THEORY OF COMPUTATION AND COMPILERS

Unit - V

Object Code Generation| Error Recovery

Object Code Generation

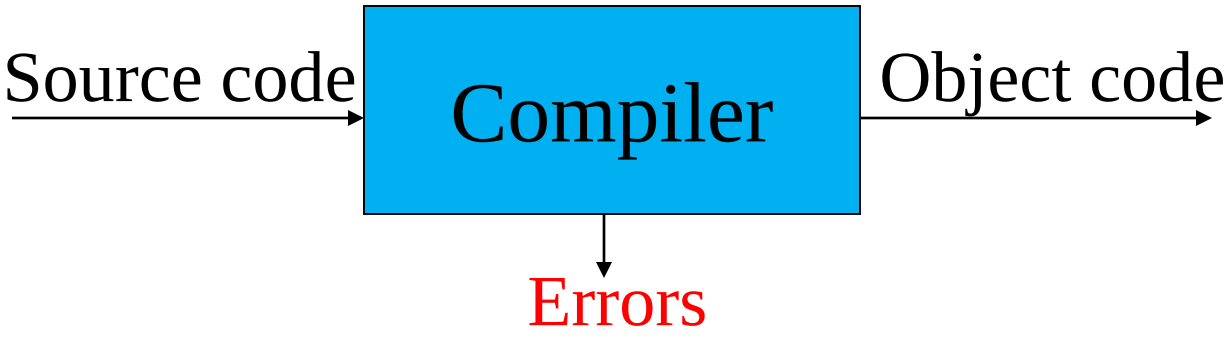
- Object code forms
- Machine dependent code optimization (Peephole Optimization)
- Register allocation and assignment
- Generic code generation algorithms

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

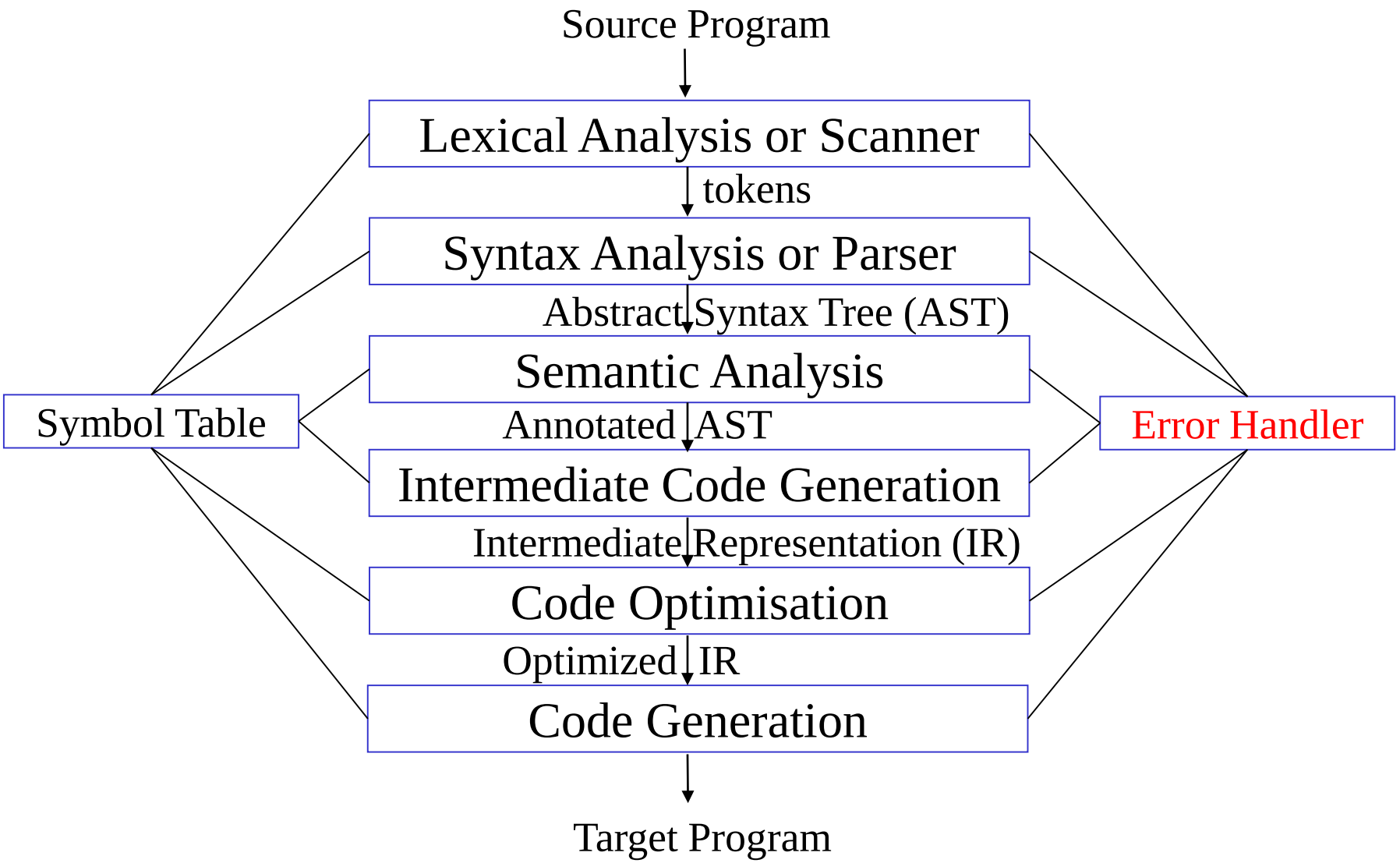
General Structure of a Compiler



The compiler:

- must generate correct code.
- must recognise errors.
- analyses and synthesises.

General Structure of a compiler: Phases of a compiler



Error Handler

- This routine is invoked whenever an error is encountered in any of the phases.
- This routine attempts a correction so that subsequent statements need not be discarded and many errors can be identified in a single compilation.

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string *fi* is encountered for the first time in a C program in the context:

  fi (a == f(x)) ...

  - a lexical analyzer cannot tell whether *fi* is a misspelling of the keyword if or an undeclared function identifier.
  - Since *fi* is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler-probably the parser in this case-handle an error due to transposition of the letters.

Lexical Errors

- However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
- The simplest recovery strategy is *"panic mode"* recovery.
- We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.
- This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Lexical Errors

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

The Role of the Parser

Parsing:

- Parser also called syntax analyzer is the one which does parsing.
- Parsing is the process of getting tokens from the lexical analyzer and obtains a derivation for the sequence of tokens and builds a parse tree.
- Thus, if the program is syntactically correct, the parse tree is generated.
- If a derivation for the sequence of tokens does not exist i.e., if the program is syntactically wrong, it results in syntax error and the parser displays the appropriate error messages.
- The parse trees are very important in figuring out the meaning of a program or part of the program.
- The parse tree is also called syntax tree or derivation tree.

Syntax Error Handling

- Two strategies, called panic-mode recovery and phrase-level recovery, are discussed in more detail in connection with specific parsing methods.
- If a compiler had to process only correct programs, its design and implementation would be simplified greatly.
- However, a compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs, despite the programmer's best efforts.

Syntax Error Handling

- Most programming language specifications do not describe how a compiler should respond to errors; error handling is left to the compiler designer.
- Planning the error handling right from the start can both simplify the structure of a compiler and improve its handling of errors.

Syntax Error Handling

Common programming errors can occur at many different levels.

- ***Lexical errors*** include misspellings of identifiers, keywords, or operators. e.g., the use of an identifier **`elipseSize`** instead of **`ellipseSize`** and missing quotes around text intended as a string.

Syntax Error Handling

Common programming errors can occur at many different levels.

- ***Syntactic errors*** include misplaced semicolons or extra or missing braces; that is, **"{"** or **"}"**. As another example, in **C** or **Java**, the appearance of a case statement without an enclosing **switch** is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

Syntax Error Handling

Common programming errors can occur at many different levels.

- **`Semantic errors`** include type mismatches between operators and operands, e.g., the return of a value in a Java method with result type **`void`**.

Syntax Error Handling

Common programming errors can occur at many different levels.

- **`Logical errors`** can be anything from incorrect reasoning on the part of the programmer to the use in a **`C`** program of the assignment operator **`=`** instead of the comparison operator **`==`**. The program containing **`=`** may be well formed; however, it may not reflect the programmer's intent.

Syntax Error Handling

- The precision of **parsing** methods allows **syntactic errors** to be detected very efficiently.
- Several **parsing methods**, such as the **LL** and **LR** methods, detect an error as soon as possible; that is, when the **stream of tokens** from the **lexical analyzer** cannot be parsed further according to the grammar for the language.

Syntax Error Handling

- More precisely, they have the ***viable-prefix property***, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.
- Another reason for emphasizing **error recovery** during **parsing** is that many errors appear **syntactic**, whatever their cause, and are exposed when **parsing** cannot continue. A few **semantic errors**, such as type mismatches, can also be detected efficiently; however, accurate detection of semantic and logical errors at compile time is in general a difficult task.

Syntax Error Handling

The error handler in a parser has goals that are simple to state but challenging to realize:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

Error-Recovery Strategies in Parser

Types of Error-Recovery Strategies

1. Panic-Mode Recovery
2. Phrase-Level Recovery
3. Error Productions
4. Global Correction

Error-Recovery Strategies in Parser

1. Panic-Mode Recovery

- With this method, on discovering an error, the **parser** discards input symbols one at a time until one of a designated set of ***synchronizing tokens*** is found.
- The ***synchronizing tokens*** are usually delimiters, such as semicolon or **}**, whose role in the source program is clear and unambiguous.
- The **compiler designer** must select the ***synchronizing tokens*** appropriate for the source language.

Error-Recovery Strategies in Parser

1. Panic-Mode Recovery

- While **panic-mode** correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of **simplicity**, and, unlike some methods to be considered later, is guaranteed not to go into an infinite loop.

Error-Recovery Strategies in Parser

2. Phrase-Level Recovery

- On discovering an error, a **parser** may perform **local correction** on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the **parser** to continue.
- A typical **local correction** is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
- The choice of the **local correction** is left to the **compiler designer**.

Error-Recovery Strategies in Parser

2. Phrase-Level Recovery

- we must be careful to choose replacements that do not lead to **infinite loops**, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol.
- **Phrase-level** replacement has been used in several **error-repairing compilers**, as it can correct any input string.
- Its **major drawback** is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error-Recovery Strategies in Parser

3. Error Productions

- By anticipating common errors that might be encountered, we can augment the **grammar** for the **language** at hand with productions that generate the erroneous constructs.
- A **parser** constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during **parsing**.
- The **parser** can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

Error-Recovery Strategies in Parser

4. Global Correction

- Ideally, we would like a **compiler** to make as few changes as possible in processing an incorrect input string.
- There are **algorithms** for choosing a minimal sequence of changes to obtain a **globally** least-cost correction.
- Given an incorrect input string **x** and grammar **G**, these algorithms will find a **parse tree** for a related string **y**, such that the number of insertions, deletions, and changes of tokens required to transform **x** into **y** is as small as possible.

Error-Recovery Strategies in Parser

4. Global Correction

- Unfortunately, these methods are in general **too costly** to implement in terms of **time** and **space**, so these techniques are currently only of theoretical interest.
- **Note** that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of **least-cost correction** provides a yardstick for evaluating **error-recovery techniques**, and has been used for finding optimal replacement strings for **phrase-level recovery**.

Error Recovery in LR parsing

- An **LR parser** will detect an error when it consults the parsing action table and finds an error entry.
- Errors are never detected by consulting the goto table.
- An **LR parser** will announce an error as soon as there is no valid continuation for the portion of the input thus far scanned.
- A **canonical LR parser** will not make even a single reduction before announcing an error.

Error Recovery in LR parsing

- **SLR** and **LALR parsers** may make several reductions before announcing an error, but they will never shift an erroneous input symbol onto the stack.

*In **LR parsing**, we can implement **panic-mode error recovery** as follows.*

1. We scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. Zero or more input symbols are then discarded until a symbol **a** is found that can follow **A**.

Error Recovery in LR parsing

Panic-mode error recovery:

2. The parser then stacks the state **`GOTO(s, A)`** and resumes normal parsing. There might be more than one choice for the nonterminal **`A`**. Normally these would be nonterminals representing major program pieces, such as an expression, statement, or block. For example, if **`A`** is the nonterminal *stmt*, **`a`** might be **`semicolon`** or **`}`**, which marks the end of a statement sequence.

Error Recovery in LR parsing

Panic-mode error recovery:

3. This method of recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from **A** contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack.
4. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can follow **A**.

Error Recovery in LR parsing

Panic-mode error recovery:

5. By removing states from the stack, skipping over the input, and pushing **`GOTO(s, A)`** on the stack, the parser pretends that it has found an instance of **`A`** and resumes normal parsing.

Error Recovery in LR parsing

Phrase-level recovery:

1. Phrase-level recovery is implemented by examining each error entry in the **LR parsing table** and deciding on the basis of language usage the most likely programmer error that would give rise to that error.
2. An appropriate recovery procedure can then be constructed; evidently the top of the stack and/or first input symbols would be modified in a way deemed appropriate for each error entry.

Error Recovery in LR parsing

- In designing specific error-handling routines for an **LR parser**, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.
- The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols.

Error Recovery in LR parsing

- We must make our choices so that the **`LR parser`** will not get into an infinite loop.
- A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached.
- Popping a stack state that covers a nonterminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

Error Recovery in LR parsing

Example:

Consider again the expression grammar:

1. E → E + E
2. E → E * E
3. E → (E) | id

Error Recovery in LR parsing

Example:

LR Parsing table for grammar is shown below:

	ACTION						GOTO
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			acc	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		r_1	S_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

Error Recovery in LR parsing

Example:

The LR parsing table modified for error detection and recovery.

	ACTION						GOTO
	id	+	*	(	)	$	E
0	s_3	e_1	e_1	s_2	e_2	e_1	1
1	e_3	s_4	s_5	e_3	e_2	acc	
2	s_3	e_1	e_1	s_2	e_2	e_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	s_3	e_1	e_1	s_2	e_2	e_1	7
5	s_3	e_1	e_1	s_2	e_2	e_1	8
6	e_3	s_4	s_5	e_3	s_9	e_4	
7	r_1	r_1	s_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_3	r_3	r_3	r_3	r_3	r_3	

Error Recovery in LR parsing

Example: Error descriptions

e1: This routine is called from states **0**, **2**, **4** and **5**, all of which expect the beginning of an operand, either an **id** or a left parenthesis. Instead, **+**, *****, or the end of the input was found.

push state **3** (the goto of states **0**, **2**, **4** and **5** on **id**);

issue diagnostic "missing operand."

Error Recovery in LR parsing

Example: Error descriptions

e2: Called from states **0**, **1**, **2**, **4** and **5** on finding a right parenthesis.

remove the right parenthesis from the input;
issue diagnostic "unbalanced right parenthesis."

Error Recovery in LR parsing

Example: Error descriptions

e3: Called from states **1** or **6** when expecting an operator, and an **id** or right parenthesis is found.

push state **4** (corresponding to symbol **+**) onto the stack; issue diagnostic "missing operator."

Error Recovery in LR parsing

Example: Error descriptions

e4: Called from state **6** when the end of the input is found.

push state **9** (for a right parenthesis) onto the stack;
issue diagnostic "missing right parenthesis."

Semantic Analysis

Some of the actions performed semantic analysis phase are:

- **Type checking** i.e., number and type of arguments in function call and in function header of function definition must be same. Otherwise, it results in semantic error.
- **Object binding** i.e., associating variables with respective function definitions.
- **Automatic type conversion** of integers in mixed mode of operations.
- Helps in **intermediate code generation**.
- Display appropriate **error messages**.

Type Checking

- ***Type checking*** uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator. For example, the **&&** operator in **Java** expects its two operands to be booleans; the result is also of type boolean.
- To do ***type checking*** a compiler needs to assign a type expression to each component of the source program.
- The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language.
- ***Type checking*** has the potential for catching errors in programs.

Type Checking

- In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element.
- A ***sound*** type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs.
- An implementation of a language is ***strongly typed*** if a compiler guarantees that the programs it accepts will run without type errors.

Type Checking

- Besides their use for compiling, ideas from ***type checking*** have been used to improve the security of systems that allow software modules to be imported and executed.
- **Java** programs compile into machine-independent bytecodes that include detailed type information about the operations in the bytecodes. Imported code is checked before it is allowed to execute, to guard against both inadvertent errors and malicious misbehavior.

Compiler-Construction Tools

- The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as *language editors*, *debuggers*, *version managers*, *profilers*, *test harnesses*, and so on.
- In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.
- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms.

Compiler-Construction Tools

Some commonly used compiler-construction tools include:

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.
2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.
4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.
6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.