Unit 3

Language Translation Issues

Programming Language Syntax

- Syntax represents the fundamental rules of a programming language we use to write functioning code.
- The syntax of a programming language is what the program looks like. To give rules of syntax for a programming language means to tell how statements, declarations, and other language constructs are written.
- Syntax provides significant information needed for understanding a program and provides much-needed information toward the translation of the source program into an object program.
- Language syntax alone is insufficient to unambiguously specify the structure of a statement.
- We also need semantics for the full description of a programming language.

Programming Language Syntax

- For example, in the statement like X = 2.45 + 3.67, syntax cannot tell us whether Variable X was declared or declared as type real. Results of X = 5, X = 6, and X = 6.12 are all possible if X and + denote integers, X denote integer and + is real addition, and X and + denote real values respectively.
- **General Syntactic Criteria:**
  - **Readability:**
    - A readable program has structure of the algorithm and data represented by the program apparent from an inspection. Readable program is often said to be self-documenting.
    - Readability is enhanced by language features such as, *natural statement formats*, *structured statements*, *liberal use of keywords* and *noise words*, *provision for embedded comments*, *unrestricted length identifiers*, *mnemonic operator symbols*, *free-filed formats*, and *complete data declarations*.
    - Languages that provide only a few different syntactic constructs in general lead to less readable programs.

Programming Language Syntax

- **Writability:**
  - Writability is enhanced by use of concise and regular syntactic structures, where as for readability a variety of more verbose constructs are helpful.
  - The use of structured statements, simple natural statement formats, mnemonic operation symbols, and unrestricted identifiers usually make program writing easier.
- **Ease of Verifiability:**
  - Related to readability and writability is the concept of program correctness or program verification. The program should be easily verifiable to be mathematically proved correct.
- **Ease of Translation:**
  - A program should be easy to translate into executable form.
  - Readability and writability are criteria directed to the needs of the human programmer. Ease of translation relates to the needs of the translator that processes the written program.
  - Programs become harder to translate as the number of special syntactic constructs increase.

Programming Language Syntax

- **Lack of Ambiguity:**
  - Ambiguity is a central problem in every language design. A language definition ideally provides a unique meaning for every syntactic construct that a programmer may write.
  - An ambiguous construction allows two or more different interpretations. The problem of ambiguity usually arise not in the structure of individual program elements but in the interplay between different structures.
  - For example, consider two different forms of conditional statements as given below:

if *Boolean expression* **then** *statement*$_1$ **else** *statement*$_2$

if *Boolean expression* **then** *statement*$_1$

  - The interpretation to be given to each statement form is clearly defined. However, when the two forms are combined by allowing statement1 to be another conditional statement, then the structure

    if *Boolean expression*$_1$ **then if** *Boolean expressin*$_2$ **then** *statement*$_1$ **else** *statement*$_2$

    termed a dangling else, is formed. This statement form is ambiguous, because it is not clear which of the two execution sequence is intended.

Programming Language Syntax

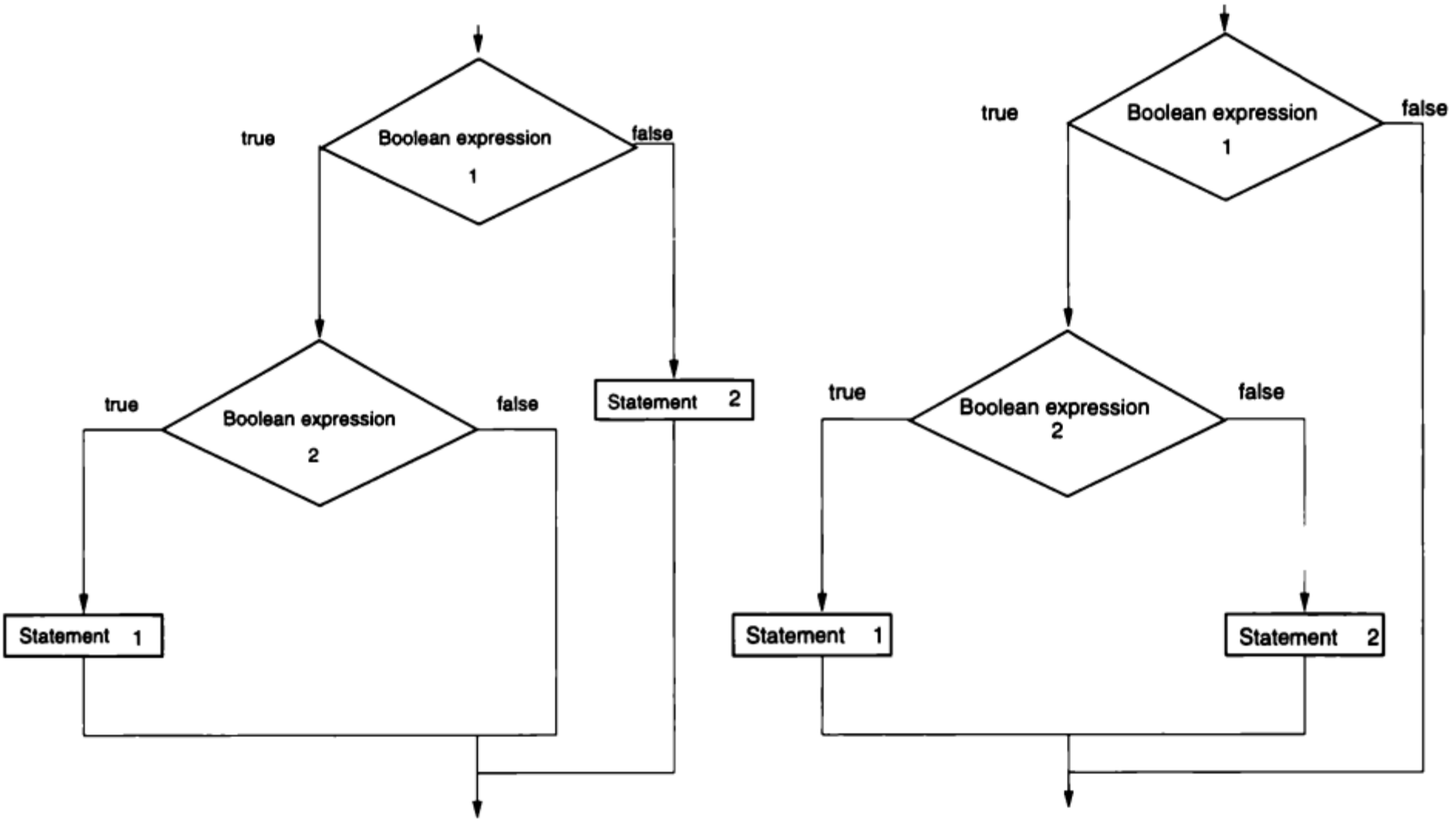


Figure Two interpretations of a conditional statement

Programming Language Syntax

- Different languages have resolved such ambiguities. For example, ALGOL uses **begin…end** delimiter pair around the embedded statement.

  if *Boolean expression*$_1$ **then begin if** *Boolean expressin*$_2$ **then** *statement*$_1$ **end else** *statement*$_2$

  if *Boolean expression*$_1$ **then begin if** *Boolean expressin*$_2$ **then** *statement*$_1$ **else** *statement*$_2$ **end**
- Ada uses **end if** delimiter with each **if** statement. In C and Pascal pair final else with the nearest then.
- The ambiguity of FORTRAN function and array references is resolved by the rule: the construct A(I, J) is assumed to be function call if not declaration for an array A is given.

Programming Language Syntax

- **Syntactic Elements of a Language:**
  - The general syntactic style of a language is set by the choice of various basic syntactic elements.
  - **Character set.** Character set is the alphabet of the language. Several different character sets are used: ASCII, EBCIDIC, Unicode.
  - **Identifiers.** Identifiers refer to name given to various program elements such as variables, functions, structures etc. The basic syntax for identifies – a string of letters and digits beginning with a letter – is widely accepted. Different languages have different length restrictions for identifiers.
  - **Operator symbols.** Most languages use special characters (such as +, -, /, *, % etc.) to represent operations. Some languages use identifiers such as PLUS, TIMES and so on to represent operations.

Programming Language Syntax

- **Keywords or reserved words.** A keyword is an identifier used as a fixed part of the syntax of a statement. A keyword is a reserved word and cannot be used as a programmer-chosen identifier. Most languages used keywords such as if, for, while, int, float, and so on.
- **Noise words.** These are optional words that are inserted in statements to improve readability. For example, in CBOBL, the **goto** statement is written as *GO TO label* and the keyword GO is required, but TO is optional.
- **Comments.** Comments are used to improve readability and for documentation purposes. A language may allow comments in several ways, such as, single line and multiline.
- **Blanks (spaces).** Rules on the use of blanks vary widely between languages. In C, for example, blanks are not significant anywhere except in literal character-string data.

Programming Language Syntax

- **Delimiters and brackets.** Delimiters are used to denote the beginning and the end of syntactic constructs. Brackets are paired delimiters.
- **Free- and fixed-field formats.** A syntax is free field if the program statements may be written anywhere on an input line without regard for positioning on the line or for breaks between lines. A fixed-field syntax utilizes the positioning on an input line to convey information.
- **Expressions.** Expressions are functions that access data objects in a program and return a value. Expressions are the basic syntactic building block from which statements (and sometimes programs) are built.
- **Statements.** Statements are the sentences of the language and describe a task to be performed. Statements can be *simple* or *nested*.

Programming Language Syntax

- **Overall Program-Subprogram Structure:**
  - How are Subprograms (Procedures, Functions) organized in a Program?
  - **Separate subprogram definition.** Each subprogram definition is treated as a separate syntactic unit. Each subprogram is compiled separately and the compiled programs are linked at load time
  - **Separate data definition.** Group together all operations that manipulate a given data object. For example, class mechanism in the languages like Java, C++ and Smalltalk.
  - **Nested subprogram definitions.** Nested program structure in which subprogram definitions appear as declarations within the main program and may contain other subprogram definitions.

Programming Language Syntax

- **Separate interface.** Separate subprograms are linked together using interfaces such as modules, packages, and header files.
- **Data descriptions separated from executable statements.** Data declarations and executable statements for all subprograms are divided into separate program *data divisions* and *procedure divisions*.
- **Unseparated subprogram definitions.** No syntactic distinction is made between main program statements and subprogram statements. A program is syntactically just a list of statements. The points where subprograms begin and end are not differentiated syntactically. For example, using GOTOs to implement subprogram concepts.

Stages in Translation

- The process of translation of a program from its original syntax into executable form is central in every programming language implementation.
- Logically, we may divide translation into two major parts: the ***analysis*** of the input source program and the ***synthesis*** of the executable object program.
- Translators are crudely grouped according to the number of *passes* they make over the source program. A simple compiler typically uses two passes. The first analysis pass decomposes the program into its constituent components and derives information from the program. The second pass typically generates an object program from this collected information.

Stages in Translation

- If compilation speed is important (such as in an educational compiler), a one-pass strategy may be employed. In this case, as the program is analyzed, it is immediately converted into object code.
- If execution speed in paramount, a three- (or more) pass compiler may be developed. The first pas analyzes the source program, the second pass rewrites the source program into a more efficient form using various well-defined optimization algorithms, and the third pass generates the object code.

Stages in Translation

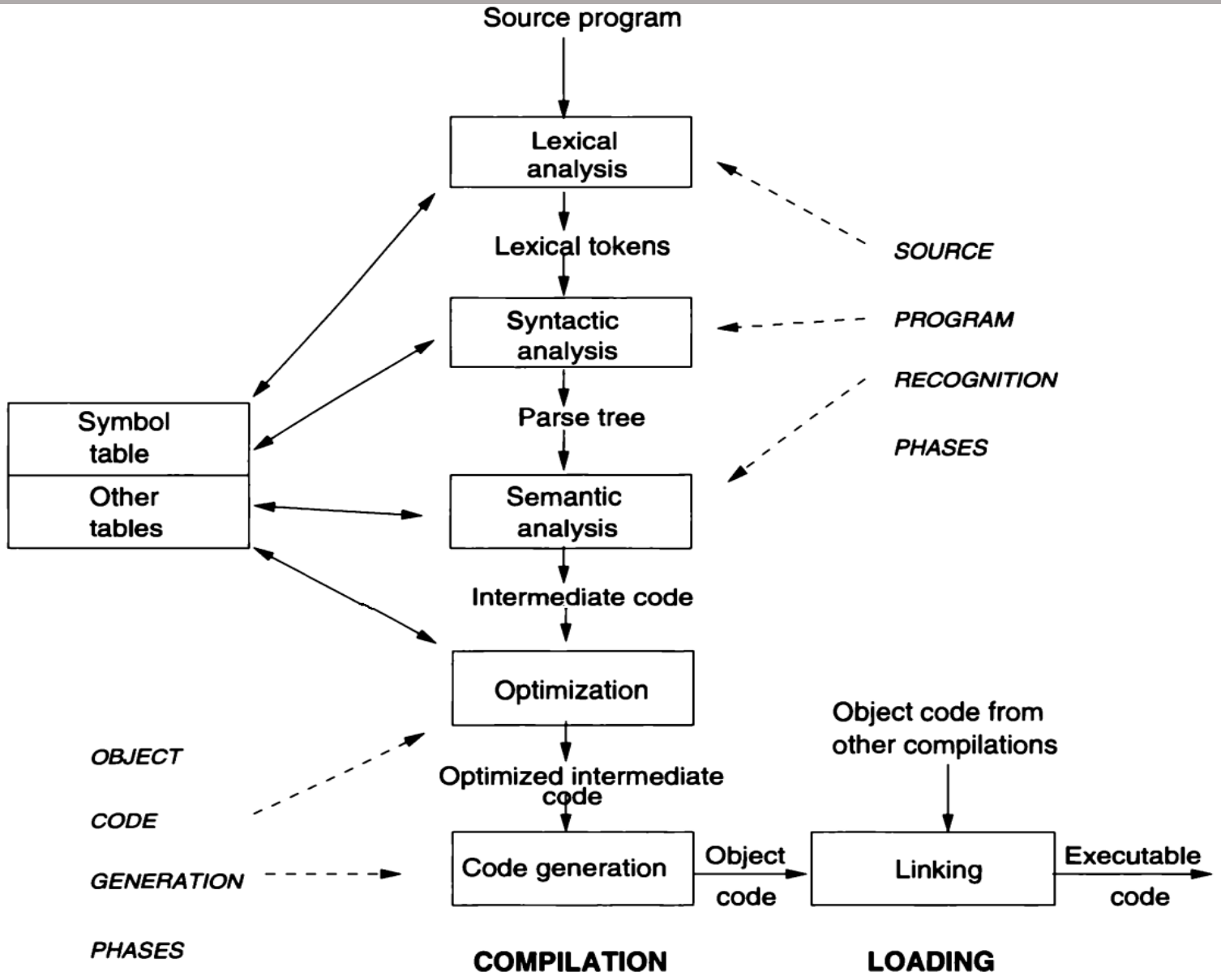


Figure Structure of a compiler

Stages in Translation

- **Analysis of the Source Program:**
  - To a translator, the source program appears initially as one long undifferentiated sequence of symbols composed of thousands or tens of thousand of characters.
  - **Lexical analysis (scanning).** This phase scans the input source code, one character at a time and identifies *lexemes*. Each lexeme corresponds to a token such as identifier, keyword, operator, and so on. The lexical analyzer then passes these tokens to the next phase in the compiler. In addition, conversion to an internal representation is often made for items such as numbers (converted to internal binary fixed- or floating-point form) and identifiers (stored in a symbol table and the address of the symbol table entry used in place of the character string). The formal model used to design lexical analyzers is the *finite-state automata*.

Stages in Translation

- **Syntactic analysis (parsing).** The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct. The syntactic analyzer enters in the stack the various elements of the syntactic unit found, and these are retrieved and processed by the semantic analyzer.
- **Semantic analysis.** Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.
  It checks type mismatches, incompatible operands, function called with improper arguments, an undeclared variable, etc.

Stages in Translation

- **Synthesis of the Object Program:**
  - Concerned with the construction of the executable program from the outputs produced by the semantic analyzer.
  - This phase involves optimization and code generation. If subprograms are translated separately, or if library subprograms are used, a final linking and loading stage is needed to produce the complete program ready for execution.
  - **Optimization.** The semantic analyzer ordinarily produces as output the executable translated program represented in some intermediate code. From this internal representation, the code generators may generate the properly formatted output object code. Before code generation, however, there is usually some optimization of the program in the internal representation.

Stages in Translation

- **Code generation.** After optimization, it must be formed into the assembly language statements, machine code, or other object program form that is to be output of the translator. The output code may be directly executable, or there may be other translation steps to follow (e.g., assembly or linking and loading).
- **Linking and loading.** In the optional final stage of translation, the pieces of code resulting from separate translations of subprograms are coalesced into the final executable program. The linking loader (or link editor) loads the various segments of translated code into memory and then uses the attached loader tables to link them together properly by filling in data and subprogram address in the code as needed. The result is the final executable program ready to be run.

Stages in Translation

- **Bootstrapping**
  - Bootstrapping is widely used in the compiler development.
  - Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
  - Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

Formal Translation Models

- The syntactic recognition parts of compiler theory are fairly standard and generally based on the context-free theory of languages.
- The formal definition of the syntax of a programming language is usually called a *grammar*.
- A grammar consists of a set of rules (called *productions*) that specify the sequences of characters (or lexical items) that form allowable programs in the language being defined.
- A formal grammar is just a grammar specified using a strictly defined notation.
- The two classes of grammars useful in compiler technology include the BNF grammar (or context-free grammar) and the regular grammar.

Formal Translation Models

- **BNF (Backus-Naur Form)Grammars:**
  - It was developed by John Backus and refined by Peter Naur as a way to express the syntactic definition of ALGOL.
  - At about the same time, a similar grammatical form, the context-free grammar (CFG), was developed by linguist Noam Chomsky.
  - These two forms are equivalent in power; the differences are only in notation. Hence, these two grammars are usually interchangeable in discussion of syntax.
  - A BNF grammar is composed of a finite set of BNF grammar rules, which define a language – in our case, a programming language.
  - The BNF is used to specify the syntactic rules of many computer languages.

Formal Translation Models

- The productions in this grammar have a single nonterminal symbol as their left-hand side.
- Instead of listing all the productions separately, we can combine all those with the same nonterminal symbol on the left-hand side into one statement.
- Instead of using the symbol $\rightarrow$ in a production, we use the symbol ::=.
- We enclose all nonterminal symbols in brackets, $\langle\ \rangle$, and we list all the right-hand sides of productions in the same statement, separating them by bars. For instance, the productions A $\rightarrow$ Aa, A $\rightarrow$ a, and A $\rightarrow$ AB can be combined into $\langle A \rangle$ ::= $\langle A \rangle$a | a | $\langle A \rangle \langle B \rangle$.

Formal Translation Models

- **Example 1:** An identifier in ALGOL 60 consists of a string of alphanumeric character and must begin with a letter. The BNF rules to describe the set of allowable identifiers:

  ⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩

  ⟨letter⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- **Example 2:** A signed integer is a nonnegative integer preceded by a plus or a minus sign. The BNF rules to describe signed integers:

  ⟨signed integer⟩ ::= +⟨integer⟩ | -⟨integer⟩

  ⟨integer⟩ ::= ⟨digit⟩ | ⟨digit⟩⟨integer⟩

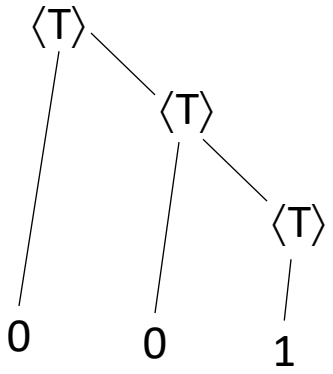
  ⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Formal Translation Models

- **Parse Tree:**
  - Any sentence which is derived using the production rules is said to be syntactically correct.
  - It is possible to check the syntax of a sentence by building a parse tree to show how the sentence is derived from the production rules.
  - If it is not possible to build such a tree then the sentence has syntax errors.
  - Consider the grammar G as given below.

⟨T⟩ ::= 0⟨T⟩ | 1⟨T⟩ | 0 | 1

The parse tree for the string 001 is as show below.

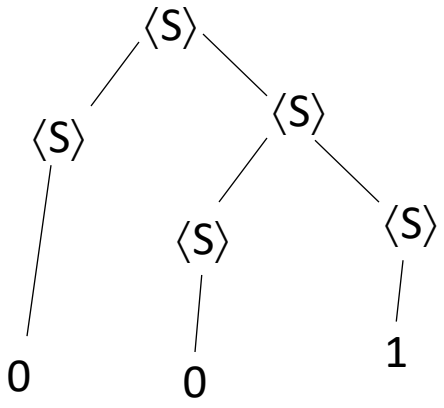


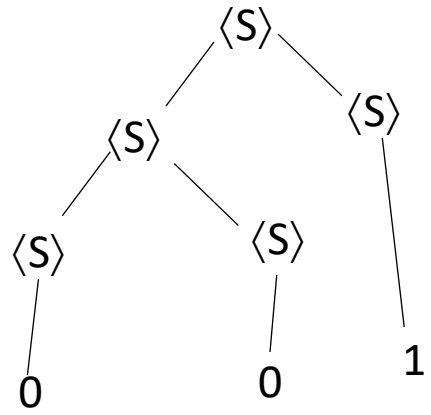
Formal Translation Models

- **Ambiguity:**
  - A grammar is ambiguous if there is some string in the language that has two distinct parse trees.
  - Consider the grammar G as given below.

⟨S⟩ ::= ⟨S⟩⟨S⟩ | 0 | 1

Two distinct parse trees of the string 001 are as show below.





Formal Translation Models

- **Extension to BNF Notation:**
  - The following notations do not change the power of the BNF grammar but allow for easier descriptions of languages:

    Alternative items are separated by a | (stroke); one item is chosen from this list of alternatives; their order is unimportant.

    The optional item is enclosed between [and] (square–brackets); the item can be either included or discarded.

    An arbitrary sequence of instances of an element may be indicated by enclosing the element in braces followed by an asterisk, {…}*.
  - **Example 1:** The BNF rules to describe the set of allowable identifiers in ALGOL 60.

⟨identifier⟩ ::= ⟨letter⟩ {⟨letter⟩ | ⟨digit⟩}*

⟨letter⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  - **Example 2:** The EBNF rules to describe signed integers.

⟨signed integer⟩ ::= [+ | -] ⟨digit⟩ {⟨digit⟩}*

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Formal Translation Models

- **Syntax Chart:**
  - A syntax chart (also called a railroad diagram because it looks like the switching yard of a railroad) is a graphical way to express extended BNF rules.
  - Each rule is represented by a path from the input on the left to the output on the right.
  - Any valid path from input to output represents a string generated by that rule.
  - Rectangles represent non terminal symbols and circles contain terminal symbols.
  - EBNF for simple assignment statements and syntax charts are given below.

⟨assignment statement⟩ ::= ⟨variable⟩ = ⟨arithmetic expression⟩

⟨arithmetic expression⟩ ::= ⟨term⟩ {[+ | -] ⟨term⟩}*

⟨term⟩ ::= ⟨primary⟩ {[× | /] ⟨primary⟩}*

⟨primary⟩ ::= ⟨variable⟩ | ⟨number⟩ | (⟨arithmetic expression⟩)

⟨variable⟩ ::= ⟨identifier⟩ | ⟨identifier⟩[⟨subscript list⟩]

⟨subscript list⟩ ::= ⟨arithmetic expression⟩ {, ⟨arithmetic expression⟩}*

Formal Translation Models

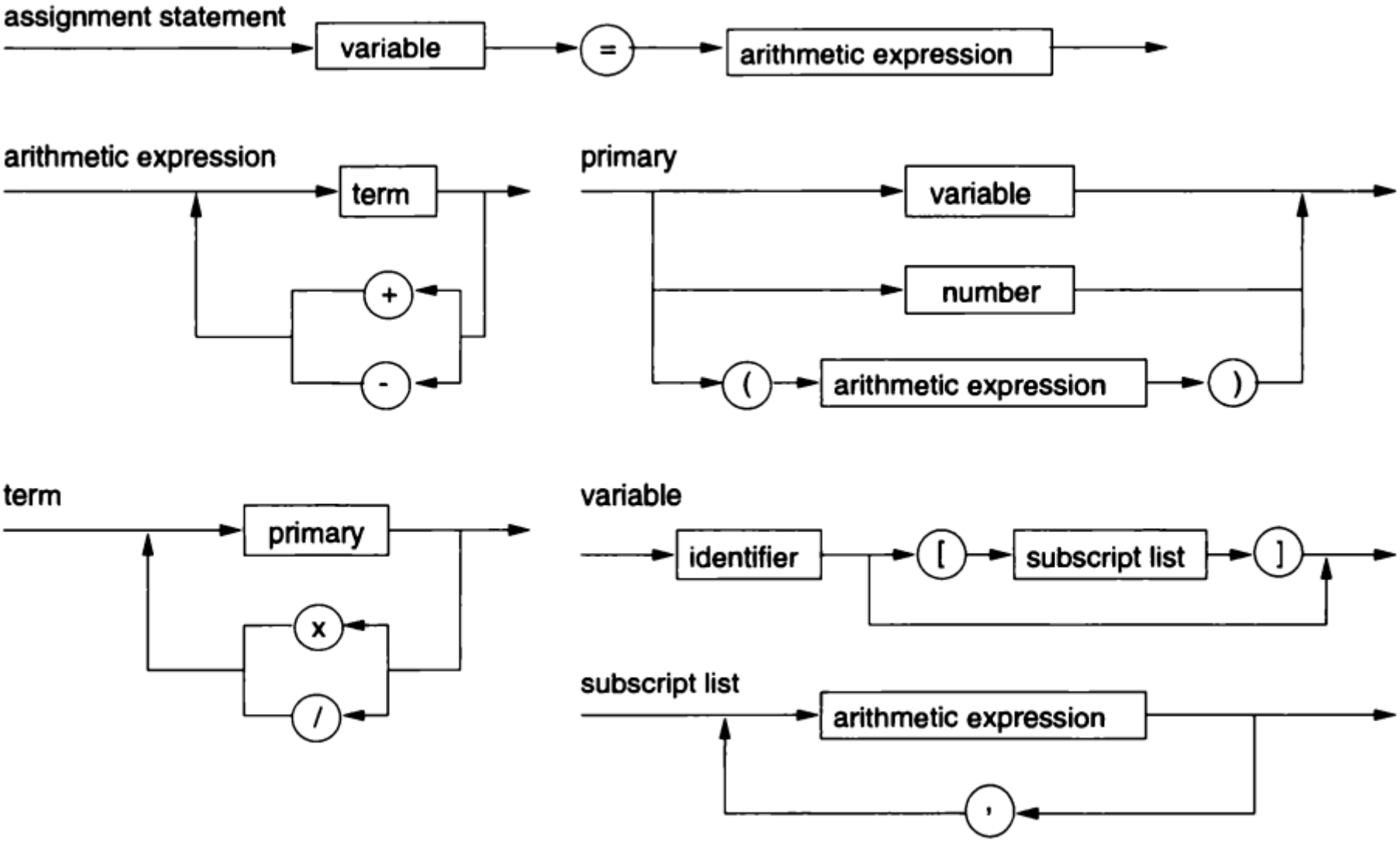


Fig: Syntax charts for simple assignment statements

Formal Translation Models

- **Finite State Automata:**
  - The lexical analysis phase of the compiler breaks down the source program into a stream of tokens.
  - There is a simple model called a ***finite-state automata*** (FSA) or a ***state machine*** that recognizes such tokens.
  - As long as we know which state we are in, we can determine whether the input we have seen is part of the token for which we are looking.
  - Whenever we are in final state (double circle), the string up to the current symbol is valid and accepted by the machine.
  - In general, an FSA has a starting state, one or more final states, and a set of transitions (labeled arcs) from one state to another. Any string that takes the machine from the initial state to a final state through a series of transitions is accepted by the machine.

Formal Translation Models

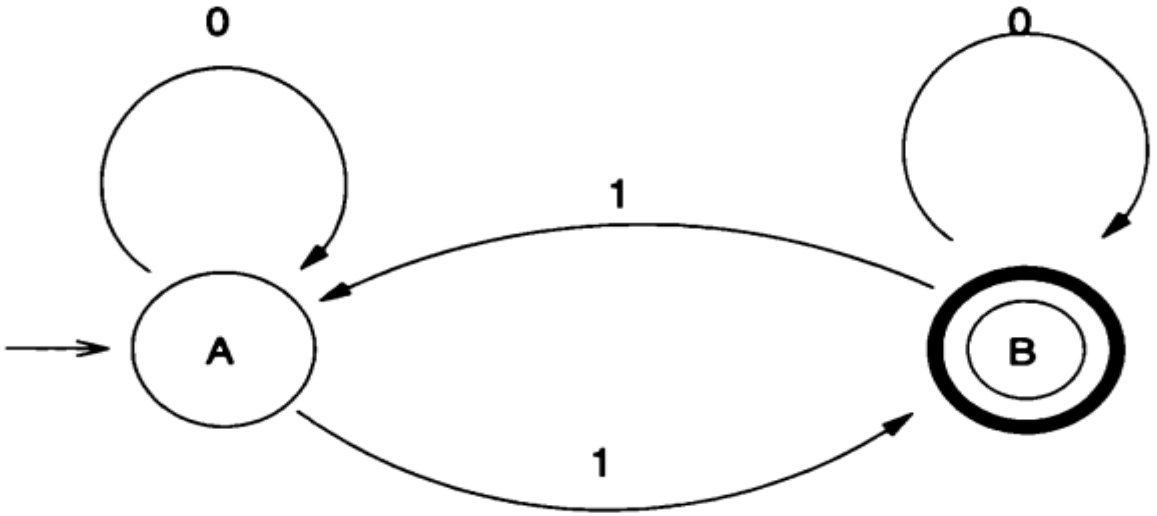


Figure FSA to recognize an odd number of 1s

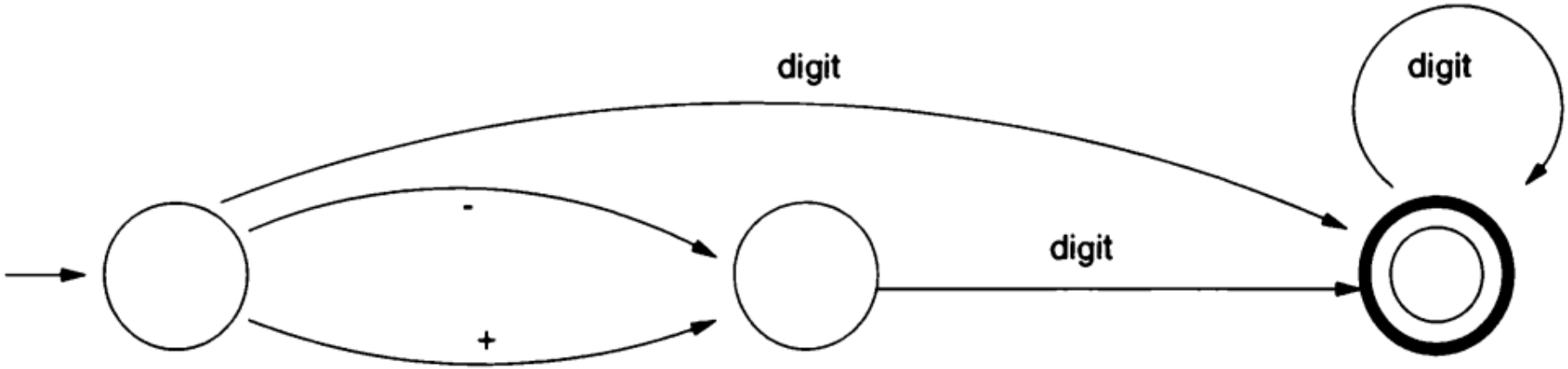


Figure FSA to recognize optionally signed integers

Formal Translation Models

- In **deterministic FSA**, there is unique transition to the same or different state. If there are n states and the input alphabet has k symbols, then the FSA will have $n \times k$ transitions (labeled arcs).
- In **nondeterministic FSA**, there are multiple arcs from a state with the same label so that you have a choice as to which way to go. From any particular node, there may be any number of (or no) arcs going to other nodes, including multiple arcs with the same label. In this case, we say that a string is accepted by the nondeterministic FSA if there is some path from the start node to one of the final nodes, although there may be other paths that do not reach a final state.

Formal Translation Models

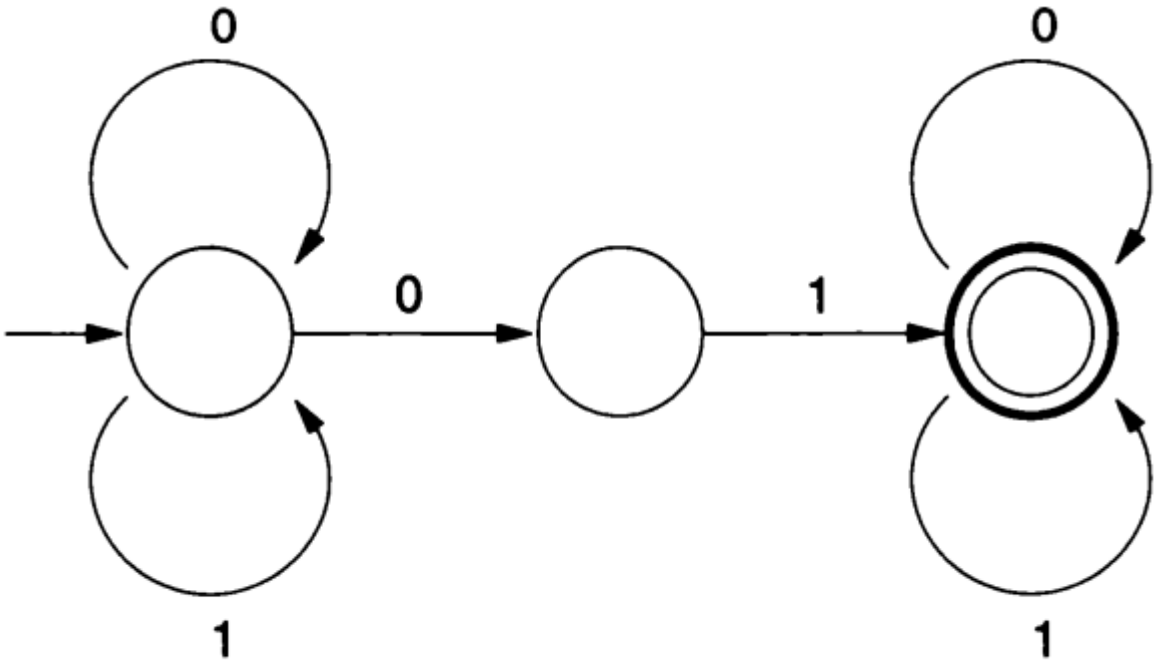


Figure: NFA to recognize binary strings containing 01

Formal Translation Models

- **Regular Grammars:**
  - Regular grammars are special cases of BNF grammars that are equivalent to FSA.
  - Regular grammars have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a non-terminal. For example, the grammar below generates binary strings ending in 0

    $A \rightarrow 0A \mid 1A \mid 0$

- **Regular Expressions:**
  - Regular expressions present a third form of language definition that is equivalent to the FSA and regular grammar.
  - We define a regular expression recursively as follows:
    1. Individual terminal symbols are regular expressions.
    2. If a and b are regular expressions, then so are: $a \vee b$, ab, (a), and a*.
    3. Nothing else is a regular expression.

Formal Translation Models

- Here, **ab** represents concatenation or sequencing of regular expressions a and b, **a** $\vee$ **b** represents the alteration of a or b, **a*** is called the Kleene closure of regular expression a and represents zero or more repetitions of a.
- We can use regular expressions to represent any language defined by a regular grammar or FSA.
- For example, letter(letter $\vee$ digit)* is the regular expression for identifiers, (0 $\vee$ 1)*0 is the regular expression for binary strings whose last symbol is 0, and (0 $\vee$ 1)*01 (0 $\vee$ 1)* is the regular expression for binary string containing 01.
- Converting any regular expression to an FSA is fairly easy as shown in the figure on the next slide.

Formal Translation Models

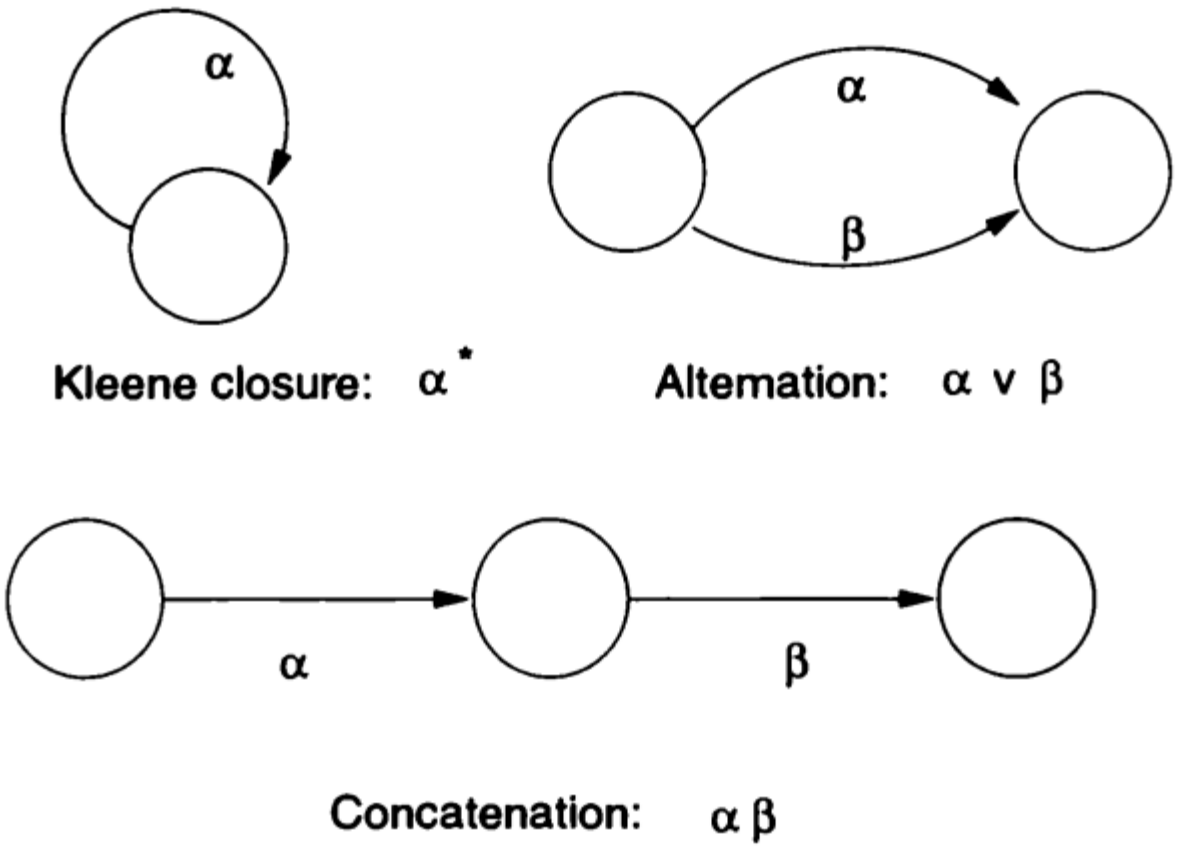


Figure Converting regular expressions into an FSA

Formal Translation Models

- **Pushdown Automata:**
  - Pushdown automata is equivalent to BNF grammars, that is, it can recognize strings generated by BNF grammars.
  - A pushdown automata is an abstract model machine similar to the FSA. It has a finite set of states. However, in addition, it has a pushdown stack. Moves of the PDA are as follows:
    1. An input symbol is read and the top symbol on the stack is read.
    2. Based on both inputs, the machine enters a new state and writes zero or more symbols onto the pushdown stack.
    3. Acceptance of a string occurs if the stack is ever empty. (Alternatively, acceptance can be if the PDA is in a final state.)
  - PDAs are more powerful than FSA. For example, strings like $a^n b^n$, $n > 0$, which cannot be recognized by an FSA, can easily be recognized by the PDA.

Formal Translation Models

- **Parsing Algorithms:**
  - Parsing algorithms are used to construct a parse tree for an input string.
  - There are three general types of parsers for grammars: **universal**, **top-down**, and **bottom-up**.
  - **Universal parsing** methods can parse any grammar. These methods are too inefficient to use in production compilers.
  - **Top-down methods** build parse trees from the top (root) to bottom (leaves). Most common top-down parsing methods are ***recursive-descent*** parsing and ***predictive parsing***. *Recursive descent parsing* may require backtracking to find correct production to be applied. In *predictive parsing*, no backtracking is required.
  - **Bottom-up methods** start form the leaves and work their way up to the root. A common bottom-up parsing method is ***shift-reduce parsing***.

Attribute Grammars

- Attribute grammars are used to define semantic model of a programming language. The idea is to associate a function with each node in the parse tree of a program giving the semantic content of that node.
- These grammars are used to pass semantic information around the syntax tree.
- Attribute grammars are created by adding functions (attributes) to each rule in a grammar.
- Attributes may be of two types – *inherited attribute* and *synthesized attribute*. An ***inherited attribute*** is a function that relates nonterminal values in a tree with nonterminal values higher up in the tree. In other words, the functional value for the nonterminals on the right of any rule are a function of the left-hand side nonterminal.

Attribute Grammars

- **Synthesized attribute** is a function that relates left-hand side nonterminal to values of right-hand side nonterminals. These attribute pass information up the tree.
- Consider the grammar given blow.

  $E \rightarrow T \mid E + T$

  $T \rightarrow P \mid T \times P$

  $P \rightarrow I \mid (E)$

  We can define the semantics of this language by a set of relationships among the nonterminals of the grammar as,

Production	**Attribute**
$E \rightarrow E + T$	$value(E_1) = value(E_2) + value(T)$
$E \rightarrow T$	$value(E) = value(T)$
$T \rightarrow T \times P$	$value(T_1) = value(T_2) \times value(P)$
$T \rightarrow P$	$value(T) = value(P)$
$P \rightarrow I$	$value(P) = value\ of\ number\ I$
$P \rightarrow (E)$	$value(P) = value(E)$

Attribute Grammars

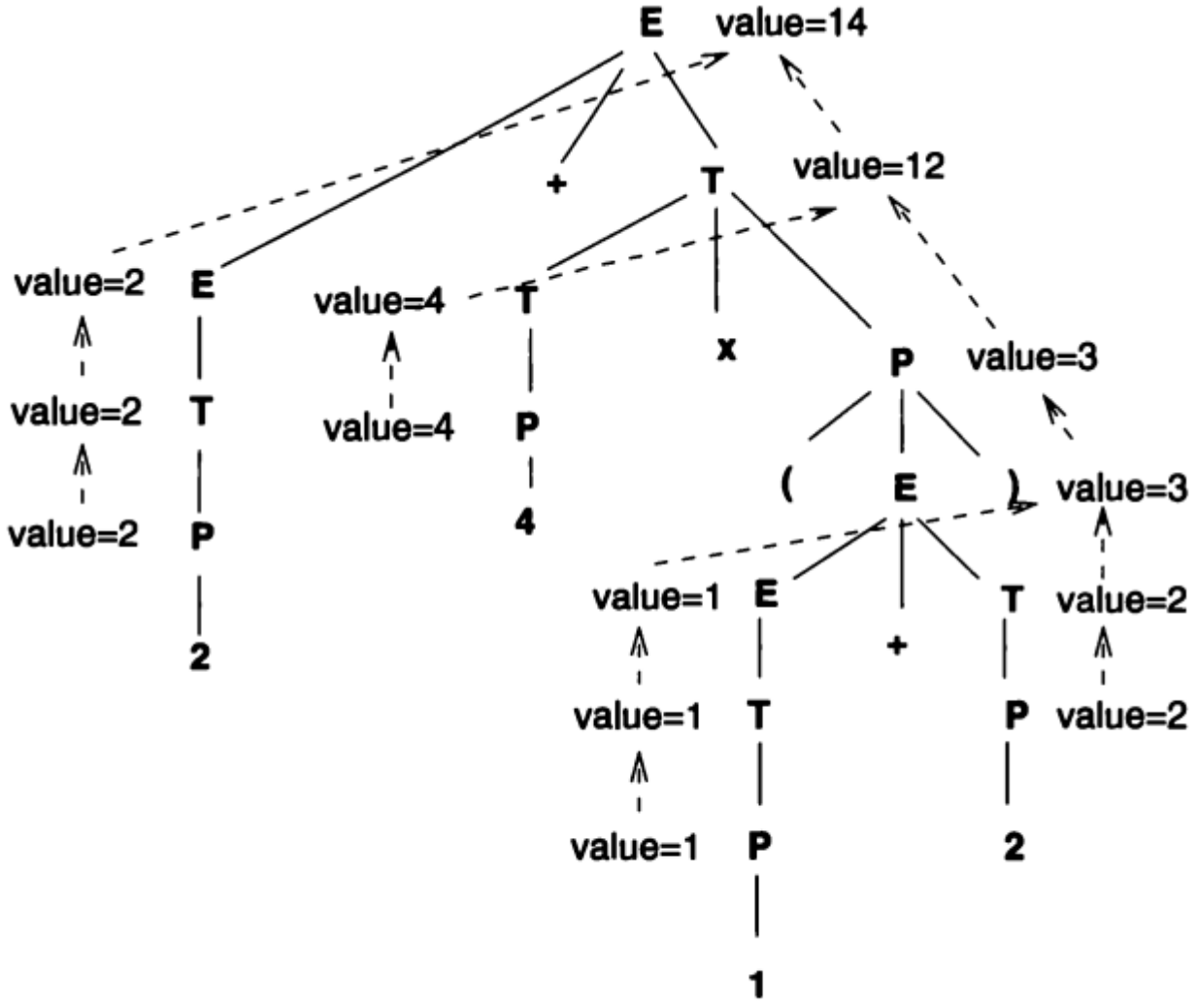


Fig: An attributed tree giving the value of the expression 2 + 4 × (1 + 2)

Denotational Semantics

- Denotational semantics is a formal method for expressing the semantic definition of a programming language.
- In denotational semantics, the idea is to map every syntactic entity in a programming language into some appropriate mathematical entity.
- Each denotational semantic definition consists of 5 parts: (1) syntactic categories, (2) BNF defining the structure of the syntactic categories, (3) value domains (the mathematical entities to be mapped), (4) semantic functions (signatures for mappings from syntax to semantic domains), and (5) semantic equations (the rules defining the semantic functions) mappings.
- In general, there will be one semantic function for each syntax category, and one semantic equation for every production in the syntax grammar.

Denotational Semantics

Syntactic Domains

N : Numeral -- nonnegative numerals

D : Digit -- decimal digits

Abstract Production Rules

Numeral ::= Digit | Numeral Digit

Digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Semantic Domain

Number = { 0, 1, 2, 3, 4, ... } -- natural numbers

Semantic Functions

value : Numeral → Number

digit : Digit → Number

Semantic Equations

value ⟦N D⟧ = *plus* (*times*(10, *value* ⟦N⟧), *digit* ⟦D⟧)

value ⟦D⟧ = *digit* ⟦D⟧

digit ⟦**0**⟧ = 0 *digit* ⟦**3**⟧ = 3 *digit* ⟦**6**⟧ = 6 *digit* ⟦**8**⟧ = 8

digit ⟦**1**⟧ = 1 *digit* ⟦**4**⟧ = 4 *digit* ⟦**7**⟧ = 7 *digit* ⟦**9**⟧ = 9

digit ⟦**2**⟧ = 2 *digit* ⟦**5**⟧ = 5

Figure : A Language of Numerals