Why Studying Programming Languages?

- Learn how to describe or define programming languages.

  Syntax: how the form or structure of the expressions, statements, and program units (main program, method/function, etc) are formed.

  Semantics: what the meaning of the expressions, statements, and program units.

- Examine carefully and evaluate the underlying features of programming languages. e.g. control structures, data structures, and data abstractions.

- Gain experience with languages other than C and Java. (Fortran, C++, Ada, Lisp, etc.).

- Improved background for choosing appropriate languages.

- Increase ability to learn new languages.

Programming Domains

• Scientific applications
– Large number of floating point computations
– Fortran

• Business applications
– Produce reports, use decimal numbers and characters
– COBOL

• Artificial intelligence
– Symbols rather than numbers manipulated
– LISP

• Systems programming
– Need efficiency because of continuous use
– C, C++

• Web Software
– HTML, PHP, Java

Language Evaluation Criteria

Evaluate features of languages, focusing on their impact on the software development process, including maintenance.

- **Readability**: the ease with which programs can be read and understood.
  - Control structures (Chap. 8)
  - Data types and structures (Chap. 6)
- **Writability**: the ease with which a language can be used to create programs.
  - Support for subprograms (Chap. 9 & 10)
  - Support data abstractions and encapsulation constructs (Chap. 11)
  - Expressions and assignment statements (Chap. 7)
- **Reliability**:
  - Type checking (Chap5)
  - Exception handling. (Chap. 14)
- **Cost**:
  - Training programming to use the language
  - Language closeness to the particular application.
  - The cost of maintaining program.

Evolution of Programming Languages

The Early Years

• Plankalkül **–** 1945
– Never implemented
– published only in 1972
– Advanced data structures: floating point, arrays, records

• FORmulaTRANslator
– I (1957), II(1958), IV(1962),77, 90
– First implemented language
– Focus on scientific applications
-- Arrays, floating point, counting loops

Languages of the Sixties

Algol 60 (1960)
- block structure
- call-by-value, call-by-name
- records
- recursion
- dynamic arrays: the size of the array is set at the time storage is allocated to the array.
- BNF syntax
  - All subsequent procedural languages based on it
  - Algorithm publication language for over 40 years

COBOL (1960):
- Business oriented:
  - Elaborate reports, decimals,
- Very English like
- Still in use today

LISP (LISt Processing language):

- Functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- Still the dominant language for AI

Languages of the Seventies

Pascal: simplified/improved Algol
Prolog
C: systems programming

Smalltalk

- First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic type binding)
- Pioneered the graphical user interface everyone now uses

Languages of Eighties and Beyond

Ada:
– Packages, exceptions

C++:
• Developed at Bell Labs
• Evolved from C and SIMULA 67
• Also has exception handling
• A large and complex language, in part because it supports both procedural and OO programming

Java:
• Developed at Sun in the early 1990s
• Based on C++
– Significantly simplified (does not include struct, union, pointer arithmetic)
– Supports *only* OOP
– Has references, but not pointers.

Chapter 3: Describing Syntax and Semantics

- Introduction
- Formal methods of describing syntax (BNF)
- Parse tree

The programmer must be able to determine how the expressions, statements, and program units of a language are formed, and their intended effect when executed.

How can we describe programming languages?

- **Syntax:** how the form or structure of the expressions, statements, and program units (main program, method/function, etc) are formed.
- **Semantics:** what the meaning of the expressions, statements, and program units.

while (<boolean_expr>) <statement>

- Syntax analysis:

    1. Lexical level: Lexical analyzer gathers characters of the source program into lexical units (tokens).

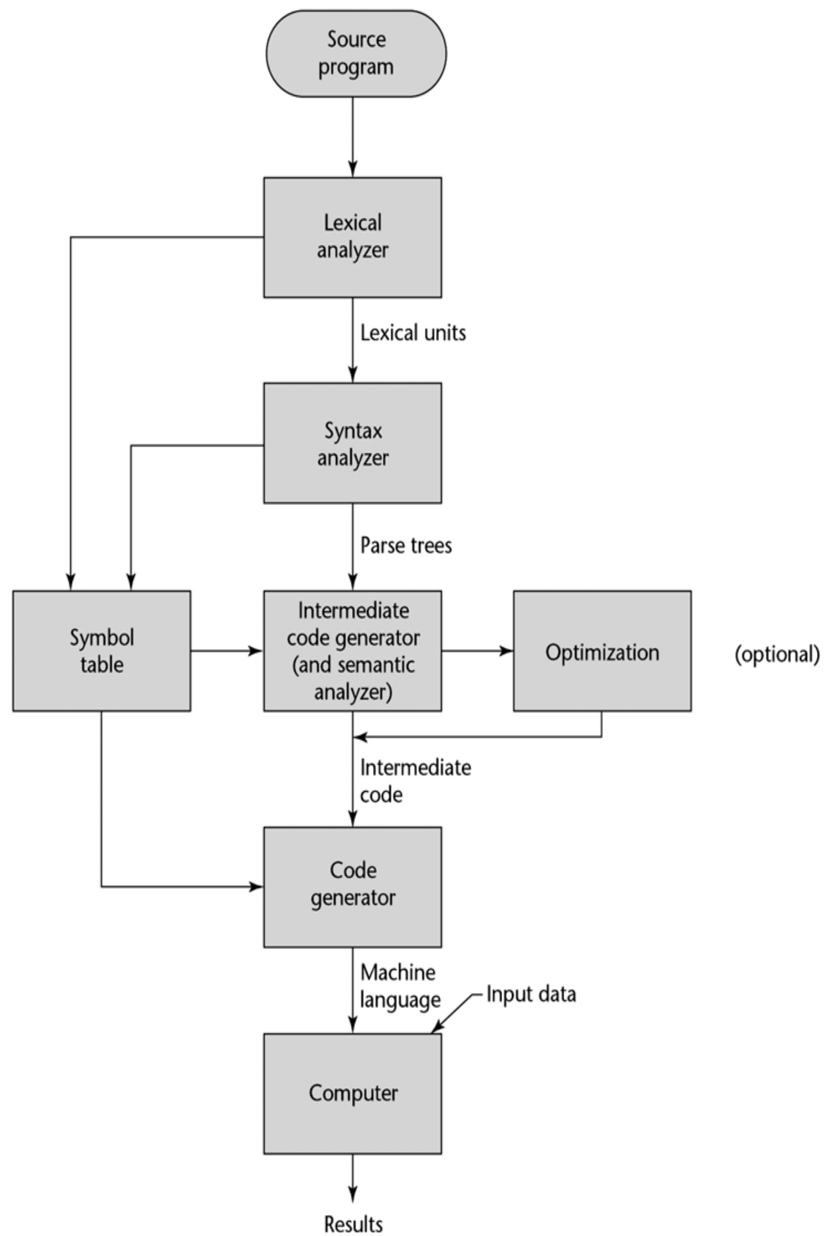
    2. Syntactic level: Syntactic analyzer constructs parse trees that represent the syntactic structure.

    - Determine whether the given programs are syntactically correct.

- Semantic analysis and intermediate code generator:
    - checks for errors like type checking, division by zero.
    - Produce a program between the source and machine language.
- Code generator: translates the intermediate code to machine code.

Figure 1.3

The compilation process



Lexical Level

The descriptions of the lowest level of syntactic units including numeric literals, operators, etc. (e.g., *, sum, begin).

-These small units are called ***Lexemes.***

A token of a language is a category of its lexemes.

Identifiers: Names representing data items, functions and procedures, etc.

Keywords: Names chosen by the language designer to represent particular language constructs which cannot be used as identifiers.

Operators: Special “keywords” used to identify operations to be performed on ***operands***, e.g. maths operators.

Punctuations: such as (or ;

Literals: direct values,

_ Numeric, e.g. 1, -123, 3.14, 6.02e23.

_ Character, e.g. ‘a’.

_ String, e.g. “Some text”.

Example: return -x;

Lexemes	***tokens***
return	keyword
-	operator
x	identifier
;	punctuation

The lexical structure ignores some characters such as whitespace and comments.

```
if (x = 0)
 y=y+1; // y is increased by one
```

Identifiers:

Keywords:

Punctuations:
Operator:
Literals:

Syntactic Level

The syntactic level describes the way that program statements are constructed from tokens.

• Precisely defined in terms of a ***context free grammar***.

- The best known examples are BNF (Backus Naur Form) or EBNF (Extended Backus Naur Form).

Formal Methods of Describing Syntax

• Backus-Naur Form (BNF)
– Most widely known method for describing programming language syntax

• Extended BNF
–Improves readability and writability of BNF

Backus-Naur Form (BNF)

– Invented by John Backus to describe Algol 58
– A *metalanguage* used to describe another language

BNF Fundamentals

• Non-terminals: acts as a placeholder for other symbols that describe the language.

• Terminals: lexemes or tokens

The notation for BNF we will use is:

- Angle brackets, <...>, for a non-terminal.
- Vertical bar, ...|..., for choice.
- Parenthesis, (...), for grouping.

A **BNF** grammar is simply a collection of **rules**.

<while_stmt> → while (<logic_expr>) <stmt>

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminal* and *nonterminal* symbols
- An nonterminal symbol can have more than one RHS

<stmt> → <single_stmt> | begin <stmt_list> end

- A grammar is a finite nonempty set of rules

<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const

- Syntactic lists are described using recursion

<ident_list> → ident | ident, <ident_list>

Derivation

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

<program> => <stmts>
=> <stmt>
=> <var> = <expr>
=> a =<expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const

<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const

BNF rules are used to generate *sentences*. A ***language*** is the set of all sentences that can be generated by the rules.

EXAMPLE 3.1	A Grammar for a Small Language
	<program> → **begin** <stmt_list> **end** <stmt_list> → <stmt> | <stmt> ; <stmt_list> <stmt> → <var> = <expression> <var> → A | B | C <expression> → <var> + <var> | <var> – <var> | <var>

A derivation of a program in this language follow:

<program> => begin <stmt_list> end
=> begin <stmt>;<stmt_list> end
=> begin <var>=<expression>;<stmt_list> end
=> begin A=<expression>;<stmt_list> end
=> begin A=<var>+<var>;<stmt_list> end
=> begin A=B+<var>;<stmt_list> end
=> begin A=B+C;<stmt_list> end
=> begin A=B+C;<stmt> end

=> begin A=B+C;<var>=<expression> end
=> begin A=B+C; B=<expression> end
=> begin A=B+C; B=<var> end
=> begin A=B+C; B=C end

Leftmost derivation: the leftmost nonterminal is the one that is expanded.
Rightmost derivation: the rightmost nonterminal is the one that is expanded.

Examples: Rightmost derivation for num+num*num

<E> -> <E> + <T> | <T>| <E> - <T>
<T> -> <T> * <F> | <F>| <T>/<F>
<F> -> (<E>) | id| -<E>| num

E => E + T
=> E + T * F
=> E + T * num
=> E + F * num
=> E + num * num
=> T + num * num
=> F + num * num
=> num + num * num

BNF is “Context Free” Language

Context-free languages can be described by grammars in which

- The left hand side is a nonterminal
- The right hand side is an arbitrary string of terminals and nonterminals.

For example, the language "$a^n b^n$" ($n > 0$) can be described by the rules:

$\langle S \rangle \rightarrow a\ b \mid a\ \langle S \rangle\ b$

Extended BNF

Adds three extensions to BNF:

1. Bracket notation indicates an optional part of the RHS.

   <if> → **if** (<expression>) <statement> [**else** <statement>]

2. Curly brackets {} indicates a sequence of 0 or more occurrences of the subsequence.

   <decl> → <type> <variable> {, <variable>};

3. Parentheses and followed by a * indicates 0 or more occurrences, or a +, indicating 1 or more occurrence.

   <decl>→ <type> <variable> (, <variable>)*;

If two RHSs are the same except for one constituent, EBNF allows that constituent to be shown in parentheses with an infix | operator.

<expression> → <variable> | <expression> (* | +) <variable>

Example:

<identifier> → <alphabetic> { <alphanumeric> }
<alphanumeric> → <alphabetic> | <numeric> | '_'
<alphabetic> → 'a'-'z' | 'A'-'Z'
<numeric> → '0'-'9'

The grammar defines a *language*, that is a set of *valid sentences*, for example:
a a1 aFoobar A_FOO   but not: @ 1a a$nake _A_BAR

Converting EBNF To BNF

- EBNF

  <expr> → <term> {(+ | -) <term>}
  <term> → <factor> {(* | /) <factor>}

- BNF

  <expr> → <expr> + <term>
  | <expr> - <term>
  | <term>
  <term> → <term> * <factor>
  | <term> / <factor>
  | <factor>

Parse Tree

A hierarchical representation of a derivation

The BNF or EBNF notation can be used to describe valid parse trees.

<exp> → <identifier>
| <literal>
| <unary> <exp>
| <exp> <binary> <exp>
<binary> → '<' | '>' | '+' | '-'
<unary> → '-'

This generates a language including:
x+1 x+y+z 1+2-3 ...

From the grammar we can generate a parse tree.

Example: **x + y + z** is parsed:

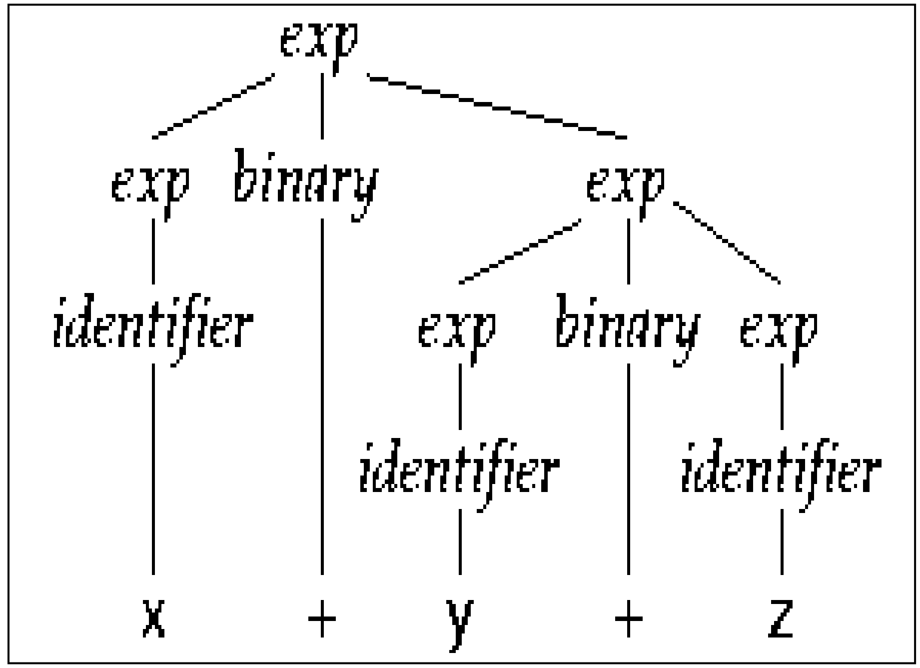
<exp> => <exp> <binary> <exp>

<exp> → <identifier>
| <literal>
| <unary> <exp>
| <exp> <binary> <exp>

<binary> → '<' | '>' | '+' | '-'

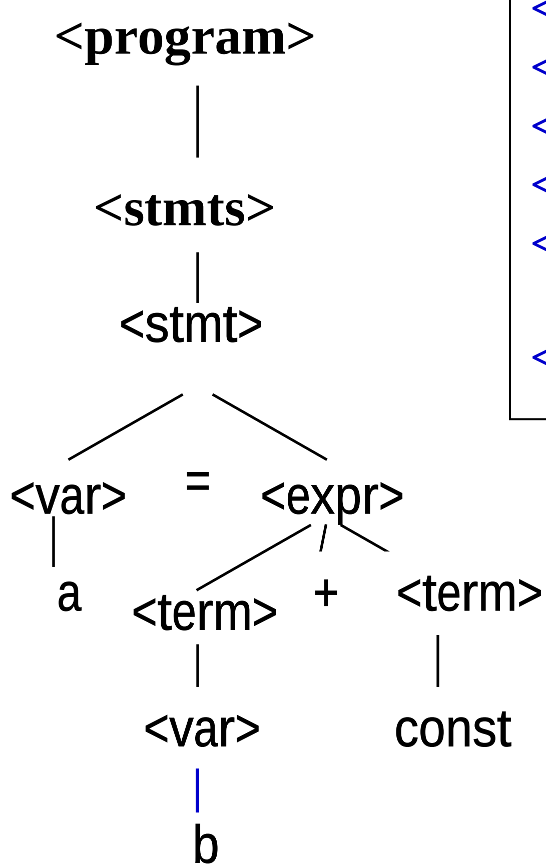
<unary> → '-'

<exp>=> <exp><binary><exp>
=> <identifier><binary> <exp>
=> x <binary> <exp>
=> **x** + <exp>
=> **x** + <exp> <binary> <exp>
=> **x** + <identifier><binary> <exp>
=> **x** + **y** <binary> <exp>
=> **x** + **y** + <exp>
=> **x** + **y** + < identifier >
=> **x** + **y** + z



Every internal node of a parse tree is a nonterminal symbol.
Every leaf is a terminal symbol.

a = b + const is parsed



<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const