# Advanced Programming Languages (CSci 460)

Lecture Notes

# Contents

# 1 Concepts of Programming Languages

## 1.1 Reasons for Studying Concepts of Programming Languages

- **Increased capacity to express ideas.** People only a weak understanding of natural language are limited in the complexity of their thoughts, particularly in depth of abstraction. In other words, it is difficult for people to conceptualize structures they cannot describe, verbally or in writing. In the software world, programmers are similarly constrained by the languages they know. Awareness of a wider variety of programming language features can reduce such limitations in software development.
- **Improved background for choosing appropriate languages.** Familiarity with a wider variety of languages and language constructs means programmers are better able to choose the language with the best features that best address a specific problem.
- **Increased ability to learn new languages.** Computer programming is a relatively young discipline, and thus it's still in a state of continuous evolution. While software development is an exciting profession, it also demands continuous learning of new technologies. Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.
- **Better understanding of the significance of implementation.** In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts which leads to an understanding of why languages are designed the way they are. In turn, this knowledge leads to the ability to use a language more intelligently, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices.
- **Better use of languages that are already known.** Most contemporary programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses. By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.
- **Overall advancement of computing.** In general, if those who choose languages were well informed, perhaps better languages would eventually squeeze out poorer ones. For example, many people argue that it'd have been better if ALGOL 60 had displaced FORTRAN in the early 1960s, because it was more elegant and had much better control statements, among other reasons. That it did not, is due partly to the programmers and software development managers of that time, many of whom did not clearly understand the conceptual design of ALGOL 60.

## 1.2 Programming Domains

- **Scientific applications**
  - The first digital computers, which appeared in the late 1940s and early 1950s, were invented and used for scientific applications.
  - Large numbers of floating-point arithmetic computations.
  - Data structures (e.g., arrays and matrices) and control structures (e.g., counting loops and selections).
  - Fortran was the first language for scientific application; for some scientific applications where efficiency was the primary concern, no language is significantly better than Fortran.
- **Business applications**
  - The use of computers for business applications began in the 1950s. Special computers were developed for this purpose, along with special languages.

  - Production of elaborate reports, precise ways of describing and storing decimal numbers and character data, and the ability to specify arithmetic operations.
  - COBOL was the first successful HLL for business.
- **Artificial Intelligence**
  - Artificial intelligence (traditional AI) is a broad area of computer applications characterized by the use of symbolic rather than numeric computations (i.e., symbols, consisting of names rather than numbers, are manipulated).
  - Less computationaly intensive, but required more flexibility than other programming domains.
  - Data structures (e.g., symbols, linked lists)
  - Lisp was the widely used programming language developed for AI applications, which appeared in 1959.
- **Web applications**
  - Pervarsive need for dynamic Web content, portability and security are also important.
  - JAVA was among the first programming languages used in the Web. Nowadays languages such as JavaScript and PHP are more common part of the Web.

## 1.3 Criteria for Evaluating Languages

### 1.3.1 Readability

Readability concerns itself with how easy is it to read/understand a program in a given language.

- **Simplicity.** A language with a large number of basic constructs is more difficult to learn than one with a smaller number. Programmers who must use a large language often learn a subset of the language and ignore its other features. Readability can be reduced ina few ways:
  - **language subset**, in which the program's author has learned a different subset from that subset with which the reader is familiar. Example: C++, Perl, etc.
  - **feature multiplicity** by which there's more than one way to to accomplish a task. Example: Variable increment as in `count = count + 1`, `count += 1`, `count++`, and `++count`.
  - **operator overloading**, in which a single operator symbol has more than one meaning. Example: In C++, users can overload built-in operators with their semantics. Some languages such as Raku mistigate this by encouraging users to create their own operators instead of overloading the built-in ones.
- **Orthogonality.** This means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Orthogonality follows from a symmetry of relationships among primitives and a lack of orthogonality leads to exceptions to the rules of the language. In general, orthogonality means every combination of constructs can be used in every context.
  - Example: Lack of orthogonality in C means `structs` can be returned from functions but arrays cannot, from functions but arrays cannot, a member of a structure can be any data type except void or a structure of the same type, an array element can be any data type except `void` or a function, parameters are passed by value unless they're arrays in which they're passed by reference, etc.
- **Data types.** The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability.
  - Example: A `Boolean` type for setting up flags, instead of using integers. Thus, `Bool verbose = True` (meaning is perfectly clear) instead of `verbose = 1` (meaning isn't clear).
  - Many languages allow the programmers to extend the language from its built-in constructs. In Raku, `subset` is a way to construct a subset using the built-in types

(e.g., `subset PosInt of Int where * >= 1` where `PosInt` is a derived type of `Int` for only positive integers).

- **Syntax design.** The syntax, or form, of the elements of a language has a significant effect on the readability of programs.
  - **Special words.** Special words. Program appearance and thus program readability are strongly influenced by the forms of a language's special words (for example, `while`, `unless`, `class`, `for`, etc.). If special words of a language can be used as names for program variables, then readibility is reduced (e.g., in Fortran 95, special words such as `Do` and `End` are legal variable names).
  - **Form and meaning.** Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability. Semantics, or meaning, should follow directly from syntax, or form. This principle is violated by two language constructs that are identical or similar in appearance but have different meanings, depending perhaps on context. For example, in C the meaning of the reserved word `static` depends on the context on which it appears.

### 1.3.2 Writability

Writability is a measure of how easily a language can be used to create programs for a chosen problem domain. Most of the language characteristics that affect readability also affect writability.

- **Simplicity and orthogonality.** If a language has a large number of different constructs, some programmers who use the language might not be familiar with all of them. Similarly, too much orthogonality can a detriment to writability since errors in programs can go undetected when nearly any combination of primitives is legal.
- **Expressivity.** Expressivity in a language can refer to several different characteristics. More commonly, it means that a language has relatively convenient, rather than cumbersome, ways of specifying computations. For example, in C `count++` is more convenient and shorter than `count = count + 1`, the `and...then` Boolean operator in Ada is a convenient way of specifying short-evaluation of a Boolean expression, use of Boolean expressions without explicit comparison/conversion in control statements such as `if` (e.g., `if verbose {...}` instead of `if verbose == true {...}`), `for` loops that iterate over iterable constructor instead of using a `while` loop which indexing (e.g., `for students { ... }` instead of `while (idx < students.size) {...}`), etc.
- **Support for abstraction.** Can you concentrate on a single level abstraction (ignoring details while programming? For example, there's no need to drop to Assembly when writing most modern applications, most HLLs provide nice levels of abstraction that allows the programmer to concentrate on the problem at hand.

### 1.3.3 Reliability

A program is said to be reliable if it performs to its specifications under all conditions.

- **Type checking.** Simply testing for type errors in a given program, either by the compiler or during program execution. Because run-time type checking is expensive, compile-time type checking is more desirable. Furthermore, the earlier errors in programs are detected, the less expensive it is to make the required repairs.
- **Exception handling support.** The ability of a program to intercept run-time errors, take corrective measures, and then continue is an obvious aid to reliability. Ada, C++, Java, and C# include extensive capabilities for exception handling, but such facilities are practically nonexistent in some widely used languages, for example, C.
- **Aliasing.** Loosely defined, **aliasing** is having two or more distinct names in a program that can be used to access the same memory cell. Nowadays aliasing considered harmful.

Still most programming languages allow some kind of aliasing—for example, two pointers (or references) set to point to the same variable, which is possible in most languages. In some languages, aliasing is used to overcome deficiencies in the language's data abstraction facilities. Other languages greatly restrict aliasing to increase their reliability.

## 1.4 Trade-offs

- **Reliability vs cost.** Java requires bound checking on all array access which makes them more reliable but results in greater execution cost. In C the opposite is the case: lower execution cost but less reliability.
- **Readability vs writability.** APL has numerous (and powerful) operators which leads to small programs but unreadable programs to programmer not acquainted with the language. Also the large number of operators usually require a custom keyboard for the language.
- **Writability vs reliability.** Pointers in C and C++ give flexibility for many low-level operations, but are unreliable and hard to verify.

## 1.5 Lifecycle Cost Factors

- **Cost of training programmers to use the language**, which is a function of the simplicity and orthogonality of the language, and the programmer's experience.
- **Cost of writing programs in the language**, which is a function of the writability of the language, which in parts depends on its closeness in purpose to the particular application.
- **Cost of executing programs written in a language** is greatly influenced by that language's design. A language that requires many run-time type checks will prohibit fast code execution, regardless of the quality of the compiler.
- **Cost of maintenance of programs in the language** which is both depended on the readability and writability of the language. While most programs are only written once, they need to be mantained by long periods of time usually by different programmers.

## 1.6 Programming Paradigms (The Traditional Way)

- **Imperative/procedural.** Statements are executed for their side-effects on environment/variables. Usually control-based.
- **Functional/applicative.** Statements are executed for their input-output functionality, and not their side-effects. Based on composition of functions (lambda calculus).
- **Logic.** Programmer states **what** the problem is, and not **how** to solve it; thus, no statements. Based on relations and first-order logic (FOL).
- **Object-oriented.** Programs are structured around abstract data types, with encapsulation, inheritance, etc.
- **Concurrent.** Intended to model parallel tasks, maybe for execution on a parallel target architecture.

## 1.7 Programming Paradigms (Another Way)

Object-oriented, concurrent, etc. are not paradigms independent of the others. Thus:

```
{Imperative, Functional, Logic} X
{Non-object-oriented, Object-oriented} X
{Sequential, Concurrent, Reactive} = {
    Imperative Non-object-oriented Sequential,
```

```
    Imperative Non-object-oriented Concurrent,
    ...,
    Logic Object-oriented Concurrent,
    Logic Object-oriented Reactive,
}
```

## 1.8 A Few Programming Languages

| | Sequential | Concurrent | Reactive |
|---|---|---|---|
| Imperative | 1955: Fortran<br>1958: Algol<br>1964: PL/I<br>1971: Pascal<br>1972: C<br>1973: Basic<br>1967: Simula<br>1972: Smalltalk<br>1983: C++<br>1986: Eiffel<br>1988: Modula-3<br>1995: Java<br>2000: C# | 1983: Ada | 1980s: VHDL, Verilog, Esterel<br>2001: SpecC<br>1987: StateChart<br>2005: SystemC |
| Functional | 1959: LISP<br>1975: Scheme<br>1962: APL<br>1968: Logo, Forth<br>1970s: ML<br>1990: Haskell<br>1989: CLOS | 2000: Alice<br>1980s: ADLs | 1993: Lustre<br>1980s: ADLs |
| Logic | 1973: Prolog<br>1978: Datalog<br>1980s: CLPs<br>1987: ObjLog | 1983: Parlog<br>1980's: GHC, KL | |

- Non-object oriented
- Object oriented

Figure 1: Examples of programming languages and the paradigms they fit in.

# 2 Implementation Methods

Two of the primary components of a computer are its internal memory and its processor:

- The **internal memory** is used to store programs and data.
- The **processor** is a collection of circuits that provides a realization of a set of primitive operations, or machine instructions, such as those for arithmetic and logic operations.

The machine language of the computer is its set of instructions, and in the absence of other supporting software, its own machine language is the only language that most hardware computers "understand." Theoretically, a computer could be designed and built with a particular high-level language as its machine language, however

- it would be very complex and expensive,
- it would be highly inflexible, because it would be difficult (but not impossible) to use it with other high-level languages.

The more practical machine design choice implements in hardware a very low-level language (e.g., machine code) that provides the most commonly needed primitive operations and requires system software to create an interface to programs in higher-level languages.

## 2.1 Layered View of a Computer System

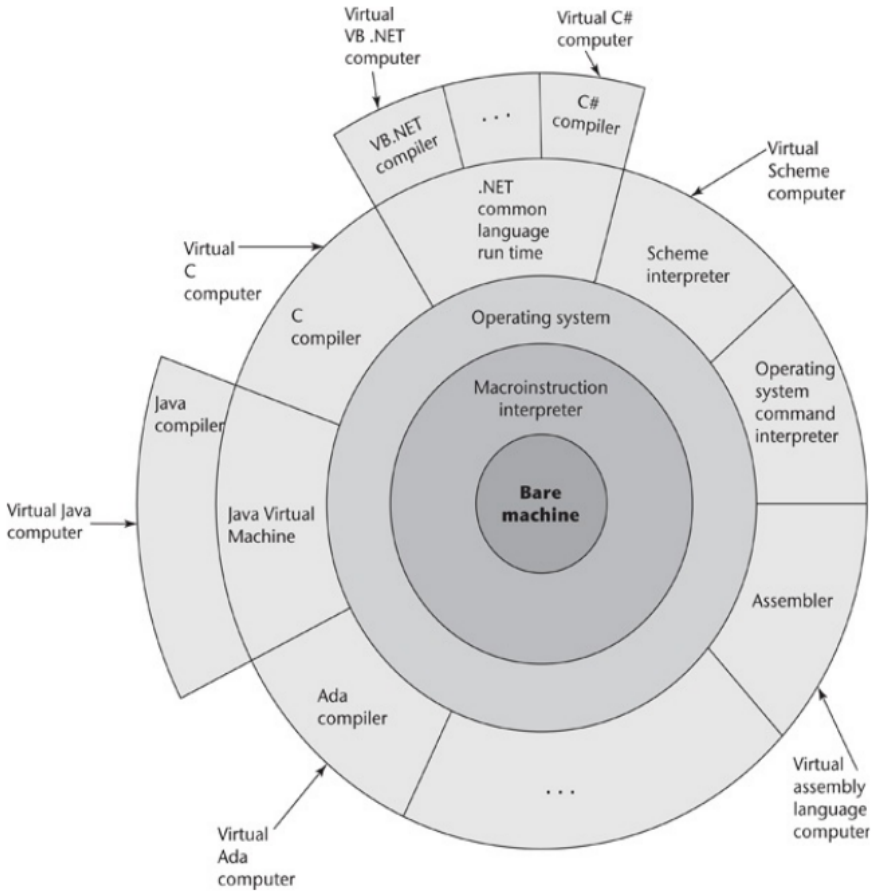



Figure 2: Layered view of computer systems.

- On top of the **bare machine**, instructions that handle basic operations such arithmetic and logic operations are known as **macro-instructions**, which are actually implemented with a set of instructions called **microinstructions**, which are defined at an even lower level. Microinstructions are never seen by software.

- The **operating system** is a large collection of programs, which supplies higher-level primitives than those of the machine language. These primitives provide system resource management, I/O operations, a file management system, text and/or program editors, etc.
- **Language implementation systems** need many of the operating system facilities, so they interface with the operating system rather than directly with the processor (in machine language). These layers can be considered as **virtual computers** that provide interfaces to the user at higher levels. For example, an operating system and a C compiler provide a virtual C compiler.
- **User programs** form another layer over the top of the layer of virtual computers.

## 2.2 von Neumann Bottleneck

The speed of the connection between a computer's memory and its processor often determines the speed of the computer, because instructions often can be executed faster than they can be moved to the processor for execution. This connection is called the **von Neumann bottleneck**; it is the primary limiting factor in the speed of von Neumann architecture computers.

The von Neumann bottleneck has been one of the primary motivations for the research and development of **parallel computers**.

## 2.3 Compilation

**Compilation** is one of the implementation methods for programming languages. By this method, programs are translated into machine code which can be executed directly on the computer. Note that usually an assembler is involved, however this is always called directly by the compiler, and for most intent and purposes, we assume the compiler translates a high-level language (HLL) directly to machine code.

The language that a compiler translates is called the **source language**. The process of compilation and program execution takes place in several phases:

1. **Source program:** The compilation process starts with the HLL's source program.
2. **Lexical analyzer:** The lexical analyzer gathers the characters of the source program into lexical units. The lexical units of a program are identifiers, special words, operators, and punctuation symbols while ignoring comments for which the compiler has no use.
3. **Syntax analyzer:** The syntax analyzer takes the lexical units from the lexical analyzer and uses them to construct hierarchical structures called **parse trees**, which represent the syntactic structure of the program.
4. **Intermediate code generator (ICG):** The ICG produces a program in a different language, at an intermediate level between the source program and the final output of the compiler: the machine language program. Intermediate languages sometimes look very much like assembly languages, and other cases the intermediate code is at a level somewhat higher than an assembly language.

   The **semantic analyzer** is an integral part of the intermediate code generator. The semantic analyzer checks for errors, such as type errors, that are difficult, if not impossible, to detect during syntax analysis.

   At this stage **optimization** can take place. Optimization improves programs (usually in their intermediate code version) by making them smaller, faster, or both. Because many kinds of optimization are difficult to do on machine language, most optimization is done on the intermediate code.

   The **symbol table** serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program. This information is placed in the symbol table by the lexical and syntax

analyzers and is used by the semantic analyzer (in the ICG) and the code generator (next stage).

5. **Code generator:** The code generator translates the optimized intermediate code version of the program into an equivalent machine language program.
6. **Computer:** The machine language generated by a compiler can be executed directly on the hardware, however it must always be run along with some other code. Most user programs also require programs from the operating system.
    1. **linking:** Before the machine language programs produced by a compiler can be executed, the required programs from the operating system must be found and linked to the user program. The linking operation connects the user program to the system programs by placing the addresses of the entry points of the system programs in the calls to them in the user program. All this is accomplished by a systems program called the **linker**.
    2. **load module:** Also known as **executable image**, this is simply the user and system code together after the linking process is done. The executable image is loaded into the computer by a systems program known as the **loader**.

## 2.4 Pure Interpretation

Unlike compilation, the method of **pure interpretation** has no translation whatsoever. Instead programs are interpreted by another program known as the **interpreter**, which acts as a software simulation of a machine whose fetch-execute deals with HLL programs rather than machine code instructions. This software simulation obviously provides a virtual machine for the language.

**Advantages:**

- Easy implementation of many source-level debugging operations because all run-time error message can refer to sourec-level units.

**Disadvantages:**

- Implementation of source-level debugging operations is slow, with the execution being 10 to 100 times slower that in compiled systems. The primary source of this slowness is the decoding of the high- level language statements, which are far more complex than machine language instructions

  Furthermore, regardless of how many times a statement is executed, it must be decoded every time. Therefore, **statement decoding, rather than the connection between the processor and memory, is the bottleneck of a pure interpreter**.
- More space is required. In addition to the source program, the symbol table must be present during interpretation.

  Furthermore, the source program may be stored in a form designed for easy access and modification rather than one that provides for minimal size.

## 2.5 Hybrid Implementation

**Hybrid implemenatation systems** are a compromise between compilers and pure interpreters; they translate high-level language programs to an intermediate language designed to allow easy interpretation. This method is faster than pure interpretation because the source language statements are decoded only once. Instead of translating intermediate language code to machine code, it simply interprets the intermediate code.

**Examples:**

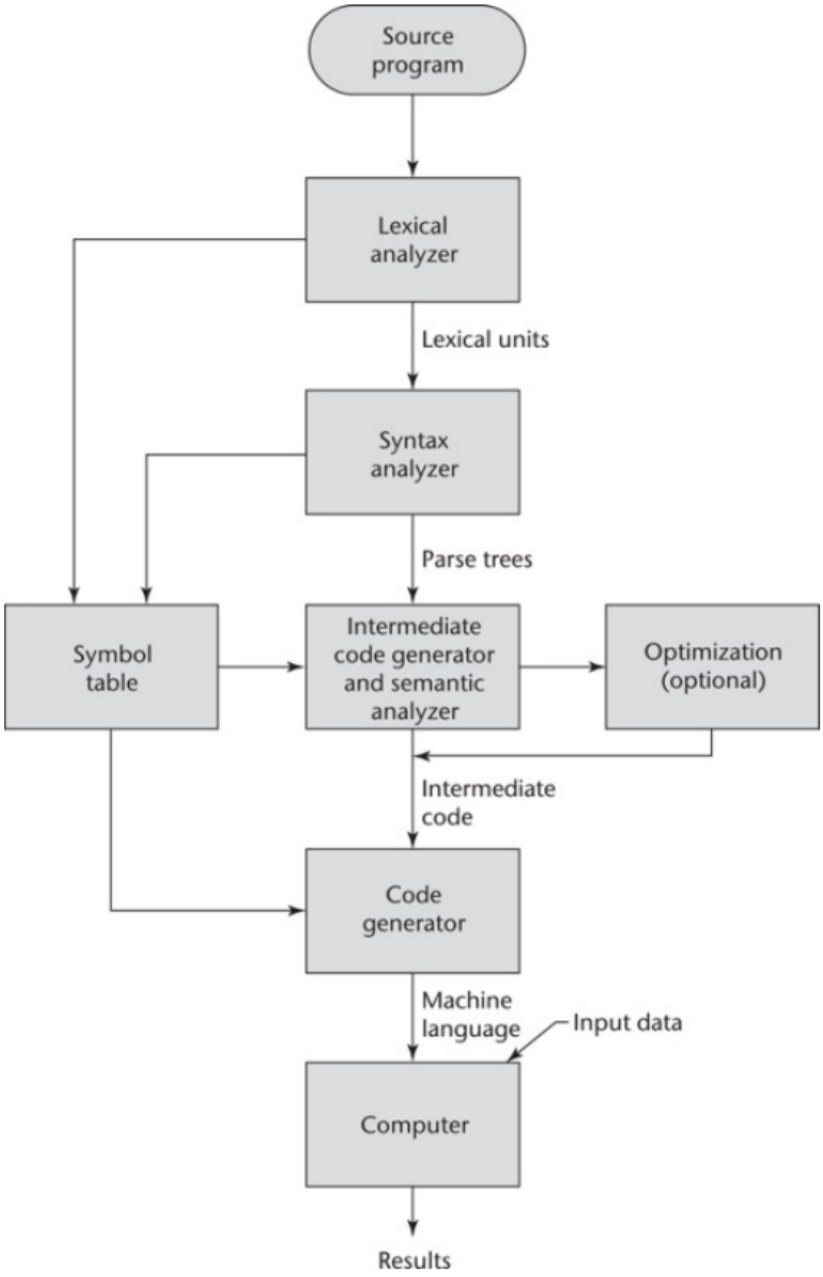



Figure 3: Compilation process

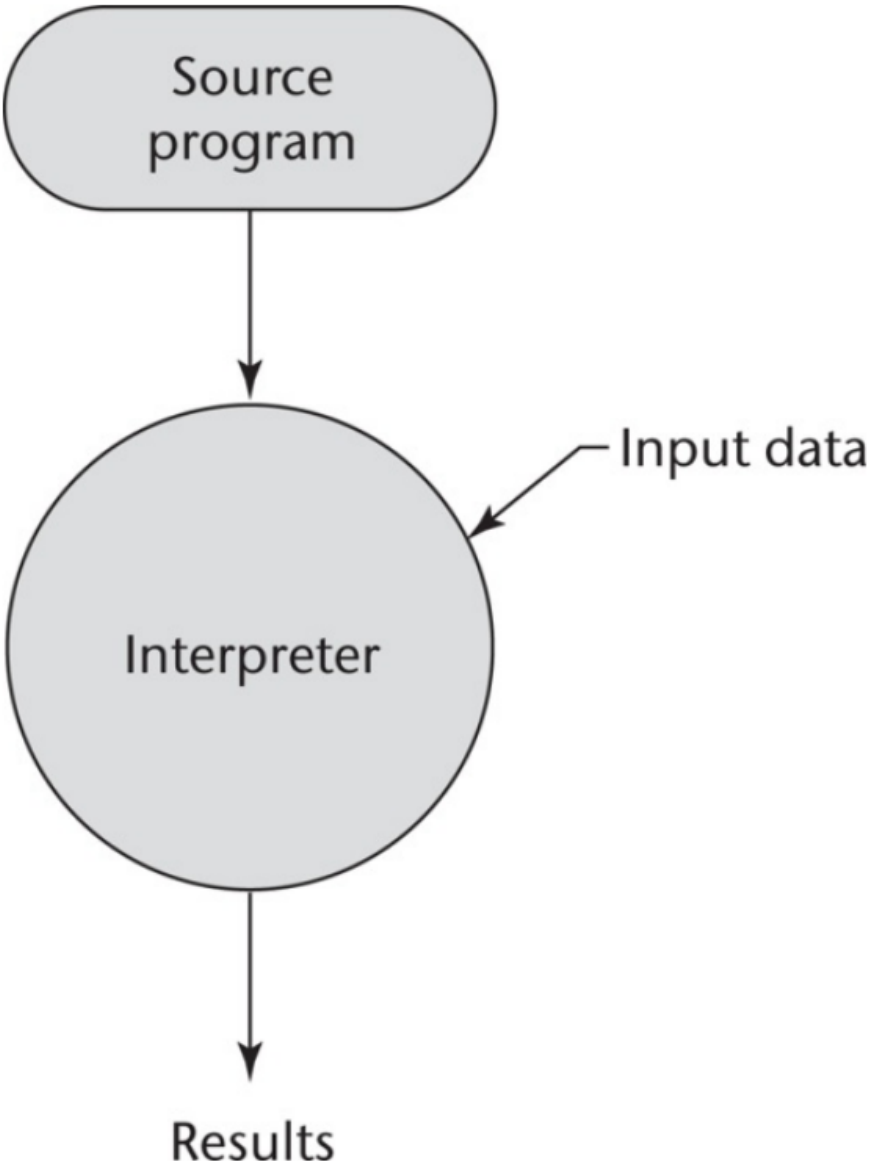



Figure 4: Pure interpretation process

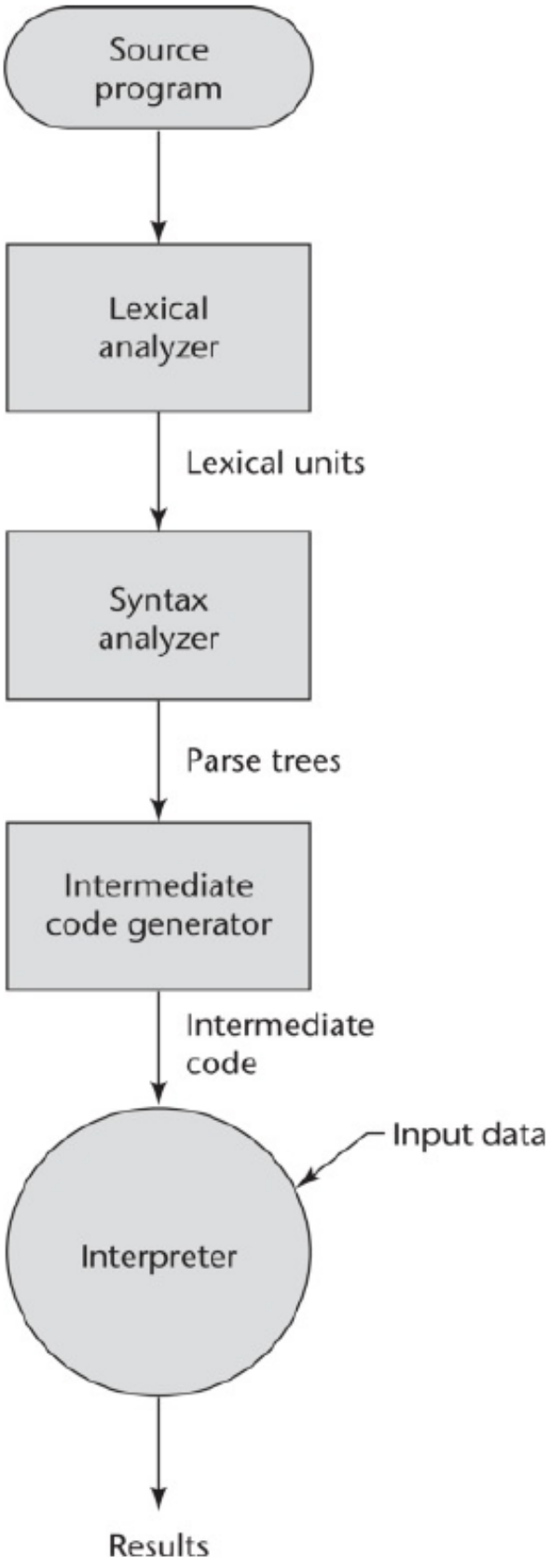



Figure 5: Hybrid implementation process

- Perl is implemented with a hybrid system. Perl programs are partially compiled to detect errors before interpretation and to simplify the interpreter.
- Initial implementations of Java were hybrid. Its intermediate form, called **bytecode**, provides portability to any machine that has a byte code interpreter and an associated run-time system. Together, these are called the **Java Virtual Machine (JVM)**.
- A **Just-in-Time (JIT)** implementation system initially translates programs to an intermediate language. Then, during execution, it compiles intermediate language methods into machine code when they are called. The machine code version is kept for subsequent calls. JIT systems now are widely used for Java programs.

## 2.6 Preprocessors

A preprocessor is a program that processes a program just before the program is compiled. Preprocessor instructions are embedded in programs. The preprocessor is essentially a macro expander. Preprocessor instructions are commonly used to specify that the code from another file is to be included. For example, the C preprocessor instruction

```
#include "myLib.h"
```

causes the preprocessor to copy the contents of `myLib.h` into the program at the position of the `#include`. This is akin to a search-and-replace operation in a text editor.

Other preprocessor instructions are used to define symbols to represent expressions. For example, one could use

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

to determine the largest of two given expressions. Thus, the expression `x = max(2 * y, z / 1.73);` would be expanded by the preprocessor to

```
x = ((2 * y) > (z / 1.73) ? (2 * y) : (z / 1.73);
```

# 3 Syntax and Grammars

The task of providing a concise yet understandable description of a programming language is difficult but essential to a language's success:

- On one hand, concise formal descriptions that aren't easily understandable due to partly new notation cause languages to lack implementations.
- On the other hand, some languages have suffered the problem of having many slightly different dialects, a result of a simple but informal and imprecise definition.

## 3.1 Syntax vs Semantic

- **Syntax** is the **form** of a programming language's expressions, statements, and program units.
  - Typically described using **grammars**.
  - It's easier to describe than semantic, partly because a concise and universally accepted notation is available for syntax description.
- **Semantic** is the **meaning** of those expressions, statements, and program units.
  - Typically described informally (e.g., English).
  - No universal notation has been developed for semantic so it's harder to describe than syntax.

Syntax and semantics are closely related. In a well-designed programming language, semantics should follow directly from syntax; that is, the appearance of a statement should strongly suggest what the statement is meant to accomplish.

### 3.1.1 Syntax Terminology

- **Sentence.** A sentence is a string of characters over some given alphabet. For example, `int foo = 5;` is a sentence over the alphabet that constitutes the C language.
- **Language.** A language is a set of sentences. For example, the set of all legal C programs.
- **Token.** A token is category of primitve units such as numeric literals, operators, keywords, etc.
- **Lexeme.** A lexeme is a small unit described by a token. For example, the lexeme 5 is described by the token numeric literal, the lexeme `for` is described by the token keyword, etc.

For example, given the following C statement:

```
index = 2 * count + 17;
```

we've the following lexemes-tokens breakdown:

| Lexemes | Tokens |
|---|---|
| `index` | identifier |
| `=` | equal_sign |
| `2` | int_literal |
| `*` | mult_op |
| `count` | identifier |
| `+` | plus_op |
| `17` | int_literal |
| ';' | semicolon |

## 3.2 Why Care About Formal Syntax?

In general, languages can be formally defined in two distinct ways:

- **Recognition:** Given an arbitrary sentence, determine whether it's legal in some given language. For example, a compiler's syntax analyzer is a **recognition device** that determines if a given sentence (e.g., whole programs) is a legal sentence in whatever language the compiler translates to. Such devices are like filters, separating legal sentences from those that are incorrectly formed.

  Recognition devices can only be used in trial-and-error mode. For example, to determine the correct syntax of a particular statement using a compiler, the programmer can only submit a speculated version and note whether the compiler accepts it. Thus they're not as useful a language description for a programmer.

- **Generation:** Given the formal sentence of a language, legal sentences are generated. Devices that generate the sentences of a language are known as **generation devices**.

  Unlike recognizers, generators are oftentimes useful to programmers because it's often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator.

There is a close connection between formal generation and recognition devices for the same language. This was one of the seminal discoveries in computer science, and it led to much of what is now known about formal languages and compiler design theory.

## 3.3 Chomsky's Hierarchy of Languages

| Type | Grammar | Machine | Applications |
|---|---|---|---|
| 3 | regular | DFA, NFA, ... | pattern matching, lexical structure |
| 2 | context-free | PDA | PLs (typically) |
| 1 | context-sensitive | bounded TM (Turing Machine) | Natural languages (exc. poetry) |
| 0 | recursive enumerable | TM | |

Remember that pushdown automatas (PDAs) have only one stack.

## 3.4 Context-Free Grammars

Typically represented using Backus-Naur Form (BNF) which is equivalent to CFG.

Formally, a CFG $< \Sigma, V, P >$ where

- $\Sigma$ is a set of **terminal symbols** (e.g., lexemes and tokens),
- $V$ is a set of **nonterminal symbols** (aka variables), containing a start symbol $S$,
- $P$ is a finite non-empty set of **rules** (or **productions**), each of the form:
  - $A \rightarrow \alpha$, where $A$ is a single nonterminal symbol, and $\alpha$ is a finite sequence of terminals and/or nonterminals,
  - $A$ belongs to $V$,
  - each nonterminal has at least one production, and
  - at least one production has $S$ on its left-hand side (LHS).

### 3.4.1 BNF Notation and Abbreviations

BNF uses abstractions for syntactic structures. For example, a simple Java assignment might represented by the following abstraction:

⟨*assign*⟩ → ⟨*var*⟩ = ⟨*expression*⟩

| Abstraction | Meaning |
|---|---|
| `<assign>` | This is the abstraction being defined; this is usually known as the **left-hand side** (LHS). |
| `<var> = <expression>` | This is the definition of the LHS; it's usually known as the **right-hand side** (RHS). |
| → | The arrow means "can have the form"; for brevity it's sometimes pronounced "goes to". |
| `<assign>` → `<var> = <expression>` | This whole definition is called a **rule** (or **production**). |

One example sentence whose syntactic structure is described by the rule is

```
total = subtotal1 + subtotal2
```

In a BNF description, or grammar:

- The abstractions are often called **nonterminal symbols**, and
- the lexemes and tokens of the rules are called **terminal symbols** (or simply **terminals**).

Overall, a BNF description, or **grammar**, is a collection of rules.

Nonterminal symbols can have two or more distinct definitions, representing two or more possible syntactic forms in the language. For example, a Java `if` statement can be described with the rules:

⟨*if_stmt*⟩ → if ( ⟨*logic_expr*⟩ ) ⟨*stmt*⟩
⟨*if_stmt*⟩ → if ( ⟨*logic_expr*⟩ ) ⟨*stmt*⟩ else ⟨*stmt*⟩

Multiple definitions can be written as a single rule, with the different definitions separated by the symbol `|`, meaning logical OR. Thus, the above description is the same as

### 3.4.2 Examples

**3.4.2.1 A Simple Arithmetic Expression** The following CFG describes the structure of an arithmetic expression:

⟨*expr*⟩ → ⟨*id*⟩
| ⟨*number*⟩
| - ⟨*expr*⟩
| ( ⟨*expr*⟩ )
| ⟨*expr*⟩ op ⟨*expr*⟩

⟨*op*⟩ → +
| -
| *
| /

Regular expressions work well for defining tokens, however they are unable to specify nested constructs, which are central to programming languages. On the contrary, the ability of CFGs

to define a construct in terms of itself is crucial. A rule is **recursive** if its LHS appears in its RHS; thus `<expr>` is a recursive rule.

**3.4.2.2 Describing Lists** Variable-length lists in mathematics are often written using an ellipsis (...); $1, 2, \ldots$ is such an example. They can be defined recursively and thus a CFG can describe them in the following manner:

$\langle id\_list \rangle \rightarrow$ id
| id, $\langle id\_list \rangle$

This defines `<id_list>` as either a single token (`id`) or an identifier followed by a comma and another instance `<id_list>`.

### 3.4.3 Derivations

A context-free grammar (CFG) shows us how to generate a syntactically valid string of terminals: > Begin with the start symbol. Choose a production with the start symbol on the > left-hand side; replace the start symbol with the right-hand side of that > production. Now choose a nonterminal A in the resulting string, choose a pro- > duction P with A on its left-hand side, and replace A with the right-hand side > of P. Repeat this process until no nonterminals remain.

As an example, we can use the grammar for arithmetic expression to generate the string $slope * x + intercept$:

$\langle expr \rangle \Rightarrow \langle expr \rangle \langle op \rangle \langle expr \rangle$
$\Rightarrow \langle id \rangle \langle op \rangle \langle expr \rangle$
$\Rightarrow$ slope $\langle op \rangle \langle expr \rangle$
$\Rightarrow$ slope * $\langle expr \rangle$
$\Rightarrow$ slope * $\langle expr \rangle$ + $\langle expr \rangle$
$\Rightarrow$ slope * $\langle id \rangle$ + $\langle expr \rangle$
$\Rightarrow$ slope * x + $\langle expr \rangle$
$\Rightarrow$ slope * x + $\langle id \rangle$
$\Rightarrow$ slope * x + intercept

The $\Rightarrow$ metasymbol is often pronounced "derives." It indicates that the right-hand side was obtained by using a production to replace some nonterminal in the left-hand side.

- A **derivation** is a series of replacement operations that shows how to derive a string of terminals from the start symbol.
- Each string of symbols along the way is called a **sentential form**.
- The **final sentential form**, consisting of only terminals, is called the **yield** of the derivation.
- We sometimes elide the intermediate steps and write $expr \Rightarrow^* slope * x + intercept$, where the metasymbol $\Rightarrow^*$ means "derives after zero or more replacements."

Different replacement strategies leads to different derivations:

- **Leftmost derivation.** In this type of derivation, at each step of the derivation the left-most nonterminal with the left-hand side of some production. This is the strategy used to generate the string $slope * x + intercept$ in the example above.
- **Rightmost derivation.** In this type of derivation, at each step of the derivation the right-most nonterminal with the right-hand side of some production.
- **Others.** There are many other possible derivations, which includes options between a leftmost and rightmost derivation.

The rightmost derivation to generate the string $slope * x + intercept$ is as follows:

$\langle expr \rangle \Rightarrow \langle expr \rangle \langle op \rangle \langle expr \rangle$
$\Rightarrow \langle expr \rangle \langle op \rangle \langle id \rangle$

$\Rightarrow \langle expr \rangle \ \langle op \rangle$ intercept
$\Rightarrow \langle expr \rangle$ + intercept
$\Rightarrow \langle expr \rangle$ * $\langle expr \rangle$ + intercept
$\Rightarrow \langle expr \rangle$ * $\langle id \rangle$ + intercept
$\Rightarrow \langle expr \rangle$ * x + intercept
$\Rightarrow \langle id \rangle$ * x + intercept
$\Rightarrow$ slope * x + intercept

There are two things to notice here:

- Different derivations result in quite different sentential forms, but
- For a context-free grammar, it really doesn't make much difference in what order we expand the variables.

### 3.4.4 Parse Trees and Ambiguity

A **parse tree** is a hierarchical syntactic entity which represents the structure of the derivation of a terminal string from some non-terminal (not necessarily the start symbol). The root of the parse tree is the start symbol of the grammar. The leaves of the tree are its yield. Each internal node, together with its children, represents the use of a production.

A parse tree for the arithmetic expression grammar is shown down below:

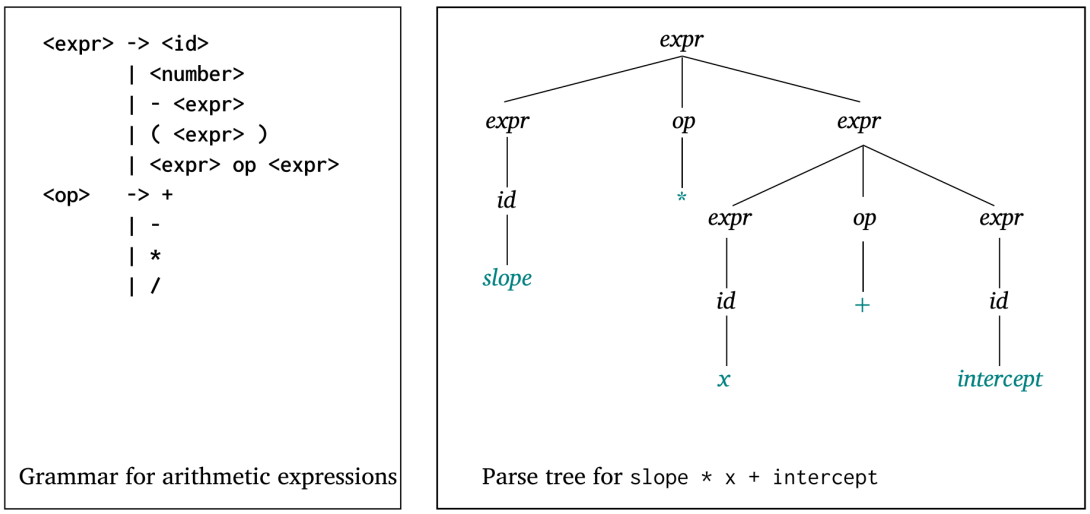



Figure 6: Parse tree for $slope * x + intercept$ using ambiguous grammar.

This tree is not unique. At the second level of the tree, we could have chosen to turn the operator into a + instead of a $*$, and to further expand the expression on the left, rather than the one on the right as shown in the figure below:

Ambiguity turns out to be a problem when trying to build a parser: it requires some extra mechanism to drive a choice between equally acceptable alternatives.

A grammar that allows the construction of more than one parse tree for some string of terminals is said to be an **ambiguous grammar**. Formally a CFG $G = (V, T, P, S)$ is **ambiguous** if there's at least one string/sentence $w$ in $T^*$ for which we can find two different parse trees, each with root labeled $S$ and yield $w$. If each string has at most one parse tree in the grammar, then the grammar is **unambiguous**.

Thus it's not a multiplicity of derivations that cause ambiguity, but rather the existence of two or more parse trees.

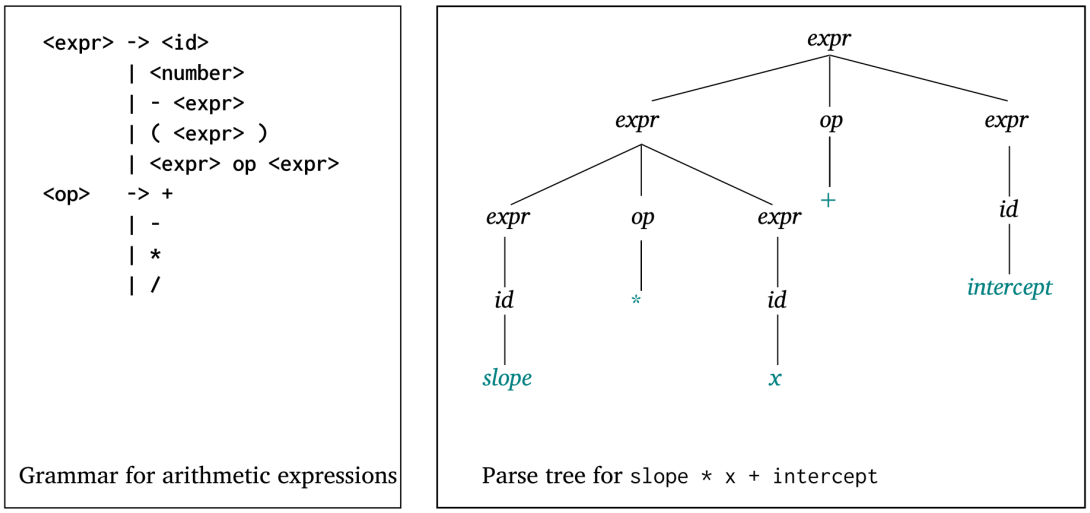



Figure 7: Parse tree for $slope * x + intercept$ using ambiguous grammar (alternative).

### 3.4.5 Removing Ambiguity from Grammars

In an ideal world, we could be able to give you an algorithm to remove ambiguity from CFGs. However the fact is that there's no algorithm whatsoever that can even tell us whether a CFG is ambiguous in the first place. Moreover there are context-free languages (CFLs) that have nothing but ambiguous CFGs; for these languages, removal of ambiguity is impossible.

For the sorts of constructs that appear in common programming languages, there are well-known techniques for eliminating ambiguity. Usually there are two causes in a programming language's ambiguity:

1. The precedence of operators is not respected; this is known as **operator precedence**.
2. A sequence of identical operators can group either from the left or from the right; this is known as **associativity of operators**. Since addition and multiplication are associative (i.e., $A + (B + C) = (A + B) + C$, $A * (B * C) = (A * B) * C$), it doesn't matter whether we group from the left or the right, but to eliminate ambiguity, we must pick one. The conventional approach is to insist on grouping from the left.

### 3.4.6 Operator Precedence

Precedence tells us that multiplication and division in most languages group more tightly than addition and subtraction, so that $3 + 4 * 5$ means $3 + (4 * 5)$ rather than $(3 + 4) * 5$. (Note that these rules are not universal). Thus, if $*$ has been assigned **higher precedence** than $+$ (by the language designer), multiplication will be done first, regardless of the order of appearance of the two operators in the expression.

The fact that an operator in an arithmetic expression is generated lower in the parse tree (and therefore must be evaluated first) can be used to indicate that it has precedence over an operator produced higher up in the tree.

In one of the parse tree for $slope * x + intercept$, the multiplication operator is generated lower in the tree, which could indicate that it has precedence over the addition operator in the expression. Its other parse tree, however, indicates just the opposite. It appears, therefore, that the two parse trees indicate conflicting precedence information.

In order to indicate the right operator precedence (and by "right" we mean the language designer's type of "right"), we must fix the grammar for arithmetic expressions. The correct ordering is specified by using separate nonterminal symbols to represent the operands of the

operators that have different precedence. This requires additional nonterminals and some new rules:

- Instead of using `<expr>` for both operands of both + (and -) and * (and /), we could use several new nonterminals to represent operands, which allows the grammar to force different operators to different levels in the parse tree.
- If `<expr>` is the root symbol for expressions, + can be forced to the top of the parse tree by having `<expr>` directly generate only + and/or - operators, using the new nonterminal, `<term>`, as the right operand of + (or -). Let's create the nonterminal `<add_op>` that yields the terminals + or -.
- Next, we can define `<term>` to generate * and/or / operators, using `<term>` as the left operand and a new nonterminal, `<factor>`, as its right operand. Now, * will always be lower in the parse tree, simply because it is farther from the start symbol than + and/or - in every derivation. Let's create the nonterminal `<mult_op>` that yields the terminals * or /.

The revised grammar for arithmetic expressions is as follows:

$$
\begin{array}{lll}
\langle expr \rangle & \rightarrow & \langle term \rangle \\
 & | & \langle expr \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
\langle term \rangle & \rightarrow & \langle factor \rangle \\
 & | & \langle term \rangle \ \langle mult\_op \rangle \ \langle factor \rangle \\
\langle factor \rangle & \rightarrow & \langle id \rangle \\
 & | & \langle number \rangle \\
 & | & \text{-} \ \langle factor \rangle \\
 & | & (\ \langle expr \rangle\ ) \\
\langle add\_op \rangle & \rightarrow & + \\
 & | & - \\
\langle mult\_op \rangle & \rightarrow & * \\
 & | & /
\end{array}
$$

This new grammar generates the same language as the previous one, but it is unambiguous and it specifies the usual precedence order of multiplication and addition operators. The leftmost and rightmost derivations for $slope * x + intercept$ are the following respectively:

Leftmost derivation of $slope * x + intercept$:

$$
\begin{array}{lll}
\langle expr \rangle & \Rightarrow & \langle expr \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \langle term \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \langle term \rangle \ \langle mult\_op \rangle \ \langle factor \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \langle factor \rangle \ \langle mult\_op \rangle \ \langle factor \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \langle id \rangle \ \langle mult\_op \rangle \ \langle factor \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \text{slope} \ \langle mult\_op \rangle \ \langle factor \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \text{slope} * \langle factor \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \text{slope} * \langle id \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \text{slope} * \text{x} \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \text{slope} * \text{x} + \langle term \rangle \\
 & \Rightarrow & \text{slope} * \text{x} + \langle factor \rangle \\
 & \Rightarrow & \text{slope} * \text{x} + \langle id \rangle \\
 & \Rightarrow & \text{slope} * \text{x} + \text{intercept}
\end{array}
$$

Rightmost derivation of $slope * x + intercept$:

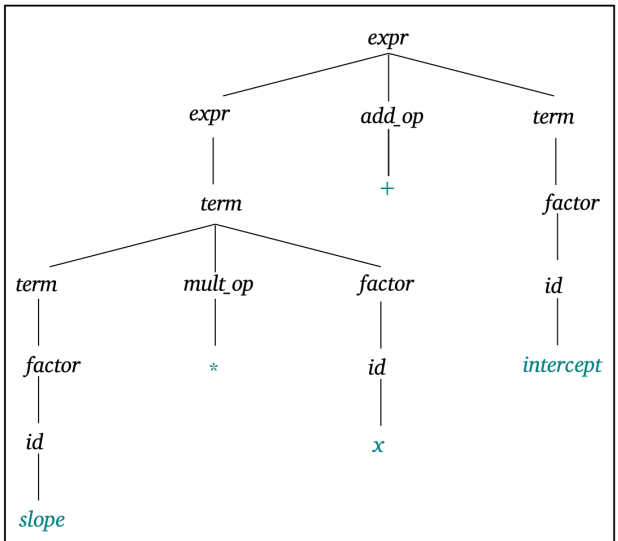
$$
\begin{array}{lll}
\langle expr \rangle & \Rightarrow & \langle expr \rangle \ \langle add\_op \rangle \ \langle term \rangle \\
 & \Rightarrow & \langle expr \rangle \ \langle add\_op \rangle \ \langle factor \rangle \\
 & \Rightarrow & \langle expr \rangle \ \langle add\_op \rangle \ \langle id \rangle \\
 & \Rightarrow & \langle expr \rangle \ \langle add\_op \rangle \ \text{intercept} \\
 & \Rightarrow & \langle expr \rangle + \text{intercept}
\end{array}
$$

$\Rightarrow \langle term \rangle$ + intercept
$\Rightarrow \langle term \rangle \langle mult\_op \rangle \langle factor \rangle$ + intercept
$\Rightarrow \langle term \rangle \langle mult\_op \rangle \langle id \rangle$ + intercept
$\Rightarrow \langle term \rangle \langle mult\_op \rangle$ x + intercept
$\Rightarrow \langle term \rangle$ * x + intercept
$\Rightarrow \langle factor \rangle$ * x + intercept
$\Rightarrow \langle id \rangle$ * x + intercept
$\Rightarrow$ slope * x + intercept

The unique parse tree for this string using the unambiguous grammar is shown down below:

```
<expr>     -> <term>
            | <expr> <add_op> <term>
<term>     -> <factor>
            | <term> <mult_op> <factor>
<factor>   -> <id>
            | <number>
            | - <factor>
            | ( <expr> )
<add_op>   -> + | -
<mult_op>  -> * | /
```

Unambiguous grammar for arithmetic expressions which specifies the usual precedence order of multiplication and addition operators.




Parse tree for `slope * x + intercept`

Figure 8: Parse tree for $slope * x + intercept$ using unambiguous grammar.

The connection between parse trees and derivations is very close: Either can easily be constructed from the other. Every derivation with an unambiguous grammar has a unique parse tree, although that tree can be represented by different derivations.

### 3.4.7 Associativity of Operators

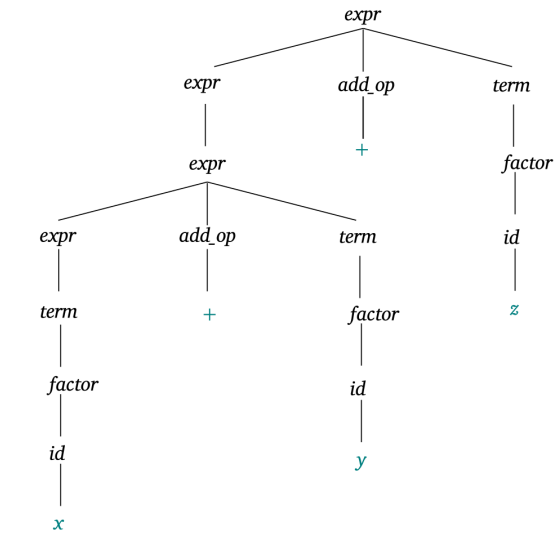
Associativity tells us that the operators in most languages group left to right, so that $12/2 * 3$ means $(12/2) * 3$ rather than $12/(2 * 3)$. When an expression includes two operators that have the same precedence (as $*$ and $/$ usually have), **associativity** is the semantic rule that specifies which should have precedence. An expression with two occurrences of the same operator has the same issue; for example $12/2/3$, i.e., $(12/2)/3$ vs. $12/(2/3)$.

Consider the parse tree for the arithmetic expression $x + y + z$:

The parse tree shows the left addition operator lower than the right addition operator. This is the correct order if addition is meant to be *left associative*, which is typical. In most cases, the associativity of addition in a computer is irrelevant. In mathematics, addition is associative, which means that left and right associative orders of evaluation mean the same thing, i.e., $(x + y) + z = x + (y + z)$. Floating-point addition in a computer, however, is not necessarily associative. Subtraction and division are not associative, whether in mathematics or in a computer. Therefore, correct associativity may be essential for an expression that contains either of them.

```
<expr>     -> <term>
            | <expr> <add_op> <term>
<term>     -> <factor>
            | <term> <mult_op> <factor>
<factor>   -> <id>
            | <number>
            | - <factor>
            | ( <expr> )
<add_op>   -> + | -
<mult_op>  -> * | /
```

Unambiguous grammar for arithmetic expressions which specifies the usual precedence order of multiplication and addition operators.




Parse tree for `x + y + z`

Figure 9: Parse tree for $x + y + z$.

- **Left associativity.** When a grammar rule has its LHS also appearing at the beginning of its RHS, the rule is said to be left recursive; this left recursion specifies left associativity.

  For the updated grammar for arithmetic expressions, the left recursion of the rules of the grammar causes it to make both addition and multiplication left associative:

  $\langle expr \rangle \rightarrow \langle expr \rangle \langle add\_op \rangle \langle term \rangle \mid \ldots \langle term \rangle \rightarrow \langle term \rangle \langle mult\_op \rangle \langle factor \rangle \ldots$

  Unfortunately, left recursion disallows the use of some important syntax analysis algorithms. When one of these algorithms is to be used, the grammar must be modified to remove the left recursion. However, disallowing the grammar from precisely specifying that certain operators are left associative. Fortunately, left associativity can be enforced by the compiler, even though the grammar does not dictate it.

- **Right associativity.** When a grammar rule has its LHS also appearing at the right end of its RHS, the rule is said to be right recursive; this right recursion specifies right associativity.

  In most languages that provide it, the exponentiation operator is right associative, e.g., $2 * *3 * *2$ means $2 * *(3 * *2)$ rather than $(2 * *3) * *2$. A rule such as the following could be used to describe exponentiation:

  $\langle factor \rangle \rightarrow \langle expr \rangle$ ** $\langle factor \rangle \mid \langle exp \rangle$
  $\langle expr \rangle \rightarrow$ ( $\langle expr \rangle$ ) $\mid \langle id \rangle$

### 3.4.8 Extended BNF (EBNF)

Due to a few minor inconveniences in BNF, it has been extended in several ways. Most extended versions are called Extended BNF, or simply EBNF, even though they are not all exactly the same. The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability. Three common extensions included in the various versions of EBNF are:

- First extension denotes an optional part of an RHS, which is delimited by brackets.

  For example, a C if-else statement can be described as

⟨*if_stmt*⟩ → if ( ⟨*expr*⟩ ) ⟨*stmt*⟩ [else ⟨*stmt*⟩]

which without the use of the brackets, the syntactic description of this statement would require the following two rules:

⟨*if_stmt*⟩ → if ( ⟨*expr*⟩ ) ⟨*stmt*⟩
| if ( ⟨*expr*⟩ ) ⟨*stmt*⟩ else ⟨*stmt*⟩

- Second extension uses braces ({}) in a RHS to indicate that the enclosed part can be repeated indefinitely or left out altogether. In other words, the enclosed part can be repeated part can have zero or more occurrences.

  For example, a list of identifiers separated by comma:

  ⟨*id_list*⟩ → ⟨*id*⟩ {, ⟨*id*⟩}

  Its BNF equivalent is:

  ⟨*id_list*⟩ → ⟨*id*⟩ | ⟨*id*⟩, ⟨*id_list*⟩

- Third extension deals with multiple-choice options. When a single element must be chosen from a group, the options are placed in parentheses and separated by the OR operator, |. For example,

  ⟨*term*⟩ → ⟨*term*⟩ (* | / | %) ⟨*factor*⟩

  requires the following rules in BNF:

  ⟨*term*⟩ → ⟨*term*⟩ * ⟨*factor*⟩
  | ⟨*term*⟩ / ⟨*factor*⟩
  | ⟨*term*⟩ % ⟨*factor*⟩

The brackets, braces, and parentheses in the EBNF extensions are **metasymbols**, which means they are notational tools and not terminal symbols in the syntactic entities they help describe.

# 4 Attribute Grammars

Context-free grammars (CFGs) allow us to specify the regular expressions that we use to create words and other strings for our input string of characters and they allow us to parse our programs, however they aren't always useful in specifying the meaning of a statement or an expression.

This is why **attribute grammars** are important; they give us a way to do this in a notation that complements BNF and they allow us to specify the semantic requirements of the language that we can use together with BNF.

An **attribute grammar** is an extension to a context-free grammar that's used to describe features of a programming language that cannot be described in BNF or can only be described in BNF with great difficulty. Examples:

- Describing the rules that float variables can be assigned integer values but the reverse is not true is difficult to describe completely in BNF.
- The rule requiring that all variables must be declared before being used is impossible to describe in BNF.

## 4.1 Static vs Dynamic Semantics

The semantics of language can be **static** or **dynamic**. Which we have depends on whether the meaning of a program's components changes at runtime.

- The **static semantics** of a language are indirectly related to the meaning of programs during execution. They're set when the program is compiled and will last through the runtime of a program.

  Its name comes from the fact that these specifications can checked at **compile-time**.
- The **dynamic semantics** refer to the meaning of various components of a program including expressions, statements, etc. Unlike static semantics, these cannot be checked at compile-time and can only be checked at **runtime**.

  In some languages such as Lisp and Schemes it's impossible to check the meaning of certain program's components at compile time because they're either set at runtime or can be changed then.

## 4.2 What's an Attribute?

- An **attribute** is a property whose value is assigned to a grammar's symbol. It can be the data type of a variable, its value, or its address in memory.
- Attribute grammars make use of two types of function:
  - **Semantic functions** (or attribute computation functions) are associated with the productions of a grammar and are used to compute the values of an attribute.
  - **Predicate functions** state some of the syntax and static semantic rules of the grammar.

## 4.3 Definition of an Attribute Grammar

An attribute grammar is a grammar with the following added features:

- Each symbol $X$ in the grammar has a set of attributes $A(X)$.
- The set $A(X)$ has two **disjoint** sets:
  - $S(X)$ is a set of **synthetized attributes**, which are used to pass semantic information up a parse tree from child to parent.

  - $I(X)$ is a set of **inherited attributes**, that also pass semantic information but does it down and across a parse tree from parent to child.

- Each production of the grammar has a set of semantic functions and a set of predicated functions (which may be an empty set).
- Intrinsic attributes are synthetized attributes whose properties are found outside the grammar (e.g., a symbol table which stores variable names and their types).

## 4.4 Examples

### 4.4.1 An Attribute Grammar for Assignments

This example illustrates how an attribute grammar can be used to check the type rules of a simple assignment statement. The syntax and static semantics of this assignment statement are as follows:

- The only variable names are $A$, $B$, and $C$.
- The right side of the assignments can be either a variable or an expression in the form of a variable added to another variable.
- The variables can be one of two types: `int` or `real`.
- When there are two variables on the right side of an assignment, they need not be the same type.
  - The type of the expression when the operand types are not the same is always `real`.
  - When they are the same, the expression type is that of the operands.
- The type of the left side of the assignment must match the type of the right side. So the types of operands in the right side can be mixed, but the assignment is valid only if the target and the value resulting from evaluating the right side have the same type.
- The attribute grammar specifies these static semantic rules.

The syntax portion of our example attribute grammar is

⟨*assign*⟩ → ⟨*var*⟩ = ⟨*expr*⟩
⟨*expr*⟩ → ⟨*var*⟩ + ⟨*var*⟩ | ⟨*var*⟩
⟨*var*⟩ → A | B | C

The attributes for the nonterminals in the example attribute grammar are described in the following paragraphs:

- `actual_type`. A synthetized attribute associated with the nonterminals `<var>` and `<expr>`. It's used to stored the *actual type* (`int` or `real`) of a variable or expression.
  - In the case of a variable, the actual type is intrinsic ().
  - In the case of an expression, it's determined from the actual types of the child node or children nodes of the `<expr>` nonterminal.
- `expected_type`. An inherited attribute associated with the nonterminal `<expr>`. It's used to store the type (either `int` or `real`) that's expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

The attribute gramma is as follows:

1. Syntax rule: `<assign> -> <var> = <expr>` Semantic rule: `<expr>.expected_value <- <var>.actual_type`
2. Syntax rule: `<expr> -> <var>[2] + <var>[3]` ([2] and [3] differentiate the three `<var>` nonterminals)

   Semantic rule:

   ```
   <expr>.actual_type <-
         if <var>[2].actual_type == int) and <var>[3].actual_type == int) then
             int
   ```

```
else
    real
```

**Predicate:** `<expr>.actual_type == <expr>.expected_type`

3. Syntax rule: `<expr> -> <var>`

   Semantic rule: `<expr>.actual_type <- <var>.actual_type`

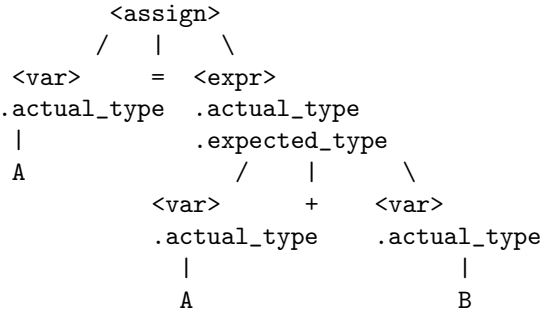
   **Predicate:** `<expr>.actual_type == <expr>.expected_type`

4. Syntax rule: `<var> -> A | B | C`

   Semantic rule: `<var>.actual_type <- lookup(<var>.string)`

The `look-up` function looks up a given variable name in the symbol table and returns the variable's type.

**The parse tree for** `A = A + B`




**How are attribute values computed?**

For `A = B + C`. Initially, there are only intrinsic attributes on the leaves.

1. `<var>.actual_type <- look-up(A)` (Rule 4)
2. `<expr>.expected_type <- <var>.actual_type` (Rule 1)
3. `<var>[2].actual_type <- lookup (A)` (Rule 4) `<var>[3].actual_type <- lookup (B)` (Rule 4)
4. `<var>[1].actual_type <- either int or real` (Rule 2)
5. `<expr>.expected_type == <expr>.actual_type` is either `TRUE` or `FALSE`

The parse tree down below shows the flow of attribute values. Solid lines show the parse tree; dashed lines show attribute flow in the tree.

The following parse tree shows the final attribute values on the nodes. In this example, A is defined as a `real` and B is defined as an `int`.

Determining attribute evaluation order for the general case of an attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies.

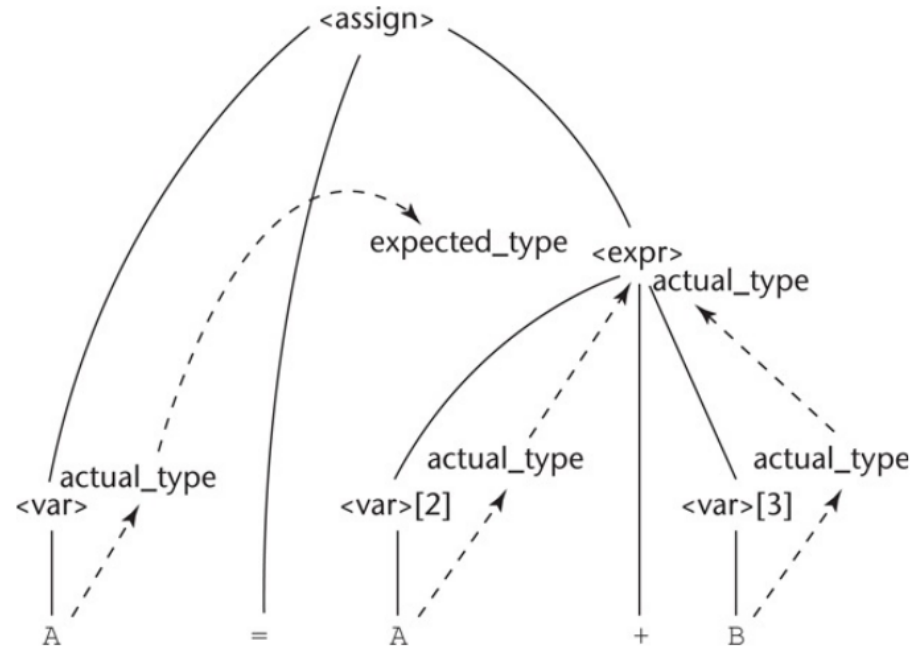



Figure 10: Flow of attributes in parse tree of $A = A + B$.

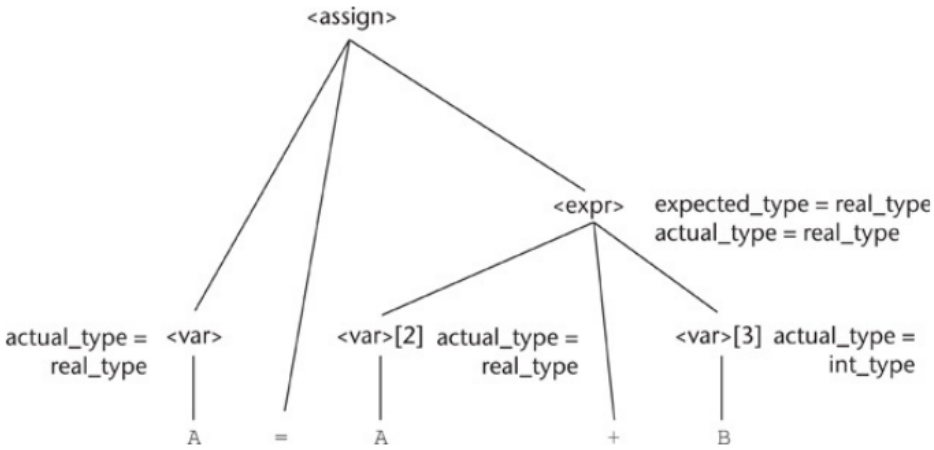



Figure 11: Flow of attributes in fully attributed parse tree of $A = A + B$.