Lecture 03: Translation Issues

This lecture contains:

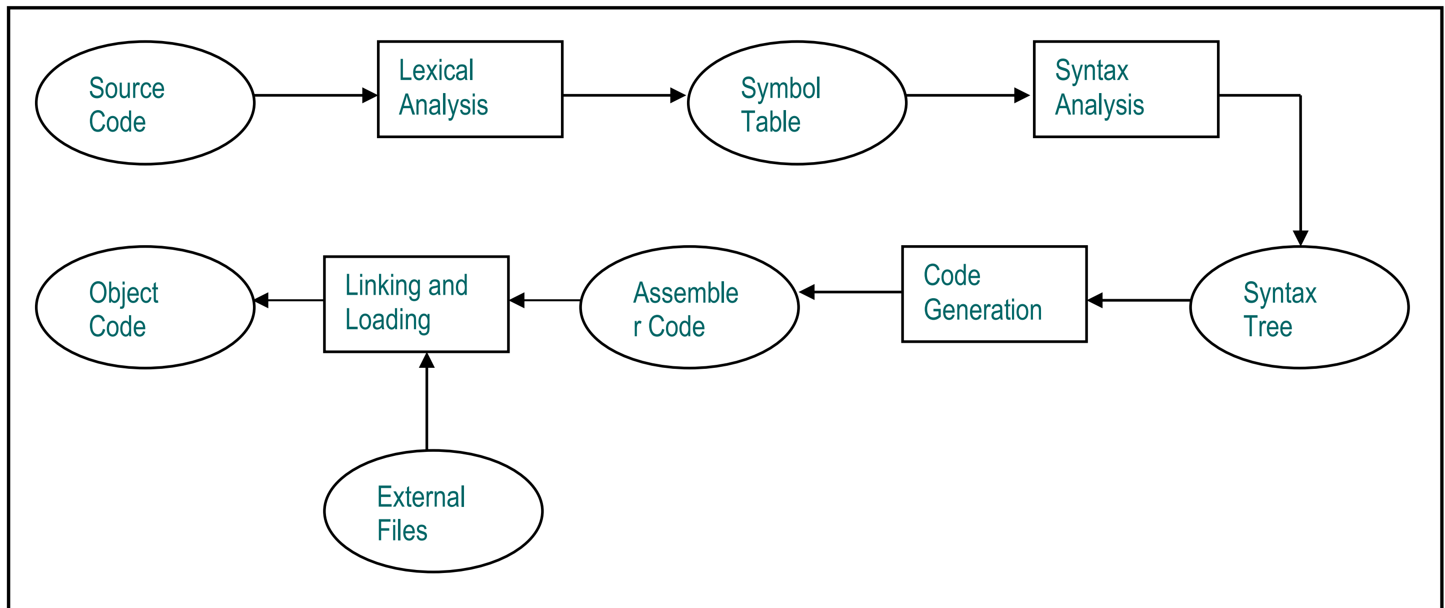
- Introduction
- Overview of Syntax
- Overview of Program Translation
- Translation Models
- BNF Notation for Syntax
- Lexical Analysis
- Syntax Analysis
- Summary and Concluding Remarks

Copyright © 2000 – 2018 by Elvis C. Foster
All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, or otherwise, without prior written permission of the author.

3.1 Introduction

The compilation process was introduced in lecture 1. For ease of reference, figure 1-4 is repeated as figure 3-1, outlining the main aspects of the translation (compilation) process.

Figure 3-1: Illustrating the Translation Process for a Typical High Level Language Program



As can be seen from the figure, the translation problem involves conversion of instructions written in a high-level language (HLL) program to machine readable instructions for implementation. The program passes through three major states: lexical analysis, syntax analysis, and code generation.

3.2 Overview of Syntax

Syntax is the arrangement of words and symbols to define and depict relationships. In computer programming, syntax serves the following purposes:

- We use syntax to define the sequence of symbols that constitute valid programs.
- The syntax provides a means of communication between programmers and the language processor.
- Syntax also facilitates communication among programmers.

Syntactic elements of a language are called *lexemes*. These lexemes fall into categories called *tokens*. Some tokens have just one possible lexeme (for example, each operator has one symbol). Others may have several possible lexemes (an *identifier* is any valid variable specified by the programmer).

3.2.1 Syntactic Criteria

The criteria for evaluating syntax were discussed in lecture 1. They include the following:

- Readability
- Support for abstraction
- Simplicity

3.2.1 Syntactic Criteria (continued)

- Orthogonality
- Ease of translation
- Support for verification
- Lack of ambiguity
- Type checking
- Control structures,
- Input/output processing
- Programming environment
- Usage cost
- Exception handling
- Functionality
- Flexibility

3.2.2 Syntactic Elements

The following are the basic components that are typically featured in the syntax of a programming language:

- **Character Set:** What are the characters (also called *terminal symbols*) supported by the language:
- **Data Types:** What are the primitive data types supported by the language? What are the advanced (programmer-defined) data types supported?
- **Identifiers:** How are identifiers defined?
- **Operator Symbols:** What operators are supported for arithmetic and logical constructions? What is the precedence of these operators?
- **Keywords and Reserve Words:** What are the keywords and/or reserve words? Ideally, the list should not be too long as this will affect how easy it is to learn the language.
- **Noise Words:** These are optional keywords.
- **Comments:** How are comments made?
- **Whitespace:** Are blank lines, tabs, and spaces allowed?
- **Delimiters:** What are the delimiters that are supported?
- **Free/Fixed Format:** Is the language a free-format or a fixed-format?
- **Expressions:** How are expressions constructed?
- **Statements:** What are the valid statements supported by the language?

3.2.3 Subprogram Structure

Different languages employ different structures for subprograms. Among the commonly known structures are the following:

- **Separate Subprograms:** Subprograms are written in separate files (as in Fortran).
- **Integrated Nested Subprograms:** Subprograms are part of the program file, and can be nested. Examples of languages that support this approach are Pascal, ALGOL, Fortran, JavaScript, PHP, RPG.
- **Integrated Independent Subprograms:** Subprograms are part of the program file, and cannot be nested. Examples of languages that support this approach are C, C++, Java, SmallTalk
- **Separate Data Description:** Data descriptions are separated from executable statements (as in COBOL and RPG).
- **Non-separated Subprograms:** Subprogram definitions are not separated from the main program (as in SNOBOL4).

3.2.4 Language Recognizers and Generators

In the most general terms, a language is the set of all valid statements (strings of characters) from a defined alphabet. A language recognizer is a mechanism for recognizing all valid statements (strings) of a language. Referring to figure 3-1, the syntax analysis portion of a compiler is essentially a language recognizer.

A language generator is a device that can be used to generate valid statements of a given language. Since the statement (sentence) produced is random, the usefulness of the language generator is limited. However, if controlled, the language generator can be quite useful.

There is actually a close relationship between a language recognizer and a language generator. This will become clear as we proceed.

3.3 Overview of Program Translation

Please revisit the translation process of figure 3-1. Program translation may be via a single pass, or multiple passes. Multi-pass compilers typically use two or three passes to convert the source code to optimized object code. Following is a summary of each category:

- **Single-pass Compilation:** In this approach, code analysis and code generation are done in the same phase. Examples of such languages include Pascal and Modula-2; several versions of single-phase C compilers have also been proposed for teaching purposes.
- **Two-pass Compilation:** Here, code analysis is typically done in the first phase, followed by code generation in the second phase. Alternately, phase 1 could be used for initial translation, and phase 2 for code optimization. Languages such as C and PHP use two-pass compilation.
- **Three-pass Compilation:** Two approaches to three-pass compilation are prevalent. In one case, phase 1 is used for source code analysis, pass 2 for initial code generation, and pass 3 for code optimization; the language Perl uses this approach. In the other case, phase 1 is used for source code analysis, phase 2 for the generation of an intermediate code, and phase 3 for the generation of the final optimized code; the language Java uses this approach.

3.3.1 Analysis of Source Code

Analysis of the source code consists of three steps: lexical analysis, syntax analysis, and semantic analysis.

In lexical analysis (LA), the source code is converted to a sequence of characters and terminal symbols from the character set of the language. These are called lexical items or tokens (more on this later). *Finite state machines* (discussed later) are useful language recognizers during lexical analysis. The LA process also commences the loading of the compiler's *symbol table* (to be clarified later).

3.3.1 Analysis of Source Code (continued)

Syntax analysis (also called parsing) produces the *syntax tree* — a hierarchical representation of the source code, based on the syntax rules of the language. The output of the lexical analyzer is used as input to the syntax analyzer. The main functions of the syntax analyzer are

- Maintenance of the symbol table
- Insertion of (formerly) implicit information
- Error detection

Semantic analysis examines the syntactic structures for meaningfulness. This process either produces an initial (draft) version of the object code, or prepares the program for subsequent generation of object code.

3.3.2 Construction of the Object Code

The output from the semantic analyzer is used as input to the (object) code generator. If the program includes subprograms, a final linking and loading state is required to produce the complete executable program. The code is then optimized before execution.

3.4 Translation Models

This section discusses translation models under the following subheadings:

- BNF Grammars
- Syntax Trees
- Finite State Machines
- Other Methodologies

3.4.1 Grammars

A grammar is defined completely by 3 finite sets and a start symbol. Mathematically, we may represent a grammar as follows:

G[Z] = {N, T, P, Z} where
N is the set of non-terminal symbols;
T is the set of terminal symbols;
P is the set of production rules that ultimately connect expressions with non-terminal symbols to expressions with terminal symbols;
Z is the start symbol such that Z Є N.

3.4.1 Grammars (continued)

From the above definition, the following constraints are normally applied:
- The intersection of sets N and T is the empty set, i.e., N ∩ T = { }.
- The alphabet of a grammar is comprised of all terminal and non-terminal symbols, i.e., N Ụ T = alphabet.
- The language of the grammar is the set of all acceptable strings for that grammar.

Reputed linguist Noam Chomsky describes four types of grammar. With some modification, we use them in the study of programming languages to explain these languages are developed. The languages are
- Phrase Structure Grammar
- Context Sensitive Grammar
- Context Free Grammar
- Regular Grammar

3.4.2 Phrase Structure Grammar

Phrase structure grammars (PSG) are used to describe natural languages. There are different dialects of PSG, for instance head-driven phrase structure grammar (HPSG), the lexical functional grammar (LFG), and the generalized phrase structure grammar (GPSG).

In a PSG, the productions are of the following form:

B ::= V where
B Є {N Ụ T) and B is not null ; alternately expressed as B Є {N Ụ T}+
V Є {N Ụ T) and V can be null; alternately expresses as V Є {N Ụ T}*

To put it in words, a non-null notation (including terminal and/or non-terminal symbols) on the left can be replaced by any valid combination of symbols (terminal and/or non-terminal) on the right, including the empty set.

Example 1:

Consider the PSG given by the following sets:
G[Z] = {(Z, A, B, C), (a, b, c), P, Z} where P consists of the following rules:
R1. Z ::= aZBC | aBC
R2. CB ::= BC
R3. aB ::= ab
R4. bB ::= bc
R5. bC ::= bc
R6. cC ::= cc
R7. BC ::= cC

Q1a. We may derive the string *abc* from the grammar (thus showing that it is valid) as follows:
By R1: Z → aBC By R3: aBC → abC By R5: abC → abc

Q1b. We may show that a^2bc^3 is a valid string as follows:
By R1a: Z → aZBC By R1b: aZBC → aaBCBC By R3: aaBCBC → aabCBC
By R5: aabCBC → aabcBC By R7: aabcBC → aabccC By R6: aabccC → aabccc

3.4.2 Phrase Structure Grammar (continued)

Notice from the forgoing example, that in order to determine that a string pattern or phrase is valid, it has to be derived. Each term in a derivation is called a *sentential form*. Alternately, a sentential form is a term that is derivable from the start symbol of a grammar. Formally, a language may be defined as a set of sentential forms (each consisting of only terminal symbols) that can be derived from the start symbol of the grammar.

3.4.3 Context-Sensitive Grammar

In a context-sensitive grammar (CSG), either sides of any given production rule may be surrounded by a context of a set of terminal and/or non-terminal symbol(s). Formally, we say that productions are of the following form:

aAy ::= xay where
A Є N
x, y Є {N Ụ T} or x, y Є { }; alternately expressed as x, y Є {N Ụ T}*
a Є {N Ụ T) and a is not null; alternately expresses as a Є {N Ụ T}+

To further paraphrase, a string may replace a non-terminal A in the context of x and y, where x and y represent valid sentential forms of the grammar. One application of this grammar is in programming languages that require variables to be declared prior to their usage (for example, Pascal, C, C++, Java, COBOL, etc.). Apart from this, the CSG is not widely used for programming languages.

3.4.4 Context-Free Grammar

In a context-free grammar (CFG), a non-terminal symbol, A, may be replaced by a string (i.e. sentential form) in any context. The production rules are typically used to recursively generate string patterns from a start symbol. CFGs appear in most programming languages. Formally, we say that productions are of the following form:

A ::= a where
A Є N
a Є {N Ụ T} or a Є { }; alternately expressed as a Є {N Ụ T}*

Example 2:

Referring to example 1, Rule 1 is also context-free. We may therefore define a grammar as follows:
Consider the CFG given by the following sets:
G2[Z] = {(Z, A, B, C), (a, b, c), P, Z} where P consists of the following rules:
R1. Z ::= aZBC | aBC | abc

Example 3:

We may define a grammar for numeric data as follows:
G[Number] = {(Number, Digit), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .), P, Number} where P consists of the following:
R1. Number ::= <Digit> | <Digit> <Number>
R2. Number ::= <Number>.<Number>
R3. Digit ::= 0|1|2|3|4|5|6|7|8|9

3.4.5 Regular Grammar

In a regular grammar (RG), productions are of the following format:

A ::= a | aB | Ba where
A, B Є N and a Є T

Example 4:

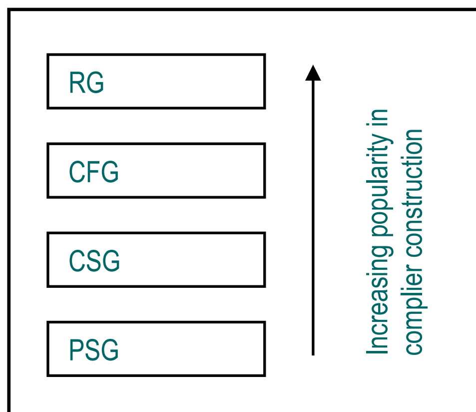
Consider the RG given by the following sets:
G3[Z] = {(Z, B), (a, b), P, Z} where P consists of the following rules:
R1. Z ::= Zb | Bb
R2. B ::= Ba | a

Q4a. We may derive the string a^3b^2 from the grammar (thus showing that it is valid) as follows:
By R1a: Z → Zb   By R1b: Zb → Bbb   By R2a: Bbb → Babb   By R2a: Babb → Baabb
By R2b: Baabb → aaabb

3.4.6 Chomsky Hierarchy

The grammars may be organized in a hierarchy (known as the Chomsky hierarchy) as shown in figure 3-2. Of the four grammars, CFG and RG are more widely used in the design of programming languages.

Figure 3-2: Chomsky Hierarchy



3.4.7 Other Notations

Two additional notations worth remembering are as follows:

- A →+ B means that B is derivable from A in one or more steps.
- A →* B means that B is derivable from A in zero or more steps.
- Let G[Z] be a grammar, and let xβy be a sentential form of G. Then β is called a *phrase* of sentential form xβy for non-terminal B if Z →* xβy and B →+ β. Moreover, β is called a *simple phrase* of sentential form xβy for non-terminal B if Z →* xβy and B → β (i.e., β is derivable from B in one step).

  In other words, if β is derivable form a non-terminal symbol in one step, and β appears as part of a sentential form, S, that is derivable in zero or more steps, then β is a simple phrase of sentential form S.

Example 5: Figure 3-3 shows an example of an RG that defines an integer.

Figure 3-3: Grammar for Integer Definition

Consider the grammar given by the following sets:
G4[<Integer>] = {(<Integer>, <Digit>), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), P, <Integer>} where P consists of the following rules:
R1. <Integer> ::= <Integer> <Digit> | <Digit>
R2. <Digit> ::= 0|1|2|3|4|5|6|7|8|9

Q5a. The BNF notation for this grammar could simply be expressed as follows:
Integer> ::= <Integer> <Digit> | <Digit>
<Digit> ::= 0|1|2|3|4|5|6|7|8|9

Q5b. We can show that <Integer>1 is a phrase as follows:
<Integer> → <Integer><Digit> → <Integer>1
So <Integer>1 is a phrase of itself for non-terminal <Integer>.

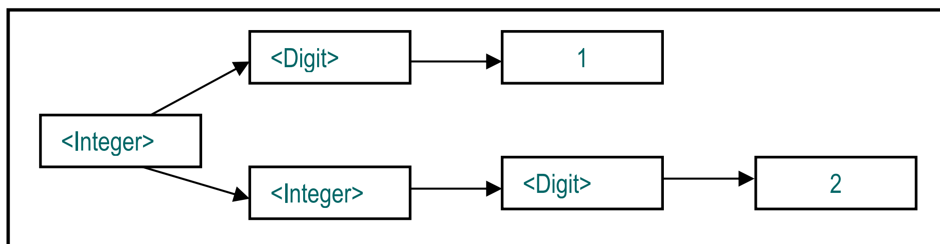
Q5c. We can also easily show that 1 is a simple phrase of sentential form <Integer>1 for non-terminal <Digit>.

3.4.8 Syntax Trees

A *syntax tree* (also called *derivation tree*) is a graphical representation used to illustrate the derivation of a sentential form from a grammar.

Example 6: Figure 3-4 shows how we may use a derivation tree to show that 21 is a valid integer of grammar in figure 3-3.

Figure 3-4: Syntax Tree to Show that 21 is Valid Based on Grammar G3 of Figure 3-3



3.4.8 Syntax Trees (continued)

Ambiguity

If two or more derivation trees exist for the same sentential form of a grammar G, then G is said to be ambiguous. The effect of ambiguity is to cause confusion: given an input, it is not known for certain which interpretation the computer will take.

Example 7:

The following grammar is used for the **If-Statement** in languages such as Pascal and Algol:

<If-Statement> ::= If <Condition> Then <Statement> [Else <Statement>]
<Statement> ::= <IfStatement> | <WhileStatement> | <ForStatement> | AssignmentStatement> | . . .
<Condition> ::= [NOT] <Comparison> | <Variable> <Operator> <Variable> | <BooleanVariable> |
<Comparison> <Connector> <Comparison>
<Connector> ::= AND | OR
<Operator> ::= < | <= | = | <> | > | >=

Now consider the pseudo-statement, and show that it is ambiguous: If C1 Then If C2 Then S1 Else S2:
The derivation trees are shown in figure 3.5.

Figure 3-5: Syntax Trees for Ambiguous If-Statement

Figure 3-5a: Ambiguity — the Else is Associated with the Second If-Statement

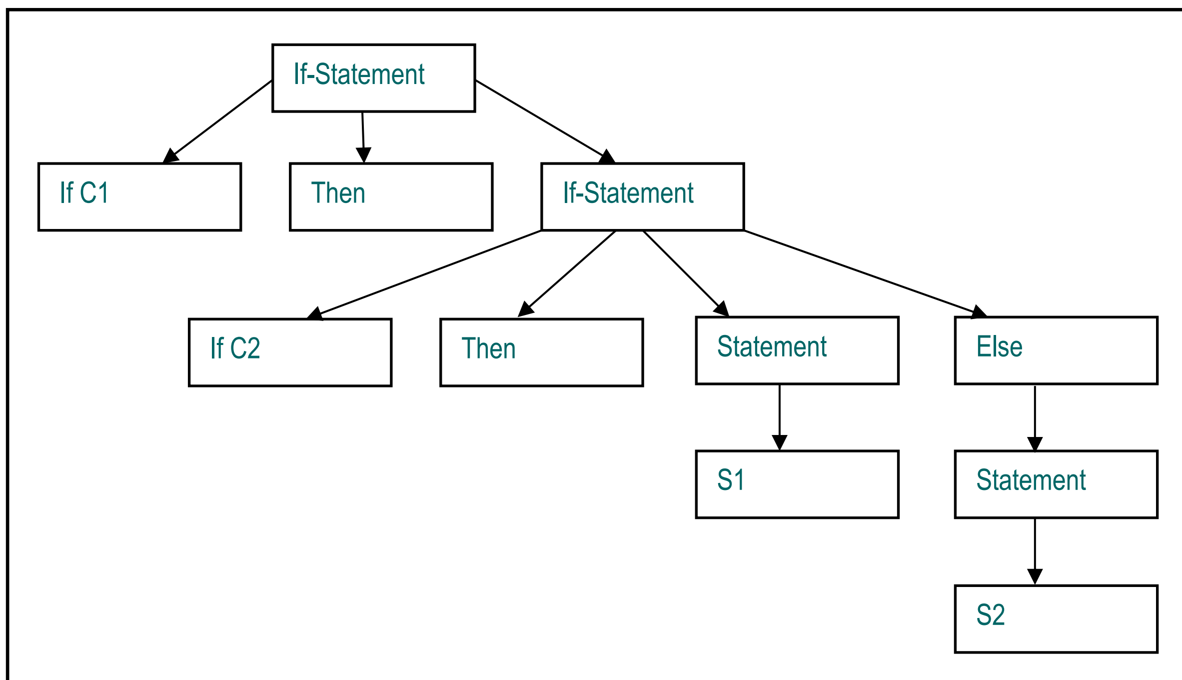
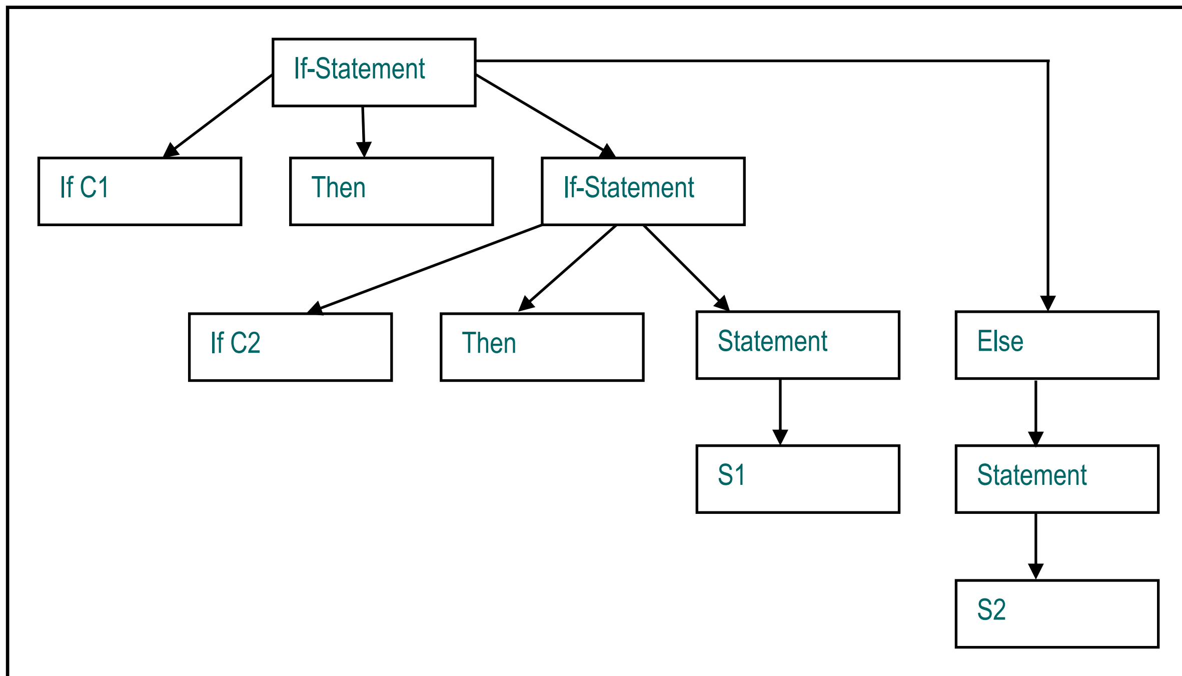


Figure 3-5b: Ambiguity — the Else is Associated with the First If-Statement



By observation, a grammar is ambiguous whenever any of the following conditions hold:

a. The grammar contains a self-embedded term, and there is a (left or right) recursion on that term
b. The grammar contains circulations of the form A →+ A

Most programming languages exhibit ambiguity in some aspect of their grammar. Of more importance is whether and how the language allows the programmer to avoid ambiguities. For instance, in many languages, you can use blocking (i.e., compound statement) to avoid ambiguity when using nested if-statements.

3.4.9 Finite State Machines

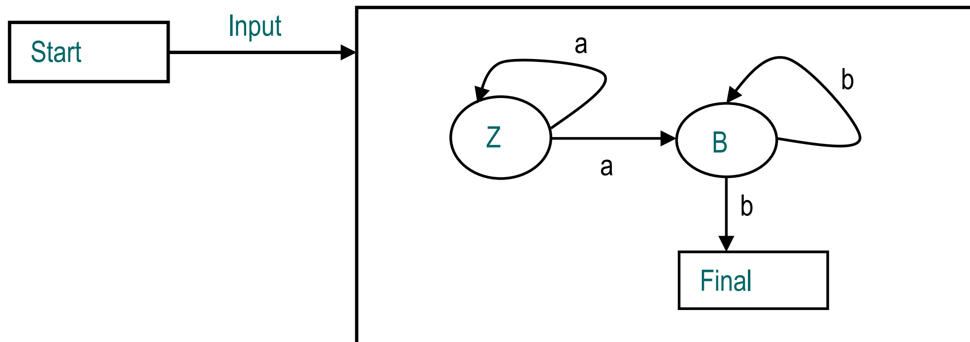
A *finite state machine* (FSM) is a graphical representation of the states and transitions among states for a software component. In an FSM (also referred to as *state diagram* or *finite state automaton*) nodes are states; arcs are transitions labeled by event names (the label on a transition arc is the event name causing the transition). The state-name is written inside the node. When used in the context of programming language design, the FSM is employed to represent the production rules of a grammar.

Example 8:

Consider the grammar given by the following sets:
G5[<Z>] = {G<Z> ::= {(Z, B), (a, b), P, Z} where P consists of the following rules:
R1. Z ::= aZ | aB
R2. B ::= bB | b

Then the language for this grammar may be expressed as follow: L<G5> = {$a^m b^n$ where m, n >= 1}.
Figure 3.6 shows the FSM for this grammar.

Figure 3-6: FSM for the Grammar of Example 8



The FSM is useful to syntax analysis in the following way: To test the validity of an input string, the final state must be reached. If a final state cannot be reached, then the input string is invalid.

A finite state machine of this form (illustrated in figure 3-6) is said to be non-deterministic. The reason for this is that it is impossible to determine which path to follow in the FSM without looking ahead. If we assume that there is no way of looking ahead to the next symbol, then only the current symbol can be considered.

Formally, we may define a *non-deterministic finite state machine* (NDFSM) as an FSM with the following properties:

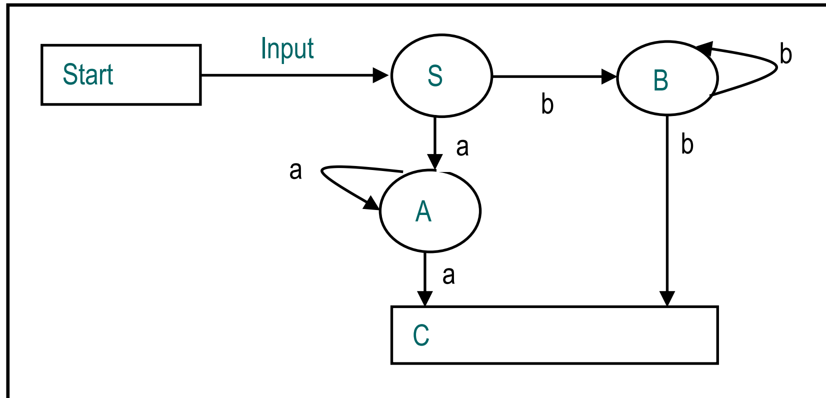
- A finite set of states (nodes)
- A finite input alphabet
- A start state (one of the nodes)
- A finite set of final states which is a subset of the set of states
- A set of transition functions (arcs) form node to node, each being labeled by an element of the input alphabet
- Given a state and an input symbol, more than one resultant states may be possible

What would be more desirable is a *deterministic finite state machine* (DFSM) — where each transition is predictable. To obtain a DFSM from an NDFSM, you represent the non-deterministic transitions as transitions to new states, as shown in figure 3-7.

Formally, we may define a deterministic finite state machine (DFSM) as an FSM with the following properties:

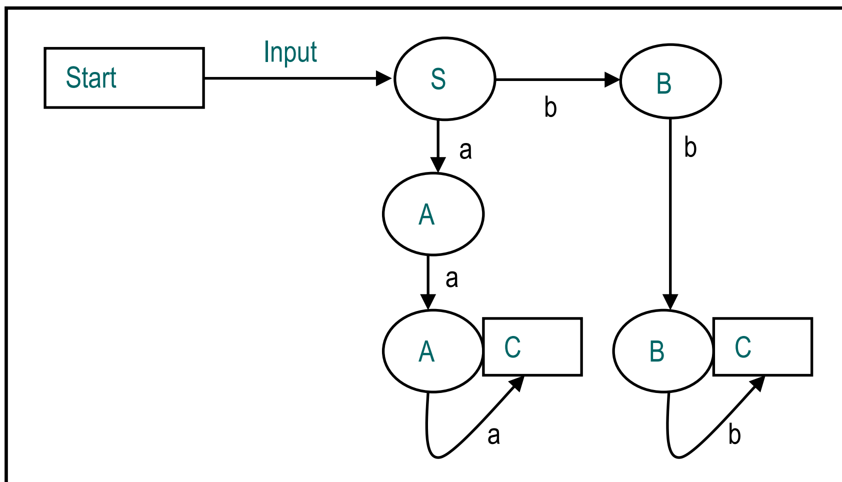
- A finite set of states (nodes)
- A finite input alphabet
- A start state (one of the nodes)
- A finite set of final states which is a subset of the set of states
- A set of transition functions (arcs) form node to node, each being labeled by an element of the input alphabet, where given a state and an input symbol, only one resultant state transition is possible

Figure 3-7: Replacing an NDFSM with a DFSM



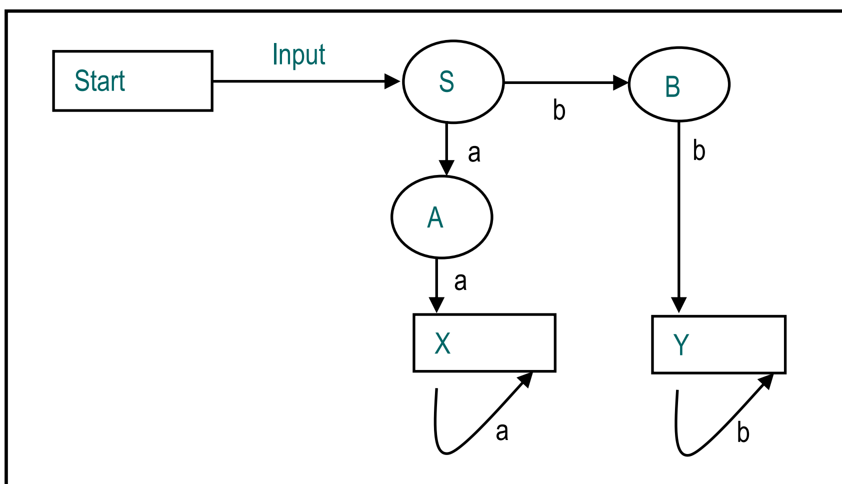
Related production rules are
S ::= aA | bB
A ::= aA | aC
B ::= bB | bC
C ::= 0

The above NDFSM can be replaced by the following DFSM:



In the hybrid state AC: If the next input is an **a**, it is pushed to state A; if the next input is a final symbol, it is pushed to C.

In the hybrid state BC: If the next input is a **b**, it is pushed to state A; if the next input is a final symbol, it is pushed to C.



Referring to the diagram above, if we replace the hybrid states as new final states, then we obtain a DFSM as shown.

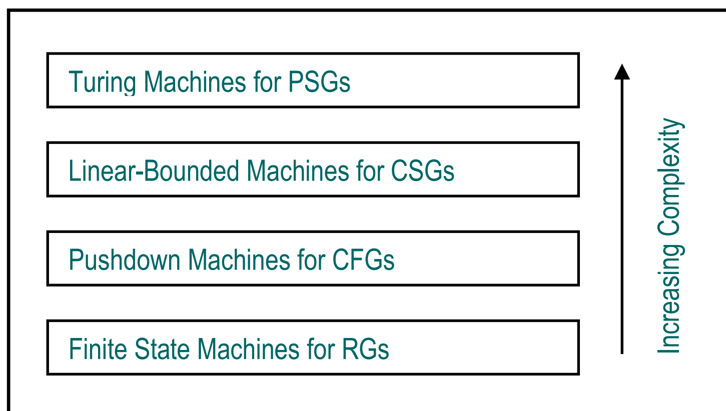
3.4.9 Finite State Machines (continued)

FSMs relate to the translation process in the following way:
- Only regular grammars (RGs) are represented by FSMs.
- Only DFSMs are useful for automatic translation. The input string must lead to a final state in a deterministic way; otherwise it is invalid.
- The DFSM may be represented internally by software.
- The DFSM is equivalent to the derivation tree, and may therefore be considered as an alternative.
- Due to all of the above, most programming languages are based on RGs.

3.4.10 Other Methodologies

Apart from FSMs, and syntax trees, other methodologies for recognizing syntax include *pushdown machines* (PM), *linear-bounded machines* (LBM), and *turing machines* (TM). Figure 3-8 provides a listing showing the relative complexity of each notation. However, knowledge of these other techniques is not required for this course.

Figure 3-8: Methodologies for Syntax Recognition



Please note:
- Turing machines ideally have infinite storage; in practice, computers qualify as Turing machines.
- A linear-bounded machine can be considered to be a Turing machine with finite storage.
- A pushdown machines can be considered to be a finite state machine with a stack.

3.5 The BNF Notation for Syntax

The BNF (Baccus-Naur Form) notation for syntax was developed by John Baccus and Peter Naur during the formative years in the development of programming languages. However, due to its profundity (despite being a simple convention), it is still widely used today. The main conventions used in the BNF are shown in figure 3-9.

Figure 3-9: BNF Notation Symbols

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td>::=</td><td>“is defined as”</td></tr>
<tr><td>[. . .]</td><td>Denotes optional content (except when used for array subscripting)</td></tr>
<tr><td><Element></td><td>Denotes that the content is supplied by the programmer and/or is non-terminal</td></tr>
<tr><td>|</td><td>Indicates choice (either or)</td></tr>
<tr><td>{<Element>}</td><td>Denotes zero or more repetitions</td></tr>
<tr><td><Element>*</td><td>Alternate notation to denote zero or more repetitions</td></tr>
<tr><td><l>*<m><Element></td><td>Denotes l to m repetitions of the specified element</td></tr>
<tr><td>[* <Element> *]</td><td>Alternate and recommended notation to denote zero or more repetitions for this course</td></tr>
</table>

Note: The construct {<Element>} is the original construct for repetition. However, C-based languages use the left curly brace ({) and right curly brace (}) as part of their syntax. To avoid confusion, it has been recommended that for these languages, the construct <l>*<m> <Element> or <Element>* be used. But that too is potentially confusing. Therefore, for this course, we will sometimes use the construct [* <Element> *] to denote zero or more repetitions. As an alternative to all of this commentary, is to use the original notation and stipulate symbols such as the left brace ({), right brace (}), left square-brace ([), or right square-brace (]) in quotations whenever they are required as part of the syntax.

Example 9:

Figure 3.10 provides the syntax for C++ variable declaration, followed by the corresponding Java syntax. You will notice that they are similar but not identical.

From figure 3.10a, you can see that the following are valid C++ declarations:
bool ExitTime;
int const LIMIT = 100;

From figure 3.10b, you can see that the following are valid Java declarations:
boolean ExitTime;
static final int LIMIT = 100;

Figure 3-10a: C++ Variable Declaration

Variable_Declaration ::= [static | register | const] <Type> <Identifier-List>;
Type ::= [short | long | unsigned | signed] int | char | float | double | enum | bool | <AdvancedType>
Identifier_List ::= <Identifier> [= <Expression>][* , <Identifier> [= <Expression>] *]
Expression ::= <ArithmeticExpression> | <BooleanExpression>
Arithmetic_Expression ::= . . . // You would need to define this
Boolean_Expression ::= . . . . // You would need to define this

Figure 3-10b: Java Variable Declaration

Variable_Declaration ::= [public | private | protected | static | final | abstract] <Type> <Identifier-List>;
Type ::= [byte | short | int | long] float | char | double | boolean| <ClassName>
Identifier_List ::= <Identifier> [= <Expression>][* , <Identifier> [= <Expression>] *]
Expression ::= <ArithmeticExpression> | <BooleanExpression>
Arithmetic_Expression ::= . . . // You would need to define this
Boolean_Expression ::= . . . . // You would need to define this

3.6 Lexical Analysis

The lexical analyzer is the front-end to the syntax analyzer. During lexical analysis, the source program is converted to a sequence of terminal symbols the character set of the language. These are called *lexical items* or *tokens*.

Example 10:

Consider the following simple Pascal program:

```
Program Add(Input, Output);
Var     a, b, Sum: Integer;
Begin
        Read (a, b);
        Sum := a + b;
        Writeln("Sum is ", Sum);
End.
```

A lexical analyzer may produce the following listing:

Program_Symbol Identifier Left_Paren Identifier Comma Identifier Right_Paren
Semicolon Var_Symbol Identifier Comma Identifier Comma Identifier
Colon Integer_Symbol Semicolon . . .
End_Symbol Period

For simplicity, regular grammars are typically used to define lexical symbols. Lexical entities directly define sequence of terminal symbols. Syntax is normally defined via regular grammars and context-free grammars. Syntactic elements directly involve sequences of lexical entities.

All symbols identified must be defined by the grammar of the language. These symbols are loaded into the *symbol table*.

3.6.1 Symbol Table

The symbol table contains critical information relating to identifiers, data items, subprograms, and other program components. Details relating to these components include related address locations as well as other execution details.

During execution, reference is not made to identifiers, but their related addresses. The symbol table is gradually built throughout the various stages of the translation process. However, its construction begins during lexical analysis.

Contents of the symbol table include the following (figure 3-11 provides an illustration):

- Name, description, address location, and accessing information for variables and constants
- Name, description, address location, and accessing information for subprograms

Figure 3-11: Example of Symbol Table Contents

Name	Object Type	Description	Address	No. Of Bytes
Add	Prog Name	Program Name	604A0	
a	Variable	Integer variable	604C1	4
b	Variable	Integer variable	604C5	4
Sum	Variable	Integer variable	604C9	4
. . .				

Each name that is encountered by the compiler/interpreter forces a reference to the symbol table. If the name is not already there it is added to the table. The table must be designed to facilitate easy searching. Alternatives include B-tree, hash table, and binary search tree.

Languages that require declaration of identifiers before they are used allow for easy and early development of the symbol table. Examples include C, C++, Java, Pascal, etc.

Languages that do not require declaration of identifiers before they are used lead to more difficult and later development of the symbol table. Examples include FoxPro, Fortran, RPG, etc.

3.6.2 Error Detection

Using a parse tree or FSM along with a symbol table, the lexical analyzer is able to detect and report a number of errors during program translation. Among the errors that can be detected and reported are the following:

- Undefined identifier
- Punctuation errors
- Recognizable comments
- Numeric overflow
- Type conflicts
- String length violations
- Incorrect use of reserve words

3.7 Syntax Analysis

Syntax analysis produces a solution to the parsing problem in the following way:

- Given a grammar and a sequence of symbols, determine whether the sequence belongs to the language of the grammar.
- If it has been determined that the input sequence belongs to the grammar's language, then recognize the structure of the sequence in terms of the production rules of the grammar.

The following are four approaches that may be employed:

- Leftmost derivation
- Rightmost derivation

} Attempt to derive the string from the start symbol.

- Leftmost reduction
- Rightmost reduction

} Start with the string and attempt to reduce to the start symbol.

3.7 Syntax Analysis (continued)

Example 11: Figure 3-12 illustrates these approaches.

Figure 3-12: Illustrating Syntax Derivation and Reduction

Consider the grammar given by the following sets:
G6[<E>] ::= {(E, T, F), (i, *, +), P, E} where P consists of the following rules:
R1. E::= E + T | T
R2. T ::= T * F | F
R3: F ::= (E) | i

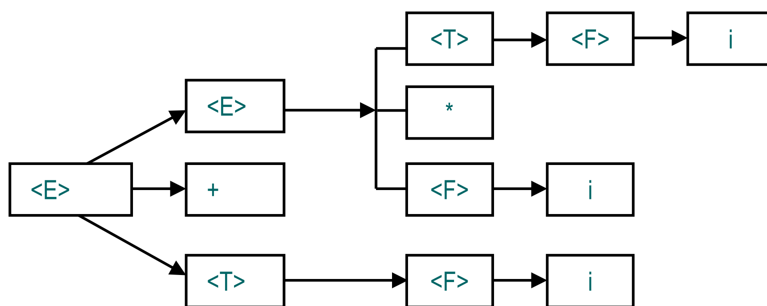
Given the above, show that ***i + i * i*** is a sentence of the grammar G6.

Leftmost Derivation:
E → E + T → T + T →F + T → i + T →i + T * F → i + F * F → i + i * i

Rightmost Derivation:
E → E + T → E + T * F → E + T * i → E + F * i →E + i * i → T + i * i →F + i * i → i + i * i

So i + i * I is a valid sentence of L<G5>.

The derivation tree is as follows:



Leftmost Reduction:
i + i * i → F + i* i →T + i * i → E + i * i → E + F * i → E + T * i → E + T * F → E + T → E

Rightmost Reduction:
i + i * i → i + i * F → i + F * F → i + T * F → i + T → F + T → T + T → E + T → E

3.7.1 Syntax Parsers

There are two types of syntax parser:

- *Top-down parsers* use the derivation approach to build a parse tree in pre-order, starting at the root.
- *Bottom-up parsers* use the reduction approach to build the parse tree in reverse order, starting at the leaves and working backwards to the root.

Figure 3-13 provides a summarized algorithm for each type of parsing. Obviously, they would need further refinement before programming, but in their current state, they should convey the essence of each approach. Note that the top-down parsing algorithm is recursive, typically requiring the first parameter (**ThisString**) to be initialized to the start symbol of the related grammar, and the second parameter (**Target String**) to be set to the input string being analyzed.

Figure 3-13: Parsing Algorithms

Top-Down Parsing Algorithm:

LeftDerivatation (ThisString, TargetString): Returns a string
```
START
  Let STARTER be the start symbol of the grammar;
  Let Left, LeftString,RightString be strings;
  If (ThisString <> TargetString)
    If (ThisString = STARTER)
        Find the appropriate production rule and replace ThisString via that rule;
    End-If; // If same as Start-Symbol

    Assign Left to the start of ThisString up the input immediately preceding the leftmost non-terminal;
    Assign RightString to the leftmost non-terminal of ThisString to the end of the string;
    LeftString := LeftDerivation(Left, TargetString);
    FinalString := LeftString + RightString;
  End-If // If same as target
  Return FinalString;
  // At the end, if FinalString <> TargetString, then there is a syntax error
STOP
```

Bottom-Up Parsing Algorithm:

LeftReduction (ThisString): Returns a string
```
START
  Let STARTER be the start symbol of the grammar;
  Let FinalString be a string;
  While (ThisString) <> STARTER)
    Replace left-most simple phrase of ThisString by its non-terminal symbol;
    Store the result in FinalString;
    Look at the next symbol in ThisString;
  End-While
  Return FinalString;
  // At the end, if FinalString <> STARTER, then there is a syntax error
STOP
```

3.7.1 Syntax Parsers (continued)

Top-down syntax analysis is more straightforward and easier to follow. Also, errors can be easily reported. However, it is more difficult to program, and not as efficient as its alternative. One common type of top-down parsing is LR (left-to-right) parsing: the input is scanned from left to right, and a right-most derivation tree is constructed in reverse.

Bottom-up syntax analysis is a bit more difficult to conceptualize, but ironically easier to program, and is more efficient than top-down parsing. One common type of bottom-up parsing is RL (right-to-left) parsing: the input is scanned from right to left, and a leftmost derivation tree is constructed in reverse.

The end-result of syntax analysis is the production of a parse tree for the program. This parse tree represents all statements included in the program.

3.7.2 Other Activities

Other activities that take place during syntax analysis include error detection and maintenance of the symbol table.

- **Error Detection and Reporting:** Errors not detected during lexical analysis are detected and reported here. These include incorrectly constructed statements, absence or incorrect use of keywords, and punctuation errors.
- **Maintenance of the Symbol Table:** Refinement of the symbol table takes place, in preparation for object-code generation. As explained earlier (in section 3.6.1), the symbol table must contain a reference for each identifier found in the program.

3.8 Summary and Concluding Remarks

Here is a summary of what has been covered in this lecture (expressed as short paragraphs):

The translation process passes through three major phases: lexical analysis, syntax analysis, and code generation. The main output from the lexical analyzer is the symbol table; the main output from the syntax analyzer is the syntax tree; the output from the code generator is the object code.

Syntax fulfills the following purposes: definition of valid programs; communication between programmer and computer; communication among programmers.

The main criteria to look for when studying or critiquing a language are readability, simplicity, orthogonality, support for abstraction, problem verification, programming environment, portability, control structures, reserve words, exception handling, input/out processing, and usage cost, lack of ambiguity, type checking, functionality, and flexibility.

Syntactic elements include the character set, data types, identifiers, operators, keywords, reserve words, noise words, comments, whitespace, delimiters, free/fixed format, expressions, and statements.

Options for sub-programming include: separate sub-programs, integrated nested sub-programs, integrated independent sub-programs, non-separated sub-programs, and separate data description.

3.8 Summary and Concluding Remarks (continued)

Program translation may occur via single-pass, two-pass, or three-pass compilation.

A grammar may be expressed via four sets: a set of start symbol(s), a set of non-terminal symbols, a set of terminal symbols, and a set of production rules that facilitate transition to terminal symbols. Valid derivations via the production rules are called sentential forms.

The Chomsky hierarchy includes four types of grammar: phrase structure grammar (PSG), context-sensitive grammar (CSG), context-free grammar (CFG), and regular grammar (RG). Of the four grammars, CFG and RG are more widely used in the design of programming languages.

A syntax tree (derivation tree) is a hierarchical structure representing the derivation of a valid sentential form from one or more related grammars.

If two or more derivation trees exist for the same sentential form of a grammar G, then G is said to be ambiguous.

A finite state machine (FSM) is a graphical representation of the states and transitions among states for a software component. Deterministic FSMs (DFSMs) are very useful in acting as language recognizers during the translation process.

The BNF notation is widely used to succinctly express the syntactic requirements of a programming language that follows the definitions of the aforementioned grammars.

A lot of the error detection takes place at the lexical analysis phase, while others take place at the syntax analysis phase.

Two approaches to parsing are derivation and reduction. Derivation attempts to derive the input string from the start symbol of the language. Reduction commences with the input string and attempts to work backwards to arrive at the start symbol of the language. Top-down syntax parsing involves the use of either left or right derivation to determine the validity of an input string. Bottom-up syntax parsing employs either left or right reduction to determine the validity of an input string.

Well, this topic is quite a handful, and there is a lot more that could have been mentioned. The intent here is to provide you with a solid overview. If this information arrests your interest, then you are encouraged to probe further by reading a text on compiler construction (see [Bornat 1989], [Holmes 1995] , and [Parsons 1992]).

3.9 Recommended Readings

[Bornat 1989] Bornat, Richard. 1989. *Understanding and Writing Compilers* 3rd Edition. London: Macmillan.

[Holmes 1995] Holmes, Jim. 1995. *Building Your Own Compiler with C++*. Upper Saddle River, NJ: Prentice Hall.

[Parsons 1992] Parsons, Thomas W. 1992. *Introduction to Compiler Construction*. New York: W.H. Freeman and Company. See chapters 2 – 8.

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapters 3 & 4.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapters 3 & 4.

[Webber 2003] Webber, Adam B. 2003. *Modern Programming Languages: A Practical Introduction*. Wilsonville, Oregon: Franklin, Beedle & Associates. See chapters 2 – 4.