CSC 310: Compiler Construction I

By

Olatunji, E.K.

Computer Sc Programme
FCAS
Thomas Adewumi University, Oko Iwo

April 2024

Course Description

- The course introduces the design and implementation of a compiler with emphasis on principles and techniques for program analysis and translation.
-
- It also gives an overview of the tools for compiler construction. Lexical analysis, token selection, transition diagrams, and finite automata. The use of context-free grammars to describe syntax, derivations of parse trees, and construction of parsers. Syntax-directed translation schemes; Intermediate code; Symbol table; Code generation; Detection, reporting, recovery, and correction of errors.

Course Objectives

- **Course Objectives are to**:
- 1. Describe and distinguish among the main PL translators
- 2. Describe the functions of the main components of a typical Compiler: lexical analyzers, Parsers, etc
- 3. Draw and describe the structure of a typical compiler
- 4. Introduce concept of formal grammar and languages

Expected Learning outcomes

- After the course, students should be able to:
- Explain functions of the 3 main translators
- Distinguish among the 3 main PL translators
- Describe functions of 5 main components of a compiler
- Draw and explain the structure of a typical compiler
- Give 5 types of internal form of source program
- List out the tokens of a program statement in a familiar PL
- Distinguish between the phase and the pass of a translator
- etc

Course Outline

- xx

S/N	week	Topics	Duration
Durati on	Week 1-2	Introduction to PL translators	6 hours
2.	Week 3-4	Compilation Processes in Brief	6 hours
3	Week 5-6	Lexical Analysis in more details	6 hours
4	Week 7	CA 1	3hours
5	Week 8-9	Syntax analysis in details	6 hours
6	Week 10-11	Semantic Analysis	6 hours
7	Week 12-13	Code generation and Optimization	6 hours
8	Week 14	CA 2	3 hours
9	Week 15	Introductory Formal grammar and languages	3 hours
10	Week 16	Final Exam	3 hours

Lecture Outlines

- ❑ Part I: Introduction
- Introduction to Compiler
- Compilation Processes in Brief
- ❑ Part II – Compilation Processes in Details
- ❑ Part III – Compilation Processes in More Details

Course Outline

- Introduction to Compiler
- Compilation process in Brief
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Code Generation

Recommended Reference materials

- 1. Compiler Techniques (An Introductory Text on Concepts and Principles) by E. K. Olatunji
- 2. Principles of Compiler Design by Aho & Ullmam
- 3. Compiler Construction for Digital Computers by David Gries
- 4. Computer Science by C.S. French, @ BookPower, 5th edition
- 5. Compilers: principles, Techniques and tools by Aho, Lam, Sethi & Ullmam @ 2007
- 6. Online Resources

Introduction to Compilers

❑**What is a compiler?**

- It is a programming language (PL) translator
- An H.L.L. source program translator, Usually into machine language (ML) equivalent
  - i.e, translates source program in HLL to its ML equivalent
- Its output is called **object program/object code**
- Its input is called **source program**
- Process of translation is called **compilation**
- Period of translation is called **compile time**

Introduction to Compilers Contd

- ❑**Other PL translators**
  - Interpreters – Pseudo translation + execution
  - Assemblers translates assembly language (AL) into ML
- ❑**Linkage editor** – combines 2 or more object modules into one single executable code/program
- ❑**Language for writing compilers**
  - AL eg, early compiler
  - HLL eg, LISP, C compilers

Evolution of Compiler

- Problems of ML & AL which led to development t of HLL
  - ➢difficulty in programming, debugging and maintenance
  - ➢Portability problem due to machine specificness of ML and AL
  - ➢Relocation problem of ML
- The need to translate HLL to make the program understandable and executable by the computer machine

Types of Compiler

- Native code/conventional/traditional
  - Translate a HLL source code to the ML of a specific target computer
  - Output medium is a disk for later execution
- Cross compiler
  - Produces object code that can be run, not only the target compiling machine, but also on other machines of different configurations
- Load-and-go compiler
  - Produces object code in ML directly into main memory for immediate execution

Types of Compiler

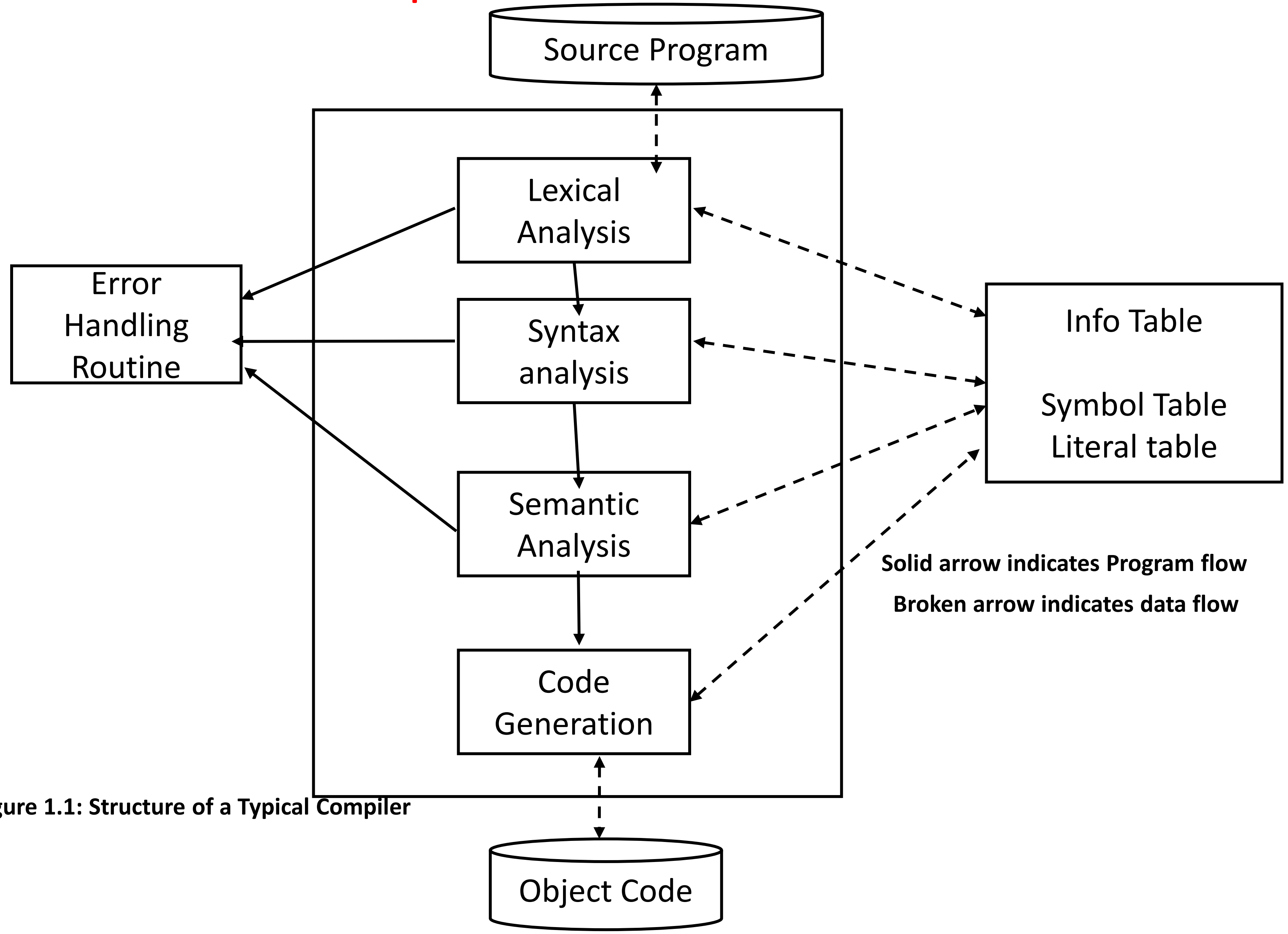
- Stage compiler
    - Produces object code in the AL of the target machine
- Just-in-time compiler
    - Produces as output byte code, which are later compiled to the native code prior to execution
    - It is subset of the stage compiler
- Source-to-source compiler
    - Its output is in another HLL

Section 2: Compilation Process in brief

- The compilation process consists of the following well-defined tasks; viz:
  - Lexical Analysis
  - Syntax analysis
  - Semantic analysis
  - Code optimization – machine - independent
  - Storage allocation
  - Code Generation
  - Code optimization II – Machine -dependent
  - Info Table management
  - Error Handling

  The structure of a typical compiler is as in the next slide (Figure 1.1)

Section 2: Compilation Process in brief Contd



Solid arrow indicates Program flow

Broken arrow indicates data flow

- **Figure 1.1: Structure of a Typical Compiler**

Section 2: Compilation Process Contd

- These compilation tasks may be grouped into a number of phases depending on the whims and caprices of the designer/implementor
- A phase of compilation usually involves performance of one or more of the well-defined tasks
- Examples:
    - 2-phases: Analysis (Lexical, syntax &semantic) and Synthesis(Code generation and code optimization); sometimes respectively known as front-end and back-end.
    - 3-Phases:Lexical analysis, syntax & semantic analysis and Code generation
    - 4-phases: Lexical Analysis, Syntax & Semantic Analysis, Preparation for code generation (Storage assignment , machine –independent optimization), and Code generation

Compilation Tasks in brief

- **Lexical Analysis**
  - Scans source program statements, character by character (from left to right)
  - Forms tokens of the language (i.e, the smallest unit of a source program)0 Examples of tokens are keywords, identifiers, literals, operators, etc
  - Sends each token in its internal representation form to the next stage (syntax analysis). This provides uniform ways of representing and handling tokens. For e.g, unique integers can be used as internal representation of each token
  - Eliminates redundancies (such as comments, spaces, etc)
  - Detects errors, eg, illegal character, improperly formed identifiers, etc.
  - The process is carried out by a routine called lexical analyzer or scanner

Compilation Tasks in brief

- **Syntax analysis**
    - Performs syntactic checks on the source program statements. That is it ensures that rules governing formation of valid statements are obeyed. Specifically it ensures that appropriate tokens of syntactic unit follow each other in appropriate sequence
    - Builds up various table of info; e,g symbol table, table of constants, etc
    - Conceptually creates a parse tree as output. This is an intermediate representation that shows the grammatical structure of the tokens stream. See example below!!
    - Reports errors, such as multiple declaration of identifiers & other violations of the syntax of the PL
    - Carried out by a routine called parser or syntax analyzer

Compilation Tasks in brief Contd

- **Semantic analysis**
  - Checks for the meaning of a syntactic unit; e.g, checking for type compatibility
  - Breaks complex statements , e.g For-lops, into its simple primitive forms
  - Ensures that every if-statement, Do statement, etc has corresponding matching ENDif, Enddo, etc respectively
  - Ensures all referenced labels are uniquely defined
  - Creates intermediate form of source program for every recognized semantically well-formed statements, using either, syntax tree, polish notation, etc
  - Carried out by semantic analyzer or interpretive routine

Compilation Tasks in brief Contd

- **Code optimization – machine – independent**
  - This is elimination / removal of semantically redundant codes ; e.g consecutive store and load operations on the same operand
  - For the purpose of improving the codes' efficiency (run-time performance & storage requirements)

- **Storage allocation**
  - Storage is allocated to all variables/identifiers (user-defined & system generated) prior to the next stage

Compilation Tasks in brief contd

- **Code Generation**
  - Actual translation of the source code (actually the intermediate codes) into the actual machine language codes

- **Code optimization II – Machine –dependent**
  - Optimizes code to improve program efficiency – in terms of storage and execution time.. Machine-dependent optimization include better utilization available registers o and instruction set

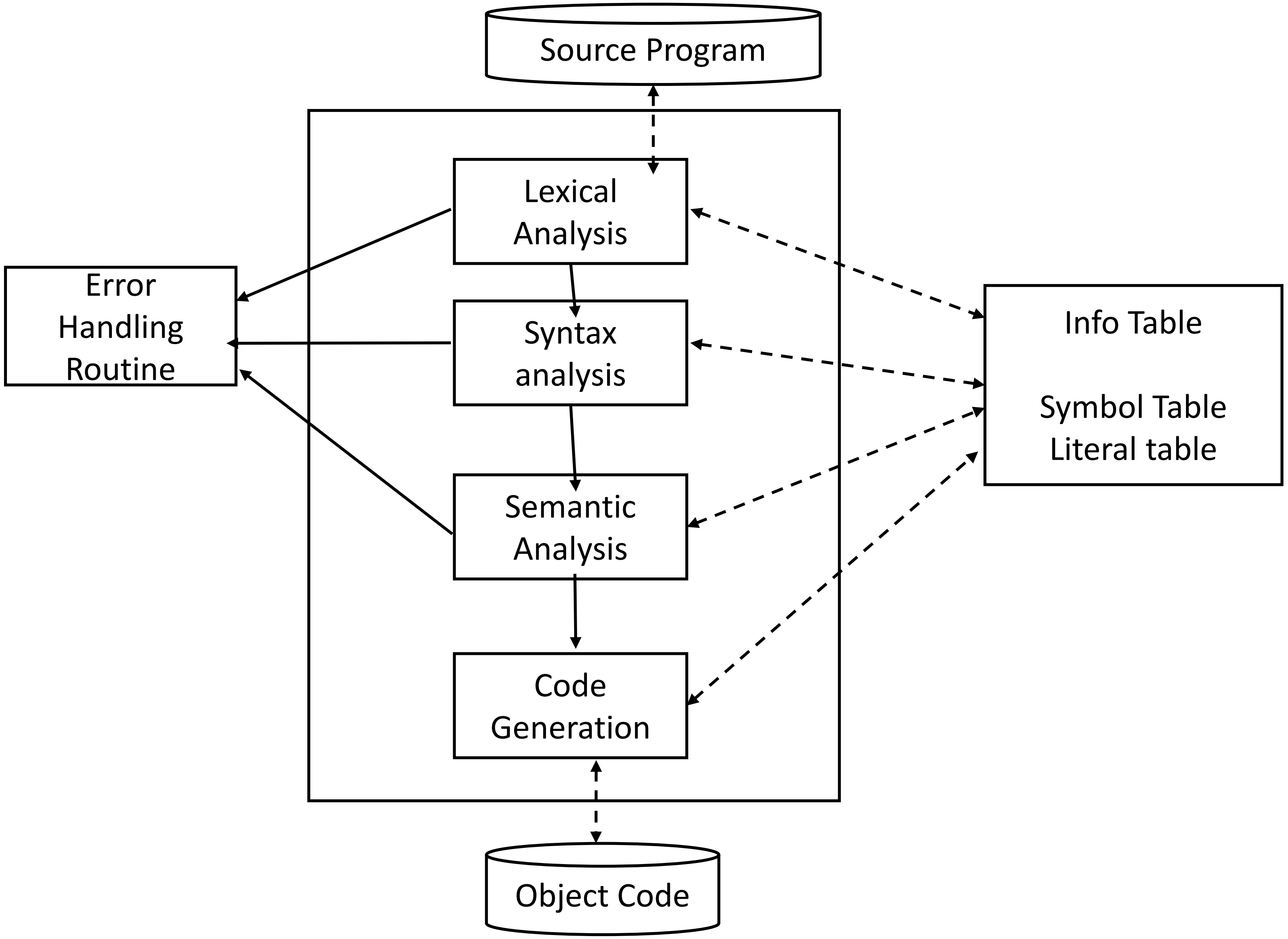
Compilation Tasks in brief contd

- **Info Table management**
  - A no. of tables are created maintained and manipulated during the process of compilation. This include symbol table, literal table and table of terminal symbol(.e, reserved words, operator symbols, etc)

- **Error Handling**
  - Errors can be detected at any stage of the compilation process, especially the first 3 stages. These errors must be reported. Aside, there would be a way to recover and be able to continue especially till the end of the semantic analysis stage.

Exercises

- 1. What gave rise to Compiler?
- 2.Describe the distinct features of the following types of compiler
    - (i) Native-code compiler (ii) cross compiler
    - (iii) load and go (iv) source-to-source compiler
- 3.Distinguish among compiler , interpreter and assembler
- 4. itemize the functions of the following routines of a compiler: (i) code optimizer (ii) scanner (iii) parser (iv) semantic analyzer (v) Code generator (vi) error handler
- 5. List and explain 3 types of info table known to you!


Source Program
Lexical Analysis
Syntax analysis
Semantic Analysis
Code Generation
Error Handling Routine
Info Table
Symbol Table
Literal table
Object Code

CSC 310: Compiler Construction I

By

Olatunji, E.K.

Computer Sc Programme
FCAS
Thomas Adewumi University, Oko

April 2024

Course Contents

❑Part I: Introduction
- Introduction to Compiler
- Compilation Processes in Brief

❑Part II – Compilation Processes in Details
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Code Generation
- Part III – Formal Grammar
- Intro to Formal Grammar

❑Part IV – Compilation Processes in More Details

Course Outline

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Code Generation

Recommended Reference materials

- 1. Compiler Techniques (An Introductory Text on Concepts and Principles) by E. K. Olatunji; 2006
- 2. Principles of Compiler Design by Aho & Ullmam
- 3. Computer Science by C.S. French, @ BookPower, 5th edition
- 4. Compilers: principles, Techniques and tools by Aho, Lam, Sethi & Ullmam @ 2007
- 5. Online Resources

Section I: Lexical Analysis

- **Lexical Analysis**
  - Scans source program statements, character by character (from left to right)
  - Forms tokens of the language (i.e, the smallest unit of a source program)0 Examples of tokens are keywords, identifiers, literals, operators, etc
  - Sends each token in its internal representation form to the next stage (syntax analysis). This provides uniform ways of representing and handling tokens. For e.g, unique integers can be used as internal representation of each token
  - Eliminates redundancies (such as comments, spaces, etc)
  - Detects errors, eg, illegal character, improperly formed identifiers, etc.
  - The process is carried out by a routine called lexical analyzer or scanner

Lexical Analysis Contd

❑ **At this stage, the source program (SP) is broken into its smallest unit called tokens**

❑ **Most PL tokens are in category**:

- Identifiers or variable names
- Key words e.g : IF, else, do, for
- Special operators e.g <, >, =, +, etc
- Delimiters e,g comma, parenthesis, semi-colon
- Literals – numeric and non-numeric
- Separators, eg blanks

❑ **A separator can be any of:**

- A blank xter
- Conventional arith operators
- Delimiters as described above
- A character that does not actually belong to d set of d expected xter that form a particular token category; for example, in d token class integer, a scan of any xter other than digits 0-9 will serve as a separator

- !

Lexical Analysis Contd

❑**Input to scanner**

A string of xters that form program source statement

❑**Output of scanner**

- PL tokens – in their internal representation form & the actual token
- Internal representation code ID, LT, DL can respectively stand for identifiers, literals, delimiters
- Alternatively, different integers can be used to repr token types
- E.g, integer 01, 02, 03, can be use d to repr token type id, int, other literal
- Unique integer can be used to repr unique reserved words/keywords!

Lexical Analysis Contd

❑**Example of Lexical Analysis**

Given the following program statement in a familiar PL:

```
for i =1; i < n; i++
    cout << I;
```

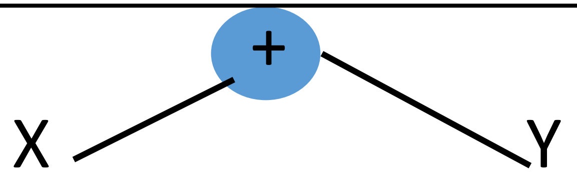
The output of the scanner will be as shown in figure/Table X.X1: on the RHS:

s/no	Internal Repr	Actuial Value=
1	RWD	for
2	IDN	i
3	OPR	=
4	NLT	1
5	DEL	;
6	IDN	i
7	OPR	<
8	IDN	n
9	DEL	;
10	IDN	i
11	OPR	++
12	RWD	cout
13	OPR	<<
14	IDN	i
15	DEL	;

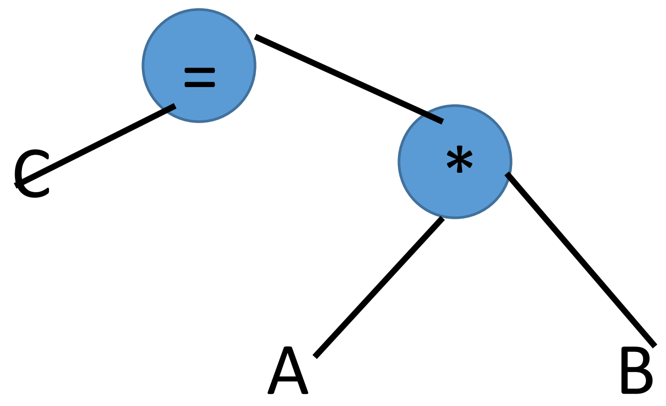
Section II: Syntax Analysis

- **Syntax Analysis**
  - Performs syntactic checks on the source program statements. That is it ensures that rules governing formation of valid statements are obeyed. Specifically it ensures that appropriate tokens of syntactic unit follow each other in appropriate sequence
  - Builds up various table of info; e,g symbol table, table of constants, etc
  - Conceptually creates a parse tree as output. This is an intermediate representation that shows the grammatical structure of the tokens stream. See example below!!
  - Reports errors, such as multiple declaration of identifiers & other violations of the syntax of the PL
  - Carried out by a routine called parser or syntax analyzer

Parse Tree



- Figure 1: Parse Tree for X+Y



- Figure 2: Parse tree for C=A*B

Syntax Analysis Contd

- **At Syntax Analysis Stage/phase**
- Complete syntactic checks are carried out on the source program (SP)
- Syntax of a PL are rules of combining tokens together in appropriate sequence to form a valid syntactic construct
- Syntactic construct are phrases with associated meaning . Examples are
    - Expression(arith, relational, etc)
    - Assignment expression
    - Declarative statements (e.g int a;
    - For-loops
    - Jump instructions
- Thru a no. of techniques, the syntax analyzer or parser is able to detect whether or not syntactic entities are formed by appropriate combination of appropriate small syntactic entities and/or token
- Various tables of info; (e,g symbol table, table of constants, etc) are built

Syntax Analysis Contd

- **Symbol Table**
  - also called identifier table
  - Contains the identifier used in d SP along with their attributes
  - Identifier attributes include : type, form, block number, etc
  - ** more detail later
- **Parsing Techniques**
  - Operator precedence – mostly suitable for parsing expression
  - Recursive descent – uses a collection of mutually recursive routines to perform the syntax analysis
- **Exercise:**

- Describe each of the above techniques with an example; citation and listing of reference expected in APA format; Due in a week's time

Assignment – due on the date specified

- 1. Describe how d tokens of PL can be recognized
- 2. What are d output and input of a scanner
- 3. For the program segment below; write out the tokens available in the program:
  - int a, float b;
  - Scanf (“%d%f”, a,b);
- 4. what is a parser? What are its input and outputs?

Section III: Semantic Analysis

- **Semantic analysis**
    - Checks for the meaning of a syntactic unit; e.g, checking for type compatibility
    - Breaks complex statements , e.g For-loops, into its simple primitive forms
    - Ensures that every if-statement, Do statement, etc has corresponding matching ENDif, Enddo, etc respectively
    - Ensures all referenced labels are uniquely defined
    - Creates intermediate form of source program for every recognized semantically well-formed statements, using either, syntax tree, polish notation, etc
    - Carried out by semantic analyzer or interpretive routine

Semantic Analysis

- Also called interpretive routine
- It consists of smaller semantic analyzing interpretive routine, each of which handles one particular type of program construct
- For e.g; array declaration might be handled by one analyzer, arith express by another; etc
- The appropriate interpretive routine is called / invoked when a syntactic unit is recognized

Semantic Analysis Contd

- **Intermediate form of SP (IFSP)**
- For practical purposes, d compiler usually creates an intermediate form of the source program(IFSP) instead of generating code directly for each instruction. The IFSP offers 2 advantages:
  - Facilitates (machine independent) code generation
  - Provides clear separation between machine independent phase (The analyzes) and machine dependent (code generation)

  ** more of this later

  - **Examples of IFSP** are: parse tree, syntax tree, polish notations, List of quadruples, etc

Section IV: Code Generation

- This phase converts an IFSP into a sequence of machine instruction
- For e.g; d statement A=B-C can become the following sequence of Assembly language (AL) codes:
- LOAD B
- SUBTRACT C
- STORE A
- Note that there is 1 – 1 translation from AL to ML

Section IV: Code Generation Contd

- Another example;. The statement: W=X*Y+Z can become the following sequence of Assembly language (AL) codes:
- LDA X
- MUL Y
- STA T1
- LDA T1
- ADD Z
- STA T2
- LDA T2
- STA W
- Where t1 & t2 are temporary compiler generated variables

Code Generation Contd

❑**Code generator**

- This is the routine that handles code generation
- **Input** to it is diff forms of IFSP
- **Output** is called object code, which can be in diff forms: ML, AL
- **Note**:
  - A code generator will make ref to the symbol table, literal table, to generate correct ML codes

Code Generation Contd

❑**Types of object code**

- Absolute ML instruction
- AL program – to be later assembled
- Relocatable ML – this is the conventional object code; requires linkage editing
- A program in another HLL
- **Note**: Each of the above has pros & cons
- ****Exercises**:
- 1. Describe the pros and cons of each type of object code
- 2. give 2 examples of compiler that have their output in each of the diff 4 types of output possible

Code Generation Contd

❑**Storage allocation**

- Prior to actual generation of code, storage is normally assigned to all identifiers (user-defined & compiler generated) and literals
- The storage allocation routine scans the **symbol table** & assigns to each scalar variable a location
- For e.g; it can assign to d 1st variable, relative location 0, d 2nd to location 4, 3rd to location 8 and so on; assuming each variable requires 4 bytes of storage
- The **literal table** is similarly scanned and allocation made to each entry

Code Generation Contd

❑**Storage Allocation Example**

- Given d following C program declaration:
- int a, float b; double c; char d;
- The relative location assigned to each
  - variable will be as in the fig/tab below:
- In d fig, and d particular machine in use;
- Variables a, b, c, d are
  - integer, float, double and char values respectively
  - requires 2, 4, 8 and 1 byte(s) of storage

Symbol Name	Other attributes	Storage address
a		0
b		2
c		6
d		14

- **Fig XX.X: Symbol table showing storage allocation in a C program**

❑**Code optimization**

- More on this later!

Section V: Error Detection & Recovery

- A good compiler should be able to:
  - Detect and report on as many errors as possible during compilation
  - Recover from where error is detected; it should be able to continue scanning thru the remaining SP
- **Program Errors**
  - Compilation errors – detectable by compilers
    - (Can be lexical, syntax or semantic errors)
  - Run-time error – detectable at execution time
  - Logical errors – detectable by programmer

Error Detection & Recovery Contd

- **Compilation Errors**
- **Lexical error**
  - E.g invalid character, improperly formed identifier
- **Syntax error**
  - This is violation of the syntax of the PL; e.g
  - Wrongly spelt reserved word
  - Extraneous blank character within an id or reserved word
  - Missing right or left parenthesis

Error Detection & Recovery Contd

- **Compilation Errors Contd**
- **Semantic error**
  - Detectable at semantic analysis stage; e.g
  - Undeclared identifier
  - Multiplicatively declared id / Multiple declaration of an id
  - Type incompatibility
  - Missing matching Endif for an if-construct
- etc

Assignment – due on the date specified

- 1. Describe d action of a semantic analyzer on the following syntactic construct:
  - Assignment stmt
  - Declaration stmt
  - Arith expr
- 2. Describe 4 forms of output of a code generator
- 3. Generate AL code on our fictitious computer for the arithmetic expression:
  - a – b (c/d + e/f)
- 4. Show a part of symbol table that shows how storage are allocated to d C declaration: int a, double c, char b;

Formal Grammar Contd

CSC 310 : Compiler Construction I

By

Olatunji, E.K.

Computer Sc Programme
FCAS
Thomas Adewumi University, Oko

March 2024

Course Contents

❑ Part I: Introduction

- Introduction to Compiler
- Compilation Processes in Brief

❑ Part II – Compilation Processes in Details

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Code Generation
- Intro to Formal Grammar

❑ Part III – Compilation Processes in More Details

Course Outline

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Code Generation
- Intro to Formal Grammar

Recommended Reference materials

- 1. Compiler Techniques (An Introductory Text on Concepts and Principles) by E. K. Olatunji
- 2. Principles of Compiler Design by Aho & Ullmam
- 3. Compiler Construction for Digital Computers by David Gries
- 4. Computer Science by C.S. French, @ BookPower, 5th edition
- 5. Compilers: principles, Techniques and tools by Aho, Lam, Sethi & Ullmam @ 2007
- 6. Online Resources

Preambles on Languages and Related Terms

- Preambles on Languages and Related Terms
- Preambles on Languages and Related Terms

Language Description / Description of Language

- **Language Description / Description of Language**
- **What is a language**?
- It is a system of communication, which consists of a set of sounds and written symbols, which are used by the people of a particular community or culture for talking or writing (Collins English dictionary) and may also be conveyed thru sign languages.
- Thus a l**anguage** consists of finite set of valid symbols (or characters, such as: A, z, @, 2, $, etc) from which valid words of the language are formed.
- A language is described by a set of rules (called **grammar**) that govern the combination of valid words to produce meaningful units like phrases and sentences

Language Description Contd

- Related Language Terms
- A **lexical item** is a single word or part of a word that forms the smallest meaningful unit of a language
- **Lexicon** refers to the complete list of valid words in a particular language.
- Lexis refer to words, while structure refers to the organized arrangement of words to form meaningful units, such as a phrases, clause, sentences, etc
- The lexical structure of a PL, for example, determines what group of symbols or characters are to be treated as identifiers, reserved words, operators, etc.

Language Description Contd

❑**Related Language Terms contd**

- In linguistics, **syntax** refers to the rules that govern the ways in which words combine together to form phrases, clause, sentences, etc. The syntax rule of a language determines why a sentence in English language such as “I hate you” is meaningful but “hate you I” is not. Every language has its own specific rules or syntax
- **Lexical syntax** determines how a character sequence is spilt into a sequence of **lexemes**, omitting non-significant parts, such as comments and whitespaces. The character sequence is assumed to be a text according to the Unicode standard (according to GNU.org (n.d)).
- Put another way, **lexical syntax** is the set of rules that enable lexical items of a language to be identified in a text consisting of sequence of (Unicode)characters.
- Oxford Dictionary (1995) describes a **grammar** as the study of science of rules for the combination of words in a language. It is a set of rules that describe a language.
-

Language Description Contd

- **Distinguishing between Grammar and Syntax of a language**
-
- Grammar Versus syntax from: https://prowritingaid.com/grammar-vs-syntax 13-03-2023
- Grammar represents an entire set of rules for language, and syntax is one section of those rules
- **Grammar** comprises the entire system of rules for a language, including syntax. **Syntax** deals with the way that words are put together to form phrases, clauses, and sentences.

Language Description Contd

- **Distinguishing between Grammar and Syntax of a language Contd**
- **Definition & Meaning of Grammar (**: https://prowritingaid.com/grammar-vs-syntax 13-03-2023
- Grammar is the set of rules a language follows to convey meaning. Grammar is a broad term that encompasses more specific areas of study including:
- **Morphology**: how words are formed and how adding or removing word parts can change the tense or part of speech. For example, “nation” is a noun. Adding “al” to the end of the word (’national’) changes this part of speech to an adjective.
- **Phonology**: how the parts of language sound.
- **Semantics**: what words or symbols mean.
- **Syntax**: how words are put together to create phrases, clauses, or sentences.
- All syntax rules are grammatical rules but not all grammatical rules are syntax rules!

Section VI:

Introduction to Formal Grammar

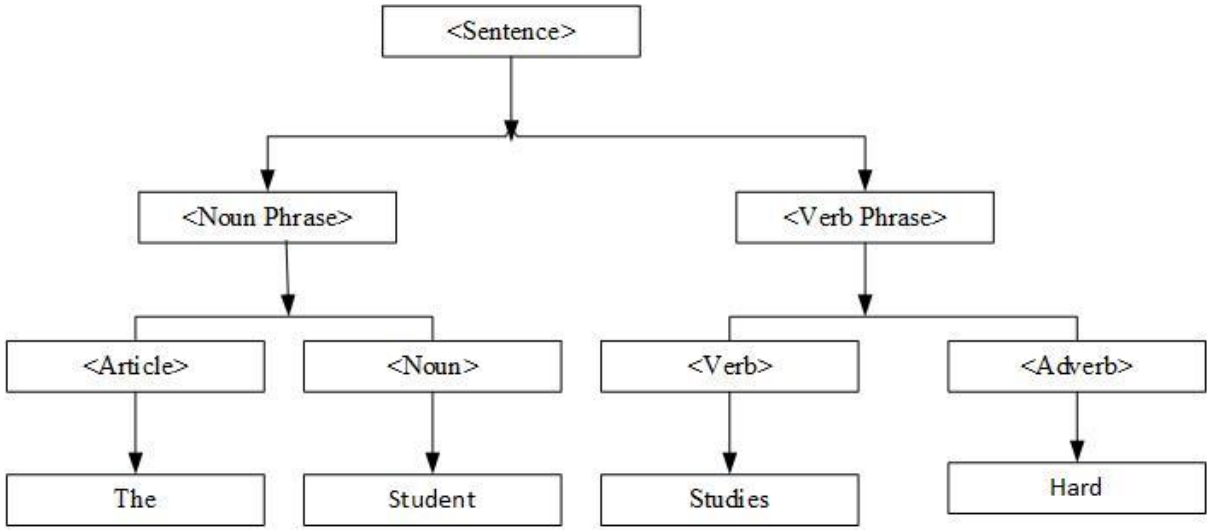
Section VI: Introduction to Formal Grammar and Languages

- **Language**
  - Informally defined as a notation for communication
- **Grammar**
  - The study of science of rules for the combination of words in a language(Oxford Dictionary, 1995)
  - E.g “Ade is a boy” –syntactically correct by the rules of English language
  - But “these is a mangoes” !! Syntactically invalid
- **Lexicons**
  - List of valid words in a particular language
- **Syntax**
  - Rules that govern combination of valid words to form a valid statement in a language
- **Semantics**
  - Literal meaning associated with a syntactic entity
  - In programming language (PL), it means the run-time effect of a syntactic entity

Introduction to Formal Grammar and Languages Contd

- **Semantics**
  - Literal meaning associated with a syntactic entity
  - In programming language (PL), it means the run-time effect of a syntactic entity
- **Pragmatics**
  - Words that do not connote a literal meaning, but has a hidden meaning
  - E.g; “the man kicks the buckcet”
  - “Okun rin na gba ekuru je lowo ebora”
  - In PL, it refers to implementation techniques, programming style / methodology & historical development of a language
- **Syntax Tree**
  - A tree-like diagram that describe the syntax or grammatical structure of a sentence by breaking it into its constituent parts

Introduction to Formal Grammar and Languages Contd

- 

- Figure X.1: A Syntax tree for a simple sentence in English Language

Introduction to Formal Grammar and Languages Contd

- The diagram in Fig. X.1 above is read or interpreted as:
  - a <sentence> is composed by a <Noun Phrase> followed by a <Verb Phrase> .
- Similarly,
  - a <Noun Phrase> is composed of an <article> followed by a <Noun>, etc
- **Meta Language**
  - A language used for describing other languages; e.g; using Yoruba to teach English in primary school
- **BNF (Backus Normal Form)**
  - A notation or meta language for describing the ngrammar that defines a PL

BNF

- **BNF (Backus Normal Form)**
  - A notation or meta language for describing the grammar that defines a PL
- There are a no. of symbols used; e.g
  - ::= or ---> denotes “may be replaced by” or “may be defined by”
  - < > is used to enclose syntactic entities or non-terminal symbols of a language or grammar; e.g <sentence>
  - | means “or”
  - E.g; <Operand> ---> X|Y|Z
  - That is, an <operand> in a language can be either X or Y or Z
- The syntax tree in Fig x.1 can be defined using a BNF notation as follows”

BNF Contd

- The syntax tree in Fig x.1 can be defined using a BNF notation as follows”
  - <sentence> ---> <noun Phrase><Verb Phrase>
  - <Noun Phrase> ---> <Article><Noun>
  - “
  - “
  - <article> ---> The
  - <Noun> ---> student
- A syntax tree written in BNF notation as above is called rules of production of a grammar.

Grammar

- **Grammar**
  - Has been informally defined
- **Formally,**
  - A grammar is defined as consisting of 4 components: G=(V,T,S,P)
- Where
  - V = the set of non-terminal symbols, e.g; <sentence>
  - T = set of terminal symbols e.g, The, student, studies, etc
  - S = the starting symbol; a distinguished non-terminal symbol from which all strings of a language are derived
  - P = production rules written as A ---> B
- The word token is synonymous with terminal symbols
- Non-terminal symbols are special symbols that denote set of strings in the language – also called syntactic variables or entities!
- Eg;
  - <statement> ---> begin <statement-list> end
  - <statement-list> ---> <statement|<statement>;<statememt-list>
- The keywords begin, end and the semi-colon are the terminal symbols or tokens of the language

- **Sometimes**, capital Roman letter can be used for non-terminals, while small Roman letters are used for terminal symbols!

Derivation from Formal Grammar

- How are strings / word (sentence or form of sentence) of a language derived from a given grammar?
- You do this by re-writing the non-terminal symbol into another form by applying one or more of the production rules in any other
- For example
  - <Print-stmt> ---> PRINT|PRINT <Expr>
  - <Expr> ---> <Variable> | <variable> , | <variable>;
  - <Variable> ---> <letter> | <letter> <digit>
  - <letter> ---> A|B|C|…|Z
  - <digit> ---> 0|1|2|…|8|9
- This is the grammar of a simple version of the PRINT statement in BASIC!
- From this grammar:
  - Terminal symbols are: PRINT, Comma, Semi-colon, Letters A-Z and Digits 0-9
  - Non-terminal symbols are: <Pint-stmt>, <Expr>, <variable>, <Letter> and <digit>

Derivation from Formal Grammar

- The following would be legal from the above grammar
    - (i) PRINT A
    - (ii) PRINT M4;
    - etc
- **How?** – by re-writing the non-terminal symbols beginning from the START symbol of the grammar
    - <Print-Stmt> ===> PRINT <Expr> (rule 2)
    - ===> PRINT <ariable> (rule 3)
    - ===> PRINT <Letter> (rule 6)
    - ===> Print A (rule 9)
- **Where**
    - Rule 1 = <Print-Stmt> ---> PRINT (of line 1 of the grammar)
    - Rule 2 = PRINT <Expr> (of line 1 of the grammar)
    - Rule 3 =<Expr> ---> <Variable> (of line 2 of the grammar)
    - Rule 5 =<Expr> ---> <Variable>; (of line 2 of the grammar)
    - Rule 10 =<Letter> ---> B (of line 4 of the grammar)

Derivation from Formal Grammar

- **Leftmost derivation**
  - This is a derivation that always re-writes the leftmost non-terminal symbol first. The derivation above is a leftmost derivation
- **Rightmost derivation**.
  - This a derivation that always re-writes the rightmost non-terminal symbol first

Sentence & Sentential-form of a Language

- **Definition 1**
- Let G(Z) be a grammar. A string x is called a **sentential form** (or a sentence-like form) if x can be derived from the distinguished starting symbol Z by the application of zero or more application of the production rules of the grammar,' i.e; Z ==>*x.
- **Examples** of sentential form from grammar G(<Print-stmt>) are
  - PRINT <Expr>, PRINT <Variable>, PRINT, PRINT M4
- **But** <Expr><Variable> and <Variable><Letter> are not sentential form
  - Because they cannot be derived from <Print-stmt>, the starting symbol of G(Z)
- **Definition 2**
- A **sentence** is a sentential form containing only terminal symbols.
  - For e.g, PRINT and PRINT M4 are sentences
- **But** PRINT <Variable> and M4 are not because
  - PRINT <Variable> contains a non-terminal symbol, though it can be derived from G(Z).
  - M4 cannot be directly derived from<Print-stmt>, the starting symbol of G(Z).

Sentence & Sentential-form of a Language

- **Definition 3**
- A **language** L defined by a grammar G, written (L(G)), is the set of sentences that can be derived from V in G; i.e.
  - L(G) = {x|x **€** T and V ==>*x}
  - Where T = terminal symbols and V = starting non-terminal symbol of the grammar
- For example, the language L(<Print-stmt>) is the set of sentences of those terminal symbols (i.e, PRINT, A-Z, 0-9, comma and semi-colon) which are derivable from the distinguished starting symbol <Print-stmt>.
- Examples are:
  - PRINT
  - PRINT A
  - PRINT M4
  - etc

The Language of a Given Grammar

- **The Language of a Given Grammar**
- Consider the G2(S) which is:
  - S - -> AB
  - A - -> aA
  - A - -> a
  - B - -> Bb | b
- The language produced by this grammar consists of all strings from a string of a’s followed by a string of b’s
- **How?**
- The language is obtained by first re-writing some of the production rules of the grammar until some sentences are generated. For the generated sentences, some pattern will be formed that can lead to a generalization of the language described by the grammar.

The Language of a Given Grammar

- **The Language of a Given Grammar Contd**
- **How?**
- The language is obtained by first re-writing some of the production rules of the grammar until some sentences are generated. For the generated sentences, some pattern will be formed that can lead to a generalization of the language described by the grammar.
- For example, from G2;
- S = = >AB = = > aB = = > ab (By applying rules 1, 3 & 5 in this order)
- S = = > AB = = > aAB = = > aaB = = > aab (Applying rules 1, 2, 3, & 5 in this order)
- S = = > AB = = > aAB = = > aaB = = > aaBb = = > aabb (Applying rules 1, 2, 3, 4. & 5 in thisorder)
- S = = > AB = = > aAB = = > aaB = = > aaBb = = > aaBbb = = > aabbb (applying rules 1, 2, 3, 4, 4, & 5 in this order)

- The language described by G2 can be written as :
  - {(a!n)(b!m), where n=1,2,3,…; m=1,2,3..} or
  - {$\mathbf{a}^{n}$ $\mathbf{b}^{m}$, n= 1, 2, 3…; m=1, 2, 3..}

Classification of Grammar- by Chomsky Noam (1950)

- **Type 0 or Phrase structure grammar**
  - α ---> β; α, β ∈VUT
  - E.g;
  - (i) aAbbaAB ---> abAB
  - (ii) S ---> QNQ
  - QN ---> QR
  - RN ---> NNR
  - Anything goes, no restriction

- **Type 1 or context-sensitive**
  - α --->B, | α | <= | β | where | α | means length or no. of symbols in a
  - In addition, d LHS cannot have more than 1 non-terminal symbol
  - e.g (i) bB ---> Bd
  - (ii) aAa ---> aaBa

  - All the rules of grammar must satisfy the conditions specified

Classification of Grammar- by Chomsky Noam (1950) Contd

- **Type 2 or context-free**
  - In addition to type 1 above, the LHS can only contain one non-terminal & no terminal symbol at all; the RHS may have more than one non-terminal symbol
  - Most PLs are defined by type 2 grammar; e.g the grammar of the Print-Stmt
- **Type 3 or Regular grammar**
  - Both LHS and RHS of a grammar must not have more than non-terminal symbol. Of course no terminal symbol at all on LHS of production
  - In general, a regular grammar may be either of the form: A --->a, A ---> Ba or A ---> aB
- It can be shown that all type 3 grammars are type 2 grammars; and all type 2 are type 1 and all type 1 are type 0 grammars
- Corresponding to hierarchy of grammars, are also hierarchy of language
  - Type 0 grammar produces type 0 languages; etc

Urgent Assignment

- 1. Explain the following terms: (i) meta language (ii) syntax (iii) BNF (iv) formal grammar
- 2. Given the following grammar:

  <Print-Stmt> --->PRINT | PRINT<Expr>
  <Expr> ---><Variable>|<Variable> , | <Variable> ;
  <variable> ---> <letter>| <Letter><Digit>
  <Letter> ---> A|B|C|…|Z
  <Digit> ---> 0|1|…|9

  i) What is the starting symbol? (ii) how many rules are in the production?
  ii) (iii) show that PRINT M4; is a string/sentence of the grammar
  iii) What type/class of grammar is the given grammar?
  iv) How will you know the the given grammar is not type 1?
- 3.what language is described by each of the following grammar?:
- i. S---> A; A ---> aAb|ab
- ii. <Num> ---> <No><Ldigit>|<Ldigit>
  - <No> ---> <No><<Digit>|<Digit>
  - <Digit> --->0|2|4|6|8|<Ldigit>
  - <Ldigit> ---> 1|3|5|7|9

  - Due date Tuesday 13th October, 2020; 3pm Latest!

Formal Grammar Contd

CSC 310: Compiler Construction I

By

Olatunji, E.K.

Computer Sc Programme
FCAS
Thomas Adewumi University, Oko Iwo

April 2024

Outline of the Lecture

- ?Curriculum
- Reference Materials -Recommended
- Introduction??
- Table Management I
- Intermediate Forms of Source program
- Code Optimization

Recommended Reference materials

- 1. Compiler Techniques (An Introductory Text on Concepts and Principles) by E. K. Olatunji
- 2. Principles of Compiler Design by Aho & Ullmam
- 3. Compiler Construction for Digital Computers by David Gries
- 4. Computer Science by C.S. French, @ BookPower, 5th edition
- 5. Compilers: principles, Techniques and tools by Aho, Lam, Sethi & Ullmam @ 2007
- 6. Online Resources

Section 1: Introduction

- Recall that the compilation process consists of well-defined tasks; viz:
  - Lexical Analysis
  - Syntax analysis
  - Semantic analysis
  - Code optimization – machine - independent
  - Storage allocation
  - Code Generation
  - Code optimization II – Machine -dependent
  - Info Table management
  - Error Handling

Section 1: Introduction Contd

- These compilation tasks may be grouped into a number of phases depending on the whims and caprices of the designer/implementor
- A **phase** of compilation usually involves performance of one or more of the well-defined tasks
- Examples:
  - 2-phases: Analysis (Lexical, syntax &semantic) and Synthesis(Code generation and code optimization)
  - 3-Phases:Lexical analysis, syntax & semantic analysis and Code generation
  - 4-phases: Lexical Analysis, Syntax & Semantic Analysis, Preparation for code generation (Storage assignment , machine –independent optimization), and Code generation

Section I: Introduction Contd

- **Passes of a compiler**
  - A **pass** in compilation is any step (or group of steps) of compilation which involves reading the entire source program or its modified version
  - There are single and multi-pass compilation with their pros and cons!
  - Examples of single pass
    - WATFOR implemented for processing Fortran IV program
    - Turbo Pascal
    - BLISS /II Compiler – a system programming language
  - Examples of Multi-pass
    - Early C compilers (2-pass)
    - IBM 360 Fortran IV H compiler (4 passes)

Section 2: Table Management I

- A no of tables are created, maintained and manipulated during the process of Compilation. These include:
- Table of Terminal symbols
  - These are reserved words, operators, delimiters
  - E.g IF, READ, +, *
- Symbol Table
  - This keeps info about all identifiers in the source program together with the attributes of each
  - Examples of attributes are data type, storage address,
- Constant or Literal Table
  - This keeps tract of all literals/ constants in the program along with their attributes
  - Separate table may be kept for string literals

Table of Terminal Symbols

- Terminal symbols are the reserved word, operators and delimiters (comma, parentheses, etc)
- There is an entry for each terminal symbol
- An entry in the table consists of a symbol and its attributes, viz:
  - Internal representation
  - category(reserved word, operator, etc)
- It is a permanent structure within the compiler

Table of Terminal Symbols Contd

- It is accessed by virtually all the phases of compilation:
- *Scanner* – on finding a token, accesses the table if the token belongs to any of the terminal symbols, otherwise checks other tables to know the token category
- *Code generator* – uses the machine code attribute of a reserved word or an operator to generate code

Table of Terminal Symbols Contd

- Sample structure is as shown below

- Table 1: Table of Terminal Symbols

Terminal symbol	Internal Repr	Symbol category	Other attributes
IF	05	KW	
DO	06	KW	
*	28	OP	
,	09	DL	

Symbol Table

- The table consists of all identifiers in the source program along with their attributes
- Attributes of identifier include:
  - Data type (integer, Boolean, etc)
  - Category (constant, simple variable, subscripted variable, subroutine name, etc))
  - Block number for structured languages
  - Storage address
  - etc
- There is only one entry for each identifier
- The table is usually created at the syntax analysis phase
- Usually, pters to (or address of) the identifiers are stored in the table, while the actual identifiers along with their lengths are stored in a special string list
- Table 2 is a sample structure of a symbol table; Table 3 is its condensed form containing 3 identifiers, Table 4 shows its modified form

Symbol Table Contd

- Table 2: Sample structure of a Symbol Table

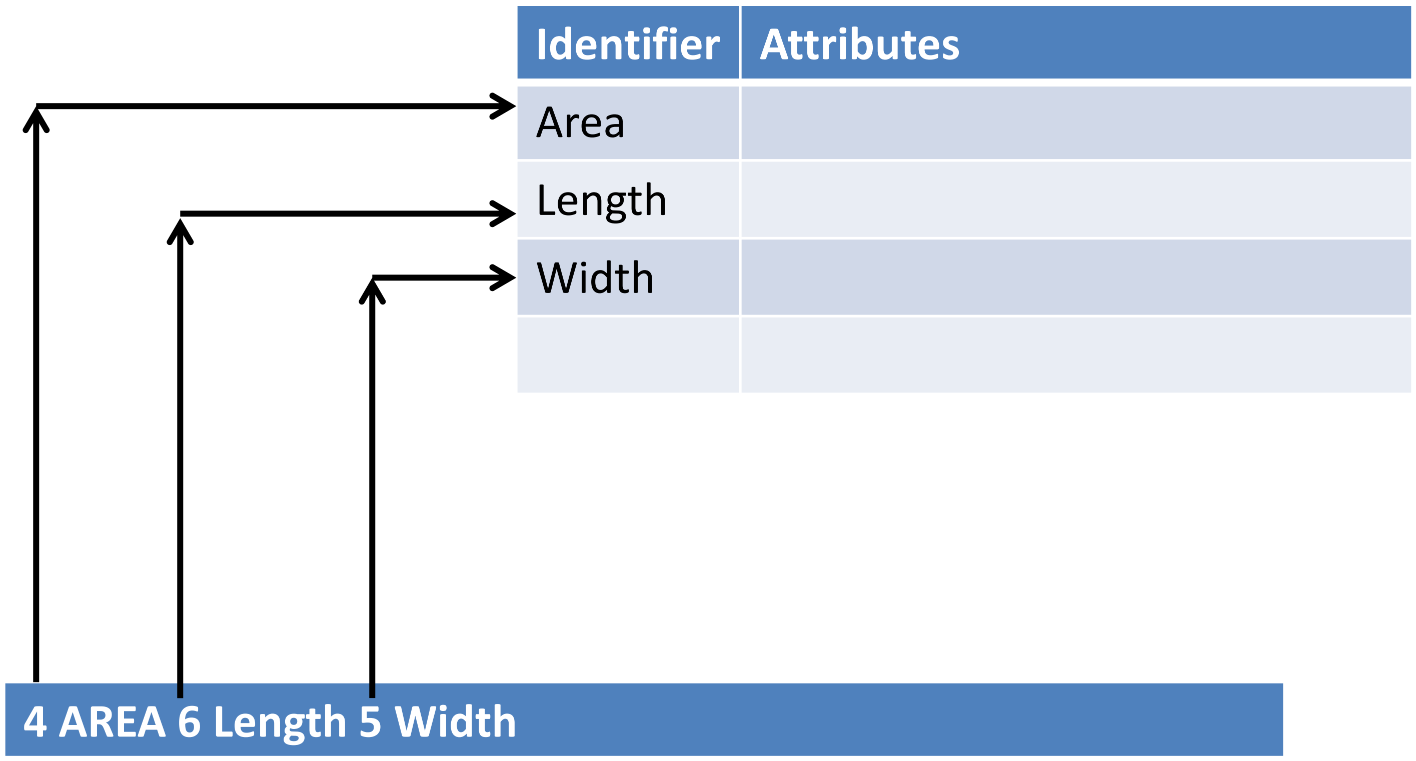
Identifier	Category	Type	Other attributes	Storage Address

- Table 3: Symbol table with 3 identifiers.

Entry No	Identifier	Attributes
1.	Area	
2.	Length	
3.	Width	
4.		

Symbol Table Contd

- Table 4: Symbol Table with 3 identifiers
-



Identifier	Attributes
Area	
Length	
Width	

4 AREA 6 Length 5 Width

Symbol Table Contd

- The semantic routine enters most of the attributes
- Storage allocation fills in the storage address
- Code Generator uses most of the info in the table to generate code
- The whole translation process revolves around the symbol table (explain!).
- Thus the table must be structured in a way to allow for efficient access for retrieval or storage.

Literal Table

- Quiz. Describe the content and structure of a literal table

Quiz on Table management

- 1. Describe the contents of 3 important tables of info kept or created by a compiler
- 2.Write brief notes on the symbol table and its importance during compilation
- 3. describe

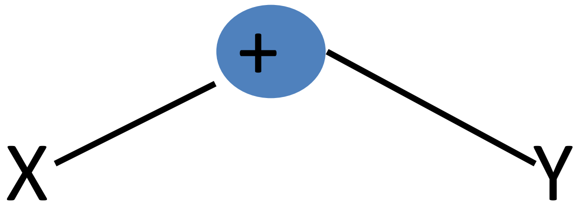
Section 3: Intermediate Forms of Source program

- More commonly, after the analysis phase, compiler usually create intermediate form of the source program (IFSP) b4 the actual code generation
- Purpose:
  - To facilitate code optimization (machine-independent)
  - To provide a clear separation between the machine-independent & machine dependent phases of compilation
- In most IFSPs, operator appear in the order in which they are to be executed
- A combination of IFSPs are used rather than using only one exclusively
- Common IFSPs are the parse tree, polish notation, 3-Address code, Quadruples, Triples

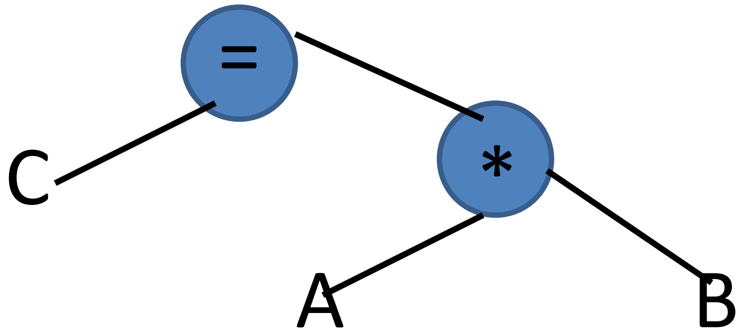
Parse Tree

- Parse Tree – also called syntax tree
- More commonly used for arithmetic expressions and /or statement
- In parse tree, every variable or constant is a terminal node
- Each operator forms a node, usually having left and right branches, and constructed in the order dictated by rules of algebra
- Figure 1 is a parse tree for the expression: X+Y
- Figure 2 is a parse tree for arith statement C=A*B

Parse Tree



- Figure 1: Parse Tree for X+Y



- Figure 2: Parse tree for C=A*B

Matrix Notation

- More practicable than Parse tree
- It is a linear list in the order the operations are to be carried out
- Table 5 is a matrix notation for the statement: Y=X+W*Z

Matrix Entry number	Operator	OPerand1	Operand2
1	*	W	Z
2	+	X	M1
3	=	Y	M2

- Table 5: Matrix for a Sample Arithmetic Statement

Polish Notation

- In this notation, the operator comes immediately after its 2 operands
- The order of evaluating the operators are kept
- Also called suffix or postfix notation
- Table 6 shows sample infix expressions with their polish notation equivalent
- Figure 6: Sample Polish notation expressions

	Infix Notation	Polish Notation
I	X+Y	XY+
Ii	X*Y+Z	XY*Z+
Iii	X+Y*Z	XYZ*+
iv	W=X*Y+Z	WXY*Z+=

List of Quadruples

- It is of the form:

  (operator, Operand1, Operand2, Result)
- Has been used by PL/1, being an optimizing compiler
- Example 1: A*B would be represented by

  *, A,B,T
- Where T is a temporary variable to which result is assigned
- Example 2: Assignment operation X=Y in this notation would be

  =,Y, ,X

bcos the result of operation is stored in X.

- Quadruple for W=X+Y*Z consists of the following:

  *,Y,Z,R1

  +,X,T1,R2

  =,R2, ,W
- R1 & r2 are temporary variables
- Note: The order of carrying out the operations are preserved

Triples

- Overcomes the many temporary variables usually generated by Quadruples
- Also takes less space than Quadruples
- It is of the form: <operator>,<operand1>,<Operand2>
- Example: A+B*C in Triples would be

  (1)*,B,C

  (2)+,A,(1)

Three-Address Code

- It is of the form X=Y op Z
- Where X, Y & z are either programmer-defined variable, constants or compiler-generated temporary variables; OP stands for operation, which can be arith or logical operation
- Example 1: A+B represented as

  T=A+B
- Example2: A+B*C becomes

  T1=B*C

  T2=A+T1
- Where T1 & T2 are compiler-generated temporaries
- Commonly used by languages that do extensive optimization, e.g Fortran, PL/I, COBOL

Quiz on ‘IFSP’

- What are the purposes of IFSP of source program?
- Give 5 different IFSPs for the expression:

  a-b(c/d+e/f)

Section 4: Code Optimization

- This is elimination / removal of semantically redundant codes
- For the purpose of improving the codes' efficiency (run-time performance & storage requirements)
- Consider the statement A=B*C+D:
- This can be represented by a list of quadruples as:

```
*,B,C,T1
+,T1,D,T2
=,T2, ,A
```

- Sample symbolic code generated for each quadruple on an hypothetical computer is as follows:

Code Optimization Contd

- Sample symbolic code generated for each quadruple on an hypothetical computer is as follows:

```
LDA b   ⎫
MUL C   ⎬   for *, B,C,T1
STA T1  ⎭
LDA T1  ⎫
ADD D   ⎬   for +,T1,D,T2
STA T2  ⎭
LDA T2  ⎫   for  =,T2, ,A
STA A   ⎭
```

Code Optimization Contd

- Removing the 3rd & 4th codes will not have negative effect on the result of the statement; rather, it leads to a more efficient code (less time & less storage)
- The 6th & 7th codes are similarly semantically redundant
- A more efficient code for the statement will be:

```
LDA B
MUL C
ADD D
STA A
```

There are 2 types:

Machine-independent &

Machine-dependent

Machine-Independent Optimization

- Compile-time computation/Constant folding
  - This is performing operations whose operands are known at compile time
  - E.g; A=7*22/7*R**2 can be come 22*R**2
- Moving invariant computation outside the loop
  - Consider the loop segment:

```
k=0;
for (i=1; I <= 100; i++)
{
    k=k+i;
    x1=5;
}
```

  - Moving the statement x1=5 from within the loop will not – negatively impact the result

Machine-Independent Optimization

- Elimination of common sub-expression
  - Consider the statement: Y(K+1) = X(K+1)
  - The expression K+1 is common to both LHS and RHS
  - The statement can be reduced to
    - J=K+1
    - Y(J)=X(J)
- Boolean expression optimization
  - Assignment: Look for explanation &example

Machine-dependent Optimization

- In this optimization, features of the actual machine are exploited; such as:
  - Better use of available registers
  - Better use of the instruction set
- The method includes:
  - Elimination of consecutive store & Load instruction on the same operand
  - Use of fast and shorter machine instruction, where possible; eg:
    - Using ADD Register instead of ADD storage

Quiz on Code Optimization

- What is code optimization? What is its purpose?
- List and explain, with an example, 2 methods of machine-independent and machine-dependent code optimization
- Give / Generate an optimized / optimal code for the HLL statement: W=X*Y-Z