**Chapter 11**

**Compilers and Language Translation**

**A Guide to This Instructor’s Manual:**

We have designed this Instructor’s Manual to supplement and enhance your teaching experience through classroom activities and a cohesive chapter summary.

This document is organized chronologically, using the same headings that you see in the textbook. Under the headings, you will find: lecture notes that summarize the section, Teaching Tips, Class Discussion Topics, and Additional Projects and Resources. Pay special attention to teaching tips and activities geared toward quizzing your students and enhancing their critical thinking skills.

In addition to this Instructor’s Manual, our Instructor’s Resources also contain PowerPoint Presentations, Test Banks, and other supplements to aid in your teaching experience.

|  |
| --- |
| **At a Glance** |

**Instructor’s Manual Table of Contents**

* Overview
* Learning Objectives
* Teaching Tips and Quick Quizzes
* Class Discussion Topics
* Additional Projects
* Additional Resources
* Key Terms
* Solutions to End-of-Chapter Exercises

|  |
| --- |
| Lecture Notes |

**Overview**

Chapter 11 describes how computers translate programs from high-level programming languages into assembly language or byte code. It describes the phases of a compiler, including lexical analysis, parsing, code generation, and optimization. The chapter introduces Backus-Naur Form (BNF) for describing the grammatical structure of a programming language, and shows how to use it to parse programs into parse trees. It explains a set of common optimizations performed on compiled code.

# **Learning Objectives**

* List the phases of a typical compiler and describe the purpose of each phase
* Demonstrate how to break up a string of text into tokens
* Understand grammar rules written in BNF and use them to parse statements
* Explain how semantic analysis uses semantic records to determine meaning
* Show what a code generator does
* Explain the historical importance of code optimization, and why it is less central today
* Give an example of local code optimization and an example of global code optimization

# **Teaching Tips**

**11.1 Introduction**

1. Describe the importance of compilers, as they translate high-level languages into executable code. Compilers must be correct, efficient, and concise. Modern IDEs also provide tools for writing, analyzing, and debugging code.
2. Use the English to Spanish translation example in the book to help students understand the complexity of compilers.

**11.2 The Compilation Process**

1. The four phases of a compiler are lexical analysis, parsing, semantic analysis and code generation, and code optimization. Explain what each phase takes as input and what it produces.
2. Introduce the terms **lexical analysis**, **lexical analyzer**, and **scanner**. Explain what a **token** is. Work through examples to demonstrate the text that a scanner starts with, and the stream of tokens it produces. Explain how certain tokens have types that are important: symbols, numbers, and so on.
3. Introduce the terms **parse tree**, **parsing**, **parser**, and **syntax**. Emphasize that grammatical structure has to do with parts of speech, types of tokens, and not with meaning. Introduce the term **Backus-Naur Form (BNF)**, used to define a set of **rules** (or **productions)** as belonging to a **grammar**. Introduce the terms **terminal** and **nonterminal**, the **goal symbol**, and the **null string**. Emphasize the importance of **metasymbols** <, >, and := that are part of BNF, not the generated grammar.
4. Discuss the **language defined by a grammar** as a collection of all statements that can be successfully parsed.
5. Discuss **metasymbols** <, >, and ::= as part of BNF rules.
6. Demonstrate how BNF rules work using several simple examples. Ask students to determine which statements are grammatically using each grammar. Construct parse trees labeled with the rule applied to each parsing step. Introduce the term **look-ahead parsing algorithms** for algorithms that look beyond the current symbol to choose the right rule to apply.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Students will understand how grammars work best by trying them out. Include a class activity where students are given a grammar and build parse trees for statements. Also ask groups of students to develop new rules that extend a language, and then swap the rules among groups and test them. |

1. Discuss programming language constructs that allow repetitions, including arithmetic, nesting conditionals, and loops. Introduce the term **recursive definition**, for rules that handle such constructs. Discuss the problem of rules that are **ambiguous**.
2. Discuss the difference between syntax and semantics. Introduce the terms **semantic records** and **semantic analysis**, and note that many problems with semantics in programming languages stem from inappropriate types. Introduce the term **code generation**. Note that both phases can be done at the same time, or they could be done one after the other.
3. Work through examples of semantic analysis and code generation on parse trees from grammars seen earlier.
4. Go over some of the history of Fortran and the resistance that programmers felt to making use of it.
5. Emphasize that it was the work of the Fortran compilers that finally helped programmers realize the usefulness of the language.
6. Introduce the term **code optimization**, and explain its historical importance in producing efficient assembly code. Introduce the terms **visual development environments** and **online debuggers** to explain the modern complexities that improve the programmer’s workflow, rather than the code produced.
7. Talk about the usefulness of **code libraries**, which contain large collections of prewritten code that can be copied and pasted into a program.
8. Talk about the birth of **integrated development environments (IDE)** and how they help programmers write better, more refined code.
9. Introduce the term **local optimization** and examples of it: **constant evaluation**, **strength reduction**, and **eliminating unnecessary operations**. If possible, save examples of code produced by hand with code generation, and use them to illustrate the effect of local optimization.
10. Introduce the term **global optimization**, and give examples of the kind of analysis it requires.

**Quick Quiz 1**

1. The phase of compilation that checks if a program is grammatical is called \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Answer: parsing or syntactic analysis

1. (True or False) Lexical analysis must record a type for some tokens.

Answer: True

1. A \_\_\_\_\_\_\_ is ambiguous when there are two different parse trees for the same statement.

Answer: grammar

1. A BNF rule that captures repeating structures uses a \_\_\_\_\_\_\_\_\_\_\_\_\_.

Answer: recursive definitions

1. The phase of a compiler that outputs statements in assembly language is called \_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Answer: code generation

1. \_\_\_\_\_\_\_\_\_\_\_\_ Optimization occurs when the optimizer looks at only a few lines of assembly code at a time.

Answer: Local

1. Modern compiler designers also must worry about providing additional tools to programmers, including \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Answer: visual development environments, online debuggers, or other similar answers

1. When a compiler is embedded within a collection of supporting software development routines such as debuggers, editors, toolkits, and libraries, it is called a(n) \_\_\_\_\_\_\_.

Answer: integrated development environment (IDE)

**11.3 Conclusion**

1. Real-world compiler design is much more complex than seen here.
2. High-level languages require elaborate systems to translate their programs into efficient, concise, correct assembly language programs.
3. Other topics touched on that are important for programming:
4. Integrated development environments (IDEs) and support tools
5. Compilers for alternative languages, such as functional, object-oriented, and parallel languages
6. Language standardization
7. Top-down versus bottom-up parsing algorithms
8. Error detection and recovery

# **Class Discussion Topics**

1. Break a given English phrase into tokens, assigning part-of-speech types to each token as needed. Think about where punctuation gets complicated: Are there punctuation marks that could mean different things?
2. Which phase of the compiler seems easiest to you? Which seems most difficult? Why?
3. Discuss the advent and troubles behind **natural language understanding** and how it’s changing the face of computing.

# **Additional Projects**

1. Given a grammar by your instructor, determine what tokens would be required by that grammar. List them, along with any token types it would need.
2. In a group of two to three students, develop new BNF rules to describe a “repeat-until” loop. This form starts with the keyword “repeat”, and then has a block of statements, and then ends with the word “until” and a Boolean expression. Assume that you have grammar rules for statement blocks and Boolean expressions.
3. Given some assembly code by your instructor, find opportunities for local optimization, using the three methods described in the book. Note that making one optimization may create an opportunity to apply another optimization. How short can you make the code?
4. Research some integrated development environments and pick a couple to write a basic piece of code in. How do they differ?

# **Additional Resources**

1. A general tutorial on BNF and EBNF: <http://www.garshol.priv.no/download/text/bnf.html>
2. A page that demonstrates parsing algorithms: <http://ag-kastens.upb.de/lehre/material/uebi/parsdemo/>
3. A summary list of groups doing research in compilers: <http://www.cs.cmu.edu/~mleone/language/projects.html>

**Key Terms**

* **Ambiguous (grammar)**: A grammar that allows the construction of two or more distinct parse trees for the same statement.
* **BNF (Backus-Naur Form)**: The most widely used notation for representing the syntax of a programming language.
* **Code generation**: A second pass over the parse tree, not to determine correctness but to produce the translated code.
* **Code libraries**: Large collections of prewritten and fully debugged program units.
* **Code optimization**: Polishing and fine-tuning code so that it runs a little faster or occupies a little less memory.
* **Constant evaluation**: Arithmetic expressions are fully evaluated at compile time, if possible, rather than at execution time.
* **Eliminating unnecessary operations**: Instructions that are correct, but not necessary, are discarded.
* **Global optimization**: The compiler looks at large segments of the program, not just small pieces, to determine how to improve performance.
* **Goal symbol**: In parsing, the final nonterminal; the nonterminal object that the parser is trying to produce as it builds the parse tree.
* **Grammar**: The collection of all rules that define the syntax of a programming language.
* **Integrated development environment (IDE)**: A compiler embedded within a collection of supporting software development routines.
* **Language defined by a grammar**: A collection of all statements that can successfully parsed.
* **Lexical analysis**: The process of grouping individual characters into grammatical units called tokens.
* **Lexical analyzer/scanner**: A program that performs lexical analysis.
* **Local optimization**: The compiler looks at a very small block of instructions, typically from one to five, and tries to determine how it can improve the efficiency of this local code block without regard for what instructions come before or after.
* **Look-ahead parsing algorithm**: A parsing algorithm that examines the upcoming tokens to see what would happen if a certain choice is made. It does this to try to avoid making incorrect choices.
* **Metasymbol**: A symbol of one language, such as BNF, that is used to describe the characteristics of another language.
* **Natural language understanding**: Getting computers to understand and use natural language.
* **Nonterminal**: In parsing, not an actual element of the language but an intermediate grammatical category used to help explain and organize the language.
* **Null string**: A string that is empty or contains nothing.
* **Online debuggers**: Web-based tools to help programmers locate and correct errors.
* **Parse tree**: A structure that starts from the individual tokens in a statement and shows how these tokens can be grouped into predefined grammatical categories.
* **Parser**: A program that parses high-level language statements.
* **Parsing**: The process of diagramming a high-level language statement.
* **Production**: Another term for rule.
* **Recursive definition**: The definition of a grammatical element in terms of itself.
* **Rule**: A description of how to group syntactic elements in a programming language to produce a new grammatical construct.
* **Scanner**: Another term for a lexical analyzer.
* **Semantic analysis**: A pass over the parse tree to determine whether all branches of the tree are semantically valid.
* **Semantic record**: A data structure that stores information about a nonterminal, such as the actual name of the object and its data type.
* **Strength reduction**: Slow arithmetic operations are replaced with faster ones.
* **Terminal**: In parsing, an actual token of the language recognized and returned by a scanner.
* **Token**: Syntactical unit that is treated as a single, indivisible entity for the purposes of translation.
* **Visual development environment**: Software development tools that use graphics and video to let the programmer see what is happening.

**Solutions to End-of-Chapter Exercises**

**1**. a. ‘if’, ‘(‘, ‘a’, ‘==’, ‘b1’, ‘)’, ‘a’, ‘=’, ‘x’, ‘+’, ‘y’, ‘;’  
 b. ‘delta’, ‘=’, ‘epsilon’, ‘+’, ‘1.23’, ‘-‘, ‘sqrt’, ‘(‘, ‘zz’, ‘)’, ‘;’  
 c. ‘print’, ’(‘,’Q’,’)’,’;’

**2.** Because an underscore character can appear in a variable name and there are no spaces in the string, the scanner will classify the string as a single five-character token.

**3**. Because both { } and (\* \*) can be used to enclose comments, it is likely that the scanner would group both { and (\* into a "begin comment" classification, and both } and \*) into an "end comment" classification. However, that would allow a comment to begin with, say, a { but end with a \*). To ensure that you use matching symbols, it might instead assign each of the four symbols its own classification number.

**4**. a. limit = begin + end  
 *Token Classification* limit 1  
 = 3  
 begin 1  
 + 4  
 end 1  
 b. a = b – 1;  
 *Token Classification* a 1  
 = 3  
 b 1  
 - 5  
 1 2  
 ; 6  
 c. if (c == 50) x = 1; else y = x + 44;  
 *Token Classification* if 8  
 ( 10  
 c 1  
 == 7  
 50 2  
 ) 11  
 x 1  
 = 3  
 1 2  
 ; 6  
 else 9  
 y 1  
 = 3  
 x 1  
 + 4  
 44 2  
 ; 6  
d. thenelse == error -  
 *Token Classification* thenelse 1  
 == 7  
 error 1  
 - 5

**5**. a. <number> ::= + <nzdigit> <digit> | <nzdigit> <digit>  
 <digit> ::= 0|<nzdigit>   
 <nzdigit> ::= 1|2|3|4|5|6|7|8|9  
   
 b. + 9 0  
 |\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_ |  
 | | |  
 | <nzdigit> <digit>  
 \ \_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_/  
 |   
 <number>

**6**. a. <phonenumber> ::= ( <d> <d> <d> ) <d> <d> <d> - <d> <d> <d> <d>  
 | <d> <d> <d> - <d> <d> <d> <d>  
 <d> ::= 1|2|3|4|5|6|7|8|9|0  
 b. <phonenumber> ::= ( <nonzeroone> <zeroone> <d> ) <nonzeroone> <d> <d>   
 - <d> <d> <d> <d> | <nonzeroone> <d> <d> - <d> <d> <d> <d>  
 <d> ::= <zeroone>|<nonzeroone>  
 <nonzeroone> ::= 2|3|4|5|6|7|8|9   
 <zeroone> ::= 1|0  
 c. ( 6 1 2 ) 5 5 5 - 1 2 1 2

<nonzeroone> <zeroone> <d> <nonzeroone> <d> <d> <d> <d> <d> <d>

<phonenumber>

**7**. a. <identifier> ::= <letter> <identifierbody> | <letter>  
 <identifierbody> ::= <symbol> <identifierbody> | <symbol>  
 <symbol>::=<letter>|<number>  
 <letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B…|Z  
 <number> ::= 0|1|2|3|4|5|6|7|8|9  
  
 b. A B 5 C 8  
 | | | \ \  
 <letter> <letter> <number> <letter> <number>  
 | | | | |  
 \ <symbol> <symbol> <symbol> <symbol>  
 \ | | | |  
 \ | | | <identifierbody>  
 \ | | | /  
 \ | | <identifierbody>   
 | | | /  
 | | | /  
 | | <identifierbody>   
 | | /  
 | | /  
 | <identifierbody>  
 | /  
 | /  
 <identifier>

**8**. a. <money> ::= $ <number> | $ <number> . <number> | $ <number> CR   
 | $ <number> . <number> CR  
 <number> :: = <digit> | <digit> <number>  
 <digit> ::= 0|1|2|3|4|5|6|7|8|9

b. <money> ::= $ <number> | $ <number> . <digit> <digit> | $ <number> CR  
 | $ <number> . <digit> <digit> CR  
 <number> :: = <digit> | <digit> <number>  
 <digit> ::= 0|1|2|3|4|5|6|7|8|9

c. $ 1 9 . 9 5 CR  
 | | | | | | |  
 | | | | | \ |  
 | <digit> <digit> | <digit> <digit> |  
 | | | | | | |   
 | | <number> | | <number> |  
 | | / | | / |  
 | <number> | <number> /  
 \ \ | / /   
 \\_\_\_\_\_\_\\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_/   
 |  
 <money>

**9**. The language consists of A and A,A

**10**. This language consists of A and A,A and AA,A and AAA,A and so forth. The <next> ::= <letter><next>  
 rule allows arbitrarily long strings of A 's before the comma and the final A.

**11**. a. <Boolean> ::= <var> AND <var> | <var> OR <var>  
 <var> ::= w|x|y|z

b. <Boolean> ::= <expr> AND <expr> | <expr> OR <expr>  
 <expr> ::= <var> | ( <var> == <var> ) | ( <var> < <var> ) | ( <var> > <var> )  
 <var> ::= w|x|y|z

c. <Booleanexpr> ::= <Boolean> | <Boolean> AND <Booleanexpr>   
 | <Boolean> OR <Booleanexpr>  
 <Boolean> ::= <expr> AND <expr> | <expr> OR <expr>  
 <expr> ::= <var> | ( <var> == <var> ) | ( <var> < <var> ) | ( <var> > <var> )  
 <var> ::= w|x|y|z

**12**. y = x + y + y + z  
 | | | \ | | | | |   
 | \ \ \ | | | | |   
 | | \ | | | / | |  
<variable> \ <variable> | <variable> | <variable> | <variable>  
 \ / | | | | | / |  
 | | | \ | | | / /  
 | | <expression> \ | | | / /  
 | | \ | | | / / /  
 | | \ | / / / / /  
 | | <expression> / / \_\_\_ / /  
 | | | \_ / / / /  
 \ \ | / / / /  
 \ \ <expression> \_\_/ /  
 \ \ \ / /  
 \ \ <expression> -----------/   
 \ \ /  
 <assignment statement>  
  
 This parse tree is unique. The operations must be done from left to right.

**13**. This is the set of all strings of 1 or more binary digits. This is the set of all possible binary numbers.

**14**. <string> ::= Λ | <letter> <string>  
 <letter> ::= a | b | c

**15**. The two different interpretations of this ambiguous sentence are that the shirt was too large, or that the store was too large.

**16**. <complexinput> ::= input ( <vars> )   
 <vars> ::= <var> | <var>, <vars>  
 <var> ::= <letter> | <letter> <char>  
 <char> ::= <letter> | <digit> | <letter> <char> | <digit> <char>  
 <letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|…|Z  
 <digit> ::= 0|1|2|3|4|5|6|7|8|9

**17**. Other information one might want to store in a semantic record, in addition to name and data type, would be whether the quantity is a constant, or, in the case of an array, the size of the array. Such information would come from the variable declaration**.**

**18**. This production does not generate any code because if it did, its effect would be to create storage for the variable x, that is, write a .DATA statement, but this was already done in the occurrence of x on the right side of the assignment operator.

**19**. The compiler would discover that it needs to do a data conversion when it tries to produce code to execute the assignment operation in y = x. At that point it would detect that the semantic record for x was type integer and the semantic record for y was type double. The binary representation of x as an integer would need to be converted to the binary representation of x as a real number, probably using some predefined assembly language code.

**20.** The concept of algebraic identities could be exploited during the code optimization phase of compilation by replacing an expression with an equivalent expression requiring fewer operations. For example, X + 0 could be replaced by X, reducing the number of operations from one to zero. This would be local optimization. Other examples would be

X \* 1 = X (reduce operations from one to zero)

X \* Y + X \* Z = X\*(Y + Z) (reduce operations from three to two)

**21**. The expression (a + b + c \* 3) appears in both statements. The compiler could compute this value once, save it, and use it in both statements instead of having to compute it twice. These two statements might be separated by many other statements, so the compiler has to take a larger view of the code. This would be a global optimization.

**22**. If all mathematical operations take 5 nsec, then the optimization done in problem 21, which saves 3 operations, saves 15 nsec. This is not much time at all. Again, the choice of a more efficient algorithm, if available, will far outstrip the small improvements achieved by compiler optimization.

**23**. The compiler first checks the type fields to be sure that the two variables being compared are the same type. Then it creates a semantic record linked to <Boolean expression> that is named *temp* and that has a Boolean data type. The machine language code generated could be:

LOAD a  
 COMPARE b

JUMPEQ EQUAL

LOAD FALSE

JUMP DONE

EQUAL: LOAD TRUE

DONE: STORE TEMP

.

.

.

A: .DATA 0

B: .DATA 0

TRUE: .DATA 1

FALSE: .DATA 0

TEMP: .DATA 0

**Challenge Work**

**1 and 2.** Try to provide students with leads to documentation of LEX and YACC that are written at a level for novice readers.