

Compiler Design

DATE: / /
PAGE NO.: / /

Unit IV Storage allocation & Error Handling

Symbol Table

Unit 4

- Symbol Table (S_1, S_2, S_3) -

Any 3 }
• common errors during lexical analysis
phase (S_4) -

- YACC (in short) (S_5) -
 - Storage Alloc
 - Activation

+ and (copy Numericals)
Unit V Code Optimization

- Block + Flowgraph + DAG
(Numericals) [Copy]

- DAG + Block (theory) ($S_7 \& S_8$)

Any 1 f. Loop Optimization

Any 1 f. Peephole -

Unit VI Code Generation

- Heuristic ordering (DAG) (S_{12})

- Labelling algorithm (S_{13})

- Simple code generator ($S_{14} \& S_{16}$)
 - [in short]

(Numericals in copy if any)

Q

Symbol Table

Ans

Definition :- The symbol table is defined as the set of Name & Value pair.

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about instances of various entities such as variable & function names, classes, objects, etc.

> It is built-in lexical and syntax analysis phases.

> The information is collected by the analysis phases of the compiler and is used by the synthesis phases of the compiler to generate code.
> It is used by the compiler to achieve compile-time efficiency.
> It is used by various phases of the compiler as follows:-

1. Lexical Analysis : Creates new table entries in the table, for example like entries about tokens.
2. Syntax analysis :- Adds information regarding attribute type, scope dimension, line of reference, use, etc in table.

3. Semantic analysis :- used available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct (type check).

and update it accordingly

4. Intermediate code generation :- refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.

5. Code optimization :- uses information present in the symbol table for machine-dependent optimization.

6. Target code generation :- generates code by using address information of identifier present in the table.

> Information used by the compiler

from symbol table.

- data type & name
- declaring procedures
- offset in storage
- structure or record than, a pointer to structure table

- For parameters, whether parameter passing by value or by reference
- Number & type of arguments passed to function • base Address.

> Operations of symbol table
The basic operations defined on a symbol table include :

Operation Function

- allocate - to allocate a new empty symbol table
- free - to remove all entries & free storage of symbol table
- lookup - to search for a name and return pointer to its entry

- insert - to insert a name in symbol table and return pointer to its entry
- set-attribute - to associate an attribute with a given entry
- get-attribute - to get an attribute associated with a given entry.

$\begin{array}{|c|c|c|c|} \hline & a & \text{int} & + \\ \hline & | & | & \downarrow \\ \hline & LB_1 & & UB_1 \\ \hline \end{array}$

$LB \rightarrow \text{lower Bound}$

$UB \rightarrow \text{upper Bound}$

$\begin{array}{|c|c|c|c|} \hline & & & + \\ \hline & & & \downarrow \\ \hline & & & LB_2 \\ \hline \end{array}$

> Various approaches to symbol table organization.

① The linear list

- It's the easiest way to implement a symbol table.
- The new names are added to the table in the order that they arrive.
- Whenever a new name is to be added to the table, the table is first searched linearly to sequentially check whether or not the same name is already present in the table.
- If the name is not present, then the record for new name is created and added to the list at a position specified by the available pointer.

② Binary search tree

- A search tree is more efficient approach to symbol table.
- We add 2 links, left & right in each record, and these links point to tree records in the search tree.
- All names are created as child of the root node that always follows the property of the binary search tree.
- Insertion & lookup are O(log n).
- On average.

3) Hash table

- In hash scheme, 2 tables are maintained - a hash table & symbol table and are the most commonly used methods.
- A hash table is an array with an index range : 0 to table size - 1.
- These entries are pointers pointing to the names of the symbol table.
- Insertion and lookup can be made very fast - $O(1)$.
- Adv :- Quick to search
- Dis :- Hashing is complicated to implement

Advantages

- Improved code reuse
- Faster code execution
- Better coding structure

Disadvantages

- Increased memory consumption
- Increased processing time
- Complexity & Scalability & functionality
- Limited applications
- Resolution of scope issues
- Error checking & code debugging

- Q What are some common errors that can occur during the lexical analysis phase of a compiler, and what are the different methods used for error recovery in such cases?
- * Common errors occurring in lexical analysis phase
- Long numerical literals
 - Too long numerical literals
 - Badly formed numerical literals
 - Input characters that are not in source code
 - Spelling mistakes
 - Unmatched string or comments.
- ⇒ Error recovery methods for lexical errors including:
1. Panic Mode Recovery
 - Involves skipping input until a designated synchronized token is found enabling the compiler to recover from lexical errors by continuing parsing
 2. Global Correction
 - Involves analyzing the input (entire IP) to identify and correct lexical errors.

Example:

Consider the following C code

with a mis-spelling variable :

```
int main()
{
    printf("The number is : %d\n", num);
```

```
''' python
def hello():
    print("Hello, world! $");
```

The presence of illegal character \$ at the end of the string triggers a lexical error. Using panic mode recovery, the compiler would discard characters until it reaches a synchronizing token, such as a semicolon. After "recovery", the code might look like this

```
''' python
def hello():
    print("Hello world!"); // $
```

discarded

Example : Consider the following Python code with an illegal character.

The miss-spelling function `printf' triggers a lexical error. With global correction, the entire compiler might analyze the entire code to suggest corrections. After correction, the code looks like this:

```
""  
#include <stdio.h>  
  
int main()  
{  
    int num = 10;  
    printf ("The number is : %d\n", num);  
}  
// Correction applied
```

3. Interactive Error Handling

- Involves interacting with the user or programmer to resolve lexical errors.

Example: Consider the following user input Enter on arithmetic expression $2 * (3 + 4) - 5$. An extra closing parenthesis is seen ")".

causing lexical error to resolve it the user might be asked to correct the expression using an interactive environment. corrected expression -

Provide an overview of YACC (Yet Another Compiler with labelled diagram in the context of compiler design, and how it is utilized).

Explain in short automatic error recovery in YACC.

- YACC (Yet another compiler compiler) is a tool used in compiler design for generating the syntax analyzer (parser) of a compiler.

- Its purpose is to automate the process of generating a parser for a given grammar specification.
- YACC takes a formal grammar specification typically in the form of a set of production rules, and generates a parser in C or other programming languages.

Overview:-

1. Grammar Specification

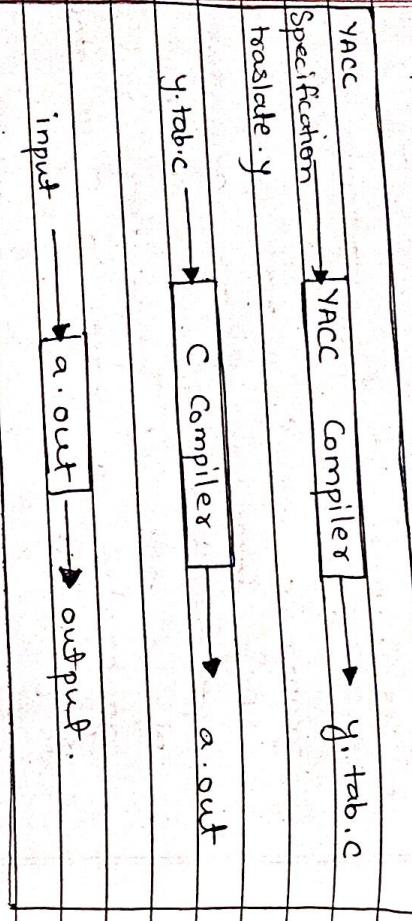
- First step in using YACC is to define the grammar of the language to be parsed.
- Typically specified using Backus-Naur Form (BNF) or Extended BNF (EBNF)

2. YACC Input File

- The grammar specification is written in a file with a '.y' extension, serves as input to the YACC tool.

4. Parser Integration
- The generated parser code is then integrated into the compiler project.

5. Compilation
- The compiler project including the generated parser is compiled to produce an executable compiler.



* Purpose

- Automated Parser Generation
- Language Independence
 - Efficiency
 - Error Handling

Utilization

- Compiler Construction
- Interpreter Development
- Language Processing Tools
- Educational Purposes

Automatic Error Recovery in YACC

- The tool YACC can generate a parser with the ability to automatically recover from the errors.
- Major non terminals, such as those for program blocks or statements, are identified; and then errors, productions of the form $A \rightarrow \text{error}$ are added to the grammar, where $\text{a} \in \Sigma$ is usually ϵ .
- When YACC generates parser encounters an error, it finds the top-most state on its stack, whose underlying set of items includes an item of the form $A \rightarrow \text{error}$. The parser shifts the token error and a reduction to A is immediately possible.
- The parser then invokes a semantic action associated with production $A \rightarrow \text{error}$, and thus semantic action takes care of recovering from the error.

Storage allocation

Q

Short Note on Storage allocation tasks that a compiler must perform is to allocate the resources of the target machine.

The resources of the data objects that are to represent the data objects that are being manipulated by the source program.

That is, a compiler must decide a run-time representation of the data objects in the source program.

Data objects in the source program run-time representation of the data objects such as integers of the data objects such as arrays and real variables.

Data structures such as arrays & strings are represented by several words of machine memory.

There are mainly 3 types of storage allocation

Storage strategy:

1. Static Allocation → Simple to understand
2. Heap Allocation → most flexible
3. Stack Allocation → Allocation of memory at run-time

Unit V

Q

Elaborate code Optimization. Discuss with suitable examples about mlc independent & machine-dependent optimization.

Code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (like CPU, memory) so that faster-running mlc code will result.

* Compiler optimization process should meet the following objectives:

- > It must be correct, it must not, in any way, change the meaning of the program
- > Should increase the speed & performance of the program
- > The compilation time must be reasonable.

Machine dependent C.O

Refers to optimizing code specifically for a particular type of computer or hardware limited to a specific hardware platform

Better Performance on

specific hardware platform not useable in any other platform

across all platforms

Equally reusable in any other platform

* Example

Machine Independent Optimization: (MIO)

Ex: Consider the following expression.

(Constant Folding)

int result = 10 * 5 + 3

During MIO, the compiler can evaluate the constant expressions at compile time. In this case the expression $10 * 5$ calculated to 50 followed by $50 + 3$ " " 53. So the optimized code would be

int result = 53

Machine dependent optimization: (MDO)

Ex: Instruction selection

If the source code contains a loop that increments a variable, the compiler might select a variable machine instruction that efficiently generates multiple instructions to achieve the same result.

DAG (Directed Acyclic Graph)

- A DAG is a finite directed graph with no directed cycles.

- In other words, you cannot follow a sequence of edges to return to a vertex you've already visited.

- DAGs have vertices (nodes) connected by directed edges, and each edge has a direction from one vertex to another.

Example $x = y \text{ OP } z$

① $a = b + c$ ② $\text{if}(i <= 20) \text{goto } L_1$

$b = a - d$

$c = b + c$

$d = a - d$

\oplus^c

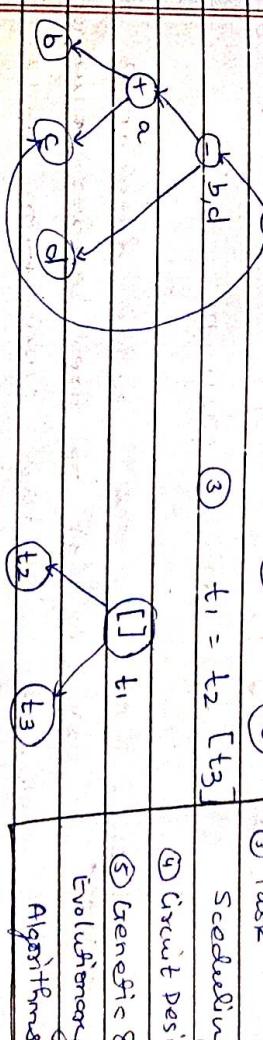
③ $t_1 = t_2 [t_3]$

Scheduling

④ Task Resolution

⑤ Circuit Design

⑥ Genetic & Evolutionary Algorithms



* Role of DAG in Code Generation

① Expression representation

② Optimizations

③ Common Subexpression Elimination (CSE)

④ Dead Code Elimination

⑤ Constant Folding & Propagation

⑥ Register Allocation

⑦ Instruction Selection

⑧ Loop Optimization

Step 2 Building Blocks

Q8. Basic Block is a sequence of statements in which flow of control enters at the beginning and leaves at the end without any possibility of branching except at the end.

Let us see simple example :-

3 Address code

- (1) $f = 1$
- (2) $i = 2$
- (3) $\text{if } i < \infty \text{ goto } 8$
- (4) $f = f * i$
- (5) $t_1 = i + 1$
- (6) $i = t_1$
- (7) $\text{goto } (3)$

Note :

1. Basic Block :- include all the statements starting from leader to up to, but not including the next leaders
2. No of leader = No of \star blocks \approx Nodes
3. $\text{if } x < 3 \text{ goto } 100 \rightarrow$ conditional goto
4. Edges keep an eye on two blocks like which block comes after which block

Step 1 First Identify the leader statements
(1), (3), (4) \rightarrow Are leader statements
Following is the explanation why.



(B_1)

- (1) \rightarrow First statement
- (3) \rightarrow Target goto
- (4) \rightarrow immediately follows a conditional statement
- (8) \rightarrow because it is a target of conditional goto statement



- B₂ Follows B₁
B₃ Follows B₂
B₄ Follows B₂
B₂ Follows B₃

PAGE NO.: 1 / 1
DATE: 1 / 1

Peephole

- Simple but effective technique for improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

Characteristics of Peephole Optimization:

- Redundant instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable

➤ Redundant loads & stores

goto L₁

L₁: goto L₂

⇒

goto L₂

➤ Algebraic simplification

- (1) mov r0, a
- (2) mov a, r0
- we can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register r0
- * Strength reduction

// Before strength reduction // After S-R

int a = 2 int a = 2

int b = a + 3

int b = a + a

➤ Unreachable code

Ex:-

define debug 1
if (debug)

if debug = 1 goto L1 goto L2
L1 : print debugging inf
L2 : ... ca

Optimized code

→ if debug ≠ 1 goto L2

L2 : --- (b) debugging inf

➤ Flow of control

goto L₁

L₁: goto L₂

⇒

goto L₂

x = x + 0 or x = x * 1
This can be easily eliminated using peephole optimization.

- Peephole optimization is type of code optimization performed on a small part of the code.

- The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.

objective

- To improve performance
- To reduce memory footprint
- To reduce code size.

Goals

- Enhance code quality.
- Reduce execution time
- Optimize resource usage.

① Expression evaluation

- Expressions represented by DAGs involve multiple nodes representing operations such as +, ×, and function calls.
- Heuristic ordering determines the sequence in which these operations should be performed.

④ Optimization Technique

- Heuristic ordering strategies aim to prioritize nodes that are likely to yield the most significant optimization benefits, thereby improving the overall quality and efficiency of the generated code.

② Instruction Selection

- By analysing the structure of DAG and considering factors such as instruction latencies, resource constraints, and target architecture characteristics, a suitable order for generating machine instructions can be determined.

③ Register Allocation

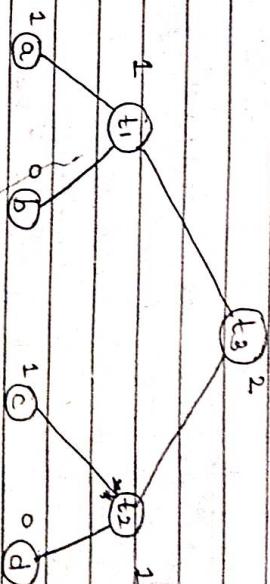
- When performing register allocation, heuristic ordering helps determine the order in which variables are assigned to register.

Example

Labelling algorithm is used by phase

- Labelling algorithm is used to compiler during code generation phase.
- Basically, this algorithm is used to find out how many registers will be required by a program to complete its execution.

- Labelling algorithm works in bottom-up fashion.
- We will start labelling firstly child nodes & then interior nodes.



Advantages

- Efficient code generation
- Flexibility
- Easy to implement

Disadvantages

- Limited optimization
- Increased code size

Overview of how it works :-

1. Intermediate Representation (IR) Parsing :

The code receives (IR) of program generated by front-end of the compiler

- If n is left child its value is 1
- If n is right child its value is 0
- If n is interior node

Assume L₁ & L₂ is left & right child of interior node

If L₁ = L₂ then value of n is MAX(L₁, L₂)

L₁ = a + b

$$L_2 = C + D$$

$$L_3 = t_1 + t_2$$

Rules

- if 'n' is a leaf node
- If n is left child its value is 1
- If n is right child its value is 0
- If n is interior node

Assume L₁ & L₂ is left & right child of interior node

If L₁ = L₂ then value of n is MAX(L₁, L₂)

If L₁ != L₂ then value of n is MAX(L₁, L₂)

For each node m/c code sequence is generated.

For each node m/c code sequence is generated.

4. Output Generation: code is written to

The generated code, buffer, or memory file for further processing or

ready for execution

Challenges

- Target Architecture Compatibility
- Register Allocation
- Instruction Selection
- Control Flow Handling
- Error Handling and Recovery

* Best practices for designing & implementing steps

a simple code generated

understand Target Architecture

modular Design

Use an Intermediate Representation (IR)

Traversal Algorithm

continuous Code Generation Rules

Improvement

Optimization Techniques

Performance Profiling

Test & Validation

Documentation & Comments