like representation of the syntax of a source program

# Compiler Design

## Running an input through different phases of a compiler

Before we start, there are a few terminologies that should be known. These will be explained in the later slides and units, but are important for this exercise.

- **3 Address Code**: an intermediate representation, where the instructions are of the form ` x = y op z ` . There can be at most one operator per instruction Complex instructions are broken up using temporary variables ` t0, t1, … tn `
- **Basic Block**: sequence of instructions that are always executed sequentially, i.e. no jumps or interruptions
- **DAG Optimization**: optimization of basic blocks using Directed Acyclic Graphs. It is used to determine common subexpressions

# Compiler Design

## Running an input through different phases of a compiler

PES UNIVERSITY
CELEBRATING 50 YEARS

### Phase 1 : Lexical Analysis or Linear Analysis or Scanner

| Pattern | Lexemes matched |
|---|---|
| Keyword | while |
| Identifier | number, factorial |
| RelOp (<, <=, >, >=, ==, !=) | > |
| LogOp (&&, \|\|, ~) | |
| Assign (=) | = |
| ArithOp (*, +, -, /) | * |
| Punctuation ( (, ), {, }, [, ], ;) | ( ) { ; ; } |
| Number | 0 |
| Literal – anything in double quotes | |
| Inc_op | |
| Dec_op | -- |

```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

# Compiler Design

## Running an input through different phases of a compiler

**Output: # of tokens : 17**

```
< keyword, while >
< punctuation, ( > or < ( >
< ID, 3 >
< RelOp, > >      or < > >
< Number, 0 >
< punctuation, ) > or < ) >
< punctuation, { > or < { >
< ID, 4 >
< Assign, = >     or < = >
< ID, 4 >
< ArithOp, * >    or < * >
< ID, 3 >
< punctuation, ; > or < ; >
< Dec_op, -- >    or < -- >
< ID, 3 >
< punctuation, ; > or < ; >
< punctuation, } > or < } >
```

**Symbol Table**

| | name | type | … |
|---|---|---|---|
| 1 | … | | |
| 2 | …. | | |
| 3 | number | | |
| 4 | factorial | | |

< ID, 3 > corresponds to the third identifier in the symbol table

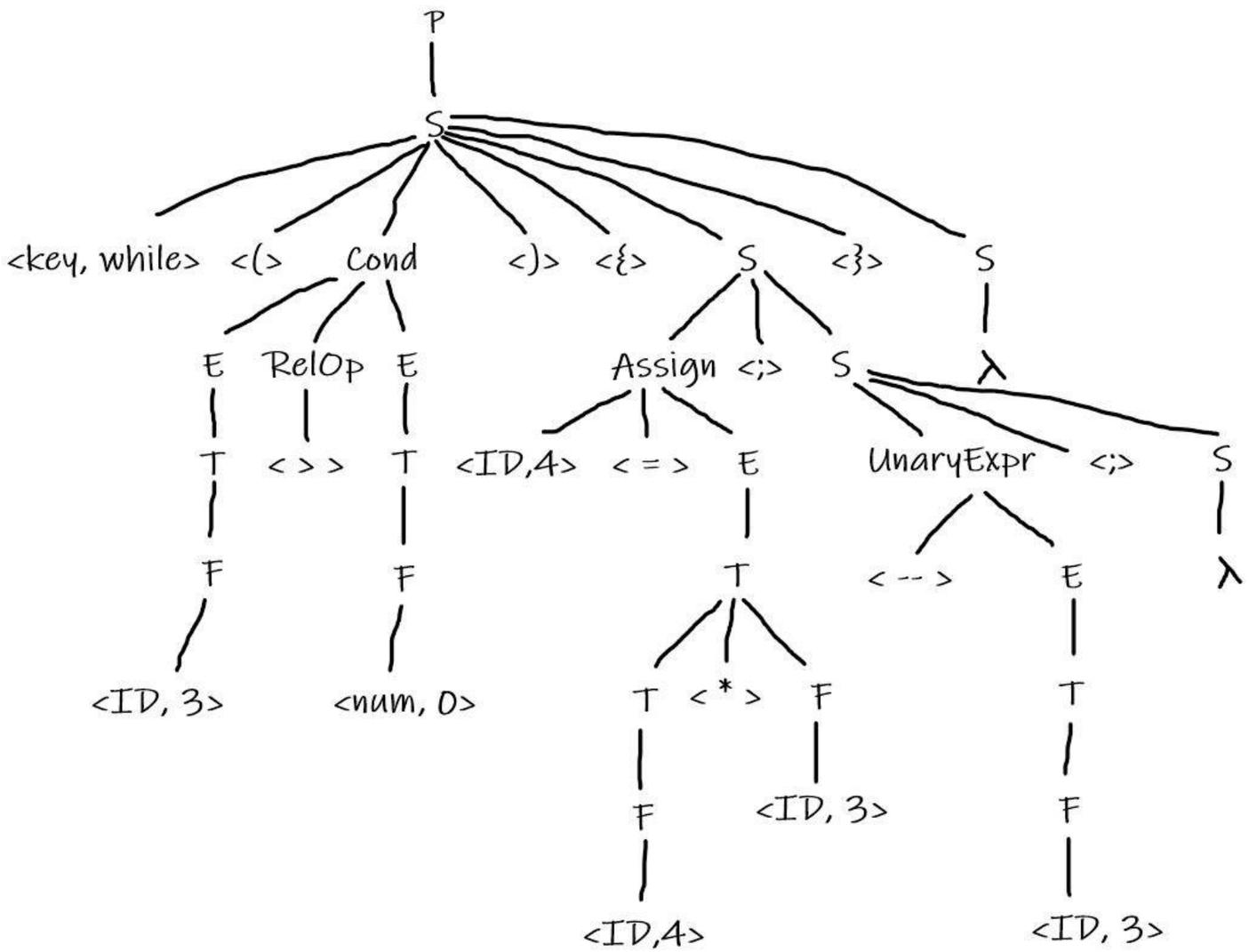
```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

# Compiler Design

## Running an input through different phases of a compiler

## Phase 2 : Syntax Analysis or Parser

### Output: Parse Tree or Syntax Tree




Use the grammar we have defined earlier

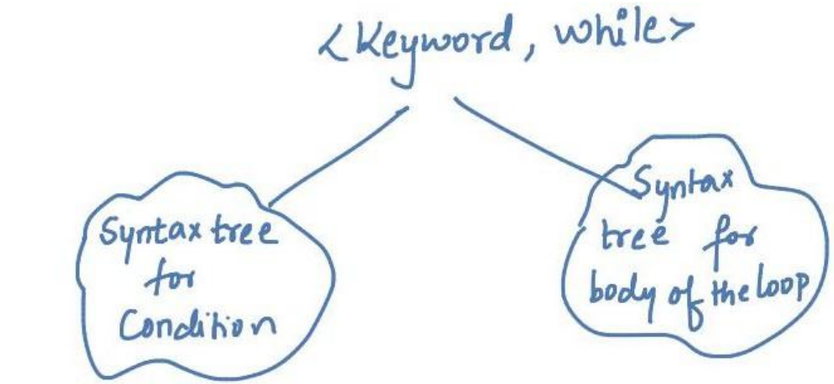
```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

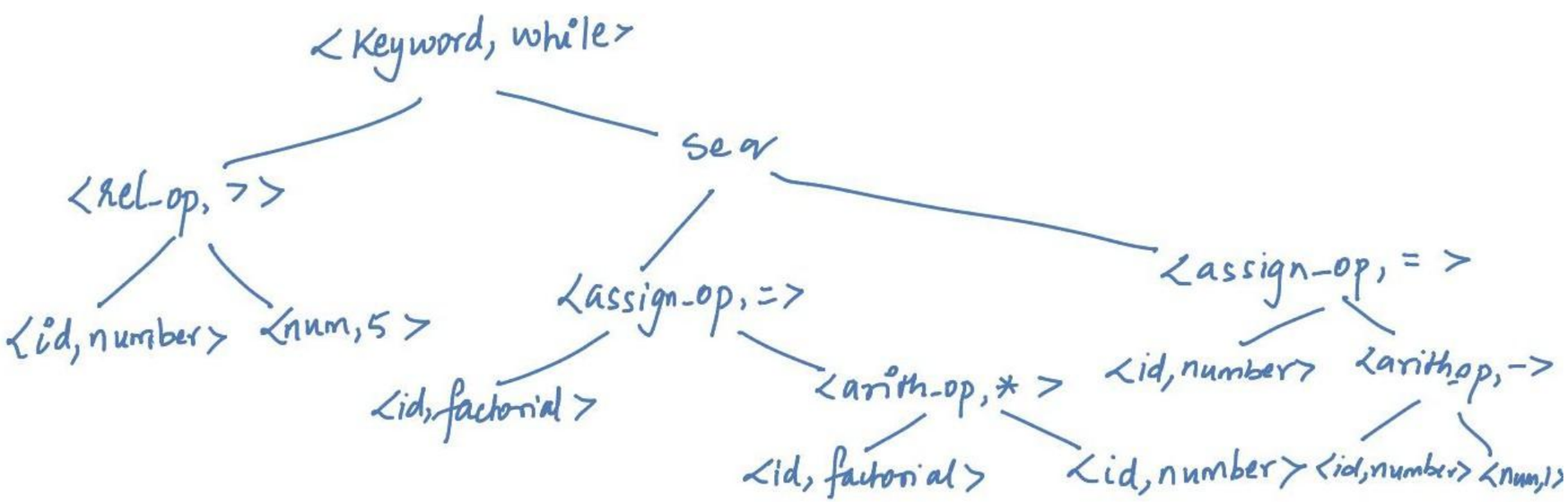
# Compiler Design

## Running an input through different phases of a compiler

## Phase 3 : Semantic Analyzer

## Output: Semantically checked Syntax Tree




```
while (number > 0)
{
factorial = factorial * number;
--number;
}
```

# Compiler Design

## Running an input through different phases of a compiler

PES UNIVERSITY
CELEBRATING 50 YEARS

### Phase 4 : Intermediate Code Generator

### Output: Three Address Code (3AC or TAC)

```
L0:  if number > 0 goto L1
     goto L2
L1:  t1 = factorial * number
     factorial = t1
     t2 = number - 1
     number = t2
     goto L0
L2:  ...
```

```
while (number > 0)
{
factorial = factorial *
number;
--number;
}
```

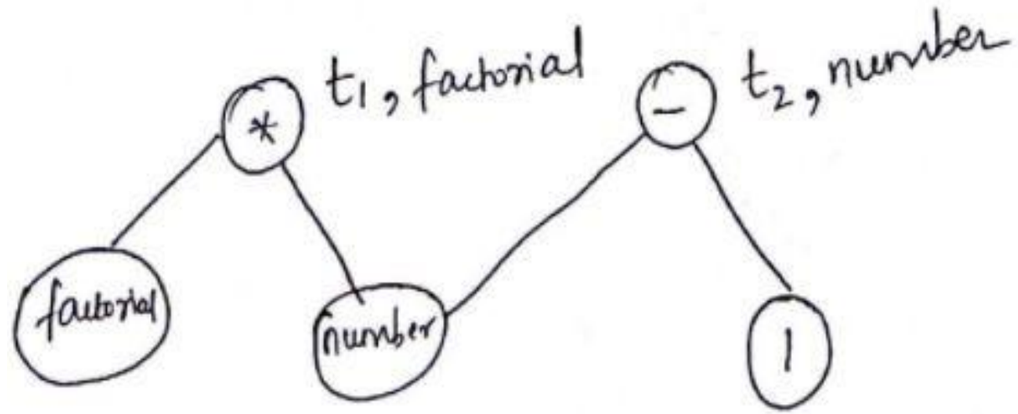
# Compiler Design

## Running an input through different phases of a compiler

## Phase 5 : Machine Independent Code Optimization

## Output: DAG Optimization on the basic block L1

```
L0 : if number > 0 goto L1
     goto L2
L1 : factorial = factorial * number
     number = number – 1
     goto L2
L2 : ...
```




```
while (number > 0)
{
factorial = factorial *
number;
--number;
}
```

# Running an input through different phases of a compiler

## Phase 6 : Code Generator

**Output: Assembly Code**

```
      LD    R1,#0
L1:   LD    R4,number
      SUB   R2,R1,R4
      BLTZ  R2,L2
      LD    R3,factorial
      MUL   R3,R3,R4
      ST    factorial,R3
      SUB   R4,R4, #1
      ST    number,R4
      BR    L1
L2:
```

```
while (number > 0)
{
factorial = factorial *
number;
--number;
}
```

## Phase 7 : Machine Independent Optimized Code

**Output: Optimized Target Code**

# Compiler Design

## Simple IR Optimizations

**Constant folding** is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.

**Constant propagation** is the process of substituting the values of known constants in expressions at compile time.

**Common Subexpression Elimination (CSE)** is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.

# Compiler Design

## Syntax Tree vs Parse Tree

Parse tree is a concrete representation of the input. It retains all the information from the input.

Syntax Tree is an abstract representation of the input. It only contains the information required to generate IR or machine code.

| Parse Tree (Concrete Syntax Tree) | (Abstract) Syntax Tree |
|---|---|
| It is huge and focuses on all lexeme | It focuses only on the syntax |
| Consists of both Non-terminals (P, S, E, Cond, RelOp, etc.) and Terminals (leaf nodes like tokens) | All nodes are tokens;<br>It also has dummy nodes (<body>) to maintain proper syntax |
| Complicated | No non-terminals; Easy |
| Has punctuation ( ; , {} ) | No punctuation symbols |

## Example 2

**Example 2 : Consider the following piece of code:**

```
int rhs,
lhs = 0;
rhs = 2;

if (lhs <= rhs)
{
   lhs = lhs – 2 * rhs;
}
```

**Run this code through the different phases of a compiler and produce the output at each stage.**

# THANK YOU

**Preet Kanwal**

Department of Computer Science & Engineering

**preetkanwal@pes.edu**

# Compiler Design

**Preet Kanwal**

Department of Computer Science & Engineering

Teaching Assistant : Gautham P Atreyas

# Compiler Design

## Unit 1

## The Phases of a Compiler

**Preet Kanwal**

Department of Computer Science & Engineering

# Compiler Design

## Lecture Overview

PES UNIVERSITY
CELEBRATING 50 YEARS

**In this lecture, you will practise:**

- **Running an input through the different phases of a compiler**

# Example 2

**Example 2 : Consider the following piece of code:**

```
int rhs,
lhs = 0;
rhs = 2;

if (lhs <= rhs)
{
   lhs = lhs – 2 * rhs;
}
```

**We will now run this code through the different phases of a compiler and see what the output will be at each stage.**

# Compiler Design

## Example 2 (continued)

### Phase 1 : Lexical Analysis or Linear Analysis or Scanner

| Pattern | Lexemes matched |
|---|---|
| Keyword | int if |
| Identifier | rhs lhs rhs lhs rhs lhs lhs rhs |
| RelOp (<, <=, >, >=, ==, !=) | <= |
| LogOp (&&, \|\|, ~) | |
| Assign (=) | = = = |
| ArithOp (*, +, -, /) | - * |
| Punctuation ( (, ), {, }, [, ], ;) | ; ; ( ) { ; } |
| Number | 0 2 2 |
| Literal – anything in double quotes | |
| Inc_op | |
| Dec_op | |

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

# Compiler Design

## Example 2 (continued)

**Output: # of tokens : 27**

```
< keyword, int >
< ID, 3 >
< punctuation, , >
< ID, 4 >
< Assign, = >
< Number, 0 >
< punctuation, ; >
< ID, 3 >
< Assign, = >
< Number, 2 >
< punctuation, ; >
< keyword, if >
< punctuation, ( >
< ID, 4 >
< RelOp, <= >
< ID, 3 >
< punctuation, ) >
< punctuation, { >
< ID, 4 >
< Assign, = >
< ID, 4 >
< ArithOp, - >
< Number, 2 >
< ArithOp, * >
< ID, 3 >
< punctuation, ; >
< punctuation, } >
```

**Symbol Table**

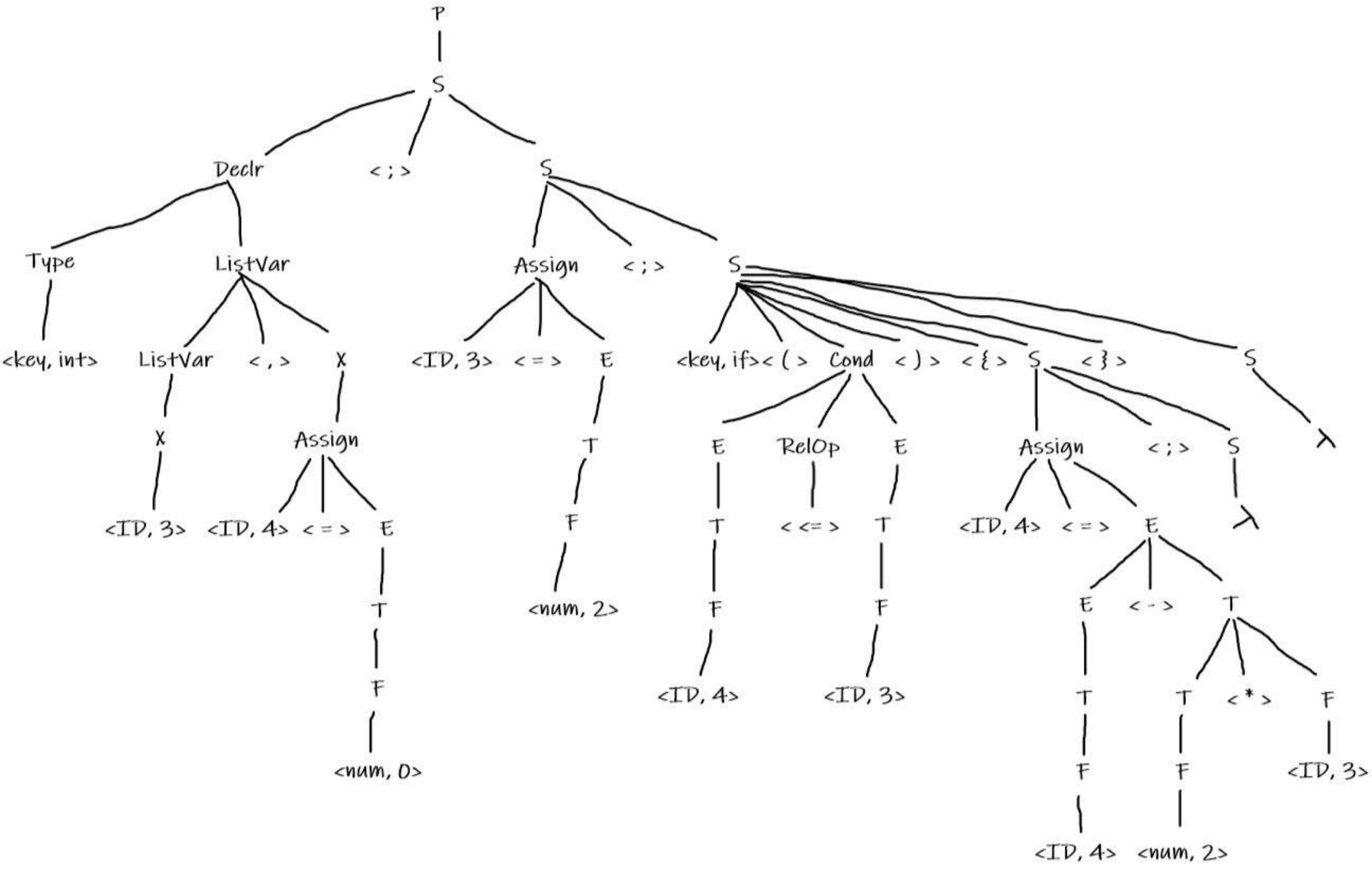
| | name | type | … |
|---|---|---|---|
| 1 | … | | |
| 2 | …. | | |
| 3 | rhs | | |
| 4 | lhs | | |

```
int rhs, lhs = 0;

rhs = 2;

if (lhs <= rhs)

{

lhs = lhs – 2 * rhs;

}
```

# Compiler Design

## Example 2 (continued)

### Phase 2 : Syntax Analysis or




Use the grammar we have defined earlier

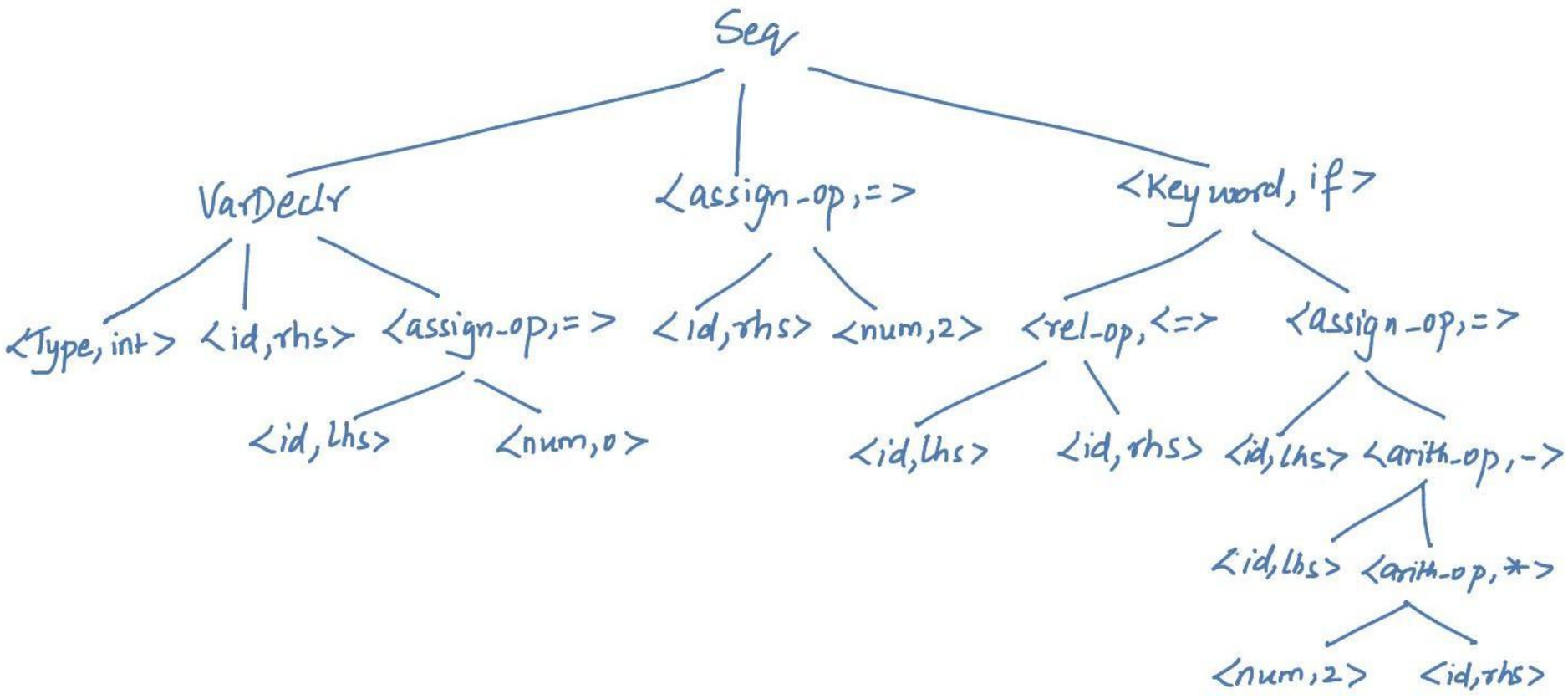
```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

# Compiler Design

## Example 2 (continued)

### Phase 3 : Semantic Analyzer

**Output: Semantically checked Syntax Tree**




```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

# Compiler Design

## Example 2 (continued)

### Phase 4 : Intermediate Code Generator

### Output: Three Address Code (3AC or TAC)

```
        lhs = 0
        rhs = 2
        t0 = lhs <= rhs
        if t0 goto L1
        goto L2
L1 : t1 = 2 * rhs
        t2 = lhs - t1
        lhs = t2
L2 : next
```

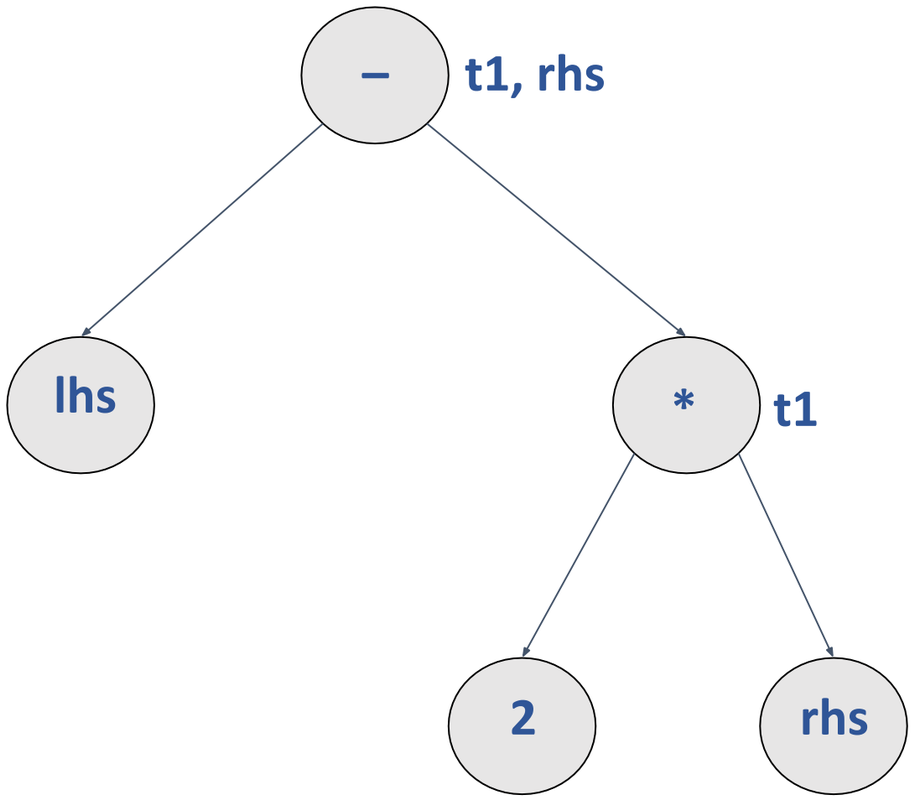
```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

# Compiler Design

## Example 2 (continued)

### Phase 5 : Machine Independent Code

### Optimization Output: Optimized IR




```
    lhs = 0
    rhs = 2
    t0 = lhs <= rhs
    if t0 goto L1
    goto L2
L1: t1 = 2 * rhs
    lhs = lhs - t1
L2: next
```

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
lhs = lhs – 2 * rhs;
}
```

# Compiler Design

## Example 2 (continued)

### Phase 6 : Code

### Generator Output:

Assembly Code

```
        MOV   R1,#0
        ST    lhs,R1
        MOV   R2, #2
        ST    rhs,R2
        SUB   R3,R1,R2
        BLTZ  R3,L2
        LD    R4,#2
        MUL   R4,R2
        SUB   R5,R1,R4
        ST    R5,lhs
L2  :   ...
```

```
int rhs, lhs = 0;

rhs = 2;

if (lhs <= rhs)

{

lhs = lhs – 2 * rhs;

}
```

# Example 3

**Example 3 : Consider the following piece of code:**

```
n = 23;
for (i = 0; i < n; i++)
{
   sum = sum * i;
}
```

**We will now run this code through the different phases of a compiler and see what the output will be at each stage.**

# Compiler Design

## Example 3 (continued)

PES UNIVERSITY
CELEBRATING 50 YEARS

### Phase 1 : Lexical Analysis or Linear Analysis or Scanner

| Pattern | Lexemes matched |
|---|---|
| Keyword | for |
| Identifier | n i i n i printf sum sum i |
| RelOp (<, <=, >, >=, ==, !=) | < |
| LogOp (&&, \|\|, ~) | |
| Assign (=) | = = = |
| ArithOp (*, +, -, /) | * |
| Punctuation ( (, ), {, }, [, ], ;) | ; ( ; ; ) { ; } |
| Number | 23 0 |
| Literal – anything in double quotes | |
| Inc_op | ++ |
| Dec_op | |

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

# Compiler Design

## Example 3 (continued)

**Output: # of tokens : 25**

```
< ID, 1 >
< = >
< Number, 23 >
< Keyword, for >
< ( >
< ID, 2 >
< assign_op, => 
< Number, 0 >
< ; >
< ID, 2 >
< < >
< ID, 1 >
< ; >
< ID, 2 >
< Inc_op, ++ >
< ) >
```

**Symbol Table**

| | name | type | … |
|---|---|---|---|
| 1 | n | | |
| 2 | i | | |
| 3 | sum | | |
| | … | | |

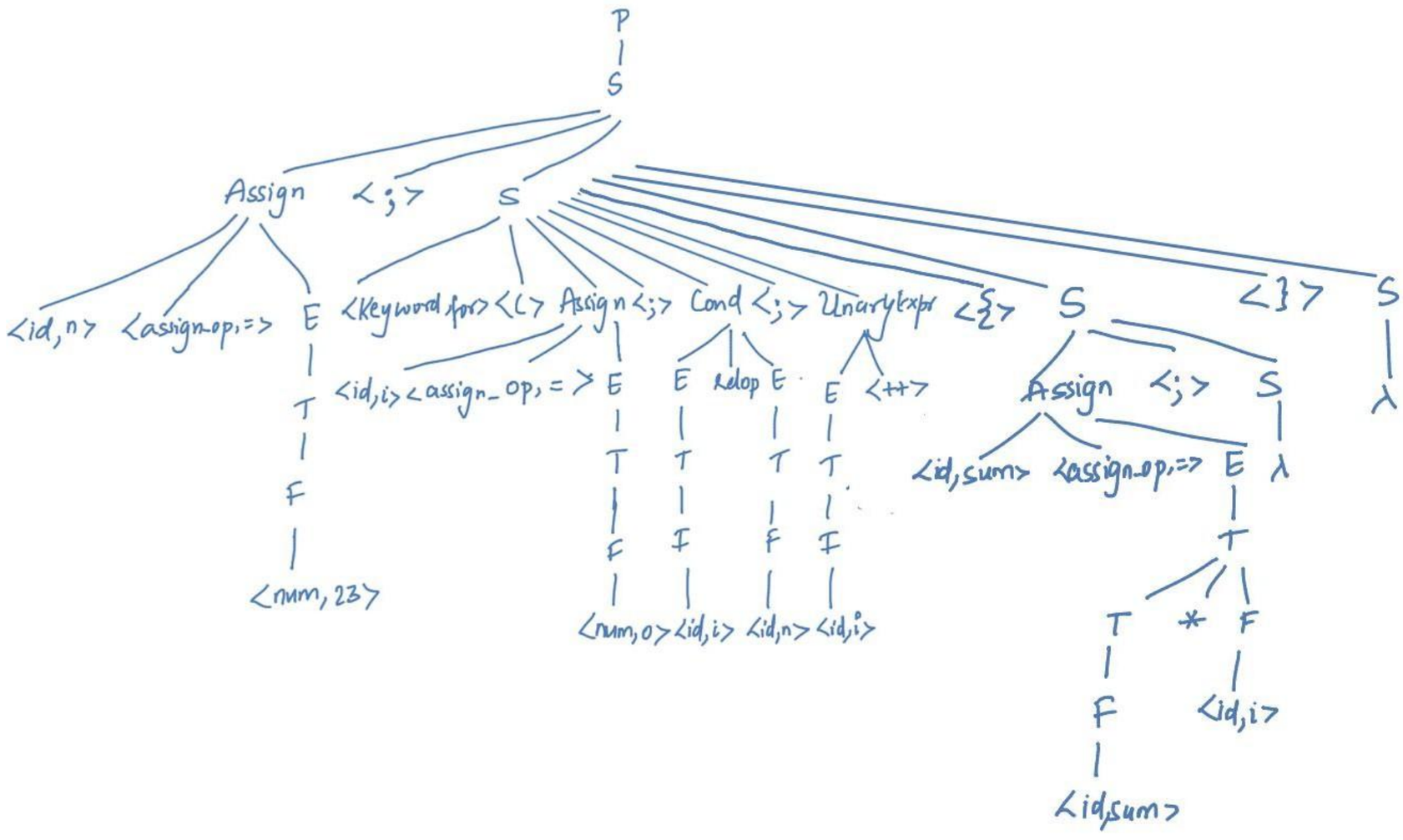
```
< { >
<ID, 3>
<assign_op, =>
<ID, 3>
<arith_op, *>
<ID, 2>
< ; >
< } >
```

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

# Compiler Design

## Example 3 (continued)

### Phase 2 : Syntax Analysis or




**Assume slightly modified grammar (with functions)**

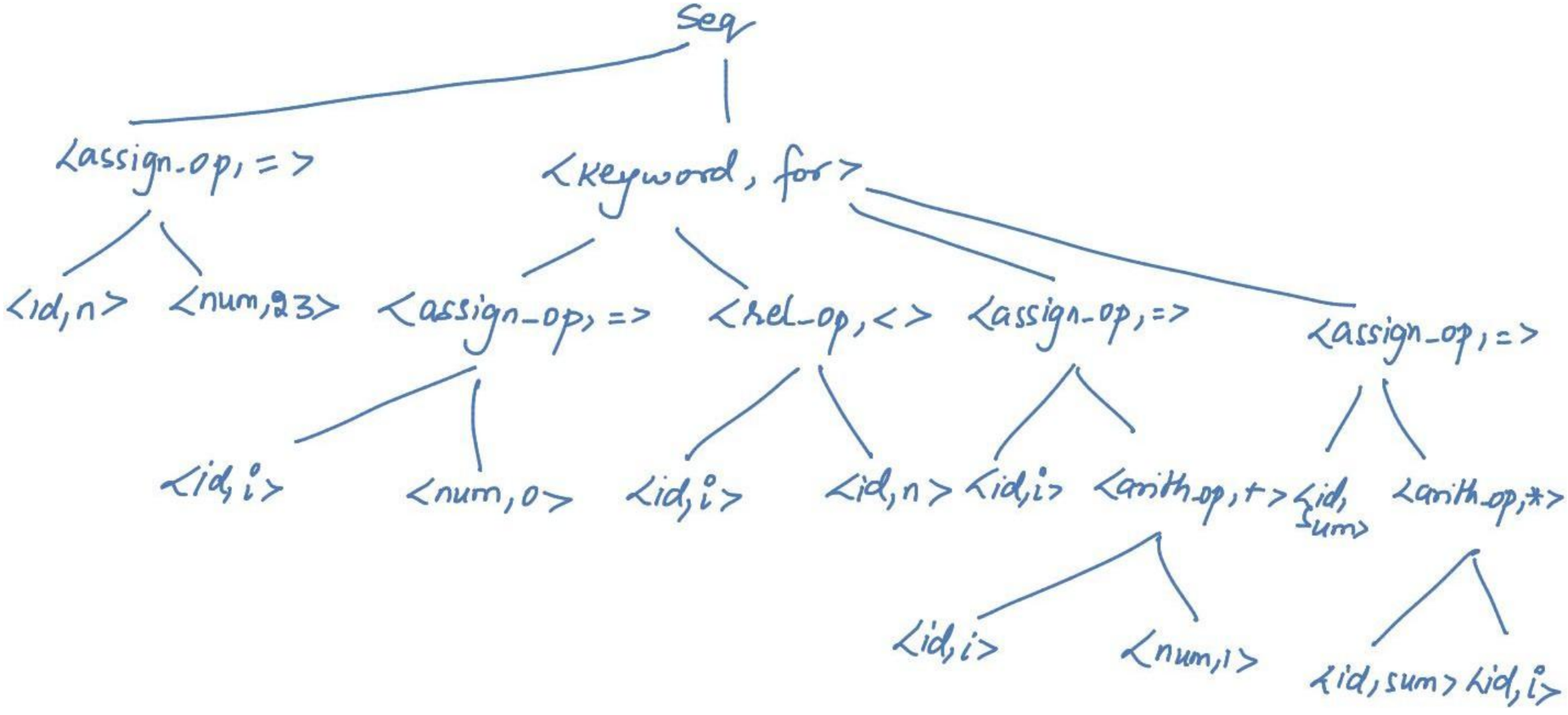
```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

# Compiler Design

## Example 3 (continued)

### Phase 3 : Semantic Analyzer

**Output: Semantically checked Syntax Tree**




```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

# Compiler Design

## Example 3 (continued)

**Phase 4 : Intermediate Code Generator**

**Output: Three Address Code (3AC or TAC)**

```
        number = 23
        i = 0
L0:     if i < n goto L1
        goto L2
L1:     t1 = sum * i
        sum = t1
        t2 = i + 1
        i = t2
        goto L0
L2 : ...
```

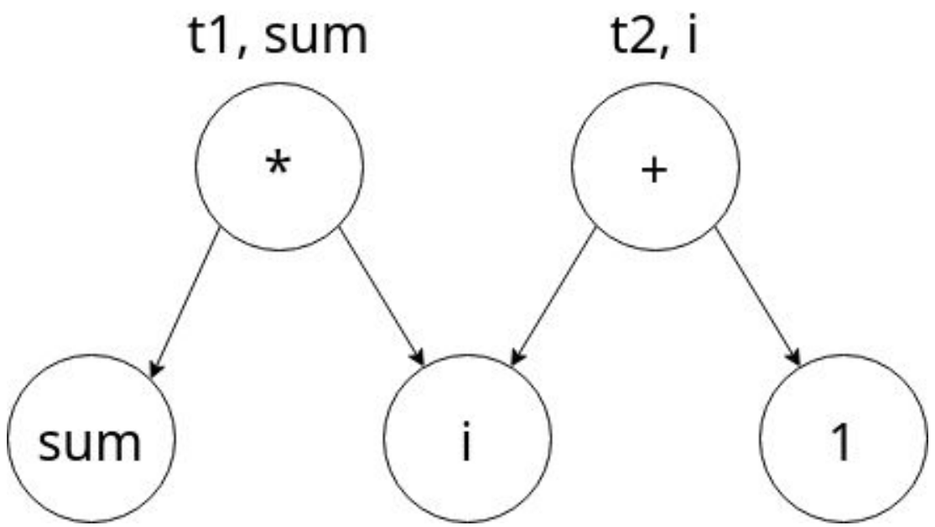
```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

# Compiler Design

## Example 3 (continued)

### Phase 5 : Machine Independent Code Optimization

Output: Optimized IR




```
    number = 23
    i = 0
L0: if i < n goto L1
    goto L2
L1: sum = sum * i
    i = i + 1
    goto L0
L2: ...
```

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

# Compiler Design

## Example 3 (continued)

### Phase 6 : Code

**Generator Output:**

**Assembly Code**

```
        MOV R1, #23    // R  → n
        MOV R2, #0     // R2 → i
        MOV R4, sum    // R4 = contents(sum)
L0:  SUB R3, R1, R2 // i - n
     BLZ R3, L1
     BR L2
L1:  MUL R4, R4, R2
     BR L0
L2:  ...
```

```
n = 23;
for (i = 0; i < n; i++)
{
sum = sum * i;
}
```

# Compiler Design

## Example 4 - Take home exercise

**Exercise Problem : Consider the following piece of code:**

```
int n = 3;
do
{
   if( (n/2)*2 == n )
   {
      n = n / 2;
   }
   else
   {
      n = 3 * n + 1;
   }
}
while(n != 1 || n != 2 || n != 4); return n;
```

This problem is for the understanding of the concept by testing you on if-else and do-while constructs.

Note: The Parse Tree may become large, try to do the if-else part separately

Hint: Consider
S -> do {S} while (S); S
in the grammar

# THANK YOU

**Preet Kanwal**

Department of Computer Science & Engineering

**preetkanwal@pes.edu**