


Unit - V

Object Code Generation | Error Recovery

Object Code Generation

- Object code forms 
- Machine dependent code optimization
- Register allocation and assignment
- Generic code generation algorithms

Error Recovery

- Various errors in phases and recovery of errors in compilation
- Introduction to tools of compiler

Dr. R. Madana Mohana

Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY

Hyderabad - 500 075, Telangana, INDIA

www.cbit.ac.in

CODE GENERATOR

Issues in the design of a Code Generator

Outline

Introduction to Code Generator

- Position of Code Generator
- Requirements

Issues in the design of a Code Generator

- Input to the Code Generator
- Target Programs (Object Code Forms)
- Memory Management
- Instruction Selection
- Register Allocation
- Choice of Evaluation Order

The Target Machine / Language

Introduction to Code Generator

- Position of Code Generator
- Requirements

Introduction to Code Generator

- **Code Generator** is the 6th (final) phase of compiler.
- Input of the **Code Generator** is the optimized Intermediate Code.
- Output of the **Code Generator** is the target code in Assembly Language.
- Using **Assembler**, Assembly Language code is converted into Machine Language code.

Code Generator

- Position of Code Generator:

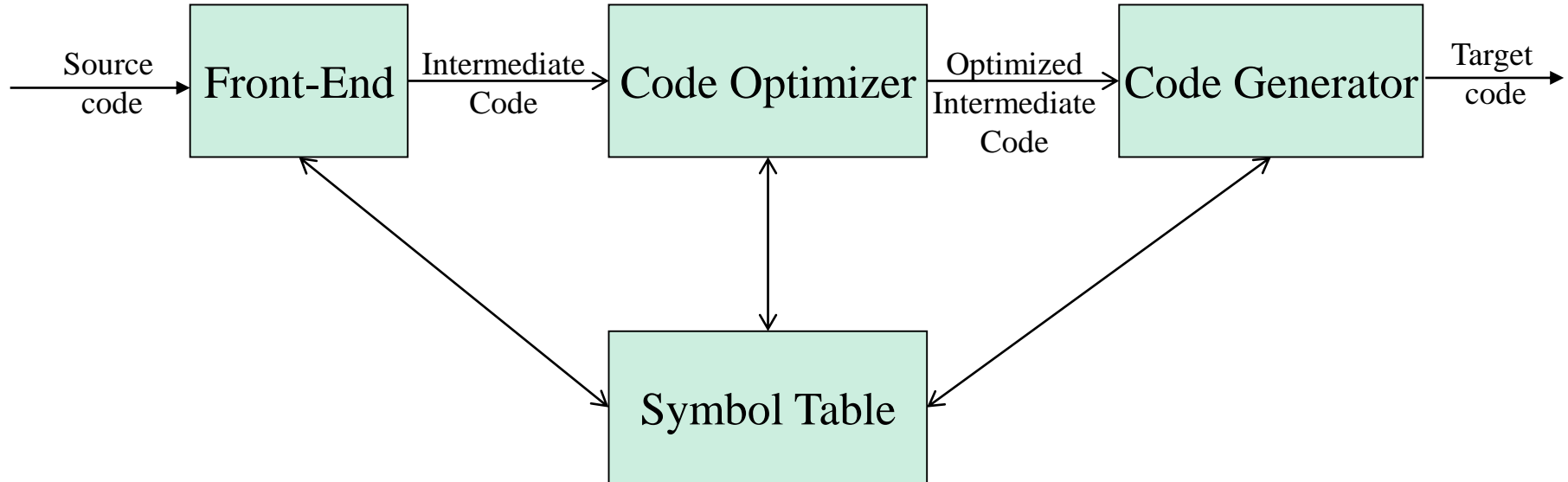


Figure: Position of Code Generator

Code Generator Requirements

- **Output code** must be **correct**:
 - The meaning of the source and the target program must remain the same i.e., given an input, we should get same output both from the target and from the source program.
 - We have no definite way to ensure this condition.
 - What all we can do is to maintain a test suite and check.
- **Output code** must be of **high quality**:
 - The target code should make effective use of the resources of the target machine.
- **Code Generator** should run **efficiently**:
 - It is also of no use if code generator itself takes hours or minutes to convert a small piece of code.

CODE GENERATOR

Issues in the design of a Code Generator

Outline

Introduction to Code Generator

- Position of Code Generator
- Requirements

Issues in the design of a Code Generator

- Input to the Code Generator
- Target Programs (Object Code Forms)
- Memory Management
- Instruction Selection
- Register Allocation
- Choice of Evaluation Order

The Target Machine / Language

Input to the Code Generator

- **Intermediate Representation (IR)** with **Symbol Table**.
 - Assume that **input** has been validated by the **front-end**.
- There are several choices for the **Intermediate Language**, including:
 - **Liner Representations** such as **Postfix Notation**
 - **Three-Address** representations such as **Quadruples**
 - **Virtual Machine** representations such as **Stack-Machine Code**
 - **Graphical** representations such as **Syntax Trees** and **DAGs** (**Directed Acyclic Graph**)

Target Programs (Object Code Forms)

- The **output** of the **Code Generator** is the **Target Program**.
- Like the **Intermediate Code**, this **output** may take on a variety of forms:
 1. **Absolute Machine Language: Fast for small programs**
 2. **Relocatable Machine Language: Requires Linker and Loader.**
 3. **Assembly Language: Requires Assembler, Linker and Loader.**

Target Programs (Object Code Forms)

- Absolute Machine Language: **Fast for small programs**
 - Producing an **absolute machine language** as **output** has the advantage that it can be placed in a **fixed location** in **memory** and **immediately executed**.
 - A **small program** can be thus **compiled** and **executed** quickly.

Target Programs (Object Code Forms)

- Relocatable Machine Language: Requires **Linker** and **Loader**.
 - Producing a **relocatable machine code** as **output** allows **subprograms** to be compiled separately.
 - Although we must pay the added expense of **linking** and **loading** if we produce **relocatable object modules**, we gain a great deal of **flexibility** in being able to **compile subroutines** separately and to call other **previously compiled programs** from an **object module**.

Target Programs (Object Code Forms)

- **Assembly Language**: Requires **Assembler**, **Linker** and **Loader**.
 - Producing an **assembly code** as **output** makes the process of **code generation** easier as we can generate **symbolic instructions**.
 - The price paid is the **assembling**, **linking** and **loading** steps after **code generation**.

Memory Management

- Mapping **names** in the **source program** to **addresses** of **data objects** in **run-time memory** is done cooperatively by the **Front-end** and the **Code Generator**.
- If **Machine code** is being generated, **labels** in **three-address** statements have to be converted to **address of instructions**.

Instruction Selection

- Choosing appropriate **target-machine instructions** to implement the **IR (Intermediate Representation)** statements.
- **Import Factors** are:
 - Uniformity
 - Completeness
 - Instruction speed
 - Power consumption

Instruction Selection

- Choosing appropriate **target-machine instructions** to implement the **IR (Intermediate Representation)** statements.
- **Import Factors** are:
 - **Uniformity**: i.e. support for different object/data types, what op-codes are applicable on what data types etc.
 - **Completeness**: Not all source programs can be converted/translated in to machine code for all architectures/machines. **E.g.**, 80x86 doesn't support multiplication.
 - **Instruction speed**: This is needed for better performance.
 - **Power consumption**

Instruction Selection

- The **Complexity** of mapping **IR program** in to **code-sequence** for **target machine** depends on:
 - **Level of IR** (high - level or low level)
 - **Nature of Instruction set** (data type support)
 - **Desired quality of generated code** (Speed and size)

Instruction Selection

- **Ex:** The sequence of statements

a := b + c

d := a + e

would be translated into

1. MOV b, R₀

2. ADD c, R₀

3. MOV R₀, a

4. MOV a, R₀

5. ADD e, R₀

6. MOV R₀, d

Instruction Selection

- **Ex:** The sequence of statements

1. MOV b, R₀

2. ADD c, R₀

3. MOV R₀, a

4. MOV a, R₀

5. ADD e, R₀

6. MOV R₀, d

- Here the **fourth statement MOV a, R₀** is **redundant**, and so is the **third** if ‘**a**’ is not subsequently used.
- This **statement (4)** can be eliminated by introducing an “**increment instruction (INC)**”, then the **three-address statement $a := a + 1$** may be implemented more efficiently by the single instruction **INC a** rather than **MOV a, R₀**.

Instruction Selection

- **Ex:** The sequence of statements

∴ **a := a + 1** would be translated into

MOV a, R₀ INC a

ADD #1, R₀

MOV R₀, a

Register Allocation

- **Register allocation and Assignment**
 - ✓ Deciding what **values** to keep in which **registers**.
- **Register Allocation:**
 - ✓ Selecting the **set of variables** that will reside in **registers** at each point in the program.
- **Register Assignment:**
 - ✓ Picking the **specific register** that a **variable** reside in.
- **Efficient utilization of registers in generating good code:**
 - ✓ **Instructions** with **register operands** are faster
 - ✓ Store **long life time** and **counters** in **registers**
 - ✓ **Temporary** Locations
 - ✓ **Even odd** register pairs.

Register Allocation

Ex :

Two three-address code sequences

(a)

$t := a + b$

$t := t * c$

$t := t / d$

(b)

$t := a + b$

$t := t + c$

$t := t / d$

The **shortest** assembly-code sequences for (a) and (b) are given below:

Register Allocation

Ex :

(a)

$t := a + b$

$t := t * c$

$t := t / d$

The **shortest** assembly-code sequence for (a) is given below: R_i stands **Register i**

L R_1, a // L-Load

A R_1, b // A-Add

M R_0, c // M- Move

D R_0, d // D- Division

ST R_1, t // ST- Store

Register Allocation

Ex :

(b)

$t := a + b$

$t := t + c$

$t := t / d$

The **shortest** assembly-code sequence for (b) is given below:

R_i stands **Register** i

L R_0, a // L-Load

A R_0, b // A-Add

A R_0, c // A-Add

SRDA $R_0, 32$ // Shifts the **dividend** into R_1 and **clears** R_0 , so all bits equal its sign bit.

D R_0, d // D- Division

ST R_1, t // ST- Store

Choice of Evaluation Order

- The **order** in which the **instructions** will be executed.
- This **increases performance of the code**.
- Selecting the order in which **computations** are performed
- Affects the **efficiency** of the target code
- Picking a **best order** is NP- complete
- **Some orders** require **fewer registers** than others

CODE GENERATOR

Issues in the design of a Code Generator

Outline

Introduction to Code Generator

- Position of Code Generator
- Requirements

Issues in the design of a Code Generator

- Input to the Code Generator
- Target Programs (Object Code Forms)
- Memory Management
- Instruction Selection
- Register Allocation
- Choice of Evaluation Order

The Target Machine / Language

The Target Machine / Language

- The **Target Machine** and its instruction set is a Prerequisite for designing a **good code generator**.
- Use as the **target computer** a **register machine** that is **representative** of several **minicomputers**.
- The **target computer** is **byte-addressable** machine with **four bytes** to a **word** and **n general-purpose registers** R_0, R_1, \dots, R_{n-1} .
- It has **two** address instructions of the form:

OP Source, destination

where **OP** is an **Op-Code** (operation code)

Source and **destination** are **data fields**.

The Target Machine / Language

- It has the following OP-Codes(among others):

MOV - **Move** Source to Destination

ADD - **add** Source to Destination

SUB - **Subtract** Source form Destination

The Target Machine / Language

Addresses in the Target Code or Addressing Modes:

- The **address modes** together with their **assembly-Language** forms and associated **costs** as follows:

MODE	FORM	ADDRESS	ADDED COST
Absolute	M	M	1
Register	R	R	0
Indexed	c (R)	c + Contents (R)	1
Indirect Register	*R	Contents (R)	0
Indirect Indexed	*c(R)	Contents (c + Contents (R))	1

contents (R) – denotes the contents of the **register** or **memory address** represented by **R**
A **Memory location M** or a **register R** represents itself when used as a **source** or **destination**.

The Target Machine / Language

Addresses in the Target Code or Addressing Modes:

For example, the instruction

Mov R₀ , M

Stores the contents of register **R₀** into memory location **M**.

- An address offset **c** from the value in register **R** is written as **c (R)**. Thus,

Mov 4 (R₀) , M

Stores the value **contents (4 + contents (R₀))** into memory location **M**

The Target Machine / Language

Addresses in the Target Code or Addressing Modes:

- **Indirect** versions of the **register** and **indexed modes** are defined by prefix *****, Thus,

Mov *4 (R₀) , M

Stores the value **contents (contents (4+contents (R₀)))** into memory location **M**

- A final address mode allows the **source** to be a **constant**:

MODE	FORM	CONSTANT	ADDED COST
Literal	#c	c	1

Thus, the instruction **MOV #1 , R₀** Loads the constant **1** into register **R₀**

The Target Machine / Language

Instruction Costs:

- **Cost of an instruction = 1 + Cost of operands** (added cost in the table for addressing modes above discussed)
- **Cost of register operand = 0**
- **Cost involving memory and constants = 1**
- **Cost of a program = sum of instruction costs**

The Target Machine / Language

Instruction Costs: Examples

- The instruction **MOV** R_0, R_1 copies the contents of register R_0 into register R_1 . This instruction has cost **one**, since it occupies only **one word of memory**.
- The (**Store**) instruction **MOV** R_5, M copies the contents of register R_5 into memory location M . This instruction has cost **two**, since the address of memory location M is in the word following the instruction.
- The instruction **ADD** $\#1, R_3$ adds the **constant 1** to the contents of register R_3 , and has cost **two**, since the **constant 1** must appear in the **next word** following the instruction.

The Target Machine / Language

Instruction Costs: Examples

- The instruction **SUB 4 (R₀) , *12 (R₁)** stores the value
contents (contents (12+contents (R₁))) - contents (contents (4+R₀)) into the destination ***12 (R₁)**.

The cost of this instruction is **three**, since the **constants 4 and 12** are stored in the **next two words** following the instruction.

The Target Machine / Language

Code Generation: Examples of code to generate for a three address statement of the form $a := b + c$ where b and c are simple variable in distinct memory locations denoted by these statements.

Examples: *Type 1*

Three-address statement	Assembly Code	Added Cost
$a := b + c$	MOV b, R_0	1
	ADD c, R_0	2
	MOV R_0, a	2
Total Cost		6

The Target Machine / Language

Code Generation: Examples of code to generate for a three address statement of the form $a := b + c$ where b and c are simple variable in distinct memory locations denoted by these statements.

Examples: *Type2*

Three-address statement	Assembly Code	Added Cost
$a := b + c$	MOV b, a	3
	ADD c, a	3
Total Cost		6

The Target Machine / Language

Code Generation: Examples of code to generate for a three address statement of the form $a := b + c$ where b and c are simple variable in distinct memory locations denoted by these statements.

Examples: *Type3*

Assuming R_0 , R_1 and R_2 contain the address of a , b and c respectively, we can use:

Three-address statement	Assembly Code	Added Cost
$a := b + c$	MOV $*R_1, *R_0$	1
	ADD $*R_2, *R_0$	1
Total Cost		2

The Target Machine / Language

Code Generation: Examples of code to generate for a three address statement of the form $a := b + c$ where b and c are simple variable in distinct memory locations denoted by these statements.

Examples: *Type4*

Assuming R_1 and R_2 contain the values of b and c respectively, and that the value of b is not needed after the assignment we can use:

Three-address statement	Assembly Code	Added Cost
$a := b + c$	ADD R_2, R_1	1
	MOV R_1, a	2
Total Cost		3