COURSE GUIDE

CIT 316
COMPILER CONSTRUCTION I (PRINCIPLES AND TECHNIQUES OF COMPILER)

Course Team	A.A. Afolorunso (Course Developer/Writer) – NOUN
	Dr. Oludele Awodele (Course Editor) - Babcock University, Ilisan-Remo
	Prof. Kehinde Obidairo (Programme Leader) – NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria Headquarters
Nnamdi Azikiwe Expressway
Airport Road
Jabi, Abuja

e-mail: centralinfo@nou.edu.ng URL: www
.nou.edu.ng

Published by:
National Open University of Nigeria

Printed 2013, 2022

ISBN: 978-058-470-6

CONTENTS **PAGE**

INTRODUCTION

CIT 316 – Complier Construction I is a three (3) credit unit course of 17 units . With the increasing diversity and complexity of computers and their applications , the development of efficient , reliable software has become increasingly dependent on automatic support from compilers and other programme analysis and translation tools . This course covers principal topics in understanding and transforming programmes at the code block, function, programme, and behaviour levels. Specific techniques for imperative languages include data flow , dependence , inter -procedural , and profiling analyses, resource allocation, and multi-grained parallelism on both CPUs and GPUs

It is a course for B. Sc. Computer science major students, and is normally taken in a student's fourth year. It should appeal to anyone who is interested in the design and implementation of programming languages. Anyone who does a substantial amount of programming should find the material valuable.

This course is divided into four modules. The first module deals with the review of grammars, languages and automata, introduction to compiler.

The second module treats, extensively, lexical analysis. Under which such concepts as the scanner, lexical analyser generation were discussed.

The third module deals with syntax analysis and discusses context-free grammars, LL (k), LR (k), operator-precedence grammars, etc. Also, implementation of various parsing techniques was discussed.

The fourth module which is the concluding module of the course discusses code generation issues such as symbol tables, intermediate representation, code optimisation techniques and code generation.

This Course Guide gives you a brief overview of the course contents, course duration, and course materials.

WHAT YOU WILL LEARN IN THIS COURSE

The main purpose of this course is to acquaint students with software tools and techniques which are applicable both to compilers and the implementation of system utility routines, command interpreters, etc.Thus; we intend to achieve this through the following

COURSE AIMS

First, students will learn the key techniques in modern compiler construction, getting prepared for industry demands for compiler engineers.

Second, students will understand the rationale of various programme analysis and optimisation techniques, able to improve their programming skills accordingly.

The third goal is to build the foundation for students to pursue the research in the areas of compiler, programme analysis, programme modelling, and operating systems

COURSE OBJECTIVES

Certain objectives have been set out to ensure that the course achieves its aims. Apart from the course objectives, every unit of this course has set objectives. In the course of the study, you will need to confirm, at the end of each unit, if you have met the objectives set at the beginning of each unit. Upon completing this course you should be able to:

- recognise various classes of grammars, languages, and automata, and employ these to solve common software problems
- explain the major steps involved in compiling a high-level programming language down to a low-level target machine language
- construct and use the major components of a modern compiler;
- work together effectively in teams on a substantial software implementation project.

Related Courses
Prerequisites: CIT 342; Computer Science students only

WORKING THROUGH THIS COURSE

In order to have a thorough understanding of the course units, you will need to read and understand the contents, practise the steps by designing a compiler of your own for a known language, and be committed to learning and implementing your knowledge.

This course is designed to cover approximately seventeen weeks, and it will require your devoted attention. You should do the exercises in the Tutor-Marked Assignments and submit to your tutors.

COURSE MATERIALS

These include:

1. Course Guide
2. Study Units
3. Recommended Texts
4. A file for your assignments and for records to monitor your progress.

STUDY UNITS

There are 17 study units in this course:

Module 1 Introduction to Compilers

Unit 1	Review of Grammars, Languages and Automata
Unit 2	What is a Compiler?
Unit 3	The Structure of a Compiler

Module 2 Lexical Analysis

Unit 1	The Scanner
Unit 2	Hand Implementation of Lexical Analyser
Unit 3	Automatic Generation of Lexical Analyser
Unit 4	Implementing a Lexical Analyser

Module 3 Syntax Analysis

Unit 1	Context-Free Grammars
Unit 2	Bottom-Up Parsing Techniques
Unit 3	Precedence Parsing
Unit 4	Top-Down Parsing Techniques
Unit 5	LR Parsers

Module 4 Code Generation

Unit 1	Error Handling
Unit 2	Symbol Tables
Unit 3	Intermediate Code Generation
Unit 4	Code Generation
Unit 5	Code Optimisation

Make use of the course materials, do the exercises to enhance your learning.

TEXTBOOKS AND REFERENCES

Aho, Alfred & Sethi, Ravi & Ullman, Jeffrey. *Compilers: Principles, Techniques, and Tools* ISBN 0201100886 The Classic Dragon book.

Appel, *A., Modern Compiler Implementation in Java*, 2nd ed., Cambridge University Press, 2002.

Appel, Andrew *Modern Compiler Implementation in C/Java/ML* (respectively ISBN 0-521-58390-X,ISBN 0-521-58388-8, ISBN 0-521-58274-1) is a set of cleanly written texts on compiler design, studied from various different methodological perspectives.

Brown, P.J. *Writing Interactive Compilers and Interpreters* ISBN 047127609X Useful practical advice, not much theory.

Fischer, Charles & LeBlanc, Richard. *Crafting A Compiler* ISBN 0805332014 Uses an ADA like pseudo-code.

Fischer, LeBlanc, Cytron, *Crafting a Compiler Implementation*, Addison-Wesley

Holub, Allen *Compiler Design in C* ISBN 0131550454 Extensive examples in "C".

Hunter, R. *The Design and Construction of Compilers* ISBN 0471280542 several chapters on theory of syntax analysis, plus discussion of parsing difficulties caused by features of various source languages.

Keith, D. Cooper & Linda Torczon, "Engineering a Compiler", Morgan Kaufmann Publishers, 2004

Pemberton, S. & Daniels, M.C. *Pascal Implementation. The P4 Compiler* ISBN 0853123586 (Discussion) and ISBN 085312437X (Compiler listing) Complete listing and readable commentary for a Pascal compiler written in Pascal.

Randy Allen and Ken Kennedy, "Optimising Compilers for Modern Architectures", Morgan Kaufmann Publishers, 2001.

Weinberg, G.M. *The Psychology of Computer Programming: Silver Anniversary Edition* ISBN 0932633420 Interesting insights and anecdotes.

Wirth, Niklaus *Compiler Construction* ISBN 0201403536 From the inventor of Pascal, Modula-2 and Oberon-2, examples in Oberon.

ASSIGNMENTS FILE

These are of two types: the self-assessment exercises and the Tutor-Marked Assignments. The self-assessment exercises will enable you monitor your performance by yourself, while the Tutor-Marked Assignment is a supervised assignment. The assignments take a certain percentage of your total score in this course. The Tutor-Marked Assignments will be assessed by your tutor within a specified period. The examination at the end of this course will aim at determining the level of mastery of the subject matter. This course includes twelve Tutor-Marked Assignments and each must be done and submitted accordingly. Your best scores however, will be recorded for you. Be sure to send these assignments to your tutor before the deadline to avoid loss of marks.

PRESENTATION SCHEDULE

The Presentation Schedule included in your course materials gives you the important dates for the completion of tutor marked assignments and attending tutorials. Remember, you are required to submit all your assignments by the due date. You should guard against lagging behind in your work.

ASSESSMENT

There are two aspects to the assessment of the course. First are the tutor marked assignments; second, is a written examination.

In tackling the assignments, you are expected to apply information and knowledge acquired during this course. The assignments must be submitted to your tutor for formal assessment in accordance with the deadlines stated in the Assignment File. The work you submit to your tutor for assessment will count for 30% of your total course mark.

At the end of the course, you will need to sit for a final three-hour examination. This will also count for 70% of your total course mark.

TUTOR MARKED ASSIGNMENTS (TMAS)

There are twenty-two tutor marked assignments in this course. You need to submit all the assignments. The total marks for the best three (3) assignments will be 30% of your total course mark.

Assignment questions for the units in this course are contained in the Assignment File. You should be able to complete your assignments from the information and materials contained in your set textbooks, reading and study units. However, you may wish to use other references to broaden your viewpoint and provide a deeper understanding of the subject.

When you have completed each assignment, send it together with form to your tutor. Make sure that each assignment reaches your tutor on or before the deadline given. If, however, you cannot complete your work on time, contact your tutor before the assignment is done to discuss the possibility of an extension.

EXAMINATION AND GRADING

The final examination for the course will carry 70% percentage of the total marks available for this course. The examination will cover every aspect of the course, so you are advised to revise all your corrected assignments before the examination.

This course endows you with the status of a teacher and that of a learner. This means that you teach yourself and that you learn, as your learning capabilities would allow. It also means that you are in a better position to determine and to ascertain the what, the how, and the when of your language learning. No teacher imposes any method of learning on you.

The course units are similarly designed with the introduction following the contents, then a set of objectives and then the dialogue and so on.

The objectives guide you as you go through the units to ascertain your knowledge of the required terms and expressions.

COURSE MARKING SCHEME

This table shows how the actual course marking is broken down.

Assessment	**Marks**
Assignment 1- 4	Four assignments, best three marks of the four count at 30% of course marks
Final Examination	70% of overall course marks
Total	100% of course marks

COURSE OVERVIEW

Unit	**Title of Work**	**Weeks Activity**	**Assessment (End of Unit)**
	Course Guide	Week 1	
	Module 1 Introduction to Compilers		
1	Unit 1 Review of Grammars, Languages and Automata	Week 1	Assignment 1
2	Unit 2 What is a Compiler?	Week 2	Assignment 2
3	Unit 3 The Structure of a Compiler	Week 2	Assignment 3
	Module 2 Lexical Analysis		
1	Unit 1 The Scanner	Week 3	Assignment 5
2	Unit 2 Hand Implementation of Lexical Analyser	Week 3	Assignment 6
3	Unit 3 Automatic Generation of Lexical Analyser	Week 4	Assignment 7
4	Unit 4 Implementing a Lexical Analyser	Week 4	
	Module 3 Syntax Analysis		
1	Unit 1 Context-Free Grammars	Week 5	Assignment 8
2	Unit 2 Bottom-Up Parsing Techniques	Week 6	Assignment 9
3	Unit 3 Precedence Parsing	Week 7 – 8	Assignment 10
4	Unit 4 Top-Down Parsing Techniques	Week 8 – 9	Assignment 11
5	Unit 5 LR Parsers	Week 10 - 11	Assignment 12
	Module 4 Code Generation		
1	Unit 1 Error Handling	Week 12	Assignment 13
2	Unit 2 Symbol Tables	Week 13	Assignment 14
3	Unit 3 Intermediate Code Generation	Week 14	Assignment 15
4	Unit 4 Code Generation	Week 15	Assignment 16
5	**Unit 5 Code Optimisation**	Week 16	Assignment 17
	Revision	Week 16	
	Examination	Week 17	
Total		17 weeks	

HOW TO GET THE BEST FROM THIS COURSE

In distance learning the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

Remember that your tutor's job is to assist you. When you need help, don't hesitate to call and ask your tutor to provide it.

1. Read this Course Guide thoroughly.
2. Organise a study schedule. Refer to the 'Course Overview' for more details. Note the time you are expected to spend on each unit and how the assignments relate to the units. Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they lag behind in their course work.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.
6. Work through the unit. The content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit you will be instructed to read sections from your set books or other articles. Use the unit to guide your reading.

7. Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.
8. When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
9. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have any questions or problems.
10. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this Course Guide).

TUTORS AND TUTORIALS

There are 12 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, together with the name and phone number of your tutor, as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail or submit your tutor-marked assignments to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, or e-mail if you need help. The following might be circumstances in which you would find help necessary. Contact your tutor if:

- you do not understand any part of the study units or the assigned readings
- you have difficulty with the self-tests or exercises
- you have a question or problem with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have face to face contact with your tutor and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussions actively.

SUMMARY

Compiler Construction I will introduce you to the concepts associated with programming , programming languages and the compilation process . The content of the course material was planned and written to ensure that you acquire the proper knowledge and skills for the appropriate situations. Real-life situations have been created to enable you identify with and create some of your own. The essence is to help you in acquiring the necessary knowledge and competence by equipping you with the necessary tools to accomplish this.

We hope that by the end of this course you would have acquired the required knowledge to view compilers, programming languages and programming environments in a new way.

We wish you success with the course and hope that you will find it both interesting and useful.

MAIN COURSE

CONTENTS **PAGE**

MODULE 1 INTRODUCTION TO COMPILERS

UNIT 1 REVIEW OF GRAMMARS, LANGUAGES AND AUTOMATA

CONTENTS

1.0 INTRODUCTION

As you learnt in CIT 342: Formal Languages and Automata Theory, in the field of Computer Science, there are different types of grammars on which different languages are defined. For each of these grammars, there is a class of automata that can parse/recognise strings form from the grammar. The set of all strings that can be generated from the grammar constitutes the language of the grammar.

In this unit, you will be taken through some of the things you learnt previously on formal grammar, formal language and automata.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define formal grammar
- define alphabet, words and strings
- state the types of formal grammars we have in the field of Computer Science
- describe the class of automata that can recognise strings generated by each grammar
- identify strings that are generated by a particular grammar
- describe the Chomsky hierarchy
- explain the relevance of formal grammar and language to computer programming.

3.0 MAIN CONTENT

3.1 Formal Grammar

A **formal grammar** (sometimes called a **grammar**) is a set of rules of a specific kind, for forming strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar describes only the form of the strings and not the meaning or what can be done with them in any context.

A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting must start. Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recogniser". Recogniser is a function in computing that determines whether a given string belongs to the language or is grammatically incorrect. To describe such recognisers, formal language theory uses separate formalisms, known as ***Automata Theory***.

The process of recognising an utterance (a string in natural languages) by breaking it down to a set of symbols and analysing each one against the grammar of the language is referred to as **Parsing**. Most languages have the meanings of their utterances structured according to their syntax – a practice known as compositional semantics. As a result, the first step to describing the meaning of an utterance in language is to break it down into parts and look at its analysed form (known as its parse tree in Computer Science).

A grammar mainly consists of a set of rules for transforming strings. (If it consists of these rules only, it would be a semi-Thue system). To generate a string in the language, one begins with a string consisting of a single *start symbol*. The *production rules* are then applied in any order, until a string that contains neither the start symbol nor designated *non-terminal symbols* is produced. The language formed by the grammar consists of all distinct strings that can be generated in this manner. Any particular sequence of production rules on the start symbol yields a distinct string in the language. If there are multiple ways of generating the same single string, the grammar is said to be ambiguous.

Example 3.1

Assuming the alphabet consists of *a* and *b*, the start symbol is *S* and we have the following production rules:

1. $S \rightarrow aSb$
2. $S \rightarrow ba$

then we start with *S*, and can choose a rule to apply to it. If we choose rule 1, we obtain the string *aSb*. If we choose rule 1 again, we replace *S* with *aSb* and obtain the string *aaSbb*. If we now choose rule 2, we replace *S* with *ba* and obtain the string *aababb*, and are done. We can write this series of choices more briefly, using symbols: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aababb$. The language of the grammar is then the infinite set $\{a^n bab^n | n \geq 0\} = \{ba, abab, aababb, aaababbb, ...\}$, where a^k is *a* repeated *k* times (and *n* in particular represents the number of times production rule 1 has been applied).

3.1.1 The Syntax of Grammars

In the classic formalisation of generative grammars first proposed by Noam Chomsky in the 1950s, a grammar *G* consists of the following components:

- a finite set *N* of *non-terminal symbols*, none of which appear in strings formed from *G*.
- a finite set Σ of *terminal symbols* that is disjoint from *N*.
- a finite set *P* of *production rules*, each rule of the form

$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$

- where * is the Kleene star operator and $\cup$ denotes set union. That is, each production rule maps from one string of symbols to another, where the first string (the "head") contains at least one non-terminal symbol. In the case that the second string (the "body") consists solely of the empty string – i.e. it contains no symbols at all, it may be denoted with a special notation (often Λ, *e* or ε) in order to avoid confusion.

A distinguished symbol $S \in N$, that is, the *start symbol*.
A grammar is formally defined as the tuple (*N*, Σ, *P*, *S*). Such a formal grammar is often called a **rewriting system** or a **phrase structure grammar** in the literature.

3.1.2 The Semantics of Grammars

The operation of a grammar can be defined in terms of relations on strings:

- given a grammar *G* = (*N*, Σ, *P*, *S*), the binary relation $\Rightarrow_G$ (pronounced as "G derives in one step") on strings in $(\Sigma \cup N)^*$ is defined by:
- $x \Rightarrow_G y \text{ iff } \exists u, v, p, q \in (\Sigma \cup N)^* : x = upv \wedge y = uqv \wedge p \rightarrow q \in P$
- the relation $\Rightarrow_G^*$ (pronounced as *G derives in zero or more steps*) is defined as the reflexive transitive closure of $\Rightarrow_G$
- a *sentential form* is a member of $(\Sigma \cup N)^*$ that can be derived in a finite number of steps from the start symbol *S*; that is, a sentential form is a member of $\{w \in (\Sigma \cup N)^* \mid S \Rightarrow_G{}^* w\}$. A sentential form that contains no non-terminal symbols (i.e. is a member of Σ^*) is called a *sentence*.
- the *language* of *G*, denoted as $\boldsymbol{L}(G)$, is defined as all those sentences that can be derived in a finite number of steps from the start symbol *S*, that is, the set $\{w \in \Sigma^* \mid S \Rightarrow_G{}^* w\}$.

Note that the grammar *G* = (*N*, Σ, *P*, *S*) is effectively the semi-Thue system $(N \cup \Sigma, P)$, rewriting strings in exactly the same way; the only difference is that, we distinguish specific *non-terminal* symbols which must be rewritten in rewrite rules, and are only interested in rewritings from the designated start symbol *S* to strings without non-terminal symbols.

Example 3.2

Note that for these examples, formal languages are specified using set-builder notation.

Consider the grammar *G* where $N = \{S, B\}$,$\Sigma = \{a, b, c\}$, *S* is the start symbol, and *P* consists of the following production rules:

1. $S \rightarrow aBSc$
2. $S \rightarrow abc$
3. $Ba \rightarrow aB$
4. $Bb \rightarrow bb$

This grammar defines the language $L(G) = \{a^n b^n c^n | n \geq 1\}$where a^n denotes a string of *n* consecutive *a*'s. Thus, the language is the set of strings that consist of one or more *a*'s, followed by the same number of *b*'s, and then by the same number of *c*'s.

Some examples of the derivation of strings in *L*(*G*) are:

$$\mathbf{S} \Rightarrow_2 \mathbf{abc}$$
$$\mathbf{S} \Rightarrow_1 \mathbf{aBSc} \Rightarrow_2 aB\mathbf{abc}c \Rightarrow_3 a\mathbf{aB}bcc \Rightarrow_4 aa\mathbf{bb}cc$$
$$\mathbf{S} \Rightarrow_1 \mathbf{aBSc} \Rightarrow_1 aB\mathbf{aBSc}c \Rightarrow_2 aBaB\mathbf{abc}cc \Rightarrow_3 a\mathbf{aB}Babccc \Rightarrow_3 aaB\mathbf{aB}bccc$$
$$\Rightarrow_3 aa\mathbf{aB}Bbccc \Rightarrow_4 aaaB\mathbf{bb}ccc \Rightarrow_4 aaa\mathbf{bb}bccc$$

(Note on notation: $P \Rightarrow_i Q$reads "String *P* generates string *Q* by means of production *i*", and the generated part is each time indicated in bold type.)

3.2 Types of Grammars and Automata

In the field of Computer Science, there are four basic types of grammars:

a. Type-0 grammars (unrestricted grammars) include all formal grammars.
b. Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages.
c. Type-2 grammars (context-free grammars) generate the context-free languages.
d. Type-3 grammars (regular grammars) generate the regular languages.

The difference between these types is that they have increasingly strict production rules and can express fewer formal languages. Two important types are *context-free grammars* (Type 2) and *regular grammars* (Type 3). The languages that can be described with such a grammar are called *context-free languages* and *regular languages*, respectively. Although much less powerful than unrestricted grammars (Type 0), which can in fact express any language that can be accepted by a Turing machine, these two restricted types of grammars are most often used because parsers for them can be efficiently implemented

Recently, there have been other types of classification of grammars such as analytical grammars identified.

3.2.1 Type-0: Unrestricted Grammars

These grammars generate exactly all languages that can be recognised by a Turing machine. These languages are also known as the *Recursively Enumerable Languages*. Note that this is different from the recursive languages which can be *decided* by an always-halting Turing machine.

3.2.2 Type-1: Context-Sensitive Grammars

These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$with *A* being a non-terminal and α, β and γ strings of terminals and non-terminals. The strings α and β may be empty, but γ must be non-empty. The rule $S \rightarrow \epsilon$is allowed if *S* does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognied by a linear bounded automaton (a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.)

3.2.3 Type2: Context-Free Grammars

A *context-free grammar* is a grammar in which the left-hand side of each production rule consists of only a single non-terminal symbol. This restriction is non-trivial; not all languages can be generated by context-free grammars. Those that can are called *context-free languages*.

The language defined above is not a context-free language, and this can be strictly proven using the pumping lemma for context-free languages, but for example the language $\{a^n b^n | n \geq 1\}$(at least 1 *a* followed by the same number of *b*'s) is context-free, as it can be defined by the grammar G_2 with$N = \{S\}$, $\Sigma = \{a, b\}$, *S* the start symbol, and the following production rules:

1. $S \rightarrow aSb$
2. $S \rightarrow ab$

A context-free language can be recognised in *O*(n^3) time by an algorithm such as Earley's algorithm. That is, for every context-free language, a machine can be built that takes a string as input and determines in *O* (n^3) time whether the string is a member of the language, where *n* is the length of the string. Further, some important subsets of the context-free languages can be recognised in linear time using other algorithms.

These are exactly all languages that can be recognised by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

3.2.4 Type-3: Regular Grammars

In regular grammars, the left hand side is again only a single non-terminal symbol, but now the right-hand side is also restricted. The right side may be the empty string, or a single terminal symbol, or a single terminal symbol followed by a non-terminal symbol, but nothing else. (Sometimes a broader definition is used: one can allow longer strings of terminals or single non-terminals without anything else, making languages easier to denote while still defining the same class of languages.)

The language defined above is not regular, but the language $\{a^n b^m \mid m, n \geq 1\}$(at least 1 *a* followed by at least 1 *b*, where the numbers may be different) is, as it can be defined by the grammar G_3 with $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, *S* the start symbol, and the following production rules:

$S \rightarrow aA$
$A \rightarrow aA$
$A \rightarrow bB$
$B \rightarrow bB$
$B \rightarrow \epsilon$

All languages generated by a regular grammar can be recognised in linear time by a finite state machine. Although, in practice, regular grammars are commonly expressed using regular expressions, some forms of regular expression used in practice do not strictly generate the regular languages and do not show linear recognitional performance due to those deviations. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

3.2.5 Analytic Grammars

Though there is a tremendous body of literature on parsing algorithms, most of these algorithms assume that the language to be parsed is initially described by means of a *generative formal grammar*, and that the goal is to transform this generative grammar into a working parser. Strictly speaking, a generative grammar does not in any way correspond to the algorithm used to parse a language, and various algorithms have different restrictions on the form of production rules that are considered well-formed.

An alternative approach is to formalise the language in terms of an analytic grammar in the first place, which more directly corresponds to the structure and semantics of a parser for the language. Examples of analytic grammar formalisms include the following:

- the Language Machine directly implements unrestricted analytic grammars. Substitution rules are used to transform an input to produce outputs and behaviour. The system can also produce the *lm-diagram* which shows what happens when the rules of an unrestricted analytic grammar are being applied.
- top-down parsing language (TDPL): a highly minimalist analytic grammar formalism developed in the early 1970s to study the behaviour of top-down parsers.
- link grammars: a form of analytic grammar designed for linguistics, which derives syntactic structure by examining the positional relationships between pairs of words.
- parsing expression grammars (PEGs): a more recent generalisation of TDPL designed around the practical expressiveness needs of programming language and compiler writers.

SELF ASSESSMENT TEST I

i. In the context of Computer Science, what do you understand by the word ‘grammar’?
ii. Enumerate the components that make up the syntax of grammars
iii. According to this course, list and describe the types of grammars
iv. Given the grammar *G* with following production rules, S → a | aS | bS, determine whether the following strings can be generated by the grammar
    (i) babaab (ii) aaabbbab (iii) bbaaba

3.3 Chomsky Hierarchy

The Chomsky hierarchy (occasionally referred to as **Chomsky–Schützenberger hierarchy**) is a containment hierarchy of classes of formal grammars.

This hierarchy of grammars was described by Noam Chomsky in 1956. It is also named after Marcel-Paul Schützenberger who played a crucial role in the development of the theory of formal languages.

3.3.1 The Hierarchy

The Chomsky hierarchy consists of the levels of grammars as presented in Section 3.2.1 through 3.2.4 above

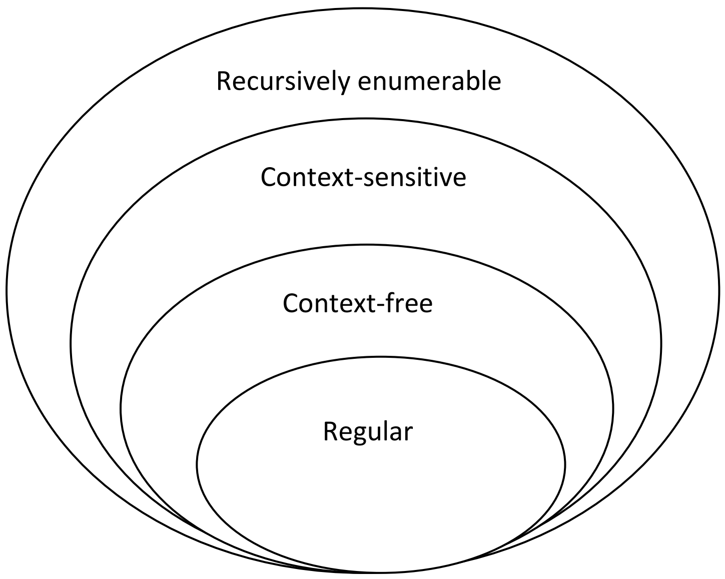


Fig. 1: Set Inclusions Described by the Chomsky Hierarchy

Note that the set of grammars corresponding to recursive languages is not a member of this hierarchy.

Every regular language is context-free, every context-free language, not containing the empty string, is context-sensitive and every context-sensitive language is recursive and every recursive language is recursively enumerable. These are all proper inclusions, meaning that there exist recursively enumerable languages which are not context-sensitive, context-sensitive languages which are not context-free and context-free languages which are not regular.

The following table summarises each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognises it, and the form its rules must have.

Table 1: Summary of the Languages, Automata and Production Rules of Chomsky's Four Types of Grammars

Grammar	**Languages**	**Automaton**	**Production rules (constraints)**
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$(no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

However, there are further categories of formal languages that you can read more about in the further reading.

3.4 Formal Language

A **formal language** is a set of *words*, i.e. finite strings of *letters*, *symbols*, or *tokens*. The set from which these letters are taken is called the *alphabet* over which the language is defined. A formal language is often defined by means of a formal grammar (also called its formation rules); accordingly, words that belong to a formal language are sometimes called *well-formed words* (or well-formed formulas). Formal languages are studied in computer science and linguistics; the field of **formal language theory** studies the purely syntactical aspects of such languages (that is, their internal structural patterns).

3.4.1 Words over an Alphabet

An **alphabet**, in the context of formal languages can be any set, although it often makes sense to use an alphabet in the usual sense of the word, or more generally a character set such as ASCII. Alphabets can also be infinite; e.g. first-order logic is often expressed using an alphabet which, besides symbols such as ∧, ¬, ∀ and parentheses, contains infinitely many elements x_0, x_1, x_2, … that play the role of variables. The elements of an alphabet are called its **letters**.

A **word** over an alphabet can be any finite sequence, or string, of letters. The set of all words over an alphabet Σ is usually denoted by Σ^* (using the Kleene star). For any alphabet there is only one word of length 0, the *empty word*, which is often denoted by e, ε or λ. By concatenation one can combine two words to form a new word, whose length is the sum of the lengths of the original words. The result of concatenating a word with the empty word is the original word.

In some applications, especially in logic, the alphabet is also known as the *vocabulary* and words are known as *formulas* or *sentences*; this breaks the letter/word metaphor and replaces it by a word/sentence metaphor.

Therefore, a **formal language** *L* over an alphabet Σ is just a subset of Σ^*, that is, a set of words over that alphabet.

In computer science and mathematics, which do not usually deal with natural languages, the adjective "formal" is often omitted as redundant. While formal language theory usually concerns itself with formal languages that are described by some syntactical rules, the actual definition of the concept "formal language" is only as above: a (possibly infinite) set of finite-length strings, no more nor less. In practice, there are many languages that can be described by rules, such as regular languages or context-free languages. The notion of a formal grammar may be closer to the intuitive concept of a "language," one described by syntactic rules. By an abuse of the definition, a particular formal language is often thought of as being equipped with a formal grammar that describes it.

Examples 3.3

a. The following rules describe a formal language *L* over the alphabet Σ = {*0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, =*}:

b. Every nonempty string that does not contain + or = and does not start with *0* is in *L*.
The string *0* is in *L*.

c. A string containing = is in *L* if and only if there is exactly one =, and it separates two valid strings in *L*.

d. A string containing + but not = is in *L* if and only if every + in the string separates two valid strings in *L*.

e. No string is in *L* other than those implied by the previous rules.

f. Under these rules, the string "*23+4=555*" is in *L*, but the string "*=234=+*" is not. This formal language expresses natural numbers, well-formed addition statements, and well-formed addition equalities, but it expresses only what they look like (their syntax), not what they mean (semantics). For instance, nowhere in these rules is there any indication that *0* means the number zero or that + means addition.

g. For finite languages one can simply enumerate all well-formed words. For example, we can describe a language *L* as just *L* = {"a", "b", "ab", "cba"}.

However, even over a finite (non-empty) alphabet such as Σ = {a, b} there are infinitely many words: "a", "abb", "ababba", "aaababbbbaab"… Therefore formal languages are typically infinite, and describing an infinite formal language is not as simple as writing *L* = {"a", "b", "ab", "cba"}. Here are some examples of formal languages:

$L = \Sigma^*$, the set of *all* words over Σ;

$L = \{a\}^* = \{a^n\}$, where *n* ranges over the natural numbers and a^n means "a" repeated *n* times (this is the set of words consisting only of the symbol "a");

h. the set of syntactically correct programmes in a given programming language (the syntax of which is usually defined by a context-free grammar);

i. the set of inputs upon which a certain Turing machine halts; or

j. the set of maximal strings of alphanumeric ASCII characters on this line, (i.e., the set {"the", "set", "of", "maximal", "strings", "alphanumeric", "ASCII", "characters", "on", "this", "line", "i", "e"}).

3.4.2 Language-Specification Formalisms

Formal language theory rarely concerns itself with particular languages (except as examples), but is mainly concerned with the study of various types of formalisms to describe languages. For instance, a language can be given as:

- those strings generated by some formal grammar;
- those strings described or matched by a particular regular expression;
- those strings accepted by some automaton, such as a Turing machine or finite state automaton;
- those strings for which some decision procedure (an algorithm that asks a sequence of related YES/NO questions) produces the answer YES.

Typical questions asked about such formalisms include:

- What is their expressive power? (Can formalism *X* describe every language that formalism *Y* can describe? Can it describe other languages?)

- What is their recognisability? (How difficult is it to decide whether a given word belongs to a language described by formalism *X*?)
- What is their comparability? (How difficult is it to decide whether two languages, one described in formalism *X* and one in formalism *Y*, or in *X* again, are actually the same language?).

Surprisingly often, the answer to these decision problems is "it cannot be done at all", or "it is extremely expensive" (with a precise characterisation of how expensive exactly). Therefore, formal language theory is a major application area of computability theory and complexity theory. Formal languages may be classified in the Chomsky hierarchy based on the expressive power of their generative grammar as well as the complexity of their recognising automaton. Context-free grammars and regular grammars provide a good compromise between expressivity and ease of parsing, and are widely used in practical applications.

3.4.3 Operations on Languages

Certain operations on languages are common. This includes the standard set operations, such as union, intersection, and complement. Another class of operation is the element-wise application of string operations.

Example 3.4

a. Suppose L_1 and L_2 are languages over some common alphabet.
b. The *concatenation* L_1L_2 consists of all strings of the form vw where v is a string from L_1 and w is a string from L_2.
c. The *intersection* $L_1 \cap L_2$ of L_1 and L_2 consists of all strings which are contained in both languages
d. The *complement* $\neg L$ of a language with respect to a given alphabet consists of all strings over the alphabets that are not in the language.
e. The Kleene star: the language consisting of all words that are concatenations of 0 or more words in the original language;
Reversal:
f. Let e be the empty word, then $e^R = e$, and
g. for each non-empty word $w = x_1 \ldots x_n$ over some alphabet, let $w^R = x_n \ldots x_1$,
h. then for a formal language L, $L^R = \{w^R \mid w \in L\}$.

String homomorphism

Such string operations are used to investigate closure properties of classes of languages. A class of languages is closed under a particular operation when the operation, applied to languages in the class, always produces a language in the same class again. For instance, the context-free languages are known to be closed under union, concatenation, and intersection with regular languages, but not closed under intersection or complement.

3.4.4 Uses of Formal Languages

Formal languages are often used as the basis for richer constructs endowed with semantics. In computer science they are used, among other things, for the precise definition of data formats and the syntax of programming languages.
Formal languages play a crucial role in the development of compilers, typically produced by means of a compiler, which may be a single programme or may be separated in tools like lexical analyser generators (e.g. lex), and parser generators (e.g. yacc). Since formal languages alone do not have semantics, other formal constructs are needed for the formal specification of programme semantics.

Formal languages are also used in logic and in foundations of mathematics to represent the syntax of formal theories. Logical systems can be seen as a formal language with additional constructs, like proof calculi, which define a consequence relation "Tarski's definition of truth" in terms of a T-schema for first-order logic is an example of *fully interpreted* formal language; all its sentences have meanings that make them either true or false.

SELF-ASSESSMENT EXERCISE II

i. Define formal languages.
ii. What is the relationship between grammar and language?

3.5 Programming Languages

A compiler usually has two distinct components/parts: the analysis part that breaks up the source programme into constant piece and creates an intermediate representation of the source programme and the synthesis part that constructs the desired target programme from the intermediate representation.

A lexical analyser, generated by a tool like **lex**, identifies the tokens of the programming language grammar, e.g. identifiers or keywords, which are themselves expressed in a simpler formal language, usually by means of regular expressions. At the most basic conceptual level, a parser, generated by a parser generator like **yacc**, attempts to decide if the source programme is valid, that is if it belongs to the programming language for which the compiler was built. Of course, compilers do more than just parse the source code; they translate it into some executable format; because of this, a parser usually outputs more than a yes/no answer. Typically an abstract syntax tree, which is used in subsequent stages of the compiler to eventually generate an executable containing machine code that runs directly on the hardware, or some intermediate code that requires a virtual machine to execute.

3.5.1 Formal Theories, Systems and Proofs

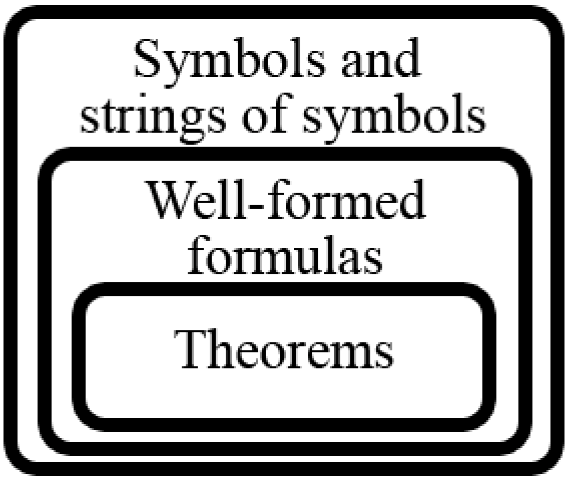


Fig. 2: Syntactic Divisions within a Formal System

Figure 2 shows the syntactic divisions within a formal system. Symbols and strings of symbols may be broadly divided into *nonsense* and *well-formed* formulas. The set of well-formed formulas is divided into theorems and non-theorems. However, quite often, a formal system will simply define all of its well-formed formula as theorems.

In mathematical logic, a formal theory is a set of sentences expressed in a formal language.

A *formal system* (also called a *logical calculus* or a *logical system*) consists of a formal language together with a deductive apparatus (also called a deductive system). The deductive apparatus may consist of a set of transformation rules which may be interpreted as valid rules of inference or a set of axioms, or have both. A formal system is used to derive one expression from one or more other expressions. Although a formal language can be identified with its formulas, a formal system cannot be likewise identified by its theorems. Two formal systems $\mathcal{FS}$and $\mathcal{FS}'$may have all the same theorems and yet differ in some significant proof-theoretic way (a formula A may be a syntactic consequence of a formula B in one but not another for instance).

A formal proof or derivation is a finite sequence of well-formed formulas (which may be interpreted as propositions) each of which is an axiom or follows from the preceding formulas in the sequence by a rule of inference. The last sentence in the sequence is a theorem of a formal system. Formal proofs are useful because their theorems can be interpreted as true propositions.

3.5.2 Interpretations and Models

Formal languages are entirely syntactic in nature but may be given semantics that give meaning to the elements of the language. For instance, in mathematical logic, the set of possible formulas of a particular logic is a formal language, and an interpretation assigns a meaning to each of the formulas - usually, a truth value.

The study of interpretations of formal languages is called formal semantics. In mathematical logic, this is often done in terms of model theory. In model theory, the terms that occur in a formula are interpreted as mathematical structures, and fixed compositional interpretation rules determine how the truth value of the formula can be derived from the interpretation of its terms; a *model* for a formula is an interpretation of terms such that the formula becomes true.

4.0 CONCLUSION

In this unit you have been taken through a brief revision of formal grammars, formal languages and automata because of crucial roles they play in the development of compilers. You should read more on these various topics before proceeding to the next unit. In the subsequent units, you will be learning about compiler construction. Precisely, unit 2 of this module will introduce you to the concept of compilers.

5.0 SUMMARY

In this unit, you learnt that a **formal grammar** is a set of rules of a specific kind, for forming strings in a formal language. It has four components that form its syntax and a set of operations that can be performed on it, which form its semantic.

Each type of grammars is recognised by a particular type of automata. For example, type-2 grammars are recognised by pushdown automata while type-3 grammars are recognised by finite state automata.

Parsing is the process of recognising an utterance by breaking it down to a set of symbols and analysing each one against the grammar of the language.

According to Chomsky hierarchy, there are four types of grammars. The difference between these types is that they have increasingly strict production rules and can express fewer formal languages.

A **formal language** is a set of *words*, i.e. finite strings of *letters*, *symbols* or *tokens*. The set from which these letters are taken is called the *alphabet* over which the language is defined. A formal language is often defined by means of a formal grammar.

Formal languages play a crucial role in the development of compilers and precise definition of the syntax of programming languages. A *formal system* consists of a formal language together with a deductive apparatus.

6.0 TUTOR-MARKED ASSIGNMENT

i. Name the class of automata that are used in recognising the following grammars:
   a. Regular grammars
   b. Context-sensitive grammars
   c. Type-0 grammars
   d. Context-free grammars
ii. What are the use(s) of formal languages?
iii. What does it mean to say a class of languages is closed under a particular operation? Hence or otherwise suppose L_1 and L_2 are languages over some common alphabet; state (with appropriate examples) the standard operations that can be performed on the languages.
iv. From what you have learnt so far in this course, justify the relevance of formal languages to computer programming?
v. Briefly discuss the Chomsky hierarchy. What is the relationship among the various types of grammars described in the Chomsky hierarchy?

7.0 REFERENCES/FURTHER READING

Arnon, Avron (1994). What is a logical system? Chapter 8. In: Dov M. Gabbay (Ed.). *What is a logical system?* Oxford University Press.

Chomsky, N. & Schützenberger, M. P. (1963). "The algebraic theory of context free languages". In: P. Braffort, D. Hirschberg.*Computer*

Programming and Formal Languages. Amsterdam: North Holland. pp. 118–161.

Chomsky, Noam (1956). "Three models for the description of language". *IRE Transactions on Information Theory* (2): 113–124. Retrieved from http://www.chomsky.info/articles/195609--.pdf.

Chomsky, Noam (1959). "On certain formal properties of grammars". *Information and Control* 2 (2): 137–167.

Davis, M. E., Sigal, R. & Weyuker, E. J. (1994). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Boston: Academic Press, Harcourt, Brace. pp. 327.

Ginsburg, S. (1975). *Algebraic and Automata Theoretic Properties of Formal Languages*. North-Holland.

Godel, Escher, Bach: An Eternal Golden Braid, Douglas Hofstadter

Grzegorz Rozenberg & Arto Salomaa (1997). *Handbook of Formal Languages:* Volume I-III, Springer. ISBN 3 540 61486 9.

Hamilton, G. (1978). *Logic for Mathematicians*. Cambridge University Press. ISBN 0 521 21838 1.

Harrison, M. A. (1978). *Introduction to Formal Language Theory*. Addison-Wesley.

Hopcroft, J. E. & Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley:Publishing, Reading Massachusetts.

Suppes, P. (1957). *Introduction to Logic*. D. Van Nostrand.

UNIT 2 WHAT IS A COMPILER?

CONTENTS

1.0 INTRODUCTION

In the previous unit you were taken through some basic concepts you learnt in an earlier course. This was done because of their relevance/importance to your understanding of this course.

In this unit you will be introduced to the concept of compilers and their importance to programme development.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define compiler and its importance in the programming world
- distinguish between a translator, compiler and an interpreter
- discuss the major challenges to be faced in building compilers
- state the qualities of compilers
- mention some of the knowledge required for building compilers
- describe the architecture of a compiler.

3.0 MAIN CONTENT

3.1 Translators

A translator is a programme that takes as input a programme written in one programming language (the source language) and produces as output a programme in another language (the object or target language). If the source language is a high-level language such as COBOL, PASCAL, etc. and the object language is a low-level language such as an assembly language or machine language, then such a translator is called a *Compiler*.

Executing a programme written in a high-level programming language is basically a two-step process, as illustrated in Figure 1. The source programme must first be compiled, that is, translated into object programme. Then the resulting object programme is loaded into memory and executed.

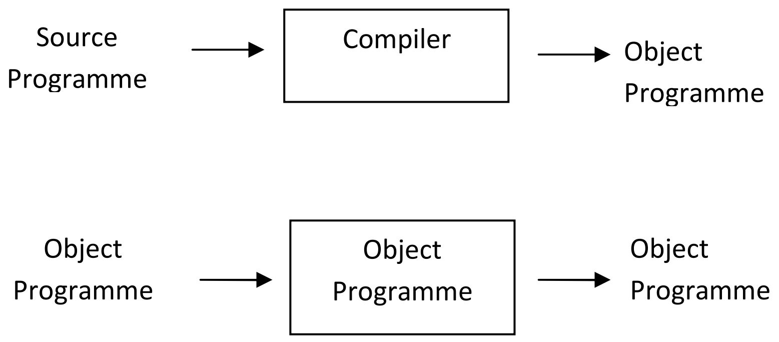


Fig. 1: Compilation and Execution

Certain other translators transform a programming language into a simplified language, called intermediate code, which can be directly executed using a programme called an interpreter. You can think of the intermediate code as the machine language of an abstract computer designed to execute the source code.

There are other important types of translators, besides compilers. If the source language is assembly language and the target language is machine language, then the translator is called an *assembler*. The term *preprocessor* is used for translators that take programmes in one high-level language into equivalent programmes in another high level language. For example, there many FORTRAN preprocessors that map 'structured' versions of FORTRAN into conventional FORTRAN.

3.2 Why Do We Need Translators?

We need translators to overcome the rigour of programming in machine language, which involves communicating directly with a computer in terms of bits, register, and primitive machine operations. As you have learnt in earlier courses in this programme, a machine language programme is a sequence of 0's and 1's, therefore, programming a complex algorithm in such a language is terribly tedious and prone to mistakes.

Translators free the programmer from the hassles of programming in machine language.

3.3 What is a Compiler?

A *compiler* is a programme that translates a source programme written in some high-level programming language (such as Java) into machine code for some computer architecture (such as the Intel Pentium architecture). The generated machine code can later be executed many times against different data each time.

An *interpreter* reads an executable source programme written in a high-level programming language as well as data for this programme, and it runs the programme against the data to produce some results. One example is the UNIX shell interpreter, which runs operating system commands interactively.

You should note that both interpreters and compilers (like any other programme) are written in some high-level programming language (which may be different from the language they accept) and they are translated into machine code. For example, a Java interpreter can be completely written in Pascal, or even Java. The interpreter source programme is machine independent since it does not generate machine code. (Note the difference between *generate* and *translated into* machine code.) An interpreter is generally slower than a compiler because it processes and interprets each statement in a programme as many times as the number of the evaluations of this statement. For example, when a for-loop is interpreted, the statements inside the for-loop body will be analysed and evaluated on every loop step. Some languages, such as Java and Lisp, come with both an interpreter and a compiler. Java source programmes (Java classes with .java extension) are translated by the java compiler into byte-code files (with .class extension). The Java interpreter, java, called the Java Virtual Machine (JVM), may actually interpret byte codes directly or may internally compile them to machine code and then execute that code.

Like was mention in section 3.1, compilers and interpreters are not the only examples of translators. In the table below are a few more:

Table 1: Table of Translators, Source Language and Target Language

Source Language	**Translator**	**Target Language**
LaTeX	Text Formater	PostScript
SQL	database query optimizer	Query Evaluation Plan
Java	javac compiler	Java byte code
Java	cross-compiler	C++ code
English text	Natural Language Understanding	semantics (meaning)
Regular Expressions	JLex scanner generator	a scanner in Java
BNF of a language	CUP parser generator	a parser in Java

This course deals mainly with compilers for high-level programming languages, but the same techniques apply to interpreters or to any other compilation scheme.

SELF-ASSESSMENT EXERCISE

i. What is a translator?
ii. What is the importance of translators in programming?
iii. Distinguish among a translator, compiler and an interpreter.

3.4 The Challenge in Compiler Development

There are various challenges involved in developing compilers; some of these are itemised below:

1) Many variations:
   a. many programming languages (e.g. FORTRAN, C++, Java)
   b. many programming paradigms (e.g. object-oriented, functional, logic)
   c. many computer architectures (e.g. MIPS, SPARC, Intel, alpha)
   d. many operating systems (e.g. Linux, Solaris, Windows)
2) Qualities of a compiler**:** these concerns the qualities that are compiler must possess in other to be effective and useful. These are listed below in order of importance:
   a. the compiler itself must be bug-free
   b. it must generate correct machine code
   c. the generated machine code must run fast

d. the compiler itself must run fast (compilation time must be proportional to programme size)
e. the compiler must be portable (i.e. modular, supporting separate compilation)
f. it must print good diagnostics and error messages
g. the generated code must work well with existing debuggers
h. must have consistent and predictable optimisation.

3) In-depth knowledge:

Building a compiler requires in-depth knowledge of:

a. programming languages (parameter passing, variable scoping, memory allocation, etc.)
b. theory (automata, context-free languages, etc.)
c. algorithms and data structures (hash tables, graph algorithms, dynamic programming, etc.)
d. computer architecture (assembly programming)
e. software engineering.

You should try building a non-trivial compiler for a Pascal-like language as the course project. This will give you a hands-on experience on system implementation that combines all this knowledge.

3.5 Compiler Architecture

As earlier mentioned, a compiler can be viewed as a programme that accepts a source code (such as a Java programme) and generates machine code for some computer architecture. Suppose that you want to build compilers for *n* programming languages (e.g. FORTRAN, C, C++, Java, BASIC, etc.) and you want these compilers to run on *m* different architectures (e.g. MIPS, SPARC, Intel, alpha, etc.). If you do that naively, you need to write *n*m* compilers, one for each language-architecture combination.

The holly grail of portability in compilers is to do the same thing by writing $n + m$ programmes only. You can do this by using a universal *Intermediate Representation* (*IR*) and you make the compiler a two-phase compiler. An IR is typically a tree-like data structure that captures the basic features of most computer architectures. One example of an IR tree node is a representation of a 3-address instruction, such as $d \leftarrow s_1 + s_2$ that gets two source addresses, s_1 and s_2, (i.e. two IR trees) and produces one destination address, *d*. The first phase of this compilation scheme, called the *front-end*, maps the source code into IR, and the second phase, called the *back-end*, maps IR into machine code. That way, for each programming language you want to compile, you write one front-end only, and for each computer architecture, you write one back-end. So, totally you have $n + m$ components.

But the above ideal separation of compilation into two phases does not work very well for real programming languages and architectures. Ideally, you must encode all knowledge about the source programming language in the front end, you must handle all machine architecture features in the back end, and you must design your IRs in such a way that all language and machine features are captured properly.

A typical real-world compiler usually has multiple phases (this will be treated to greater details in unit 3 of this module. This increases the compiler's portability and simplifies retargeting. The front end consists of the following phases:

- *scanning*: a scanner groups input characters into tokens
- *parsing*: a parser recognises sequences of tokens according to some grammar and generates *Abstract Syntax Trees* (ASTs)
- *semantic analysis*: performs *type checking* (i.e. checking whether the variables, functions etc. in the source programme are used consistently with their definitions and with the language semantics) and translates ASTs into IRs
- *optimisation*: optimises IRs.

The back end consists of the following phases:

- *instruction selection*: maps IRs into assembly code
- *code optimisation*: optimises the assembly code using control-flow and data-flow analyses, register allocation, etc
- *code emission*: generates machine code from assembly code.

The generated machine code is written in an object file. This file is not executable since it may refer to external symbols (such as system calls). The operating system provides the following utilities to execute the code:

- *linking*: A linker takes several object files and libraries as input and produces one executable object file. It retrieves from the input files (and puts them together in the executable object file) the code of all the referenced functions/procedures and it resolves all external references to real addresses. The libraries include the operating system libraries, the language-specific libraries, and, maybe, user-created libraries.
- *loading*: A loader loads an executable object file into memory, initialises the registers, heap, data, etc. and starts the execution of the programme.
- Relocatable shared libraries allow effective memory use when many different applications share the same code.

4.0 CONCLUSION

In this unit you have been taken through the definition, functions, and architecture of a compiler in the next unit you will be learning more about the structure of a compiler and the various phases involved in compilation process

5.0 SUMMARY

In this unit, you learnt that:

- a translator is a programme that takes as input a programme written in one programming language (the source language) and produces as output a programme in another language (the object or target language)
- a *compiler* is a programme that translates a source programme written in some high-level programming language (such as Java) into machine code for some computer architecture (such as the Intel Pentium architecture)
- an *interpreter* reads an executable source programme written in a high-level programming language as well as data for this programme, and it runs the programme against the data to produce some results
- translators are needed to free the programmer from the hassles of programming in machine language and its attendant problems
- there are several challenges to be surmounted in developing a compiler.

6.0 TUTOR-MARKED ASSIGNMENT

i. What are the challenges involved in developing compilers?
ii. Enumerate the essential qualities of a compiler.
iii. Distinguish between the function of loader and a linker.
iv. Outline some specific knowledge require for building a compiler.

7.0 REFERENCES/FURTHER READING

Aho, A. V. & Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley Publishing Company. ISBN 0-201-00022-9.

Chomsky, Noam & Schützenberger, Marcel P. (1963). "The algebraic Theory of Context free Languages". In: P. Braffort & D. Hirschberg. *Computer Programming and Formal Languages*. Amsterdam: North Holland. pp. 118–161.

Davis, Martin E., Sigal, Ron & Weyuker, Elaine J. (1994). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Boston: Academic Press, Harcourt, Brace. pp. 327. ISBN 0122063821.

Grzegorz Rozenberg and Arto Salomaa (1997). *Handbook of Formal Languages*. Volume I-III. Springer. ISBN 3 540 61486 9.

John, E. Hopcroft & Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing, Reading Massachusetts. ISBN 0-201-029880-X.

Michael, A. Harrison(1978). *Introduction to Formal Language Theory*, Addison-Wesley.

UNIT 3 THE STRUCTURE OF A COMPILER

CONTENTS

1.0 INTRODUCTION

In the previous unit you were introduced to the concept of compilers and their roles in programming.

In this unit you will learn about the structure of a compiler and the various phases involved in the compilation process.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- list the various components of a compiler
- describe the activities that take place at each of the compilation phases
- explain cross compilation
- analyse hand implementation.

3.0 MAIN CONTENT

3.1 The Structure of a Compiler

We can identify four components

1. Front end
2. Back-end
3. Tables of information
4. Runtime library

i) **Front-End: t**he front-end is responsible for the analysis of the structure and meaning of the source text. This end is usually the analysis part of the compiler. Here we have the syntactic analyser, semantic analyser, and lexical analyser. This part has been automated.

ii) **Back-End:** The back-end is responsible for generating the target language. Here we have intermediate code optimiser, code generator and code optimiser. This part has been automated.

iii) **Tables of Information:** It includes the symbol-table and there are some other tables that provide information during compilation process.

iv) **Run-Time Library:** It is used for run-time system support.

Languages for Writing Compiler

a. Machine language
b. Assembly language
c. High level language or high level language with bootstrapping facilities for flexibility and transporting.

3.2 Phases of a Compiler

A compiler takes as input a source programme and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of sub-processes called phases as shown in the figure 1 below. A phase is a logically cohesive operation that takes as input one representation of the source programme and produces as output another representation.

3.2.1 The Lexical Analyser

this is the first phase and it is also referred to as the *Scanner*. It separates characters of the source language into groups that logically belong together; these groups are called *tokens*. The usual tokens are keywords, such as DO or IF, identifiers such as X or NUM, operator symbol such as <= or +, and punctuation symbol such as parentheses or commas. The output of the lexical analyser is a stream of tokens, which is passed to the next phase, the *syntax analyser* or *parser*. The tokens in this stream can be represented by codes which we may regard as integers. Thus DO might be represented by 1, + by 2, and "identifier" by 3. In the case of a token like "identifier", a second quantity, telling which of those identifiers used by the programme is represented by this instance of token "identifier" is passed along with the integer code for "identifier". For Example, in the FORTRAN statement:

IF (5 .EQ. MAX) GO TO 100
we find the following eight tokens: IF; (; 5; .EQ; MAX;); GOTO; 100.

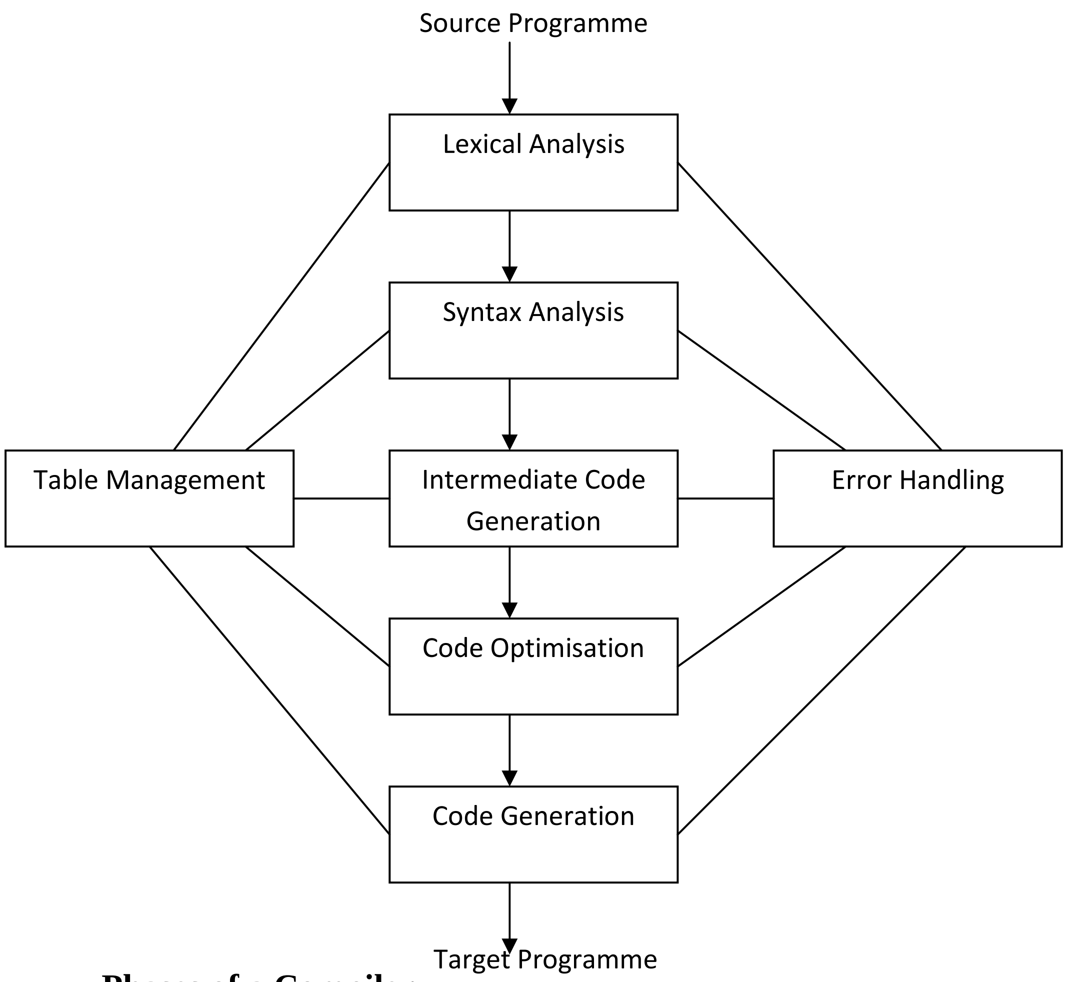


Fig. 1: Phases of a Compiler

3.2.2 The Syntax Analyser

This groups tokens together into syntactic structures. For example, the three tokens representing A+B might be grouped into a syntactic structure called an *expression*. Expressions might further be combined to form statements. Often the syntactic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together. The parser has two functions. It checks that the tokens appearing in its input, which is the output of the lexical analyser, occur in patterns that are permitted by the specification for the source language. It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

3.2.3 The Intermediate Code Generator

This uses the structure produced by the syntax analyser to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instructions with one operator and a small number of operands. These instructions with one operator and a small number of operands. These instructions can be viewed as simple macros like the macro ADD2. the primary difference between intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.

3.2.4 Code Optimisation

This is an optional phase designed to improve the intermediate code so that the ultimate object programme runs faster and/or takes less space. Its output is another intermediate code programme that does the same job as the original, but perhaps in a way that saves time and/or space.

3.2.5 Code Generation

This is the final phase and it produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done. Designing a code generator that produces truly efficient object programmes is one of the most difficult parts of a compiler design, both practically and theoretically.

3.2.6 The Table Management or Bookkeeping

This portion of the compiler keeps track of the names used by the programme and records essential information about each, such as its type (integer, real, etc.). The data structure used to record this information is called a *symbol table*.

3.2.7 The Error Handler

This is invoked when a flaw in the source programme is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. It is desirable that compilation be completed on flawed programmes, at least through the syntax-analysis phase, so that as many errors as possible can be detected in one compilation. Both the table management and error handling routines interact with all phases of the compiler

SELF-ASSESSMENT EXERCISE

i. List the various phases of the compilation process.
ii. Why is error handler important in a compiler?
iii. Which is the first phase of the compiler?

3.2 Passes

In an implementation of a compiler, portions of one or more phases are combined into a module called a pass. A pass reads the source programme or the output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

The numbers of passes, and the grouping of phases into passes, are usually dictated by a variety of considerations germane to a particular language and machine, rather than by any mathematical optimality criteria.

The structure of the source language has strong effect on the number of passes. Certain languages require at least two passes to generate code easily. For example, languages such as PL/I or ALGOL 68 allow the declaration of a name to occur after uses of that name. Code for expression containing such a name cannot be generated conveniently until the declaration has been seen.

3.3 Reducing the Number of Passes

Since each phase is a transformation on a stream of data representing an intermediate form of the source programme, you may wonder how several phases can be combined into one pass without the reading and writing of intermediate files. In some cases one pass produces its output with little or no memory of prior inputs. Lexical analysis is typical. In this situation, a small buffer serves as the interface between passes. In other cases, you may merge phases into one pass by means of a technique known as 'back patching'. In general terms, if the output of a phase cannot be determined without looking at the remainder of the phase's input, the phase can generate output with 'slots' which can be filled in later, after more of the input is read.

3.4 Cross Compilation

Consider porting a compiler for C written in C from an existing machine A to an existing machine B.

Steps to Cross Compilation

a. Write new back-end in C to generate code for computer B
b. Compile the new back-end and using the existing C compiler running on computer A generating code for computer B.
c. We now have a compiler running on computer A and generating code for computer B.
d. Use this new compiler to generate a complete compiler for computer B. In other words, we can compile the new compiler on computer A to generate code for computer B
e. We now have a complete compiler for computer B that will run on computer B.
f. Copy this new compiler across and run it on computer B (this is cross Compilation).

3.5 Operations of a Compiler

As you have seen in section 3.1 of this unit, the operation of a compiler includes:

- The lexical analysis (or scanning)
- The syntax analysis Front-end
- Semantic analysis

- Intermediate code optimisation
- Code generation } Back-end
- Code optimisation

In this course, we will concern ourselves with mostly the front-end i.e. those parts of compilation that can be automated which are lexical, syntax and probably semantic analyses.

3.6 T-Diagrams

They are used to describe the nature of a compilation. It is usually in the form of T and is diagrammatically represented as in figure 2 below:

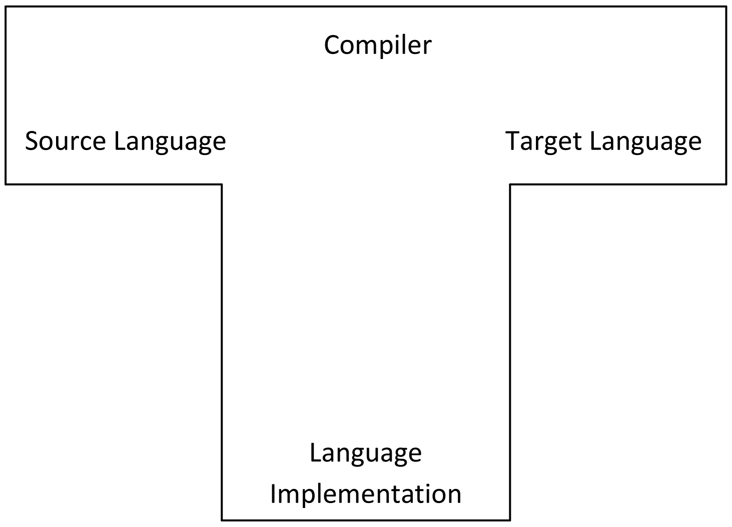


Fig. 2: T-Diagrams

4.0 CONCLUSION

In this unit you have been taken through the structure, phases and functions of each phase of a compiler. In the next unit you will be learning more about the compilation process; such as cross compilation and hand implementation.

5.0 SUMMARY

In this unit, you learnt that:

- the compilation process can be partitioned into a series of sub-processes called phases
- several phases can be grouped into one pass, so that the operation of the phases may be interleaved, with control alternating among several phases
- the numbers of passes, and the grouping of phases into passes, are usually dictated by a variety of considerations germane to a particular language and machine
- the operations of a compiler can be classified into two: front-end comprising the lexical analysis (or scanning), the syntax

analysis, and semantic analysis and the back-end comprising of intermediate code optimisation, code generation, and code optimisation.

6.0 TUTOR-MARKED ASSIGNMENT

i. Describe what happens at each phase of the compilation process.
ii. Distinguish between the intermediate code generator and the code generator phase of the compiler.
iii. Which of the phases of the compiler is optional?
iv. In your own opinion would error handler be part of an interpreter?
v. Enumerate the steps in cross compilation.

7.0 REFERENCES/FURTHER READING

Aho, A. V. & Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley Publishing Company. ISBN 0-201-00022-9.

Chomsky, Noam & Schützenberger, Marcel P. (1963). "The algebraic Theory of Context free Languages." In: P. Braffort & D.Hirschberg. *Computer Programming and Formal Languages*. Amsterdam: North Holland. pp. 118–161.

MODULE 2 LEXICAL ANALYSIS

UNIT 1 THE SCANNER

CONTENTS

1.0 INTRODUCTION

As you learnt in unit 3 of the previous module, the function of the lexical analyser is to read the source programme, one character at a time, and translate it into a sequence of primitive units called tokens. Keywords, identifiers, constants, and operators are examples of tokens.

This module, starting from this unit, exposes you to the problem of designing and implementing lexical analysers.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- state the role of a compiler
- state the need of a compiler
- define the scanner
- state the functions of the scanner.

3.0 MAIN CONTENT

3.1 The Role of the Lexical Analyser

The lexical analyser could be a separate pass, placing its output on an intermediate file from which the parser would then take its input. But, more commonly, the lexical analyser and the parser are together in the same pass; the lexical analyser acts as a subroutine or co-routine, which is called by the parser whenever it needs a new token. This organisation eliminates the need for the intermediate file. In this arrangement, the lexical analyser returns to the parser a representation for the token it has found. The representation is usually an integer code if the token is a simple constructs such as left parenthesis, comma, or colon; it is a pair consisting of an integer code and a pointer to a table if the token is a more complex element such as an identifier or constant. The integer code gives the token type, the pointer points to the value of that token.

3.2 The Need for Lexical Analyser

The purpose of splitting analysis of the source programme into two phases, lexical analysis and syntactic analysis, is to simplify the overall design of the compiler. This is because it is easier to specify the structure of a token than the syntactic structure of the source programme. Therefore, a more specialised and more efficient recogniser can be constructed for tokens than for syntactic structures.

By including certain constructs in the lexical rather than the syntactic structure, we can greatly simplify the design of the syntax analyser.

Lexical analyser also performs other functions such as keeping track of line numbers, producing an output listing if necessary, stripping out white space (such as redundant blanks and tabs), and deleting comments.

SELF- ASSESSMENT EXERCISE

i. Enumerate the functions performed by the lexical analyser.
ii. What is the advantage of implementing the lexical analyser and the parser in the same pass?

3.3 The Scanner

A *scanner* groups input characters into tokens. For example, if the input is:

```
x = x*(b+1);
```

then the scanner generates the following sequence of tokens

```
id(x)
=
id(x)
*
(
id(b)
+
num(1)
)
;
```

where id(x) indicates the identifier with name x (a programme variable in this case) and num(1) indicates the integer 1. Each time the parser needs a token, it sends a request to the scanner. Then, the scanner reads as many characters from the input stream as it is necessary to construct a single token. The scanner may report an error during scanning (e.g. when it finds an end-of-file in the middle of a string). Otherwise, when a single token is formed, the scanner is suspended and returns the token to the parser. The parser will repeatedly call the scanner to read all the tokens from the input stream or until an error is detected (such as a syntax error).

Tokens are typically represented by numbers. For example, the token * may be assigned number 35. Some tokens require some extra information. For example, an identifier is a token (so it is represented by some number) but it is also associated with a string that holds the identifier name. For example, the token id(x) is associated with the string, "x". Similarly, the token num(1) is associated with the number, 1. Tokens are specified by patterns, called *regular expressions*. For example, the regular expression [a-z][a-zA-Z0-9]* recognises all identifiers with at least one alphanumeric letter whose first letter is lower-case alphabetic.

A typical scanner:

- recognises the *keywords* of the language (these are the reserved words that have a special meaning in the language, such as the word class in Java);
- recognises special characters, such as (and), or groups of special characters, such as := and ==;
- recognises identifiers, integers, reals, decimals, strings, etc;
- ignores whitespaces (tabs and blanks) and comments;
- recognises and processes special directives (such as the #include "file" directive in C) and macros.

A key issue is speed. One can always write a naive scanner that groups the input characters into lexical words (a lexical word can be either a sequence of alphanumeric characters without whitespaces or special characters, or just one special character), and then tries to associate a token (i.e. number, keyword, identifier, etc.) to this lexical word by performing a number of string comparisons. This becomes very expensive when there are many keywords and/or many special lexical patterns in the language. In this unit you will learn how to build efficient scanners using regular expressions and finite automata. There are automated tools called *scanner generators*, such as *flex* for C and *JLex* for Java, which construct a fast scanner automatically according to specifications (regular expressions). You will first learn how to specify a scanner using regular expressions, then the underlying theory that scanner generators use to compile regular expressions into efficient programmes (which are basically finite state machines), and then you will learn how to use a scanner generator for Java, called JLex.

4.0 CONCLUSION

In this unit you have been taken through the fundamental concepts and importance of the lexical analyser. In the next unit you will be learning more about the workings and implementation of a lexical analyser

5.0 SUMMARY

In this unit, you learnt that:

- the lexical analyser could be a separate pass, placing its output on an intermediate file from which the parser would then take its input
- when the lexical analyser and the parser are together in the same pass, the lexical analyser acts as a subroutine or co-routine, which is called by the parser whenever it needs a new token
- the analysis of the source programme is usually split into two phases, lexical analysis and syntactic analysis, to simplify the overall design of the compiler
- a scanner groups input characters into tokens
- the key issue in the design of a scanner is speed.

6.0 TUTOR-MARKED ASSIGNMENT

i. What is a scanner?
ii. Enumerate the functions of the scanner.
iii. Write a naive scanner that groups the input characters of a compiler into lexical words.

7.0 REFERENCES/FURTHER READING

Aho, A. V. & Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley Publishing Company. ISBN 0-201-00022-9

Chomsky, Noam & Schützenberger, Marcel P. (1963). "The algebraic Theory of Context Free Languages." In: P. Braffort & D. Hirschberg. *Computer Programming and Formal Languages*. Amsterdam: North Holland. pp. 118–161.

Davis, Martin E., Sigal, Ron & Weyuker, Elaine J. (1994). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Boston: Academic Press, Harcourt, Brace. pp. 327. ISBN 0122063821.

Grzegorz Rozenberg & Arty Salomaa (1997). *Handbook of Formal Languages:* Volume I-III, Springer. ISBN 3 540 61486 9.

John, E. Hopcroft & Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing, Reading Massachusetts. ISBN 0-201-029880-X.

Michael, A. Harrison (1978). *Introduction to Formal Language Theory*. Addison-Wesley.

UNIT 2 HAND IMPLEMENTATION OF LEXICAL ANALYSER

CONTENTS

1.0 INTRODUCTION

Having learnt about the role and need of a lexical analyser in the previous unit, in this unit we will go on to discuss various methods of constructing and implementing a lexical analyser.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- list the various methods of constructing a lexical analyser
- describe the input buffering method of constructing a lexical analyser
- explain the transition diagram method of constructing a lexical analyser
- state the problems with hand implementation method of constructing lexical analysers
- construct transition diagrams to handle keywords, identifiers and delimiters.

3.0 MAIN CONTENT

3.1 Lexical Analysis

In lexical analysis, we read the source programme character by character and converge them to tokens.

A token is the smallest unit recognisable by the compiler. There are basically a few numbers of tokens that are recognised. Generally, we have four classes of tokens that are usually recognised and they are:

1. Keywords
2. Identifies
3. Constants
4. Delimiters

3.2 Construction of Lexical Analyser

There are 2 general ways to construct lexical analyser:

- Hand implementation
- Automatic generation of lexical analyser

3.2.1 Hand Implementation

There are two ways to use hand implementation:

- Input Buffer approach
- Transitional diagrams approach

3.2.1.1 Input Buffering

The lexical analyser scans the characters of the source programme one at a time to discover tokens. Often, however, many characters beyond the next token may have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyser to read its input from an input buffer. There are many schemes that can be used to buffer input but we shall discuss only one of these schemes here. Figure 1shows a buffer divided into two halves of, say, 100 characters. One pointer marks the beginning of the token being discovered. We view the position of each pointer as been between the character last read and the character next to be read. In practice, each buffering scheme adopts one convention; either a pointer is at the symbol last read or the symbol it is ready to read.

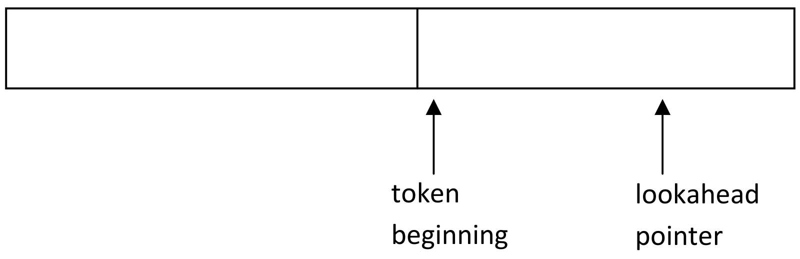


Fig. 1: Input Buffer

The distance which the lookahead pointer may have to travel past the actual token may be large. For example, in a PL/I programme we may see

DECLARE (ARG1, ARG2... ARG*n*)

- without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the lookahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source.

Since the buffer of Figure 1 is of limited size, there is an implied constraint on how much lookahead can be used before the next token is discovered. For example, in Figure 1, if the lookahead travelled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we choose or use another buffering scheme, we cannot ignore the fact that lookahead is limited.

By using buffer, we mean that you read part of the text into the buffer (temporary storage) and then begin to scan by using get character (GETCHAR) to scan and you form the token as you go along.

e.g.
real x, y, z
integer r, q
read (6, 7) (a (i), I = 1, 20)

You can prepare a buffer of 80 characters and read the first line to it and begin to scan, and then go to the next line.

Problems associated with hand implementation

Where the language allows keyword to be used as an identifier e.g. DC: in PL2

Where the language does not recognise blanks as a separator, you will not stop scanning until you reach a comma. Like in FORTRAN: using hand implementation for real x, y, z; you take "realx" as an identifier.

3.2.1.2 Transition Diagram (TD)

One way to begin the design of any programme is to describe the behaviour of the programme by a flowchart. This approach is particularly useful when the programme is a lexical analyser, because the action taken is highly dependent on what character has been seen recently. Remembering previous characters by the position in a flowchart is a valuable too, so much so that a specialised kind of flowchart for lexical analysers, called *transition diagram*, has evolved. In a TD, the boxes of the flowchart are drawn as circles and called *states*. The states are connected by arrows, called *edges*. The labels on the various edges leaving a state indicate the input characters that can appear after that state.

In TD, we try to construct a TD for each token, and then link up.

For example, the TD for keywords and identifiers are the same. See in Figure 2 below an automation you can use for an identifier

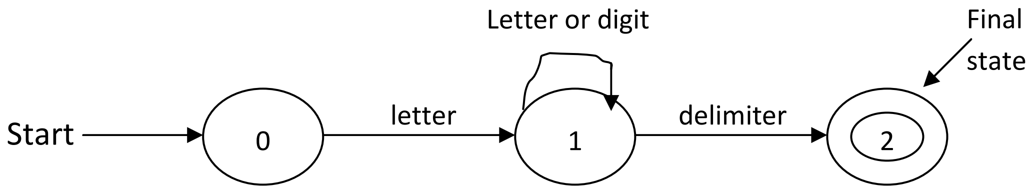


Fig. 2: Transition Diagram for Identifier

If the automaton sees a letter in state 0, it goes to state 1, if it sees a letter or digit in state 1 it remains there. But if it sees a delimiter while in state 1, it moves to state 2, which is the final state.

3.2.1.3 How to Handle Keywords

There are two ways we can handle keywords.

We can use the transition diagrams for the identifier and when you get a delimiter, you look up a dictionary (that contains all the keywords) to see if the identifier you are seeing is a keyword or not.

Another way is to bring all the keywords together in a TD i.e. construct a TD for each keyword.

E.g. Suppose the following keywords exist in a language: BEGIN, END, IF, THEN, ELSE. You can construct a single TD for all of them and then you have something like the diagram in Figure 3 below.

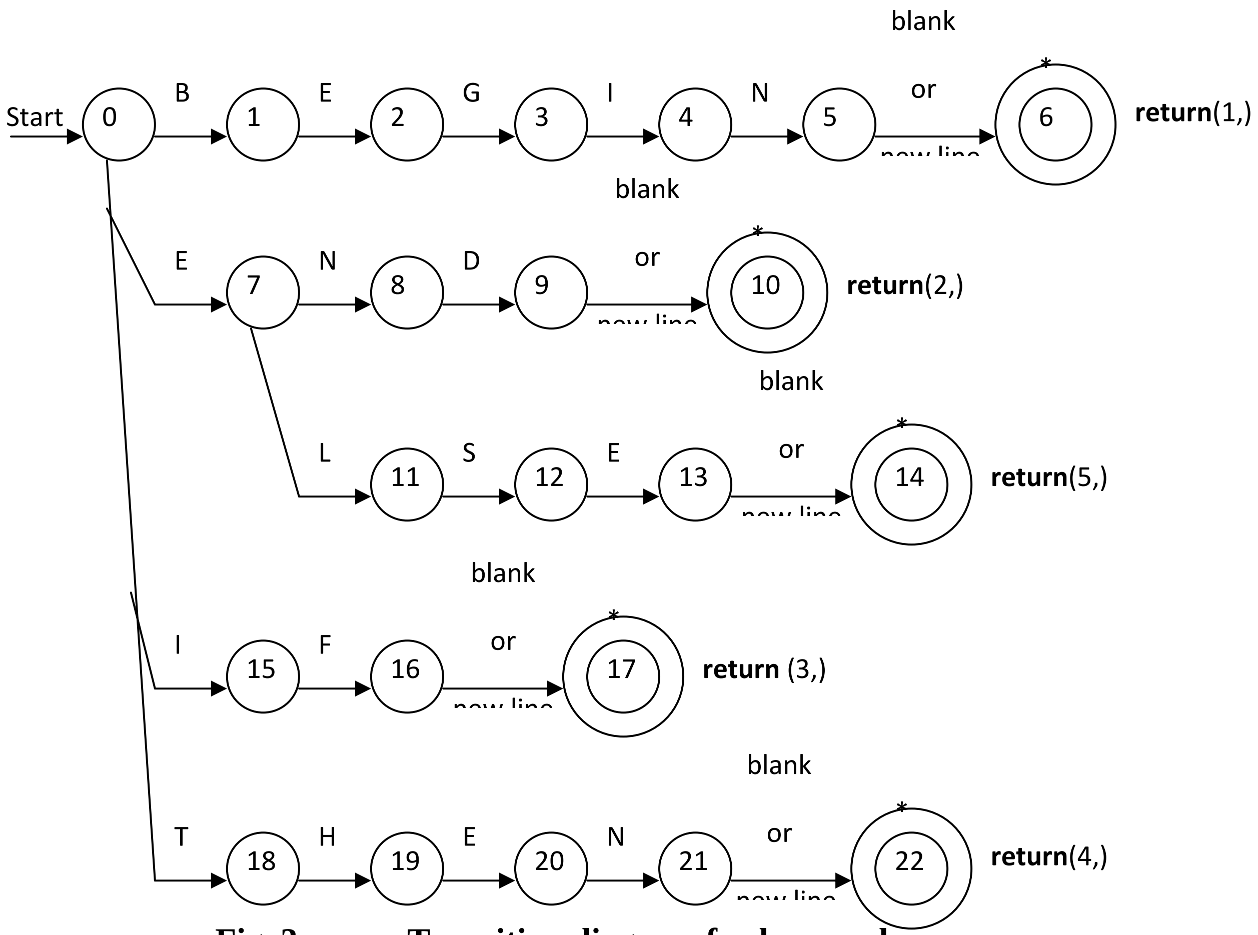


Fig. 3: Transition diagram for keywords

NOTE:
Here, we make the keywords the basis of our search whereas in the first method, we make the identifiers the basis of our search.

We are also assuming here that the keyword cannot be the beginning of an identifier. Therefore, if it sees something like "THENPAP", it will break it into two like "THEN PAP" because you are not allowed to use keywords as the beginning of an identifier.

For delimiters especially relational operators, you can also construct a TD like the one above.

SELF -ASSESSMENT EXERCISE

i. What are the different ways to construct a lexical analyser?
ii. State the problems with hand implementation.

4.0 CONCLUSION

In this unit you have been taken through hand implementation method of lexical analyser construction. This method can be carried out in two ways: input buffer and transition diagram. Due to the attendant problems of this method, there are now tools that are used in automatic generation of lexical analyser. This will be discussed in later unit of this module.

5.0 SUMMARY

In this unit, you learnt that:

- the lexical analyser could be constructed by using hand implementation or automatic generation
- hand implementation can be done by input buffering or transition diagram
- hand implementation has some attendant problems
- using TD, keywords can be handled two ways.

6.0 TUTOR-MARKED ASSIGNMENT

i. Describe how to construct a TD.
ii. How is a TD similar to a flowchart?
iii. Construct a TD to recognise relational operators.

7.0 REFERENCES/FURTHER READING

Aho, A V. & Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley Publishing Company. ISBN 0-201-00022-9

Chomsky, Noam & Schützenberger, Marcel P. (1963). "The algebraic theory of context free languages." In: P. Braffort & D. Hirschberg. *Computer Programming and Formal Languages*. Amsterdam: North Holland. pp. 118–161.

Davis, Martin E., Sigal, Ron & Weyuker, Elaine J. (1994). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Boston: Academic Press, Harcourt, Brace. pp. 327. ISBN 0122063821.

John, E. Hopcroft & Jeffrey, D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing, Reading Massachusetts. ISBN 0-201-029880-X.

UNIT 3 AUTOMATIC GENERATION OF LEXICAL ANALYSER

CONTENTS

1.0 INTRODUCTION

In the previous unit you learnt about hand implementation and the two ways it can be carried out. You also learnt about the problems with hand implementation.

In this unit, we will discuss regular expressions and the automatic generation of lexical analysers.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define regular expressions (Res)
- define basic terms such as tokens, patterns, lexemes and attributes
- distinguish between normal regular expressions and lex regular expressions
- Construct Lex-style regular expressions for patterns
- Write the language that is denoted by any regular expressions
- describe tools for generating lexical analysers.

3.0 MAIN CONTENT

3.1 Automatic Generation of Lexical Analyser

There are tools that can generate lexical analyser for you. But before we begin the discussion of the design of a programme for generating lexical analysers, you will first be introduced to a very useful notation, called regular expressions, suitable for describing tokens.

3.2 Language Theory Background

3.2.1 Definitions

Symbol: (character, letter)
Alphabet: a finite nonempty set of characters. E.g. {0, 1}, ASCII, Unicode
String (sentence, word): a finite sequence of characters, possibly empty.
Language: a (countable) set of strings, possibly empty.

3.2.2 Operations on Strings

i. concatenation
ii. exponentiation
   a. x^0 is the empty string ε.
   b. $x^i = x^{i-1}x$, for $i > 0$
iii. prefix, suffix, substring, subsequence

3.2.3 Operations on Languages

i. union
ii. concatenation
iii. exponentiation
   a. L^0 is { ε }, even when L is the empty set.

b. $L^i = L^{i-1}L$, for $i > 0$

iv. Kleene closure

a. $L^* = L^0 \cup L^1 \cup \dots$

v. Note that L^* always contains the empty string.

3.3 Regular Expressions (REs)

Regular expressions are a very convenient form of representing (possibly infinite) sets of strings, called *regular sets*. For example, the RE $(a|\ b)^*aa$ represents the infinite set {``*aa*'',``*aaa*'',``*baa*'',``*abaa*'', ··· }, which is the set of all strings with characters *a* and *b* that end in *aa*.

Many of today's programming languages use regular expressions to match patterns in strings. E.g., awk, flex, lex, java, javascript, perl, python.

3.3.1 Definition of a Regular Expression and the Language it Denotes

3.3.1.1 Basis

- ε is a regular expression that denotes { ε }.
- A single character *a* is a regular expression that denotes { *a* }.

3.3.1.2 Induction

Suppose *r* and *s* are regular expressions that denote the languages L(*r*) and L(*s*):

- $(r)|(s)$ is a regular expression that denotes $L(r) \cup L(s)$
- $(r)(s)$ is a regular expression that denotes $L(r)L(s)$
- $(r)^*$ is a regular expression that denotes $L(r)^*$
- (r) is a regular expression that denotes $L(r)$.

We can drop redundant parenthesis by assuming:

- the Kleene star operator * has the highest precedence and is left associative
- concatenation has the next highest precedence and is left associative
- the union operator | has the lowest precedence and is left associative

  E.g., under these rules r|s*t is interpreted as (r)|((s)*(t)).

3.3.1.3 Extensions of Regular Expressions

- Positive closure: *r*+ = *rr**
- Zero or one instance: *r*? = ε | *r*
- Character classes:
  a. [abc] = a | b | c
  b. [0-9] = 0 | 1 | 2 | … | 9

We can freely put parentheses around REs to denote the order of evaluation.

For example, (*a*| *b*)*c*. To avoid using many parentheses, we use the following rules: concatenation and alternation are associative (i.e. *ABC* means (*AB*)*C* and is equivalent to *A*(*BC*)), alternation is commutative (i.e., *A*| *B* = *B*| *A*), repetition is idempotent (i.e., $A^{**} = A^{*}$), and concatenation distributes over alternation (eg, (*a*| *b*)*c* = *ac*| *bc*).

For convenience, we can give names to REs so we can refer to them by their name. For example:

for – keyword	= *For*
Letter	= [*a* - *zA* - *Z*]
digit	= [0 - 9]
identifier	= *letter* (*letter* | *digit*)*
sign	= + | - | ε
integer	= *sign* (0 | [1 - 9]*digit**)
decimal	= *integer* . *digit**
real	= (*integer* | *decimal*) *E sign digit*$^{+}$

There is some ambiguity though: If the input includes the characters for8, then the first rule (for *for-keyword*) matches 3 characters (for), the fourth rule (for *identifier*) can match 1, 2, 3, or 4 characters, the longest being for8. To resolve this type of ambiguities, when there is a choice of rules; scanner generators choose the one that matches the maximum number of characters. In this case, the chosen rule is the one for *identifier* that matches 4 characters (for8). This disambiguation rule is called the *longest match rule*. If there are more than one rule that match the same maximum number of characters, the rule listed first is chosen. This is the *rule priority* disambiguation rule. For example, the lexical word for is taken as a *for-keyword* even though it uses the same number of characters as an identifier.

Today regular expressions come in many different forms.

a. The earliest and simplest are the Kleene regular expression

b. Awk and egrep extended grep's regular expressions with union and parentheses.
c. POSIX has a standard for Unix regular expressions.
d. Perl has an amazingly rich set of regular expression operators.
e. Python uses pcre regular expressions.

SELF-ASSESSMENT EXERCISE

i. What language is denoted by the following regular expressions?
   a. (a*b*)*
   b. a(a|b)*a
   c. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*
   d. a(ba|a)*
   e. ab(a|b*c)*bb*a

3.3.2 Lex Regular Expressions

The lexical analyser generators flex and lex use extended regular expressions to specify lexeme patterns making up tokens. The declarative language Lex has been widely used for creating many useful lexical analysis tools including lexers. The following symbols in Lex regular expressions have special meanings:

\ " . ^ $ [ ] * + ? { } | ( ) /

To turn off their special meaning, precede the symbol by \. Thus,

* matches *.
\\ matches \.

Examples of Lex regular expressions and the strings they match are:

1. "a.*b" matches the string a.*b.
2. . matches any character except a newline.
3. ^ matches the empty string at the beginning of a line.
4. $ matches the empty string at the end of a line.
5. [abc] matches an a, or a b, or a c.
6. [a-z] matches any lowercase letter between a and z.
7. [A-Za-z0-9] matches any alphanumeric character.
8. [^abc] matches any character except an a, or a b, or a c.
9. [^0-9] matches any nonnumeric character.
10. a* matches a string of zero or more a's.
11. a+ matches a string of one or more a's.
12. a? matches a string of zero or one a's.
13. a{2,5} matches any string consisting of two to five a's.
    (a) matches an a.

14. a/b matches an a when followed by a b.
15. \n matches a newline.
16. \t matches a tab.

Lex chooses the longest match if there is more than one match. E.g., ab* matches the prefix abb in abbc.

3.4 Tokens/Patterns/Lexemes/Attributes

A *token* is a pair consisting of a token name and an optional attribute value. e.g., <id, ptr to symbol table>, <=>

Some regular expressions corresponding to tokens:

- For the token keyword, we can say
- Keyword = BEGIN │ END │ IF│ THEN │ ELSE
- Identifier = letter (letter │ digit)*
- Constant = digit*
- Relop = <│<=│ <> │ > │ >=

A *pattern* is a description of the form that the lexemes making up a token in a source program may have. We will use regular expressions to denote patterns. e.g., identifiers in C: [_A-Za-z][_A-Za-z0-9]*

A lexeme is a sequence of characters that matches the pattern for a token, e.g.,

- identifiers: count, x1, i, position
- keywords: if
- operators: =, ==, !=, +=

An *attribute* of a token is usually a pointer to the symbol table entry that gives additional information about the token, such as its type, value, line number, etc.

3.5 Languages for Specifying Lexical Analyser

There are tools that can generate lexical analyzers. An example of such tools is LEX which is discussed in the next section. You are to read about other tools in your further reading.

3.5.1 Specifying a Lexical Analyser with Lex

Lex is a special-purpose programming language for creating programmes to process streams of input characters. Lex has been widely used for constructing lexical analysers. A Lex programme has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty. The translation rules are each of the form

```
pattern        {action}
```

Each pattern is a regular expression which may use regular definitions defined in the declarations section. Each action is a fragment of C-code. The auxiliary functions section starting with the second %% is optional. Everything in this section is copied directly to the file lex.yy.c and can be used in the actions of the translation rules.

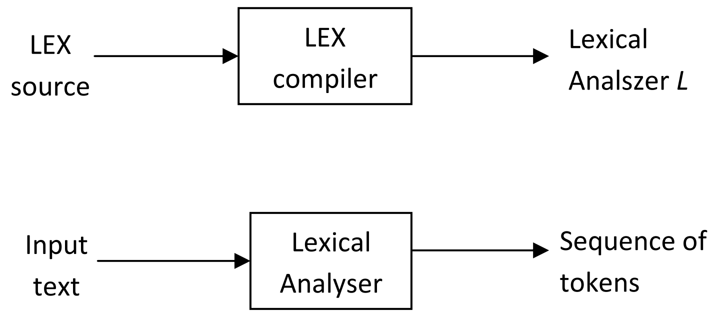


Fig. 1: The role of LEX

The input to the LEX compiler is called LEX source and the output of LEX compiler is called lexical analyser i.e. the LEX compiler generate lexical analyser.

3.5.2 LEX Source

LEX source is an input that states the characteristics of the lexical analyser for the language. In actual fact, it will define the tokens for the language. LEX source differs from language to language.

The LEX source programme consists of two parts: A sequence of auxiliary definitions followed by a sequence of translation rules.

1) **The auxiliary definitions:** are statements of the form:

$D_1 = R_1$
$D_2 = R_2$
.

.

$D_n = R_n$
Where each D_i is a distinct name and R_i is a regular expression whose symbols are chosen from the alphabets of the language

$D_i = \Sigma \cup \{ D_1, D_2, \ldots, D_{i-1}\}$ i.e. the characters of previously defined names (Note that what you have not defined earlier cannot be used later)

2) **The Translation Rules**: these are of the form:
$P_1\{A_1\}$
$P_2 \{A_2\}$
.

.

$P_m \{A_m\}$

Where each P_i is a regular expression called Pattern over the alphabet consisting of sigma ($\sum$) and the auxiliary definition names. The patterns described the form of the tokens.

Each A_i is a programme segment describing what action the lexical analyser should take when token P_i is found.

To create the lexical analyser '*L*', each of the A_is, must be compile into machine codes just like any other program written in the language of the A_is.

In summary, the lexical analyser *L* created by LEX behaves in the following manner:

- *L* reads its input one character at a time until it has found the largest prefix in the input which matches one of the regular expressions P_i .
- Once *L* has found that prefix, *L* removes it from the input and places it in a buffer called token. *L* then executes the actions A_i. After completing A_i, *L* returns control to the parser.
- When requested to, *L* repeats these series of actions on the remaining input.

3.5.3 Steps in lex implementation

1. Read input lang. spec
2. Construct NFA with epsilon-moves (Can also do DFA directly)
3. Convert NFA to DFA
4. Optimise the DFA
5. Generate parsing tables & code

3.5.4 Sample Lex programmes

Example 1: Lex programme to print all words in an input stream
The following Lex programme will print all alphabetic words in an input stream:

```
%%
[A-Za-z]+     { printf("%s\n", yytext); }
.|\n          { }
```

The pattern part of the first translation rule says that if the current prefix of the unprocessed input stream consists of a sequence of one or more letters, then the longest such prefix is matched and assigned to the Lex string variable yytext. The action part of the first translation rule prints the prefix that was matched. If this rule fires, then the matching prefix is removed from the beginning of the unprocessed input stream.

The dot in pattern part of the second translation rule matches any character except a newline at the beginning of the unprocessed input stream. The \n matches a newline at the beginning of the unprocessed input stream. If this rule fires, then the character of the beginning of the unprocessed input stream is removed. Since the action is empty, no output is generated.

Lex repeatedly applies these two rules until the input stream is exhausted.

Example 2: Lex programme to print number of words, numbers, and lines in a file

```
        int num_words = 0, num_numbers = 0, num_lines = 0;
word     [A-Za-z]+
number   [0-9]+
%%
{word}   {++num_words;}
{number} {++num_numbers ;}
\n       {++num_lines;}
.        { }
%%
int main()
{
  yylex();
  printf("# of words = %d, # of numbers = %d, # of lines = %d\n",
        num_words, num_numbers, num_lines );
}
```

Example 3: Lex programme for some typical programming language tokens

```
%{ /* definitions of manifest constants */
   LT, LE,
   IF, ELSE, ID, NUMBER, RELOP */

/* regular definitions */
delim   [ \t\n]
ws      {delim}+
letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}|{digit})*
number          {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws}    { }
if      {return(IF);}
else    {return(ELSE);}
{id}    {yylval = (int) installID(); return(ID);}
{number} {yylval = (int) installNum(); return(NUMBER);}
"<"     {yylval = LT; return(RELOP); }
"<="    {yylval = LE; return(RELOP); }
%%
int installID()
{
  /* function to install the lexeme, whose first character
     is pointed to by yytext, and whose length is yyleng,
     into the symbol table; returns pointer to symbol table
     entry */
}
```

```
int installNum() {
  /* analogous to installID */
}
```

3.5.5 Creating a Lexical Processor with Lex

Put lex programme into a file, say file.l.

Compile the lex programme with the command:

- lex file.l

This command produces an output file lex.yy.c.
Compile this output file with the C compiler and the lex library -ll:
gcc lex.yy.c -ll

The resulting a.out is the lexical processor.

3.5.6 Lex History

The initial version of Lex was written by Michael Lesk at Bell Labs to run on UNIX.

The second version of Lex with more efficient regular expression pattern matching was written by Eric Schmidt at Bell Labs.

Vern Paxson wrote the POSIX-compliant variant of Lex, called Flex, at Berkeley.

All versions of Lex use variants of the regular-expression pattern-matching technology described in section 3.3.2

Today, many versions of Lex use C, C++, C#, Java, and other languages to specify actions.

SELF- ASSESSMENT EXERCISE 2

i. Construct Lex-style regular expressions for the following patterns.
   a. All lowercase English words with the five vowels in order.
   b. All lowercase English words with exactly one vowel.
ii. Write a Lex programme that copies a file, replacing each nonempty sequence of whitespace consisting of blanks, tabs, and newlines by a single blank.

4.0 CONCLUSION

In this unit, you have been taken through the concept of regular expressions, and the languages that are used to automatically generate lexical analysers. In the next unit you will be taken through how to convert regular expressions to non-deterministic finite automata (NFA) and deterministic finite automata (DFA)

5.0 SUMMARY

In this unit, you learnt that:

- regular expressions are a very convenient form of representing (possibly infinite) sets of strings
- parentheses can be put around REs to denote the order of evaluation
- lexical analyser generators Flex and Lex use extended regular expressions to specify lexeme patterns making up tokens
- Lex chooses the longest match if there is more than one match
- a *pattern* is a description of the form that the lexemes making up a token in a source programme may have
- a lexeme is a sequence of characters that matches the pattern for a token
- an *attribute* of a token is usually a pointer to the symbol table entry that gives additional information about the token, such as its type, value, line number, etc
- Lex is a special-purpose programming language for creating programmes to process streams of input characters
- the input to the LEX compiler is called LEX source and the output of LEX compiler is called lexical analyser LEX source is an input that states the characteristics of the lexical analyser for the language
- LEX source differs from language to language
- the LEX source programme consists of two parts: A sequence of auxiliary definitions followed by a sequence of translation rules
- there are several tools for automatically generating lexical analysers of which LEX is one.

6.0 TUTOR-MARKED ASSIGNMENT

i. Construct Lex-style regular expressions for the following patterns.
   a. All lowercase English words beginning and ending with the substring "ad".

   b. All lowercase English words in which the letters are in strictly increasing alphabetic order.
   c. Strings of the form abxba where x is a string of a's, b's, and c's that does not contain ba as a substring.

ii. Write a Lex programme that converts a file of English text to "Pig Latin." Assume the file is a sequence of words separated by whitespace. If a word begins with a consonant, move the consonant to the end of the word and add "ay". (E.g., "pig" gets mapped into "igpay".) If a word begins with a vowel, just add "ay" to the end. (E.g., "art" gets mapped to "artay".)

iii. Describe a language (different from the one discussed in this course) for specifying Lexical analyser.

iv. How is the language described in question (3) above similar or different from LEX.

7.0 REFERENCES/FURTHER READING

Alfred, V. Aho *et al.* (2007). *Compilers: Principles, Techniques, and Tools*. Second Edition. Pearson Addison-Wesley.

Andrew, W. Appel (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Keith, D. Cooper & Linda Torczon (2004). *Engineering a Compiler*. Morgan Kaufmann.

Steven, S. Muchnick (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Michael, L. Scott (2009). *Programming Language Pragmatics*. Third Edition. Morgan Kaufman.

Robert, W. Sebesta (2010). *Concepts of Programming Languages*. Ninth Edition. Addison-Wesley.

UNIT 4 IMPLEMENTING A LEXICAL ANALYSER

CONTENTS

1.0 INTRODUCTION

In the previous unit you were taken through automatic generation of lexical analyser. The importance of regular expressions in the generation of lexical analysers was brought out.

In this unit, we will discuss finite state machines, which are the recognisers that recognise regular expressions and also how to convert REs into finite automata and vice versa.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define finite automata
- convert REs to NFA
- convert NFA to DFA.

3.0 MAIN CONTENT

3.1 Finite Automata

A recogniser for a language *L* is a programme that takes as input a string *x* and answers "yes" if *x* is a sentence of *L* and "no" otherwise. Clearly, the part of a lexical analyser that identifies the presence of a token on the input is a recogniser for the language defining that token.

Variants of finite automata are commonly used to match regular expression patterns.

3.1.1 Nondeterministic Finite Automaton (NFA)

A nondeterministic finite automaton (NFA) consists of:

- a finite set of states *S*
- an input alphabet consisting of a finite set of symbols Σ
- a transition function δ that maps $S \times (\Sigma \cup \{\varepsilon\})$ to subsets of *S*. This transition function can be represented by a transition graph in which the nodes are labelled by states and there is a directed edge labelled *a* from node *w* to node *v* if δ(*w*, *a*) contains *v*
- an initial state s_0 in *S*
- *F*, a subset of *S*, called the final (or accepting) states.

An NFA accepts an input string *x* if there is a path in the transition graph from the initial state to a final state that spells out *x*. The language defined by an NFA is the set of strings accepted by the NFA.

3.1.2 Deterministic Finite Automata (DFAs)

A deterministic finite automaton (DFA) is an NFA in which:

- there are no ε moves, and
- for each state *s* and input symbol *a* there is exactly one transition out of *s* labelled *a*.

Therefore, a DFA represents a finite state machine that recognises a RE. For example, the following DFA:

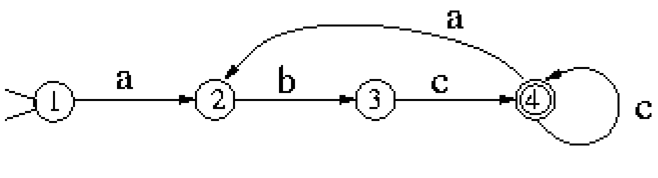


Fig. 1: DFA Recognising the String $(abc^+)^+$

recognises ($abc^+)^+$. A finite automaton consists of a finite set of states, a set of transitions (moves), one start state, and a set of final states (accepting states). In addition, a DFA has a unique transition for every state-character combination. For example, Figure 1 has 4 states, state 1 is the start state, and state 4 is the only final state.

A DFA accepts a string if starting from the start state and moving from state to state, each time following the arrow that corresponds the current input character, it reaches a final state when the entire input string is consumed. Otherwise, it rejects the string.

The Figure 1 represents a DFA even though it is not complete (i.e. not all state-character transitions have been drawn). The complete DFA is as in Figure 2 below.

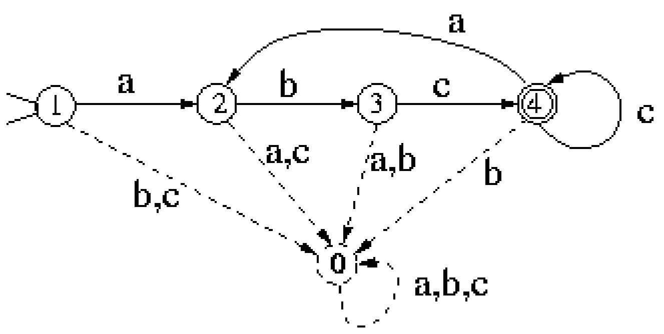


Fig. 2: Complete DFA Recognising the String ($abc^+)^+$

But it is very common to ignore state 0 (called the *error state*) since it is implied. (The arrows with two or more characters indicate transitions in case of any of these characters.) The error state serves as a black hole, which does not let you escape.

A DFA is represented by a *transition table T*, which gives the next state *T*[*s*, *c*] for a state *s* and a character *c*. For example, the *T* for the DFA above is:

	A	b	c
0	0	0	0
1	2	0	0
2	0	3	0
3	0	0	4
4	2	0	4

Suppose that we want to build a scanner for the REs:

$$for\text{-}keyword = For$$
$$Identifier \quad = [a - z][a - z0 - 9]^*$$

The corresponding DFA has 4 final states: one to accept the *for-keyword* and 3 to accept an identifier:

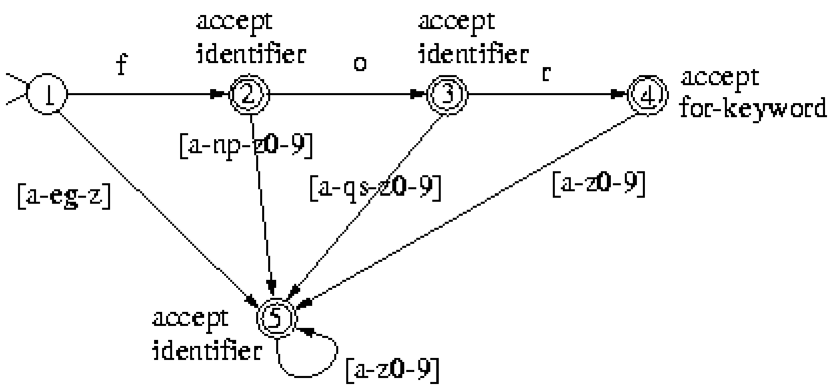


Fig. 3: DFA Recognising Identifier and for-keyword

(The error state is omitted again). Notice that for each state and for each character, there is a single transition.

A scanner based on a DFA uses the DFA's transition table as follows:

```
state = initial_state;
current_character = get_next_character();
while ( true )
{   next_state = T[state,current_character];
   if (next_state == ERROR)
     break;
   state = next_state;
   current_character = get_next_character ();
   if ( current_character == EOF )
     break;
};
if ( is_final_state(state) )
  `we have a valid token'
else `report an error'
```

This programme does not explicitly take into account the longest match disambiguation rule since it ends at EOF. The following programme is more general since it does not expect EOF at the end of token but still uses the longest match disambiguation rule.

```
state = initial_state;
final_state = ERROR;
```

```
current_character = get_next_character();
while ( true )
{   next_state = T[state,current_character];
    if (next_state == ERROR)
      break;
    state = next_state;
    if ( is_final_state(state) )
      final_state = state;
    current_character = get_next_character();
    if (current_character == EOF)
      break;
};
if ( final_state == ERROR )
   `report an error'
else if ( state != final_state )
   `we have a valid token but we need to backtrack
        (to put characters back into the input stream)'
else `we have a valid token'
```

Is there any better (more efficient) way to build a scanner out of a DFA? Yes! We can hardwire the state transition table into a programme (with lots of gotos). You've learned in your programming language course never to use gotos. But here we are talking about a programme generated automatically, which no one needs to look at. The idea is the following. Suppose that you have a transition from state s_1 to s_2 when the current character is *c*. Then you generate the programme:

```
s1: current_character = get_next_character ();
    ...
    if ( current_character == 'c' )
      goto s2;
    ...
s2: current_character = get_next_character();
```

3.2 Equivalence of Regular Expressions and Finite Automata

Regular expressions and finite automata define the same class of languages, namely the regular sets. In Computer Science Theory we showed that:

- every regular expression can be converted into an equivalent NFA using the McNaughton-Yamada-Thompson algorithm.
- every NFA can be converted into an equivalent DFA using the subset construction.
- every finite automaton can be converted into a regular expression using Kleene's algorithm.

3.3 Converting a Regular Expression to an NFA

The task of a scanner generator, such as JLex, is to generate the transition tables or to synthesise the scanner programme given a scanner specification (in the form of a set of REs). So it needs to convert REs into a single DFA. This is accomplished in two steps: first it converts REs into a non-deterministic finite automaton (NFA) and then it converts the NFA into a DFA.

An NFA, as earlier stated, is similar to a DFA but it also permits multiple transitions over the same character and transitions over ε. In the case of multiple transitions from a state over the same character, when we are at this state and we read this character, we have more than one choice; the NFA succeeds if at least one of these choices succeeds. The ε-transition does not consume any input characters, so you may jump to another state for free.

Clearly DFAs are a subset of NFAs. But it turns out that DFAs and NFAs have the same expressive power. The problem is that when converting a NFA to a DFA we may get an exponential blow-up in the number of states.

We will first learn how to convert a RE into a NFA. This is the easy part. There are only 5 rules, one for each type of RE:

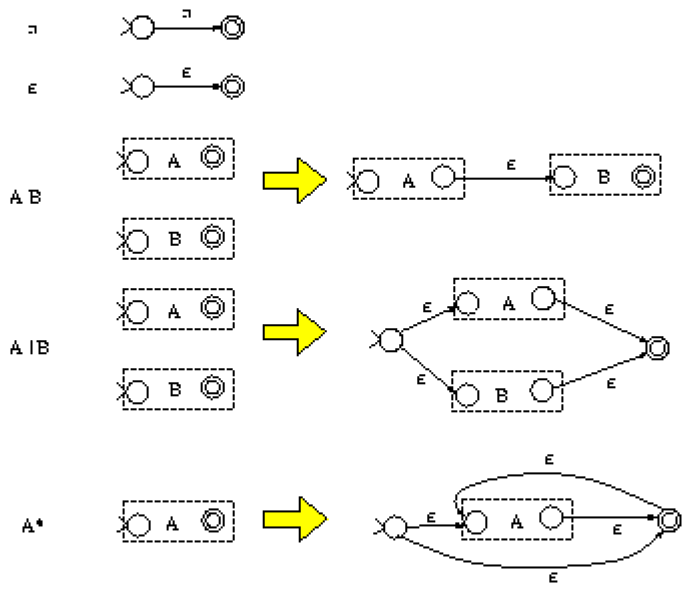


Fig. 4: NFA for each RE Characteristics

As it can been shown inductively, the above rules construct NFAs with only one final state. For example, the third rule indicates that, to construct the NFA for the RE *AB*, we construct the NFAs for *A* and *B*, which are represented as two boxes with one start state and one final state for each box. Then the NFA for *AB* is constructed by connecting the final state of *A* to the start state of *B* using an empty transition.

For example, the RE (*a*| *b*)*c* is mapped to the following NFA:

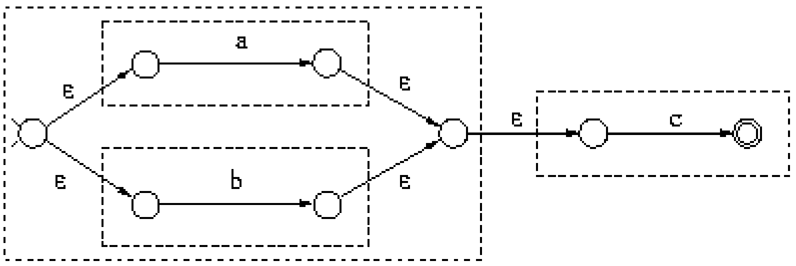


Fig. 5: NFA for the RE (a|b)c

Theorem 3.1: Any regular expression can be converted to a finite automaton (FA) or recogniser or finite state machine usually a Non-deterministic one (NFA) to generate the possibilities in that transition.

A better way to convert a regular expression to a recogniser is to construct a generalised transition diagram (TD) from the expression. The TD is usually a non-deterministic one but before you can program it, you have to convert it to a deterministic one. To turn a collection of TDs into a programme, we construct a segment of code for each state. The first step, in the code for any state is to obtain the next character from input buffer. So for this purpose, we use GETCHAR

The finite state diagram for the regular expression l (l│d)*:

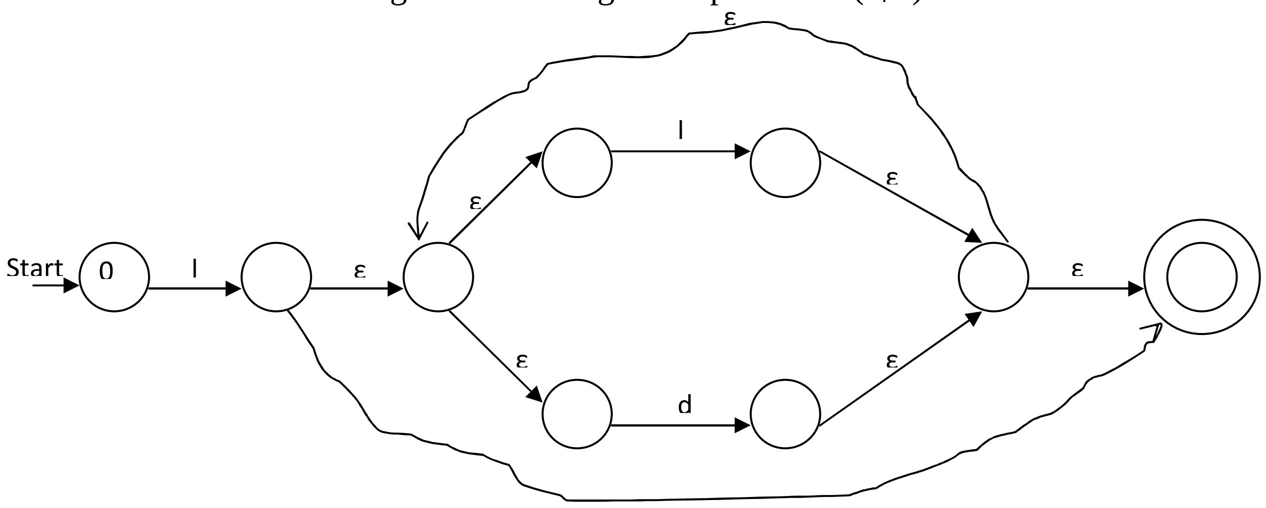


Fig. 6: NFA Recognising Identifier

The algorithm below can be used to convert the TD above into a finite state automaton.

Algorithm:

```
State 0: C:= GETCHAR()
         If LETTER(C) then go to state 1
         Else FAIL ()

State 1: C: = GETCHAR ()
         If LETTER(C) or DIGIT(C) then go to state 1
         Else if DELIMITER(C) then go to state 2
         Else FAIL ()
```

SELF -ASSESSMENT EXERCISE

i. Define finite automata.
ii. Distinguish between NFA and DFA.
iii. Convert the following REs into NFAs:
   a. (a*b*)*
   b. a(a|b)*a
   c. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*
   d. a(ba|a)*
   e. ab(a|b*c)*bb*a

3.4 Converting a Regular Expression into a Deterministic Finite Automaton

To convert a RE into a DFA, you first convert the RE into an NFA as discussed in section 3.3. The next step is to convert the NFA to a DFA (called *subset construction*). Suppose that you assign a number to each NFA state. The DFA states generated by subset construction have sets of numbers, instead of just one number. For example, a DFA state may have been assigned the set {5, 6, 8}. This indicates that arriving to the state labelled {5, 6, 8} in the DFA is the same as arriving to the state 5, the state 6, or the state 8 in the NFA when parsing the same input (Recall that a particular input sequence when parsed by a DFA, leads to a unique state, while when parsed by a NFA it may lead to multiple states).

First, we need to handle transitions that lead to other states for free (without consuming any input). These are the εtransitions. We define the *closure* of a NFA node as the set of all the nodes reachable by this node using zero, one, or more εtransitions. For example, the closure of node 1 in the left Figure 7 below

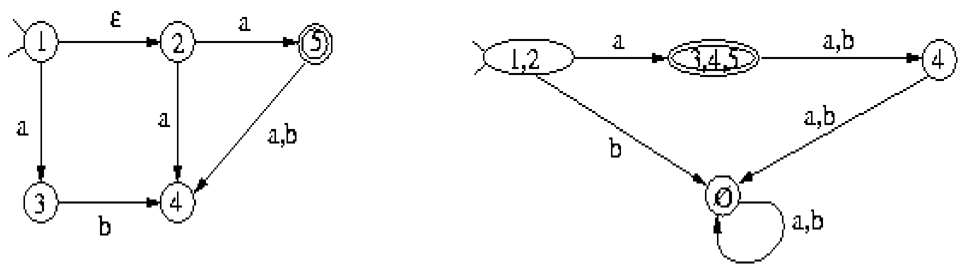


Fig. 7: Closure of NFA

- is the set {1, 2}. The start state of the constructed DFA is labelled by the closure of the NFA start state. For every DFA state labelled by some set {s_1,..., s_n} and for every character *c* in the language alphabet, you find all the states reachable by s_1, s_2, ..., or s_n using *c* arrows and you union together the closures of these nodes. If this set is not the label of any other node in the DFA constructed so far, you create a new DFA node with this label. For example, node {1, 2} in the DFA above has an arrow to a {3, 4, 5} for the character *a* since the NFA node 3 can be reached by 1 on *a* and nodes 4 and 5 can be reached by 2. The *b* arrow for node {1, 2} goes to the error node which is associated with an empty set of NFA nodes.

The following NFA recognises (*a*| *b*)*(*abb* | *a*$^+$*b*), even though it was not constructed with the above RE-to-NFA rules. It has the following DFA:

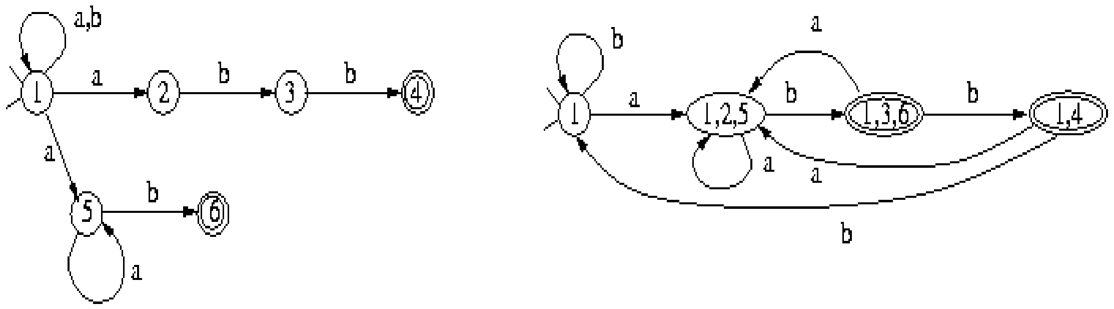


Fig. 8: NFA Recognising (a| b)*(abb | a$^+$b)

SELF-ASSESSMENT EXERCISE 2

i. Convert the NFAs in question (3) of Self Assessment test 1 into DFAs
ii. Construct an NFA to accept the string *aa*|bb**
iii. Convert the NFA drawn in question (2) above to DFA

4.0 CONCLUSION

In this unit, you have been taken through the concept of finite automata, equivalence of REs and finite automata and also how to convert REs into NFAs and subsequently DFAs.

5.0 SUMMARY

In this unit, you learnt that:

- a DFA represents a finite state machine that recognises a RE
- a finite automaton consists of a finite set of states, a set of transitions (moves), one start state, and a set of final states (accepting states). In addition, a DFA has a unique transition for every state-character combination
- to covert a RE into a DFA, you will first have to convert it into a non-deterministic finite automaton (NFA) and then convert the NFA into a DFA.
- An NFA is similar to a DFA but it also permits multiple transitions over the same character and transitions over ε
- there are several tools for automatically generating lexical analysers of which LEX is one.

6.0 TUTOR-MARKED ASSIGNMENT

i. Write down deterministic finite automata for the following regular expressions:
   a. (a*b*)*
   b. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*
   c. a(ba|a)*
   d. ab(a|b*c)*bb*a
ii. Construct a deterministic finite automaton that will recognise all strings of 0's and 1's representing integers that are divisible by 3. Assume the empty string represents 0.
iii. Use the McNaughton-Yamada-Thompson algorithm to convert the regular expression a (a|b)*a into a nondeterministic finite automaton.
iv. Convert the NFA of (3) into a DFA.
v. Minimise the number of states in the DFA of (4).

7.0 REFERENCES/FURTHER READING

Aho, A. V. & Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley Publishing Company. ISBN 0-201-00022-9.

Alfred, V. Aho *et al.* (2007). *Compilers: Principles, Techniques, and Tools*. Second Edition. Wesley: Pearson Addison.

Andrew, W. Appel (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Davis, Martin E., Sigal, Ron & Weyuker, Elaine J. (1994). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Boston: Academic Press, Harcourt, Brace. pp. 327. ISBN 0122063821.

John, E. Hopcroft & Jeffrey, D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading Massachusetts :Addison-Wesley Publishing. ISBN 0-201-029880-X.

Keith, D. Cooper& Linda, Torczon (2004). *Engineering a Compiler*. Morgan Kaufmann.

Michael, L. Scott (2009). *Programming Language Pragmatics*. Third Edition. Morgan Kaufman.

Robert, W. Sebesta (2010). *Concepts of Programming Languages*. Ninth Edition. Wesley: Addison.

Steven, S. Muchnick (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

MODULE 3 SYNTAX ANALYSIS

UNIT 1 CONTEXT-FREE GRAMMAR

CONTENTS

1.0 INTRODUCTION

In the previous module, you have been taken through the lexical analysis phase of a compiler. The module showed you that the lexical structure of tokens could be specified by regular expressions and that from a regular expression you could automatically construct a lexical analyser to recognise the tokens denoted by the expression. Module 3 will give similar treatment of the next phase of a compiler, which is the syntax analysis phase. For the syntactic specification of a programming language, we shall use a notation called a context-free grammar (grammar, for short), which is also called BNF (Backus-Naur Form) description.

In this first unit of the third module, you will be exposed to grammars, how a grammar defines a language, and what features of programming languages can, and cannot, be specified by context-free grammars.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define context-free grammars (CFGs)
- define and state the roles of a parser
- describe the concept of ambiguity
- generate parse trees for sentences
- verify grammars.

3.0 MAIN CONTENT

3.1 The Parser

The Parser takes tokens from lexer (scanner or lexical analyser) and builds a parse tree. The parser has two basic functions. It checks that the tokens appearing in its input, which is the output of the lexical analyser, occur in patterns that are permitted by the specification for the source language. It also imposes on the token a tree-like structure that is used by the subsequent phases of the compiler.

3.1.1 Role of the Parser

- The parser reads the sequence of tokens generated by the lexical analyser
- It verifies that the sequence of tokens obeys the correct syntactic structure of the programming language by generating a parse tree implicitly or explicitly for the sequence of tokens
- It enters information about tokens into the symbol table
- It reports syntactic errors to the user.

3. 2 Context-Free Grammars (CFG's)

CFG's are very useful for representing the syntactic structure of programming languages. A CFG is sometimes called Backus-Naur Form (BNF). It consists of

1. A finite set of terminal symbols,
2. A finite nonempty set of nonterminal symbols,
3. One distinguished nonterminal called the start symbol, and
4. A finite set of rewrite rules, called productions, each of the form $A \rightarrow \alpha$ where A is a nonterminal and α is a string (possibly empty) of terminals and nonterminals.

A context- free grammar is formally represented by the tuple:
CFG = (N, T, P, s)

where: N and T are finite sets of nonterminals and terminals under the assumption that N and T are disjoint
P is a finite set of productions such that each production is of the form: $A \rightarrow \alpha$ (where A is a nonterminal and α is a string of symbols)
s is the start symbol

3.2.1 Notational Conventions

Terminals are usually denoted by:

- lower-case letters early in the alphabet: *a*, *b*, *c*;
- operator symbols: +, -, *, etc.;
- punctuation symbols: (,), {, }, ; etc.;
- digits: 0, 1, 2, ..., 9;
- boldface strings: **if**, **else**, etc.;

Nonterminals are usually denoted by:

- upper-case letters early in the alphabet: A, B, C;
- the letter S representing the start symbol;
- lower-case italic names: *expr*, *stmt*, etc.;

Grammar symbols, that is either terminals or nonterminals, are represented by upper-case letters late in the alphabet: X, Y, Z

Strings of Terminals only are represented by lower-case letters late in the alphabet: *u*, *v*, *w* ... *z*

Productions are represented in the following way: $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$ etc.

Alternatives in productions are represented by $A \rightarrow \alpha_1 \mid \alpha_2$ etc.
Consider the context-free grammar G with the productions

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

- The terminal symbols are the alphabet from which strings are formed. In this grammar the set of terminal symbols is {id, +, *, (,) }. The terminal symbols are the token names.
- The nonterminal symbols are syntactic variables that denote sets of strings of terminal symbols. In this grammar the set of nonterminal symbols is { E, T, F}.
- The start symbol is E.

3.3 Derivations and Parse Trees

3.3.1 Derivations

The central idea to how a CFG define a language is that production may be applied repeatedly to expand the nonterminals in a string of nonterminals and terminals. For example, consider the following grammar for arithmetic expressions:

$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid$ **id** **.............** **(1)**

The nonterminal E is an abbreviation for expression. The production $E \rightarrow -E$ signifies that an expression preceded by a minus sign is also an expression. This production can be used to generate more complex expressions from simpler expressions by allowing you to replace any instance of an E by -E. In the simplest case you can replace a single E by -E. This action can be described by writing:

$E \Rightarrow -E$

which is read as “E derives -E". The production $E \rightarrow (E)$ tells us that we could also replace one instance of an E in any string of grammar symbols by (E); e.g.

$E * E \Rightarrow (E) * E$ or $E * E \Rightarrow E * (E)$.

You can take a single E and repeatedly apply productions in any order to obtain a sequence of replacements. For example,

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

Such sequence of replacements is called a derivation of – (id) from E. This derivation provides a proof that one particular instance of an expression is the string –(id).

In a more abstract setting, we say that $\alpha A\beta \rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production and α and β are arbitrary strings of grammar symbols. If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$, we say α_1 derives α_n. The symbol $\Rightarrow$ means "derives in one step". Often we wish to say "derives in zero or more steps". For this purpose we can use the symbol $\Rightarrow^*$. Therefore:

- $\alpha \Rightarrow^* \alpha$ for any string α, and
- If $\alpha \Rightarrow^* \beta$ and $\alpha \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$

Similarly, we use $\alpha \Rightarrow+ \beta$ to mean "α derives β in a derivation of one or more steps".

Given a context-free grammar G with start symbol S we can use the $\Rightarrow^*$ relation to define *L*(G), the *language generated* by G. Strings in *L*(G) may consists of all strings of terminal symbols that can be derived from the start symbol of G. We say a string of terminals ω is in *L*(G) if and only if $S \Rightarrow^* \omega$. The string w is called a *sentence* of G. If $S \Rightarrow^+ \alpha$, where α may contain nonterminals, then we say α is a *sentential form* of G.

Example 1:
The string -(id + id) is a sentence of grammar (1) above because
$E \Rightarrow -E \Rightarrow \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$

The strings E, -E, -(E), …, -(id + id) appearing in this derivation are all sentential forms of this grammar.

To understand how certain parsers work, we need to consider derivation in which only the leftmost nonterminal in a sentential form is replaced at each step. Such derivations are termed leftmost. Analogously, rightmost derivations are those in which the rightmost nonterminal is replaced at each step.

3.3.1.1 Leftmost Derivations

Derivations in which only the leftmost nonterminal in any sentential form is replaced at each step are called *leftmost*: $\alpha \Rightarrow_{lm} \beta$.

if $w\, A\, \gamma \Rightarrow w\, \delta\, \gamma$ then $\alpha \Rightarrow_{lm} \beta$.

where $A \rightarrow \delta$ is a production and w is a string of terminals, and
γ is a string of grammar symbols.

To emphasise the fact that α derives β by a leftmost derivation may be written:

$\alpha \Rightarrow_{lm}^{*} \beta$.

If $S \Rightarrow_{lm}^{*} \alpha$ then α is a *left-sentential form* of the grammar.

Example 2: Consider the grammar G with the following productions:
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

A leftmost derivation expands the leftmost nonterminal in each sentential form:

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \\
&\Rightarrow id + T \\
&\Rightarrow id + T * F \\
&\Rightarrow id + F * F \\
&\Rightarrow id + id * F \\
&\Rightarrow id + id * id
\end{aligned}
$$

3.3.1.2 Rightmost Derivations

Derivations in which only the rightmost nonterminal in any sentential form is replaced at each step are called *rightmost*: $\alpha \Rightarrow_{rm} \beta$.

if $\gamma A w \Rightarrow \gamma \delta w$ then $\alpha \Rightarrow_{rm} \beta$.
where $A \rightarrow \delta$ is a production and w is a string of terminals, and γ is a string of grammar symbols.

To emphasize the fact that α derives β by a rightmost derivation may be written:

$\alpha \Rightarrow_{rm}^{*} \beta$.

If $S \Rightarrow_{rm}^{*} \alpha$ then α is a *right-sentential form* of the grammar.

Example 3: Consider the grammar G with the following productions:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

A rightmost derivation expands the rightmost nonterminal in each sentential form:

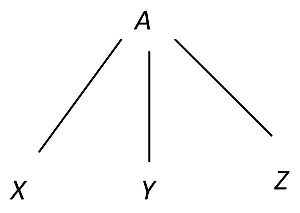
$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow E + T * F \\
&\Rightarrow E + T * id \\
&\Rightarrow E + F * id \\
&\Rightarrow E + id * id \\
&\Rightarrow T + id * id \\
&\Rightarrow F + id * id \\
&\Rightarrow id + id * id
\end{aligned}
$$

Note that these two derivations have the same parse tree.

3.3.2 Parse Trees

Parse tree is a graphical representation for derivation that filters out the choice regarding replacement. It has the important purpose of making explicit the hierarchical syntactic structure of sentences that is implied by the grammar.

Each interior node of the parse tree is labelled by some nonterminal A, and the children of the node are labelled, from left to right, by the symbol in the right side of the production by which this A was replaced in the derivation. For example, if $A \rightarrow XYZ$ is a production used at some step of a derivation, then the parse tree for that derivation will have the subtree illustrated below:



The leaves of the parse tree are labelled by terminals or nonterminals and, read from left to right. They constitute a sentential form, called the *yield* or *frontier* of the tree. For example, the parse tree for –(id + id) implied by the derivation of Example 1 above is as shown in Figure 1 below:

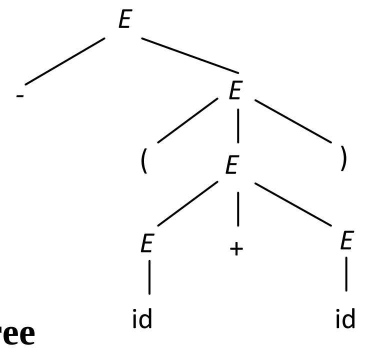


Fig. 1: Parse Tree

SELF-ASSESSMENT EXERCISE 1

i. Let G be the grammar S → a S b S | b S a S | ε.
ii. What language is generated by this grammar?
iii. Draw all parse trees for the sentence abab

3.4 Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous i.e. An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence. For certain types of parsers, it desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

Consider the context-free grammar G with the productions

E → E + E | E * E | (E) | id

This grammar has the following leftmost derivation for id + id * id

E ⇒ E + E
⇒ id + E
⇒ id + E * E
⇒ id + id * E
⇒ id + id * id

This grammar also has the following leftmost derivation for id + id * id

E ⇒ E * E
⇒ E + E * E
⇒ id + E * E
⇒ id + id * E
⇒ id + id * id

These derivations have different parse trees.

A grammar is *ambiguous* if there is a sentence with two or more parse trees.

The problem is that the grammar above does not specify

- the precedence of the + and * operators, or
- the associativity of the + and * operators

However, the grammar in section (3.2) generates the same language and is unambiguous because it makes * of higher precedence than +, and makes both operators left associative.

A context-free language is *inherently ambiguous* if it cannot be generated by any unambiguous context-free grammar.

The context-free language $\{ a^m b^m a^n b^n \mid m > 0 \text{ and } n > 0\} \cup \{ a^m b^n a^n b^m \mid m > 0 \text{ and } n > 0\}$ is inherently ambiguous.

Most natural languages are inherently ambiguous but no programming languages are inherently ambiguous.

Unfortunately, there is no algorithm to determine whether a CFG is ambiguous; that is, the problem of determining whether a CFG is ambiguous is undecidable.

We can, however, give some practically useful sufficient conditions to guarantee that a CFG is unambiguous.

3.5 Left Recursion

"A grammar is left-recursive if we can find some non-terminal A which will eventually derive a sentential form with itself as the left-symbol."

3.5.1 Immediate Left Recursion

Immediate left recursion occurs in rules of the form

$$A \to A\alpha \mid \beta$$

where α and β are sequences of nonterminals and terminals, and β doesn't start with *A*. For example, the rule

$$Expr \to Expr + Term$$

is immediately left-recursive. The *recursive descent parser* for this rule might look like:

```
function Expr()
{
   Expr();  match('+');  Term();
}
```

and a recursive descent parser would fall into infinite recursion when trying to parse a grammar which contains this rule.

3.5.2 Indirect Left Recursion

Indirect left recursion in its simplest form could be defined as:

$$A \rightarrow B\alpha \mid C$$
$$B \rightarrow A\beta \mid D,$$

possibly giving the derivation $A \Rightarrow B\alpha \Rightarrow A\beta\alpha \Rightarrow \ldots$

More generally, for the nonterminals $A_0, A_1, \ldots, A_n$, indirect left recursion can be defined as being of the form:

$$A_0 \rightarrow A_1\alpha_1 \mid \ldots$$
$$A_1 \rightarrow A_2\alpha_2 \mid \ldots$$
$$\vdots$$
$$A_n \rightarrow A_0\alpha_{n+1} \mid \ldots$$

where $\alpha_1, \alpha_2, \ldots, \alpha_n$are sequences of nonterminals and terminals.

3.5.3 Removing Left Recursion

3.5.3.1 Removing Immediate Left Recursion

The general algorithm to remove immediate left recursion follows. Several improvements to this method have been made, including the ones described in "Removing Left Recursion from Context-Free Grammars", written by Robert C. Moore. For each rule of the form

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

where:
A is a left-recursive nonterminal
α is a sequence of nonterminals and terminals that is not null ($\alpha \neq \epsilon$)
β is a sequence of nonterminals and terminals that does not start with A.
replace the A-production by the production:

$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A'$$

And create a new nonterminal

$$A' \rightarrow \epsilon \mid \alpha_1 A' \mid \ldots \mid \alpha_n A'$$

This newly created symbol is often called the "tail", or the "rest".
As an example, consider the rule

$$Expr \rightarrow Expr + Expr \mid Int \mid String$$

This could be rewritten to avoid left recursion as

$$Expr \rightarrow Int\, ExprRest \mid String\, ExprRest$$
$$ExprRest \rightarrow \epsilon \mid +\ Expr\, ExprRest$$

The last rule happens to be equivalent to the slightly shorter form
$ExprRest \rightarrow \epsilon \mid +\ Expr$

3.5.3.2 Removing Indirect Left Recursion

If the grammar has no ε-productions (no productions of the form $A \rightarrow \ldots |\epsilon| \ldots$) and is not cyclic (no derivations of the form $A \Rightarrow \ldots \Rightarrow A$ for any nonterminal A), this general algorithm may be applied to remove indirect left recursion :

Arrange the nonterminals in some (any) fixed order $A_1, \ldots A_n$.

```
for i = 1 to n {
for j = 1 to i – 1 {

let the current $A_j$ productions be

$A_j \rightarrow \delta_1| \ldots |\delta_k$

replace each production $A_i \rightarrow A_j\gamma$ by

$A_i \rightarrow \delta_1\gamma| \ldots |\delta_k\gamma$

remove direct left recursion for $A_i$
}
}
```

The left-recursive pair of productions

$A \rightarrow A\alpha \mid \beta$

could be replaced by the non-left-recursive productions

$A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \in$

without changing the set of strings derivable from A.

The left-recursive productions

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid ... \mid \beta_n$

could be replaced by the non-left-recursive productions

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$ and
$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \in$

3.5.4 Algorithm for Elimination of Left Recursion

```
Initialise: Arrange the nonterminals in some order $A_1, A_2, ..., A_n$
    Repeat:   for $i$ := 1 to $n$ do
                  for $j$ := 1 to $i$ - 1 do
                      replace any production the form $A_i \rightarrow A_j\gamma$
                      by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$
                      where Aj $\rightarrow \delta_1 \mid \delta_2 \mid ... \mid \delta_k$
                  end
                  eliminate the immediate left-recursion among the $A_i$
                  productions ( $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \in$ )
```

Example 4: $S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \in$

for $i = 1$ nothing happens

for $i = 2$ we obtain $A \rightarrow Ac \mid Aad \mid bd \mid \in$

after eliminating the immediate left recursions

$S \rightarrow Aa \mid b$
$A \rightarrow bd\, A' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \in$

3.6 Verifying Grammars

Given a grammar G, we may want to prove that *L*(G) = L for some given language L. To do this, we need to show that every string generated by G is in L, and every string in L can be generated by G.

Consider the grammar with the productions: S → (S) S | ε.
Let L be the set of strings of balanced parentheses.

We can use induction on the number of steps in a derivation to show that every string generated from S is balanced, thus showing that every string generated by the grammar is in L.

We can use induction on the length of a string to show that every balanced string can be generated from S, thus showing that every balanced string is generated by the grammar.

SELF ASSESSMENT EXERCISE

i. Let G be the grammar $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \varepsilon$.
ii. Is this grammar ambiguous?

4.0 CONCLUSION

In this unit, you have been taken through the concept of context-free grammars, derivations and parse trees. In the rest of this module, you will learn about parsing techniques.

5.0 SUMMARY

In this unit, you learnt that:

- a leftmost derivation expands the leftmost nonterminal in each sentential form
- a rightmost derivation expands the rightmost nonterminal in each sentential form
- parse tree is a graphical representation for derivation that filters out the choice regarding replacement
- the leaves of the parse tree are labelled by terminals or nonterminals and, read from left to right and they constitute a sentential form, called the yield or frontier of the tree
- an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence.

6.0 TUTOR-MARKED ASSIGNMENT

i. Let G be the grammar $S \rightarrow a\ S\ b \mid \varepsilon$. Prove that L (G) = $\{ a^n b^n \mid n \geq 0 \}$.
ii. Consider a sentence of the form id + id + ... + id where there are n plus signs. Let G be the grammar in section (3.4) above. How many parse trees are there in G for this sentence when n equals
   a. 1
   b. 2
   c. 3

d. 4
e. *m*?

iii. Consider the grammar in section (3.6) above. How many sentences does this grammar generate having *n* left parentheses where *n* equals
a. 1
b. 2
c. 3
d. 4
e. *m*?

7.0 REFERENCES/FURTHER READING

Aho, Alfred V. *et al* (2007).*Compilers: Principles, Techniques, and Tools*. Second Edition. Wesley:Pearson Addison.

Appel, Andrew W. (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press

Cooper, Keith D. & Torczon, Linda(2004). *Engineering a Compiler*. Morgan Kaufmann.

Muchnick, Steven S. (1997).*Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Scott, Michael L. (2009). *Programming Language Pragmatics*. Third edition, Morgan Kaufman.

Sebesta, Robert W. (2010). *Concepts of Programming Languages*. Ninth edition. Wesley: Pearson Addison.

UNIT 2 BOTTOM-UP PARSING TECHNIQUES

CONTENTS

1.0 INTRODUCTION

In unit 1 of this module, you have been exposed to context free grammars, how a grammar defines a language, and what features of programming languages can, and cannot, be specified by context-free grammars. In this unit, you are going to learn about parsing techniques. There are various kinds of parsing techniques and these can be broadly categorised into two viz: top-down and bottom-up parsing techniques.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define parsing techniques
- distinguish between top-down and bottom-up parsing techniques
- describe shift reduce parsing
- define handle
- analyse an input string using shift-reduce parsing.

3.0 MAIN CONTENT

3.1 Parsing Techniques

There are two types of Parsing

a. Top-down Parsing (start from start symbol and derive string) A Top-down parser builds a parse tree by starting at the root and working down towards the leaves. It is easy to generate by hand. Examples are Recursive-descent parser and Predictive parser.

b. Bottom-up Parsing (start from string and reduce to start symbol) A bottom-up parser builds a parser tree by starting at the leaves and working up towards the root. Bottom-up parsing is characterised by the following:
   - It is not easy to handle by hands, usually compiler-generating software generate bottom-up parser
   - It handles larger class of grammar

Example is LR parsers, simple precedence parser, operator precedence parsers, etc.

3.2 Bottom-Up Parsing

In **programming language** compilers, bottom-up parsing is a parsing method that works by identifying terminal symbols first, and combines them successively to produce non-terminals. The productions of the parser can be used to build a parse tree of a programme written in human-readable source code that can be compiled to assembly language or pseudo code.

3.2.1 Types of Bottom-up Parsers

The common classes of bottom-up parsers are:

- LR parser (you will learn more about this in unit 5 of this module)
  a. LR(0) - No lookahead symbol
  b. SLR(1) - Simple with one lookahead symbol
  c. LALR (1) - Lookahead bottom up, not as powerful as full LR (1) but simpler to implement. YACC deals with this kind of grammar.
  d. LR (1) - Most general grammar, but most complex to implement.
  e. LR(*k*) - (where *k* is a positive integer) indicates an LR parser with *k* lookahead symbols; while grammars can be designed that require more than 1 lookahead, practical grammars try to avoid this because increasing *k* can

theoretically require exponentially more code and data space (in practice, this may not be as bad). Also, the class of LR (*k*) languages is the same as that of LR (1) languages.

- Precedence parsers (you will learn more about this in unit 5 of this module)
  a. Simple precedence parser
  b. Operator-precedence parser
  c. Extended precedence parser.

3.2.2 Shift-Reduce Parsers

The most common bottom-up parsers are the shift-reduce parsers. These parsers examine the input tokens and either shift (push) them onto a stack or reduce elements at the top of the stack, replacing a right-hand side by a left-hand side.

3.2.2.1Action Table

Often an action (or parse) table is constructed which helps the parser determine what to do next. The following is a description of what can be held in an action table.

Actions

a. Shift - push token onto stack
b. Reduce - remove handle from stack and push on corresponding nonterminal
c. Accept - recognise sentence when stack contains only the distinguished symbol and input is empty
d. Error - happens when none of the above is possible; means original input was not a sentence.

3.2.2.2Shift and Reduce

A shift-reduce parser uses a stack to hold the grammar symbols while awaiting reduction. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the non-terminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input.

The parser is a stack automaton which is in one of several discrete states. In reality, in the case of LR parsing, the parse stack contains states, rather than grammar symbols, as you will learn in unit 5 of this module. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

3.2.2.3 Algorithm: Shift-Reduce Parsing

Step 1: Start with the sentence to be parsed as the initial sentential form
Step 2: Until the sentential form is the start symbol do:

a. Scan through the input until we recognise something that corresponds to the RHS of one of the production rules (this is called a handle)
b. Apply a production rule in reverse; i.e., replace the RHS of the rule which appears in the sentential form with the LHS of the rule (an action known as a reduction)

In step 2(a) above we are "shifting" the input symbols to one side as we move through them; hence a parser which operates by repeatedly applying steps 2(a) and 2(b) above is known as a shift-reduce parser.

A shift-reduce parser is most commonly implemented using a stack, where we proceed as follows:

- start with an empty stack
- a "shift" action corresponds to pushing the current input symbol onto the stack
- a "reduce" action occurs when we have a handle on top of the stack. To perform the reduction, we pop the handle off the stack and replace it with the nonterminal on the LHS of the corresponding rule. This is referred to as handle pruning.

Example 1

Take the grammar:

```
 Sentence   --> NounPhrase VerbPhrase
 NounPhrase --> Art Noun
 VerbPhrase --> Verb | Adverb Verb
 Art        --> the | a | ...
 Verb       --> jumps | sings | ...
 Noun       --> dog | cat | ...
```

And the input:
the dog jumps

Then the bottom up parsing is:

Stack	**Input Sequence**	**Action**
()	(the dog jumps)	
(the)	(dog jumps)	SHIFT word onto stack
(Art)	(dog jumps)	REDUCE using grammar rule
(Art dog)	(jumps)	SHIFT..
(Art Noun)	(jumps)	REDUCE..
(NounPhrase)	(jumps)	REDUCE
(NounPhrase jumps)	()	SHIFT
(NounPhrase Verb)	()	REDUCE
(NounPhrase VerbPhrase)	()	REDUCE
(Sentence)	()	SUCCESS

Example 2

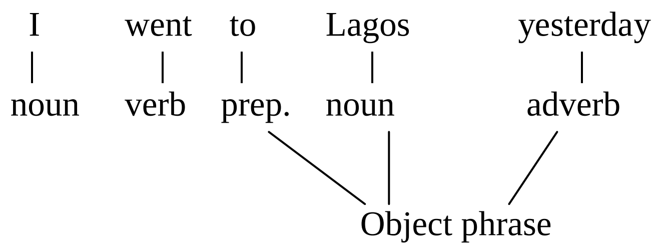
Given the grammar:

<Expression> --> <Term> | <Term> + <Expression>
<Term> --> <Factor> | <Factor> * <Term>
<Factor> --> [<Expression>] | 0...9

Stack	**Input String**	**Action**
()	(2 * [1 + 3])	SHIFT
(2)	(* [1 + 3])	REDUCE
(<Factor>)	(* [1 + 3])	SHIFT
(<Factor> *)	([1 + 3])	SHIFT
(<Factor>*[)	(1 + 3])	SHIFT
(<Factor>*[1)	(+ 3])	REDUCE
(<Factor>*[<Factor>)	(+ 3])	REDUCE
(<Factor>*[<Term>)	(+ 3])	SHIFT
(<Factor>*[<Term>+)	(3])	SHIFT
(<Factor>*[<Term>+3)	( ])	REDUCE
(<Factor>*[Term>+<Factor>)	( ])	REDUCE
(<Factor>*[<Term>+<Term>)	( ])	REDUCE
(<Factor>*[<Term>+<Expression>)	( ])	REDUCE
(<Factor>*[<Expression>)	( ])	SHIFT
(<Factor>*[<Expression>])	()	REDUCE
(<Factor>*<Factor>)	()	REDUCE
(<Factor>*<Term>)	()	REDUCE
(<Term>)	()	REDUCE
(<Expression>)	()	SUCCESS

3.2.3 Shift-Reduce Parsing

This technique is a bottom-up parsing technique. It attempts to constructs a parse tree for an input string beginning at the links and walking towards the root. This process is called reduction. For instance construct a parse tree for:



This is analogous to replacing the right hand side of production with left hand side.

The problem is how to decide which production to reduce at which points. This bottom-up parsing method is called shift-reduce because it consists of shifting input symbols unto a stack until the right side of a production appears on the top of the stack. The right side may then be replaced by the symbols of the left side of the production, and the process repeated.

Informally, a substring which is the right side of a production such that a replacement of that substring by the variable on the left side eventually leads to a reduction to the stack symbols through the reverse of a rightmost derivation is called a "handle"

The process of bottom up parsing may be viewed as one of finding and reducing handle.

Technically, given grammar G, we say a string of terminal omega ω
$\omega \varepsilon L(G)$ iff, $S \Rightarrow \omega$

ω sentence of G, if $S \Rightarrow^* \alpha$ where α may contains non-terminal then say α is a sentential form of G.

3.2.3.1 Handles

A handle of a string is a substring that matches the right side of a production whose reduction to the nonterminal on the left represents one step along the reverse of a rightmost derivation. A *handle* of a right-sentential form γ is a production, $A \rightarrow \beta$ (A derives β) at a position of γ where the string β may be found and replaced by A to produce the previous sentential form in the rightmost derivation of γ.

i.e. If $S \Rightarrow^* \alpha A \omega \Rightarrow \alpha\beta\omega$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta\omega$.

For example let us consider the grammar:

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow i$

Using top-down parsing (rightmost derivation) to find the handles, we have:

$$
\begin{aligned}
E \Rightarrow E + E &\Rightarrow_{rm} E + \underline{E * E} \\
&\Rightarrow_{rm} E + E * \underline{i_1} \\
&\Rightarrow_{rm} E + \underline{i_2} * i \\
&\Rightarrow_{rm} \underline{i_3} + i * i
\end{aligned}
$$

(Note that handles have been underlined above)

Another rightmost derivation: E

$$
\begin{aligned}
&\Rightarrow_{rm} \underline{E * E} \\
&\Rightarrow_{rm} E * \underline{\mathbf{i}_3} \\
&\Rightarrow_{rm} \underline{E + E} * \mathbf{i}_3 \\
&\Rightarrow_{rm} E + \underline{\mathbf{i}_2} * \mathbf{i}_3 \\
&\Rightarrow_{rm} \underline{\mathbf{i}_1} + \mathbf{i}_2 * \mathbf{i}_3
\end{aligned}
$$

3.2.4 Stack Implementation of Shift-Reduce Parsing

There are two problems to be solved in parsing by handle pruning. These are:

- how to locate the substring to be reduced in the right-sentential form
- which production to choose in order to implement the reduction.

A convenient way for dealing with such difficulties is to design shift-reduce parser which uses a stack to hold the grammar symbols and an input buffer to hold the string to be parsed.

The *shift-reduce parser* operates by shifting zero or more symbols onto the stack until a handle appears on the top. The parser then reduces the handle to the left side of the corresponding production. This process continues until an error occurs or the start symbol remains on the stack.

The *shift-reduce parser* performs four operations:

- *shift* - the next input symbol is shifted onto the top of the stack
- *reduce* - the parser knows the right end of the handle is at the top of thestack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle
- *accept* - the parser announces successful completion of parsing
- *error* - the parser discovers that a syntax error has occurred.

Example 3: Successive steps in rightmost derivations:

a) $S \Rightarrow^*_{rm} \alpha A z \Rightarrow_{rm} \alpha\beta B yz \Rightarrow_{rm} \alpha\beta\gamma yz$

	Stack	Input
1)	$ αβγ	*yz$*
2)	$ αβB	*yz$*
3)	$ αβB*y*	*z$*

b) $S \Rightarrow^*_{rm} \alpha B x A z \Rightarrow_{rm} \alpha B xyz \Rightarrow_{rm} \alpha\gamma xyz$

	Stack	Input
1)	$ αγ	*xyz$*
2)	$ αB*xy*	*z$*

Example 4: The shift-reduce parser for the context-free grammar

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow$ **id**

performs the following steps when analysing the input string: **id**$_1$ + **id**$_2$ ***** **id**$_3$

Stack	**Input**	**Action**
$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ $	shift
$ $\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3$ $	reduce by $E \to \mathbf{id}$
$ E	$+ \mathbf{id}_2 * \mathbf{id}_3$ $	shift
$ $E+$	$\mathbf{id}_2 * \mathbf{id}_3$ $	shift
$ $E+\mathbf{id}_2$	$* \mathbf{id}_3$ $	reduce by $E \to \mathbf{id}$
$ $E+E$	$* \mathbf{id}_3$ $	shift
$ $E+E*$	$\mathbf{id}_3$ $	shift
$ $E+E* \mathbf{id}_3$	$	reduce by $E \to \mathbf{id}$
$ $E+E*E$	$	reduce by $E \to E*E$
$ $E+E$	$	reduce by $E \to E+E$
$$E$	$	accept

Another way of analysing this could be as below:

Stack	**Input**	**Action**
$	$i_1 + i_2 * i_3$ $	Shift
$$i_1$	$+ i_2 * i_3$ $	reduce
$E	$+ i_2 * i_3$ $	shift
$E+	$i_2 * i_3$ $	shift
$E+ i_2	$* i_3$ $	reduce
$E + E	$* i_3$ $	reduce
$E	$* i_3$ $	shift
$E *	i_3 $	Shift
$E * i_3	$	reduce
$E * E	$	reduce
$E	$	Accept

Shift reduce parsing is not adequate. In parsing there are four possible actions:

- Shift
- Reduce
- Accept
- Reject/error

SELF-ASSESSMENT EXERCISE

i. Given the grammar in Example 4 above analyse the following input string using shift-reduce parsing
   a. (id + id) * id
   b. id + (id * id).

4.0 CONCLUSION

In this unit, you have been taken through bottom-up parsing using as an example shift-reduce parsing. You have also learnt how to implement shift-reduce parser using stack. In the next unit of this module, you will learn about another type of bottom-up parsing called precedence parsing.

5.0 SUMMARY

In this unit, you learnt that:

- a top-down parser builds a parse tree by starting at the root and working down towards the leaves
- a bottom-up parser builds a parser tree by starting at the leaves and working up towards the root
- a *handle* of a right-sentential form G is a production *A* → b and a position in g where b may be found and replaced by *A* to produce the previous right-sentential form in a rightmost derivation of a grammar G.
- the process to construct a bottom-up parse is called *handle-pruning*
- a shift-reduce parser has just four canonical actions: *shift, reduce, accept, and error.*

6.0 TUTOR-MARKED ASSIGNMENT

i. Given the grammar below:
   a. E → E + T | T
   b. T → T * F | F
   c. F → (E) | id
   d. Find the rightmost derivation of the input string (id * id) + id using top-down parsing
   e. Using shift-reduce parsing, analyse the following input string:
   f. (id * id) + id
   g. id(id + id) * id
ii. Consider the following grammar for list structure:
   a. S → a | ^ | (T)
   b. T → T,S | S
iii. find the rightmost derivations for
   (i) (a, (a, a))
   (ii) (((a, a), ^, (a)), a)
   a Indicate the handle of each right sentential form for the derivations in (b) above

c) show the steps of a shift-reduce parser corresponding to these rightmost derivations

7.0 REFERENCES/FURTHER READING

Alfred, V. Aho, *et al*. (2007). *Compilers: Principles, Techniques, and Tools*. Second Edition. Wesley: Pearson Addison.

Andrew, W. Appel (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Keith, D. Cooper & Linda, Torczon (2004). *Engineering a Compiler*. Morgan Kaufmann.

Steven, S. Muchnick (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

UNIT 3 PRECEDENCE PARSING

CONTENTS

1.0 INTRODUCTION

In the previous unit you have been introduced to bottom-up parsing and a type of bottom-up parsing referred to as shift-reduce parsing. In this unit you will be exposed to another type of bottom-up parsing known as precedence parsing. This parser can be developed using operator grammars.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define operator grammars and operator precedence grammars
- explain the methods of generating relationship between operators
- describe operator precedence parsing
- state the advantages of operator precedence parsing
- compute the Wirth-Weber precedence relationship table for any precedence grammar
- determine if a grammar is operator grammar or operator precedence grammar
- parse input strings using the precedence relationship table
- construct precedence functions.

3.0 MAIN CONTENT

3.1 Operator Precedence Parser

An **operator precedence parser** is a bottom-up parser that interprets an operator-precedence grammar. For example, most calculators use operator precedence parsers to convert from the human-readable infix notation with order of operations format into an internally optimised computer-readable format like Reverse Polish notation (RPN).

3.1.1 Relationship to other Parsers

An operator-precedence parser is a simple shift-reduce parser capable of parsing a subset of LR(1) grammars. More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive nonterminals never appear in the right-hand side of any rule.

Operator-precedence parsers are not used often in practice, however they do have some properties that make them useful within a larger design. First, they are simple enough to write by hand, which is not generally the case with more sophisticated shift-reduce parsers. Second, they can be written to consult an operator table at runtime, which makes them suitable for languages that can add to or change their operators while parsing.

To show that one grammar is **operator precedence**, first it should be **operator grammar.** Operator precedence grammar is the only grammar which can construct the parse tree even though the given grammar is ambiguous.

We try to look at the relationship between operators to guide us in parsing.

3.1.2 Operator Grammars

Operator grammars have the property that no production right side is empty or two or more adjacent non-terminals. This property enables the implementation of efficient *operator-precedence parsers*. These parsers rely on the following three precedence relations:

3.1.3 Operator Precedence Grammar

An operator precedence grammar is an ε-free operator grammar in which the precedence relations define above are disjoint i.e. for any pair of terminals *a* and *b*, never more than one of these relations: $a \lessdot b$, $a \doteq b$ $a \gtrdot b$ is true.

3.1.4 Precedence Relations between Operators (terminals)

For terminals *a* and *b*,

i. **$\lessdot$:** if $a \lessdot b$, we say a "yields precedence to" *b*. This means that a production involving *a* has to yield to a production involving *b*.

ii. **$\doteq$:** if $a = b$, we say *a* "has the same precedence as" *b*. This means that *a* and *b* are part of the same production.

iii. **$\gtrdot$:** if $a \gtrdot b$, we say *a* "has precedence over" *b*. This means that production involving *a* will be reduced before the production involving *b*.

An operator grammar is one that has no production whose right-hand side has two or more adjacent non-terminals.

3.1.4.1 Methods of Generating Relationship between Operators

i. Intuition
ii. Formal

3.1.4.1.1Intuition Method

We can use associativity and precedence of the operations to assign the relation intuitively as follows:

i. If operator θ_1 and θ_2 has higher precedence than $\theta_{2,}$ then we make $\theta_1 \gtrdot \theta_2$ and $\theta_2 \lessdot \theta_1$

ii. If θ_1 and θ_2 are operators of equal precedence then either we make $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$, if the operators are left associative or make $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$, if they are right associative.

SELF-ASSESSMENT EXERCISE 1

i. Consider the grammar G:
   a. $E \rightarrow E + T / T$
   b. $T \rightarrow T * F / F$
   c. $F \rightarrow (E) / i$

ii. Compute the operator precedence relationship table for this grammar.

3.1.4.1.2Formal Method

For each two terminals symbols *a* and *b*,

$a \doteq b$ if there is a right side of a production of the form $\alpha a\beta b\gamma$, where β is either ε or a single non-terminal. i.e. $a \doteq b$ if *a* appears immediately to the left of *b* in a right side or they appear separated by one non-terminal e.g. S → iCtSeS implies that $i \doteq t$ and $t \doteq e$

$a \lessdot b$ if for some non-terminal A there is a right side of the form $\alpha a A\beta$ and $A \Rightarrow^* \gamma b\delta$, where γ is either ε or a single non-terminal. i.e. $a \lessdot b$ if a non-terminal A appears immediately to the right of *a* and derives a string in which *b* is the first terminal symbol. E.g. S → iCtS and $C \Rightarrow^*$ b, therefore $i \lessdot$, b. Also define $\$ \lessdot b$ if there is a derivation $S \Rightarrow^* \gamma b\delta$ and γ is either empty or a single non-terminal.

$a \gtrdot b$ if for some non-terminal A there is a right side of the form $\alpha A b\beta$ and $A \Rightarrow^* \gamma a\delta$, where δ is either ε or a single non-terminal. i.e. $a \gtrdot b$ if a non-terminal A appearing immediately to the left of *b* and derives a string whose last terminal symbol is *a*. E.g. in S → iCtS and $C \Rightarrow^*$ b, therefore $b \gtrdot t$. Also define $a \gtrdot \$$ if there is a derivation $S \Rightarrow^* \gamma a\delta$ and δ is either empty or a single non-terminal.

3.1.5 Construction of Matrix of all Precedence Relations for a Grammar

To construct the matrix of precedence operations for a grammar, you need to first deduce for each nonterminal those terminals that can be the first or last terminal in a string derived from that nonterminal. To demonstrate this, let us consider the grammar below:

$E \rightarrow E + T / T$
$T \rightarrow T * F / F$
$F \rightarrow (E) / i$

For this grammar, all derivations from F will have the symbols (or i as the first terminal and the symbols) or i as the last. A derivation from T could begin T $\Rightarrow$ T * F, showing that * could be both first and last terminal. Or a derivation could begin T $\Rightarrow$ F, meaning that every first and last terminals derivable from F is also a first or last terminal derivable from T. Thus, the symbols *,), and i can be first and *,), and i can be last in a derivation from T. A similar argument applies to E, and we see that +, *, (, or i can be first and *, +,), or i can be last. These facts are summarised in the table below.

Table 1: First and Last Terminals

Nonterminal	**First terminal**	**Last terminal**
E	+, *, (, i	*, +,), i
T	*, (, i	*,), i
F	(, i	), i

Now, to compute the $\doteq$ relation, we look for right sides with two terminals separated by nothing or by a nonterminal. Only one right side, (E), qualifies, so we determine ($\doteq$).

Next, consider $\lessdot$. We look for right sides with a terminal immediately to the left of a nonterminal to play the roles of *a* and *A* in rule (ii). For each such pair, *a* is related by $\lessdot$ to any terminal which can be first in a string derivable from *A*. The candidates in the grammar above are + and T in the right side E + T, * and F in T * F, and (and E in (E). The first of these gives + $\lessdot$ *, + $\lessdot$ (, and + $\lessdot$ i. The *:F pair gives * $\lessdot$ (and * $\lessdot$ i. The (:E pair gives ($\lessdot$ *, ($\lessdot$ +, ($\lessdot$ (, and ($\lessdot$ i. We then add the relationships \$ $\lessdot$ *, \$ $\lessdot$ +, \$ $\lessdot$ (, \$ $\lessdot$ i, since \$ must be related by $\lessdot$ to all possible first terminals derivable from the start symbol E.

Symmetrically, we can construct the $\gtrdot$ relation. We look for the right sides with a nonterminal immediately to the left of a terminal, to play the roles of *A* and *b* of rule (iii). Then, every terminal that could be the last in a string derivable from *A* is related by $\gtrdot$ to *b*. In our grammar, the pair corresponding to A and b are E:+, T:*, and E:). Thus, we have the relations * $\gtrdot$ +, + $\gtrdot$ +,) $\gtrdot$ +, i $\gtrdot$ +, * $\gtrdot$ *,) $\gtrdot$ *, i $\gtrdot$ *, * $\gtrdot$), + $\gtrdot$),)

$\gtrdot$), and i $\gtrdot$). We add the relations * $\gtrdot$ \$, + $\gtrdot$ \$,) $\gtrdot$ \$, and i $\gtrdot$ \$ according to rule (iii). The precedence relations for our grammar are as shown in the table below:

Table 2: Operator-Precedence Relations

	+	*	(	)	i	$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
)	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
I	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	

These operator precedence relations allow us to delimit the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

Let us assume that between the symbols a_i and a_{i+1} there is exactly one precedence relation. Suppose that \$ is the end of the string. Then for all terminals we can write: \$ <· *b* and *b* ·> \$. If we remove all nonterminals and place the correct precedence relation:

<·, =·, ·> between the remaining terminals, there remain strings that can be analysed by easily developed parser.

Example: The input string:

$\mathbf{i}_1 + \mathbf{i}_2 * \mathbf{i}_3$

after inserting precedence relations becomes

\$ <· $\mathbf{i}_1$ ·> + <· $\mathbf{i}_2$ ·> * <· $\mathbf{i}_3$ ·> \$

Having precedence relations allows us to identify handles as follows:

- scan the string from left until seeing ·>
- scan backwards the string from right to left until seeing <·
- everything between the two relations <· and ·> forms the handle

Note that not the entire sentential form is scanned to find the handle.

SELF-ASSESSMENT EXERCISE 2

Using this formal method, compute the precedence relationship matrix table for the grammar G in the Self -Assessment Exercise 1 above.

3.1.6 Operator Precedence Parsing Algorithm

```
Initialise: Set ip to point to the first symbol of w$
Repeat:     Let X be the top stack symbol, and a the symbol pointed
to by ip
 if $ is on the top of the stack and ip points to $ then return
 else

Let a be the top terminal on the stack, and b the symbol pointed to by ip
            if $a <\cdot b$ or $a =\cdot b$ then
            push b onto the stack
            advance ip to the next input symbol
            else if $a \cdot> b$ then
            repeat
            pop the stack
            until the top stack terminal is related by $<\cdot$
             to the terminal most recently popped
            else error()
            end
```

3.1.7 Making Operator Precedence Relations

The operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a special way.

Operator precedence parsers use precedence functions that map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison.

Not every table of precedence relations has precedence functions but in practice for most grammars such functions can be designed.

3.2 Simple Precedence Parser

A **Simple precedence parser** like the operator precedence parser discussed earlier in this unit is a type of bottom-up parser for context-free grammars that can be used only by Simple precedence grammars.

The implementation of the parser is quite similar to the generic bottom-up parser. A stack is used to store a viable prefix of a sentential form from a rightmost derivation. Symbols $\lessdot$, $\doteq$ and $\gtrdot$ are used to identify the **pivot**, and to know when to **Shift** or when to **Reduce**.

3.2.1 Implementation of Simple Precedence Parser

To implement the simple precedence parser, follow the following steps:

```
Step 1      Compute the Wirth-Weber precedence relationship table
            as you learnt in section 3.1.5 above
Step 2      Start with a stack with only the starting marker $.
Step 3      Start with the string being parsed (Input) ended with an
            ending marker $.
Step 4      While not (Stack equals to $S and Input equals to $) (S =
            Initial symbol of the grammar)
      a.    Search in the table the relationship between Top(stack)
            and NextToken(Input)
      b.    if the relationship is ≐ or ⋖
            a.    Shift:
            b.    Push(Stack, relationship)
            c.    Push(Stack, NextToken(Input))
            d.    RemoveNextToken(Input)
      c.    if the relationship is ⋗
            a.    Reduce:
            b.    SearchProductionToReduce(Stack)
            c.    RemovePivot(Stack)
            d.    Search in the table the relationship between the
                  Non terminal from the production and first symbol
                  in the stack (Starting from top)
            e.    Push(Stack, relationship)
            f.    Push(Stack, Non terminal)
Step 5:     SearchProductionToReduce (Stack)
      a.    search the Pivot in the stack the nearest ⋖ from the top
      b.    search in the productions of the grammar which one have
            the same right side than the Pivot
```

Example 1

Given the grammar below:

$E \rightarrow E + T' \mid T'$
$T' \rightarrow T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E') \mid num$
$E' \rightarrow E$

num is a terminal, and the lexer parses any integer as **num**.

By step 1, the Parsing table is generated as below:

Table 3: Parsing Table for the Grammar in Example 1 above

	E	E'	T	T'	F	+	*	(	)	num	$
E						≐			⋗		⋗
E'									≐		
T						⋗	≐		⋗		⋗
T'						⋗			⋗		⋗
F						⋗	⋗		⋗		⋗
+			⋖	≐	⋖			⋖		⋖	
*					≐			⋖		⋖	
(	⋖	≐	⋖	⋖	⋖			⋖		⋖	
)						⋗	⋗		⋗		⋗
num						⋗	⋗		⋗		⋗
$	⋖		⋖	⋖	⋖			⋖		⋖	

By steps 2 through step 5, we have:

STACK	**PRECEDENCE**	**INPUT**	**ACTION**
$	<	2*(1+3)$	SHIFT
$<2	>	*(1+3)$	REDUCE (F→num)
$<F	>	*(1+3)$	REDUCE (T→F)
$<T	=	*(1+3)$	SHIFT
$<T= *	<	(1+3)$	SHIFT
$<T=*<(	<	1+3)$	SHIFT
$<T=*<(<1	>	+3)$	REDUCE 4times (F→num)(T→ F)(T'→T)(E→T')
$<T=*<(<E	=	+3)$	SHIFT
$<T=*<(<E=+	<	3)$	SHIFT
$ < T = * < (< E = + < 3	>	)$	REDUCE 3 times (F → num) (T → F) (T' → T)
$ < T = * < (< E = + = T	>	)$	REDUCE 2 times (E → E + T) (E' → E)
$ < T = * < (< E'	=	)$	SHIFT
$ < T = * < (= E' =)	>	$	REDUCE (F → (E'))
$ < T = * = F	>	$	REDUCE (T → T * F)
$ < T	>	$	REDUCE 2 times (T'→T) (E→T')
$ < E	>	$	ACCEPT

3.3 Algorithm for Constructing Precedence Functions

1. Create functions f_a for each grammar terminal *a* and for the end of string symbol;
2. Partition the created symbols into as many groups as possible such that f_a and g_b are in the same group if $a =\cdot b$ (there can be symbols in the same group even if they are not connected by this relation);
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a <\cdot b$, otherwise if $a \cdot> b$ place an edge from the group of f_a to that of g_b;
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

Example: Consider the table below:

	Id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Using the algorithm leads to the following graph:

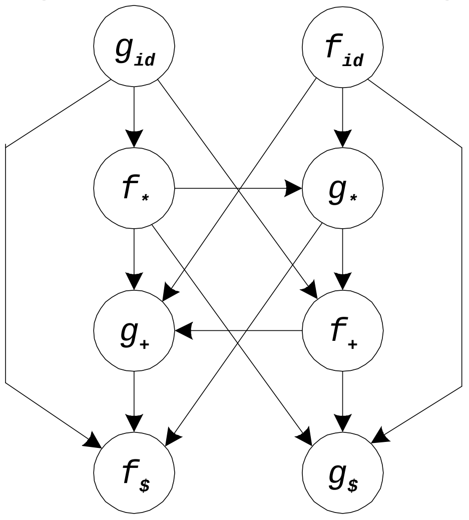


Fig. 1: Graph Representing Precedence Functions

from which we extract the following precedence functions:

Table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

4.0 CONCLUSION

In this unit, you have been taken through another type of bottom-up parsing technique known as precedence parsing. This technique is very suitable for languages that can add to or change their operators while parsing. In the next unit you will learn more about top-down parsing techniques.

5.0 SUMMARY

In this unit, you learnt that:

- an **operator precedence parser** is a bottom-up parser that interprets an operator-precedence grammar
- an operator-precedence parser is a simple shift-reduce parser capable of parsing a subset of LR(1) grammars
- *operator grammars* have the property that no production right side is empty or two or more adjacent non-terminals
- an operator precedence grammar is an ε-free operator grammar in which the precedence relations define above are disjoint i.e. for any pair of terminals *a* and *b*, never more than one of these relations: $a \lessdot b$, $a \doteq b$ $a \gtrdot b$ is true
- precedence parsing techniques is suitable for languages that can add to or change their operators while parsing.

6.0 TUTOR-MARKED ASSIGNMENT

i. Consider the following grammar for list structures:
   S → a | ^ | (T)
   T → T,S | S
   a) Compute the operator-precedence relations for this grammar. Is it an operator-precedence grammar?
   b) Using the operator-precedence relations computed in (a) above, analyse the following input string:
      (i) (a, (a, a))
      (ii) (((a, a), ^, (a)), a)
   c) Find precedence functions for the relations computed in (a) above.

7.0 REFERENCES/FURTHER READING

Alfred, V. Aho *et al.* (2007). *Compilers: Principles, Techniques, and Tools*. Second edition. Wesley: Pearson Addison.

Andrew, W. Appel, (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Keith, D. Cooper & Linda, Torczon (2004). *Engineering a Compiler*. Morgan Kaufmann.

Steven, S. Muchnick (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Michael, L. Scott (2009). *Programming Language Pragmatics*. Third edition. Morgan Kaufman.

Robert, W. Sebesta (2010).*Concepts of Programming Languages*. Ninth edition. Wesley: Addison.

Aho, A.V & Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing*. Englewood Cliffs, N.J.: Prentice-Hall.

UNIT 4 TOP-DOWN PARSING TECHNIQUES

CONTENTS

1.0 INTRODUCTION

In the previous units of this module you have been introduced to some types of bottom-up parsing. In this unit you will be exposed to some top-down parsing techniques such as recursive descent parsing and nonrecursive predictive parsing.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define top-down parsing
- state the difficulties with top-down parsing and how they can be overcome
- describe recursive descent parsing
- define LL(k) grammars
- describe nonrecursive predictive parsing
- write an algorithm for nonrecursive predictive parsing
- construct a predictive parser for LL(k) grammars

- use the predictive parsing table for a grammar to analyse/determine input strings belonging to the grammar.

3.0 MAIN CONTENT

3.1 Top-Down Parsing Techniques

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string or as attempting to construct a parse tree for the input string starting from the root and creating the nodes of the parse tree in pre-order.

3.1.1 Difficulties with Top-Down Parsing

i. Which production will you apply? If the right production is not applied you will not get the input string.
ii. If you are not careful, and there is a left recursive production, it can lead to continue cycling without getting to the answer i.e. input string.
iii. The sequence in which you apply the production matters as to whether you are going to get the input string or not. That is, there is a particular sequence that will lead you to the input string.
iv. If you apply a production and find out that the production cannot work, you have to start all over again.

3.1.1.1 Minimising the Difficulties

However, some of these difficulties can be minimised/removed.

i. Left recursion can be removed from the grammar. This can be done as follows:

   If you have a production like:
   $A \rightarrow A\alpha$. This is left recursive but it can be removed by having something like:
   $A \rightarrow A\alpha \mid \beta$
   You can set
   $A \rightarrow \beta A'$
   Then set $A' \rightarrow \alpha A' \mid \varepsilon$
   By so doing, we have removed left recursion and they generate the same language. But instead of two productions, you have three productions.
ii. Order of trying the alternatives may influence your getting the input or not. You can use a method called **factoring** i.e left factoring is used to ensure that the right sequence/order is used or followed.

e.g. $A \rightarrow \alpha\beta$
$A \rightarrow \alpha\gamma$

If we have the above, which one do we expand A to? We can get rid of this dilemma by using left factoring to determine the right sequence. That is the above becomes:

$A \rightarrow \alpha A'$
$A' \rightarrow \beta \mid \gamma$

3.2 LL (K) GRAMMARS

LL (K) grammars are those for which the left parser can be made to work deterministically, if it is allowed to look at K-input symbols, to the right of its current input position. The left parser expands the leftmost non-terminal or variable.

In other words, a context-free grammar is called LL *grammar* when it is elaborated to enable scanning the input from left to right (denoted by the first L) and to produce a leftmost derivation (denoted by the second L).

Often we consider LL (1) grammars that use one input symbol of lookahead at each step to make parsing decisions.

These two properties allow us to perform efficient recursive parsing. Moreover there can be automatically constructed predictive parsers from LL grammars.

We use what is called a predictive parsing method to parse certain things derivable from LL (K)

3.3 Recursive Descent Parsing

Recursive descent is a strategy for doing top-down parsing. As you learnt earlier in this unit, *Top-down parsing* aims to identify a leftmost derivation of a given input string with respect to the predefined grammar. The top-down parsing process constructs a parse tree starting from the root and proceeding to expand the tree by creating children nodes till reaching the leaves.

Recursive descent parsers however operate doing backtracking, that is, they make repeated scans of the input in order to decide which production to consider next for tree expansion. Using backtracking makes the recursive descent parsers inefficient, and they are only a theoretically possible alternative. That is why recursive parsers that do not need backtracking to obtain the parse tree have been developed.

Therefore, today, recursive descent parsing is a top-down approach that avoids backtracking.

The Recursive procedures can be quite easy to write and fairly efficient if written in a language that implement procedure calls efficiently.

Example 1
Consider the following grammar

(a)	$E \to TE'$	(b)	$E \to E+T \mid T$
	$E^1 \to TE' \mid E$		$T \to T*F \mid F$
	$T \to FT'$		$F \to (E) \mid i$
	$T^1 \to *FT' \mid \varepsilon$		
	$F \to (E) \mid i$		

Grammar (a) and (b) are alike/identical because grammar (b) has left recursion, we can transform it to the form written in grammar (a) above. We can then easily implement the parser by writing a procedure for each non-terminal like the one below.

For grammar (a), we have five non-terminals: E, E', T, T', F
By writing five (5) procedures for each of these non-terminals, then our parser is complete.

```
Procedure E ( );
     Begin
    T ( ) ; EPRIME ( )
     End

Procedure EPRIME ( );
     Begin
If input-symbol = "+" then
     Begin
ADVANCE ( ) ; T( ); Eprime ( )
            End
     end
.
.
.

Procedure F( ) ;
     Begin
If input-symbol = "i" then Advance ( )
Else if input-symbol = "(" then
begin
ADVANCE ( );   E ( );
If input-symbol = ")" then ADVANCE ( );
```

```
Else ERROR ( );
End
Else ERROR ( ) ;
End
```

Fig. 1:Recursive Procedures for Top-down Parsing

Example 2:

Suppose we have a grammar G:

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow a$

Let us say we want to find the left parse of the sentence a * (a + a)

Solution

We need to generate or derive to get a*(a+a)

Remember, we derive leftmost

$E \rightarrow T$	(2)
$\rightarrow T * F$	(3)
$\rightarrow F * F$	(4)
$\rightarrow a * F$	(6)
$\rightarrow a * (E)$	(5)
$\rightarrow a * (T + T)$	(1)
$\rightarrow a * (T+T)$	(2)
$\rightarrow a * (F + T)$	(4)
$\rightarrow a * (a + T)$	(6)
$\rightarrow a * (a + F)$	(4)
$\rightarrow a + (a + a)$	(6)

Sequence of derivation is 23465124646

3.4 Non-Recursive Predictive Parsing

Nonrecursive parsers rely on careful rewriting of the context-free grammar (as you learnt in unit 1 of this module and also in section 3.1.1.1 of this unit) so as to decide unambiguously which production to apply next, without recursive backtracking, upon seen a particular input symbol.

After rewriting of the grammar there is made a parsing table which allows us to implement an efficient stack-based parser.

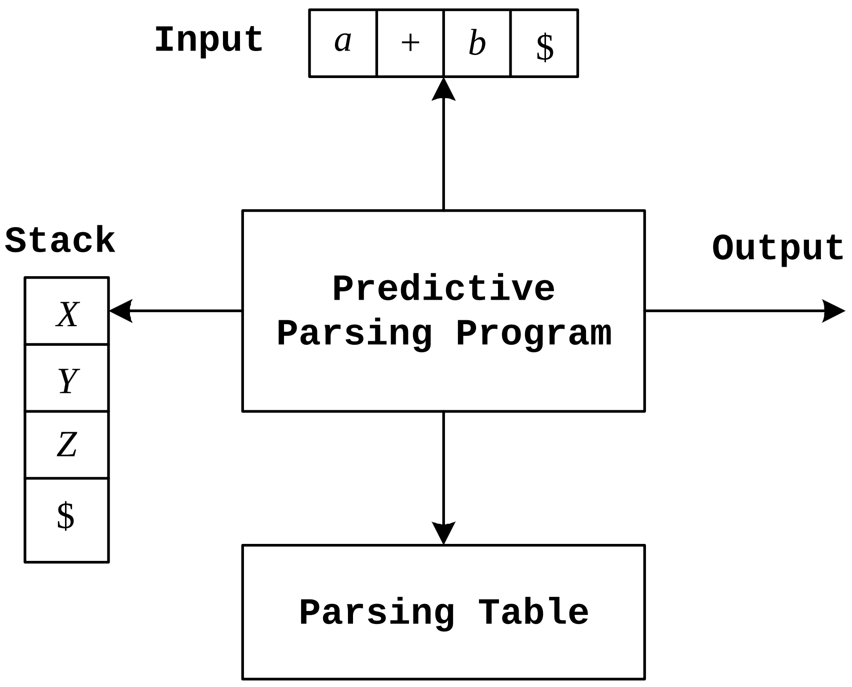


Fig. 1: Model of a Predictive Parser

A *nonrecursive parser* is an implementation of a predictive parser that keeps the symbols on a stack explicitly, without maintaining them implicitly via recursive calls.

The input of a nonrecursive parser consists of a string *w* and a parsing table *M* for the grammar *G*.

The output of a nonrecursive parser is *leftmost derivation* of *w*, if the string *w* belongs to the grammar, that is it is in *L* (*G*), otherwise it announces an error.

We can picture a predictive parser as in figure 1 above.

A predictive parser has an input, a stack, a parsing table and an output. The input contains the string to be parsed followed by dollar ($) sign, the right-end marker. The stack contains a sequence of grammar symbols preceded by the dollar sign, the bottom of stack marker. Initially the stack contains the start symbol of the grammar preceded by the dollar sign. The parsing table is a 2-dimensional array M[A, *a*]. Where A is a non-terminal and *a* is a terminal or the dollar sign

The parser is controlled by a programme that behaves as follows:
The program determines X, the symbol on top of the stack and *a*, the current input symbol to be parsed. These two symbols determine the action of the parser.

Looking at the symbols, there are three possibilities:

1. If X is a terminal and is also the end of string, the right end marker (i.e. X = *a* = $), the parser halts and announces the successful completion of parsing.
2. If X= *a* ≠ $, i.e. *a* is on top of the stack and *a* is the input to the process, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal, the programme consults the entry M[X, *a*] of the parsing table M and behaves accordingly. This entry will either be an X-production of the grammar or an error entry. If M[X, *a*] = {X → UVW}, the parser replaces X on top of the stack by WVU (with U on top). As output, the grammar does the semantic action associated with this production, which, for the time being, we shall assume is just printing the production used. If M[X, *a*] = ***error***, the parser calls an error recovery routine.

3.4.1 Nonrecursive Predictive Parsing Algorithm

We shall describe the behaviour of the parser in terms of its configurations, which give the stack contents and the remaining input. Initially, the parser is in configuration:

Stack	**Input**
$*S*	ω $

where *S* is the start symbol of the grammar and ω is the string to be parsed. The program that utilises the predictive parsing table to produce a parse is shown in Figure 2 below.

```
Initialise: Set ip to point to the first symbol of w$
    Repeat: Let X be the top stack symbol, and a the symbol pointed to by ip
        if X is a terminal or $ then
            if X = a then
                pop X from the stack and advance ip
            else error()
        else /* X is a nonterminal */
            if M[X, a] = $X \to Y_1, Y_2, ..., Y_k$ then
            begin
                pop X from the stack
                push $Y_k, Y_{k-1}, ..., Y_1$ onto the stack, with $Y_1$ on top
                output the production $X \to Y_1, Y_2, ..., Y_k$
            end
            else error()
    until X = $
```

Fig. 2: Predictive Parsing programme

3.4.2 Functions Definitions

To guide us in constructing the parsing table, we need to define the following functions.

3.4.2.1 FIRST and FOLLOW

To fill in the entries of a predictive parsing table, we need two functions associated with grammar G. These functions, FIRST and FOLLOW, will indicate the proper entries in the table fo G, if such a parsing table for G exists.

3.4.2.1.1FIRST

If α is any string of grammar symbols, define FIRST (α) to be the set of terminals that begin strings derived from α* If $\alpha \Rightarrow^* \varepsilon$, then ε is also in FIRST (α).

The general definition of FIRST is $\text{FIRST}_k(\alpha)$ meaning the first k symbols in any string that α can derive.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals of ε can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is (X)
2. If X is nonterminal and $X \rightarrow a\alpha$ is a production, then add *a* to FIRST(X). If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. If $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then for all *i* such that all of $Y_{1,} \ldots, Y_{i-1}$ are nonterminals and FIRST(Y_j) contains ε for *j* = 1, 2, ..., *i*-1 (i.e. $Y_{1,}.Y_2..,Y_{i-1} \Rightarrow^* \varepsilon$), add every non-ε symbol in FIRST(Y_i) to FIRST(X). If ε is in FIRST (Y_j) for all for *j* = 1, 2, ..., *k*, then add ε to FIRST(X).

Now we can compute FIRST for any string $X_1X_2 \ldots X_n$ as follows. Add to FIRST($X_1X_2 \ldots X_n$) all the non-ε symbols of FIRST(X_1). Also add the the non-ε symbols of FIRST(X_2) if ε is in FIRST(X_1), the non-ε symbols of FIRST(X_2) if ε is in both FIRST(X_1) and FIRST(X_2), etc. Finally, add ε to FIRST($X_1X_2 \ldots X_n$) if, for all *i*, FIRST(X_i) contains ε.

Example: Consider the Grammar below:

G: $E \rightarrow TE'$
$E' \rightarrow +TE'/\varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'/\varepsilon$

$F \rightarrow (E) / i$

Suppose we want to find FIRST(E) i.e. What are the first elements of strings which E can derive?

Solution
Elements of FIRST(E) = { (, i }
Elements of FIRST(E') = { +, ε }
Elements of FIRST T') = { * , ε }

3.4.2.1.2FOLLOW

Define FOLLOW(A), for non-terminal A to be the set of terminals *a*, that can appear immediately to the right of A in some sentential form. i.e. $s \rightarrow^* \alpha A a \beta$

If A can be the rightmost symbol in some sentential form, then ε is in FOLLOW(A)

If A is the distinguish or is the start symbol, then FOLLOW (A) contains ε

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any follow set.

ε is in FOLLOW(*S*), where *S* is the start symbol.

If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) but ε is in FOLLOW(B). Note that ε may still wind up in FOLLOW(B) by rule (3)

If there is a production $A \rightarrow \alpha\beta$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε (i.e. $\beta \Rightarrow^* \varepsilon$, then everything in FOLLOW(A) is in FOLLOW(B)

Example 3:

Find FOLLOW (E), FOLLOW (E') , FOLLOW (T), FOLLOW (T') , and FOLLOW (F) for the grammar below:

G: $E \rightarrow TE'$
$E' \rightarrow +TE'/\varepsilon$
$T \rightarrow FT'$

T’ → *FT’/ε
F → (E) / i

Solution:
FOLLOW (E) = FOLLOW (E’) = {ε,)}
FOLLOW (T) = FOLLOW (T’) = {ε,), +}
FOLLOW (F) = {ε,), +, *}

3.4.3 How to Construct a Parsing Table

The following algorithm can be used to construct a predictive parsing table for a grammar G. The idea behind the algorithm is simple. Suppose A→ α is a production with *a* in FIRST(α). Then whenever the parser has A on top of the stack with *a* the current symbol, the parser will expand A by α. The only complication occurs when α = ε or α ⇒* ε. In this case, you should also expand A by α if the current input symbol is in FOLLOW(A), or if the $ on the input has been reached and ε is in FOLLOW(A).

The algorithm for constructing the parsing table is as follows:

Input: Grammar G
Output: Parsing table M

Method:
Step 1: For each production A derives α (i.e. A→ α) of the grammar do steps 2 and 3.
Step 2: For each terminal *a* in FIRST (α), add production A→ α to M [A, *a*]

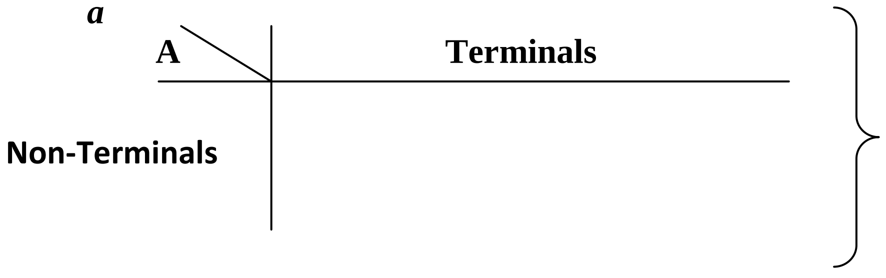


Fig. 2: The Parsing Table M[A, a]

Step 3: If ε is in FIRST (α), add A → α to M[A, *b*] for each terminal *b* in FOLLOW(A). If ε is in FIRST (α) and also in FOLLOW(A) add A →α to M[A, $]
Step 4: Make each undefined entry of M error

Example 4

Using the grammar G below, let us construct a parsing table.

G: E → TE'
E' → +TE'/ε
T → FT'
T' → *FT'/ε
F → (E) / i

We want to construct the parsing table for each production

a. Considering the 1st production E→TE' ,
A ≡ E; α ≡ TE'
∴ FIRST(TE') = FIRST(E) = {(, i} and ε is not in FIRST(TE')

b. Considering the 2nd production E' → +TE'
A ≡ E; α ≡ +TE'
∴ FIRST(+TE') = {+} and ε is not in FIRST(+TE')

c. Considering 3rd production, E' → ε
A ≡ E'; α ≡ ε
∴ FIRST(ε) = {ε}.
Since ε is in FIRST(ε), therefore, we find FOLLOW(E') = {), ε}

d. Considering the 4th production T → FT'
A ≡ T; α ≡ FT'
∴ FIRST(FT') = {i, (} and ε is not in FIRST(FT')

e. Considering the 5th production T' → *FT'
A ≡ T'; α ≡ *FT'
∴ FIRST(*FT') = {*} and ε is not in FIRST(*FT')

f. Considering 6th production, T' → ε
A ≡ T'; α ≡ ε
∴ FIRST(ε) = {ε}.
Since ε is in FIRST(ε), therefore, we find FOLLOW(T') = {+,), ε}

g. Considering the 7th production F' → (E)
A ≡ F; α ≡ (E)
∴ FIRST ((E)) = {(} and ε is not in FIRST((E))

g. Considering the 8th production F → i
A ≡ F; α ≡ i
∴ FIRST(i) = {i} and ε is not in FIRST(i)

Table 2: Parsing Table for the Grammar in Example 4

	I	+	*	**(**	**)**	**$**
E	E → TE’			E → E’		
E’		E’ → TE’			E’ → ε	E’ → ε
T	T → FT’			T → FT’		
T’		T’ → ε	T → *FT’		T’ → ε	T’ → ε
F	F → i			F → (E)		

We want to know if the sentence i+i*i can be formed from the grammar above using the parsing table above

Solution

Using Top-down parsing

Stack	**Input**	**Output**
$E	i + i * i $	
$E’T	i + i * i $	E → TE’
$E’T’F	i + i * i $	T → FT’
$E’T’i	i + i * i $	F → i
$E’T’	+ i * i $	pop i
$E’	+ i * i $	T → ε
$E’T+	+ i * i $	E’ → +TE’
$E’T	i * i $	pop +
$E’T’F	i * i $	T → FT’
$E’T’i	i * i $	F → i
$E’T’	* i $	Pop i
$E’T’F*	* i $	T → *FT’
$E’T’F	i $	pop *
$E’T’i	i $	F → i
$E’T’	$	pop i
$E’	$	T’ → ε
$	$	E’ → ε

Since the stack and input is empty, therefore the sentence i+i*i can be formed from the grammar and the parsing table.

The problem with the parsing table above is that the grammar must be non-left recursive otherwise it will not be easy to construct the above parsing table.

SELF -ASSESSMENT EXERCISE

i. Given the grammar in example 4 above and the corresponding parsing table in figure 3 above, determine whether the following input strings can be generated from the grammar.
   a) (i * i) * (i + i)
   b) i (i + i) * i

4.0 CONCLUSION

In this unit, you have been taken through some top-down parsing techniques such as recursive descent parsing and predictive parsing techniques. These two techniques like their counterpart bottom-up techniques that you have learnt about so far in this module and be implemented by hand. The predictive parser has the limitation that it cannot be constructed for left-recursive grammars. In the next unit you will learn about another type of parser that cannot be implemented by hand. These are LR parsers which are so called because they can scan the input from left-to-right and construct a rightmost derivation in reverse. They are efficient bottom-up parsers that can be constructed for a large class of context-free grammars.

5.0 SUMMARY

In this unit, you learnt that:

- LL(K) grammars are those for which the left parser can be made to work deterministically, if it is allowed to look at K-input symbols, to the right of its current input position
- *Top-down parsing* aims to identify a leftmost derivation of a given input string with respect to the predefined grammar. The top-down parsing process constructs a parse tree starting from the root and proceeding to expand the tree by creating children nodes till reaching the leaves
- *Recursive descent* is a top-down parsing strategy that operate doing backtracking
- Using backtracking makes the recursive descent parsers inefficient, and they are only a theoretically possible alternative
- A *nonrecursive parser* is an implementation of a predictive parser that keeps the symbols on a stack explicitly, without maintaining them implicitly via recursive calls
- The general definition of FIRST is $\text{FIRST}_k(\alpha)$ meaning the first k symbols in any string that α can derive
- FOLLOW(A), for non-terminal A is the set of terminals *a*, that can appear immediately to the right of A in some sentential form. i.e. $s \rightarrow^* \alpha A a \beta$

6.0 TUTOR-MARKED ASSIGNMENT

i. The grammar below is an LL(1) grammar for regular expressions over alphabet {*a*, *b*}, with + standing for the union operator (|) and Є for the symbol ε.

$E \rightarrow TE'$

$E' \rightarrow +E \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow T \mid \varepsilon$

$F \rightarrow PF'$

$F' \rightarrow *F' \mid \varepsilon$

$P \rightarrow (E) \mid a \mid b \mid Є$

a. Compute FIRST and FOLLOW for each nonterminal of the above grammar
b. Show that the grammar is LL(1)
c. Construct the predictive parsing table for the grammar
d. Construct a recursive-descent parser for the grammar.

ii. Consider the following grammar:

$S \rightarrow aSa \mid aa$

Clearly the grammar generates all even length strings of *a*'s except for the empty string

a. By tracing through the steps of a top-down parser (with backtracking) which tries the alternate *a*S*a* before *aa*, show that S succeeds on 2, 4, or 8 *a*'s, but fail on 6 *a*'s
b. What strings cause the parser to succeed?

7.0 REFERENCES/FURTHER READING

Alfred, V. Aho *et al.* (2007). *Compilers: Principles, Techniques, and Tools*. Second Edition. Wesley: Pearson Addison.

Andrew, W. Appel (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Cooper, Keith D. & Torczon, Linda (2004). *Engineering a Compiler*. Morgan Kaufmann.

Muchnick, Steven S. (1997).*Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Scott, Michael L. (2009). *Programming Language Pragmatics*. Third edition, Morgan Kaufman.

Sebesta,Robert W. (2010). *Concepts of Programming Languages*. Ninth edition. Wesley: Pearson Addison.

UNIT 5 LR PARSERS

CONTENTS

1.0 INTRODUCTION

The problem with the parsing table discussed in the previous unit is that the grammar must be non-left recursive otherwise it will not be easy to construct the parsing table. In this unit, you will be shown how to construct efficient bottom-up parsers for a large class of context-free grammars. These parsers are called LR parsers because they can scan the input from left-to-right and construct a rightmost derivation in reverse. These class of parsers are attractive for a variety of reasons amongst which is that they can be constructed to recognise virtually all programming language constructs for which context-free grammars can be written.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define LR(k) grammars
- state the advantages and drawbacks of LR parsers
- construct a simple LR parser
- distinguish among the different types of LR parsers.
- compute all the LR(0) items for a given grammar
- construct an NFA whose states are the LR(0) items for a given grammar
- define basic functions involved in the generation of LR parsers such as GOTO, CLOSURE, etc.

3.0 MAIN CONTENT

3.1 LR (K) GRAMMARS

These are grammars for which the right parser can be made to work deterministically if it is allowed to look at k-input symbols to the left of its current input position. That is, a context-free grammar is called *LR grammar* when it is elaborated to enable scanning the input from left to right (denoted by L) and to produce a rightmost derivation (denoted by R).

3.1.1 Why Study LR Grammars?

We study LR grammars for a variety of reasons amongst which are the following:

- LR (1) grammars are often used to construct parsers. We call these parsers LR(1) parsers and it is everyone's favourite parser
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parse-able by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognised by predictive (i.e., LL) parsers

LL (*k*)**:** recognise use of a production $A \rightarrow \beta$ seeing first *k* symbols of β
LR (*k*)**:** recognise occurrence of β □(the handle) having seen all of what is derived from β □plus *k* symbols of lookahead.

3.2 LR Parsing

LR parsing can be generalised as arbitrary context-free language parsing without a performance penalty, even for LR (k) grammars. This is because most programming languages can be expressed with LR (k) grammars, where k is a small constant (usually 1). Note that parsing of non-LR (k) grammars is an order of magnitude slower (cubic instead of quadratic in relation to the input length).

3.2.1 Benefits of LR Parsing

LR parsing is attractive for a variety of reasons amongst which are the following reasons:

a. LR parsing can handle a larger range of languages than LL parsing, and is also better at error reporting, i.e. it detects syntactic errors when the input does not conform to the grammar as soon as possible. This is in contrast to an LL (k) (or even worse, an LL (*) parser) which may defer error detection to a different branch of the grammar due to backtracking, often making errors harder to localise across disjunctions with long common prefixes.
b. LR parsers can be constructed to recognise virtually all programming language constructs for which context-free grammars can be written
c. It is more general than operator precedence or any other common shift-reduce techniques discussed so far in this module, yet it can be implemented with the same degree of efficiency as these other methods.
d. LR parsing also dominates the common forms of top-down parsing without backtrack. That is it is the most general non-backtracking parsing method.

3.2.2 Drawback of LR Parsing

The principal drawback of the method is that it is too much work to implement an LR parser by hand for a typical programming language grammar. However, an LR parser generator is now generally available to assist. You need a specialised tool called an LR parser generator. The design of one of such will be discussed in this unit.

3.2.3 Techniques for Producing LR Parsing Tables

There are three common techniques for building tables for an “LR” parser. These are itemised below with their peculiar characteristics.

The first method is called simple LR (SLR (1) for short). It is easiest to implement. Unfortunately, it may fail to produce a table for certain grammars on which the other methods succeed. In summary, SLR has the following characteristics:

- smallest class of grammars
- smallest tables (number of states)
- simple, fast construction

The second method is called canonical LR (or LR (1)). It is the most powerful and works on a very large class of grammars. Unfortunately, the canonical LR method can be very expensive to implement. In summary, the characteristics of the LR(1) are as follows:

- full set of LR(1) grammars
- largest tables (number of states)
- slow, expensive and large construction

The third method is called lookahead LR (LALR (1) for short). It is intermediate in power between the SLR and the canonical LR. The LALR method works on most programming-language grammars and, with some effort, can be implemented efficiently. In summary, the characteristics of the LALR(1) are as follows:

- intermediate sized set of grammars
- same number of states as SLR(1)
- canonical construction is slow and large
- better construction techniques exist

Note that an LR (1) parser for either Algol or Pascal has several thousand states, while an SLR (1) or LALR (1) parser for the same language may have several hundred states.

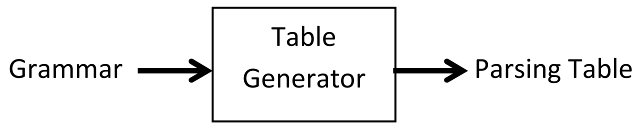
In subsequent sections of this unit you will be exposed to how ambiguous grammars can be used to simplify the description of languages and produce efficient parsers.

3.3 LR (K) PARSER

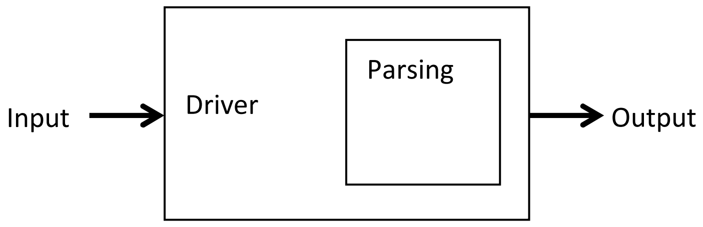
As you learnt earlier, LR (K) parsers are so called because they scan the input from left to right and construct a rightmost derivation in reverse. LR parsing method is more general than operator precedence or any other shift-reduce technique. They generally can recognise virtually all programming language constructs for which context-free grammars can be written. LR parser can detect syntactic errors as soon as it is possible to do so on a left to right scan of the input.

3.3.1 Configuration of LR Parser

Logically, an LR parser consists of two parts, a driver routine and a parsing table. The driver routine is the same for all LR parsers, only the parsing table changes from one parser to another. The schematic form of an LR parser is shown in figure 1. As the driver routine is simple to implement, we shall often consider the LR parser construction process as one of producing the parsing table for a given grammar as in figure 1(a).



(a) Generating the parser



(b) Operation of the parser

Fig. 1: Generating an LR Parser

The parser has an input, a stack and a parsing table. The input is read from left to right one symbol at a time. The stack contains a string of the form:

$S_oX_1S_1X_2S_2 \ldots X_mS_m$

where S_m is on top; each X_i is a grammar symbol and each S_i is a grammar symbol called a state. Each state symbol summarises the information contained in the stack below it and it is used to guide the shift-reduce decision. In actual implementation, the grammar symbols need not appear on the stack. We include them there only to help explain the behaviour of an LR parser.

The parsing table consist of two parts:

- a parsing action function called ACTION, and

- a go to function called GOTO

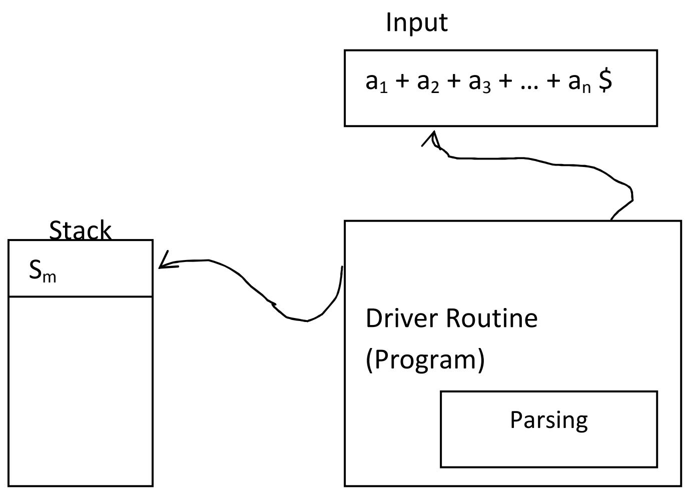


Fig. 2: LR Parser

The programme driving the LR parser behaves as follows. It determines S_m, the state currently on top of the stack, and ai, the current input symbol. It then consults ACTION[S_m, a_i], the parsing action table entry for state S_m and input a_i. The entry ACTION[S_m, a_i] can have one of four values:

Shift S
Reduce $A \rightarrow \beta$
Accept
Reject (error)

The function GOTO takes a state and grammar symbol as arguments and produces a state. It is essentially the transition the transition table of a deterministic finite automaton whose input symbols are the terminals and nonterminals of the grammar.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input.

($SoX_1S_1X_2S_{2.}X_3 \ldots X_mS_m$, $a_i\ a_{i+1,}\ a_{i+2} \ldots a_n$\$)

The next move of the parser is determined by reading a_i, the current input symbol and S_m, the state on top of the stack and then consulting the parsing action table entry ACTION[S_m, a_i]

($S_oX_1S_1X_2S_2 \ldots\ldots X_mS_m$, $a_i, a_{i+1}, a_{i+2}, \ldots, a_n$\$)

Note that:

(i) ACTION () will take care of parsing
(ii) GOTO() will take care of states

The configurations resulting after each of the four types of move are as follows:

1. If ACTION [S_m, a_i] = Shift S, the parser executes a shift move, entering the configuration:
   {$S_oX_1S_1X_2S_2 \ldots X_mS_ma_iS$, $a_{i+1}\ a_{i+2} \ldots a_n\$$)
   i.e. Shift a_i to join S_m and go to a new state S.
   Here the parser has shifted the current input symbol a_i and the next state S = GOTO [S_m, a_i] onto the stack; a_{i+1} becomes the new current input symbol.
2. If ACTION[S_m, a_i] = Reduce A → β, then the parser executes a reduce move entering the configuration:
   ($S_oX_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}$ A S, a_i, a_{i+1}, a_{i+2}, …, $a_n\$$)
   Where S = GOTO[S_{m-r}, A] and r is the length of β, the right side of the production. Here the parser first pops two r symbols off the stack (r state symbols and r grammar symbols), exposing states S_{m-r}. The parser then pushed both A, the left side of the production, and S, the entry for ACTION[S_{m-r}, A], onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \ldots X_m$, the sequence of grammar symbols popped off the stack, will always match β, the right side of the reducing production.
3. IF ACTION[S_m, a_i] = Accept, parsing is completed
4. IF ACTION[S_m, a_i] = Error, the parser has discovered an error and would call the error recovery routine.

The LR parsing algorithm is very simple. Initially the LR parser is in the configuration: (S_o, $a_1\ a_2 \ldots a_n\$$) where S_o is a designated initial states and $a_1a_2 \ldots a_n$ is the string to be parsed. Then the parser executes moves until an accept, or error action is encountered. You will either accept or reject the string. All LR parsers behave in this manner. The only difference between one LR parser and another is the information in the parsing action and goto fields of the parsing table.

3.4 Simple LR (SLR) Parser

3.4.1 Definitions

3.4.1.1Viable Prefix

Suppose S $\Rightarrow^*$ αAω $\Rightarrow$ αβω in a rightmost derivation in grammar G. Then γ is a viable prefix of G if γ is a prefix of αβ. That is, γ is a string which is a prefix of some right sentential form but which does not extend past the right end of the handle of that right sentential form (i.e. γ do not contain any symbols to the right of the handle.

A viable prefix is so called because it is always possible to add terminal symbols to the end of a viable prefix to obtain a right sentential form. Therefore, there is apparently no error as long as the portion of the input seen to a given point can be reduced to a viable prefix.

3.4.1.2 LR (0) Item (or Simple Item)

LR (0) item or simple item of a grammar G is a production of G with a dot at some position on the right side. Thus, if you have a production of the form: A → XYZ, then it will generate the following four items:

A→ .XYZ
A → X.YZ
A → XY.Z
A → XYZ. are all LR (0) item.

Also the production A→ ε generates only one item A → .

Inside the computer, items are easily represented by pairs of integers, the first giving the number of the production and the second the position of the dot.

We group items together into sets, which give rise to the states of an LR parser. The items can be viewed as the states of an NFA recognising viable prefixes.

One collection of sets of items, which we call the *canonical LR (0)* collection, provides the basis for constructing a class of LR parsers called simple LR (SLR). To construct the canonical LR (0) collection for a grammar, we need to define an augmented grammar and two functions, CLOSURE and GOTO.

3.4.1.3 Augmented Grammar

If G is a grammar with start symbol S, then G’ (augmented Grammar for G) is G with a new start symbol S’ and new production S’ → S. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. This would occur when the parser was about to reduce S’ → S.

Example 1:

Consider the grammar,
G: $E \rightarrow E + T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid i$

The augmented grammar for this grammar is grammar G' below:

G': $E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid i$

3.4.1.4 CLOSURE

If I is a set of items for a grammar G, then the set of items CLOSURE[I] is constructed from I by the following rules:

Every item in I is in CLOSURE[I]

If $A \rightarrow \alpha.B\beta$ is in CLOSURE [I] and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \gamma$ is a production, then we add the item $B \rightarrow .\gamma$ to I, if it is not already there.

The function CLOSURE can be computed as in the algorithm below.

```
Procedure CLOSURE(I);
begin
  repeat
    for each item $A \rightarrow \alpha.B\beta$ in I and each production
           $B \rightarrow \gamma$ in G such that $B \rightarrow .\gamma$ is not in I
         do add $B \rightarrow .\gamma$ to I
  until no more items can be added to I;
  return I;
end
```

Fig. 3: Computation of CLOSURE

Example 2:

Consider the augmented grammar
$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid i$

I = {E’ → .E},

Find CLOSURE[I]

Solution:
If I is the set of one item {[E’ → E]}, then CLOSURE[I] contains the items
E’ → .E
E → .E + T
E → .T
T → .T*F
T → .F
F → .(E)
F → .i

That is, E’ → .E is in CLOSURE (I) by rule (i). Since there is an E immediately to the right of a dot, by rule (ii) we are forced to add the E productions with dots at the left end, that is, E → .E + T and E → .T. Now there is a T immediately to the right of a dot, so we add T → .T * F and T → .F. Next, the F to the right of a dot forces F → .(E) and F → .i to be added. No other items are put into CLOSURE (I) by rule (ii).

3.4.1.5 GOTO Function

If I is a set of items and X is a grammar symbol then GOTO (I, X) is defined to be the closure of the set of all items [A → αX.β] such that [A → α.Xβ] is in I.

Example 3:
Consider the Grammar
E’ → E
E → E + T | T
T → T*F | F
F → (E) | i

If I is the set of items {[E’ → E.], [E → E.+T]}, then GOTO[I,+] consists of
E→ E + .T
T → .T*F
T → .F
F → .(E)
F →.i

That is, we examine I for items with + immediately to the right of the dot. E’ → E. Is not such an item, but E → E.+T is. We move the dot over the + to get [E → E+.T] and take the closure of this set.

i.e. I = CLOSURE (E → E+.T)

This implies that I = E → E + .T
T → .T*F
T → .F
F → .(E)
F →.i

Therefore, CLOSURE (E → E+.T) = {E→ E + .T, T → .T*F, T → .F, F → .(E), F →.i}

3.4.1.6 The Sets-of-Item Construction

You are now ready to learn the algorithm to construct *C*, the canonical collection of sets of LR(0) items for an augmented grammar G' (i.e. CLOSURE [S' → .S]), the algorithm is shown if figure 4 below:

```
procedure ITEMS (G');
begin
      C := {CLOSURE({S '→ .S})};
      repeat
           for each set of items I in C and each grammar symbol X
                  such that GOTO(I, X) is not empty and is not in C
                do add GOTO(I, X) to C
      until no more sets of items can be added to C
end
```

Fig. 4: The Sets-of Items Construction

Example 4:

For the augmented grammar below,
Find the canonical collection of sets of items (i.e. CLOSURE(E' → .E))
Represent the GOTO function for this set of items as the transition diagram of a deterministic finite automaton *D*.

E' → E
E → E + T | T
T → T*F | F
F → (E) | i

Solution:

C = CLOSURE (E' → .E)

I_0: E' → .E

E → .E + T | .T
T → .T*F | .F
F → .(E) | .i

Expanding I_0

I_1: GOTO (I_0, E)
E' → E.
E → E. + T

I_2: GOTO (I_0, T)
E → T.
T → T. * F

I_3:GOTO (I_2, F)
T → F.

I_4: GOTO (I_0, ()
F → (.E)
E → .E + T | .T
T → .T*F | .F
F → .(E) | .i

I_5: GOTO (I_0, i)
F → i.

Expanding I_1

I_6: GOTO (I_1, +)
E → E+ .T
T → .T*F | .F
F → .(E) | .i

Expanding I_2

I_7: GOTO (I_2, *)
F →T * .F
F → .(E) | .i

Expanding I_4

I_8: GOTO (I_4, E) GOTO (I_4, T) = I_2 GOTO (I_4, () = I_4

GOTO (I_4, F) = I_3 GOTO (I_4, i) = I_5

Expanding I_6

I_9: GOTO (I_6, T) GOTO ($I_{6,}$ F) = I_3 GOTO ($I_{6,}$ () = I_4
E → E + T.
T → .T*F GOTO ($I_{6,}$ i) = I_5

Expanding I_7

I_{10}: GOTO (I_7, F) GOTO (I_7, () = I_4 GOTO (I_7, i) = I_5
T → T*F.

Expanding I_8

I_{11}: GOTO (I_8,)) GOTO (I_8, +) = I_6
F → (E).

Expanding I_9

GOTO (I_9, +) = I_7

NOTE:
The final state for the given grammar G is I_{11} since it cannot be expanded further. Therefore I_{11} is the closure.

ii)

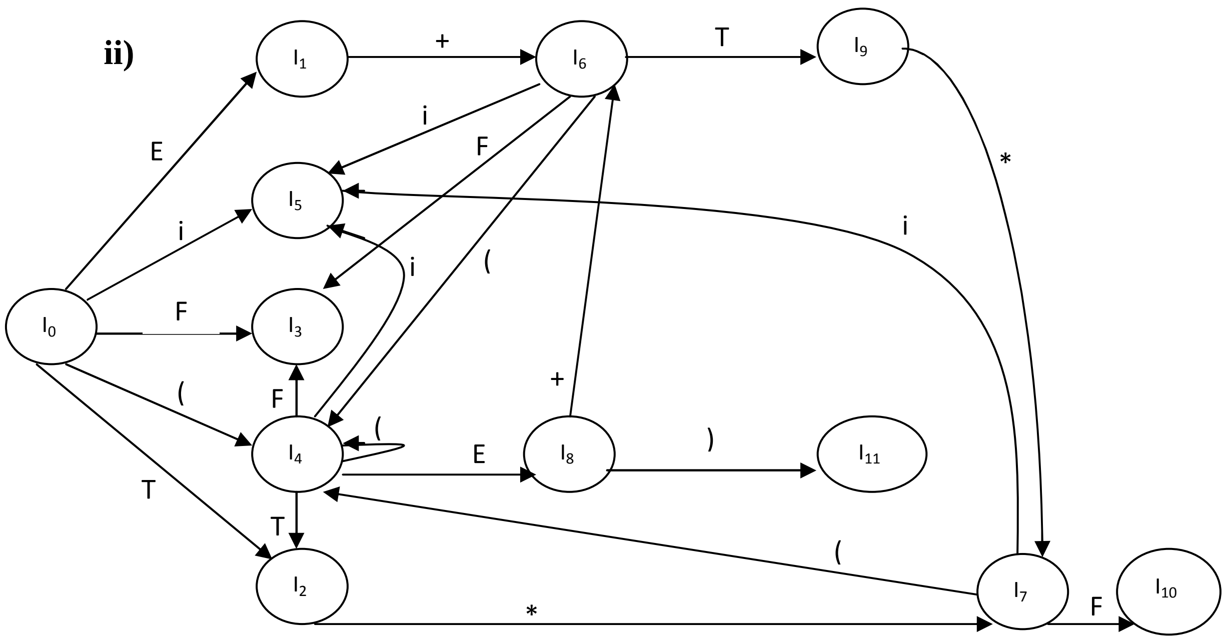


Fig. 5: Deterministic Finite Automaton D

3.4.2 Constructing SLR Parsing Table

In this section you will learn how to construct the SLR parsing action and GOTO functions from the deterministic finite automaton that recognises viable prefix. It will not produce uniquely-defined parsing tables for all grammars but does succeed on many grammars for programming languages. Given a grammar G, we augment G to produce G', and from G' we construct *C*, the canonical collection of sets of items for G'. We construct ACTION, the parsing action function, and GOTO, the goto function, from *C* using the following "simple" LR (SLR for short) parsing table construction technique. It requires you to know FOLLOW (A) for each nonterminal A of a grammar as you earlier learnt in section 3.4.2.1.2 of the previous unit.

3.4.2.1 Algorithm for Construction of an SLR Parsing Table

Input: *C*, the canonical collection of sets of simple items for an augmented grammar G'.

Output: If possible, an LR parsing table consisting of a parsing action function ACTION and a goto function, GOTO.

Method: Let $C = \{I_0, I_1, \ldots, I_n\}$. The states of the parser are 0, 1, …, *n*, state i being constructed from I_i. The parsing actions for state i are determined as follows:

1. If $[A \rightarrow \alpha.a\beta]$ is in I_i and GOTO $(I_i, a) = I_j$ then set ACTION (i, *a*) to "shift j". Here *a* is a terminal
2. If $[A \rightarrow \alpha.]$ is in I_i , then set ACTION(i, *a*) to "reduce $A \rightarrow \alpha$" for all *a* in FOLLOW(A). If ε is in FOLLOW (A), set ACTION (i, \$) to "reduce $A \rightarrow \alpha$".
3. If $[S' \rightarrow S.]$ is in I_j, then set ACTION (i, \$) to "accept"

REMARK: If any conflicting actions are generated by the above rules, then we say the grammar is not SLR (1). The algorithm fails to produce a valid parser in this case.

The GOTO transitions for state i are constructed according to the rules:

4. If GOTO $(I_j, A) = I_j$; then GOTO(i, A) = j
5. All entries not defined by rules (1) through (4) are made "error"
6. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S]$

The parsing table consisting of the parsing action and goto functions determined by the algorithm in section 3.4.2.1 above is called the *SLR table for G*. An LR parser using the SLR table for G is called the SLR parser for G, and a grammar having an SLR parsing table is said to be SLR(1)

Example 5:
Using the augmented grammar and the corresponding computed canonical collection of sets of items in example 4 above, construct an SLR (1) parsing table for the grammar.

We use S to denote states

$E' \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow i$

Table 1: SLR (1) Parsing Table for the Grammar in Example 5

State	← A C T I O N →						← GOTO →		
	i	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	S5			S4			1	2	3
1		S6				Accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Where Sj means "shift j" and rk means "reduce k"

Example 6:

Use the SLR parsing table developed in example 5 above to process the string:

i * i + i

Solution:

Table 2: String i*i+i Processing Table

Line	Stack	Input
	0	i * i + i $
	0i5	* i + i $
	0F3	* ij + i $
	0T2	i + i $
	0T2*7	+ i $
	0T2*7i5	+ i $
	0T2*7F10	+ i $
	0T2	+ i $
	0E1	+ i $
	0E1+6	i $
	0EH6i5	$
	0E1+6F3	$
	0E1+6T9	$
	0E1	$
	Accept	

SELF- ASSESSMENT EXERCISE

i. Consider the grammar
   S → AS | b
   A → SA | a
   Compute all the LR(0) items for the above grammar
   Construct an NFA whose states are the LR(0) items from (a)
   Show that the canonical collection of LR (0) items for the grammar is the same as the states of the equivalent DFA.

ii. Is the grammar SLR? If so, construct the SLR parsing table.

4.0 CONCLUSION

In this concluding unit of this module, you have been taken through the concept of LR parsing techniques and how to develop SLR (1) parser for any grammar. You also learnt how to use the generated SLR parser to parse sentences derived from the grammar. In the next module you will be taken through the code generation phase of the compiler.

5.0 SUMMARY

In this unit, you learnt that:

- a context-free grammar is called *LR grammar* when it is elaborated to enable scanning the input from left to right (denoted by L) and to produce a rightmost derivation (denoted by R)
- there are three techniques/algorithms to build tables for an “LR” parser viz:
  - SLR
  - Canonical LR
  - LALR
- an LR parser consists of two parts, a driver routine and a parsing table
- the driver routine is the same for all LR parsers, only the parsing table changes from one parser to another.

6.0 TUTOR-MARKED ASSIGNMENT

i. Consider the grammar
   S → L = R | R
   L → *R | i
   R → L
   a. Compute all the LR(0) items for the above grammar
   b. Construct an NFA whose states are the LR(0) items from (a)

c. Show that the canonical collection of LR (0) items for the grammar is the same as the states of the equivalent DFA.

d. Is the grammar SLR? If so, construct the SLR parsing table.

ii. Consider the following SLR (1) grammar generating strings of balanced symbols *a* and *b* terminating at end marker *c* :

$$E' \to Ec$$
$$E \to ES \mid S$$
$$S \to aEb \mid ab$$

a. Develop the canonical LR(0) collection of sets of items for this grammar using the sets of items construction algorithm.

b. Construct the SLR parsing table

c. Use the developed parsing table to process the string: *aababbc*\$

7.0 REFERENCES/FURTHER READING

Aho, Sethi, & Ullman (1986). *Compilers: Principles, Techniques, and Tools*. -Wesley: Addison

Alfred, Aho,V. *et al*. (2007).*Compilers: Principles, Techniques, and Tools*. Second edition. Wesley: Pearson Addison.

Alfred, V. Aho & Ullman, Jeffrey D. *The Theory of Parsing, Translation, and Compiling*, available from ACM.org

Appel, Andrew W. (2002). *Modern Compiler Implementation in Java*. Second edition Cambridge University Press.

Chapman, Nigel P. (1987). *LR Parsing: Theory and Practice*. Cambridge University Press.

Cooper, Keith D. & Torczon, Linda (2004). *Engineering a Compiler*. Morgan Kaufmann.

CS.VU.nl, Parsing Techniques - A Practical Guide web page of book includes downloadable pdf.

Hopcroft, J. E. & Motwani, R. (2000).*Introduction to Automata Theory, Languages, and Computation* . Wesley: Addison

Leermakers, R. (1993). The Functional Treatment of Parsing. Boston: Kluwer Academic.

Muchnick, Steven S. (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Scott, Michael L. (2009). *Programming Language Pragmatics*. Third edition, Morgan Kaufman.

Sebesta, Robert W. (2010). *Concepts of Programming Languages*. Ninth edition, Wesley: Addison.

MODULE 4 CODE GENERATION

UNIT 1 ERROR HANDLING

CONTENTS

1.0 INTRODUCTION

So far in this course you have learnt about the front end of the compiler. In this concluding module of the course you will be learning more about the back end of the compiler namely, intermediate code generation, code generation and code optimisation. But this first unit will be introducing you to the concept of error detection and recovery in compilers.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- classify errors based on the stage at which they occur
- design a better error handling compiler for a particular programming language
- distinguish between runtime errors and compile time errors
- explain how error detection and recovery mechanism of a compiler can affect the performance of an application.

3.0 MAIN CONTENT

3.1 Dealing with Errors

Even experienced programmers make mistakes, so they appreciate any help a compiler can provide in identifying the mistakes. Novice programmers may make lots of mistakes, and may not understand the programming language very well, so they need clear, precise and jargon-free error reports. Especially in a learning environment, the main function of a compiler is to report errors in source programmes; as an occasional side-effect you might actually get a programme translated and run.

As a general rule, compiler writers should attempt to express error messages in moderately plain English, rather than with reference to the official programming language definition (some language definitions use somewhat obscure or specialised terminology).

For example, a message "can't convert string to integer" is probably clearer than "no coercion found."

3.2 Historical Notes

In the 1960s and much of the 1970s, batch processing was the normal way of using a (large) mainframe computer (personal computers only started to become household items in the early 1980s). It could well be several hours, or even a day, from when you handed your deck of punched cards to a receptionist until you could collect the card deck along with a printed listing of your programme, accompanied either by error messages or by some useful results.

Under such circumstances, it was important that compilers report as many errors as possible, so part of the job of writing a compiler was to 'recover' from an error and continue checking (but not translating) in the hope of finding more errors. Unfortunately, once an error has occurred (especially if the error affects a declaration), it is quite possible for the compiler to get confused and produce a host of spurious error reports.

Programmers then had the task of deciding which errors to try and fix, and which ones to ignore in the hope that they would vanish once earlier errors were fixed. Some compilers were particularly prone to producing spurious error reports. The only useful advice that helpdesk staff could provide was: fix the first error, since the compiler hasn't had a chance to confuse itself at that point.

A significant amount of compiler development effort was often devoted to attempts at error recovery. You could try and guess what the programmer might have intended, or insert some token to at least allow parsing to continue or just give up on that statement and skip to the next semicolon. The latter action could skip an **end** or other significant programme structure token and so get the compiler even more confused.

3.3 Integrated Development Environment (IDE)

Fast personal computers are now available, so IDEs are becoming more popular, with an editor and compiler tightly coupled and usable from a single graphical interface. Many IDEs also include a debugger as well. In some cases the editor is language-sensitive, so it can supply matching brackets and/or statement schemas to help reduce the number of trivial errors. An IDE may also use different colours for different concepts within a source language, e.g. reserved words in **bold**, comments in green, constants in blue, or whatever.

This speed and tight coupling allows the compiler writer to adopt a much simpler approach to errors: the compiler just stops as soon as it finds an error, and the editor then places the cursor at the point in the source text where the error was detected and displays some specific error message. Note that the point where an error was detected could well be some distance after the point where the error actually occurred.
There were line-mode IDEs back in 1964, many BASIC systems were examples of such systems; we are going to implement something like this in the book section case study - a simple interpreter.

3.4 Compile-time Errors

During compilation it is always possible to give the precise position at which the error was detected. This position could be shown by placing the editor cursor at the precise point, or (batch mode) by listing the offending line followed by a line containing some sort of flag (e.g.'|') positioned under the point of error, or (less conveniently) by providing the line number and column number of that point.

Remember that the actual position of the error (as distinct from where it was detected) may well be at some earlier point in the programme; in some cases (e.g. bracket mismatch) the compiler may be able to indicate the nature of the earlier error.

It is important that error messages be clear, correct, and relevant.

The worst counter-example that Murray Langton has encountered was a compiler which reported "Missing semicolon" when the actual error was an extra space in the wrong place. To further confuse matters, no indication was given as to where in the programme the error was. Just to add insult to injury, the source language didn't even use semicolons!

3.4.1 Errors during Lexical Analysis

There are relatively few errors which can be detected during lexical analysis. Some of these are as follows:

- **Strange characters:** Some programming languages do not use all possible characters, so any strange ones which appear can be reported. Note however that almost any character is allowed within a quoted string.
- **Long quoted strings I:** Many programming languages do not allow quoted strings to extend over more than one line; in such cases a missing quote can be detected. Languages of this type often have some way of automatically joining consecutive quoted strings together to allow for really long strings.
- **Long quoted strings II:** If quoted strings can extend over multiple lines then a missing quote can cause quite a lot of text to be 'swallowed up' before an error is detected. The error will probably then be reported as somewhere in the text of the next quoted string, which is unlikely to make sense as part of a programme.
- **Invalid numbers:** A number such as 123.45.67 could be detected as invalid during lexical analysis (provided the language does not allow a full stop to appear immediately after a number). Some compiler writers prefer to treat this as two consecutive numbers 123.45 and .67 as far as lexical analysis is concerned and leave it to the syntax analyser to report an error. Some languages do not allow a number to start with a full stop/decimal point, in which case the lexical analyser can easily detect this situation.

3.4.2 Errors during Syntax Analysis

During syntax analysis, the compiler is usually trying to decide what to do next on the basis of expecting one of a small number of tokens. Hence in most cases it is possible to automatically generate a useful error message just by listing the tokens which would be acceptable at that point.

```
Source:  A + * B
Error:      | Found '*', expect one of: Identifier, Constant, '('
```

More specific hand-tailored error messages may be needed in cases of bracket mismatch.

```
Source:  C := ( A + B * 3 ;
Error:                  | Missing ')' or earlier surplus '('
```

3.4.3 Errors during Semantic Analysis

One of the most common errors reported during semantic analysis is "identifier not declared"; either you have omitted a declaration or you have misspelt an identifier.

Other errors commonly detected during semantic analysis relate to incompatible use of types, e.g. attempt to assign a logical value such as **true** to a string of characters. Some of these errors can be quite subtle, but again it is easy to automatically generate fairly precise error messages.

Source: SomeString := **true**;
Error: Can't assign logical value to character string

The extent to which such type checking is possible depends very much on the source language.

PL/1 allows an amazingly wide variety of automatic type conversions, so relatively little checking is possible.

Pascal is much more fussy; you can't even assign a real value to an integer variable without explicitly specifying whether you want the value to be rounded or truncated.

Some writers have argued that type checking should be extended to cover the appropriate units as well for even more checking, e.g. it doesn't make sense to multiply a distance by a temperature.

Other possible sources of semantic errors are parameter miscount and subscript miscount. It is generally an error to declare a subroutine as having 4 parameters and then call that routine with 5 parameters (but some languages do allow routines to have a variable number of parameters). It is also generally an error to declare an array as having 2 subscripts and then try and access an array element using 3 subscripts (but some languages may allow the use of fewer subscripts than declared in order to select a 'slice' of the array).

3.5 Reporting the Position of Run-Time Errors

There is general agreement that run-time errors such as division by 0 should be detected and reported. However, there is considerable variation as to how the location of the error is reported.

Some systems merely provide the hexadecimal address of the offending instruction. If your compiler/linker produced a load map you might then be able to do some hexadecimal arithmetic to identify which routine it is in.

Some systems do tell you the name of the routine the error was in, and possibly the names of all the routines which were active at the time.

A few kind systems give you the source line number, which is very helpful. Note however that extensive programme optimisation can move code around and intermingle statements, in which case line numbers may only be approximate. From the implementor's viewpoint there are several ways in which line number details or equivalent can be provided. The compiled programme can contain instructions which place the current line number in some fixed place; this makes the programme longer and slower. Of course the compiler need only add these instructions for statements which can actually cause an error.

The compiled programme can contain a table indicating the position at which each source line starts in the compiled code. In the event of an error, special code can then consult this table and determine the source line involved. This makes the compiled code longer but doesn't slow it down.

In some unoptimised systems, it may be possible to deduce some source information from the compiled code, e.g. the Elliott 503 Algol 60 compiler could report: "divide by 0 at second division after third **begin** of routine 'xyz'". This doesn't affect code size or speed, but may not always be feasible to implement.

3.6 Run-Time Speed versus Safety

There are some potential run-time errors which many systems do not even try to detect. The language definition may merely say that the result of breaking a certain language rule is undefined, i.e. you might get an error message, or you might get the wrong answer without any warning, or you might on some occasions get the right answer, or you might get a different answer every time you run the programme, or you might trigger off World War III ('undefined' does mean that anything could happen).

In the past there have been some computers (Burroughs 5000+, Elliott 4130) which had hardware support for fast detection of some of these errors. Many current IDE's do have a debugging option which may help detect some of these run-time errors:

- attempt to divide by 0
- overflow (and possibly underflow) during arithmetic operations.
- attempt to use a variable before it has been set to some sensible value (undefined variable)
- attempt to refer to a non-existent array element (invalid subscript).
- attempt to set a variable (defined as having a limited range) to some value outside this range
- various errors related to pointers:
- Attempt to use a pointer before it has been set to point to somewhere useful.
- attempt to use a **nil** pointer, which explicitly doesn't point anywhere useful
- attempt to use a pointer which points outside the array it should point to.
- attempt to use a pointer after the memory it points to has been released.

Historically, the main reason for not doing these checks is the effect on performance. When FORTRAN was first developed (circa, 1957), it had to compete with code written in assembler; indeed many of the optimising techniques used in modern compilers were first developed and used at that time. C was developed (circa, 1971) initially as a replacement for assembler for use by experienced system programmers when writing operating systems.

In both the above cases there was a justifiable reason for not doing these checks. Nowadays, computer hardware is very much faster than it was in 1957 or 1971, and there are many more less-experienced programmers writing code, so the arguments for avoiding checks are much weaker. Actually adding the checks on a supposedly working programme can be enlightening/surprising/embarrassing; even programmes which have been 'working' for years may turn out to have a surprising number of bugs.

Hoare was responsible for an Algol 60 compiler in the early 1960s; subscript checking was always done. Indeed Hoare has said in "Hints on Programming Language Design" that: "Carrying out checks during testing and then suppressing then in production is like a sailor who wears a lifejacket when training on dry land and then removes the lifejacket when going to sea."

In his book "The Psychology of Computer Programming", Wienberg recounts the following anecdote:

- After months of effort, a particular application was still not working, so a consultant was called in from another part of the company. He concluded that the existing approach could never be made to work reliably. While on his way home he realised how it could be done. After a few days work he had a demonstration programme working and presented it to the original programming team.
- Team leader: How long does your programme take when processing?
- Consultant: About 10 seconds per case.
- Team leader: But our programme only takes 1 second. {Team look smug at this point}
- Consultant: But your programme doesn't work. If the programme doesn't have to work then I can make it as fast as you like.
- Wirth designed Pascal as a teaching language (circa 1972); for many Pascal compilers the default was to perform all safety checks. Some Pascal systems had an option to suppress the checks for some limited part of the programme.
- When a programming language allows the use of pointers and pointer arithmetic for accessing array elements, the cost of doing checks for access to non-existent array elements might be significant. Note that it can indeed be done: each pointer is large enough to contain three addresses, the first being the one which is directly manipulated and used by the programmer, and the other two addresses being the lower and upper limits on the first. This approach may have problems when the language allows interconversion between integers and pointers.
- In the case of 'undefined variables', note that setting all variables initially to 0 is a really **bad idea** (unless the language mandates

this of course). Such an initial setting reduces programme portability and may also disguise serious logic errors.

3.6.1 Cheap Detection of 'Undefined'

Murray Langton has had some success in checking for 'undefined' in a 140,000 line safety-critical legacy FORTRAN programme. The fundamental idea is to set all global variables to recognisably strange values which are highly likely to produce visibly strange results if used. For an IBM mainframe, the strange values were:

- REAL set to -9.87654E70
- INTEGER set to -123456789
- CHAR set to '?'

Note that the particular values used depend on your system; in particular the large number used for REAL is definitely hardware-dependent. For a machine with IEEE floating point arithmetic (most PC's) the best choice for REAL is NaN (not a number), with a possible alternative being -9.87654E37.

The reason for choosing large negative numerical values is that they tend to be very obvious when printed or displayed as output, and they tend to cause numerical errors (overflow) if used for arithmetic. Also, in FORTRAN, all output is in fixed-width fields, and any output which will not fit in the field is displayed as a field full of asterisks instead, which is very easy to spot.

In the safety-critical example quoted above, a programme was written which identified all global variables (by analysing COMMON blocks), excluded those (in BLOCK DATA) which were explicitly initialised, and then wrote a FORTRAN routine which set all these silly values. If any changes were made to a COMMON block, it was a simple matter to rerun this analysis programme.

During execution, the routine which sets silly values uses less than 0.1% of the total CPU time. When these silly values were first used, it took several months to track down and eliminate the resulting flood of asterisks and question marks which appeared in the output, despite the fact that the program had been 'working' for over 20 years.

3.6.2 How to Check for 'Undefined'

The basic idea is to ensure that all variables are flagged as 'undefined' when declared. Some languages allow simultaneous declaration and initialization, in which case a variable is flagged as 'defined'. Whenever a value is assigned to a variable the flag is changed to 'defined'. Whenever a variable is used the flag is checked and an error is reported if it is 'undefined'.

In the past a few lucky implementors have had hardware assistance in the form of an extra bit attached to each word in memory (Burroughs 5000+). On modern byte-addressable machines you could attach an extra byte to each variable to hold the flag. Unfortunately, due to alignment requirements, this would tend to double the amount of memory needed for data (many systems require 4-byte items such as numbers to have an address which is a multiple of 4; even if misalignment is allowed its use may slow the programme down significantly).

The simplest way of providing a flag is to use some specific value which is (hopefully) unlikely to appear in practice. Particular values depend on the type of the variable involved.

- **Boolean:** Such variables are most likely to be allocated one byte of storage with 0 for false and 1 for true. A value such as 255 or 128 is a suitable flag.
- **Character:** When used for binary input/output, any value could appear, so no checking is possible. Hence it must be possible to switch off checking in such cases.
  When used as a character there are many possible non-printing characters. 127 or 128 or 255 may be suitable choices.
- **Integer:** Most computer systems use two's complement representation for negative numbers which gives an asymmetric range (for 16-bits, range is -32768 to +32767). We can restore symmetry by using the largest negative number as the 'undefined' flag.
- **Real:** If your hardware conforms to the IEEE standard (most PC's do) you can use NaN (not a number).

3.6.3 How to Check at Compile-Time

You may well be thinking that all this checking (for undefined, bad subscript, out of range, etc.) is going to slow a programme down quite a lot. Things are not as bad as you think, since a lot of the checking can actually be done at compile-time, as detailed below.

- First, some statistics to show you what can be done:
- Just adding checking to an existing compiler resulted in 1800 checks being generated for a 6000-line programme.
- Adding a few hundred lines to the compiler allowed it do many checks at compile-time, and reduced the number of run-time

checks to just 70. The programme then ran more than 20% faster than the version with all checks included.

- We have already mentioned that variables which are given an initial value when declared need never be checked for undefined.
- The next few tests require some simple flow-control analysis e.g. variables which are only set in one branch of an **if** statement become undefined again after the **if** statement, unless you can determine that a variable is defined on all possible branches.
- Once a variable has been set (by assignment or by reading it from a file) it is then known to be defined and need not be tested thereafter.
- Once a variable has been tested for 'undefined', it can be assumed to be defined thereafter.
- If your programming language allows you to distinguish between input and output parameters for a routine, you can check as necessary before a call that all input parameters are defined. Within a routine you can then assume that all input parameters are defined.
- For discrete variables such as integers and enumerations, you can often keep track at compile time of the maximum and minimum values which that variable can have at any point in the programme. This is particularly easy if your source language allows variables to be declared as having some limited range (e.g. Pascal). Of course any assignment to such a bounded variable must be checked to ensure that the value is within the specified range.
- For many uses of a bounded variable as a subscript, it often turns out that the known limits on the variable are within the subscript range and hence need not be checked.
- In a count-controlled loop you can often check the range of the control variable by checking the loop bounds before entering the loop which may well reduce the subscript checking needed within the loop.

SELF-ASSESSMENT EXERCISE

List out some of the run-time errors that can be encountered and how they can be reported and handled

4.0 CONCLUSION

In this unit, you have been taken through the concept of error detection and recovery in compilers. Certain errors occur at certain stage of compiling and the way the error is handled depends on the type of error and at what stage it was detected. In the next unit you will be learning about another concept that is paramount to error handling which is symbol table organisation.

5.0 SUMMARY

In this unit, you learnt that:

- compiler writers should attempt to express error messages in moderately plain English, rather than with reference to the official programming language definition
- during compilation it is always possible to give the precise position at which the error was detected
- it is important that error messages be clear, correct, and relevant
- there is variation as to how the location of the error of division by 0 is reported
- the extent to which type checking is possible depends very much on the source language
- in a count-controlled loop you can often check the range of the control variable by checking the loop bounds before entering the loop.

6.0 TUTOR-MARKED ASSIGNMENT

i. Briefly itemise and describe the different stages at which errors might occur during compilation.
ii. Outline the different ways that error of division by 0 can be reported.
iii. Distinguish between run-time errors and compile time errors.
iv. Briefly describe some ways to check for ‘undefined.’

7.0 REFERENCES/FURTHER READING

Alfred, Aho, V. *et al.* (2007). *Compilers: Principles, Techniques, and Tools*. Second edition. Wesley: Pearson Addison.

Alfred, V. Aho & Jeffrey, D. Ullman. *The theory of Parsing, Translation, and Compiling,* available from ACM.org

Andrew, Appel W. (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Chapman, Nigel P. (1987). *LR Parsing: Theory and Practice*. Cambridge University Press.

CS.VU.nl, Parsing Techniques - A Practical Guide web page of book includes downloadable pdf

Keith, D. Cooper & Linda, Torczon (2004). *Engineering a Compiler*. Morgan Kaufmann.

Leermakers, R. (1993). *The Functional Treatment of Parsing*. Boston: Kluwer Academic.

Michael, L. Scott (2009). *Programming Language Pragmatics*. Third edition. Morgan Kaufman.

Robert, W. Sebesta (2010). *Concepts of Programming Languages*. Ninth edition. Addison-Wesley.

Steven, S. Muchnick (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

UNIT 2 SYMBOL TABLES

CONTENTS

1.0 INTRODUCTION

A compiler needs to collect and use information about the names appearing in the source programme. This information is usually entered into a data structure called a symbol table. The information collected about a name includes the string of characters by which it is denoted, its type, its form, its location in memory and other attributes depending on the language. Information about a name is entered into the table during lexical and syntactic analysis.

The information collected in the symbol table is used during several stages in the compilation process. There are also a number of ways in which the symbol table can be used to aid in error detection and correction. Also space in the symbol table can be used for code optimisation purposes.

In this unit you will learn the principal ways of organising and accessing symbol tables.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define symbol tables
- state their uses in the compilation process
- list item that usually entered into symbol tables
- describe ways of organising the symbol table
- list the kind of information needed by the compiler
- construct symbol table for block-structured programs
- describe collision resolution methods in hashing and advantages.

3.0 MAIN CONTENT

3.1 Semantic Analysis

Semantic analysis is roughly the equivalent of checking that some ordinary text written in a natural language (e.g. English) actually means something (whether or not that is what it was intended to mean).

The purpose of semantic analysis is to check that we have a meaningful sequence of tokens. Note that a sequence can be meaningful without being correct; in most programming languages, the phrase "I + 1" would be considered to be a meaningful arithmetic expression. However, if the programmer really meant to write "I - 1", then it is not correct.

3.2 Symbol Tables

A compiler uses a symbol table to keep track of scope and binding information about names.

The symbol table is searched every time a name is encountered in the source text. Changes to the symbol table occur if a new name or new information about an existing name is discovered.

A symbol table mechanism must allow us to add new entries and find existing entries. The two symbol table mechanisms are 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.

It is useful for a compiler to be able to grow the symbol table dynamically at compile time. If the symbol table is fixed when the compiler is written, the size must be chosen large enough to handle any source programme that might be presented

For *compile-time* efficiency, compilers use a *symbol table* which associates lexical *names* (symbols) with their *attributes*

The items that are usually entered into a symbol table are:

- variable names
- defined constants
- procedure and function names
- literal constants and strings
- source text labels
- compiler-generated temporaries

usually separate tables are constructed for structure layouts (types) (*field offsets and lengths*)

Therefore, a symbol table is a compile-time structure

3.2.1 Symbol Table Information/Entries

Each entry in the symbol table is for the declaration of a name. The format of entries does have to be uniform, because the information saved about a name depends on the usage of time. Each entry can be implemented as a record consisting of a sequence of consecutive words of memory. To keep symbol table entries uniform, it may be convenient for some of the information about a name to be kept outside the table entry, with only a pointer to this information stored in the record.

Information is entered into the symbol table at various times. Keywords are entered initially. The lexical analyser looks up sequences of letters and digits in the symbol table to determine if a reserved keyword or a name has been collected. With this approach, keywords must be in the symbol table before lexical analysis begins. Alternatively, if lexical analyser intercepts reserved keywords, they should be entered into the symbol table with a warning of their possible use as a keyword.

The symbol table entry itself can be set up when the role of a name becomes clear, with the attribute values being filled in as the information become available. In some cases, the entry can be initiated from the lexical analyser as soon as a name is seen in the input. More often, one name may denote several different objects, even in the same block or procedure.

For example, the C declarations

```
int x;
struct x {float y, z;};
```

use x as both an integer and as the tag of a structure with two fields. In such cases, the lexical analyser can only return to the parser the name itself rather than a pointer to the symbol table entry. The record in the symbol table is created when the syntactic role played by the name is discovered. Attributes of a name are entered in response to declarations. Labels are identifiers followed by a colon, so one action associated with recognising such an identifier may be to enter this fact into symbol table.

The compiler usually needs the following kind of information:

- textual name
- data type
- dimension information (*for aggregates*)
- declaring procedure
- lexical level of declaration
- storage class (*base address*)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions

3.2.2 Symbol Table Organisation

Symbol tables may be implemented using linear lists, hash-tables and various sorts of tree structures. An issue is the speed with which an entry can be added or accessed. A linear list is slow to access but simple to implement. A hash table is fast but more complex. Tree structures give intermediate performance. In summary, each of these structures has the following characteristics:

Linear List

- **O**(*n*) probes per lookup
- easy to expand — no fixed size

- one allocation per insertion

Ordered Linear List

- **O**(log2 *n*) probes per lookup using binary search
- insertion is expensive (to reorganise list)

Binary Tree

- **O**(*n*) probes per lookup, if the tree is unbalanced
- **O**(log2 *n*) probes per lookup, if the tree is balanced
- easy to expand with no fixed size
- one allocation per insertion

Hash Table

- **O**(1) probes per lookup on the average
- expansion costs vary with specific scheme

In the abstract, a symbol table is merely a table with two fields, a name field and an information field. We require several capabilities of the symbol table. We need to be able to:

- determine whether a given name is in the table,
- add a name to the table,
- access the information associated with a given name, and
- add new information for a given name,
- delete a name or group of names from the table.
- in a compiler, the names in the symbol table denote object of various sorts. There may be separate tables for variable names, labels, procedure names, constants, field names (for structures) and other types of names depending on the programming language.

3.2.3 Attribute information

Attributes are internal representation of declarations. Symbol table associates names with attributes. Names may have different attributes such as below depending on their meaning:

a. **Variables**: type, procedure level, frame offset
b. **Types**: type descriptor, data size/alignment
c. **Constants**: type, value
d. **Procedures**: formals (names/types), result type, block information (local declarations), frame size.

3.2.4 Characters in a Name

There is a distinction between the token id for an identifier or name, the lexeme consisting of the character string forming the name, and the attributes of the name. The lexeme is needed when a symbol table entry is set up for the first time and when we look up a lexeme found in the input to determine whether it is a name that has already appeared. A common representation of a name is a pointer to a symbol table entry for it.

If there is a modest upper bound on the length of a name, then the characters in the name can be stored in the symbol table entry as shown in figure. If there is no limit on the length of a name the indirect scheme can be used. Rather than allocating in each symbol table entry the maximum possible amount of space to hold a lexeme, utilise space more efficiently if there is only space for a pointer in the symbol table entry. In the record for a name, a pointer is placed to a separate array of characters giving the position of the first character of the lexeme. The indirect scheme permits the size of the name field of the symbol table entry itself to remain a constant.

The complete lexeme constituting a name must be stored to ensure that all uses of the same name can be associated with the symbol table record.

Table 1: Symbol Table Record

Name										Attributes
s	o	r	t							
a										
r	e	a	d	a	r	r	a	Y		
i										

In fixed size space within a record.

3.3 Table Organisation

General form of symbol table organisation is as in Table 2 below:

Table 2: General Form of Symbol Table Organisation

	Argument	Value
Entry 1		
Entry 2		
.		
.		

Arguments are the symbols or identifiers while values are the attributes obtained from declaration and usage on the symbols. Arguments can be stored by letting the argument portion of the symbol table contain pointers to the actual symbols stored in the string space. The argument portion may also contain the length of the symbol in the string space.

Two ways to implement tables are:

i. One-table for all entries
ii. One-table for each entry i.e. if k-entries, then k-tables.

A sorted table is searched using binary search and the average search number of comparisons is $n\log_2 n$. An unsorted table requires on the average $\frac{n}{2}$ comparisons.

3.3.1 Storage Allocation Information

Information about the storage locations that will be bound to names at run time is kept in the Symbol table. If machine code is to be generated by the compiler, then the position of each data object relative to a fixed origin, such as the beginning of an activation record must be ascertained. The same remark applies to a block of data loaded as a module separate from the programme. For example, COMMON blocks in FORTRAN are loaded separately, and the positions of names relative to the beginning of the COMMON block in which they lie must be determined.

3.3.2 The List Data Structure for Symbol Tables

The simplest and easiest to implement data structure for a symbol table is a linear list of records. Arrays are used to store their names and their associated information. New names are added to the list in the order in which they are encountered. The position of the end of the array is marked by the pointer available, pointing to where the next symbol table entry will go. The search for a name proceeds backwards from the end of the array to the beginning. When the name is located, associated information can be found in the words following next. If we reach the beginning of the array without finding the name, a fault occurs- an expected name is not in the table.

Making for an entry for a name and looking up the name in the symbol table are independent operations. In a block-structured language, an occurrence of a name is in the scope of the most closely nested declaration of the name. This scope can be implemented by making a fresh entry for a name every time it is declared. A new entry is made in the words immediately following the pointer available; that pointer is increased by the size of the symbol table record. Since entries are inserted in order, starting from the beginning of the array, they appear in the order they are created in.

Table 3: A Linear List of Records

id1	
Info1	
Id2	
Info2	
………..	available________
Info*n*	

If the symbol table contains *n* names the work necessary to insert a new name is constant if we do the insertion without checking to see the name is already in the table. If multiple entries for names are not allowed look the entire table before discovering that a name is not in the table. Insertions and inquiries take time proportional to *n*, names and m inquiries is at most $c\ n\ (n + e), c$ is a constant.

3.3.3 Hash Addressing

This is a method for converting symbols into indices of *n*-entries in the symbol table with fixed size. Collision occurs when two or more symbols hashed into the same index.

3.3.3.1 Hash Tables

Variation of the searching technique is known as hashing. Open hashing refers to the property that there need be no limit on the number of entries that can be in the table. This scheme gives the capability of performing e enquiries on *n* names in time proportional to *n (n + e)/m*. Since *m* can be made large up to n this is more efficient than linear list. In the basic hashing scheme, there are two parts to the data structure:

a. A hash table consisting of a fixed array of m pointers to table entries.
b. Table entries organised into m separate linked lists, called buckets. Each record in the symbol table appears on exactly one of these lists. Storage for the records may be drawn from an array of records. The dynamic storage allocations facilities of the implementation language can be used to obtain space for the records.

To determine whether there is an entry for string s in the symbol table, apply a hash function *h* to *s*, such that *h(s)* returns an integer between 0 and *m*-1. If s is in the symbol table, then it is on the list numbered *h(s)*. If s is not in the symbol table, it is entered by creating a record for s that is linked at the front of the list numbered *h (s)*. The average list is *n/m* record long if there are *n* names in a table of size *m*. By choosing m so that *n/m* is bounded by a small constant the time to access a table entry is constant. This space taken by the symbol table consists *m* words for the hash table and *cn* words for the table entries, where *c* is the number of words per table entry. Thus the space for the hash table depends only on *m* and the space for table entries depends only on the number of entries.

The choice of *m* depends on the intended application for a symbol table. One suitable approach for computing hash functions is to proceed as follows:

a. Determine a positive integer *h* from the characters *c1, c2, ..., ck* in string *s*. The conversion of single characters to integers is usually supported by the implementation language.
b. Convert the integer *h* determined above into the no. of the list. Divide by m and take the reminder.

A technique for computing *h* is to add up the integer values of the characters in a string. Multiply the old value of *h* by a constant @ before adding in the next character. That is, h0=0, hi=@ hi-1+ci

1. #define PRIME 211

```
#define EOS '\0'
int hashpjw(s)
char      *s;
{
char  *p;
unsigned h=0, g;
for( p=0;*p !=EOS; p=p+1){
h=(h << 4)+(*p)'
if(g= h@0xf0000000){
h=h^(g >>24);
h=h ^ g ;
}
}
return h % PRIME;
}
```

In the hash function *hashpjw*, the sizes included the first primes larger than 100,200,...,1500. A close second was the function that computed the old value by 6559,i goring overflows, adding in the next character. Function *hashpjw* is computed by starting with h=0. For each character *c*, shift bits of *h* left 4 positions and add in *c*. If any of four high-order bits of h is 1, shift four bits right 24 positions , exclusively-or them into *h,* and reset to 0 any of the high order bits that was 1.

pThe sim

3.3.4 Collision Resolution

Two ways of resolving collision are

i. Re-hashing
ii. Chaining

3.3.4.1 Re-Hashing

Suppose we hash a symbol *s* to *h* and find that a different symbol already occupies the entry *h*. Then a collision has occurred. We then compare *s* against an entry h + p_1(modulo the table size) for some integer p_1. If a collision occurs again, we compare with an entry h + p_2 (modulo the table size) for some integer p_2.

This continues until $h = h + p_i$ (modulo table size) is empty, contains *s* or is again entry *h*. In other words $p_i = 0$.

In the latter case, we stop the programme because the table is full.

If $\{P_i\}$ is the set if natural numbers then it is a linear rehash otherwise it is non-linear.

3.3.4.2 Chaining

Suppose we hash a symbol *s* to *h* and find that a different symbol already occupies the entry *h*, a collision has occurred.

Chaining method uses a hash table called bucket of a fixed size as the symbol table. It is a table of pointers to the elements of the symbol table and points to nothing initially. Another pointer points to the last symbol entered into the symbol table. Symbols hash to buckets of the hash table. Each bucket points to nil or to the first element in the symbol table that hashes to it.

Symbols are entered in first-come-first-served FCFS manner in the symbol table. The symbol table has an additional pointer field used to chain entries which hash to the same bucket.

Bucket is a table of pointers

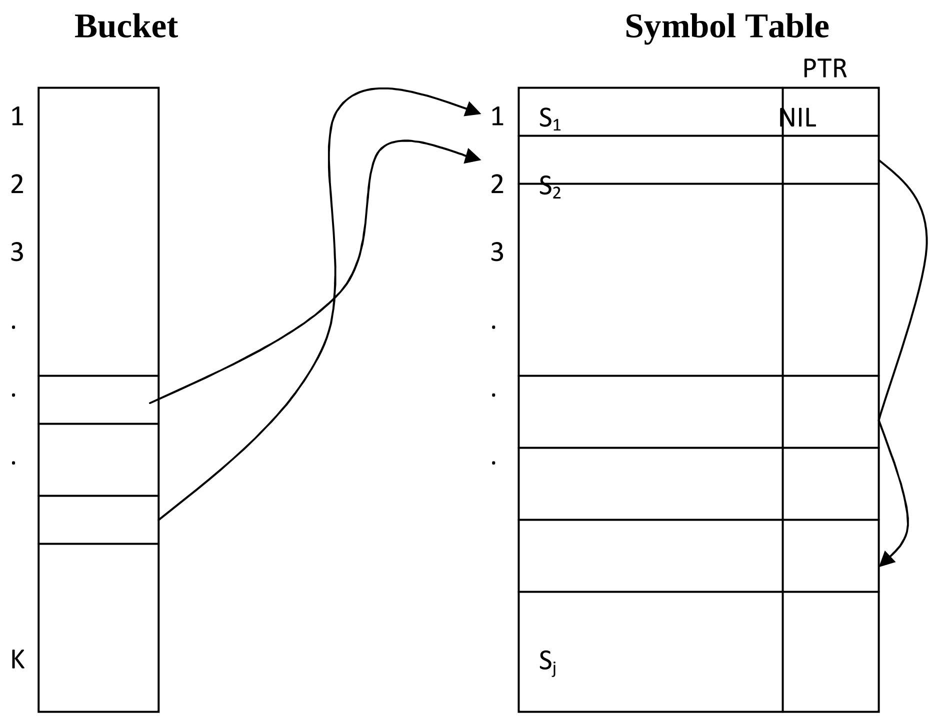


Fig. 1: Chaining

If s_2, s_j, s_m hash into K, the chain is as above.

3.3.5 Representing Scope Information

The entries in the symbol table are for declarations of names. When an occurrence of a name in the source text is looked up in the symbol table, the entry for the appropriate declaration of that name must be returned.

A simple approach is to maintain a separate symbol table for each scope. Information for the nonlocals of a procedure is found by scanning the symbol tables for the existing programme. With this approach the symbol table is integrated into the intermediate representation of the input. Most closely nested scope rules can be implemented by adapting the data structures. We keep track of the local names of the procedure by giving each procedure a unique number. The number of each procedure can be computed in a syntax directed manner from semantic rules that recognise the beginning and ending of each procedure. The procedure number is made a part of all locals declared in the procedure.

When we look up a newly scanned name,a match occurs only if the characters of the name match an entry character for character,and the associated number in the symbol table entry is the number of the procedure which is processed. Most closely nested scope rules can be implemented in terms of the following operations on a name:

- Lookup : find the most recently created entry
- Insert : make a new entry
- Delete : remove the most recently created entry.

Deleted entries must be preserved, they are just removed from the active symbol table. In a one-pass compiler ,information in the symbol table about a scope consisting of a procedure body, is not needed at compile time after the procedure body is processed. However ,it may be needed at run time. In this case, the information in the symbol table must be added to the generated code for user by the linker.

When a linear list consisting of an array of records was described, it was said that lookup can be implemented by inserting entries at one end .A scan starting from the end and proceeding to the beginning of an array, finds the most recently created entry for the name. A pointer front point to the most recently created entry in the list. The implementation of the insert takes constant time because a new entry is created at the front of the list. The implementation of the lookup is done by scanning the list starting from entry pointed by front and following links until the desired one is found.

A hash table consists of *m* lists accessed through an array. For implementing the delete operation, we would rather not have to scan the entire hash table. Suppose each entry has two links:

1. A hash link that chains the entry to other entries whose names hash to the same value
2. A scope link that chains all entries in the same scope.

Deletion of entries from the hash table must be done with the care, because deletion of an affects the previous one on its list. When we delete the i-1st entry points to i+1st entry.

The i-1st entry can be found if the hash links from a circular link list. We can use a stack to keep track of the lists containing entries to be deleted. A marker is placed in the stack when a new procedure is scanned. When we finish processing the procedure, the list numbers can be popped from the stack until the marker for the procedure is reached.

3.3.6 Implementation of Block Structured Languages

Languages with block structure such as ALGOL present certain complexities not found in C. First, in a block-structured language, not only procedures, but blocks as well, may define their own data. Thus activation records or portions of activation records must be reserved for blocks. Second, many languages, ALGOL, for example, permit arrays of adjustable length. Third, the data-referencing environment of a procedure or block includes all procedures and blocks surrounding it in the programme.

Consider the code segment below:

```
1       begin
                real a, B, C, D;
                  ⋮
                begin real E, F;
LI :
⋮
end
begin
real  G, H;
               L2 : begin
        4                  real A ;
                   end
                      L3:
                end
        end
```

Fig. 2: **Block Structured Programme**

The symbol table for this block structure can be arranged as in Figure 3 below:

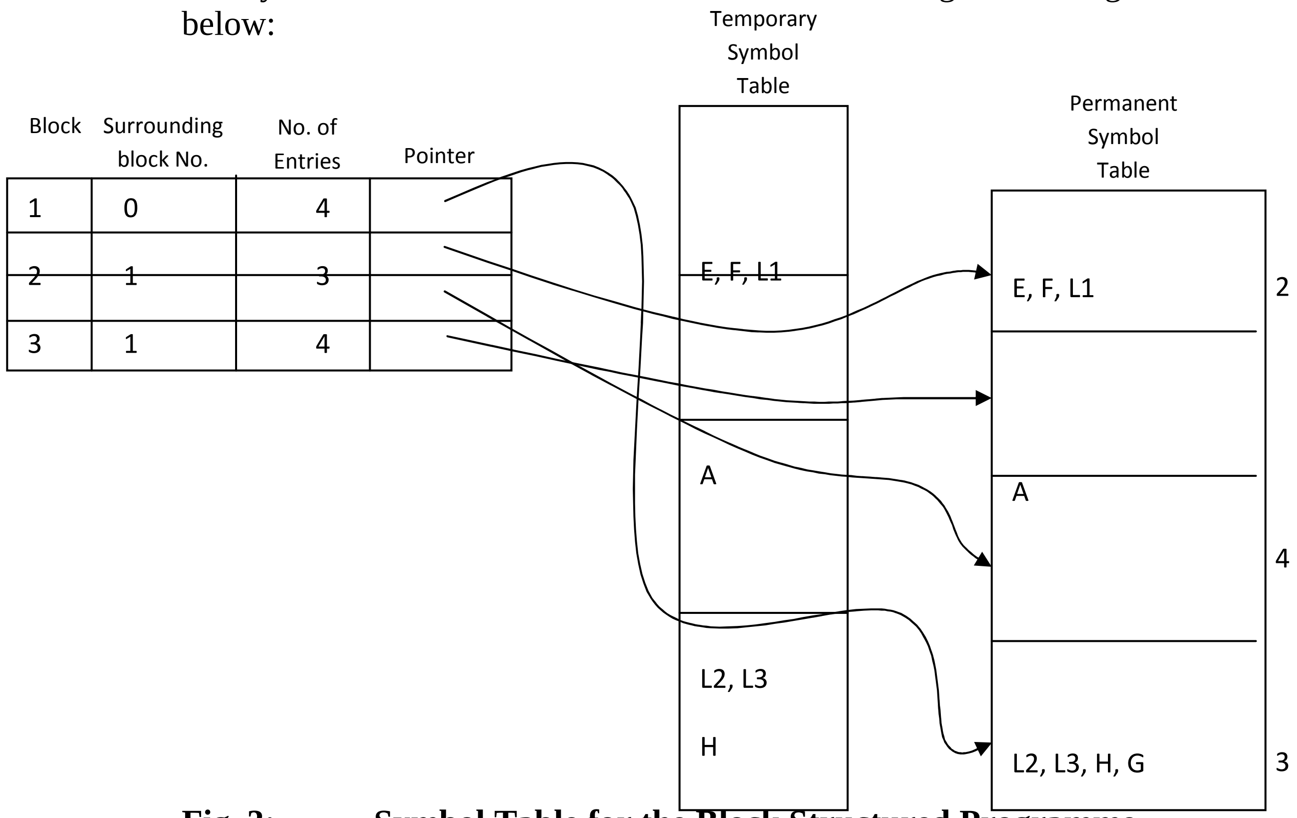


Fig. 3: Symbol Table for the Block Structured Programme

The way to build the symbol table is to construct a block list consisting of block numbers, surrounding block numbers, Number of entries and Pointers.

Symbols are entered into the table in the order in which their blocks closed. A block list containing entries, surrounding block numbers, number of entries and pointers is used to implement this (symbol table) on a stack.

The rule for finding the current declaration corresponding to the use of an identifier/symbol is first to look in the current block (the one in which the identifier is used), then the surrounding block and so on until a declaration of that identifier is found.

Temporary locations may be reserved for identifiers declared whose blocks have not been closed and then transferred into the main symbol table when their blocks have closed. We number the blocks in the manner they open.

3.4 Parameter Passing

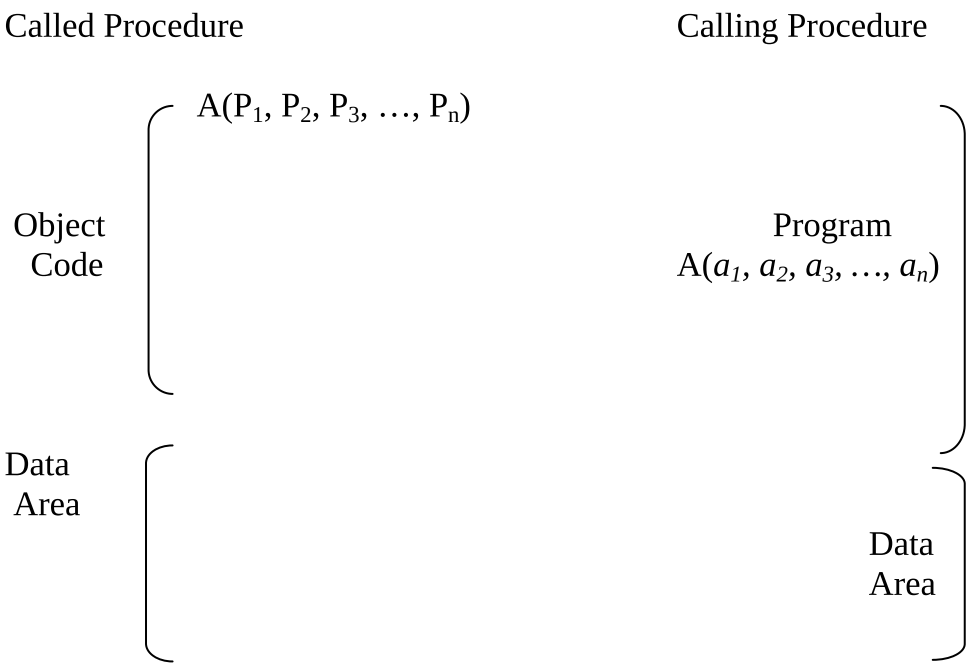


Fig. 4: Actual-Formal Parameter Correspondence

A(P_1, P_2, P_3, …, P_n) are called formal parameters
A(a_1, a_2, a_3, …, a_n) are called actual parameters

3.4.1 Call by Reference

It is the easiest to implement. At run time prior to the call in the calling procedure, the actual parameter is processed. If it is not a variable or constant, it is evaluated and stored into a temporary location. The address of the variable or constant or temporary variable is passed to the called procedure. The called procedure uses this address to refer to the parameter.

3.4.2 Call by Value

The called procedure in this type of correspondence has a location allocated in its data area for a value of the type of the formal parameter. The calling procedure calculates and passes the address containing the value of the actual parameter. Before execution the called procedures takes the value from the address and puts it in its own location and uses this location for further reference to the parameter. There is no way the called procedure can change the value of the actual parameter. i.e. Here the calling procedure passes the value of a_1, a_2, a_3, …, a_n to the called procedure and when the called procedure finish running the result of P_1, P_2, P_3, …, P_n will not be stored in a_1, a_2, a_3, …, a_n in the calling procedure.

3.4.3 Call by Result

This is similar to the call by value but no initialisation. But when the called procedures finishes, the final value of the parameter is stored at the address of the actual parameter. i.e. Initially, the calling procedure does not pass anything to the called procedure, but when the called procedure finishes, the results P_1, P_2, P_3, …, P_n will be stored in a_1, a_2, a_3, …, a_n.

3.4.4 Call by Value Result

A parameter can be stored as both value and result. In this case, the local location of the formal parameters is initialised to the value contained in the address of the actual parameter and the called procedure returns the result back to the actual parameter.

3.4.5 Call by Name

This call implementation requires a textual substitution of the formal parameter name by the actual parameter. It is implemented by using a routine called THUNK to evaluate the actual parameter at each reference and returns its address. i.e. Here, the called procedure will be recompiled substituting a_1, a_2, a_3, …, a_n for the parameters P_1, P_2, P_3, …, P_n (the name without the address or the value).

SELF- ASSESSMENT EXERCISE

i. Compare rehashing and chaining method of collision resolution.
ii. Describe each of the following parameter passing method:
   a. Call by value
   b. Call by reference
   c. Call by name
   d. Call by value result.

4.0 CONCLUSION

In this unit, you have been taken through symbol table and how it can be constructed for a block structured language. As you have learnt in this unit, symbol table is a very important feature of all compilers and it is made use of at all phases of compilation. In the next unit you will learn about how codes are generated.

5.0 SUMMARY

In this unit, you learnt that:

- semantic analysis is roughly the equivalent of checking that some ordinary text written in a natural language actually means something
- the purpose of semantic analysis is to check that we have a meaningful sequence of tokens
- a symbol table contains the environmental information concerning the attributes of various programming language constructs
- symbol tables may be implemented using linear lists, hash-tables and various sorts of tree structures
- in block-structured implementation, symbols are entered into the table in the order in which their blocks closed
- parameters can be passed by value, by result, by reference, etc.

6.0 TUTOR-MARKED ASSIGNMENT

i. Suppose we have a hash table with 10 locations and we wish to enter “names” which are integers, using the hash function h(i) = i mod 10, that is, the remainder when i is divided by 10. Show the links created in the hash and storage tables if the first 2, 3, 5, ..., 29 are entered in that order. As you hash more primes into the table, do you expect them to distribute randomly among the ten list> why or why not?

ii. Compare the performance of linear list structured symbol table and tree structured symbol table.

7.0 REFERENCES/FURTHER READING

Alfred, V. Aho, Monica Lam, Ravi Sethi, & Jeffrey D. Ullman (2007). *Compilers: Principles, Techniques, and Tools*. Second Edition. Pearson Addison-Wesley.

Andrew, W. Appel (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Keith, D. Cooper & Linda, Torczon (2004). *Engineering a Compiler*. Morgan Kaufmann.

Steven, S. Muchnick (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Michael, L. Scott (2009). *Programming Language Pragmatics*. Third edition. Morgan Kaufman.

Robert, W. Sebesta (2010). *Concepts of Programming Languages*. Ninth edition. Addison-Wesley.

UNIT 3 INTERMEDIATE CODE GENERATION

CONTENTS

1.0 INTRODUCTION

Having learnt about symbol tables in the previous unit, you will be taken further into code generation in this unit. Code generation phase of the compiler starts with intermediate code generation. In this unit you will learn specifically about three-address code which is the most popular type of intermediate language representation.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define intermediate representation
- define three-address code
- state with examples types of three-address code
- describe stack-based implementation of intermediate representation
- convert representations like three-address code to the stack-based code
- convert representations stack-based code to three-address code
- generate intermediate code for Declarations, Expressions, Commands, and Procedures.

3.0 MAIN CONTENT

3.1 Machine Independent Languages

The front part of the compiler usually translates the source code programme into *an intermediate language representation*, which after that is converted into a machine code for the concrete target computer.

Using an intermediate language representation entails two properties:

- the compiler becomes *retargetable* and can be ported easily, with a little effort to another computer
- the compiler becomes *optimisable* and can be considerably improved.

3.1.1 Intermediate Code Generator

The data structure passed between the analysis and synthesis phases is called the **intermediate representation** (**IR**) of the programme. A well designed intermediate representation facilitates the independence of the analysis and syntheses (front- and back-end) phases. Intermediate representations may be:

- assembly language like or
- be an abstract syntax tree.

In Intermediate code generation we use syntax directed methods to translate the source programme into an intermediate form programming language constructs such as declarations, assignments and flow-of-control statements.

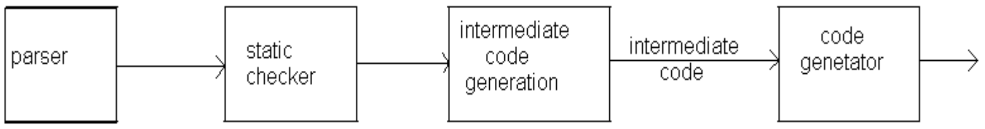


Fig. 1: Intermediate Code Generator

3.2 Intermediate Languages/ Representations

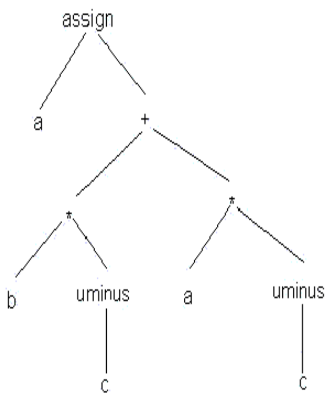
There are three types of intermediate representation:

a. Syntax Trees
b. Postfix notation
c. Three Address Code

Semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees of for generating postfix notation.

3.2.1 Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source programme. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified. A syntax tree for the assignment statement a:=b*-c+b*-c appear in figure 2 below.



.

Fig. 2: A Syntax Tree for the Assignment Statement a:=b*-c+b*-c

Postfix notation is a linearised representation of a syntax tree; it is a list of the nodes of the in which a node appears immediately after its children. The postfix notation for the syntax tree in figure 1 is:

a b c uminus + b c uminus * + assign

The edges in a syntax tree do not appear explicitly in postfix notation. They can be recovered in the order in which the nodes appear and the no. of operands that the operator at a node expects. The recovery of edges is similar to the evaluation, using a staff, of an expression in postfix notation.

Syntax tree for assignment statements are produced by the syntax directed definition in Table 1 below

Table 1: Syntax -Directed Definition

Production	**Semantic Rule**
S → **id** := E	S.nptr := mknode('assign', mkleaf(**id**, **id.**place), E.nptr)
E → E1 + E2	E.nptr := mknode('+', E1.nptr ,E2.nptr)
E → E1 * E2	E.nptr := mknode('* ', E1.nptr ,E2.nptr)
E → - E1	E.nptr := mkunode('uminus', E1.nptr)
E → (E1)	E.nptr := E1.nptr
E → **id**	E.nptr := mkleaf(**id**, **id.**place)

This same syntax-directed definition will produce the dag if the functions mkunode(op, child) and mknode(op, left, right) return a pointer to an existing node whenever possible, instead of constructing new nodes. The token id has an attribute place that points to the symbol-table entry for the identifier id.name, representing the lexeme associated with that occurrence of id. lf the lexical analyser holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme. Two representations of the syntax tree in Figure 1 appear in Figure 3. Each node is represented as a record with a field for its operator and additional fields for pointers to its children. In Figure 3(b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position IO.

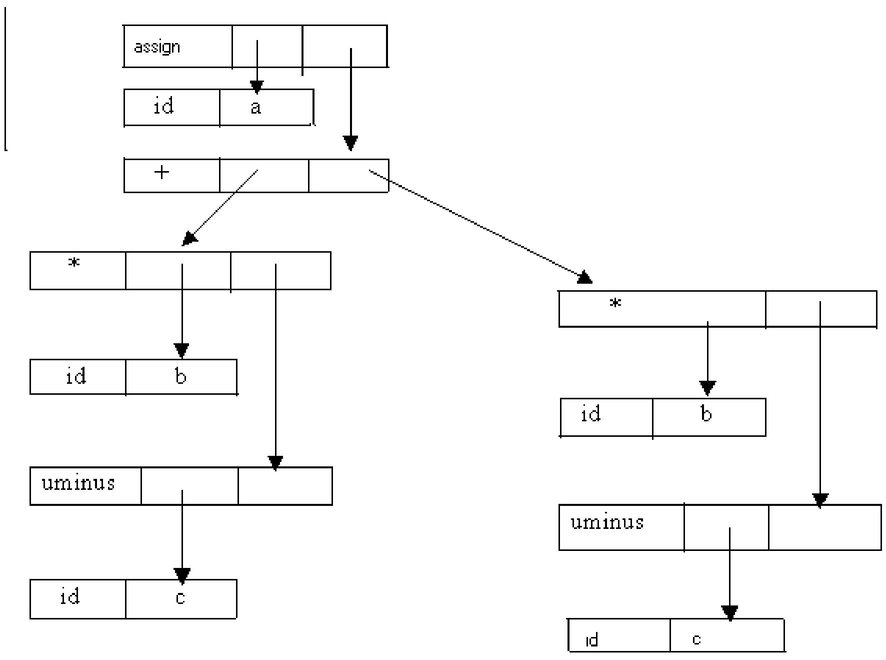


Fig. 3(a): Two Representations of the Syntax Tree

0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	……		

Fig. 3(b): Two Representations of the Syntax Tree

The form of the internal representation among different compilers varies widely. If the back end is called as a subroutine by the front end then the intermediate representation is likely to be some form of annotated parse tree, possibly with supplementary tables. If the back end operates as a separate programme then the intermediate representation is likely to be some low-level pseudo assembly language or some register transfer language (it could be just numbers, but debugging is easier if it is human-readable).

A popular intermediate language is the so called *three-address code*.

3.2.2 Three-Address Code

Three-address code is a sequence of statements of the general form:

x := y *op* z

where: x,y,z are names, constants or compiler-generated temporaries, *op* stands for an operator, such as a fixed or floating-point arithmetic operator, or a logical operator on a Boolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement.

Example 1:

An expression of the kind x + y * z will be translated into the sequence
t_1 := y * z
t_2 := x + t_1

where t_1 and t_2 are compiler-generated temporary names.

This unravelling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimisation. The use of names for the intermediate values computed by a programme allow- three-address code to be easily rearranged – unlike postfix notation. Three-address code is a linearised representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

The syntax tree and dag in Figure 1 are represented by the three-address code sequences in Figure 4. Variable names can appear directly in three-address statements, so Figure 4(a) has no statements corresponding to the leaves in Figure 3.

t_1 := -c
t_2 := b * t_1
t_3 := -c
t_4 := b * t3
t_5 := t_2 + t_4
a := t_5

Fig. 4(a): Three-address Code for Syntax Tree

t_1 := -c
t_2 := b * t_1
t_5 := t_2 + t_2
a := t_5

Fig. 4(b): Three-address Code for DAG

The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result. In the implementations of three-address code given later in this section, a programmer-defined name is replaced by a pointer to a symbol-table entry for that name.

3.2.2.1 Types of Three-Address Statements

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding inter- mediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using ”back patching,”. Here are the most common three-address statements and they are the ones used in the remainder of this course material:

1) *Assignment statements* of the form: x := y *op* z
   where *op* is a binary arithmetic or logical operation;
2) *Assignment statements* of the form: x := *op* y
   where *op* is a unary operation. Essential unary operations include unary minus, logical negation, and shift operators;
3) *Copy statements* of the form: x := y
   where the value of y is assigned to x;
4) The *unconditional jump* goto L. The three-address statement with label L is the next to be executed;
5) *Conditional jumps* such as: if x *relop* y goto L.
   This instruction applies a relational operator (<, =, >, <=, etc.) to x and y, and executes the statement with label L next if x stands in relation*relop* to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.;
6) *Procedure calls*: call p,*n* and *returned values* from functions:
   return y.

   Their typical use is the following:

   param x_1
   param x_2
   ...
   param x_n

call p,*n*

generated as part of a call of the function: p(x_1, x_2,... x_n). The integer n indicating the number of actual parameters in"call p, n" is not redundant because calls can be nested;

7) *Indexed assignments* of the form: x := y[i] and x[i] := y. The first one sets x to the value in the location i memory units beyond y. The statement x[i] := y sets the contents of the location i units beyond x to the value of y. In both these instructions, x, y, and i refer to data objects;
8) *Address and pointer assignments*: x := &y and x := *y. The first of these sets the value of x to be the location of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an I-value such as A[i, j], and x is a pointer name or temporary. That is, the r-value of x is the l-value (location) of some object!. In the statement x: = ~y, presumably y is a pointer or a temporary whose r- value is a location. The r-value of x is made equal to the contents of that location. Finally, +x: = y sets the r-value of the object pointed to by x to the r-value of y.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must clearly be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restricted instruction set may force the front end to generate long sequences of statements for some source, language operations. The optimiser and code generator may then have to work harder if good code is to be generated.

3.2.3 Stack-Based Representation

In this unit, we discuss the stack-based representation of intermediate code. It has a number of advantages, some of which are:

- an interpreter for the stack-based language tends to be more compact and straightforward
- a syntax of the language tends to be simple.
- But the representation also has the following disadvantages, which make it unsuitable for manipulating and improving code:
- it is not trivial to change the order of instructions
- little research has been done to the stack-based code.

Complications with the stack-based code arise often with control flows.

3.3 Conversion algorithms

It is usually trivial to convert representations like three-address code to the stack-based code, so the case is left as an exercise. It is the inverse of this that is a challenging problem.

The main task behind the algorithm converting the stack-based code is to identify dependencies among operations. And it is conditional and unconditional jumps that make hard to figure these dependencies. So the code without them can be transformed into the three-address code in a straightforward way, as follows:

- push 2
- push 3
- push 4
- add
- mul

We can see each stack position has a corresponding temporary variable. Put in another way, *store* and *load* are done only by *push* and *pop*, respectively, and a temporary variable that can be accessed at a time is limited to only the top as opposed to a usual case in which variables are specified freely.

```
s0 = 2
s1 = 3
s2 = 4
s1 = s1 + s2
s0 = s0 * s1
```

When a variable is typed, it may be beneficial to adopt SSA form. This dispenses with the need to analyse what type each variable holds at a moment, which, as illustrated below, can be quite tricky. The adaptation can be done after the conversion or simultaneously as the code is being converted.

Now, suppose the execution may not go from top to bottom. In that case, we basically have to analyse the control flow before translating the code. More specifically, we calculate how each instruction contributes to the depth of the stack. For example,

```
...
goto A // unconditionally jump to label A
...
A:    // a label
add  // push the sum of two values popped from the stack.
...
```

As we can see, at label A, the status of the stack depends on the operation before instruction "goto A".

One conservative approach is to annotate the byte-code before converting it. The basic idea is that when we interpret the code, we know both where we are and how tall the stack is. So by pretending as if we were evaluating the code, we can calculate the height of the stack for each position. An algorithm for this would be like (Note that in actual writing you have to arrange the code so that it will terminate):

```
procedure eval(start, depth)
{
  for i from start to code.length
  {
    depth_at[i] = depth
    case code[i]
    {
      'push': depth = depth + 1
      'pop':  depth = depth - 1
      'goto': i = code[i].target
      'if_so_goto': eval(code[i].target, depth)
      ...
    }
  }
}
eval(0, 0) // start the calculation
```

Coding the above solution may be tedious in practice, especially when a number of instructions in the language is large. Java byte code is a notable example of this. So a radical alternative below is to convert the stack-based code not in the usual sequential way but per basic block (i.e., a block that has no jumps). To see this, consider the following:

```
0 (A): push 10
1 (A): push 13
2 (A): less_than // pop < pop
3 (A): if_not_goto 6
4 (B): push '10 < 13'
5 (B): goto 7
6 (C): push 'it is not 10 < 13; your computer is broken!'
7 (C): print
```

In the above we can identify three basic blocks: the first (A) from line 0 to 3, the second (B) from line 4 to 5 and the third (C) from line6 to 7. We first compile A, then we know the height of the stack with which either B or C begins. After each block is compiled, we output blocks in the order they appear in the source code.

If you are an astute reader/learner, you would notice that throughout this section we are assuming the depth of the stack is fixed at each instruction position and thus can be determined at compiler time. If the assumption does not hold, then we have to have some kind of stack at runtime.

SELF-ASSESSMENT EXERCISE

i. Write the stack-based code for each of following high-level expressions:
   10 * (20 + 30);
   If a < b then -a else -b;
   case a % 3 { 0: x; 1: y; 2: z; }
ii. Write a piece of stack-based code so that the depth of the stack may vary after the piece, depending on an execution path.

3.4 Intermediate Code Generation for Declarations, Expressions, Commands, and Procedures

3.4.1 Intermediate Code Generation for Declarations

P	→ D	{ *offset* := 0 }
D	→ D ; D	
D	→ **id** : T	{ *enter*(**id**.*name*, T.*type*, *offset*) *offset* := *offset* + T.*width* }
T	→ **integer**	{ T.*type* := *integer* T.*width* := 4 }
T	→ **real**	{ T.*type* := *real* T.*width* := 8 }
T	→ **array**[**num**] **of** T_1	{ T.*type* := *array*(0..**num**.*val*, T_1.*type*) T.*width* := **num**.*val*× T_1.*width*}

Translation scheme for processing declarations in nested functions

P→ M D	{*addwidth*(*top*(*tblptr*), *top*(*offset*))**;** *pop*(*tblptr*)**;** *pop*(*offset*)**;**}
M → ∈	{*t* := *mktable*(*nil*)**)** **;** *push*(*t, tblptr*)**;** *push*(*0, offset*)**;**}
D → **func id** ;	{*t* := *top*(*tblptr*) **;**
N D_1; S	*addwidth*(*t*, *top*(*offset*))**;**

```
                                  pop(tblptr);
                                  pop(offset);
                                  enterfunc(top(tblptr), id.name, t);}
D → id : T           { enter(top(tblptr), id.name, T.type, top(offset));
                                  top(offset) := top(offset) + T.width }
N → ∈                     { t := mktable(top(tblptr)) ;
                                  push(t, tblptr);
                                  push(0, offset); }
```

3.4.2 Intermediate Code Generation for Assignment Statements

```
S      → id := E     { p := lookup(id.name) ;
                         if p ≠ nil then emit(p ':=' E.place )
                          else error }
E      → E1 + E2      { E.place := newtemp;
                        emit(E.place ':=' E1.place '+' E2.place)}
E      → E1 * E2      { E.place := newtemp;
                         emit(E.place ':=' E1.place '*' E2.place)}
E      → - E1         { E.place := newtemp;
                         emit(E.place ':='-' E1.place)}
E      → (E1)         { E.place := E1.place}
S      → id          { p := lookup(id.name) ;
                            if p ≠ nil then E.place := p
                            else error }
```

3.4.3 Intermediate Code Generation for Boolean Expressions

```
E      → E1 or E2          { E.place := newtemp;
                            emit(E.place ':=' E1.place 'or' E2.place)}
E      → E1 and E2         { E.place := newtemp;
                            emit(E.place':='E1.place 'and'E2.place)}
E      → not E1            { E.place := newtemp;
                            emit(E.place ':=' 'not' E1.place)}
E      → (E1)              { E.place := E1.place}
E      → id1 relop id2     { E.place := newtemp;
                 emit('if' id1.place relop id2.place 'goto' nextstat + 3) ;
                        emit(E.place ':=' 0) ;
                        emit('goto' nextstat + 2) ;
                        emit(E.place ':=' '1') }
E      → true          { E.place := newtemp;
                        emit(E.place ':=' '1') }
```

Example 2: a or b and not c

```
$t_1$ := not c
$t_2$ := b and $t_1$
$t_3$ := a or $t_2$
```

Example 3: a < b

```
100:   if (a < b) goto 103
101:   t := 0
102:   goto 104
103:   t := 1
104:
```

3.4.4 Intermediate Code Generation for Commands

```
S    → if E then S1        { E.true := newlabel ;
                             E.false := S.next ;
                             S1.next := S.next ;
                             S.code := E.code||gen(E.true ':')|| S1.code}

S    → if E then S1        { E.true := newlabel ;
                 else S2     E.false := newlabel ;
                             S1.next := S.next ;
                             S2.next := S.next ;
                             S.code := E.code|| gen(E.true ':')||S1.code
                             gen('goto' S.next) || gen(E.false ':') || S2.code}

S    → while E do S1       { S.begin := newlabel ;
                             E.true := newlabel ;
                             E.false := S.next ;
                             S1.next := S.begin ;
                             S.code := gen(S.begin ':') || E.code
                             gen(E.true ':')   || S1.code
gen('goto' S.begin) }
```

Example 4: while a < b do

```
  if c < d then
    x := y + z
  else
    x := y - z

L1:    if (a < b) goto L2
       goto Lnext
L2:    if (c < d) goto L3
```

```
          goto L4
L3:       t1 := y + z
          x := t1
          goto L1
L4:       t2 := y - z
          x := t2
          goto L1
Lnext:
```

3.4.5 Generating Intermediate Code for a Simple Programme

Generate three-address code for the following program fragment:

```
void Ssort(int a[], int N)
{
        int i, j, k, min;

        i = 0;
        while (i < N)
        {
                        min = i; j = i + 1;
                        while (j < N + 1)
                        {
                        if (a[j] < a[min]) min = j;
                        ++j;
                        }
                        k = a[min];
                        a[min] = a[i];
                        a[i] = k;
                        ++i;
                        }
                        }

                        void main()
                        {
                int i, N;
                int a[10];

                N = 10;
                i = 0;
                while (i < N)
                {
                                a[i] = getch();
                                ++i;
                }
Ssort(a, N);
}
```

Three-address Code:

1) t := mktable(nil)
2) s := mktable(nil)
3) enter(s, i, int, 4)
4) enter(s, j, int, 4)
5) enter(s, k, int, 4)
6) enter(s, min, int, 4)
7) i := 0
8) if (i >= N) goto (30)
9) min := i
10) j := i + 1
11) if (j >= N+1) goto (20)
12) t_1 := 4 * j
13) t_2 := a[t_1]
14) t_3 := 4 * min
15) t_4 := a[t_3]
16) if (t_2 >= t_4) goto (20)
17) min := j
18) j := j + 1
19) goto (11)
20) t_5 := 4 * min
21) t_6 := a[t_5]
22) t_7 := 4 * i
23) t_8 := a[t_7]
24) k := t_6
25) a[t_5] := t_8
26) a[t_7] := k
27) i := i + 1
28) goto (8)
29) enterproc(t, s, proc, 4 * 4)
30) m := mktable(nil)
31) enter(m, i, int, 4)
32) enter(m, N, int, 4)
33) enter(m, a, record, 40)
34) N := 10
35) i := 0
36) if (i >= N) goto (41)
37) t_9 := 4 * i
38) a[t_9] := read
39) i := i + 1
40) goto (36)
41) param a
42) param N
43) call s, 2
44) enterproc(t, m, proc, 40+2*4)

4.0 CONCLUSION

In this unit, you have learnt about intermediate code generation in compilers. In the next unit you will learn about code generation.

5.0 SUMMARY

In this unit, you learnt that:

- the front part of the compiler usually translates the source code programme into *an intermediate language representation*, which after that is converted into a machine code for the concrete target computer
- using an intermediate language representation entails two properties: retargetability and optimisability
- a popular intermediate language is the *three-address code*

- certain conversion algorithms can be applied to convert stack-based code of intermediate representation to three-address code and vice versa.

6.0 TUTOR-MARKED ASSIGNMENT

i. Sketch the algorithm for converting three-address code to the stack-based code, assuming no jumps. Hint: view each position in the stack has a corresponding temporary variable.
ii. Write a stack-based code such that the height of the stack at each position cannot be determined at a compiler time.

7.0 REFERENCES/FURTHER READING

Alfred, Aho V. *et al*. (2007). *Compilers: Principles, Techniques, and Tools*. Second Edition. Wesley: Pearson Addison.

Andrew, Appel W. (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Cooper, Keith D. & Torczon, Linda (2004). *Engineering a Compiler*. Morgan Kaufmann.

Muchnick, Steven S. (1997).*Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Scott, Michael L. (2009). *Programming Language Pragmatics*. Third Edition. Morgan Kaufman.

Sebesta Robert W.(2010). *Concepts of Programming Languages*. Ninth Edition. Wesley: Addison

UNIT 4 CODE GENERATION

CONTENTS

1.0 INTRODUCTION

In the previous unit you have learnt about intermediate code generation and the various intermediate languages that can be used for intermediate code generation. In this unit, you will learn about the final phase of a compiler, which is code generation. It takes as input an intermediate representation of the source programme and produces as output an equivalent target programme.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define code generation
- diagrammatically show the position of code generator in a compiler
- explain basic code generation issues such as input, output, memory management, instruction selection, etc.
- describe runtime storage allocation.

3.0 MAIN CONTENT

3.1 Code Generation

The primary objective of the **code generator** is to convert atoms or syntax trees to instructions.
The final phase in our compiler model is the **code generator**. It takes as input an intermediate representation of the source programme and produces as output an equivalent target programme.

The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

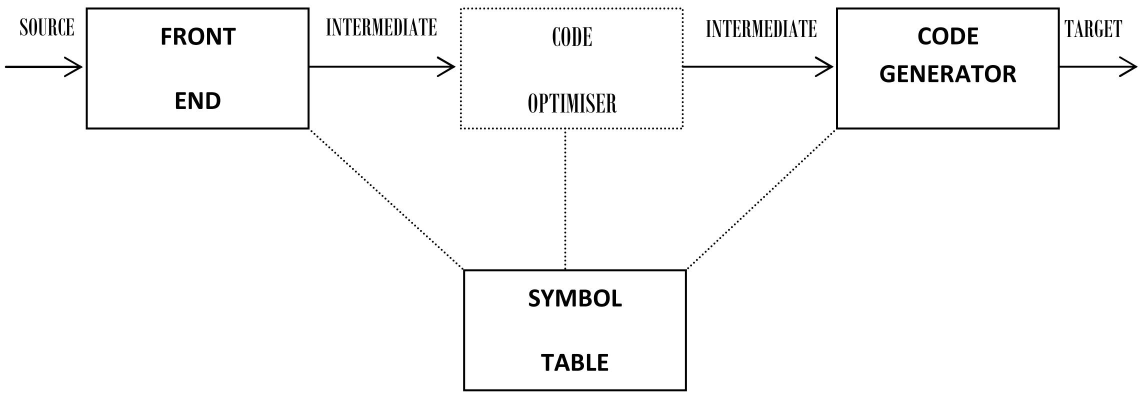


Fig. 1: Position of Code Generator

3.2 Issues in the Design of a Code Generator

While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation, and evaluation order are inherent in almost all code generation problems.

3.2.1 Input to the Code Generator

The input to the code generator consists of the intermediate representation of the source programme produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation.

There are several choices for the intermediate language, including: linear representations such as postfix notation, three address representations such as quadruples, virtual machine representations such as syntax trees and dags.

We assume that prior to code generation the front end has scanned, parsed, and translated the source programme into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.). We also assume that the necessary type checking has take place, so type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.

3.2.2 Target Programmes

The output of the code generator is the target programme. The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

Producing an absolute machine language programme as output has the advantage that it can be placed in a location in memory and immediately executed. A small programme can be compiled and executed quickly. A number of “student-job” compilers, such as WATFIV and PL/C, produce absolute code.

Producing a relocatable machine language programme as output allows subprogrammes to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. 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 programmes from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled programme segments.

Producing an assembly language programme as output makes the process of code generation somewhat easier .We can generate symbolic instructions and use the macro facilities of the assembler to help generate code .The price paid is the assembly step after code generation. Due to the fact that producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must uses several passes.

3.2.3 Memory Management

Mapping names in the source programme to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the “back patching”. Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as j: *goto i* is encountered, and i is less than j, the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple i. If, however, the jump is forward, so i exceeds j, we must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. Then we process quadruple i, we fill in the proper machine location for all instructions that are forward jumps to i.

3.2.4 Instruction Selection

The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target programme, instruction selection is straightforward. For each type of three- address statement we can design a code skeleton that outlines the target code to be generated for that construct.

For example, every three address statement of the form x: = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
MOV y, R0   /* load y into register R0  */
ADD z, R0   /* add z to R0 */
MOV R0, x   /* store R0 into x */
```

Unfortunately, this kind of statement-by-statement code generation often produces poor code. For example, the sequence of statements:

```
a := b + c
d := a + e
```

would be translated into

```
MOV   b, R0
ADD    c, R0
MOV   R0, a
MOV   a, R0
ADD    e, R0
MOV   R0, d
```

Here the fourth statement is redundant, and so is the third if ‘a’ is not subsequently used.

The quality of the generated code is determined by its speed and size.

A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may lead to correct, but unacceptably inefficient target code. For example if the target machine has 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 by a more obvious sequence that loads a into a register, add one to the register, and then stores the result back into a.

```
MOV   a, R0
ADD    #1,R0
MOV   R0, a
```

Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain. Deciding which machine code sequence is best for a given three address construct may also require knowledge about the context in which that construct appears.

3.2.4 Register Allocation

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilisation of register is particularly important in generating good code. The use of registers is often subdivided into two sub-problems:

1. During **register allocation,** we select the set of variables that will reside in registers at a point in the programme.
2. During a subsequent **register assignment** phase, we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require **register pairs** (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form:

```
M    x, y
```

where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form

```
D    x, y
```

where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

Now consider the two three address code sequences (a) and (b) in figure 2 below in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in (c).

Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which ‘a’ is to be loaded depends on what will ultimately happen to e.

```
t  :=  a + b                          t := a + b
t  :=  t * c                          t  :=  t + c
t  :=  t / d                          t  := t / d
   (a)                                    (b)
```

Fig. 2 Two three address code sequences

```
 L    R1, a                    L       R0, a
 A    R1, b                    A       R0, b
 M    R0, c                    A       R0, c
 D    R0, d                    SRDA    R0, 32
 ST   R1, t                    D       R0, d
                               ST      R1, t
 (a)                            (b)
```

Fig. 3: Optimal machine code sequence

3.2.5 Choice of Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

3.3 Approaches to Code Generation

The most important criterion for a code generator is that it produces correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

3.4 Run-Time Storage Management

The semantics of procedures in a language determines how names are bound to storage during allocation. Information needed during an execution of a procedure is kept in a block of storage called an activation record; storage for names local to the procedure also appears in the activation record.

An activation record for a procedure has fields to hold parameters, results, machine-status information, local data, temporaries and the like. Since run-time allocation and de-allocation of activation records occurs as part of the procedure call and return sequences, we focus on the following three-address statements:

1. call
2. return
3. halt

4. action, a placeholder for other statements

For example, the three-address code for procedures c and p in fig. 4 contains just these kinds of statements. The size and layout of activation records are communicated to the code generator via the information about names that is in the symbol table. For clarity, we show the layout in Fig. 4 rather than the form of the symbol-table entries.

We assume that run-time memory is divided into areas for code, static data, and a stack.

three-address code activation record for c activation record for p
(64 bytes)
(64 bytes)

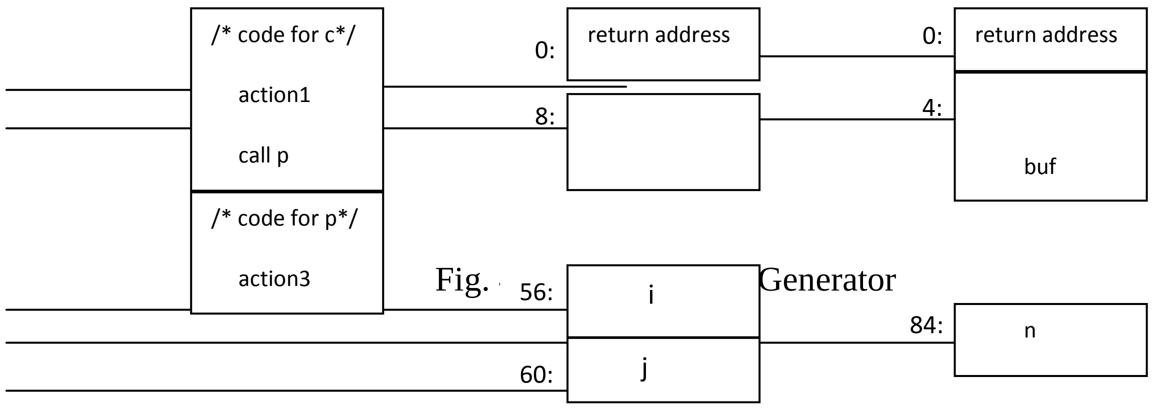


Fig. Generator

3.4.1 Static Allocation

Consider the code needed to implement static allocation. A call statement in the intermediate code is implemented by a sequence of two target-machine instructions. A MOV instruction saves the return address, and a GOTO instruction transfers control to the target code for the called procedure:

MOV *#here* +20, *callee.static_area*
GOTO *callee.code_area*

The attributes *callee.statatic_area* and *callee.code_area* are constants referring to the address of the activation record and the first instruction for the called procedure, respectively. The source *#here*+20 in the MOV instruction is the literal return address; it is the address of instruction following the GOTO instruction.

The code for a procedure ends with a return to the calling procedure ends with a return to the calling procedure, except the first procedure has no caller, so its final instruction is HALT, which presumably returns control to the operating system. A return from procedure callee is implemented by

```
 GOTO    *callee.static_area
```

this transfers control to the address saved at the beginning of the activation record.

Example 1:

The code in Figure 5 is constructed from the procedures c and p in Figure 4. We use the pseudo-instruction ACTION to implement the statement action, which represents three-address code that is not relevant for this discussion. We arbitrarily start the code for these procedures at addresses 100 and 200, respectively, and assume that each ACTION instruction takes 20 bytes. The activation records for the procedures are statically allocated starting at location 300 and 364, respectively.

```
                                         /*code for c*/
 100:   ACTION1
 120:   MOV   #140,364            /*save return address 140 */
 132:    GOTO 200                /* call p */
 140:   ACTION2
 160:    HALT
  ……
                                        /*code for p*/
 200:    ACTION3
 220:    GOTO *364         /*return to address saved in location 364*/
 ……
                                /*300-363 hold activation record for c*/
300:                                /*return address*/
304:                                /*local data for c*/
 ……                             /*364-451 hold activation record for p*/
364:                                /*return address*/
368:                                /*local data for p*/
```

Fig. 5: Target Code for Input in Figure 4

The instructions starting at address 100 implement the statements

action1; call p; action2; halt

of the first procedure c. Execution therefore starts with the instruction ACTION1 at address 100. The MOV instruction at address 120 saves the return address 140 in the machine-status field, which is the first word in the activation record of p. The GOTO instruction at address 132 transfers control to the first instruction is the target code of the called procedure.

Since 140 was saved at address 364 by the call sequence above, *364 represents 140 when the GOTO statement at address 220 is executed. Control therefore returns to address 140 and execution of procedure c resumes.

3.4.2 Stack Allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. The position of the record for an activation of a procedure is not known until run time. In stack allocation, this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register. The indexed address mode of our target machine is convenient for this purpose.

Relative addresses in an activation record can be taken as offsets from any known position in the activation record. For convenience, we shall use positive offsets by maintaining in a register SP a pointer to the beginning of the activation record on top of the stack. When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure. After control returns to the caller, it decrements SP, thereby de-allocating the activation record of the called procedure.

The code for the first procedure initialises the stack by setting SP to the start of the stack area in memory.

```
MOV   #stackstart, SP         /*initialise the stack*/
code for the first procedure
HALT                          /*terminate execution*/
```

A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD   #caller.recordsize, SP
MOV   #here+16, SP               /* save return address*/
GOTO   callee.code_area
```

The attribute *caller.recordsize* represents the size of an activation record, so the ADD instruction leaves SP pointing to the beginning of the next activation record. The source #*here*+16 in the MOV instruction is the address of the instruction following the GOTO; it is saved in the address pointed to by SP.

The return sequence consists of two parts. The called procedure transfers control to the return address using

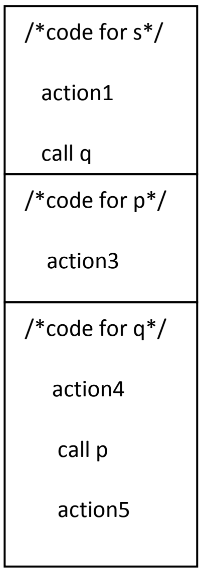
```
GOTO   *0(SP)          /*return to caller*/
```

The reason for using *0(SP) in the GOTO instruction is that we need two levels of indirection: 0(SP) is the address of the first word in the activation record and *0(SP) is the return address saved there.

The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. That is, after the subtraction SP points to the beginning of the activation record of the caller:

SUB #*caller.recordsize*, SP

Example 2:
The programme in figure 6 is a condensation of the three-address code for the Pascal program for reading and sorting integers. Procedure q is recursive, so more than one activation of q can be alive at the same time.



```
/*code for s*/
  action1
  call q
/*code for p*/
  action3
/*code for q*/
  action4
  call p
  action5
```

Fig. 6: Three-Address Code to Illustrate Stack Allocation

Suppose that the sizes of the activation records for procedures s, p, and q have been determined at compile time to be *ssize, psize,* and *qsize*, respectively. The first word in each activation record will hold a return address. We arbitrarily assume that the code for these procedures starts at addresses 100,200 and 300 respectively, and that the stack starts at 600. The target code for the programme in figure 6 is as follows:

```
                              /*code for s*/
100:  MOV   #600, SP    /*initialize the stack*/
108:  ACTION1
128:  ADD  #ssize, SP     /*call sequence begins*/
136:  MOV #152, *SP     /*push return address*/
144:  GOTO  300          /*call q*/
152:  SUB #ssize, SP      /*restore SP*/
160:  ACTION2
180:  HALT
      .........
                               /*code for p*/
200:  ACTION3
220:  GOTO *0(SP)        /*return*/
      .........
                              /*code for q*/
300:  ACTION4           /*conditional jump to  456*/
320:  ADD #qsize, SP
328:  MOV #344, *SP     /*push return address*/
336:  GOTO 200          /*call p*/
344:  SUB #qsize, SP
352:  ACTION5
372:  ADD #qsize, SP
380:  MOV #396, *SP     /*push return address*/
388:  GOTO 300            /*call q*/
396:  SUB #qsize, SP
404:  ACTION6
424:  ADD #qsize, SP
432:  MOV #448, *SP      /*push return address*/
440:  GOTO 300             /*call q*/
448:  SUB #qsize, SP
456:  GOTO *0(SP)         /*return*/
       .........
 600:                          /*stack starts here*/
```

We assume that ACTION4 contains a conditional jump to the address 456of the return sequence from q; otherwise, the recursive procedure q is condemned to call itself forever. In an example below, we consider an execution of the programme in which the first call of q does not return immediately, but all subsequent calls do.

If *ssize*, *psize*, and *qsize* are 20,40, and 60, respectively, then SP is initialized to 600, the starting address of the stack, by the first instruction at address 100. SP holds 620 just before control transfers from s to q, because *ssize* is 20. Subsequently, when q calls p, the instruction at address 320 increments SP to 680, where the activation record for p begins; SP reverts to 620 after control returns to q. If the next two recursive calls of q return immediately, the maximum value of SP during this execution is 680. However, the last stack location used is 739, since the activation record for q starting at location 680 extends for 60 bytes.

3.4.3 Run-Time Addresses for Names

The storage allocation strategy and the layout of local data in an activation record for a procedure determine how the storage for names is accessed.

If we assume that a name in a three-address statement is really a pointer to a symbol-table entry for the name; it makes the compiler more portable, since the front end need not be changed even if the compiler is moved to a different machine where a different run-time organisation is needed. On the other hand, generating the specific sequence of access steps while generating intermediate code can be of significant advantage in an optimising compiler, since it lets the optimiser take advantage of details it would not even see in the simple three-address statement.

In either case, names must eventually be replaced by code to access storage locations. We thus consider some elaborations of the simple three-address statement x: = 0. After the declarations in a procedure are processed, suppose the symbol-table entry for x contains a relative address 12 for x. First consider the case in which x is in a statically allocated area beginning at address *static.* Then the actual run-time address for x is *static*+12. Although, the compiler can eventually determine the value of *static*+12 at compile time, the position of the static area may not be known when intermediate code to access the name is generated. In that case, it makes sense to generate three-address code to “compute” *static*+12, with the understanding that this computation will be carried out during the code-generation phase, or possibly by the loader, before the programme runs. The assignment x := 0 then translates into

```
static [12] := 0
```

If the static area starts at address 100, the target code for this statement is

```
MOV #0, 112
```

On the other hand, suppose our language is one like Pascal and that a display is used to access non-local names. Suppose also that the display is kept in registers, and that x is local to an active procedure whose display pointer is in register R_3. Then we may translate the copy x := 0 into the three-address statements

t_1:= 12+R_3
*t_1:= 0

in which t1 contains the address of x. This sequence can be implemented by the single machine instruction

MOV #0, 12 (R_3)

The value in R_3 cannot be determined at compile time.

4.0 CONCLUSION

In this unit, you have been taken through the basic issues in code generation. In the next unit, which is the concluding unit of this course you will be learning about code optimisation.

5.0 SUMMARY

In this unit, you learnt that:

- the primary objective of the **code generator,** the final phase of the compiler, is to convert atoms or syntax trees to instructions
- code generator takes as input an intermediate representation of the source programme and produces as output an equivalent target programme
- producing an assembly language programme as output makes the process of code generation somewhat easier
- producing a relocatable machine language programme as output allows subprograms to be compiled separately
- mapping names in the source programme to addresses of data objects in run time memory is done cooperatively by the front end and the code generator
- information needed during an execution of a procedure is kept in a block of storage called an activation record
- static allocation can become stack allocation by using relative addresses for storage in activation records
- relative addresses in an activation record can be taken as offsets from any known position in the activation record.

6.0 TUTOR-MARKED ASSIGNMENT

i. For the following prograe in Simple, generate stack code.

A programme in Simple

```
let
  integer n,x,n.
in
  read n;
  if n < 10 then x := 1;  else skip; fi;
   while  n  <  10  do   x  :=  5*x;  n  :=  n+1;
end;
  skip;
  write n;
  write x;
end
```

ii. Describe the various approaches to code generation.

7.0 REFERENCES/FURTHER READING

Alfred, V. Aho *et al.* (2007). *Compilers: Principles, Techniques, and Tools*. Second Edition. Wesley: Pearson Addison.

Andrew, W. Appel (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Cooper, Keith D. & Torczon, Linda (2004). *Engineering a Compiler*. Morgan Kaufmann.

Muchnick, Steven S. (1997).*Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Scott, Michael L. (2009). *Programming Language Pragmatics*. Third edition, Morgan Kaufman.

Sebesta,Robert W. (2010). *Concepts of Programming Languages*. Ninth edition. Wesley: Pearson Addison.

UNIT 5 CODE OPTIMISATION

CONTENTS

1.0 INTRODUCTION

In this concluding unit of the course, you will be learning about code optimisation. Code optimisation is code transformation techniques that are applied to the intermediate codes to make faster and better in terms of performance and memory management. Although, the optimisation phase is an optional phase in compilers it makes for better and efficient code generation when it is present.

Optimisation is a very rich and complex topic, so this unit will only attempt to introduce the basics.

Now let us go through your study objectives for this unit.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- define code optimisation
- state criteria for code improving transformation
- list categories of optimisation
- state properties of optimising compilers
- list and describe common optimisation algorithms.

3.0 MAIN CONTENT

3.1 Code Optimisation

On modern computers, a compiler can be considered to have satisfactory performance if it translates a moderate size source programme (say about 1000 lines) in a matter of seconds. The way to get a compiler with satisfactory performance is more or less the same way you would get any programme performing well.

Design using good algorithms.

- Ensure your data structures match the algorithms.
- Structure using modules with clean simple interfaces.
- Implement using clear straightforward code.
- When there is an overall performance problem
- Measure the actual performance in reasonable detail.
- Identify the troublesome areas.
- Redesign and re-implement these problem areas.

In this unit, we will consider various algorithms and data structures and discuss their likely impact on performance.

Note that actual measurement is crucial, since the problems are often not where you guess they might be. For your initial implementation you may well have selected simple algorithms which are known to perform poorly in order to get something working quickly. Nevertheless, you should still measure performance in detail, since there may be some other source of (at least some of) your problems.

If you are very lucky, your implementation language might have some optional facility for selectively measuring CPU time. Take care to only activate such a feature for a few crucial routines; the timing overhead could easily exceed the execution time for small routines and distort the result.

More commonly you will be unlucky and will have to explicitly add timing code to selected routines; make sure you can easily disable it and enable it as required. Typically you will have to insert calls to some CPU timing routine at the beginning and end of a routine, and then subtract the two values to get the time for that routine, which will include the time for any routines called by it.

Various measurements on the performance of actual compilers have been reported over the years. Specific areas which have been known to cause problems include:

- multiple routine calls during lexical analysis for each and every source character
- skipping white space during lexical analysis
- skipping over a comment
- decoding a tightly packed parse table during syntax analysis
- looking things up in the name table during semantic analysis
- determining whether some name is a reserved keyword or a user-definable identifier.

Optimisation within a compiler is concerned with improving in some way the generated object code **while ensuring the result is identical**. Technically, a better name for this unit might be "Improvement", since compilers only attempt to improve the operations the programmer has requested. Optimisations fall into three categories:

a. **Speed**: improving the runtime performance of the generated object code. This is the most common optimisation
b. **Space**: reducing the size of the generated object code
c. **Safety**: reducing the possibility of data structures becoming corrupted (for example, ensuring that an illegal array element is not written to).

Unfortunately, many "speed" optimisations make the code larger, and many "space" optimisations make the code slower. This is known as the space-time trade-off.

3.2 Criteria for Code-Improving Transformations

Simply stated, the best programme transformations are those that yield the most benefit for the least effort. The transformations provided by an optimising compiler should have several properties.

First, a transformation must preserve the meaning of programmes. That is, an "optimisation" must not change the output produced by a programme for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source programme. The influence of this criterion pervades this chapter; at all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the programme does.

Second, a transformation must, on the average, speed up programmes by a measurable amount. Sometimes we are interested in reducing the space taken by the compiled code, although the size of code has less importance than it once had. Of course, not every transformation succeeds in improving every programme, and occasionally an "optimisation" may slow down a programme slightly, as long as on the average it improves things.

Third, a transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programmes if this effort is not repaid when the target programmes are executed. Certain local or "peephole" transformations of the kind are simple enough and beneficial enough to be included in any compiler.

Some transformations can only be applied after detailed, often time-consuming, analysis of the source program, so there is little point in applying them to programs that will be run only a few times. For example, a fast, nonoptimising, compiler is likely to be more helpful during debugging or for "student jobs” that will be run successfully a few times and thrown away. Only when the programme in question takes up a significant fraction of the machine's cycles does improved code quality justify the time spent running an optimising compiler on the programme.

3.3 Improving Transformations

The code produced by straightforward compiling algorithms can be made to run faster using *code improving transformations*. Compilers using such transformations are called *optimizing compilers*.

The main subjects of research are *machine-independent optimisations*. They are implemented by algorithms that improve the target code without taking into consideration any properties of the target machine. Making machine-dependent optimisations, such as register allocation and utilisation of machine idioms is also possible.

The purpose behind making such optimisations is to make more efficient the most frequently executed parts of the compiler.

3.4 Optimising Compiler

An optimising compiler should provide the following properties:

- the transformations should preserve the semantics of the programmes, that is the changes should guarantee that the same input produces the same outputs (and should not cause errors)
- the transformations should speed up considerably the compiler on the average (although occasionally on some inputs this may not be demonstrated, on most of the inputs it should become faster)
- the transformation should be worth the intellectual effort.

3.5 Common Optimisation Algorithms

Common optimisation algorithms deal with specific cases. The possibilities to improve a compiler can be explained with the following most frequently applied transformation techniques:

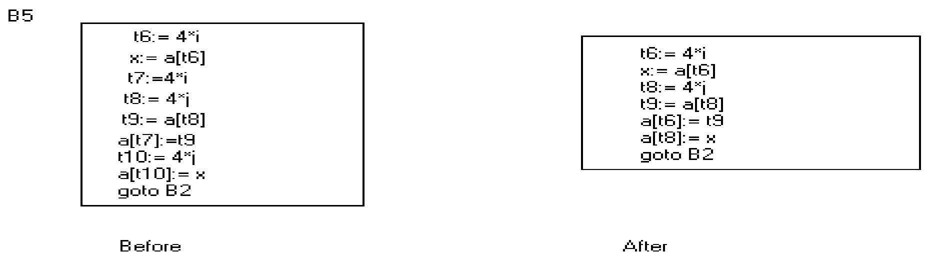
- Function-preserving transformations
- common sub-expressions identification/elimination
- copy propagation
- dead-code elimination
- Loop optimisations
- induction variables and reduction in strength
- code motion
- Function Chunking.

A code improving transformation is called *local* if it is performed by looking at statements within one concrete block. Respectively, a code improving transformation is *global* if it is performed by looking at statements not only in one concrete block, but also outside in global and other outside blocks.

3.5.1 Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common sub-expression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimisations are performed.

Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B5 shown in figure 1 recalculates 4*i and 4*j.



```
t6:= 4*i
x:= a[t6]
t7:=4*i
t8:= 4*j
t9:= a[t8]
a[t7]:=t9
t10:= 4*j
a[t10]:= x
goto B2
```

```
t6:= 4*i
x:= a[t6]
t8:= 4*j
t9:= a[t8]
a[t6]:= t9
a[t8]:= x
goto B2
```

Fig. 1: Local Common Sub Expression Elimination

```
void quick sort( int m, int n )
{
int i,j;
int v,x;

if ( n <= m ) return;
i = m - 1;
j = n;
v = a[ n ];
while ( 1 )
{
do i = i + 1; while ( a[ i ] < v );
do j = j - 1; while ( a[ j ] > v );
if ( I >= j ) break;
x = a[ i ];
a[ i ] = a[ j ];
a[ j ] = x;
}
x = a[ i ];
a[ i ] = a[ n ];
a[ n ] = x;
quicksort( m, j );
quicksort( i+1, n );
}
```

Three-address Code :

1) i := m - 1
2) j := n
3) t_1 := 4 * n
4) v := a[t_1]
5) i := i + 1
6) t_2 := 4 * i

16) t_7 := 4 * i
17) t_8 := 4 * j
18) t_9 := a[t_8]
19) a[t_7]:= t_9
20) t_{10} := 4 * j
21) a[t_{10}]:= x

7) t_3 := a[t_2]
8) if ($t_3 < v$) goto (5)
9) j := j - 1
10) t_4 := 4 * j
11) t_5 := a[t_4]
12) if ($t_5 > v$) goto (9)
13) if (i >= j) goto (23)
14) t_6 := 4 * i
15) x := a[t_6]

22) goto (5)
23) t_{11} := 4 * i
24) x := a[t_{11}]
25) t_{12} := 4 * i
26) t_{13} := 4 * n
27) t_{14} := a[t_{13}]
28) a[t_{12}]:= t_{14}
29) t_{15} := 4 * n
30) a[t_{15}]:= x

B_1
i := m - 1
j := n
t_1 := 4 * n
v := a[t_1]

B_2
i := i + 1
t_2 := 4 * i
t_3 := a[t_2]
if ($t_3 < v$) goto B_2

B_3
j := j - 1
t_4 := 4 * j
t_5 := a[t_4]
if ($t_5 > v$) goto B_3

B_4
if (i >= j) goto B_6

B_5
t_6 := 4 * i
x := a[t_6]
t_7 := 4 * i
t_8 := 4 * j
t_9 := a[t_8]
a[t_7]:= t_9
t_{10} := 4 * j
a[t_{10}]:= x
goto B_2

B_6
t_{11} := 4 * i
x := a[t_{11}]
t_{12} := 4 * i
t_{13} := 4 * n
t_{14} := a[t_{13}]
a[t_{12}]:= t_{14}
t_{15} := 4 * n
a[t_{15}]:= x

3.5.1.1 Copy Propagation

Input: A flow graph with a set of copies x= y that reach a block in the graph along every path, with no assignment of x or y following the last occurrence of x= y on the path to the block.

Output: A revised flow graph

Algorithm: For each copy x := y , do the following:

1) Determine those uses of x that are reached by this definition of x namely, *s* : x := y .
2) Determine whether for every use of x found in 1) **no** definitions of x or y can occur prior to this use of x within the ultimate block of use.
3) If the block *s* meets the conditions of 2) then remove *s* and replace all uses of x found in 1) by y.

Block B_5 in the code below can be further improved by eliminating x using two new transformations. One concerns assignments of the form f:=g called copy statements, or copies for short. For example, when the common sub expression in c:=d+e is eliminated in Figure 2, the algorithm uses a new variable t to hold the value of d+e. Since control may reach c:=d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c:=d+e by either c:=a or by c:=b.

The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement f:=g. For example, the assignment x:=t_3 in block B_5 of Figure 2 is a copy.

Copy propagation applied to B_5 yields:

x:=t_3
a[t_2]:=t_5
a[t_4]:=t_3
goto B_2

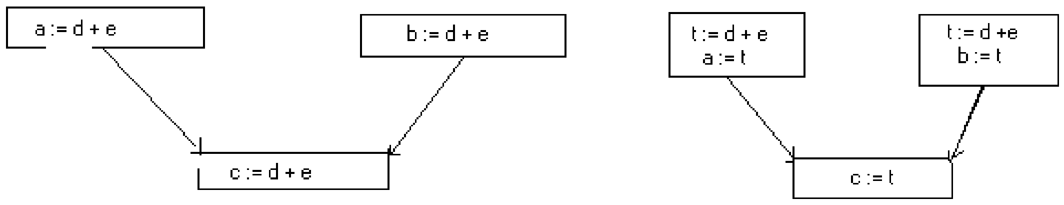


Fig. 2: Copies Introduced during Common Sub Expression Elimination

This may not appear to be an improvement, but as we shall see, it gives us the opportunity to eliminate the assignment to x

B_1
i := m - 1
j := n
t_1 := 4 * n
v := a[t_1]

B_2
i := i + 1
t_2 := 4 * i
t_3 := a[t_2]
if (t_3 < v) goto B_2

B_3
j := j - 1
t_4 := 4 * j
t_5 := a[t_4]
if (t_5 > v) goto B_3

B_4
if (i >= j) goto B_6

B_5
//x := t_3
a[t_2]:= t_5
a[t_4]:= t_3
goto B_2

B_6
//x := t_3
t_{14} := a[t_1]
a[t_2]:= t_{14}
a[t_1]:= t_3

3.5.1.2 Dead Code Elimination

Dead code elimination is a size optimisation (although it also produces some speed improvement) that aims to remove logically impossible statements from the generated object code. Dead code is code which will never execute, regardless of input

Consider the following programme:

```
a = 5
if (a != 5) {
// some complicated calculation
}
...
```

It is obvious that the complicated calculation will never be performed; since the last value assigned to **a** before the **IF** statement is a constant, we can calculate the result at compile-time. Simple substitution of arguments produces if (5 != 5), which is **false**. Since the body of an if(false) statement will never execute **-** it is *dead code* we can rewrite the code:

```
a = 5
// some statements
```

The algorithm was used to identify and remove sections of dead code

3.5.1.3 Common Sub-expression Identification/Elimination

Common sub-expression elimination is a speed optimisation that aims to reduce unnecessary recalculation by identifying, through code-flow, expressions (or parts of expressions) which will evaluate to the same value: *the re-computation of an expression can be avoided if the expression has previously been computed and the values of the operands have not changed since the previous computation*.

3.5.1.3.1Common Sub-expression Identification Algorithm

Input: A flow graph with available expression information
Output: A revised flow graph

Algorithm: For every statement *s* of the form x := y + z such that y + z is available at the beginning of *s*'s block, and neither y nor z is defined prior to statement *s* in that block, do the following:

1) To discover the evaluations of y + z that reach *s*'s block follow going through any block that evaluates y + z /. The last evaluation of y + z in each block encountered is an evaluation of y + z that reaches x.
2) Create a new variable u
3) Replace each statement w: = y + z found in (1) by

u := y + z
w := u

4) Replace statement *s* by x := u

Example 1: u := 4 * i

t_2 := 4 * i	t_2 := u
t_3 := a[t_2]	t_3 := a[t_2]
t_6 := 4 * i	t_6 := u
t_7 := a[t_6]	t_7 := a[t_6]

B_1
i := m - 1
j := n
t_1 := 4 * n
v := a[t_1]

B_2
i := i + 1
t_2 := 4 * i
t_3 := a[t_2]
if (t_3 < v) goto B_2

B_3
j := j - 1
t_4 := 4 * j
t_5 := a[t_4]
if (t_5 > v) goto B_3

B_4
if (i >= j) goto B_6

B_5	B_6
x := t_3	x := t_3
a[t_2]:= t_5	t_{14} := a[t_1]
a[t_4]:= x	a[t_2]:= t_{14}
goto B_2	a[t_1]:= x

Example 2:

Consider the following programme:
a = b + c
d = e + f
g = b + c

In the above example, the first and last statement's right hand side are identical and the value of the operands do not change between the two statements; thus this expression can be considered as having a *common sub-expression*.

The common sub-expression can be avoided by storing its value in a temporary variable which can cache its result. After applying this Common Sub-expression Elimination technique the programme becomes:

t_0 = b + c
a = t_0
d = e + f
g = t_0

Thus in the last statement the re-computation of the expression b + c is avoided.

3.5.2 Loop Optimisations

We now give a brief introduction to a very important place for optimisations, namely loops, especially the inner loops where programmes tend to spend the bulk of their time. The running time of a programme may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for loop optimisation: code motion, which moves code outside a loop; induction-variable elimination, which we apply to eliminate I and j from the inner loops B2 and B3 and, reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

There are three main techniques for loop optimisation (loops are usually processed inside out):

- *Strength Reduction* which replaces an expensive (time consuming) operator by a faster one;
- *Induction Variable Elimination* which eliminates variables from inner loops;
- *Code Motion* which moves pieces of code outside loops.

3.5.2.1 Strength Reduction

This concept refers to the compiler optimisation method of substituting some machine instruction by a cheaper one and still maintaining equivalence in results. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

This type of optimisation can generate high gains especially when targeting different hardware and the compiler is aware of the subtle differences it can benefit from.

Example 3:

Apply strength reduction to the code below:

```
$B_1$
i := m - 1
j := n
$t_1$ := 4 * n
v := a[ $t_1$ ]
$B_2$
$B_3$
j := j - 1
$t_4$ := 4 * j
$t_5$ := a[ $t_4$ ]
if ( $t_5$ > v ) goto $B_3$

$B_4$
if ( i >= j ) goto $B_6$

$B_5$                                    $B_6$
```

Solution:

As the relationship t_4:=4*j surely holds after such an assignment to t_4 in the code above and t4 is not changed elsewhere in the inner loop around B_3, it follows that just after the statement j:=j-1 the relationship t_4 := 4*j-4 must hold. We may therefore replace the assignment t_4:= 4*j by t_4:= t_4-4. The only problem is that t_4 does not have a value when we enter block B_3 for the first time. Since we must maintain the relationship t_4=4*j on entry to the block B_3, we place an initialisations of t_4 at the end of the block where j itself is initialised, shown by the last line of block B_1 in the code below

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines

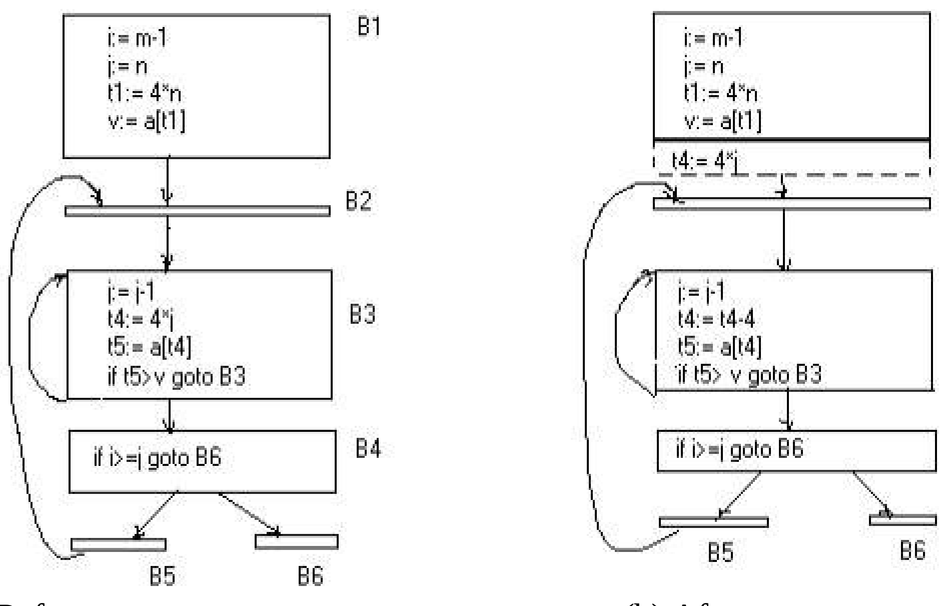
After Strength Reduction is applied to 4*j in block B_3 we have:

B_1
i := m - 1
j := n
t_1 := 4 * n
v := a[t_1]
t_4 := 4 * j
B_2
B_3
j := j - 1
t_4 := t_4 - 4
t_5 := a[t_4]
if (t_5 > v) goto B_3

B_4
if (i >= j) goto B_6

B_5 B_6

Diagrammatically it can be depicted as in figure 1 below



Before (b) After

Fig. 1: Strength Reduction Applied to 4*J In Block B_3

3.5.2.2 Induction Variable Elimination

Some loops contain two or more induction variables that can be combined into one induction variable. Induction variable elimination can reduce the number of additions (or subtractions) in a loop, and improve both run-time performance and code space. Some architectures have auto-increment and auto-decrement instructions that can sometimes be used instead of induction variable elimination.

Example 4:

For the code in example 3 above, consider the loop around B_3.
Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4.Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B_3 in Fig. we cannot get rid of either j or t_4 completely. t_4 is used in B_3 and j in B_4.

After Induction Variable Elimination is applied we have:

```
$B_1$
i := m - 1
j := n
$t_1$ := 4 * n
v := a[ $t_1$ ]
$t_2$ := 4 * i
$t_4$ := 4 * j

$B_2$
$t_2$ := $t_2$ + 4
$t_3$ := a[ $t_2$ ]
if ( $t_3$ < v ) goto $B_2$

$B_3$
$t_4$ := $t_4$ - 4
$t_5$ := a[ $t_4$ ]
if ( $t_5$ > v ) goto $B_3$

$B_4$
if ( $t_2$ >= $t_4$ ) goto $B_6$
```

B_5	B_6
a[t_2]:= t_5	t_{14} := a[t_1]
a[t_4]:= t_3	a[t_2]:= t_{14}
goto B_2	a[t_1]:= t_3

Example 5:

The code fragment below has three induction variables (i1, i2, and i3) that can be replaced with one induction variable, thus eliminating two induction variables.

```
int a[SIZE];
int b[SIZE];

void f (void)
{
int i1, i2, i3;

for (i1 = 0, i2 = 0, i3 = 0; i1 < SIZE; i1++)
a[i2++] = b[i3++];
return;
}
```

The code fragment below shows the loop after induction variable elimination.

```
int a[SIZE];
int b[SIZE];

void f (void)
{
int i1;

for (i1 = 0; i1 < SIZE; i1++)
a[i1] = b[i1];
return;
}
```

3.5.2.3 Code Motion

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop.

This optimisation technique mainly deals to reduce the number of source code lines in the programme. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
While (i<= limit-2 )
```

Code motion will result in the equivalent of

```
t= limit-2;
while (i<=t)
```

Example 6:

Consider the code below:

```
for (i = 0; i < n; ++i) {
x = y + z;
a[i] = 6 * i + x * x;
}
```

The calculations x = y + z and x * x can be moved outside the loop since within they are looping invariant (i.e. they do not change over the iterations of the loop) so our optimised code will be something like this:

```
x = y + z;
t1 = x * x;
for (i = 0; i < n; ++i) {
a[i] = 6 * i + t1;
}
```

This code can be optimized further. For example, strength reduction could remove the two multiplications inside the loop (6*i and a[i]).

Example 7:

```
for(i=0;i<10;i++)
{
a = a + c;
}
```

In the above mentioned code, a = a + c can be moved out of the **'for'** loop, and the new code is

```
a = a + 10*c;
```

3.5.3 Function Chunking

Function chunking is a compiler optimisation for improving code locality. Profiling information is used to move rarely executed code outside of the main function body. This allows for memory pages with rarely executed code to be swapped out.

4.0 CONCLUSION

In this concluding unit of the course, you have been taken through the concept of code optimisation and common optimisation algorithms. Dramatic improvements in the running time of a programme-such as cutting the running time form a few hours to a few seconds-are usually obtained by improving the programme at all levels, from the source level to the target level. At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations are performed.

5.0 SUMMARY

In this unit, you learnt that:

- optimisations fall into three categories: speed, space, safety
- the best programme transformations are those that yield the most benefit for the least effort
- the code produced by straightforward compiling algorithms can be made to run faster using *code improving transformations*
- compilers using such transformations are called *optimising compilers*
- a code improving transformation is called *local* if it is performed by looking at statements within one concrete block
- a code improving transformation is *global* if it is performed by looking at statements not only in one concrete block.

6.0 TUTOR-MARKED ASSIGNMENT

i. Define the following concepts:
   a) Optimising compilers
   b) loop optimisation
   c) Function chunking
   d) Strength reduction.
ii. State criteria for code-improving transformations.
iii. Briefly explain what you understand by space-time trade off in optimisation.

7.0 REFERENCES/FURTHER READING

Alfred, Aho V. *et al*. (2007).*Compilers: Principles, Techniques, and Tools*. Second edition. Wesley:Pearson Addison.

,Andrew, Appel W. (2002). *Modern Compiler Implementation in Java*. Second edition. Cambridge University Press.

Cooper, Keith D. & Torczon, Linda (2004). *Engineering a Compiler*. Morgan Kaufmann.

Muchnick, Steven S.(1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Scott,Michael L. (2009). *Programming Language Pragmatics*. Third edition. Morgan Kaufman.

Sebesta, Robert W. (2010).*Concepts of Programming Languages*. Ninth edition, Addison-Wesley.